# 105.731 AKSTA Statistical Computing

Script

Laura Vana-Gür

Last compiled on 01 June, 2022

## Contents

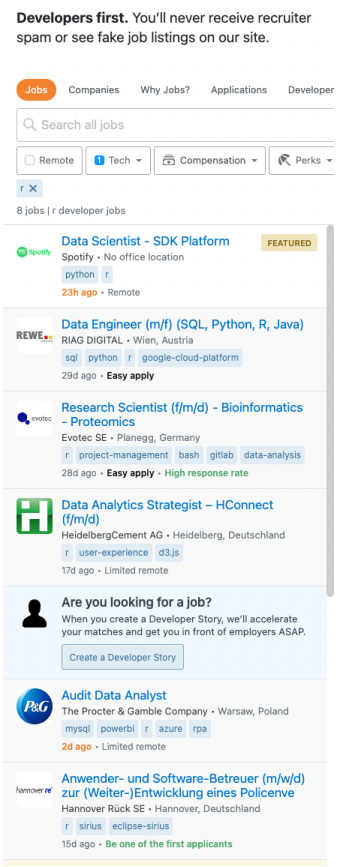
*Disclaimer: This script is work in progress. It is compiled by using materials developed over the years by the instructor as well as a generous collection of materials from colleagues Klaus Nordhausen, Alexandra Posekany, Matthias Templ, Peter Filzmoser.*

# 1 Part 1: Intro to R programming

## 1.1 R & RStudio

### What is R

- R was developed by Ross Ihaka and Robert Gentleman (the "R & R's" of the University of Auckland).
- Ihaka, R., Gentleman, R. (1996): R: A language for data analysis and graphics, Journal of Computational and Graphical Statistics, 5, 299-314.
- R is a environment and language for data manipulation, calculation and graphical display.
- R is a GNU program. This means it is an open source program (as e.g. Linux) and is distributed for free.
- R is used by more than 2 million users worldwide (according to R Consortium).
- R was originally used by the academic community but it is currently also used by companies like Google, Pfizer, Microsoft, Bank of America ...




### R communities

- R has local communities worldwide for users to share ideas and learn.
- R events are organized all over the world bringing its users together:
  - Conferences (e.g. useR!, WhyR?, eRum)
  - R meetups: check out meetup.com

### R and related languages

- R can be seen as an implementation or dialect of the S language, which was developed at the AT & T Bell Laboratories by Rick Becker, John Chambers and Allan Wilks.
- The commercial version of S is S-Plus.
- Most programs written in S run unaltered in R, however there are differences.
- Code written in C, C++ or FORTRAN can be run by R too. This is especially useful for computationally-intensive tasks.

### How to get R

- R is available for most operating systems, as e.g. for Unix, Windows, Mac and Linux.
- R can be downloaded from the R homepage http://www.r-project.org
- The R homepage contains besides the download links also information about the R Project and the R Foundation, as well as a documentation section and links to projects related to R.
- R is available as 32-bit and 64-bit
- R comes normally with 14 base packages and 15 recommended packages

### CRAN

- CRAN stands for Comprehensive R Archive Network
- CRAN is a server network that hosts the basic distribution and R add-on packages
- Central server: http://cran.r-project.org
  - New R versions are usually released every few weeks.
  - current R version: 4.1.2 (Bird Hippie, released on 2021-11-01) as of 2022-06-01
- The R version used in the course is 4.1.2 (as of summer semester 2022).

**R extension packages** R can be easily extended with more packages, most of them can be downloaded from CRAN too. Installation and updating of those packages is however also possible with using R itself

In R, packages can be directly installed by typing in the console `install.packages()`. Once installed (locally), each time functions from the packages should be employed, the packages get loaded and attached to the workspace by `library()`.

Currently 19020 are available on CRAN).

- Packages for the analysis and comprehension of genomic data can be downloaded from the Bioconductor pages (http://www.bioconductor.org).
- but R packages are available form many other sources like R-forge, Github, ...

### Other distributions of R

- As R is open source and published under a GNU license one can make also a **own** version of R and distribute it.
- For example Microsoft has **Microsoft R Open** https://mran.microsoft.com/open
- But there are many others too. We use however here the standard R version from CRAN.

**What R offers** Among other things R offers:

- an effective data handling and storage facility.
- a suite of operators for calculations on arrays and matrices (R is a vector based language).
- a large, coherent, integrated collection of tools for data analysis.
- graphical facilities for data analysis and display.
- powerful tools for communicating results. R packages make it easy to produce html or pdf reports, or create interactive websites.
- a well-developed, simple and effective programming language.

Therefore is not only a plain statistics software package, but it can be used as one. Most of the standard statistics and a lot of the latest methodology is available for R.

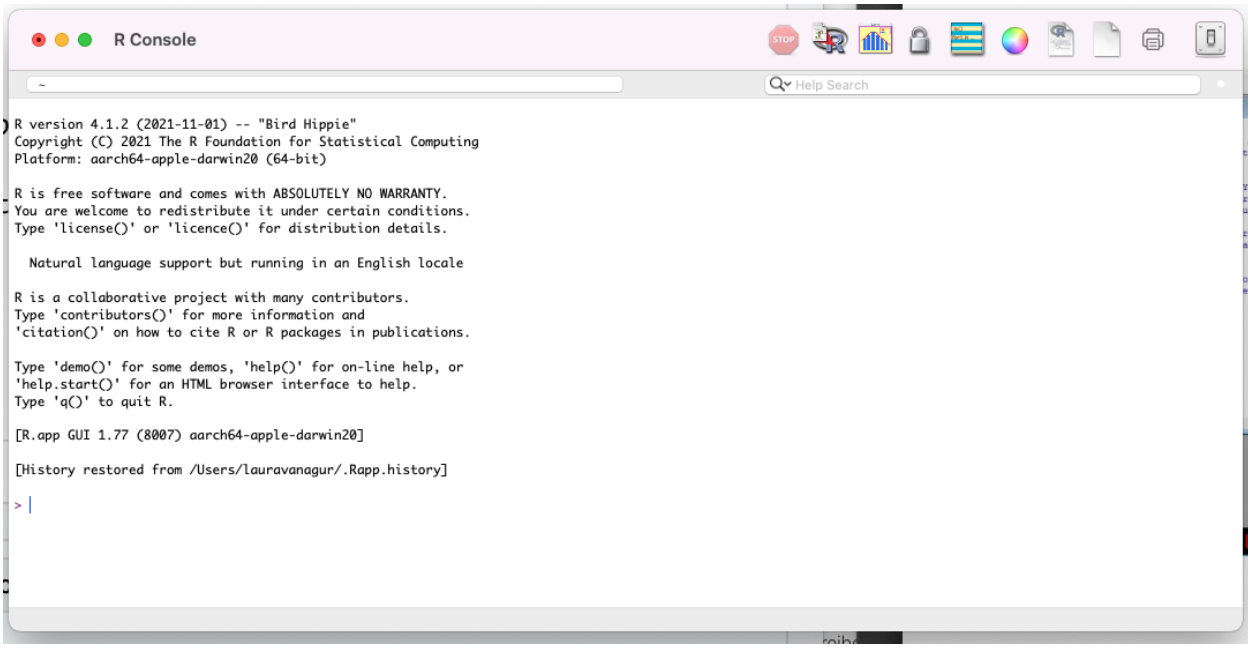
**More on R** R, at its heart, is a **functional language**.

- It lends itself to a style of problem solving centered on functions.
- It has first-class functions, i.e., functions that behave like any other data structure.
- It does not require functions to be pure (output only depends on inputs and no side-effects) but they are recommended.

We will discuss more on this once we come to functions.

Base R provides three **object oriented programming** (OOP) systems: S3, S4, and reference classes (RC), but there are more CRAN packages which provide other OOP systems (**R6**, **R.oo** etc). In general, a reason to use OOP is polymorphism, which means that it is possible to use the same function on different types of input to obtain different results.

**R screenshot**



**R console**

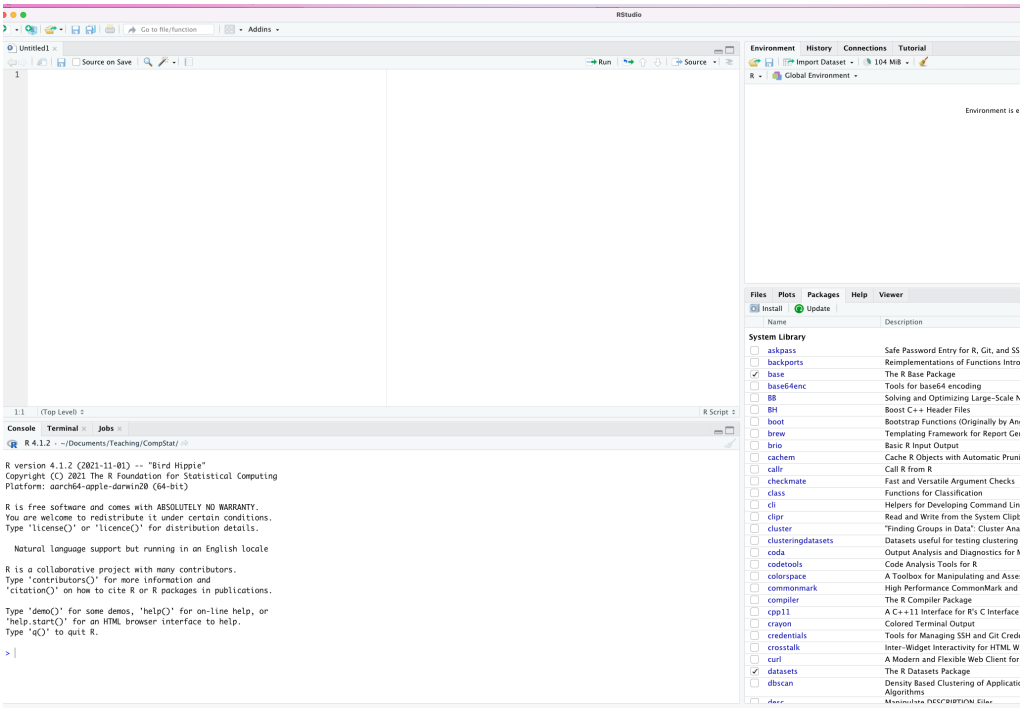
- R by default has no graphical interface and the so called Console has to be used instead.
- The Console or Command Line Window is the window of R in which one writes the commands and in which the (non-graphic) output will be shown.
- Commands can be entered after the prompt (>).
- In one row one normally types one command (enter submits the command). If one wants to put more commands in one row the commands have to be separated by a ";"
- When a command line stars with a "+" instead of ">"it means that the last submitted command was not completed and one should finish it now.
- All submitted commands of a session can be recalled with the up and down arrows ↑↓.

