Learning Data Science in R

Antonio Fidalgo

Last update: April 01, 2019

Contents

Foreword

> "Data science is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it…"
>
> — paraphrase of Dan Ariely's quote on big data

There is already a large number of excellent and free references for learning data science in R. The list would be too vast, but two names stand out: Hadley Wickham and Yihui Xie.
Work of the first includes the books Wickham and Grolemund (2016) and Wickham (2014) (freely available, including source code, on github.com/hadley) as well as some of the most popular packages in R such as `ggplot2` (Wickham et al., 2018) and `tidyverse`(Wickham, 2017).
Work of the second is used precisely in the current file thanks to the `rmarkdown` package (Allaire et al., 2018), `knitr` (Xie, 2018b) and `bookdown` (Xie, 2018a). His source code is also on github.com/yihui/.

So, the question arises of why one should even bother to write on the topic. The present "book" is justified on the following grounds.

1. There is no better way of learning data science and R than **doing oneself** the pages of code.

2. The material gathered here is a **personal selection** made on what I judge most relevant for my work, the most important techniques in general or, sometimes, the least obvious for the regular practitioner.

3. These notes serve as an **archive** of what I did in the domain so that I can easily access it in the future.

Chapter 1

Getting started

1.1 Install applications

Download the following free applications, available on all platforms:

1. R
2. RStudio, free Desktop version
3. A Latex distribution (e.g., MacTex for Mac or MiKTeX for Windows machines)

The first two are easily and quickly installed. The last is a very large program (a few Gb) and needs time to install.

Another application needed is a

4. Git distribution

This is also a free software.
Once you have installed Git for version control, activate it in RStudio: *Tools> Global Options> Git/SVN* and click on *Enable version control interface for RStudio projects*.
Also generate a SHH RSA key. We will use it to identify at the GitHub repo.

1.2 Sign for GitHub

Create an account at GitHub at https://github.com.

1.3 Create your project

File> New project> Existing Directory and chose that book folder.
Now, every time you create content for your book, you must start a Rstudio session *File> Open Project...*
All the files of the project are the files of the folder, and vice versa.

1.4 Create GitHub repo and link your machine to it

Create a new repository (pronounced 'repo') whose name is **exactly** the same as your R project / book folder (e.g., 'myRbook').

On the top left menu in GitHub, go to *Settings> SHH and GPG keys* and click on 'New SHH key'. Paste the SHH key generated by RStudio.

In RStudio go to *Tools> Project Options...> Git/SVN*. Under *Version control system*, select 'Git'.
Still in RStudio, *Tools> Terminal> New Terminal*. This open a Terminal where you type commands.

Start by initializing Git on the terminal.

```
git init
```

Then paste the message shown at the creation of the repo (changing the names, of course):

```
git remote add origin https://github.com/YOURNAME/YOURREPO.git
git push -u origin master
```

Your local `master` should now be connected to the `master` on GitHub.
If necessary, restart RStudio. At the restart, a Git thumbnail should appear in a pane. You are ready to commit and push your files.

1.4.1 Troubleshooting

If the procedure above works, great! One must be aware, however, that it sometimes hits some problem unexpected and difficult to understand problem. In these cases, a classic solution is to copy and google the error message with a first term such as "RStudio".
In my experience, the following hack worked. Find the invisible file of the repo *.git> config*. Hidden files on Mac are turned visible by the keyboard key `CMD + SHIFT + .`. In that file, delete the following part (including the space it takes)

```
[remote "origin"]
    url = git@github.com:USERNAME/repo.git
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Then, try again the procedure above.

1.5 Collaboration on GitHub

There are various established ways for collaborating on a GitHub repo. As a matter of facts, collaboration on a project is the *raison d'etre* of GitHub.
We illustrate here the easiest of them, namely the joint reading and writing into one repo by all the members of a team.
Importantly, it is assumed that all members of the group are able to push/pull code from RStudio to an experimental repo on GitHub (see above).
The following are the steps in the collaboration's setup.

1.5.1 Owner of the organization

One member of the group creates an organization on GitHub under *Settings> Organizations> New organization*.
Then, under that organization, this member of the team creates a new repo whose name is the same as the folder and R project that the group will work on (e.g., 'ouRbook').
The owner of the organization connects from RStudio to the repo and populates it with the current files of the project.
Still as a task of the owner, create a team. Under the organization page in GitHub, simply click on 'New team' and give it a name.
A team now appears in the organization page. In the page of that team, click on 'Add a member'. A window

appears where the member to be added can be searched for and added.
Once that member is found and selected, make the double step of adding him/her to the team AND assign him/her to the repo. This second step can still be made later, but it's simpler to do it right then.
Importantly, make sure that the new member has reading and **writing** rights into the repo. This again, can be changed later by navigating the team's page.

1.5.2 Member of the team

After the steps above, the newly added member receives an email to confirm the participation the organization / team.
Navigate now to the team's page and locate the repo that the owner has created.
A button-menu allows to 'Clone or download the repo'. Click to show the https address of the repo, `https://github.com/ORGANIZATION/repo.git` and copy that link.
Open a simple RStudio session (not a project!). Open a new Terminal (*Tools> Terminal> New Terminal*) and type `cd` (change directory) and a path to the folder where you want the repo to be saved. For instance,

```
cd Dropbox/Fresenius/DS4B
```

The terminal is writing into that folder, as indicated by the path before the `$` sign in command line of the terminal.
Recall that you copied the address of the team's repo. Then, in command line, now type the following commands and your copied address:

```
git clone https://github.com/ORGANIZATION/repo.git
```

The whole repo is now a new folder in your directory. If all went well, you can now pull and push into that repo in GitHub.

Part I

Source and output files

Chapter 2

`.Rmd` source files

This chapter gathers general comments about `.Rmd` files.

2.1 Options for all chunks

It is convenient to set options for all the R chunks of the document. This saves time when writing these chunks.
A natural place to set these options is in a first R chunk.

```
knitr::opts_chunk$set(OPTION1 = TRUE/FALSE,
                      OPTION2 = TRUE/FALSE,
                      ...)
```

Importantly, these options are overridden by the particular chunk options.

```
```{r, OPTION2=FALSE}
```

Options actually take R code. So, the following are examples that could be used to define the option.

```
```{r, eval=4>3, echo=format(Sys.Date(), '%Y-%B-%d') > '2019-March-10'}
# eval is always TRUE
# echo = TRUE if current date is after March 10, 2019
```
```

The list of options can be found here https://yihui.name/knitr/options/. Below are some comments on some of these options (the least trivial for the author).

- `collapse` determines whether the source code and the output should be merged into a single block. Here is the same chunk with different values of the option:

`collapse=TRUE`

```
2+ 2
#R> [1] 4
3* 5
#R> [1] 15
```

`collapse=FALSE`

```
2+ 2
```

```
#R> [1] 4
```

```
3* 5
```
```

```
#R> [1] 15
```

- `comment` gives the string to be printed before the output.

```
comment='##'
```

```
2+ 2
## [1] 4
```

```
comment='R>'
```

```
2+ 2
R> [1] 4
```

Worth noting: a `#` as a first character of the comment string (with `collpase=TRUE`) turns the output font into a comment-like text.

- `child` allows a document to call and use another file as input in the document.

```
```{r, child='PATH/TO/OTHER/file.Rmd'}
```

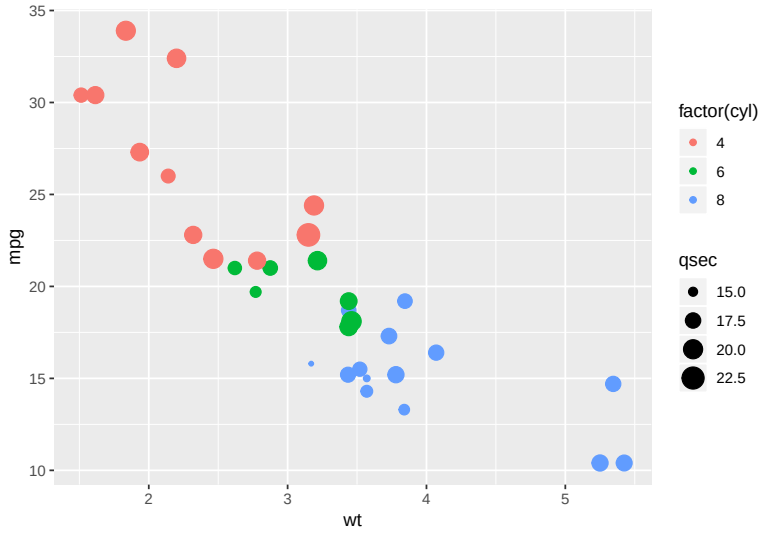
The path can be either absolute or relative.
For relative paths, the following applies:

- `~/` starts a path a the root,
- `../` indicates the parent directory,
- `../../` for parent of the parent directory,
- to move forward, start with the name of the included folder in the current directory.

2.2 Latex code

The overwhelming reason to introduce Latex code in a `.Rmd` file is for typesetting mathematical expressions. There are two main ways to type math in Latex:

- in the text, surrounded by special delimiters, `\( math \)` (alternatively, one can use the deprecated `$ math $`),
- in an equation, surrounded by special delimiters, `\[ math equation \]`, or in a dedicated environment such as `\begin{equation} math equation \end{equation}` (also deprecated, `$$ math equation $$`).

Math format is usually distinct from the text format. For instance, compare $a^2 + b^2 = c^2$ (simple markdown) to $a^2 + b^2 = c^2$ (Latex). Notice, however, that most mathematical symbols are not supported by markdown. This why basic knowledge of Latex is essential for producing documents with math formula.
The first part of that knowledge consists in the list of Latex symbols. These are typed into a document starting with a slash `\`. For instance, the Latex symbol α is produced by the code `\alpha` while $\sum$ is given by `\sum`.
Superscripts and subscripts are produced with the characters `^` and `_`, respectively, and the content of the super- and subscripts enclosed within `{ }`. For instance, x_2^3 is produced by the code `x_{2}^{3}` (or `x^{3}_{2}`) and $\sum_{i=1}^{4}$ is given by `\sum_{i=1}^{4}`.
A comprehensive list of symbols (some of them requiring special Latex packages) can be found in Pakin (2009). Shorter lists can be found online, including on wiki.
A web-based application helps beginners by offering suggestions to a hand-drawn symbol, detexify.
A second bulk in the knowledge of Latex is the understanding of special environments. But then, if a document requires lots of special Latex environments, then `.Rmd` files are not the most appropriate type of files to use.
Some of these environments, however, have a perfect integration in `.Rmd` files, such as the most common, the equation. Here is an example, given by the code
```



Figure 2.1: Including a plot.

```
\[
\sum_{i=0}^{\infty} ak^{i} = \frac{a}{1-k} \text{, for }  \lvert k \rvert < 1
\]
```

$$\sum_{i=0}^{\infty} ak^{i} = \frac{a}{1-k}, \text{ for } |k| < 1$$

2.3 Figures

Figures usually appear after the chunk that calls them, except on `.pdf` format because Latex makes them float and they may appear in the next page.
Beyond the actual R calls that produces the figure, the figure's appearance will be tweaked by the options specified in the R chunk. The following options are worth noticing:

- `fig.width` and `fig.height` set in inches the graphical device size of the R plots.
- `out.width` and `out.height` set the size of the R plots, including possibly rescaled images. Easier use with percentages as `out-width = "50%"` means that the figure stretches over 50% of the page's width.
- `fig.align` takes the value `left`, `right` or `center` and is self-explained.
- `fig.cap` sets the caption of the figure.
- `fig.show` takes the default value `asis` to produce the command as it is read by R. Contrast this with the value `hold` which forces R to wait until the end of the chunk to produce the plots. It may be useful to hold if one wants to put multiple plots on the same line (side-by-side).

As an example, the figure below was created with the chunk options

```
```{r plot-one, echo=FALSE, out.width="60%",
fig.align="center", fig.cap="Including a plot."}
```

For two plots next to each other, the chunk options could be

```
```{r plot-two, echo=FALSE, out.width="30%", fig.align="center",
fig.cap="Including two plots.", fig.show="hold"}
```

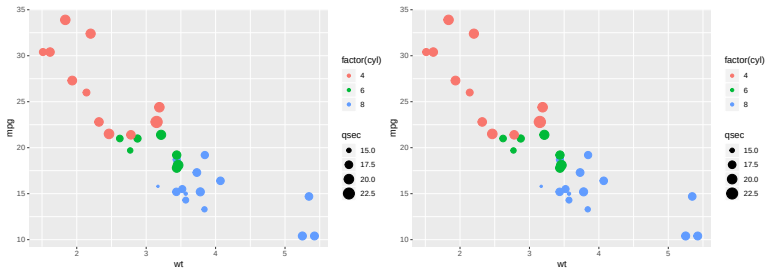


Figure 2.2: Including two plots.

2.4 Cross-references

Cross-references are references to different elements somewhere in the content such as a specific chapter, figure, equation, etc.
These work under a simple rule: the element to be referenced takes a **label**, e.g., 'foo', and this label is called from anywhere in the text with the command `\@ref(foo)`.
Slightly different rules apply depending on which element is to be referenced. Below is a selected list.
Notice that all labels in `bookdown` can only contain alphanumeric characters, or the special characters `:`, `-`, and `/`.

2.4.1 Chapters, sections, ...

Headers (chapters, sections, ...) have an ID that serves as a label. By default, it is formed by the words of the header in small letters separated by dashes (e.g., `my-great-chapter`). It is better practice, however, to assign a label to useful headers by appending to them the code `{#foo}` where `foo` is the chosen label for the header. This subsection, for instance, has the header

```
### Chapters, sections, ... {#cr-chapters}
```

The cross-reference is then achieved with `\@ref(foo)`.
As an example, notice that this line belongs to Subsection 2.4.1, included in Section 2.4 of Chapter 2.
These cross-references were obtained by the text/code:

```
As an example, notice that this line belongs to Subsection \@ref(cr-chapters),
included in Section \@ref(cr) of Chapter \@ref(rmd).
```

For `.html` formats, one may use a direct link to the header by using the format for regular links but with `#label` as the destination. For instance, since the label of this section is `cr`, the code `[Cross-references](#cr)` produces this link: Cross-references.

2.4.2 Figures

The label of a figure is given by the name of the chunk where it was created preceded by `fig:`.
The chunk that created Figure 2.1 had the name `plot-one`. Therefore, the cross-reference is achieved through `\@ref(fig:plot-one)`.

2.4.3 Tables

The label of a table is given by the name of the chunk where it was created preceded by `tab:`.
If the chunk that created a Table had the name `table-one`, then cross-reference is achieved through `\@ref(tab:table-one)`.

2.4.4 Equations

As explained in Section 2.2, good rendering of equations requires coding in a Latex environment. For numbering of equations, various environments can be used, such as `equation`, `eqnarray`, etc.
The label is then included in this environment within parentheses and by adding the prefix `\#eq:`, e.g., `\#eq:foo`.
As for the cross-reference, one uses `\@ref(eq:foo)`.
Below is a modified version of the equation in Section 2.2 to include a number and a label.
This is Equation (2.1).

$$\sum_{i=0}^{\infty} ak^i = \frac{a}{1-k}, \text{ for } |k| < 1 \tag{2.1}$$

The previous lines were obtained with the code:

```
This is Equation \@ref(eq:series).

\begin{equation}
\sum_{i=0}^{\infty} ak^{i} = \frac{a}{1-k} \text{, for }  \lvert k \rvert < 1
  (\#eq:series)
\end{equation}
```

2.5 Citations

It is of utmost importance to give credit to sources. This subsection illustrates a simple way to make citations in `bookdown`. For further details, including for changing references styles, see this RStudio page.
The general idea is similar to the cross-references. The source has a label `foo` and the citation is made with `@foo`.
There are numerous ways to tweak the format of the citation, but default format is often good enough.
The citation procedure in `bookdown` is borrowed from the Latex constellation.
Sources are listed in a separated bibliography file, `.bib`.
For use in `bookdown`, this file must be given in the `YAML` (along with its path if it is not in the same folder) under the entry `bibliography`:

```
---
title: "Great title"
output: pdf_document
bibliography: BIBLIOGRAPHY_FILE.bib
---
```

The `BIBLIOGRAPHY_FILE.bib` is a simple text file that has one entry for each source.
The format of this entry depends on the type of source (article, book, blogpost, etc). Its construction, however, is quite similar and simply requires some fields (some compulsory, other optional) such as in the following example for a book.

```
@book{wickham2014,
  title={Advanced {R}},
  author={Wickham, Hadley},
  year={2014},
  publisher={Chapman and Hall/CRC}
}
```

All the fields shown (`title`, `author`, `year` and `publisher`) are required for a book. Further fields such as `volume` or `edition` would be optional.

Importantly, notice the very first element in the curly brackets: that is the label of the source, freely chosen by the author.
The citation of this particular book would then be made thanks to the label, with the code `@wickham2014`.
A few comments are the following:

- a reference inside square brackets with or without a prefix and a suffix puts the citation inside parentheses: `[see the reference (prefix) @wickham2014 for further details (suffix)]` produces (see the reference (prefix) Wickham, 2014, for further details (suffix)),
- without square brackets, no parentheses in the output,
- multiple references are separated by a semi-column `;`: `[@wickham2014; @xie2015]`produces (Wickham, 2014; Xie, 2015),
- a minus sign `-` in front of the `@` suppresses the name of the author in the text: `-@wickham2014` produces (2014),
- the references are **automatically** put at the end of the book; the last level-one header should therefore be 'Bibliography' or 'References'.

2.5.1 Writing `.bib` entries

One advantage of using `.bib` files is that the fields for each entry, i.e., all the information about the source, are usually available online in the easiest of forms.
The site Google Scholar, for instance, provides these entries in `.bib` form.
As an example, search there the book mentioned above, e.g., "wickham advanced r". Under each hit, there is an icon of inverted commas. By clicking it, the reference is given in different formats but also with a link to the BibTeX code (along with others). One can then simply copy this BibTex code and paste it anywhere in the bibliography file. For easier use, the label could be changed.

Chapter 3

Customize output

This chapter is about choosing and/or modifying the way the output file looks like.

3.1 Multiple built-in output types

[complete this section] From pdf to slides, through webpages and notebooks.

3.2 New types provided by packages

[complete this section] These are more variations of the above.

3.3 CSS: custom html

A file to change how html outputs look like.

3.4 Latex preamble

This file is added to the preamble of the Latex file to modify how the `.pdf` output is compiled.
Literally, these commands are placed in the Latex file before the self-explaining command:

```
\begin{document}
```

The most common commands call the packages to be used in the compilation of the Latex file, e.g.,

```
\usepackage{multirow}
```

Each package, in turn, includes a set of functions that can be used in the file to produce some particular output.
The other set of commands are typically definitions of new commands or environments as well as redefinitions of existing commands to produced a special output.
These latter require an excellent command of the language. The good news is that people are keen to share their expertise. Here is an example that I included in another `preamble.tex` for counting the sections in reverse order. I found it on a blog, texblog.