**R as a pocket calculator** In the console we can for example do basic calculations

```
> 7 + 11
[1] 18
> 57 - 12
[1] 45
> 12 / 3
[1] 4
> 5 * 4
[1] 20
> 2 ^ 4
[1] 16
> sin(4)
[1] -0.7568025
```

**R editors and IDEs**

- Using the R Console can be quite cumbersome, especially for larger projects. An alternative to the Command Line Window is the usage of **editors** or **IDEs** (integrated development environments).
- Editors are stand-alone applications that can be connected to an installed R version and are used for editing R source code. The commands are typed and via the menu or key combinations the commands are submitted. The user has here usually the choice to submit one command at the time or several commands at once.
- IDEs integrate various development tools (editors, compilers, debuggers, etc.) into a single program - the user does not have to worry about connecting the individual components
- R has only a very basic editor included which can be started form the menu "File" New script.
- Better editors are **EMACS** together with **ESS**, **Tinn-R** or **WinEdt** together with **R-WinEdt**. These editors offer syntax highlighting and sometimes also templates for certain R structures.
- The most popular IDE is currently probably **RStudio**.




#### 1.1.0.0.1 RStudio screenshot

The default view is RStudio is the following:

- The main window in RStudio contains five parts: one Menu and four Windows (“Panes”)
- From the drop-down menu RStudio and R can be controlled.
- Pane 1 (top left) - Files and Data: Editing R-Code and view of data sets
- Pane 2 (top right) - Workspace and History:
  - Workspace lists all objects in the workspace
  - History shows the complete code that was typed or executed in the console.
- Pane 3 (bottom right) - Files, Plots, Packages, Help:
  - Files, to manage files
  - Plots, to visualise and export graphics
  - Packages, to manage extension packages
  - Help, to access information and help pages for R functions and datasets
- Pane 4 (bottom left) - Console: Execution of R-Code
- This pane layout (and the pane contents) can be adapted using the options menu.

**R: a short statistical example** A more sophisticated example will demonstrate some features of R:

```
> options(digits = 4)
> # setting random seed to get a reproducible example
> set.seed(1)
> # creating data
> eps <- rnorm(100, 0, 0.5)
> eps[1:5]
```

```
[1] -0.31323  0.09182 -0.41781  0.79764  0.16475
> group <- factor(rep(1:3, c(30, 40, 30)),
+   labels = c("group 1", "group 2", "group 3"))
> x <- runif(100, 20, 30)
> y <- 3 * x + 4 * as.numeric(group) + eps
```

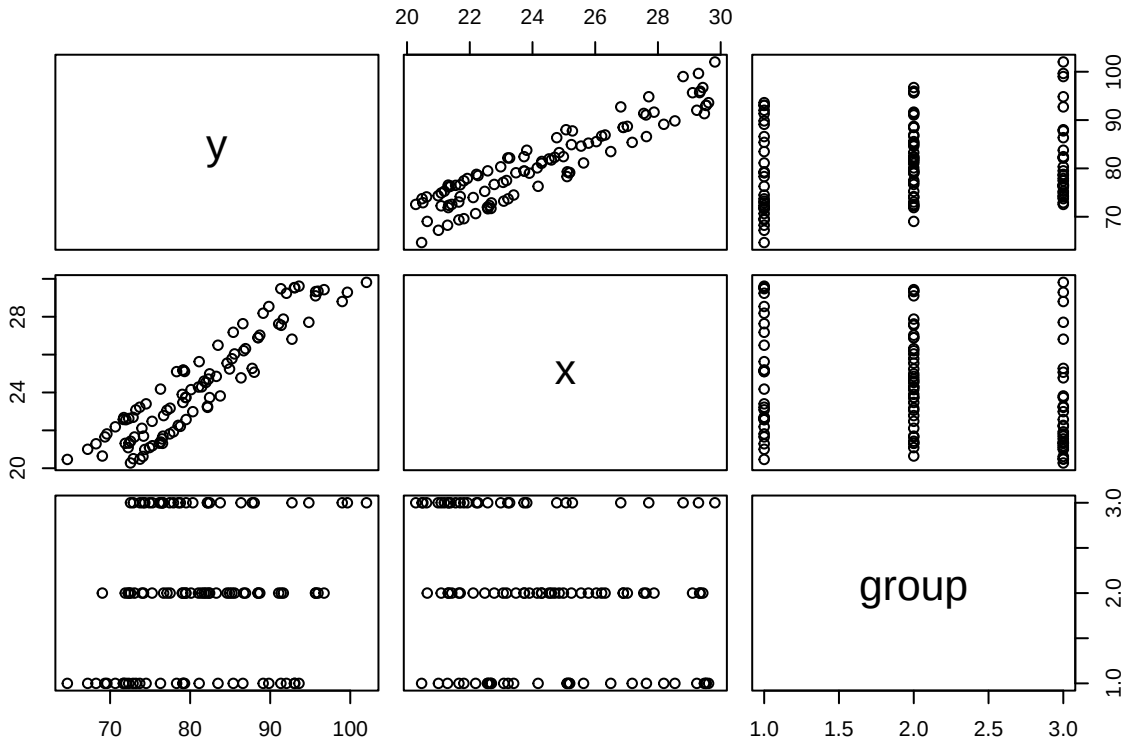
```
> # putting the variables into a dataset as could be
> # observed in reality
> data.ex <- data.frame(y = y, x = x, group = group)
> # looking at the data
> str(data.ex)
'data.frame':   100 obs. of  3 variables:
 $ y    : num  71.7 70.7 79.1 72.9 69.6 ...
 $ x    : num  22.7 22.2 25.2 22.7 21.8 ...
 $ group: Factor w/ 3 levels "group 1","group 2",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Summary of the data can be obtained:

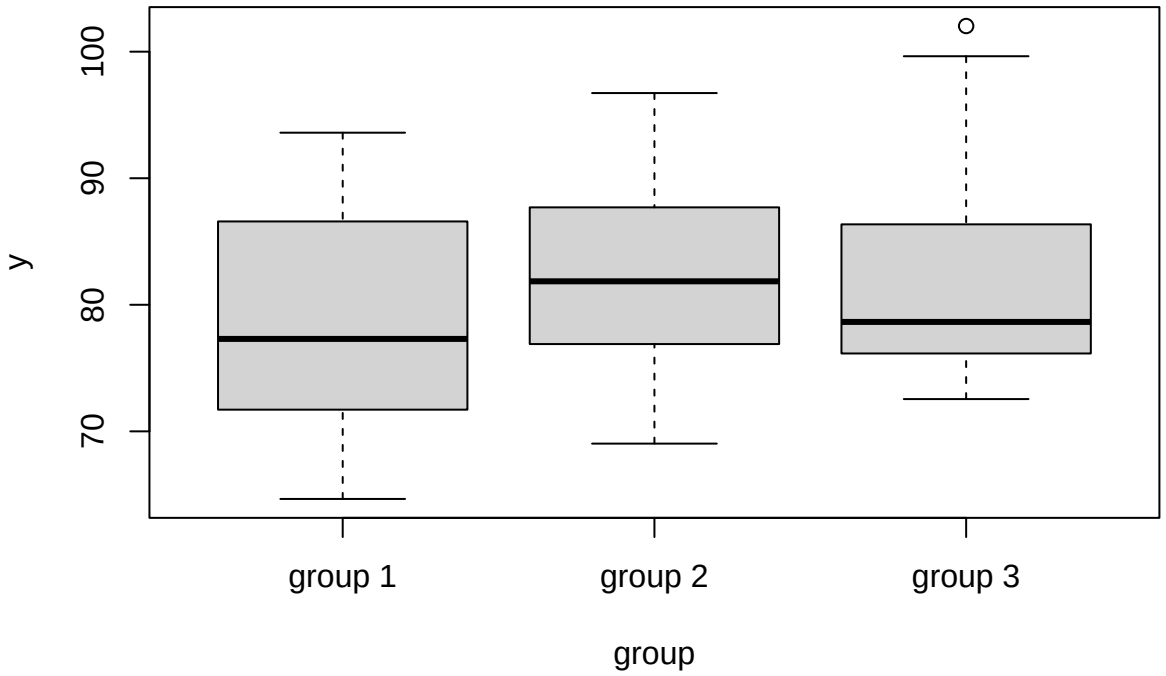
```
> summary(data.ex)
       y                x           group
 Min.   : 64.7   Min.   :20.3   group 1:30
 1st Qu.: 74.3   1st Qu.:21.9   group 2:40
 Median : 79.5   Median :23.8   group 3:30
 Mean   : 81.1   Mean   :24.4
 3rd Qu.: 86.8   3rd Qu.:26.4
 Max.   :102.0   Max.   :29.8
```

Now some plots:

```
> plot(data.ex) # plot 1
```



```
> plot(y ~ group) # plot 2
```




Build a linear model:

```
> # fitting a linear model and looking at it
> lm.fit <- lm(y ~ x + group)
> lm.fit

Call:
lm(formula = y ~ x + group)

Coefficients:
 (Intercept)             x  groupgroup 2  groupgroup 3
        3.77          3.01          4.07          7.98
```

```
> # more detailed output
> summary(lm.fit)

Call:
lm(formula = y ~ x + group)

Residuals:
    Min      1Q  Median      3Q     Max
-1.1988 -0.2797  0.0198  0.2792  1.0893

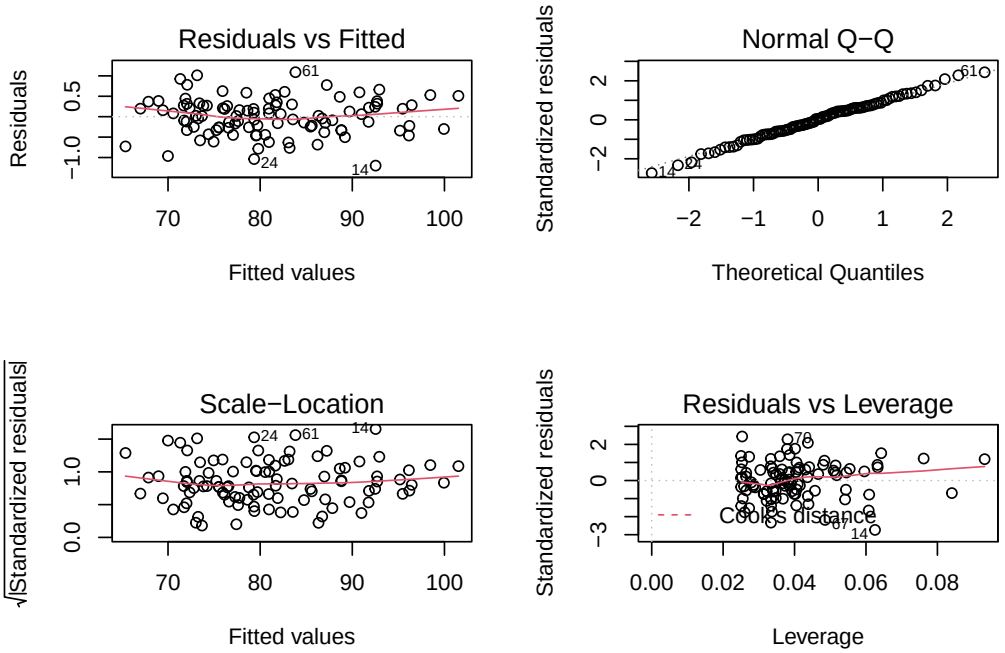
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    3.7682     0.4288    8.79    6e-14 ***
x              3.0110     0.0169  178.19   <2e-16 ***
groupgroup 2   4.0666     0.1094   37.18   <2e-16 ***
groupgroup 3   7.9754     0.1201   66.38   <2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.453 on 96 degrees of freedom
Multiple R-squared:  0.997, Adjusted R-squared:  0.997
F-statistic: 1.1e+04 on 3 and 96 DF,  p-value: <2e-16
```

Check some diagnostic plots:

```
> # diagnostic plots
> par(mfrow = c(2, 2)); plot(lm.fit)
```




What can we notice from the example?

1. Doing statistics with R is mainly calling ready available **functions**
2. The main assignment operator in R is "<-"
3. Assigning results or values produces no output
4. Results can be seen by calling the object or not assigning a function
5. R is **object oriented**, this means depending on the type of input functions perform different tasks (E.g. the functions `plot()` or `summary`)
6. Text or commands after "#" are ignored by R (can be used for commenting code)

## 1.2 Help for using R

On first sight R looks a bit difficult but already with a few basic commands statistical analyses can be done. To learn about those commands several sources are available:

- Online manuals and tutorials
- Books
- R's inbuild help systems and example and demo features
- R has its own journal, the R Journal (earlier called R Newsletter) where different topics are explained.

  - For example how to handle date formats in R would be in Newsletter 4(1) 2004, pp. 29-32.
- Add on packages are often described in journal articles usually published in the free online journal Journal of Statistical Software.

### Manuals and tutorials for R

- On the R homepage one can find the official manuals under `Documentation -> Manuals`. Especially the "An Introduction to R" Manual is recommended.
- "Unofficial" tutorials and manuals, also in other languages than English can be found also on the R homepage under Documentation -> Other or on CRAN under Documentation -> Contributed. Very useful from here is the R reference card by Tom Short.

**R tutorials for SAS, Stata or SPSS users** A lot of new R users are familiar with SAS, Stata and/or SPSS and therefore special charts for an overview how to do things they used to do in SAS, Stata or SPSS can be done in R and a extended manual for an easier move to R are available.

The following references might then be helpful:

- http://r4stats.com
- Muenchen, R.A. (2008): R for SAS and SPSS Users
- Muenchen, R.A. and Hilbe, J. (2010): R for Stata Users

### Help within R

- There are three type of help types available in R. They can be accessed via the menu or the command line. Here only the command line versions will be explained
- Using an internet browser:
  `> help.start()` will evoke an internet browser with links to manuals, FAQs the help pages off all functions sorted by packages together with an search engine.
- The `help` command:
  `> help(command)` will show command. A shorter version that does the same is `> ?command`. For a few special commands the help works only when the command is quoted, e.g. `> help("if")`
- The `help.search` command
  `> help.search("keyword")` one can search all titles and aliases of the help files for keywords. A shorter version that does the same is `> ??keyword`. This is however not a full text search.

There are also three other functions useful to learn about functions.

- `apropos`: `apropos("string")` searches all functions that have the string in their function name
- `demo`: The `demo` function runs some available scripts to demonstrate their usage. To see which topics have a demo script submit `> demo()`
- `example`: `> example(topic)` runs all example codes from the help files that belong to the topic `topic` or use the function `topic`.

Also in case you remember the beginning of a function or are just lazy - R has also an auto completion feature. If you start typing a command and hit `tab` R will complete the command if there are no alternatives or will you give all the alternatives.

**Mailing lists for R**

- R as one of the main statistical software programs has several mailing lists. There are general mailing lists or lists of special interest groups like a list for mixed effects models or robust statistics (for details see the R homepage).
- The general mailing list is **R-help** where questions are normally answered pretty quickly. But make sure to read the posting guide before you ask something yourself! The R-help mails are also archived and can be searched.
- Using on the R homepage the **search**-link will lead to more information on search resources.
- And last but not least, there is also Stack Overflow.

## 1.3 Reproducibility

**Reproducibility** is one of the key concepts of scientific research. It has several meanings:

- the same researcher or another researcher should be able to replicate an entire experiment or study and yield results with a high agreement with the previous results.
- given the data from an experiment also the analysis should be reproducible and anyone analyzing it should come to the same conclusion.

The **reproducibility crisis** refers to a methodological crisis in science, in which scientists have found in recent years that the results of many scientific experiments are difficult or impossible to replicate. Even by the same researcher.

Sciences especially hit by this crisis are psychology and medicine.

For example in the the so-called **reproducibility project** collaborated 270 researchers from around the world to replicate 100 empirical studies from three top Psychology journals. Fewer than half of the attempted replications were successful.

In a Nature paper Begley, C. G. & Ellis, L. M. (2012) showed that from 53 medical papers on cancer research 47 were not reproducible.

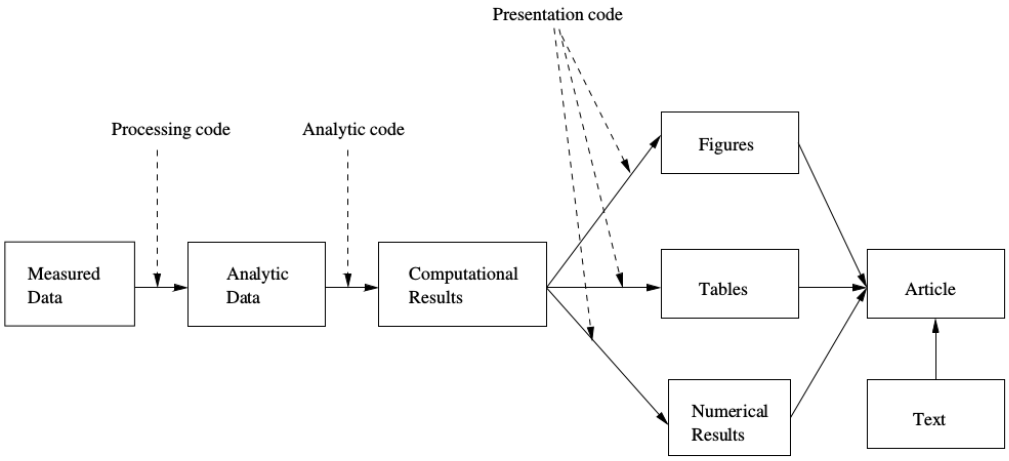
As a reaction to the reproducibility crisis the concept of **reproducible research** was introduced. The aim there is that when research is published it also releases:

- the original data
- detailed description of the computational environment
- the complete code of the analysis

Hence everyone can reanalyze the data step for step.

The goal is even that the data and the code are **part** of the published report. And it is nowadays easy to **include** R into documents and whenever the document is compiled the R code is executed and the computed results included in the report.

### 1.3.1 Workflow of reproducible research




The workflow of reproducible research has as main steps:

1. Collect the raw data (often coming from many sources)
2. Clean and combine the data in to have it in a format needed for the analysis
3. Statistical analysis
4. Present the results
    - In the form of books, articles, slides (preferable format is hence pdf)
    - In the web (preferable format is hence html)

During a project and all its steps it is assumed that everything is well documented and many files belong to the project.

In general it is preferred that the files are all **text** files and human readable.

- data files should end for example as **.txt** or **.csv** and **not** for example .xlsx or .sav.
- Analysis files should for example script files written in an editor and for R users end with **.R**.
- Files making the presentation files should be **.tex** or **.md** (**.Rmd*').

Using human readable text formats makes it the least likely that it depends on the versions of the software with which it was created and any user, even those not knowing for example R or LaTeX will be able to open the files and get an idea about the content.

**Costs and benefits of reproducible research** For a person producing reproducible research this comes at a cost:

- The knowledge how to do reproducible research needs to be acquired.
- Additional software needs to be learned.

It is however assumed that in the long run there are many benefits:

- Better working habits. Individual steps are often better planned and code is more through and commented compared to the case when it is assume that **no one is looking**.

- It helps for teamwork as it is easier for each collaborator to see what is going on.
- Changes are easier as your workflow is dynamic and changes will directly be considered in later parts. For example when you get more data or so on.
- Higher research impact - it is usually considered that reproducible research gets more attention as it is considered "trustworthier."

**Commenting** All files maybe except the data files should be well documented. Comments should be understandable for another user and not only for the author.

- In R everything is a comment that appears after a hash #.
- In LaTeX everything after the percent sign % is treated as a comments.
- In html /markdown comments are placed inside `<!- ->`.

Data files could have their own files "commenting" them.

**Literate programming** A key concept of reproducible research is **literate programming** which means that a source file can be "read" by a computer **woven** (weave) or **knitted** (knit) together with a formatted presentation document.

In this course we will focus on how R code can be combined with other formatting languages to produce dynamically pdf or html output.

### 1.3.2 Sweave and R Markdown

**Sweave** and **R Markdown** are two easy ways for reproducible research with R. In both document types embedded R code chunks are executed and the results included in the document.

- **Sweave** combines LaTeX and R and yields a pdf.
- **R Markdown** combines **markdown** and R and can yield for example pdf, html and word documents.

Note that R Markdown is heavily using **knitr** which is another popular way for doing this but not specifically discussed in this course.

**Sweave** **Sweave** is part of every R installation and details can be obtained using