```
\usepackage{totcount}
\regtotcounter{section}
\makeatletter
    \renewcommand{\thesection}{\number\numexpr\c@section@totc-\c@section+1\relax}
\makeatother
```

Part II

R Basics

Chapter 4

R as a calculator

R can be used as a simple calculator. For instance, 45+17=62 .
Here are a few more examples of the commands.

4.1 Usual operators

4.1.1 Simple operations

The usual symbols `+, -, *, /` apply .

```
2 + 6
#R> [1] 8
56/ 6
#R> [1] 9.333333
```

4.1.2 Parentheses

The parentheses work as expected. But they are also a common source of error when they are not matched.

```
(4+3)*((7-3)/(1+.05))
#R> [1] 26.66667
```

The next expression will generate an error and prevent the compilation of the whole book.

```
(4+3)*((7-3/(1+.05))
```

4.1.3 Exponents

There are two ways of expressing the power of a number: `^` and `**`.

```
3^4
#R> [1] 81
3**4
#R> [1] 81
```

4.2 Unusual operators

4.2.1 Special operations

The symbols **%/%** and **%%** return the entire part of the result of the division and the rest of the division, respectively.

```
56/6
#R> [1] 9.333333
56%/%6
#R> [1] 9
56%%6
#R> [1] 2
```

4.3 Usual functions

The common functions found in any calculator also have an equivalent on R. The following examples need no further comment.

```
log(100)
#R> [1] 4.60517
sqrt(100)
#R> [1] 10
```

4.4 Unusual functions

There are several other functions applied to numbers that one does not usually find in a calculator.
A few examples below:

```
# floor / ceiling for the closest integer
floor(13.47)
#R> [1] 13
ceiling(13.47)
#R> [1] 14
```

Chapter 5

Data stuctures

This chapter lists the most common objects to store data.

5.1 Atomic vectors

The basic data structure in R is the vector. Vectors have three characteristics:

- a type, `typeof()`,
- a length, `length()`,
- some attributes, `attributes()`.

5.1.1 Types of data

There are four common types of atomic vectors:

- logical,
- integer,
- double (often called numeric),
- character.

Note that if vector elements not all of the same type, R makes coercion. In that case, the order becomes: "logical < numeric < character".

Here are a few illustrations.

```
# logical vector
log_vector <- c(TRUE, FALSE, FALSE)
log_vector
#R> [1]  TRUE FALSE FALSE

# numeric vector
num_vector <- c(12, 10, 3)
typeof(num_vector)
#R> [1] "double"

# integer vector
int_vector <- c(12L, 10L, 3L)
typeof(int_vector)
#R> [1] "integer"
```

```
# character vector
chr_vector <- c("a", "b", "c")
typeof(chr_vector)
#R> [1] "character"

# mixed types vector
mix_vector <- c(2, "a")
is.numeric(mix_vector)
#R> [1] FALSE
typeof(mix_vector)
#R> [1] "character"
```

Numeric vectors can be created with the shortcut `:`, which is used to imply all the integers from the number on the left of `:` to the number on its right. Even if it is not good practice, these do not require the `c()` call.

```
vector_A <- 1:5
vector_A
#R> [1] 1 2 3 4 5
length(vector_A)
#R> [1] 5
```

5.1.2 Factor vector

This is a special type of vector. It has a limited number of values, called levels. These levels can be unordered (e.g., gender is either "Female" or "Male") or ordered (e.g. school level is "Primary", "Secondary", "Tertiary")

```
gender <- factor(c("Male", "Male", "Female", "Male", "Female", "Female", "Male"))
gender
#R> [1] Male   Male   Female Male   Female Female Male
#R> Levels: Female Male
levels(gender)
#R> [1] "Female" "Male"
summary(gender)
#R> Female   Male
#R>      3      4
school <- factor(c("Primary", "Secondary", "Tertiary"),
                 ordered=TRUE)
school
#R> [1] Primary   Secondary Tertiary
#R> Levels: Primary < Secondary < Tertiary

school2 <- factor(c("Primary", "Secondary","Secondary", "Tertiary"),
                  labels=c( "Secondary", "Tertiary","Primary"),
                  ordered=TRUE)
school2
#R> [1] Secondary Tertiary  Tertiary  Primary
#R> Levels: Secondary < Tertiary < Primary
```

5.2 Matrices and arrays

R is not often used for matrices calculations: it is too slow for that and there are better programs for it out there (e.g. Matlab).

```
# populate a matrix with the elements in of the vector, and give the dimensions
M <- matrix(c(4, 1, 0, 3, 6, 8), nrow=3, ncol=2)
M
#R>      [,1] [,2]
#R> [1,]    4    3
#R> [2,]    1    6
#R> [3,]    0    8
```

If we think of a matrix as a 2 dimensions vector, then arrays are n dimensions vectors. Is it important? It might be in some very specific cases.

```
mya <- array(data=1:18, dim=c(2,3,3))
mya
#R> , , 1
#R>
#R>      [,1] [,2] [,3]
#R> [1,]    1    3    5
#R> [2,]    2    4    6
#R>
#R> , , 2
#R>
#R>      [,1] [,2] [,3]
#R> [1,]    7    9   11
#R> [2,]    8   10   12
#R>
#R> , , 3
#R>
#R>      [,1] [,2] [,3]
#R> [1,]   13   15   17
#R> [2,]   14   16   18
```

5.3 Lists

These are the one-size fit all structure… A list is an object composed of any other object, even… another list! Very useful data structure!

```
school<-factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE)
mylist <- list(numbers=c(1:60),
               somenames=c("Jim","Jules"),
               results= c(T,F,F,T),
               school=school)
mylist
#R> $numbers
#R>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
#R> [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
#R> [47] 47 48 49 50 51 52 53 54 55 56 57 58 59 60
#R>
#R> $somenames
#R> [1] "Jim"   "Jules"
```

```
#R>
#R> $results
#R> [1]  TRUE FALSE FALSE  TRUE
#R>
#R> $school
#R> [1] Primary   Secondary Tertiary
#R> Levels: Primary < Secondary < Tertiary

# change the names of the elements of the list
names(mylist) <- c("N", "O","R","S")
```

5.4 Data frames

Data frames are the second most important data structure in R. We can think of it as a better version of a data set in Excel. It stacks together observations over many variables, each of these variables being a vector.

```
# mtcars is a built-in data set
data(mtcars)
class(mtcars)
#R> [1] "data.frame"
mtcars
#R>                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#R> Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#R> Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#R> Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#R> Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#R> Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#R> Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#R> Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
#R> Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#R> Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#R> Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#R> Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#R> Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
#R> Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
#R> Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
#R> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
#R> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#R> Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
#R> Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#R> Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#R> Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#R> Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#R> Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#R> AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
#R> Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#R> Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
#R> Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#R> Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#R> Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#R> Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#R> Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
#R> Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

```
#R> Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
head(mtcars)
#R>                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
#R> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#R> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#R> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
#R> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#R> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
#R> Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
str(mtcars)
#R> 'data.frame':   32 obs. of  11 variables:
#R>  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#R>  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
#R>  $ disp: num  160 160 108 258 360 ...
#R>  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
#R>  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
#R>  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
#R>  $ qsec: num  16.5 17 18.6 19.4 17 ...
#R>  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
#R>  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
#R>  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
#R>  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...

# names of the variables in the data frame
names(mtcars)
#R>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#R> [11] "carb"
length(mtcars)
#R> [1] 11
nrow(mtcars)
#R> [1] 32
```

5.4.1 Data frame creation

A data frame can be created manually by providing the function `data.frame` with the corresponding vectors. These vectors should be of same length, otherwise R recycles values. The data frame requires a name for each vector.

```
df <- data.frame(let=LETTERS[1:7],
                 num1=10:16,
                 num2=floor(rnorm(7,100,10)))
df
#R>   let num1 num2
#R> 1   A   10  113
#R> 2   B   11   94
#R> 3   C   12   96
#R> 4   D   13   99
#R> 5   E   14  109
#R> 6   F   15   88
#R> 7   G   16  119
names(df)
#R> [1] "let"  "num1" "num2"

students <- data.frame(name= c("tr", "ga", "mi", "st", "eb", "ha", "fo", "fi"),
```

```
                      age=c(23, 23, 24, 33, 28, 24, 33, 41),
                      like= c(T, T, T, T, F, F, T, T))
students
#R>   name age  like
#R> 1   tr  23  TRUE
#R> 2   ga  23  TRUE
#R> 3   mi  24  TRUE
#R> 4   st  33  TRUE
#R> 5   eb  28 FALSE
#R> 6   ha  24 FALSE
#R> 7   fo  33  TRUE
#R> 8   fi  41  TRUE
```

5.5 A word about attributes

Attributes are important characteristics of vectors (and other data structures).
They can be seen as some meta data that defines the nature of the object. This is to say that a given attribute can determine how a function applies to an object. Here are three important attributes:

- the names, `names()`,
- the dimensions, `dim()`,
- the class, `class`.

To illustrate the use of attributes, recall the case of a factor vector. The fact that the vector on which it builds is given the attribute of a class (factor) turns that vector into a special one.

```
vec <- c(1.7, 1.3, 4, 3.3, 3.3, 2, 2.3, 2.3)
vec
#R> [1] 1.7 1.3 4.0 3.3 3.3 2.0 2.3 2.3
table(vec)
#R> vec
#R> 1.3 1.7   2 2.3 3.3   4
#R>   1   1   1   2   2   1

vec.f <- factor(vec,
                levels=c(1, 1.3, 1.7, 2, 2.3, 2.7, 3, 3.3, 3.7, 4, 5),
                ordered=TRUE)
vec.f
#R> [1] 1.7 1.3 4   3.3 3.3 2   2.3 2.3
#R> Levels: 1 < 1.3 < 1.7 < 2 < 2.3 < 2.7 < 3 < 3.3 < 3.7 < 4 < 5
table(vec.f)
#R> vec.f
#R>   1 1.3 1.7   2 2.3 2.7   3 3.3 3.7   4   5
#R>   0   1   1   1   2   0   0   2   0   1   0
```

5.6 Environments

An environment is similar to a bag/list of names. It can be seen as the space were these names “live”.
Environments can include environments. In that case, the enclosing environment is called the parent environment.
The most common environment is the `globalenv()` also called the `workspace`.

Chapter 6

Simple functions on vectors

In this section, we illustrate calculations with a single vector.

6.1 Element by element evaluation

The main point to notice is how these functions are carried **element by element**. This means that the function is typically applied for each element individually and in sequence.

The following simple example illustrates this procedure and emphasizes the effect on the length of the vectors.

```
# create two numeric vector of same length
visits1 <-  c(12, 2, 45, 75, 65, 11, 3)
visits2 <- c(23, 4, 5, 78, 12, 0, 200)
length(visits1)
#R> [1] 7
length(visits1) == length(visits2)
#R> [1] TRUE

total <- visits1 + visits2
# their sum is the vector of same length, with the element by element sums

length(total)
#R> [1] 7
total
#R> [1]  35   6  50 153  77  11 203
```

Another clear example with the function `^2`.

```
total.p2 <- total^2
total.p2
#R> [1]  1225    36  2500 23409  5929   121 41209

(trop <- total - 2)
#R> [1]  33   4  48 151  75   9 201
length(trop)
#R> [1] 7
```

6.2 Recycling rule

With the emphasis on the length of the vectors above, one could wonder what happens when vectors' length differ. This is an important point to remember: the shortest vector has its elements **recycled** as much as it is necessary.

```
l3 <- c(12, 34, 50)
l2 <- c(10, 3)
tt <- l3 + 5
tt
#R> [1] 17 39 55

ltotal <- l3+l2 # recycling!!!
#R> Warning in l3 + l2: longer object length is not a multiple of shorter
#R> object length
ltotal
#R> [1] 22 37 60
```

6.3 Functions for the whole vector

Some functions are meant to be applied to whole vector and do not necessarily return a vector of the same length of the vector to which it is applied.
Here is an illustration of some of these functions.

```
ages <- c(28, 33, 39, 56, 34, 45, 27, 40)
ages
#R> [1] 28 33 39 56 34 45 27 40
max(ages)
#R> [1] 56
sum(ages)
#R> [1] 302
length(ages)
#R> [1] 8

ages <- c(28, 33, 39, 56, 34, 45, 27, 40, NA)
mean(ages)
#R> [1] NA

# for help on a command, simply type ? in front of it
?mean
```

Chapter 7

Subsetting

7.1 Generalities about subsetting

There are three subsetting operators: `$`, `[]` and `[[]]`.
Some functions also allow to create subsets. We can combine subsetting and assignment to change some parts of an object.
Sub-setting is better used in complement with `str()`.

```
va <- c(13.1, -15.2, 0.3, 2.4, 10.5, -3.6, 9.7) # create the vector va
str(va)
#R>  num [1:7] 13.1 -15.2 0.3 2.4 10.5 -3.6 9.7
str(mtcars)
#R> 'data.frame':   32 obs. of  11 variables:
#R>  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#R>  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
#R>  $ disp: num  160 160 108 258 360 ...
#R>  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
#R>  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
#R>  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
#R>  $ qsec: num  16.5 17 18.6 19.4 17 ...
#R>  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
#R>  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
#R>  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
#R>  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

One can see from this call, that `va` is a simple vector with 7 elements. Subsetting means choosing among these.

7.2 []

Applies to vectors, matrices, lists and data frames.
Can be used with:

```
- positive or negative values,
- many values in a vector,
- logical, `NA`,
- character vectors when names exist.
```

7.2.1 On a vector

Here are a few examples of that object used on a vector.

```
va <- c(13.1, -15.2, 0.3, 2.4, 10.5, -3.6, 9.7)
va[1] # element 1
#R> [1] 13.1
va[c(3:5)] # elements 3 to 5
#R> [1]  0.3  2.4 10.5
va[c(2,7)]
#R> [1] -15.2   9.7
va[-1] # all elements minus the element 1
#R> [1] -15.2   0.3   2.4  10.5  -3.6   9.7
va[c(-2,-7)]
#R> [1] 13.1  0.3  2.4 10.5 -3.6
va[c(-2,-7)]
#R> [1] 13.1  0.3  2.4 10.5 -3.6
va[c(TRUE,TRUE,TRUE,FALSE,FALSE,TRUE,FALSE)]
#R> [1]  13.1 -15.2   0.3  -3.6
va[c(TRUE,FALSE)] # notice the recycling at play here
#R> [1] 13.1  0.3 10.5  9.7
va[NA]
#R> [1] NA NA NA NA NA NA NA
va[] # nothing selected gives the full vector
#R> [1]  13.1 -15.2   0.3   2.4  10.5  -3.6   9.7
names(va)<-letters[1:length(va)] # give names to va
# notice that we subset the vector letters (given by R) and that we don't
# specify the length but give a general value
# now we can subset using names
va
#R>     a     b     c     d     e     f     g
#R>  13.1 -15.2   0.3   2.4  10.5  -3.6   9.7
va[c("a","e","b")]
#R>     a     e     b
#R>  13.1  10.5 -15.2
```

7.2.2 On a list

```
mylist<- list(numbers=c(1:20),
              ournames=c("Jim","Jules"),
              results= c(T,F,F,T),
              school=factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE))
str(mylist)
#R> List of 4
#R>  $ numbers : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
#R>  $ ournames: chr [1:2] "Jim" "Jules"
#R>  $ results : logi [1:4] TRUE FALSE FALSE TRUE
#R>  $ school  : Ord.factor w/ 3 levels "Primary"<"Secondary"<..: 1 2 3

mylist[1] # first element of the list
#R> $numbers
#R>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
class(mylist[1])
#R> [1] "list"
```

```
mylist[[3]][3]
#R> [1] FALSE
```

Notice that the result of this subsetting is a list!

7.2.3 On a matrix

```
set.seed(42)
my.mat <- matrix(floor(runif(30)*10), nrow=5)
my.mat
#R>      [,1] [,2] [,3] [,4] [,5] [,6]
#R> [1,]    9    5    4    9    9    5
#R> [2,]    9    7    7    9    1    3
#R> [3,]    2    1    9    1    9    9
#R> [4,]    8    6    2    4    9    4
#R> [5,]    6    7    4    5    0    8
str(my.mat)
#R>  num [1:5, 1:6] 9 9 2 8 6 5 7 1 6 7 ...
length(my.mat)
#R> [1] 30
```

The structure shows that there are r length(dim(my.mat)) dimensions. For subsetting, we must give r length(dim(my.mat)) dimensions! Same rules as for the vectors apply.

```
my.mat[2,3] # 2nd row, 3rd row
#R> [1] 7
my.mat[-1,]
#R>      [,1] [,2] [,3] [,4] [,5] [,6]
#R> [1,]    9    7    7    9    1    3
#R> [2,]    2    1    9    1    9    9
#R> [3,]    8    6    2    4    9    4
#R> [4,]    6    7    4    5    0    8
colnames(my.mat) <- letters[1:ncol(my.mat)] # give names to the columns
rownames(my.mat) <- LETTERS[1:nrow(my.mat)] # give names to the rows
my.mat
#R>   a b c d e f
#R> A 9 5 4 9 9 5
#R> B 9 7 7 9 1 3
#R> C 2 1 9 1 9 9
#R> D 8 6 2 4 9 4
#R> E 6 7 4 5 0 8
my.mat["C",c("a","c","e")]
#R> a c e
#R> 2 9 9
```

7.3 [[]]

This object is used mainly for lists.

```
mylist<- list(numbers=c(1:20),
              ournames=c("Jim","Jules"),
              results= c(T,F,F,T),
              school=factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE))
```

```
str(mylist)
#R> List of 4
#R>  $ numbers : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
#R>  $ ournames: chr [1:2] "Jim" "Jules"
#R>  $ results : logi [1:4] TRUE FALSE FALSE TRUE
#R>  $ school  : Ord.factor w/ 3 levels "Primary"<"Secondary"<..: 1 2 3
mylist[[1]]
#R>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

7.4 $

This object is usually for data frames, where it gives the variable.
It allows partial matching (e.g., `mtcars$gear` is the same as `mtcars$gea`)

```
data(mtcars)
names(mtcars)
#R>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#R> [11] "carb"
mtcars$mpg
#R>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
#R> [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
#R> [29] 15.8 19.7 15.0 21.4
mtcars[["mpg"]] # just exactly the same, but R users prefer the $
#R>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
#R> [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
#R> [29] 15.8 19.7 15.0 21.4
mtcars["mpg"] # NOT the same at all! [] preserves the class
#R>                      mpg
#R> Mazda RX4           21.0
#R> Mazda RX4 Wag       21.0
#R> Datsun 710          22.8
#R> Hornet 4 Drive      21.4
#R> Hornet Sportabout   18.7
#R> Valiant             18.1
#R> Duster 360          14.3
#R> Merc 240D           24.4
#R> Merc 230            22.8
#R> Merc 280            19.2
#R> Merc 280C           17.8
#R> Merc 450SE          16.4
#R> Merc 450SL          17.3
#R> Merc 450SLC         15.2
#R> Cadillac Fleetwood  10.4
#R> Lincoln Continental 10.4
#R> Chrysler Imperial   14.7
#R> Fiat 128            32.4
#R> Honda Civic         30.4
#R> Toyota Corolla      33.9
#R> Toyota Corona       21.5
#R> Dodge Challenger    15.5
#R> AMC Javelin         15.2
#R> Camaro Z28          13.3
#R> Pontiac Firebird    19.2
```

```
#R> Fiat X1-9           27.3
#R> Porsche 914-2       26.0
#R> Lotus Europa        30.4
#R> Ford Pantera L      15.8
#R> Ferrari Dino        19.7
#R> Maserati Bora       15.0
#R> Volvo 142E          21.4
class(mtcars["mpg"])
#R> [1] "data.frame"
```

7.5 Combining subsetting

Notice that we can often subset further until having the desired subset. Here are a few examples.

```
mtcars$mpg[1:4]
#R> [1] 21.0 21.0 22.8 21.4
mylist[["ournames"]][1]
#R> [1] "Jim"
mylist$ournames[1]
#R> [1] "Jim"
```

7.6 Subsetting with one condition

We can use conditions for subsetting

```
mtcars[mtcars$cyl==8,]
#R>                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#R> Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#R> Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
#R> Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
#R> Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
#R> Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
#R> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
#R> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#R> Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
#R> Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#R> AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
#R> Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#R> Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
#R> Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#R> Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
mtcars[mtcars$cyl==8 & mtcars$carb==4,]
#R>                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
#R> Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
#R> Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
#R> Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
#R> Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
#R> Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
#R> Ford Pantera L      15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
mtcars[mtcars$cyl==8, c("cyl", "mpg", "wt")]
#R>                     cyl  mpg    wt
#R> Hornet Sportabout     8 18.7 3.440
```

```
#R> Duster 360           8 14.3 3.570
#R> Merc 450SE           8 16.4 4.070
#R> Merc 450SL           8 17.3 3.730
#R> Merc 450SLC          8 15.2 3.780
#R> Cadillac Fleetwood   8 10.4 5.250
#R> Lincoln Continental  8 10.4 5.424
#R> Chrysler Imperial    8 14.7 5.345
#R> Dodge Challenger     8 15.5 3.520
#R> AMC Javelin          8 15.2 3.435
#R> Camaro Z28           8 13.3 3.840
#R> Pontiac Firebird     8 19.2 3.845
#R> Ford Pantera L       8 15.8 3.170
#R> Maserati Bora        8 15.0 3.570
```

7.7 Subsetting and assignment

Subsetting can be used to change a part of an object through assignment. Assign `NULL` to delete subset

```
my.mat
#R>   a b c d e f
#R> A 9 5 4 9 9 5
#R> B 9 7 7 9 1 3
#R> C 2 1 9 1 9 9
#R> D 8 6 2 4 9 4
#R> E 6 7 4 5 0 8
my.mat[,1]<-1:nrow(my.mat)
my.mat
#R>   a b c d e f
#R> A 1 5 4 9 9 5
#R> B 2 7 7 9 1 3
#R> C 3 1 9 1 9 9
#R> D 4 6 2 4 9 4
#R> E 5 7 4 5 0 8
mtcars$mpg[1]<-1234
names(mtcars)
#R>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#R> [11] "carb"
mtcars$drat<-NULL # delete variable drat in data frame mtcars
# does not delete in matrix
names(mtcars)
#R>  [1] "mpg"  "cyl"  "disp" "hp"   "wt"   "qsec" "vs"   "am"   "gear" "carb"
```

7.8 Using `which()`

`which()` gives the integers that correspond to the boolean (logical) `TRUE`.
This can help subsetting

```
vb<-1500:1530
vb
#R>  [1] 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513
#R> [15] 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527
#R> [29] 1528 1529 1530
```

```
which(vb%%5==0) # which is divisible by 5 (modulo is 0) ?
#R> [1]  1  6 11 16 21 26 31
# notice that this is asking each element of vb,
# which() reports the positions for which the answer is TRUE
vb[which(vb%%5==0)]
#R> [1] 1500 1505 1510 1515 1520 1525 1530
```

7.9 More advanced stuff

7.9.1 Difference between simplifying and preserving

We say 'preserve' to say the same structure is maintained when subsetting (e.g., a subset of a data frame remains a data frame).
Simplifying does not keep the structure but gives the simplest output possible.
`drop` argument allows to preserve (`drop=FALSE`) or not (`drop= TRUE`).
`[]` usually preserves, `[[]]` usually simplifies.
To better understand, check the classes:

```
all.equal(class(mylist[1]), class(mylist[[1]]))
#R> [1] "1 string mismatch"
class(mylist[1])
#R> [1] "list"
class(mylist[[1]])
#R> [1] "integer"
all.equal(class(mylist), class(mylist[1]))
#R> [1] TRUE
all.equal(class(mylist), class(mylist[[1]]))
#R> [1] "1 string mismatch"
```

We see that `[]` has preserved the class of `mylist` (r class(mylist)), while `[[]]` has not! Another striking example is the following.

```
ma2 <- my.mat[1:2,1:3]
class(ma2)
#R> [1] "matrix"
ma3 <- my.mat[1,1:3]
class(ma3)
#R> [1] "numeric"
```

`ma3` is NOT a matrix anymore!!! This is because one of its dimensions is 1. Losing track of the class when subsetting can generate lots of problems in the middle of a large code. And it is a common source of error! To be sure of keeping the class, we can use `drop=FALSE` (the class will not be dropped to, usually, a vector).

```
ma4 <- my.mat[1,1:3, drop=FALSE]
ma4
#R>   a b c
#R> A 1 5 4
class(ma4)
#R> [1] "matrix"
```

Chapter 8

Conditions

The general purpose of conditions is to control the flow of our code when executed by R.
In R, it builds on statements such as `if` and `else`.

8.1 `if` statement

The simplest form for a condition uses a `if` statement.
The form is then:

```
if (condition) {
# code to be executed if the condition is met
}
```

The key point is that R will run the code until it finds a condition that is met. If it doesn't find any, it continues to the next lines of code.
Here is an example.

```
gains <- c(10, 3,-5,0,-4,12,4)
sum(gains)
#R> [1] 20
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
}
#R> [1] "Congratulations, you are winning!"

gains[1] <- -10 # change the first element of gains
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
} # notice that there is no output, because the condition was not met
```

8.2 `else` statement

What happens when the condition is not met? It's on the user to decide. It can be nothing or... something else!

```
if (condition) {
# code to be executed if the condition is met
} else {
```

```
# code to be executed if the condition is NOT met
}
```

Here is an example.

```
gains<- c(10, 3,-5,0,-4,12,4)
gains[1] <- -10 # change the first element of gains
sum(gains)
#R> [1] 0
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
} else {
  print("You are not winning!")
} # notice that now there is an output!
#R> [1] "You are not winning!"
```

8.3 `else if` statement

`else if` allows us to introduce another condition to our flow of code

```
if (condition_1) {
# code to be executed if the condition_1 is met
} else if (condition_2) {
# code to be executed if the condition_2 is met
} else {
# code to be executed if NEITHER condition_1 NOR condition_2 is met
}
```

We can use as many `else if` statements as we want.
Here is an example with only one `else if`

```
gains <- c(10, 3,-5,0,-4,12,4)
gains[1] <- -10 # change the first element of gains
sum(gains)
#R> [1] 0
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
} else if (sum(gains)==0) {
  print("You just break even!")
} else {
  print("You are losing!")
}
#R> [1] "You just break even!"
```

Notice the potential problem of not putting a `else` statement at the end of the conditions system.

8.4 `ifelse` statement

For **short** conditions, we can use `ifelse`.
The form is

```
ifelse(condition, value if condition met, value if value not met)
```

Here is an example.

```
gains <- c(10, 3,-5,0,-4,12,4)
sign <- ifelse(sum(gains)>=0,"+","-")
sign
#R> [1] "+"
```

8.5 Logical operators: &, | and !

Logical operators are used to combine, mix or negate several conditions: - & means AND - | means OR - ! means NOT

De Morgan's laws may help here.

```
(10%%2==0 & 27%%3==0) # equivalent to (TRUE and TRUE), hence TRUE
#R> [1] TRUE
TRUE & FALSE
#R> [1] FALSE
!TRUE
#R> [1] FALSE
!(TRUE & TRUE)
#R> [1] FALSE
!(TRUE & !TRUE)
#R> [1] TRUE
((10%%2==0 & 27%%2==0)) # equivalent to (TRUE and FALSE), hence FALSE
#R> [1] FALSE
((10%%2==0 | 27%%2==0)) # equivalent to (TRUE or FALSE), hence TRUE
#R> [1] TRUE
```

8.6 Writing and interpreting a condition

Notice that what R looks for in a condition is a either a TRUE or a FALSE.
If it encounters a TRUE, it executes the commands, otherwise, it doesn't.
Remember there are many ways to obtain one of these two logicals. Any of these ways will work as a condition.
Here are examples of less trivial ways of writing a condition

```
gains<- c(10, 3,-5,0,-4,12,4)

if (is.numeric(gains)) {
  print("Seems like it is a numeric vector...")
}
#R> [1] "Seems like it is a numeric vector..."
# here, is.numeric(gains) evaluates to TRUE, hence, the code is executed!
# the beginner's way would be to replace the condition by
# if (class(gains)=="numeric")

if (!is.factor(gains)) {
  print("Yeah! We avoided the factor...")
}
#R> [1] "Yeah! We avoided the factor..."
# the beginner would write if (class(gains)!="factor")
```

Chapter 9

Functions

Everything that happens in R is a function call. If we want R to do something, we must use a function, may it be as simple as a parentheses `(`. If the function is not built-in or coming in a package, one must write it. A function is itself an object. When applied to another object, the function "makes something" to that object and returns an object.

9.1 Structure of a simple function

Here is a simple example of a function that returns the square of the object provided.

```
pow_two <- function(a){
  a^2
}
pow_two(12)
#R> [1] 144
```

9.1.1 Name of the function

In our example above, the name was `pow_two`. To call, i.e., to use the function later in the code, type `pow_two()`.

9.1.2 Arguments

The arguments are given in the parentheses right after `function`.
Here, we use an argument `a`, which can be any object, e.g., a vector

```
pow_two(1:10)
#R>  [1]   1   4   9  16  25  36  49  64  81 100
```

When using this function, we can see that one argument only must be passed.
A value for the argument is then given in the parentheses when calling the function, e.g., `pow_two(12)`.
A function can have several arguments

9.1.3 Commands

The commands will be executed, if possible.
One command source of error is when the class of the argument provided is NOT compatible with the commands in the function

```
pow_two("hello") # this can't work
#R> Error in a^2: non-numeric argument to binary operator
```

9.1.4 Return of a function

The return of a function is given by the last expression executed inside the function.

```
my_f <- function(a){
  a^2
  a^3 # THIS is the last expression executed
}
my_f(5)
#R> [1] 125
```

We can use `return()` in the function in order to chose what it returns.

```
my_f2 <- function(a){
  return(a^2)
  a^3
}
my_f2(5)
#R> [1] 25
```

A function can return only one object; this is not really a problem as we can always stack everything in a `list()` and have the function return the list.

```
my_f3 <- function(a){
  list(ptwo=a^2, pthree=a^3, pfour=a^4)
}
my_f3(5:20)
#R> $ptwo
#R>  [1]  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361 400
#R>
#R> $pthree
#R>  [1]  125  216  343  512  729 1000 1331 1728 2197 2744 3375 4096 4913 5832
#R> [15] 6859 8000
#R>
#R> $pfour
#R>  [1]    625   1296   2401   4096   6561  10000  14641  20736  28561  38416
#R> [11]  50625  65536  83521 104976 130321 160000
```

If we want to use the return of our function, we must assign it to an object.

```
x <- my_f3(1:5) # evaluates the function above on the vector 1:5
x$pthree # shows the element pthree of the saved list x
#R> [1]   1   8  27  64 125
```

An example where we better use `return` is the following

```
pow_three <- function(a){
  if (!is.numeric(a)) {
    return(print("You must give a numeric vector!"))
```

```
  } else {
    return(a^3)
  }

}
pow_three(1:4)
#R> [1]  1  8 27 64
```

9.2 Multiple arguments and their identification

As mentioned above, a function can have several arguments.

```
my_f4 <- function(abc, xyz){
  c(abc^2, xyz^3)
}
my_f4(5) # this won't work because xyz is missing
#R> Error in my_f4(5): argument "xyz" is missing, with no default
my_f4(5, 10)
#R> [1]   25 1000
```

R has at least two ways of identifying arguments, by position and by name.
If no name is specified, R uses the arguments according to their positions in the definition of the function.
In the example above, R understands that `abc=5` and `xyz=10` because of the way they are given in the call `my_f4(5, 10)`.
Identification by names would be

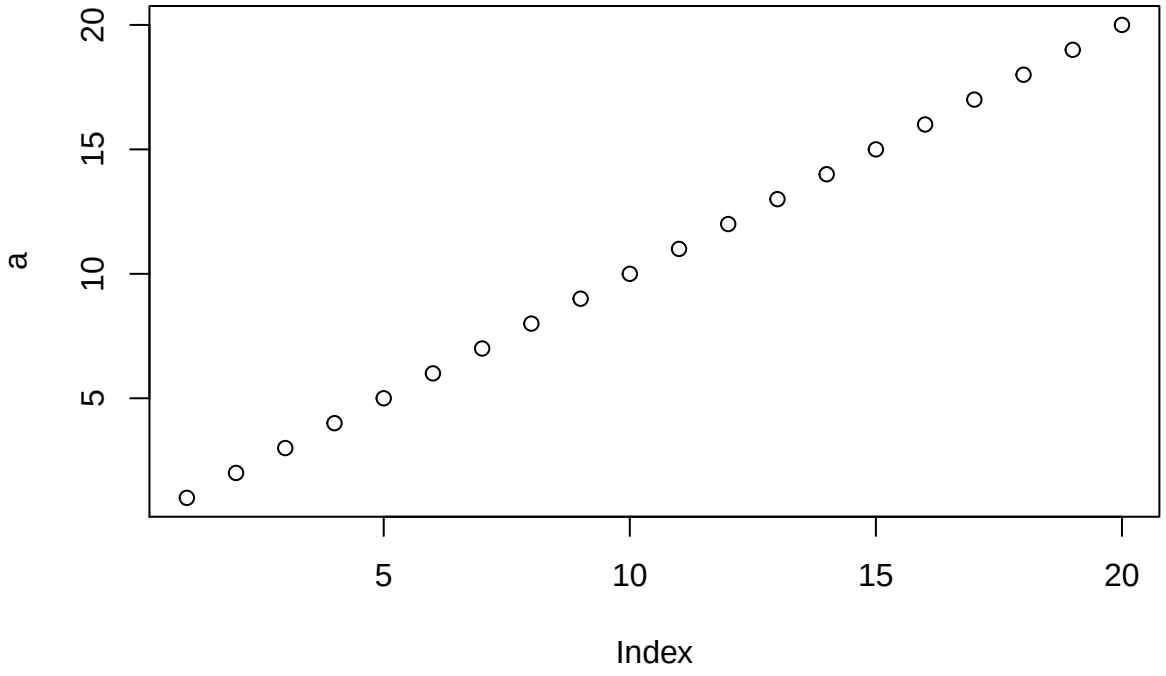
```
my_f4(abc=5, xyz=10) # same as before
#R> [1]   25 1000
my_f4(xyz=5, abc=10)
#R> [1] 100 125
# different from before because the names are more important than the position
my_f4(xyz=5, 10) # a mix between name and position
#R> [1] 100 125
```

9.2.1 Default values

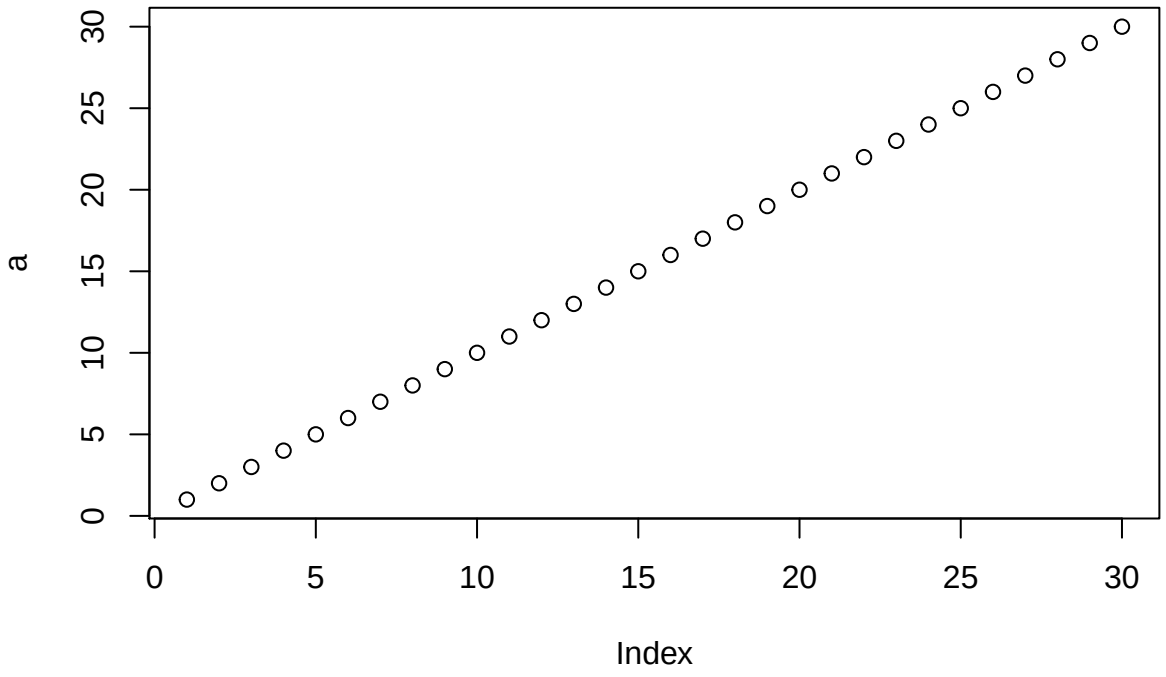
A function can be defined with default values for their arguments.
R will use the values provided in the call; if one is missing, R will use the default value.

```
my_f4 <- function(abc, xyz=2){
  c(abc^2, xyz^3)
}
my_f4(5) # no second argument? doesn't matter, the function has a
#R> [1] 25  8
# default value for it, i.e., 2

my_f5 <- function(a, tada){
  if (tada==TRUE){
    plot(a)
  } else {
    sum(a)
  }
```

```
}

my_f5(1:20, TRUE)
```




```
my_f6 <- function(a, tada=TRUE){
  if (tada==TRUE){
    plot(a)
  } else {
    sum(a)
  }
}
my_f6(1:30)
```