```
help("Sweave", package="utils")
```

To use Sweave, the user needs to know R and LaTeX. The basic idea is that in a Sweave file (ending **.Rnw**) R code chunks are embedded in the tex paper using the so-called noweb style. Then first the file is run in R using

```
Sweave("file.Rnw")
```

which executes all the R code chunks and replaces the code with the output and creates the figures, yielding the corresponding **file.tex** which can then be compiled into a pdf.

An R code chunk in a Sweave file has the form

```
<<label=Name, options>>=

Some R code

@
```

Nothing else should be in the lines that start and end the chunk.

To include within LaTeX short R output one can use `\Sexpr{R_code}`.

The most important code chunk options are:

- `eval = TRUE`: Should the code chunk be evaluated?
- `echo = TRUE`: Should the code be included in the output or only the results?
- `results = hide`: the chunk is evaluated but any output suppressed.
- `fig = TRUE`: If the chunk should make a figure and include it. By default it will make a eps and pdf figure.
- `eps = FALSE`: If eps figure should be made.
- `width = 6.5`: width of the figure in inch.
- `height = 8.5`: height of the figure in inch.

To extract all the code from code chunks (not from `\Sexpr`) one can use in R the function

```
Stangle("file.Rnw")
```

This will produce the file **file.R** which can be for example sourced.

**R Markdown** While Sweave is mainly for combining R and LaTeX (but HTML is for example also possible) there are meanwhile more modern and flexible alternatives such as **R Markdown**.

R Markdown is a relatively easy to learn markup language which calls usually first a program called **Pandoc** which converts the file in a weaveable or knittable file appropriate for the corresponding target output (for example pdf of html).


All R Markdown files (file extension .Rmd) start with a so-called **YAML** header which tells Pandoc some key information.

A header has usually the form:

```
---
title: "A Pdf report"
author: "Your name"
date: "06.09.2016"
output: pdf_document
    toc: true
---
```

This header for example defines that the output will be a pdf and `toc: true` says the document should have a table of contents.

The easiest way to get a basic header is using in RStudio `File -> New File -> R Markdown ...` and choosing there the desired type.

There are many different output types and the different types have different options.

The options are for example listed in the R markdown reference guide in the sections about Pandoc.

For this course we'll only look at the basic options to create:

- html documents
- pdf documents
- beamer slides

All YAML headers should contain title, author, data and output specifications.

Subarguments are then usually a bit indented.

An example of a html YAML header:

```
---
title: "A webpage"
author: "It was me"
date: "01.01.2001"
output:
  html_document:
    theme: united
    highlight: zenburn
---
```

For more details see http://rmarkdown.rstudio.com/html_document_format.html

Example YAML header for pdf document:

```
---
title: "A Book"
author: "Jane Doe"
date: "1 September 2014"
output:
  pdf_document:
    toc: true
    toc_depth: 1
    fig_width: 5
    fig_height: 3
    fig_caption: true
documentclass: "book"
fontsize:  12
---
```

For more details see http://rmarkdown.rstudio.com/pdf_document_format.html

Example YAML header for beamer slides (with bibtex style bibliography saved in a `.bib` file in the same folder as the .Rmd)

```
---
title: "Statistical Programming"
subtitle: "Reproducible Research"
author: "Max Mustermann"
date: "Autumn 2016"
output:
  beamer_presentation:
    incremental: true
    theme: "Berkeley"
    colortheme: "crane"
    fonttheme: "structurebold"
bibliography: bibliography.bib
---
```

For more details see http://rmarkdown.rstudio.com/beamer_presentation_format.html

The **body** of markdown files can be specific to the target output type and for example, if known to be only LaTeX, can contain direct LaTeX code. But naturally the preference is to be as general as possible.

First we will look how to add R **code chunks**. This is actually quite similar as in Sweave. A general R code chunk in R Markdown starts with three back ticks followed by r plus possible options in curly brackets. Then after the R code the chunk is closed using three back tick in a separate line:

```
```{r options}
 R code
```
```

And to place code inline one uses one back tick followed by r and the R code and one closing back tick:

```
`r  R code`
```

**Some hints using RStudio and R Markdown** When using RStudio to write R Markdown files, one case you `Ctrl + Alt + I` to insert an R code chunk.

When then pointing the cursor after the `r` and pressing `tab` RStudio opens a dropdown menu with code chunk options.

**Useful R code chunk options** Some useful code chunk options are:

- eval: TRUE/FALSE. Should the code be executed or not?
- include: TRUE/FALSE. The code will be executed but should it be included in the final document?
- collapse: TRUE/FALSE. Should the source and output blocks be collapsed into a single block or not?
- echo: TRUE/FALSE. Should the code be displayed in the final document or not?
- results:
  1. "markup": the default
  2. "hide": the results will not be shown in the output
  3. "asis": the results of the R code will not be reformatted. Useful for example when the R function returns html or LaTeX.
- message/warning/error: TRUE/FALSE. Should messages/warnings/errors produced by the code be included in the final document?
- comment: default is `"##"` but can be any character string. The string will be added in front of each line of output in the final document.
- prompt: FALSE/TRUE. Should a `>` be added in front of each line of code in the final document?
- highlight: TRUE/FALSE. Should source code be highlighted in the final document?
- tidy: FALSE/TRUE. Should code chunks be tidied for display using the `tidy_source` function from the `formatR` package?
- dev: the R function name of device to be used to record plots made in the chunk. Default is "png."
- dev.args: list of arguments passed on to the device in R.

- fig.align: How to align figures in the final graph. Options are `"left"`, `"center"` and `"right"`
- fig.cap: character string to be used as the Figure caption in R.
- fig.height/fig.height: Width and Height in inches for the plots created in R.

It is possible to change R code chunk options globally but that can be rather advanced. For example I do not like the default for comment. So I add usually at the beginning of the document the chunk:

````
```{r include=FALSE}
knitr::opts_chunk$set(comment = NA)
```
````

While here we discuss only R as computing language - markdown can be used with other programming languages.

The code chunks are then quite similar and after `{` the `r` is replaced with the name of the other language and the some options are different. (One can use otherwise the option `engine` to specify which programming language should execute the code.)

I personally use as a trick often to get a verbatim kind of environment:

````
```{bash eval=FALSE}
Block in verbatim
```
````

**Tables in R Markdown** Making basic tables in R Markdown is quite simple - a vertical line specifies a new column and the column names are "underscored" using dashes. Then each table row is simply a new line. Note that in each row the columns do not have to be aligned in the code - as long as there is the same number of vertical lines.

So the code

```
Header 1 | Header 2 | Header 3
-------- | -------- | --------
row 1    | row 1    | row 1
row 2    | row 2    | row 2
```

yields

| Header 1 | Header 2 | Header 3 |
|---|---|---|
| row 1 | row 1 | row 1 |
| row 2 | row 2 | row 2 |

To **justify columns** - colons are used in the dashed row.

So the code

```
Header 1 | Header 2 | Header 3
:------- | :------: | -------:
left     | centered | right
left     | centered | right
```

yields

| Header 1 | Header 2 | Header 3 |
|---|---|---|
| left | centered | right |
| left | centered | right |

Naturally, in most cases the content for tables is created in R and the R results should be put straight into the table.

Several functions and packages are useful here, to name a few:

- `kable`: is original function from **knitr**. Can make LaTeX or html tables based on R data.frames and matrices.
- **xtable** (Dahl et al. 2019): a package popular for making LaTeX or html tables. The package provides extreme flexibility over the tables.
- `texreg`: a package that provides automatically high quality tables for LaTeX or html outputs. More or less however only for R objects of specific classes.

In this context the code chunk option `results = "asis"` is important.

For tables using the **kable** package, we basically just have to call the `kable` function for the output we want the table for in an R code chunk. To fine tune the table then further arguments should be passed on to the function. See `?kable` for all options.

For example:

```
```{r echo=FALSE}
library("MASS") # here is the data
library("knitr") # for the kable function
kable(head(crabs))
```
```

yields the following table:

| sp | sex | index | FL | RW | CL | CW | BD |
|---|---|---|---|---|---|---|---|
| B | M | 1 | 8.1 | 6.7 | 16.1 | 19.0 | 7.0 |
| B | M | 2 | 8.8 | 7.7 | 18.1 | 20.8 | 7.4 |
| B | M | 3 | 9.2 | 7.8 | 19.0 | 22.4 | 7.7 |
| B | M | 4 | 9.6 | 7.9 | 20.1 | 23.1 | 8.2 |
| B | M | 5 | 9.8 | 8.0 | 20.3 | 23.0 | 8.2 |
| B | M | 6 | 10.8 | 9.0 | 23.0 | 26.5 | 9.8 |

Nicer would be for example:

```
```{r results="asis", echo=FALSE}
kable(head(crabs[,c(1:2,4:5)]),
  caption = 'Title of Table',
  col.names = c('Species', 'Sex', 'Frontal Lobe',
                'Rear Width'),
  align = c('c','c','c','c'))
```
```

Table 4: Title of Table

| Species | Sex | Frontal Lobe | Rear Width |
|---|---|---|---|
| B | M | 8.1 | 6.7 |
| B | M | 8.8 | 7.7 |
| B | M | 9.2 | 7.8 |
| B | M | 9.6 | 7.9 |
| B | M | 9.8 | 8.0 |
| B | M | 10.8 | 9.0 |

More sophisticated tables can be achieved with the **xtable** package. It works especially well with objects from classes it "knows" about.

For a detailed list run `methods(xtable)` in R.

The tables are then made with the commands

```
library("xtable")
MyTable <- xtable(object, options)
print(MyTable, more options)
```

By default xtable assumes that the target output is created using LaTeX so in `print` a frequent argument is `type="html"`.

Consider the code:

````
```{r results="asis", echo=FALSE}
fit1 <- lm(FL ~ sp + sex + RW, data=crabs)
library(xtable)
fit1table <- xtable(summary(fit1))
print(fit1table, comment=FALSE, floating=FALSE)
```
````

gives the output:

| | Estimate | Std. Error | t value | Pr(>\|t\|) |
|---|---|---|---|---|
| (Intercept) | -2.5964 | 0.3426 | -7.58 | 0.0000 |
| spO | 0.9439 | 0.1265 | 7.46 | 0.0000 |
| sexM | 2.2507 | 0.1255 | 17.93 | 0.0000 |
| RW | 1.3017 | 0.0258 | 50.52 | 0.0000 |

**Tables using texreg** Making tables with **xtable** is easiest when there is only one model object and if `xtable` knows the class.

The **texreg** package (Leifeld 2013) knows more classes and also allows tables combining multiple objects.

The package has mainly two functions for making tables:

- `texreg` for making tables for documents using LaTeX.
- `htmlreg` for documents needing tables in html format.

The arguments of both functions are similar but of course differ a bit.

| | Model 1 |
|---|---|
| (Intercept) | $-2.60^{***}$ |
| | (0.34) |
| spO | $0.94^{***}$ |
| | (0.13) |
| sexM | $2.25^{***}$ |
| | (0.13) |
| RW | $1.30^{***}$ |
| | (0.03) |
| $R^2$ | 0.94 |
| Adj. $R^2$ | 0.94 |
| Num. obs. | 200 |

$^{***}p < 0.001$; $^{**}p < 0.01$; $^{*}p < 0.05$

Table 5: Statistical models

| | Model 1 | Model 2 |
|---|---|---|
| (Intercept) | −2.60 | −2.62 |
| | (0.34) | (0.34) |
| Species | 0.94 | 0.79 |
| | (0.13) | (0.18) |
| Sex | 2.25 | 2.12 |
| | (0.13) | (0.17) |
| Rear Width | 1.30 | 1.31 |
| | (0.03) | (0.03) |
| Species*Sex | | 0.28 |
| | | (0.25) |
| $R^2$ | 0.94 | 0.94 |
| Adj. $R^2$ | 0.94 | 0.94 |
| Num. obs. | 200 | 200 |

Table 6: Comparing two LM fits

**Linear regression table with texreg** Consider the code

```
```{r results="asis", echo=FALSE}
library(texreg)
texreg(fit1)
```
```

which gives the output in Table 5

Consider we have a second model to compare to and also want nicer output (see Table 6).

```
```{r results="asis", echo=FALSE}
fit2 <-  lm(FL ~ sp + sex + RW + sp*sex, data=crabs)
CoefNames <- c('(Intercept)','Species', 'Sex',
               'Rear Width', 'Species*Sex')
texreg(list(fit1,fit2),caption='Comparing two LM fits',
       custom.coef.names=CoefNames, fontsize='tiny',
       stars = numeric(0))
```
```

**Tables of non-standard objects** All three functions presented here to make tables directly from R work on specific objects.

However not all objects classes are known and sometimes one wants to get a table based on some of these objects.

Often `xtable` is used for that. The idea is usually that `xtable` when encountering a matrix or data.frame makes the rows to the rows of the table and the columns to the columns of the table.

Hence a strategy to make tables for unsupported class objects is to

1. find and extract the relevant information from that class object
2. convert these information into a matrix or data.frame
3. use `xtable` on this matrix or data.frame.

**Including figures in R Markdown** Similar to tables also figures need often to be included in documents and again we will first consider the case of figures which are entered "manually" or which are dynamically created in R code chunks. Assume we have a file `Rlogo.png` in the same folder as our .Rmd file and would like to add that.

The code is then simply

```
![R](Rlogo.png)
```

which yields Figure 1 below.

So the structure of the command is

```
![TEXT](PATH TO FILE)
```

This is really simple - however so simple that there is no control over the size or positioning the figure. `TEXT` is optional and the `PATH TO FILE` specifies where to find the file and its name. Most common file types like pdf, jpeg and png are acceptable.

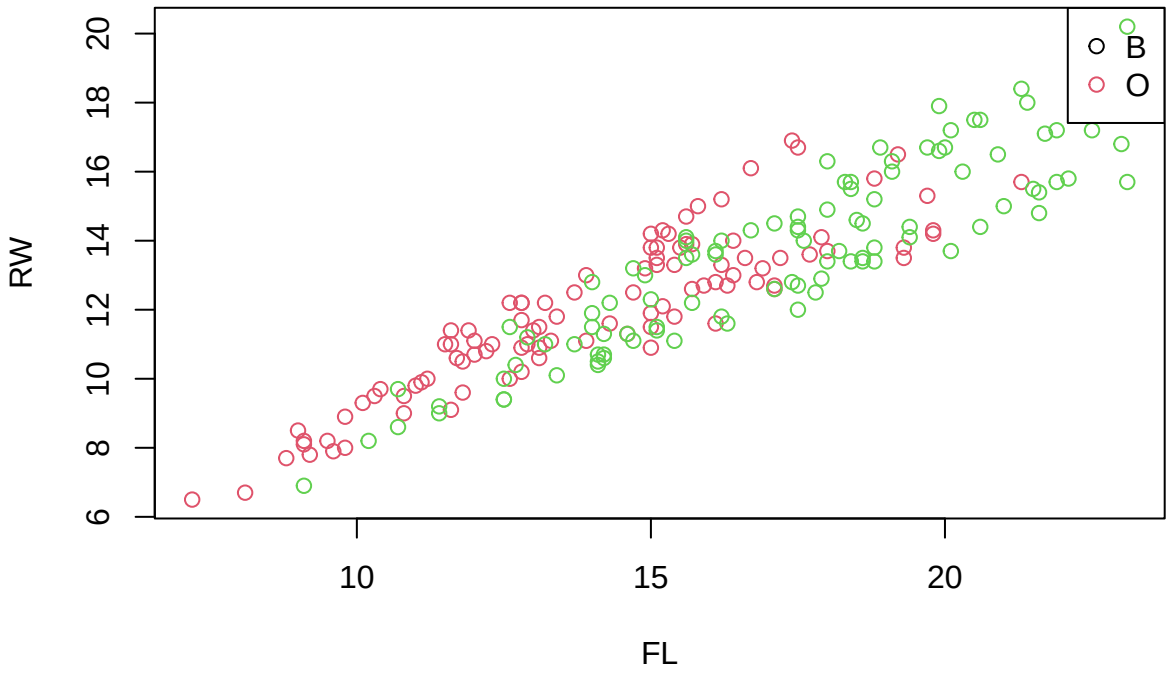
Hence for more sophisticated figures to be added one has to use direct html or latex markup depending on the target document format.

**Including figures created in R Markdown** This is actually already covered - everything is specified in the code chunk options. And it does not matter here if we use basic plotting function, lattice figures or ggplot figures.

For example

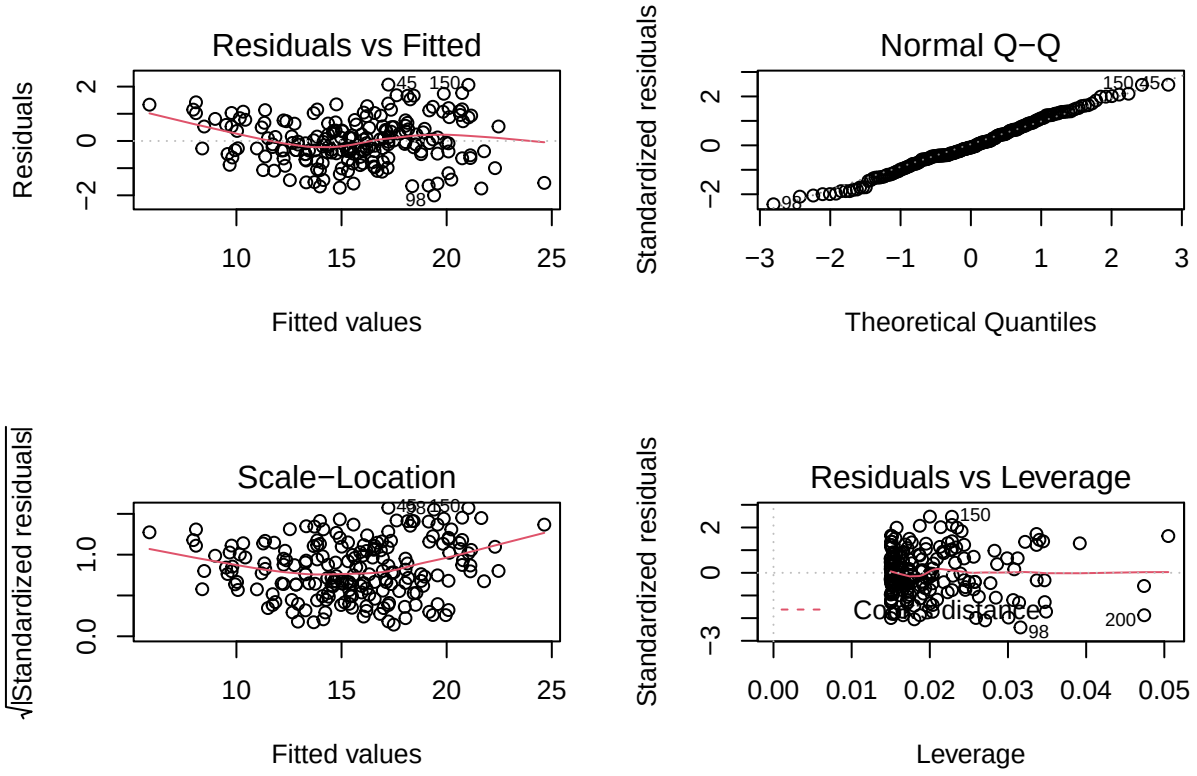
```
```{r scatterplot, results='hide', echo=FALSE}
with(crabs, plot(FL, RW, col=as.numeric(sp)+1))
legend('topright', levels(crabs$sp), pch=1, col=1:2)
```
```

Figure 1: R




To see the regression diagnostic plots of the previous regression model `fit1` we have:

```
```{r fit1plot, results='hide', echo=FALSE}
par(mfrow=c(2,2))
plot(fit1)
```
```



**Equations in R Markdown** As one of the last building blocks of statistical documents, we also have a look at equations. They are in R Markdown basically the same as in LaTeX. Inline equations are specified with in `$equation$` and otherwise as

```
$$
equation
$$
```

Note that for inline equations now spaces can be after the opening `$` and before the closing `$`.

The aim of the course is not to teach LaTeX so we'll give in the following just a few examples to demostrate how it works.

- `$\sum_{i=1}^n x_i^2$` : $\sum_{i=1}^n x_i^2$
- `$\prod_{i=1}^n \log(x_i)$` : $\prod_{i=1}^n \log(x_i)$
- `$\int_a^b  \alpha y dy$` : $\int_a^b \alpha y dy$
- `$\frac{\sin(x)}{3\beta}$` : $\frac{\sin(x)}{3\beta}$
- `$\Sigma^\top$` : $\Sigma^\top$
- `$\Omega^{-1/2}$` : $\Omega^{-1/2}$
- `$p \times q $` : $p \times q$
- `$j=1,\ldots,n$` : $j = 1, \ldots, n$
- `$f(\cdot)$` : $f(\cdot)$
- `$a \leq b \geq \sigma$` : $a \leq b \geq \sigma$
- `$n \ll p$` : $n \ll p$
- `$\lim \omega_i \rightarrow 0$` : $\lim \omega_i \rightarrow 0$
- `$\bar{x}$` : $\bar{x}$
- `$\tilde{x}$` : $\tilde{x}$
- `$\mathcal{A,B,C}$`: $\mathcal{A}, \mathcal{B}, \mathcal{C}$
- `$\mathbb{R,N,Z}$`: $\mathbb{R}, \mathbb{N}, \mathbb{Z}$

```
$$
\left(\begin{array}{cc}
0.8944272 & 0.4472136\\
-0.4472136 & -0.8944272
\end{array}\right)^\top
\left(\begin{array}{cc}
10 & 0\\
0 & 5
\end{array}\right)
$$
```

$$\left(\begin{array}{cc} 0.8944272 & 0.4472136 \\ -0.4472136 & -0.8944272 \end{array}\right)^\top \left(\begin{array}{cc} 10 & 0 \\ 0 & 5 \end{array}\right)$$

```
$$
\alpha = \sqrt{\frac{1}{\pi-3}} \quad \mbox{and} \quad
\sigma^2 = \left(\sum_{i=1}^n (x_i - \bar x)^2\right)
$$
```

$$\alpha = \sqrt{\frac{1}{\pi-3}} \quad \text{and} \quad \sigma^2 = \left(\sum_{i=1}^n (x_i - \bar x)^2\right)$$

**Writing the document** Now that we have all the bricks to weave different parts into our document we just need to learn still how to actually "write" in R Markdown. This is pretty simple!

**Normal text** is just written as normal text. And to get a new paragraph the end of the line needs two spaces.

And otherwise we have the following syntax.

| Syntax | Outcome |
|---|---|
| `plain text` | plain text |
| `*italics*` | *italics* |
| `**bold**` | **bold** |
| `~~strikethrough~~` | ~~strikethrough~~ |
| `[linkToTU](www.tuwien.ac.at)` | linkToTU |
| `# Header 1` | # Header 1 |
| `## Header 2` | ## Header 2 |
| `### Header 3` | ### Header 3 |
| `#### Header 4` | #### Header 4 |
| `##### Header 5` | ##### Header 5 |
| `***` | horizontal rule or slide break |
| `> block quote` | > block quote |

Verbatim text is embedded in single ticks when inline and in three ticks for bigger blocks.

To make an **unordered list** use:

```
* item 1
* item 2
    + subitem 1
    + subitem 2
```

- item 1
- item 2
    - subitem 1
    - subitem 2

For subitems it seems that before the sublist initiator there should be 4 spaces.

To make an **ordered list** use:

```
1. item 1
2. item 2
    a. subitem 1
    b. subitem 2
```

1. item 1
2. item 2
    a. subitem 1
    b. subitem 2

**Note on session info** In statistical analyses results might depend on the software versions used. Therefore in R it is of importance from a reproducibility point of view to give all the relevant details.

For example R and all packages should be properly cited and the version number reported.

To find out what you all used for an analyses in R you can use the `sessionInfo()` function. Here is the session info for the session used to compile this script:

```
sessionInfo()
```

```
R version 4.1.2 (2021-11-01)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS Monterey 12.0.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.1-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.1-arm64/Resources/lib/libRlapack.dylib

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] texreg_1.38.5 xtable_1.8-4  knitr_1.36    MASS_7.3-54

loaded via a namespace (and not attached):
 [1] digest_0.6.29   R6_2.5.1        magrittr_2.0.2  evaluate_0.15
 [5] httr_1.4.2      highr_0.9       rlang_1.0.2     stringi_1.7.5
 [9] cli_3.2.0       rstudioapi_0.13 rmarkdown_2.13  tools_4.1.2
[13] stringr_1.4.0   xfun_0.30       yaml_2.2.1      fastmap_1.1.0
[17] compiler_4.1.2  htmltools_0.5.2
```

**Further reading** To learn more about these there is a lot of free material on the web available.

- For LaTeX for example the "The Not So Short Introduction to LaTeX" is quite popular.
- For Sweave a good starting place is The Sweave Homepage
- Christopher Gandrud: Reproducible Research with R and R Studio, Chapman and Hall/CRC.
- R Markdown Cheatsheet in TUWEL.

## 1.4 Names and values

In R, it is important to understand the distinction between an object and its name. The following code performs an assignment:

```
x <- 1
```

and `x` now appears in the `Global Environment` pane in RStudio. It can then we used in subsequent operations e.g.,

```
x + 1
```

```
[1] 2
```

It's easy to read the assignment command as: "create an object named `x`, containing the value 1." The more proficient R users must note that this is a **simplification** and that it is not representative of what R is actually doing behind the scenes. It's more accurate to say that this code is doing two things:

- It's creating an object with value `1`.
- And it's binding that object to a name, `x`.

In other words, the object, or value, doesn't have a name; it's actually the name that has a value. This subtle distinction is important to understand how objects are stored in R. We will return to this later in the course.

R has strict rules about what constitutes a valid name. A syntactic name must consist of letters, digits, `.` and `_` but can't begin with `_` or a digit. Additionally, you can't use any of the reserved words like `TRUE`, `NULL`, `if`, and `function` (see the complete list in `?Reserved`). A name that doesn't follow these rules is a non-syntactic name; if you try to use them, you'll get an error:

```
_abc <- 1
if <- 10
```

```
Error: <text>:1:1: unexpected input
1: _
    ^
```

It's possible to override these rules and use any name, i.e., any sequence of characters, by surrounding it with backticks:

```
`_abc` <- 1
`_abc`
```

```
[1] 1
```

```
`if` <- 10
`if`
```

```
[1] 10
```

Such weird names typically appear when you load data that has been created outside of R. For some data structures (see e.g., data frames later) the non-syntactic names are converted to syntactic names by the function `make.names()`:

```
make.names(c("_abc", "", "a-b", ".123e1"))
```

```
[1] "X_abc"   "X"       "a.b"     "X.123e1"
```

For an overview of the rules that `make.names()` uses see `?make.names`.

## 1.5 Basic data types in R

The above assignments used a scalar, i.e., a vector of length one.

R has 4 common basic data types for scalar or vectors:

- character
- double: basic type of numeric variables floating point representation of double precision (i.e., 64 bits).
- integer: basic type of integer variables (represented using 32 bits)
- logical (i.e., boolean): `TRUE/FALSE/NA`

and two less common:

- complex
- raw: used to hold raw bytes.

It is easy to check the type of a vector. The basic function is `typeof()`.

```
> x <- "a"
> typeof(x)
[1] "character"
```

```
> y <- 1.5
> typeof(y)
[1] "double"
```

```
> z <- 1L
> typeof(z)
[1] "integer"
```

```
> w <- TRUE
> typeof(w)
[1] "logical"
```

```
> k <- 2 + 4i
> typeof(k)
[1] "complex"
```

Collectively integer and double vectors are known as **numeric** vectors.

**Missing values and other special values in R** **Missing values** in R are specified as `NA` which is internally a logical vector of length 1.

```
typeof(NA)
```

```
[1] "logical"
```

To create NAs of a specific type one can use `NA_real_`, `NA_integer_` or `NA_character_`.

```
typeof(NA_character_)
```

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

`Inf` is infinity. You can have either positive or negative infinity.

```
> 1 / 0
[1] Inf
> -1 / 0
[1] -Inf
> is.infinite(-Inf)
[1] TRUE
> is.finite(-Inf)
[1] FALSE
```

`NaN` means Not a Number. It's an undefined value.