9.3 Scoping

A function creates its own environment. In turn, this environment of the function is a child of the environment where the function was created.
Scoping refers to the set of rules on what environment R looks for the value of a name. For instance, if the code contains `myv <- 12`, then later typing `myv` forces R to retrieve 12 from an environment.
R uses lexical scoping and, under some circumstances, dynamic scoping. We're interested in lexical scoping.

```
mean <- function(x) {
  x+1
}
mean(1:10)
#R>  [1]  2  3  4  5  6  7  8  9 10 11

rm(mean)

mean(1:10)
#R> [1] 5.5
```

9.3.1 Rule 1: name masking

If a name is not defined in a function, R looks one level up, i.e., in the parent:

```
a <- 10

f <- function(){
  b <- 2
  a + b # same as return(a + b)
}

f()
```

```
#R> [1] 12

c <- 11
g <- function() {
  d <- 4
  h <- function() {
    e <- 5
    c + 2*d + e
  }
  h()
}

g()
#R> [1] 24



c <- 11
d <- 100
g <- function() {
  d <- 4
  h <- function() {
    e <- 5
    c + 2*d + e
  }
  h()
}
 g()
#R> [1] 24
```

9.3.2 Rule 2: fresh start

Every time a function is called, a new environment is created. In other words, the function does not know that it has been called before.

```
a
#R> [1] 10
t <- function() {
  if (exists("a")) {
    a <- a + 2
    a
  } else {
    a <- 20
    a
  }
}
t()
#R> [1] 12

t()
#R> [1] 12
t()
#R> [1] 12
```

```
a <- 4
# gives 6 because when the function t looks for 'a', NOW, it finds it in the parent
t()
#R> [1] 6


l <- function() {
  a <- 5
  # 'a' is defined in the environment, so there is no need to look into the parent
  if (exists("a")) {
    a <- a + 2
    a
  } else {
    a <- 10
    a
  }
}
l()
#R> [1] 7
```

9.3.3 Rule 3: dynamic lookup

R looks for values when the function is called. If the environment where R will look for values has changed in between calls, the result of the function may be different. This is a source of error that should avoided.

```
u <- function() a^2
a <- 2
u()
#R> [1] 4
# needs a value for 'a' because it does not define one within its environment;
# therefore, u() looks up to, in this case, the global environment
a <- 3
u()
#R> [1] 9
```

9.3.4 Functions inside functions

Consider the following example of a function, it returns a function!

```
m.power <- function(pow){
  m.exp <- function(b){ b^pow }
  m.exp # return the m.exp function
}
```

We can create many functions with that function. Importantly, notice that functions preserve the environment in which they were created.

```
mycube <- m.power(3)
mycube(4)
#R> [1] 64

mysquare <- m.power(2)
mysquare(5)
```

```
#R> [1] 25

# ls(environment(m.power)) gives the global environment
ls(environment(mycube))
#R> [1] "m.exp" "pow"

w <- function(k){
  q <- 10
  k*q
}

z <- function() w(2) # z is itself a function that inherits the environment of w
# but z, here, is forced to use 2 as an argument

z()
#R> [1] 20
```

9.3.5 Examples

The following example illustrates lexical scoping.

```
yy <- 10

ff <- function(xx){
  yy <- 2
  yy^2 + gg(xx)
}

gg <- function(xx){
  xx*yy
}

ff(3)
#R> [1] 34
```

As another illustration, we could try to guess what `g(2)` produces after the following commands. Hint: remember the in which environment the functions were created because it tells you what are the correct bindings (parent).

```
a <- 1
b <- 2
f <- function(x){
  a*x + b
}
g <- function(x){
  a <- 2
  b <- 1
  f(x)
}

g(2)
#R> [1] 4
```

Compare to `g(2)` here.

```
a <- 1
b <- 2
f <- function(a,b){
  return( function(x) {a*x + b})
}
g <- f(2,1)
g(2)
#R> [1] 5
```

Chapter 10

Loops (avoid them)

10.1 Generalities about loops

Loops allow to iterate a set of functions / lines of code over a predefined list of elements.
They are common in many languages but are not highly regarded in R because:

- they do not represent an efficient way of producing a result,
- functionals, functions that return a vector such as the `apply` family, are preferred.

This section serves as an introduction to the functionals because it helps understand the structure of the problem.

10.2 Introducing loops with a silly example (bad R coding)

Suppose our gains at a game over the week are, in euros,

```
gains <- c(10, 3,-5,0,-4,12,4)
gains
#R> [1] 10  3 -5  0 -4 12  4
```

Now, suppose we want to transform each gain into dollars, i.e., we must multiply each gain by 1.30. An intuitive way would be to take each gain *separately* and multiply it 1.12. Let's do it by hand:

```
gains_d <-numeric(length(gains)) # create a numeric vector of same length as gains
gains_d[1] <- gains[1]*1.12
gains_d[2] <- gains[2]*1.12
gains_d[3] <- gains[3]*1.12
gains_d[4] <- gains[4]*1.12
gains_d[5] <- gains[5]*1.12
gains_d[6] <- gains[6]*1.12
gains_d[7] <- gains[7]*1.12
gains_d
#R> [1] 11.20  3.36 -5.60  0.00 -4.48 13.44  4.48
```

We recognize that the structure of each line of code: the first line is for the element 1 of the vectors, the second line for the second element of the vectors, etc.
Doing a loop builds on that observation. But it asks the program to make the change between each line automatically.
In this silly case, we want the program to evaluate a line with the value 1 and then, when it has finished, do

the same line but with a 2 instead of a 1. And we want it to do it **for** the elements 1, 2, 3, 4, ..., 7.
This is what a loop does!

```
# create an empty numeric vector of same length as gains
gains_d2 <-numeric(length(gains))

# now the loop
# first for i=1, then i=2, then i=3, etc... until i=length(gains)
for (i in 1:length(gains)) {
  gains_d2[i] <- gains[i]*1.12
}
gains_d2
#R> [1] 11.20  3.36 -5.60  0.00 -4.48 13.44  4.48
```

10.3 General form of a `for` loop

The general form of a loop is the following

```
for (i in list-of-elements) {
# code invoking i identifier
}
```

A few words about this simple form are the following.

- The `i` is the identifier to be used in the code; we can use anything (e.g., `x`, `kk`, `element`, ...); we only need to be consistent and, if any, use that identifier in the code inside the loop.
- The `in` is self-describing.
- The list of elements is generally a vector, but it can also be a list.
- The loop executes some code but does not need to return an object.

10.4 No identifier in the code?

The program will run the code in the loop until it finds the identifier, replaces it with the current value of the loop and executes all the commands.
If it doesn't find it, then the program executes everything till the end of the loop and then starts again with the next value of the loop.

```
for (kk in 1:4) {
  print("I'm working on it...")
}
#R> [1] "I'm working on it..."
#R> [1] "I'm working on it..."
#R> [1] "I'm working on it..."
#R> [1] "I'm working on it..."

for (kk in 1:5) {
  print(paste("This is loop number", kk))
}
#R> [1] "This is loop number 1"
#R> [1] "This is loop number 2"
#R> [1] "This is loop number 3"
#R> [1] "This is loop number 4"
#R> [1] "This is loop number 5"
```

10.5 A good example of a loop

Suppose we want to know, every day, what is our cumulative gain until and including that day. This implies that we must to create another `cumulative_gains` with the same length as `gains`.

Then, we could run the following loop.

```
# create an empty vector
cumulative_gains <-numeric(length(gains))

cumulative_gains[1] <- gains[1]

for (k in 2:length(gains)){
  cumulative_gains[k]<-cumulative_gains[k-1] + gains[k]
}
gains
#R> [1] 10  3 -5  0 -4 12  4
cumulative_gains
#R> [1] 10 13  8  8  4 16 20
```

10.6 Elements may not need to be numeric

The examples above could give the impression that we must loop over numeric values. But it is not so!
The identifiers above simply took the values, one after the other, of the vector provided. Since we always gave a numeric vector, the identifier was always numeric.
Here, we show that we can also loop over other types of vectors, for instance a character vector.
The example is simple: we want to count the number of letters in each element of the character vector and print the result in the console. You can use the built-in function `nchar()`; `nchar("Cologne")=7`.

```
cities <- c("Lisbon", "Paris", "Washington", "Antananarivo")
cities
#R> [1] "Lisbon"       "Paris"        "Washington"   "Antananarivo"

for (cit in cities) {
  print(paste(cit, "has", nchar(cit), "letters."))
}
#R> [1] "Lisbon has 6 letters."
#R> [1] "Paris has 5 letters."
#R> [1] "Washington has 10 letters."
#R> [1] "Antananarivo has 12 letters."
```

To store these values in a vector `number_letters`, we could write

```
number_letters <- NULL

for (cit in cities) {
  number_letters <- c(number_letters, nchar(cit))
}

names(number_letters) <- cities
number_letters
#R>       Lisbon        Paris   Washington Antananarivo
#R>            6            5           10           12
```

10.7 More advanced stuff

10.7.1 Loop over a `list`

We can loop over the elements of a `list`. Remember, however, that the elements of a `list` can be of any type. Therefore, we must make sure that the code inside the loop can be applied to each element of the `list`.

```
mylist<- list(numbers=c(1:20),
              cities=c("Cologne","Lisbon"),
              results= c(T,F,F,T),
              school=factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE))

for (kk in mylist) {
  print(length(kk))
}
#R> [1] 20
#R> [1] 2
#R> [1] 4
#R> [1] 3
```

Notice that the previous call could be done with a loop over a numeric vector.

```
# (kk in 1:4)
for (kk in 1:length(mylist)) {
  print(length(mylist[[kk]]))
}
#R> [1] 20
#R> [1] 2
#R> [1] 4
#R> [1] 3
```

10.7.2 `while` loop

The command `while` introduces a loop that is run while a certain condition is satisfied. Here is an example.

```
kk <- 6
while (kk < 10) {
  print(kk)
  kk <- kk + 1
}
#R> [1] 6
#R> [1] 7
#R> [1] 8
#R> [1] 9
```

Notice that the most important line is the last in the loop `kk <- kk + 1`. This makes the identifier change value, in this case, it increases of 1.

If the identifier does not change, the program keeps running as long as the condition `(kk < 10)` is satisfied… i.e., forever! Make sure that the condition is not satisfied at some point after some loops!

```
repetition <- 0
while (repetition <= 9 ) {
  print(paste("That's " , repetition, "! One more! Come on!", sep=""))
  repetition<- repetition +1
```

```
}
#R> [1] "That's 0! One more! Come on!"
#R> [1] "That's 1! One more! Come on!"
#R> [1] "That's 2! One more! Come on!"
#R> [1] "That's 3! One more! Come on!"
#R> [1] "That's 4! One more! Come on!"
#R> [1] "That's 5! One more! Come on!"
#R> [1] "That's 6! One more! Come on!"
#R> [1] "That's 7! One more! Come on!"
#R> [1] "That's 8! One more! Come on!"
#R> [1] "That's 9! One more! Come on!"
```

10.7.3 `break` & `next`

If we insert `break` in a loop, the program stops everything, i.e., the remaining code in the loop is skipped and the loop is not iterated over anymore.

You may want to use it in combination of a condition

```
if (condition==TRUE) {
  break
}
```

```
repetition <- 0
while (repetition <= 9 ) {
  if (repetition==5){
    print(paste("That's " , repetition,
                "! Well done! I know we said 10, but that's enough for today!", sep=""))
    break
  }
  print(paste("That's " , repetition, "! One more! Come on!", sep=""))
  repetition<- repetition +1
}
#R> [1] "That's 0! One more! Come on!"
#R> [1] "That's 1! One more! Come on!"
#R> [1] "That's 2! One more! Come on!"
#R> [1] "That's 3! One more! Come on!"
#R> [1] "That's 4! One more! Come on!"
#R> [1] "That's 5! Well done! I know we said 10, but that's enough for today!"
```

If we insert `next` in a loop, the program skips the remainder of the code in the loop and starts with the next value in the loop. Again, it's usually combined with a condition.

Chapter 11

Importa data to R

11.1 General idea

The first step in order to analyze data with R is to get the data in R.
Data out there exists in many formats and it is not always consistently recorded. In R speaking, we will need to `read` the data from a file and put it in an object, often a data frame.
Base R has has base functions to read some types of files. For special types of files, special packages are needed.

Important note: these base functions are introduced here for documentation purpose only. Indeed, their task is usually better achieved by the functions developed in the `tidyverse` (see Chapter 13 and following).

11.2 Reading rectangular data

Rectangular data is intended here as a data set with values of variables in columns and each row representing a case. Arguably, this is the simplest and probably most common way to store data sets.
The ad hoc function in R is `read.table`. Other similar functions exist, e.g., `read.csv`, but they are essentially a wrapper of `read.table`.
This function reads rectangular data and assigns it to an object, in almost all cases a data frame. The arguments of the function are numerous are, depending on the case, potentially very useful (check `?read.table`). We mention here a few of them in the typical call:

```
my.data <- read.table(...)
```

`file` is the name of the file to be read. It requires a valid path. It can also be an `url`, with some adaptations.
`header` is a logical indicating whether the data's first line gives the names of the variables.
`sep` indicates the character that separates the values between columns.

```
# assuming we have the file AH001.txt in the same folder
df <- read.table(file="AH001.txt",
                 header=TRUE,
                 sep=",")
head(df)
#R>   Area AreaCode AreaCode2 Grain Year Month LPrice HPrice
#R> 1   AH        1        NA    BO 1738     9     98    120
#R> 2   AH        1        NA    BO 1738    10     99    120
#R> 3   AH        1        NA    BO 1738    11    100    120
#R> 4   AH        1        NA    BO 1738    12     93    120
```

```
#R> 5   AH        1        NA    BO 1739     1     90    120
#R> 6   AH        1        NA    BO 1739     2     98    120
```

11.3 Read other data types

R can import virtually all types of data. For special types, a ad hoc package will be necessary. `foreign` is one of them and it allows to import data from other applications. Its functions are of the form

`read.XXX`

where the `XXX` is an extension specific to the external software. Examples include "read.dta" for Stata data files, `read.spss` for SPSS files or `read.dbf` for dBASE files.

A very important data source type is Excel. Functions for this type of data are covered in Chapter 14 from the part on `tidyverse`.

11.4 Scanning a file

The base function `scan` can sometimes be useful, though it is not used to import data. `scan()` imports to a vector (or a list), which can then be used.

- `what` gives the type of vector to be imported.
- `skip` is the number of lines to be skipped in the file.
- `nlines` determines the number of lines to be read and imported.

```
df <- scan("AH001.txt", what=character())
#df
df <- scan("AH001.txt", what=character(), skip=1, sep=",")
#df
given.line <- scan("AH001.txt", what=character(), nlines=1, sep=",")
given.line
#R> [1] "Area"      "AreaCode"  "AreaCode2" "Grain"     "Year"      "Month"
#R> [7] "LPrice"    "HPrice"
```

Chapter 12

Simple plots

This short chapter has also only an information purpose. The package `ggplot2` described in Chapter 15 allows to make much richer and elegant plots.

12.1 Line plots

The simplest plot is a scatter plot of one variable plotted against an index (1, 2, 3, ...) on the horizontal axis. In order to (scatter) plot one vector against the other, it is compulsory that they are both numeric and have the same length.
The following plots illustrate the creation of a line plot in incremental steps. In the steps, the following functions and arguments are introduced:

- `plot` is the main function for the plot.
- `type` argument type gives how the vector should be rendered in the plot; the `type` of the plot is any of Table 12.1.

```
sales <- c(20, 18, 24, 36, 30)
date <- c(15, 16, 17, 18, 19)
# left
plot(sales)
# middle
plot(date, sales)
# right
plot(date, sales, type="l")
```

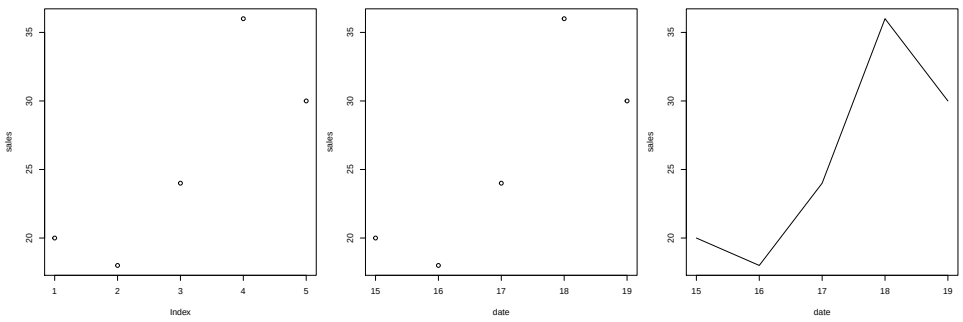


Figure 12.1: Creating a simple line plot.

Table 12.1: Plot types.

Symbol	Description
p	Points
l	Lines
b	Both
c	The lines part alone of b
o	Both overplotted
s	Steps
h	Histogram like (vertical lines)
n	No plotting

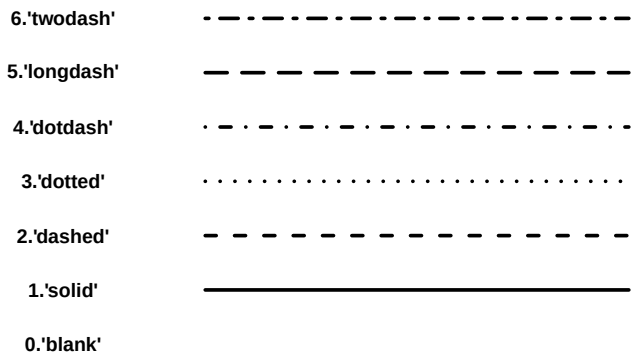


Figure 12.2: Line types.

We now add further customization with new functions and arguments.

- `col` determines the color; see this Rcolor.pdf online for details.
- The `lty` argument refers to the type of line, to be chosen from the values shown in Figure 12.2,
- `lines` adds the plot of a vector to a previously opened plot.
- `axis` is a function that changes the axis given in its first argument with 1, 2, 3, 4 referring to the bottom, left, top, and right axis, respectively.
- `at` simply states for what values of the axis the labels should correspond.
- `las` gives the orientation of the labels (e.g., `1=horizontal`).
- `xlab` and `ylab` are the x and y axes labels, respectively.
- `xlim` and `ylim` set a numerical limit for the x and y axes labels, respectively, notice that a vector of length 2 is necessary for each.

```
sales2 <- c(12, 19, 22, 28, 32)
# left
plot(date, sales, type="l", col="blue")
lines(date, sales2, type="b", col="red", lty=3)
# middle
s.range <- c(min(sales,sales2),max(sales,sales2))
plot(date, sales, type="l", col="blue", ylim=s.range)
lines(date, sales2, type="b", col="red", lty=3)
# right
plot(date, sales, type="l", col="blue",
     ylim=s.range,
     axes= FALSE,
     xlab = "Day",
     ylab= "Sales")
lines(date, sales2, type="b", col="red", lty=3)
```

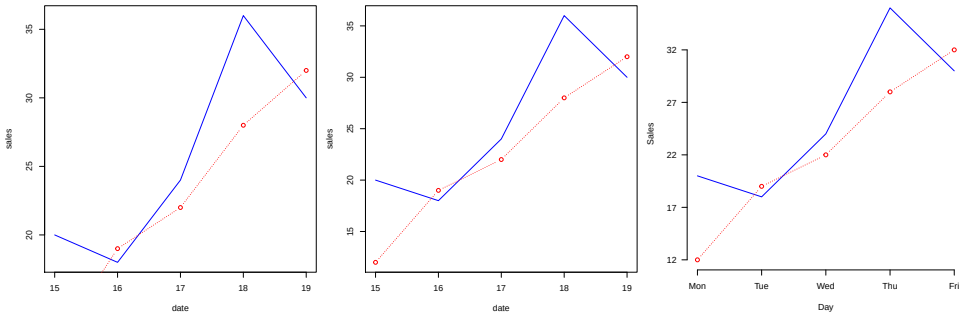


Figure 12.3: Custom a simple line plot.

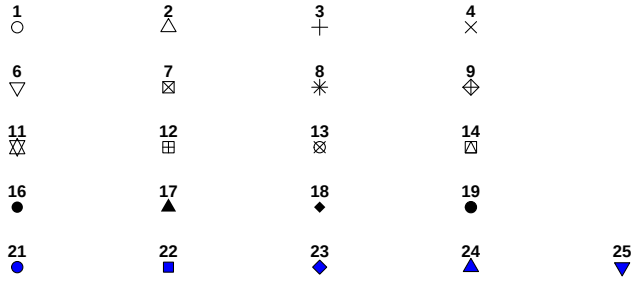


Figure 12.4: Point types.

```
axis(1, at=date, lab=c("Mon","Tue","Wed","Thu","Fri"))
axis(2, las=1, at=seq(min(s.range),max(s.range),5))
```

We now add further elements such as a legend or a title.

- `legend` gives and places a legend; its first arguments are the location on the x and y axis respectively (this can be replaced by expressions such as `topleft`); the third is the legend itself, completed with colors and other formatting; notice that since the legend specifies two series, the formatting must also be given for each of them in vectors of length 2.
- `pch` determines the shape of the point in the plot, to be chosen from Figure 12.4.
- `cex` gives the expanding factor of the text.
- `bty` specifies whether of not the legend should be in a box (default, "o") or not "n".
- `title` gives the title to the plot.

```
plot(date, sales, type="l", col="blue", ylim=s.range,
      axes= FALSE, xlab = "Day", ylab= "Sales")
lines(date, sales2, type="b", col="red", lty=3)
axis(1, at=date, lab=c("Mon","Tue","Wed","Thu","Fri"))
axis(2, las=1, at=seq(min(s.range),max(s.range),5))
legend(date[1], s.range[2], c("Tom","Mark"), cex=1,
   col=c("blue","red"), pch=c(NA,1), lty=c(1,3), bty="n")
title(main="Week Sales", col.main="black", font.main=2)
```

One can help but notice how cumbersome the coding of the graph is, in particular when one needs to retype the whole code for the same graph.

12.2 Bar graphs and histograms

This subsection illustrates the creation of these graphs with a few commented examples.

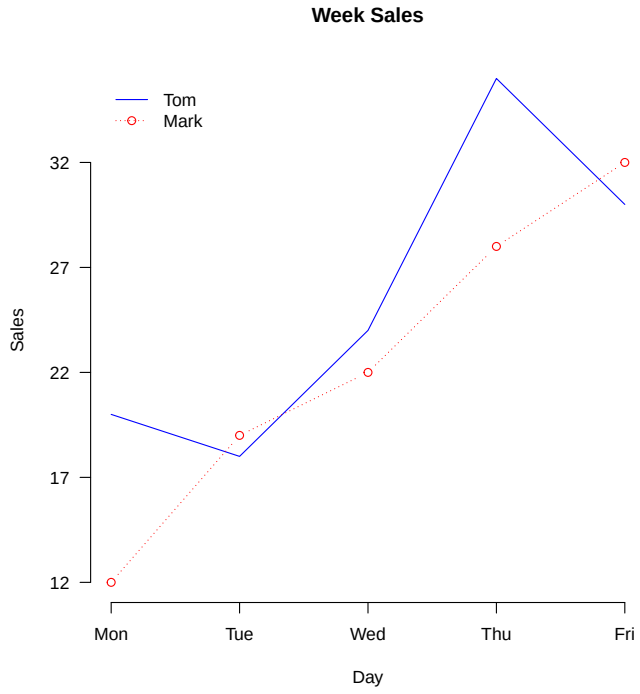


Figure 12.5: Further customization of a simple line plot.

- `barplot` takes a vector or a matrix as first argument. It is therefore important to double check how this latter is created.
- `density` applies to the colors and a single value would be recycled.
- `horiz` defaults to `FALSE` and gives the direction of the plot.
- `border` applies to the borders of the bars.
- `beside` forces side-by-side bars instead of stacking bars.

```
barplot(sales,
        main="Week Sales - Tom",
        names.arg=c("Mon","Tue","Wed","Thu","Fri"),
        border="lightblue",
        col = "blue")
barplot(sales,
        main="Week Sales - Mark",
        names.arg=c("Mon","Tue","Wed","Thu","Fri"),
        border="red",
        col = "red",
        horiz = TRUE)

barplot(matrix(c(sales, sales2), ncol=5, byrow = TRUE),
        beside = TRUE,
        names.arg=c("Mon","Tue","Wed","Thu","Fri"),
        col=c("blue", "red"),
        density = 30,
        main="Week Sales")
```

Turning to histogram, the function and some arguments are the following.

- `hist` is the function to create an histogram from the values of a vector.
- `breaks` is the important argument as it controls the width of the bars.

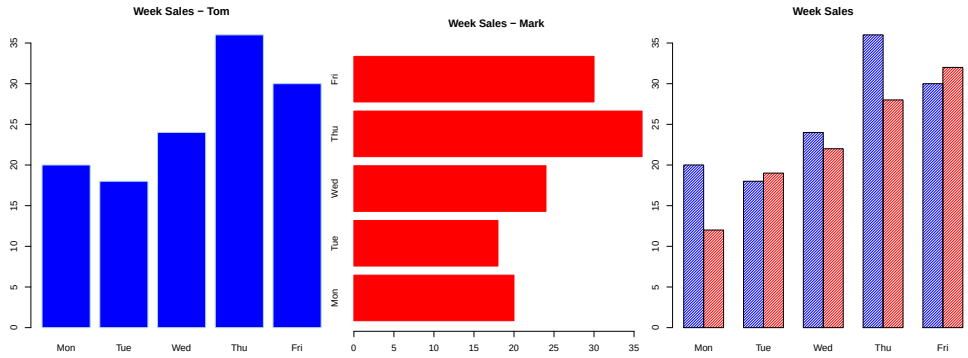


Figure 12.6: Simple bar plots.

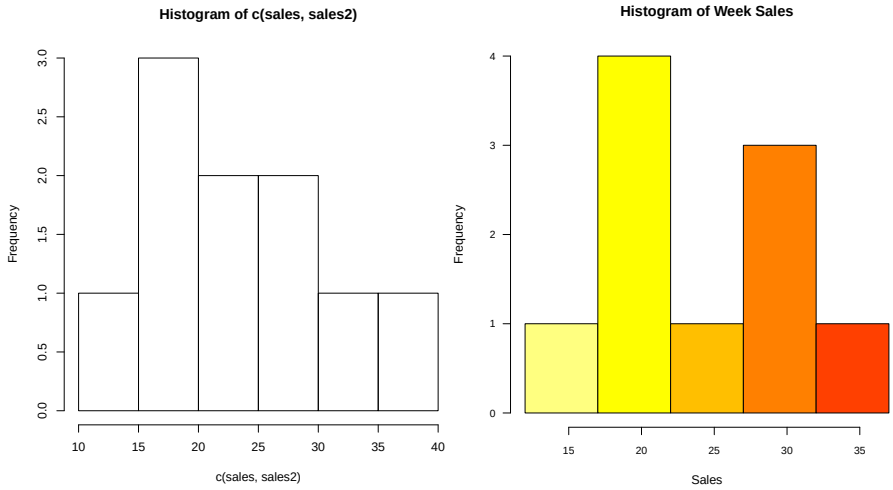


Figure 12.7: Simple histograms.

- `freq` determines whether the frequency (default) of the density should be plotted.

```
hist(c(sales, sales2))

brks <- seq(min(s.range), max(s.range)+3, 5)
hist(c(sales, sales2),
    col=rev(heat.colors(length(brks))),
    breaks=brks,
    main="Histogram of Week Sales",
    las=1, cex.axis=0.8,
    freq=TRUE,
    xlab="Sales")
```

12.2.1 Other plots

Other types of plots are called by the following functions.

- `pie` for the pie (round) charts.
- `boxplot` for the box-and-whisker plot.
- `dotchart` for the Cleveland dot plot.
- …

Part III

Tidyverse

Chapter 13

What is `tidyverse`

`tidyverse` is a bundle of packages due mainly by Hadley Wickham. Essentially, it is a redesign of the core functions of R with a coherent philosophy. It comes close to become a complete and self-contained system on its own thanks to all its packages.
Arguably, it is also an improvement over the standard R packages: faster, more consistent and more beautiful.
The huge benefits of embracing the `tidyverse` come at the cost needing to learn a sub-dialect of R. This can be daunting for a beginner in R programming that put lots of effort into learning basic R functions.
I understand, however, that these initial efforts will still pay off with this new system. And the advantages will become overwhelming.
The next chapters all develop a relevant package, selected from the `tidyverse`. The list of all packages along with a fuller description of the system can be found on tidyverse.org.

To begin,

```
install.packages("tidyverse")
```

Chapter 14

Read: `readr` and `readxl`

`readr`

As shown in Chapter @ref(#import), base R has many functions to read some types of files.
`readr` improves on them because it:

- is much faster,
- has mainly functionalities,
- converts data directly into tibbles,
- handles the conversion of the type of data in a better way.

Notice that `readr` uses names of functions that are very close to the base R functions: e.g., `read_csv` instead of `read.csv` in base R.
The functions form a `read_xxx` family where 'xxx' stands for the way the data you want read was recorded:

- `csv` for comma separated values,
- `tsv` for tab separated values,
- `delim` for any delimiter between values,
- other exist as well, for instance for columns separated by a white space.

In my experience, using the right member of the family yields better results, for instance in parsing the columns.

14.0.1 `read_delim`

We'll describe `read_delim` because functions of the family behave are special cases of this function. For instance, `read_csv` is equivalent to `read_delim` with a `,` as a character to separate the columns.

- The `file` argument gives the name of a file, possibly with the path to it, if it is not in the same folder. Again, the path can also be an url.

- `delim` provides the character that separates columns in the data.

```
# assuming that the AH001.txt is in the folder
df <- read_delim("AH001.txt", delim = "," , trim_ws=TRUE)
#R> Parsed with column specification:
#R> cols(
#R>   Area = col_character(),
#R>   AreaCode = col_character(),
#R>   AreaCode2 = col_character(),
#R>   Grain = col_character(),
```

```
#R>   Year = col_integer(),
#R>   Month = col_integer(),
#R>   LPrice = col_integer(),
#R>   HPrice = col_integer()
#R> )
df
#R> # A tibble: 9,765 x 8
#R>    Area  AreaCode AreaCode2 Grain  Year Month LPrice HPrice
#R>    <chr> <chr>    <chr>     <chr> <int> <int>  <int>  <int>
#R>  1 AH    001      <NA>      BO     1738     9     98    120
#R>  2 AH    001      <NA>      BO     1738    10     99    120
#R>  3 AH    001      <NA>      BO     1738    11    100    120
#R>  4 AH    001      <NA>      BO     1738    12     93    120
#R>  5 AH    001      <NA>      BO     1739     1     90    120
#R>  6 AH    001      <NA>      BO     1739     2     98    120
#R>  7 AH    001      <NA>      BO     1739     3    100    124
#R>  8 AH    001      <NA>      BO     1739     4    100    121
#R>  9 AH    001      <NA>      BO     1739     9     90    125
#R> 10 AH    001      <NA>      BO     1740     6     75    130
#R> # ... with 9,755 more rows
class(df)
#R> [1] "tbl_df"     "tbl"        "data.frame"
```

Notice an important message, the types of the columns were guessed by the function. For instance, the column "Area" was guessed to be of type "character". With that respect, the next arguments are particularly useful.
- First, `trim_ws` is a logical for whether the leading white space in each recording should be erased before parsing. With a white space, columns of numeric values with different number of digits could be guessed as character vectors.
Second, we can override the guess of `read_delim` by specifying the types: this is called "to parse" a file.
We can parse into many types and use abbreviations to set them: c = character, i = integer, n = number, d = double, l = logical, D = date, T = date time, t = time, ? = guess.

All of these parsing calls can cause unexpected problems. For instance, one may need to change the symbol for decimal with the extra argument for the column type `locale=locale(decimal_mark=",")`. Notice the behavior when an observation is not of the same type as expected: it is changed to `NA`.
The function `problems` lists the rows with problems.

```
df <- read_delim("AH001.txt", delim=",", trim_ws=TRUE,
                 col_types = cols(LPrice="d", HPrice ="d") )
df
#R> # A tibble: 9,765 x 8
#R>    Area  AreaCode AreaCode2 Grain  Year Month LPrice HPrice
#R>    <chr> <chr>    <chr>     <chr> <int> <int>  <dbl>  <dbl>
#R>  1 AH    001      <NA>      BO     1738     9     98    120
#R>  2 AH    001      <NA>      BO     1738    10     99    120
#R>  3 AH    001      <NA>      BO     1738    11    100    120
#R>  4 AH    001      <NA>      BO     1738    12     93    120
#R>  5 AH    001      <NA>      BO     1739     1     90    120
#R>  6 AH    001      <NA>      BO     1739     2     98    120
#R>  7 AH    001      <NA>      BO     1739     3    100    124
#R>  8 AH    001      <NA>      BO     1739     4    100    121
#R>  9 AH    001      <NA>      BO     1739     9     90    125
#R> 10 AH    001      <NA>      BO     1740     6     75    130
#R> # ... with 9,755 more rows
problems(df)
```

```
#R> # tibble [0 x 4]
#R> # ... with 4 variables: row <int>, col <int>, expected <chr>, actual <chr>
```

Notice that the names of the variables are not within `" "`. This is a general feature of the `tidyverse`.
By default, the first line of the data file is used for naming the columns. This can be changed.
- `col_names` is the vector of names set for the variables. - `skip` determines how many lines to be skipped in the file. This is useful when the file does contain names that one does not want to keep.

```
df <- read_delim("AH001.txt", delim = "," , trim_ws=TRUE,
                 skip = 1, col_names = FALSE)
#R> Parsed with column specification:
#R> cols(
#R>   X1 = col_character(),
#R>   X2 = col_character(),
#R>   X3 = col_character(),
#R>   X4 = col_character(),
#R>   X5 = col_integer(),
#R>   X6 = col_integer(),
#R>   X7 = col_integer(),
#R>   X8 = col_integer()
#R> )
df
#R> # A tibble: 9,765 x 8
#R>    X1    X2    X3    X4       X5    X6    X7    X8
#R>    <chr> <chr> <chr> <chr> <int> <int> <int> <int>
#R>  1 AH    001   <NA>  BO     1738     9    98   120
#R>  2 AH    001   <NA>  BO     1738    10    99   120
#R>  3 AH    001   <NA>  BO     1738    11   100   120
#R>  4 AH    001   <NA>  BO     1738    12    93   120
#R>  5 AH    001   <NA>  BO     1739     1    90   120
#R>  6 AH    001   <NA>  BO     1739     2    98   120
#R>  7 AH    001   <NA>  BO     1739     3   100   124
#R>  8 AH    001   <NA>  BO     1739     4   100   121
#R>  9 AH    001   <NA>  BO     1739     9    90   125
#R> 10 AH    001   <NA>  BO     1740     6    75   130
#R> # ... with 9,755 more rows
colnames(df) <- LETTERS[1:length(df)]
df
#R> # A tibble: 9,765 x 8
#R>    A     B     C     D         E     F     G     H
#R>    <chr> <chr> <chr> <chr> <int> <int> <int> <int>
#R>  1 AH    001   <NA>  BO     1738     9    98   120
#R>  2 AH    001   <NA>  BO     1738    10    99   120
#R>  3 AH    001   <NA>  BO     1738    11   100   120
#R>  4 AH    001   <NA>  BO     1738    12    93   120
#R>  5 AH    001   <NA>  BO     1739     1    90   120
#R>  6 AH    001   <NA>  BO     1739     2    98   120
#R>  7 AH    001   <NA>  BO     1739     3   100   124
#R>  8 AH    001   <NA>  BO     1739     4   100   121
#R>  9 AH    001   <NA>  BO     1739     9    90   125
#R> 10 AH    001   <NA>  BO     1740     6    75   130
#R> # ... with 9,755 more rows
```

Data often contain specific special character, in particular for comments and NA's.

- `comment` takes the character that signals comments.