```
> 0 / 0
[1] NaN
> is.nan(0 / 0)
[1] TRUE
```

## 1.6 Basic data structures in R

The five most used data structures in R can be categorized using their dimensionality and whether all content must be of the same type, i.e. if they are homogeneous or heterogeneous.

| | Homogeneous | Heterogeneous |
|---|---|---|
| 1D | vector | list |
| 2D | matrix | data frame |
| nD | array | |

Scalars as on the previous slide are treated as vectors of length 1. And almost all other types of objects in R are build upon these five structures.

To understand the structure of an object in R the best is to use

```
str(object)
```

### 1.6.1 Vectors

The most basic structure is a vector. They come as two different flavors:

- atomic vector
- list

And a vector must have three properties:

- of what type it is (`typeof`)
- how long it is (`length`)
- which attributes it has (`attributes`)

#### Difference of an atomic vector and a list

- In an atomic vector **all** elements must be of the same **type**, whereas in the list the different elements can be of **different types**.
- The basic function to create atomic vectors is `c`.

**The function `c()`** The most direct way to create a vector is the `c` function where all values can be the entered. The values are then **concatenated**.

```
object.x <- c(value1, value2, ...)
```

A single number is also treated like a vector but can be easier assigned to an object:

```
object.x <- value
```

Examples of atomic vectors:

```
LogVector <- c(TRUE, FALSE, FALSE, TRUE)
IntVector <- c(1L, 2L, 3L, 4L)
DouVector <- c(1.0, 2.0, 3, 4)
ChaVector <- c("a", "b", "c", "d")

LogVector
[1]  TRUE FALSE FALSE  TRUE
IntVector
[1] 1 2 3 4
DouVector
[1] 1 2 3 4
ChaVector
[1] "a" "b" "c" "d"
```

**Checking for the types of a vector** Aside from `typeof`, to check for a specific type the "`is`"-functions can be used:

- `is.character`
- `is.double`
- `is.integer`

- is.logical
- is.atomic

```
typeof(IntVector)
[1] "integer"
typeof(DouVector)
[1] "double"
is.atomic(IntVector)
[1] TRUE
is.character(IntVector)
[1] FALSE
is.double(IntVector)
[1] FALSE
is.integer(IntVector)
[1] TRUE
is.logical(IntVector)
[1] FALSE
```

The function `is.numeric` checks if a vector is of type `double` or `integer`.

```
is.numeric(LogVector)
[1] FALSE
is.numeric(IntVector)
[1] TRUE
is.numeric(DouVector)
[1] TRUE
is.numeric(ChaVector)
[1] FALSE
```

**Sequences and replications** To create a vector that has a certain start and ending point and is filled with points that have equal steps between them, the function `seq` can be used.

```
x <- seq(from = 0, to = 1, by = 0.2)
x
[1] 0.0 0.2 0.4 0.6 0.8 1.0
y <- seq(length = 6, from = 0, to = 1)
y
[1] 0.0 0.2 0.4 0.6 0.8 1.0
z <- 1:5
z
[1] 1 2 3 4 5
```

The function `rep` can be used to replicate objects in several ways. For details see the help of the function.

Here are some examples

```
x <- rep(c(2, 1), 3)
x
[1] 2 1 2 1 2 1
y <- rep(c(2, 1), each = 3)
y
[1] 2 2 2 1 1 1
z <- rep(c(2, 1), c(3, 5))
```

```
z
[1] 2 2 2 1 1 1 1 1
```

The `sample` function allows us to obtain of random sample of a specified size from certain elements given in a vector. The following code corresponds to the results of a 6-sided die:

```
sample(1:6, size = 8, replace = TRUE)
```

```
[1] 1 1 3 1 1 6 6 6
```

**Logical operators in R** Logical vectors are usually created by using logical expressions. The logical vector is of the same length as the original vector and gives elementwise the result for the evaluation of the expression.

The logical operators in R are:

| Operator | Meaning |
|---|---|
| `==` | $=$ |
| `!=` | $\neq$ |
| `<` | $<$ |
| `>` | $>$ |
| `>=` | $\geq$ |
| `<=` | $\leq$ |

Two logical expressions $L_1$ and $L_2$ can be combined using:

| | |
|---|---|
| $L_1$ & $L_2$ | for $L_1$ and $L_2$ |
| $L_1 \mid L_2$ | for $L_1$ or $L_2$ |
| $!L_1$ | for the negation of $L_1$ |

Logical vectors typically created in the following way:

```
age <- c(42, 45, 67, 55, 37, 73, 77)
older50 <- age > 50
older50
[1] FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
```

When one wants to enter a logical vector `TRUE` can be abbreviated with `T` and `FALSE` with `F`, this is however not recommended.

**Vector arithmetic** With numeric vectors one normally wants to perform calculations.

When using arithmetic expressions they are usually applied to each element of the vector.

If an expression involves two or more vectors these vectors do not have to be of the same length, the shorter vectors will be **recycled** until they have as many elements as the longest vector.

Important expressions here are:

```
+, -, *, /, ^, log, sin, cos, tan, sqrt,
min, max, range, mean, var
```

Here is an short example for vector arithmetic and the recycling of vectors

```
x <- 1:4
x
[1] 1 2 3 4
y <- rep(c(1,2), c(2,4))
y
[1] 1 1 2 2 2 2
x ^ 2
[1]  1  4  9 16
x + y
Warning in x + y: longer object length is not a multiple of shorter object
length
[1] 2 3 5 6 3 4
```

**Basic operations on character vectors** Taking substrings using `substr` (alternatively `substring` can be used but it has slightly different argument):

```
cols <- c("red", "blue", "magenta", "yellow")
substr(cols, start = 1, stop = 3)
```

```
[1] "red" "blu" "mag" "yel"
```

Building up strings by concatenation within elements using `paste`:

```
paste(cols, "flowers")
```

```
[1] "red flowers"     "blue flowers"    "magenta flowers" "yellow flowers"
```

```
paste(cols, "flowers", sep = "_")
```

```
[1] "red_flowers"     "blue_flowers"    "magenta_flowers" "yellow_flowers"
```

```
paste(cols, "flowers", collapse = ", ")
```

```
[1] "red flowers, blue flowers, magenta flowers, yellow flowers"
```

### Coercion

- As all elements in an atomic vector must be of the same type it is of course of interest what happens if they aren't.
- In that case the different elements will be **coerced** to the most flexible type.
- The most flexible type is usually character. But for example a logical vector can be coerced to an integer or double vector where `TRUE` becomes `1` and `FALSE` a `0`.
- Coercion order: `logical` -> `integer` -> `double` -> (`complex`) -> `character`

```
v1 <- c(1, 2L)
typeof(v1)
[1] "double"
v2 <- c(v1, "a")
typeof(v2)
[1] "character"
v3 <- c(2L, TRUE, TRUE, FALSE)
typeof(v3)
[1] "integer"
```

- Coercion often happens **automatically**. Most mathematical functions try to coerce vectors to numeric vectors. And on the other hand, logical operators try to coerce to a logical vector.
- In most cases if coercion does not work, a warning or error message is returned.
- In programming to avoid coercion to a possibly wrong type the coercion is **forced** using the "`as`"-functions like `as.character`, `as.double`, `as.numeric`,...

```
LogVector
```

```
[1]  TRUE FALSE FALSE  TRUE
```

```
sum(ChaVector)
```

```
Error in sum(ChaVector): invalid 'type' (character) of argument
```

```
as.numeric(LogVector)
```

```
[1] 1 0 0 1
```

```
ChaVector2 <- c("0", "1", "7")
as.integer(ChaVector2)
```

```
[1] 0 1 7
```

```
ChaVector3 <- c("0", "1", "7", "b")
as.integer(ChaVector3)
```

```
Warning: NAs introduced by coercion
```

```
[1]  0  1  7 NA
```

### 1.6.2 Lists

Lists are different from atomic vectors as their elements do not have to be of the same type.

To construct a list one usually uses `list`.

```
List1 <- list(INT = 1L:3L,
              LOG = c(FALSE, TRUE),
              DOU = DouVector,
              CHA = "z")
str(List1)
```

```
List of 4
 $ INT: int [1:3] 1 2 3
 $ LOG: logi [1:2] FALSE TRUE
 $ DOU: num [1:4] 1 2 3 4
 $ CHA: chr "z"
```

The number of components of a list can be obtained using `length`.

```
length(List1)
[1] 4
```

To initialize a list with a certain number of components `vector` can be used.

```
List2 <- vector("list", 2)
List2
[[1]]
NULL

[[2]]
NULL
```

Several lists can be combined into one list using `c`. If a combination of lists and atomic vectors is given to `c` then the function will first coerce each atomic vector to lists before combining them.

```
List3 <- c(List1, list(new = 7:10, new2 = c("G", "H")))
str(List3)
List of 6
 $ INT : int [1:3] 1 2 3
 $ LOG : logi [1:2] FALSE TRUE
 $ DOU : num [1:4] 1 2 3 4
 $ CHA : chr "z"
 $ new : int [1:4] 7 8 9 10
 $ new2: chr [1:2] "G" "H"
```

```
List4 <- list(a = 1, b = 2)
Vec1 <- 3:4
Vec2 <- c(5.0, 6.0)
List5 <- c(List4, Vec1, Vec2)
List6 <- list(List4, Vec1, Vec2)
```

```
str(List5)
List of 6
 $ a: num 1
 $ b: num 2
 $  : int 3
```

```
 $  : int 4
 $  : num 5
 $  : num 6
str(List6)
List of 3
 $ :List of 2
  ..$ a: num 1
  ..$ b: num 2
 $ : int [1:2] 3 4
 $ : num [1:2] 5 6
```

**More on lists**

- The `typeof` a list is a list.
- `is.list` can be used to check if an object is a list.
- `as.list` can be used to coerce to a list.
- To convert a list to an atomic vector `unlist` can be used. It uses the **same coercion** rules as `c`.
- From many statistical functions which return more complicated data structures the results are actually lists.

### 1.6.3 Attributes

- All objects in R can have additional attributes to store metadata about the object. The number of attributes is basically not limited. And it can be thought of as a named list with unique component names.
- Individual attributes can be accessed using the function `attr` or all at once using the function `attributes`.

Examples:

```
VecX <- 1:5
attr(VecX, "attribute1") <- "I'm a vector"
attr(VecX, "attribute2") <- mean(VecX)
attr(VecX, "attribute1")
[1] "I'm a vector"
attr(VecX, "attribute2")
[1] 3
attributes(VecX)
$attribute1
[1] "I'm a vector"

$attribute2
[1] 3
typeof(attributes(VecX))
[1] "list"
```

In R 3 attributes play a special role and we will came back later to them in more detail and just mention them now shortly:

- **names**: the names attributes is a character vector giving each element a name. This will be discussed soon.

- **dimension**: the dim for dimension attribute will turn vectors in matrices and arrays.
- **class**: the class attribute is very important in the context of **S3** classes discussed later.

Depending on the function used attributes might or might not get lost. The three special attributes mentioned earlier have special roles and are usually not lost, many other attributes get however often lost.