- `na` signals what values should be consider as NA's.

```
df <- read_delim("AH001.txt", delim = "," , trim_ws=TRUE,
                 skip = 1, col_names = FALSE,
                 comment = "%", na = "9999")
```

14.1 readxl

Lots of rectangular data out there is in Excel files. One way is to handle it could be to first transform the data into a `.csv` file and then import it as described above.
Thanks to the `readxl` package, one can also read it directly.[^Notice that R can also write to an Excel file. Why would one do that? Maybe because collaborators only work with Excel. This is achieved with the package `XLConnect` but we won't dig into these details here.] As part of the `tidyverse` bundle, `readxl` shares features and rules with `readr`.
Below are some of the specific features.

- `read_excel` is the function that reads the data in the file and assigns it to an object, as illustrated in the following chunk.

```
df <- read_excel("data_china.xlsx")
df
#R> # A tibble: 19 x 8
#R>    Area  AreaCode AreaCode2 Grain  Year Month LPrice HPrice
#R>    <chr>    <dbl> <lgl>     <chr> <dbl> <dbl>  <dbl>  <dbl>
#R>  1 AH           1 NA        BO     1738     9     98    120
#R>  2 AH           1 NA        BO     1738    10     99    120
#R>  3 AH           1 NA        BO     1738    11    100    120
#R>  4 AH           1 NA        BO     1738    12     93    120
#R>  5 AH           1 NA        BO     1739     1     90    120
#R>  6 AH           1 NA        BO     1739     2     98    120
#R>  7 AH           1 NA        BO     1739     3    100    124
#R>  8 AH           1 NA        BO     1739     4    100    121
#R>  9 AH           1 NA        BO     1739     9     90    125
#R> 10 AH           1 NA        BO     1740     6     75    130
#R> 11 AH           1 NA        BO     1740     7     75    130
#R> 12 AH           1 NA        BO     1740     8     75    150
#R> 13 AH           1 NA        BO     1740     9     75    150
#R> 14 AH           1 NA        BO     1740    10     75    150
#R> 15 AH           1 NA        BO     1740    11     75    150
#R> 16 AH           1 NA        BO     1740    12     75    150
#R> 17 AH           1 NA        BO     1741     1     75    150
#R> 18 AH           1 NA        BO     1741     2     75    138
#R> 19 AH           1 NA        BO     1741     3     90    125
```

Importantly, notice that the exact extension does not matter as `readxl` recognizes both `.xls` and `.xlsx`. Notice that if the file contains multiple sheets, these can all be accessed.

- `excel_sheets` is a function that yields the names of the different sheets in a file.

```
excel_sheets("data_china.xlsx")
#R> [1] "AH" "FJ"
```

To know the specific sheets allows to call them directly by name. Alternatively, this can also be done by a number.

- The `sheet` argument specifies the required sheet either by a name or by a number.

```
df <- read_excel("data_china.xlsx", sheet="FJ")
```

A specific range for the data to be read can also be specified. Here are arguments that allow that.

- `n_max` for the maximum number of line to be read.
- `range` for a specific range, describe in one of the possible ways:
  - extreme cells of rectangular data: `range="B1:D10"`,
  - with `cell_rows` for the required rows: `range=cell_rows(1:10)`,
  - with `cell_cols` for the required columns: `range=cell_cols("B:E")`.

Chapter 15

ggplot2

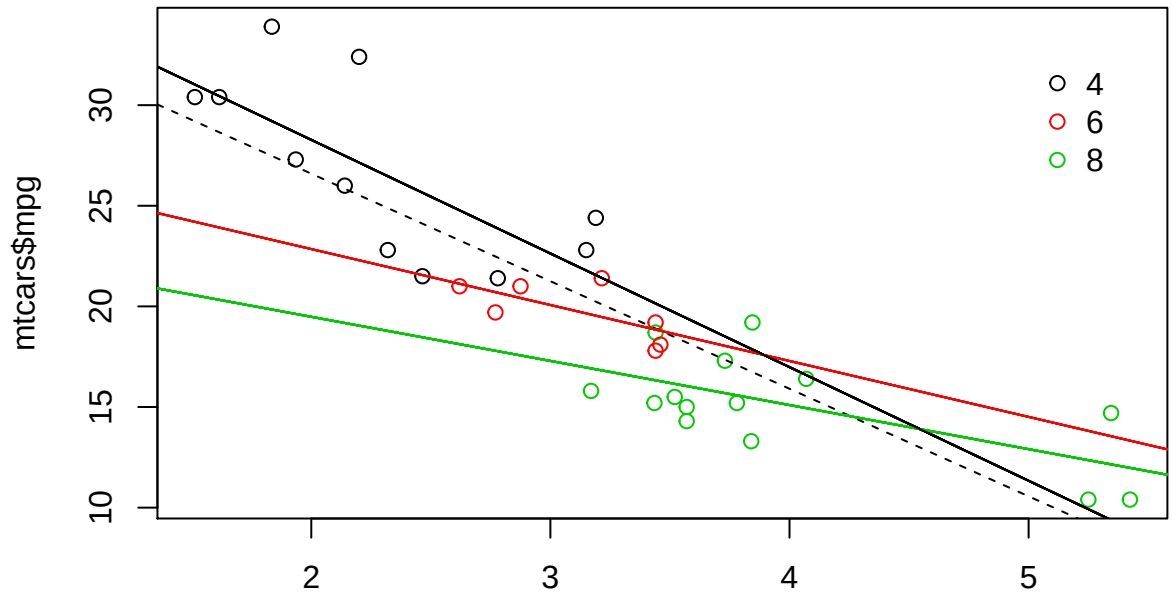
15.1 General purpose

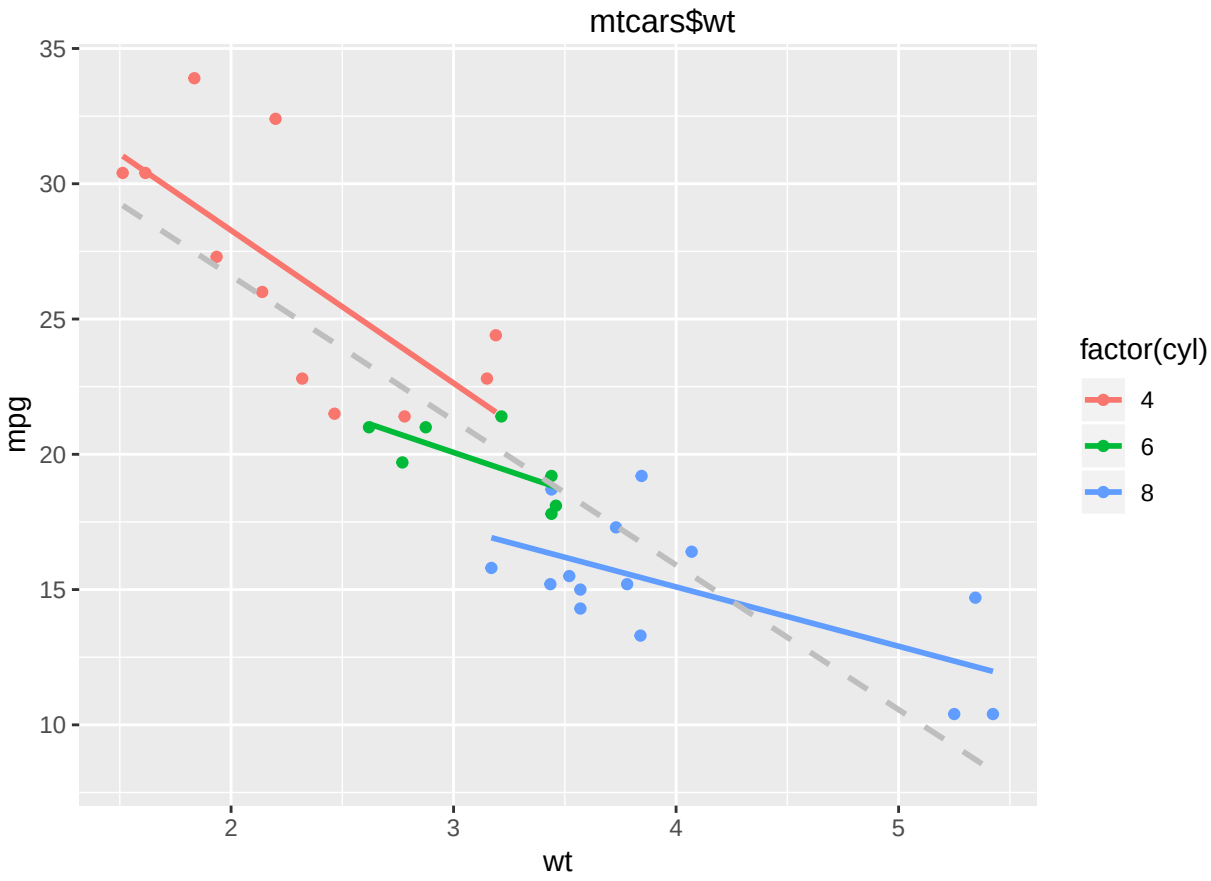
Data visualization is part of the skill set of a data scientist Made to, either - explore (confirm, analyse) - explain (inform, convince) Depends who you communicate to (can be you)

15.2 Comparison with base `plot`

Limitations of base plot - plot does not redraw, i.e., the range will not adapt to new data - plot is drawn as an image, i.e., it's not an object - manual legend - no unified framework for plotting, i.e., need to master other commands for other types of graphs

```
library(ggplot2)
rm(mtcarts)
#R> Warning in rm(mtcarts): object 'mtcarts' not found
data("mtcars")
mtcars$cyl <- as.factor(mtcars$cyl) # treat 'cyl' as a factor

plot(mtcars$wt, mtcars$mpg, col = mtcars$cyl)
abline(lm(mpg ~ wt, data = mtcars), lty = 2)
lapply(mtcars$cyl, function(x) {
  abline(lm(mpg ~ wt, mtcars, subset = (cyl == x)), col = x)
  })
legend(x = 5, y = 33, legend = levels(mtcars$cyl),col = 1:3, pch = 1, bty = "n")

plot1 <- ggplot(mtcars, aes(x = wt, y = mpg, col = factor(cyl))) +
         geom_point() +
         geom_smooth(method=lm , se=FALSE, aes(col=cyl)) +
         geom_smooth(method=lm , se=FALSE, linetype=2, col="grey", aes(group=1))
plot1 # print the object 'plot1'
```


mtcars$mpg
10
15
20
25
30
2
3
4
5
4
6
8


mtcars$wt
mpg
35
30
25
20
15
10
2
3
4
5
wt
factor(cyl)
4
6
8

15.3 Grammar of graphics - Leland Wilkinson

15.3.1 A graphic = layers of grammatical elements

The layers are the adjectives and the nouns Seven grammatical elements, three are essential

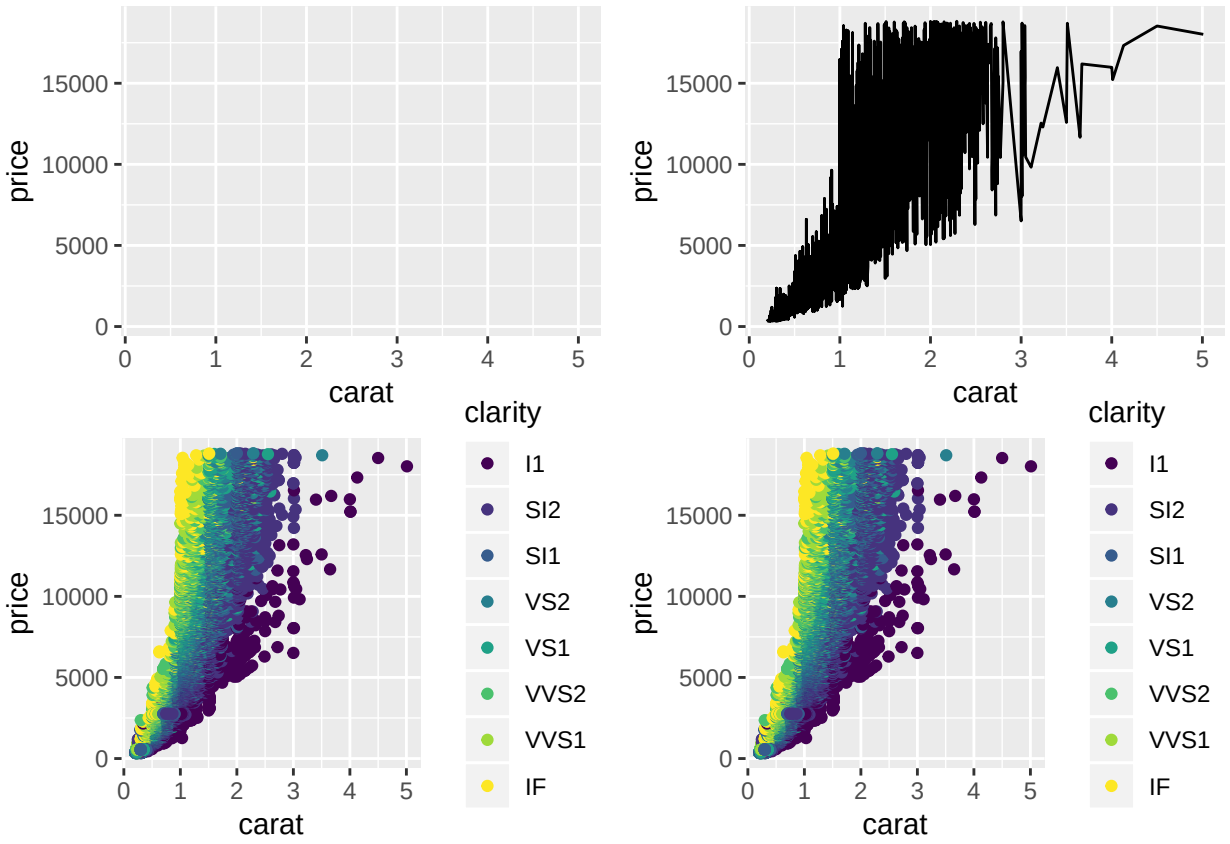
15.3.2 Meaningful plots are built around appropriate aesthetic mappings

The mappings are the rules to assemble the nouns and adjectives

15.4 Understanding the grammar

Building an example

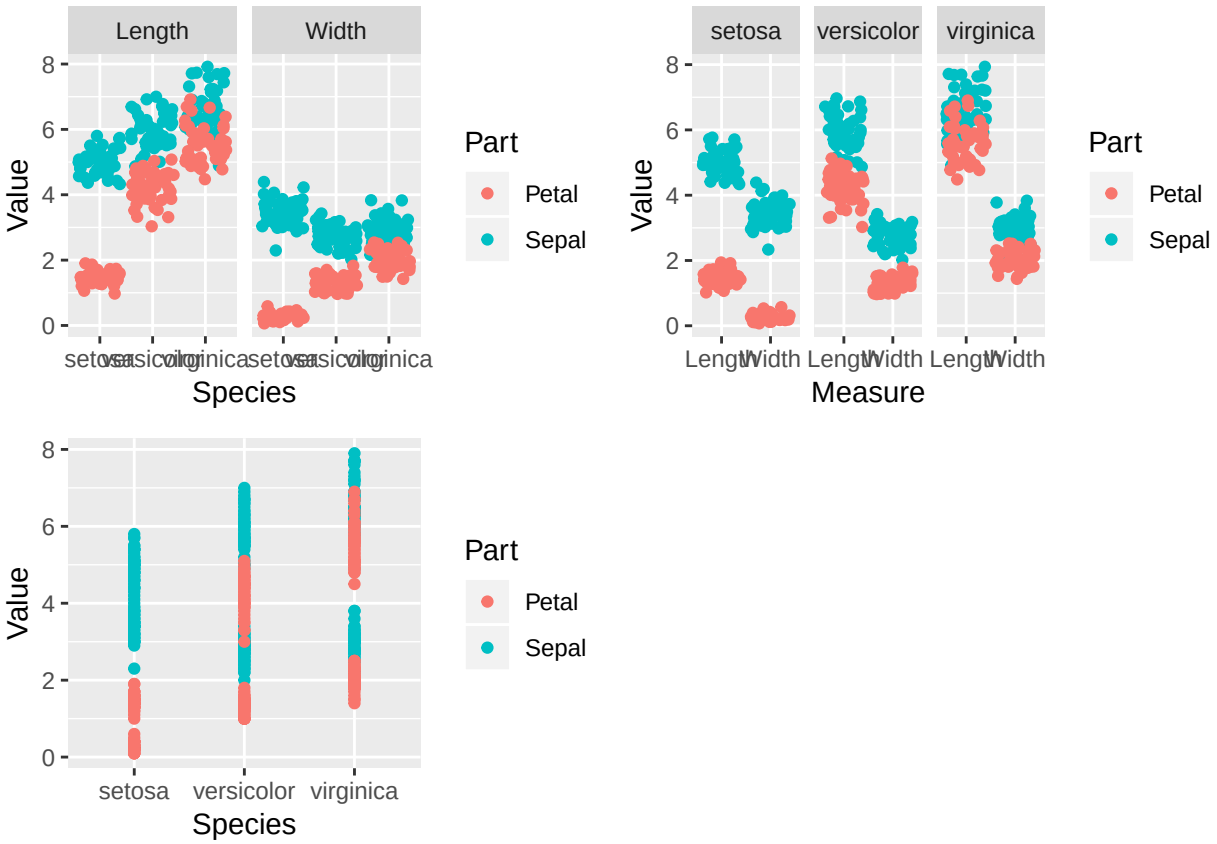
```
p1 <- ggplot(data=diamonds, mapping= aes(x=carat, y= price)) # data and mappings

#p2 <- p1 + geom_point()  # + the general form

p2 <- p1 + geom_line()  # + the general form

p3 <- ggplot(diamonds, aes(x=carat, y= price, col=clarity)) +
      geom_point() # adding a mapping / variable

p3b <- ggplot(diamonds, aes(x=carat, y= price)) +
      geom_point(col="red") # adding a attribute


p4 <- p1 + geom_point(aes(col=clarity)) # or changing some attributes
grid.arrange(p1, p2, p3, p4, ncol=2)
```



15.4.1 Data and proper data format

The data being plotted Includes: variables of interest Tidy data helps make good `ggplot()`s

```
iris.tidy <- iris %>%
  gather(key, Value, -Species) %>%
  separate(key, c("Part", "Measure"), "\\.")

p1 <- ggplot(iris.tidy, aes(x = Species, y = Value, col = Part)) +
geom_jitter() +
facet_grid(. ~ Measure)

p2 <- ggplot(iris.tidy, aes(x = Measure, y = Value, col = Part)) +
geom_jitter() +
facet_grid(. ~ Species)


p3 <- ggplot(iris.tidy, aes(x = Species, y = Value, col = Part)) +
geom_point()

grid.arrange(p1, p2, p3, ncol=2)
```



15.4.2 Aesthetics

The scales onto which we map our data Includes: x-axis, y-axis, colour, fill, size, labels, line width, line type,...

15.4.3 Geometries

The visual elements (shape) used for our data in the plot Includes: point, line, histogram, bar, boxplot,...

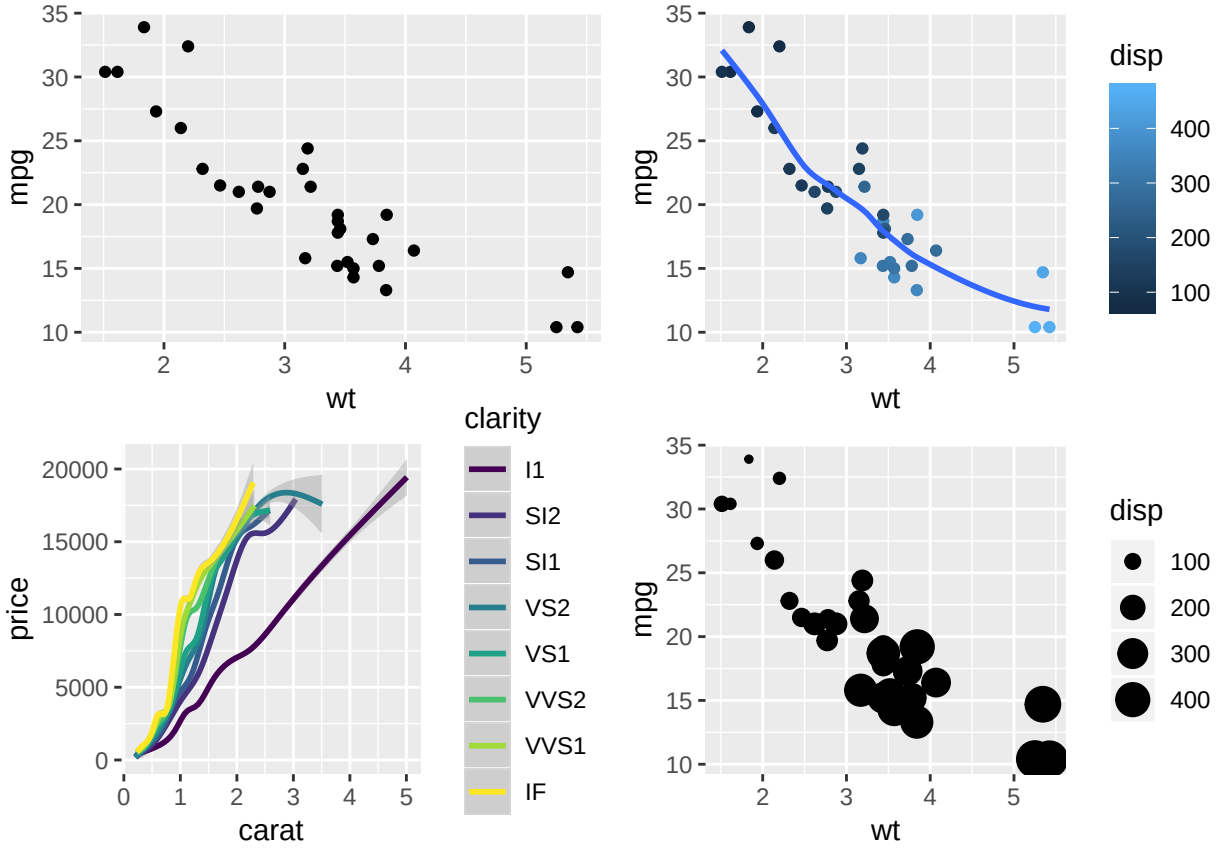
15.4.4 Other grammatical elements

These are - facets - statistics - coordinates - themes

15.4.5 Juggling with `aes`

```
data("mtcars")
p1 <- ggplot(mtcars, aes(x = wt, y = mpg)) +
      geom_point()

# color dependent on 'disp',
# 'disp' is continuous, hence the shades of same colour
p2 <- ggplot(mtcars, aes(x = wt, y = mpg, col = disp)) +
      geom_point()  + geom_smooth(se=FALSE)
```

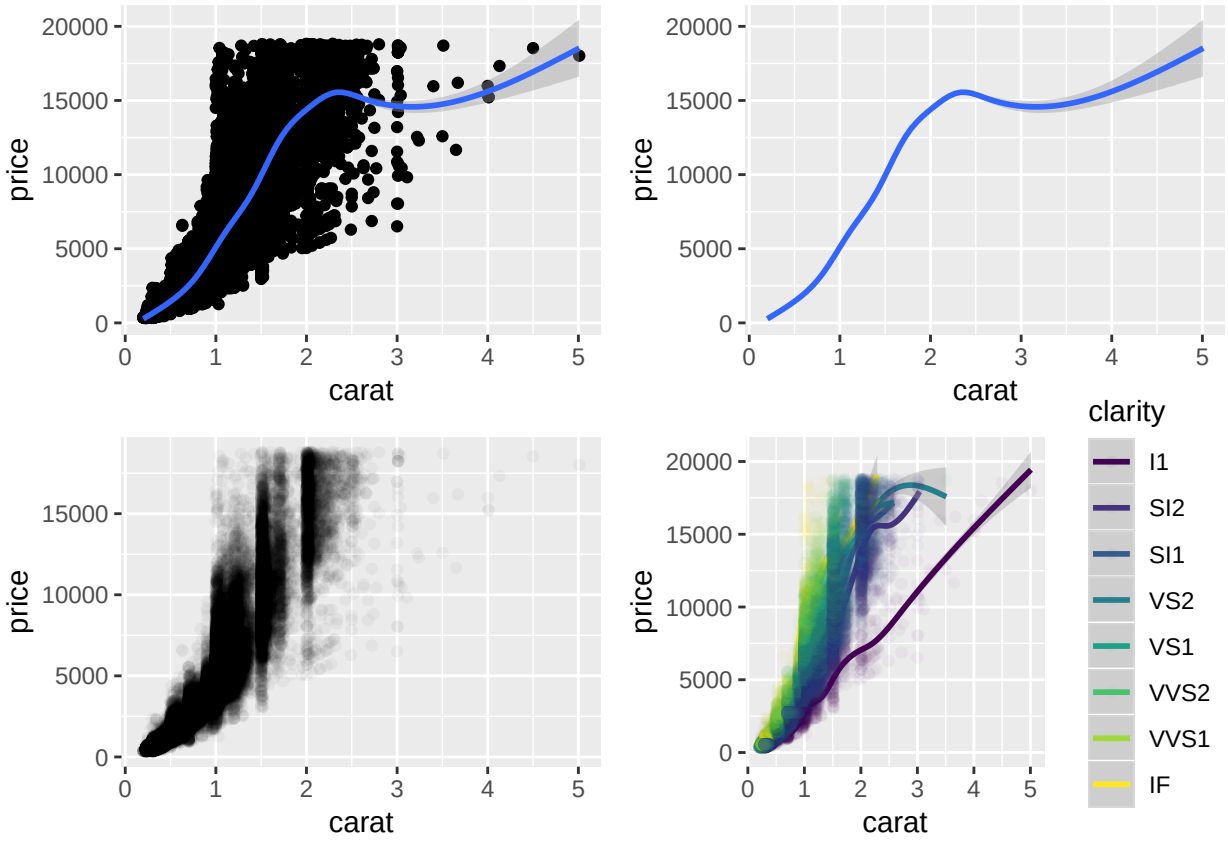
```
# changing the colour depending on the variable 'clarity',
# 'clarity' is a factor, hence the different colours
p3 <- ggplot(diamonds, aes(x = carat, y = price, col=clarity)) +
      geom_smooth()

# size dependent on 'disp'
p4 <- ggplot(mtcars, aes(x = wt, y = mpg, size = disp)) +
      geom_point()

grid.arrange(p1, p2, p3, p4, ncol=2)
#R> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
#R> `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



15.4.6 Juggling with geom

```
# points and smoother layer
p1 <- ggplot(diamonds, aes(x = carat, y = price)) +
      geom_point() +
      geom_smooth(se=TRUE) # se = T by default

# only the smoother layer
p2 <- ggplot(diamonds, aes(x = carat, y = price)) +
      geom_smooth()

# changing the parameters of the point geometry, much more transparent, here
```

```
p3 <- ggplot(diamonds, aes(x = carat, y = price)) +
      geom_point(alpha=.04)

# only the smoother layer and different aesthetics,
# aes recognizes the groups and therefore separates for them
p4 <- ggplot(diamonds, aes(x = carat, y = price, col=clarity)) +
      geom_smooth() + geom_point(alpha=.04)

grid.arrange(p1, p2, p3,p4,  ncol=2)
#R> `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
#R> `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
#R> `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



15.5 Aesthetics

15.5.1 Understanding aesthetics

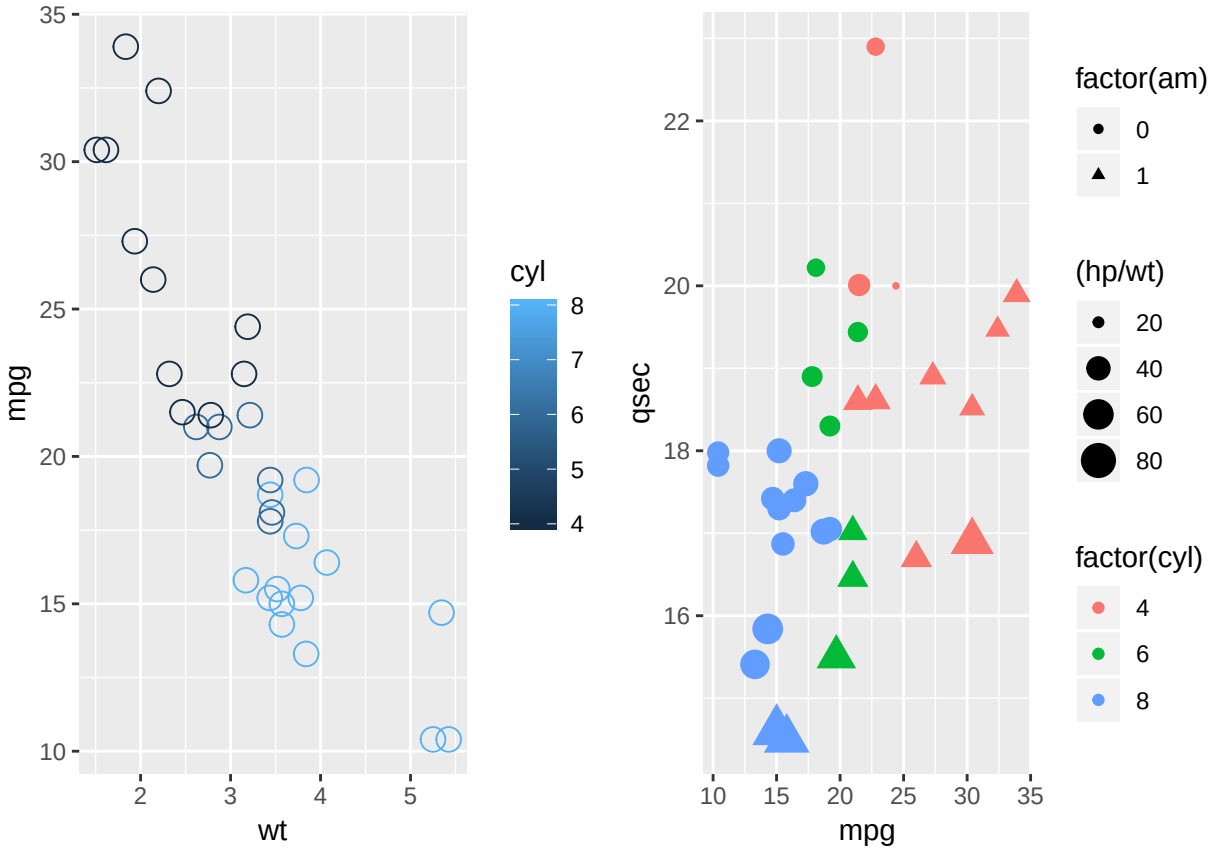
Usually considered as how something looks, as attributes: that's *notcorrect in* `ggplot()` *Aesthetics refers to what a variable is* mappedonto it *Aesthetics is mapping* We want to map as many variables as possible in a plot, that's visible aesthetics Aesthetics/mappings are called in `aes()` while attributes are called in `geom_()`

For instance: - `aes(x = variable1, ...)` means 'variable1' is mapped onto the x-axis - `aes(..., col = variable2)` means that 'variable2' is mapped onto a colour

Simply changing the colour of the dots, is NOT aesthetics - `ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point(col="red")` because there is no mapping `aes()` includes: x-axis, y-axis, colour, fill, size, alpha,

labels, line width, line type,… Some aesthetics are only applicable to categorical variables: e.g., label and shape

```
p1 <- ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
      geom_point(shape = 1, size = 4)
p2 <- ggplot(mtcars, aes(x=mpg, y=qsec,size=(hp/wt), shape=factor(am), col=factor(cyl))) +
      geom_point()

grid.arrange(p1, p2,  ncol=2)
```



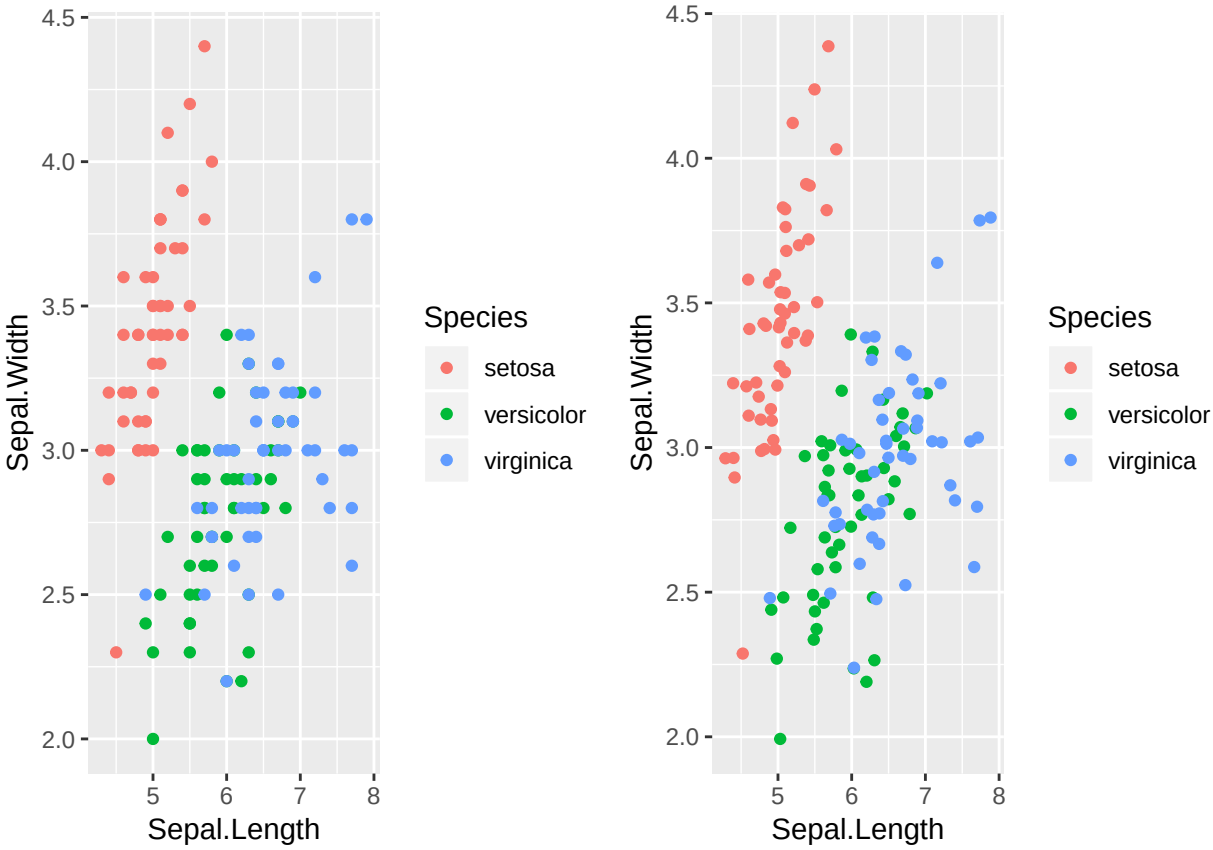
Attributes go to `geom_`

15.5.2 Modifying aesthetics

Modifying aesthetics is modifying the mapping

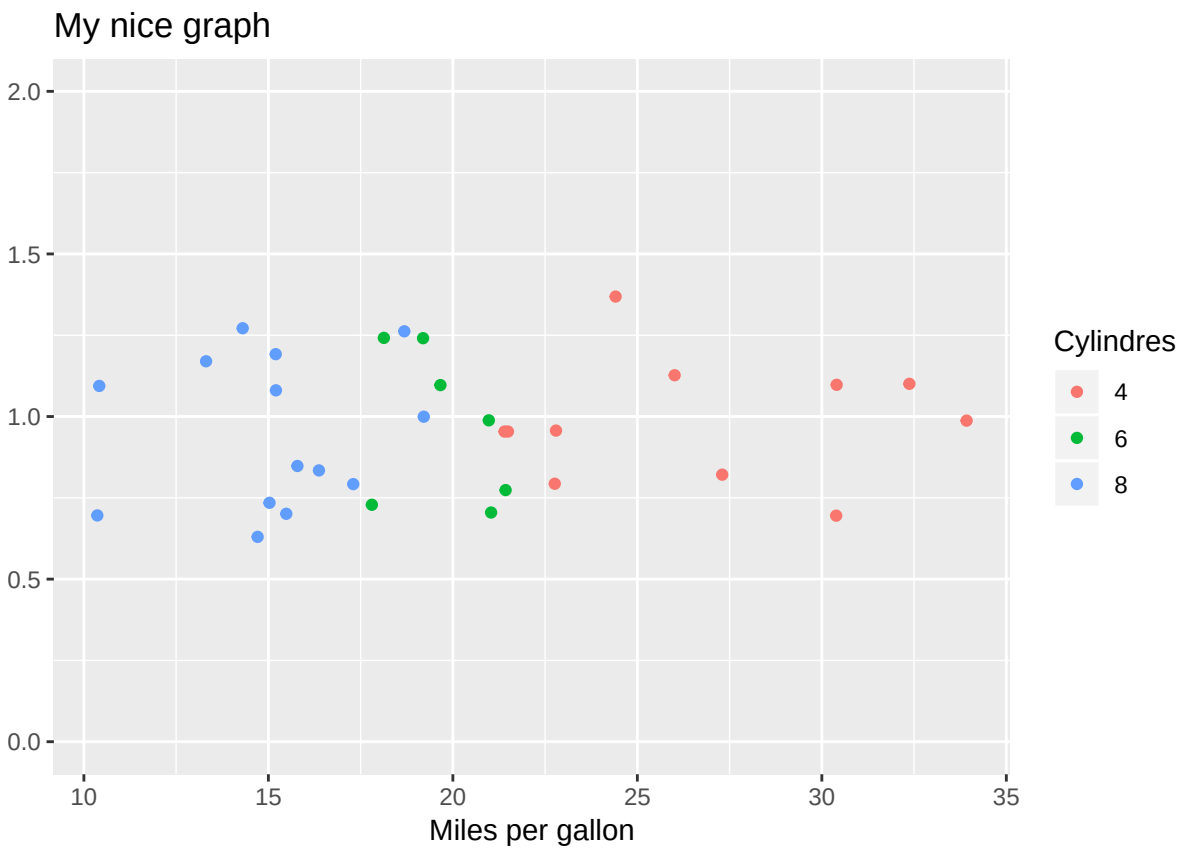
`position` defaults to `identity`, `jitter` adds some random noise to the observations so that they do not overlap

```
p1 <- ggplot(iris, aes(x=Sepal.Length, y=Sepal.Width, col=Species)) +
      geom_point()
p2 <- ggplot(iris, aes(x=Sepal.Length, y=Sepal.Width, col=Species)) +
      geom_point(position="jitter")

grid.arrange(p1, p2,  ncol=2)
```



`scale_..._...(variable1, args)` modifies the scale of the mapping of 'variable1' with several arguments such as `limits`, `breaks`, etc... Sometimes, you need to assign a dummy to the aesthetics, e.g., when you want to plot only one variable

```
mtcars$mydummy <- 1

ggplot(mtcars, aes(x = mpg, y=mydummy, col=factor(cyl))) +
geom_jitter() +
scale_y_continuous(limits=c(-0,2)) +
labs(title="My nice graph", x="Miles per gallon", y="", col="Cylindres")
```



15.6 Geometries

Geometries control how your plot is going to look like: currently, there are more than 37 gemoetries available Common plots are: scatter, bar, line plots Geometries required some aesthetics and can take other aesthetics as optional Geometries can take their own aesthetics (useful to control mappings of each layer) Geometries can take their own data, too!

15.6.1 Scatter plots

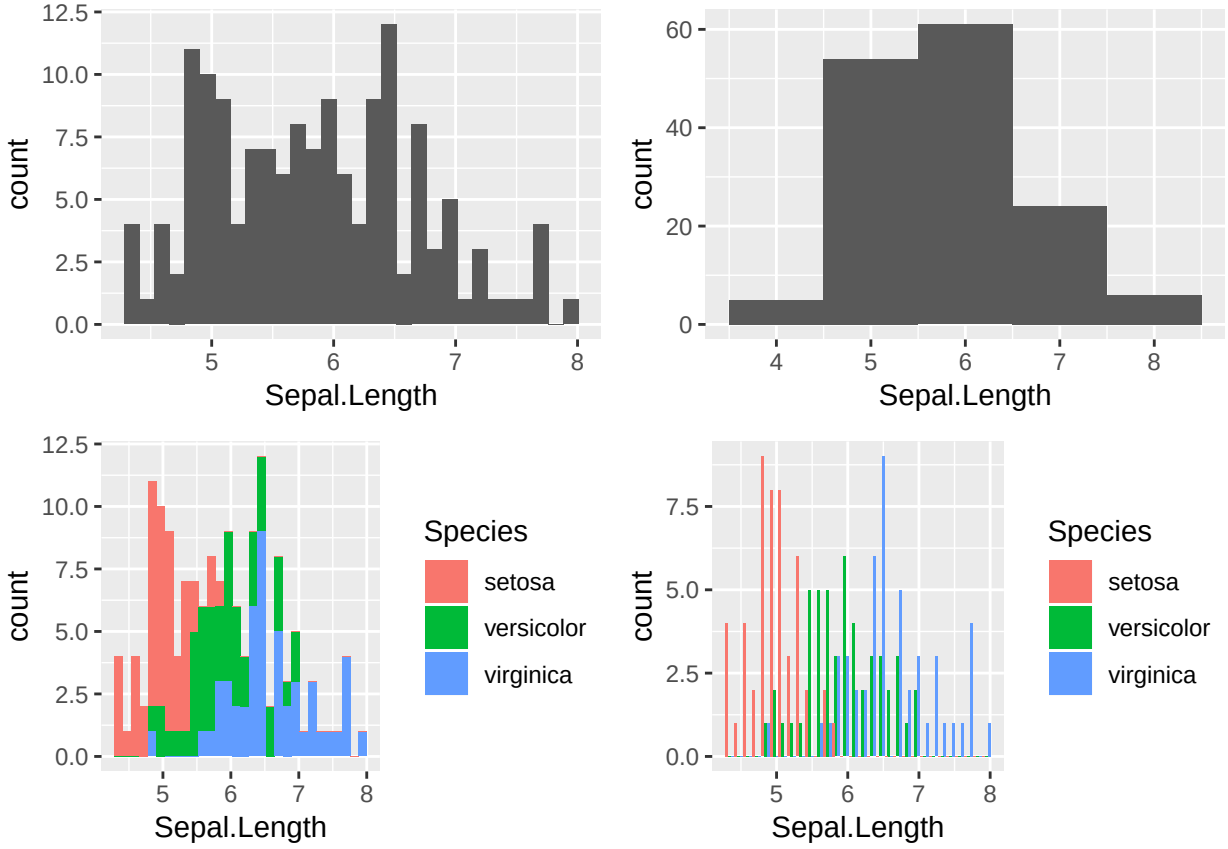
We've seen a lot, already: recall `geom_point()`

15.6.2 Bar plots

The simplest is a histogram

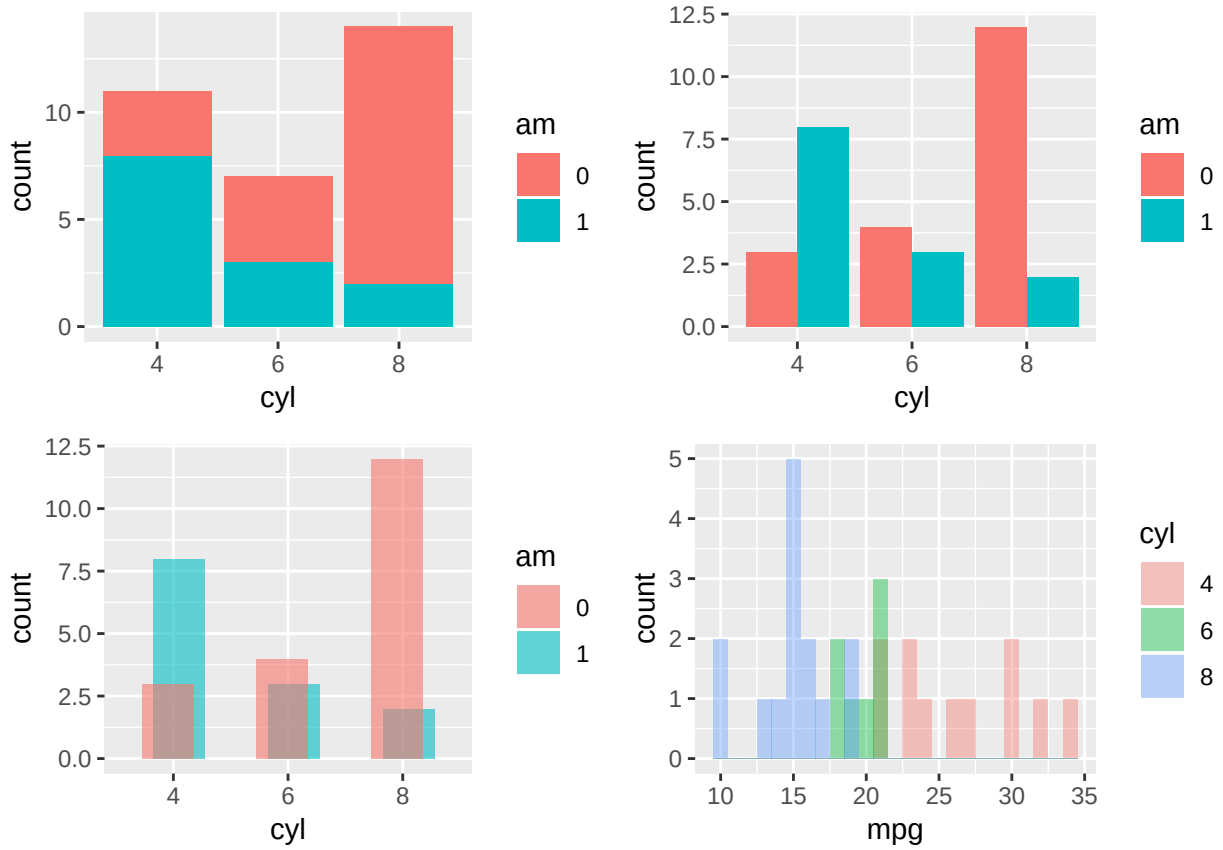
```
p1 <- ggplot(iris, aes(x=Sepal.Length)) + geom_histogram()
p2 <- ggplot(iris, aes(x=Sepal.Length)) +
      geom_histogram(binwidth = 1)
p3 <- ggplot(iris, aes(x=Sepal.Length, fill=Species)) +
      geom_histogram()
p4 <- ggplot(iris, aes(x=Sepal.Length, fill=Species)) +
      geom_histogram(position="dodge") # or "stack", or "fill"

grid.arrange(p1, p2, p3, p4,  ncol=2)
```

```
#R> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
#R> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
#R> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

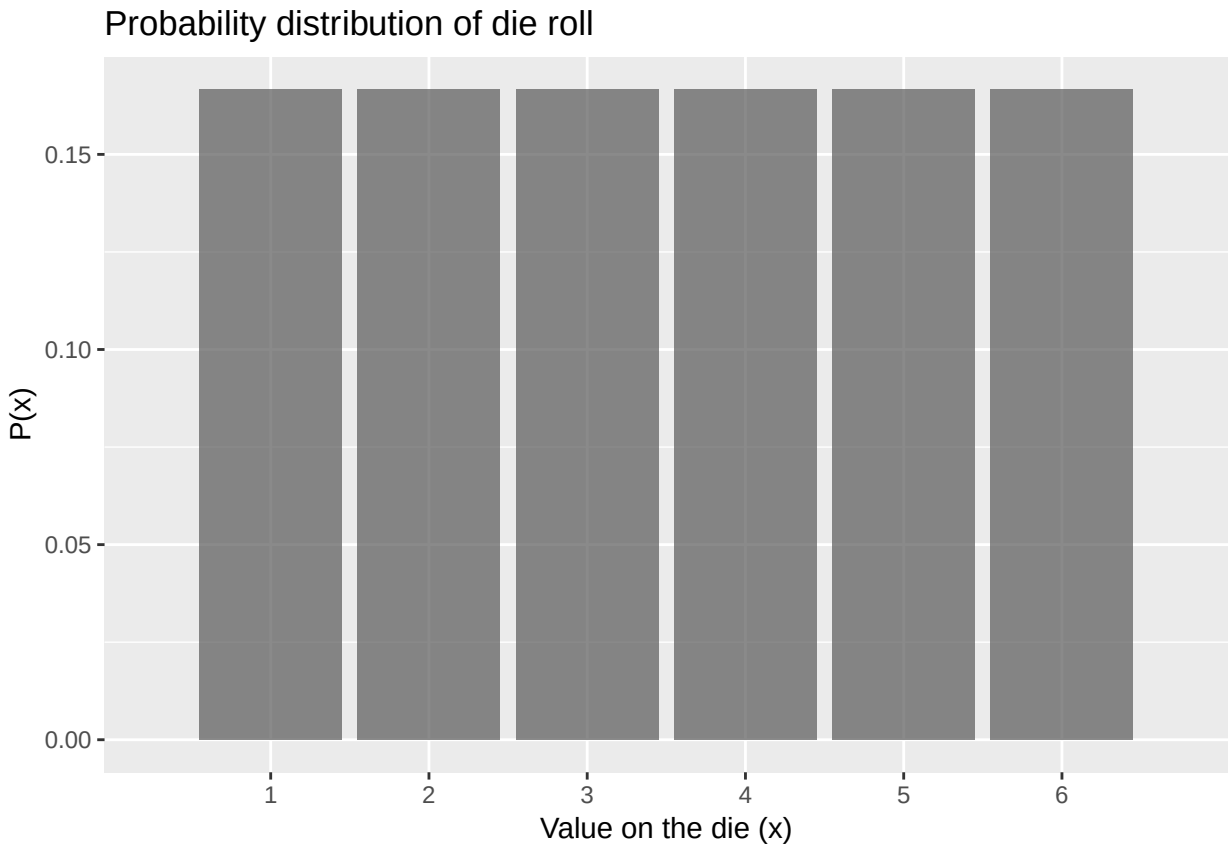


A more general bar plot can be obtained with `geom_bar()`

```
data("mtcars")
mtcars$cyl <- as.factor(mtcars$cyl) # changing 'cyl' to factor, which is actually is
mtcars$am <- as.factor(mtcars$am)
p1 <- ggplot(mtcars, aes(x=cyl, fill=am)) +
      geom_bar(position="stack")
p2 <- ggplot(mtcars, aes(x=cyl, fill=am)) +
      geom_bar(position="dodge") # or "stack", or "fill"

posn_d <- position_dodge(width=.2) # like for jitter
p3 <- ggplot(mtcars, aes(x=cyl, fill=am)) +
      geom_bar(position=posn_d, alpha=.6)

p4 <- ggplot(mtcars, aes(mpg, fill=cyl)) +
      geom_histogram(binwidth = 1, position="identity", alpha=.4)


grid.arrange(p1, p2, p3, p4,  ncol=2)
```



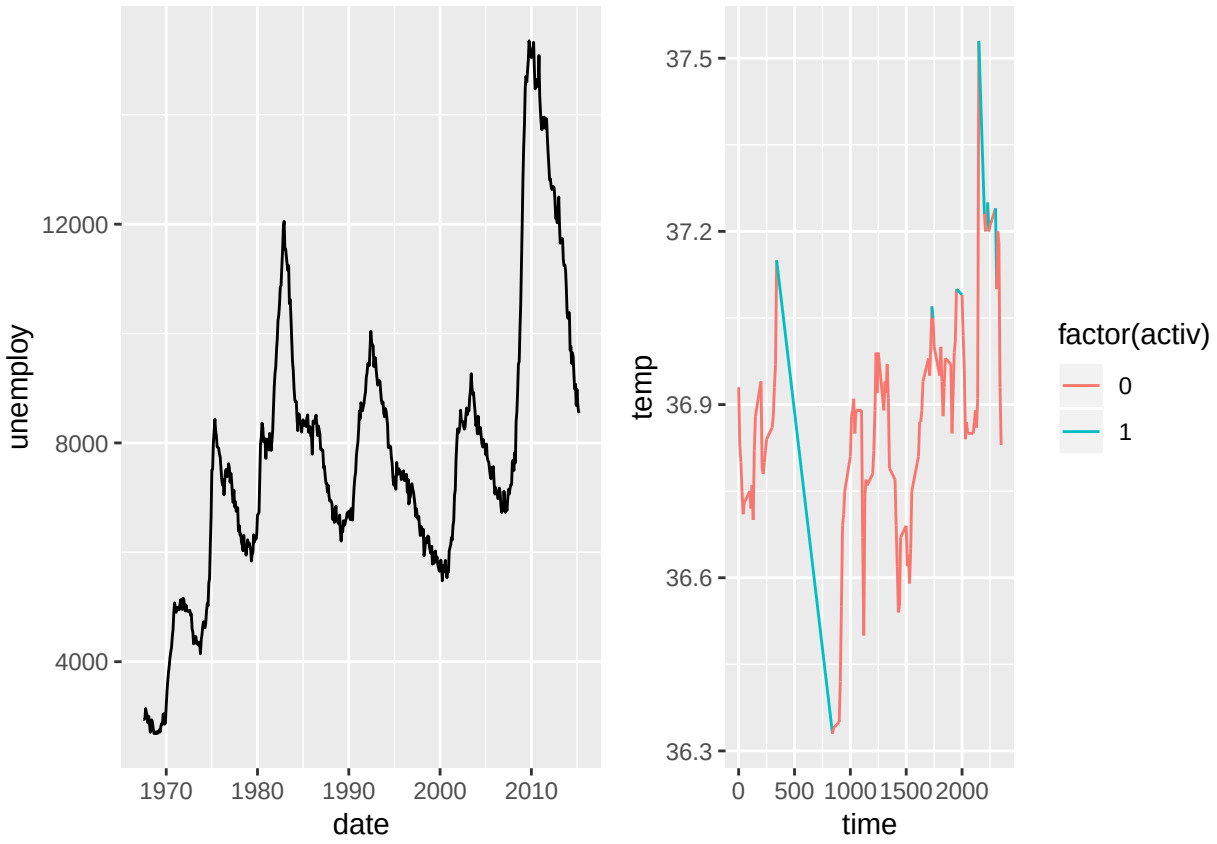

```
data <- data.frame(a=1:6, b=rep(1/6,6))

p1 <- ggplot(data, aes(x=a, y=b)) +
      geom_bar(position="stack", alpha = 0.7, stat = "identity") +
      labs(title = "Probability distribution of die roll", x = "Value on the die (x)", y = "P(x)") +
  scale_x_discrete(limits=1:6, labels=c("1", "2", "3", "4", "5", "6"))

p1
```



15.6.3 Line plots

Particularly suited for line plots over time

```
p1 <- ggplot(economics, aes(x = date, y = unemploy)) +
geom_line()

data("beavers")
p2 <- ggplot(beaver1, aes(x = time, y = temp, col=factor(activ))) +
geom_line(aes(group=1)) # group = 1 indicates you want a single line connecting all the points, you can

grid.arrange(p1, p2, ncol=2)
```


unemploy
12000
8000
4000
1970
1980
1990
2000
2010
date
temp
37.5
37.2
36.9
36.6
36.3
0
500
1000
1500
2000
time
factor(activ)
0
1

Part IV

Conclusion

Chapter 16

Conclusion

We did a great book!

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2018). *rmarkdown: Dynamic Documents for R*. R package version 1.11.

Pakin, S. (2009). The comprehensive latex symbol list.

Wickham, H. (2014). *Advanced R*. Chapman and Hall/CRC.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., and Woo, K. (2018). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.1.0.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. " O'Reilly Media, Inc.".

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2018a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.9.

Xie, Y. (2018b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.21.