```
attributes(5 * VecX - 7)
$attribute1
[1] "I'm a vector"

$attribute2
[1] 3
attributes(sum(VecX))
NULL
attributes(mean(VecX))
NULL
```

**The names attribute** There are three different ways to name a vector:

1. Directly at creation:

```
Nvec1 <- c(a = 1, b = 2, c = 3)
Nvec1
a b c
1 2 3
```

2. By modifying an existing vector in place:

```
Nvec2 <- 1:3
Nvec2
[1] 1 2 3
names(Nvec2) <- c("a", "b", "c")
Nvec2
a b c
1 2 3
```

3. By creating a modified copy:

```
Nvec3 <- setNames(1:3, c("a", "b", "c"))
Nvec3
a b c
1 2 3
```

Properties of names:

- Names do not have to be unique
- Not all elements need names. If no element has a name the names attribute value is `NULL`. If some elements have names but others not, then missing elements get an empty string as name.
- Names are usually the most useful if all elements have a name and if the names are all unique.
- Name attributes can be removed by assigning `names(object) <- NULL`.

```
names(c(a = 1, 2, 3))
[1] "a" ""  ""
names(1:3)
NULL
```

### 1.6.4 S3 Atomic vectors

One vector attribute is class, which underlies the S3 object system. Having a class attribute turns an object into an S3 object, which means it will behave differently from a regular vector when passed to a generic function. Here we will see a variety of atomic vector with an additional class attribute which are widely used in R.

**1.6.4.1 Factors** Categorical data is an important data type in statistics - in R they are usually represented by **factors**.

A factor in R is basically an integer vector with two attributes:

1. The class attribute which has the value `factor` and which makes it behave differently compared to standard integer values.
2. The levels attribute which specifies a set of admissible integers the vector can have.

A factor is usually created with the function `factor`.

```
Fac1 <- factor(c("green", "green", "blue"))
Fac1
```

```
[1] green green blue
Levels: blue green
```

```
class(Fac1)
```

```
[1] "factor"
```

```
levels(Fac1)
```

```
[1] "blue"  "green"
```

```
Fac1[2] <- "red"
```

```
Warning in `[<-.factor`(`*tmp*`, 2, value = "red"): invalid factor level, NA
generated
```

```
Fac1 <- factor(c("green", "green", "blue"))
Fac2 <- factor(c("green", "blue", "blue"))
Fac3 <- factor(c("green", "green"))
Fac4 <- c(Fac1, Fac2)
Fac4
```

```
[1] green green blue  green blue  blue
Levels: blue green
```

```
Fac5 <- c(Fac1, Fac3)
Fac5
```

```
[1] green green blue  green green
Levels: blue green
```

Hence all possible values of a factor should be specified, even when they are not all appearing in the observed vector. This will also often be more informative when analyzing data.

```
buyCha <- c("bought", "bought", "bought")
buyFac <- factor(buyCha, levels = c("bought", "didn't buy"))

table(buyCha)
buyCha
bought
     3
table(buyFac)
buyFac
    bought didn't buy
         3          0
```

**1.6.4.2 Dates and times in R** Dates and times are among the most complicated types to work with on computers.

- Standard calendar is complicated (leap years, months of different lengths, historically different calendars - Julian vs. Gregorian).
- Times depend of an unstated time zone (add daylight savings :-() and some years have leap seconds to keep the clocks consistent with the rotation of the earth!

R can flexibly handle dates and times and has different classes for them with different complexity levels. Most classes offer then also arithmetic functions and other tools to work with date and time objects. A good overview over the different classes is given in the Helpdesk section of the R News 4(1).

- The built-in `as.Date()` function handles dates (without times).
- The contributed library **chron** handles dates and times, but does not control for time zones.
- The `POSIXct` and `POSIXlt` classes allow for dates and times with control for time zones.
- The various `as.` functions can be used for converting strings or among the different date types when necessary.

**The Date class** Objects of class `Dates` are obtained by the `as.Date()` function and can represent years since 1 AD and are normally created by transforming a **character vector** which contains information about the day, month and year.

Note that the behavior of the function when some of the information is missing depends on your system! Also if some of the formats can be read depends on your locale.

To get the current date:

```
Sys.Date()
```

```
[1] "2022-06-01"
```

**as.Date() function** The `as.Date()` function allows for a variety of formats through the `format=` argument.

| Code | Value |
|---|---|
| %d | Day of the month (decimal number) |
| %m | Month (decimal number) |
| %b | Month (abbreviated) |
| %B | Month (full name) |
| %y | Year (2 digit) |
| %Y | Year (4 digit) |
| %C | Century |

*Note:* %y is system dependent so must be employed with care.

Example:

```
x <- c("02.06.1987","6/21/98","31may1960")
as.Date(x[1], format = "%d.%m.%Y")
```

```
[1] "1987-06-02"
```

```
as.Date(x[2], format = "%m/%d/%y")
```

```
[1] "1998-06-21"
```

```
as.Date(x[3], format = "%d%b%Y")
```

```
[1] "1960-05-31"
```

```
lct <- Sys.getlocale("LC_TIME")
lct # my locale is in English
```

```
[1] "en_US.UTF-8"
```

```
Sys.setlocale("LC_TIME", "de_DE.UTF-8") # different for different OSs
```

```
[1] "de_DE.UTF-8"
```

```
as.Date(x[3], format = "%d%b%Y") # may doesn't work for German...
```

```
[1] NA
```

```
Sys.setlocale("LC_TIME", lct) # reset to English
```

```
[1] "en_US.UTF-8"
```

**Internal storage of dates and times** Except for the `POSIXlt` class, dates are stored internally as the number of days or seconds from some reference date. Thus dates in R will generally have a numeric mode, and the class function can be used to find the way they are actually being stored. The `POSIXlt` class stores date/time values as a list of components (hour, min, sec, mon, etc.) making it easy to extract these parts. E.g., internally, `Date` objects are stored as the number of days since January 1, 1970, using negative numbers for earlier dates.

```
y <- as.Date(x[3], format = "%d%b%Y")
y
```

```
[1] "1960-05-31"
```

```
as.numeric(y)
```

```
[1] -3502
```

**Example: Finnish social security number** Every Finn and foreigner working in Finland gets a Finnish social security number. The number is used as a personal identifier and therefore unique for each individual.

The structure of this number is: `DDMMYY C ZZZQ` where

- `DDMMYY` gives the date of birth.
- `C` specifies the century of birth. `+` = 19th Cent., `-` = 20th Cent. and `A` = 21st Cent.
- `ZZZ` is the personal identification number. It is even for females and odd for males
- `Q` is a control number or letter to see if the total number is correct.

We know now that the Finnish social security number contains a lot of useful information. We will discuss now, how to extract the birthday and sex of an individual from their id using R.

The following functions will be needed for this task:

- `substr` extracts a substring from a character vector.
- `paste` can be used to collapse elements of different vectors to one.
- `ifelse` does a vectorized evaluation of an expression. `ifelse(expression, A, B)`, so if expression is true, the result will be `A` and if false `B`.
- `%in%` is a logical operator which is `TRUE` if the for the right side is a match in the left side of the and otherwise `FALSE`.

We will use three Finnish fake ids

```
x <- c("010199-123N", "301001A1234","130620-4567")
xDates <- substr(x, 1, 6)
xSex <- substr(x, 10, 10)
centuries <- ifelse(substr(x, 7, 7) == "+", 19,
                    ifelse(substr(x, 7, 7) == "-", 20, 21))
x2Dates <- paste(xDates, centuries - 1, sep = "")
```

```
birthDates1 <- as.Date(xDates, format = "%d%m%y") # wrong
birthDates2 <- as.Date(x2Dates, format = "%d%m%y%C")
sex <- ifelse(xSex %in% c(0, 2, 4, 6, 8),"Female","Male")
cbind.data.frame(sex, birthDates1, birthDates2)
```

```
     sex birthDates1 birthDates2
1   Male  1999-01-01  1999-01-01
2   Male  2001-10-30  2001-10-30
3 Female  2020-06-13  1920-06-13
```

### 1.6.5 Arrays and matrices

- Adding a `dim` attribute to an atomic vector allows it to behave like a **multidimensional array**.
- A special case of an array is a matrix - there the dimension attribute is of length **2**.
- While matrices are an essential part of statistics, arrays are much rarer but are still useful.
- Usually matrices and arrays are not created by modifying atomic vectors but by using the functions `matrix` and `array`.

```
M1 <- matrix(1:6, ncol = 3, nrow = 2)
M1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
A1 <- array(1:24, dim = c(3, 4, 2))
A1
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

#### Length and names for matrices

- Naturally, also the "length" attribute of a matrix is then two-dimensional. The corresponding functions are `ncol` and `nrow`.
- Similarly "names" has the two version `colnames` and `rownames`.

```
ncol(M1)
[1] 3
nrow(M1)
```

```
[1] 2
colnames(M1) <- LETTERS[1:3]
rownames(M1) <- letters[1:2]
M1
  A B C
a 1 3 5
b 2 4 6
rownames(M1)
[1] "a" "b"
length(M1) ## number of elements in matrix!
[1] 6
c(M1) ## columns are apended in an atomic vector
[1] 1 2 3 4 5 6
```

**Length and names for arrays** The counterpart of `length` for an array is `dim` and the counterpart of `names` is `dimnames` which is list of character vectors of appropriate length.

```
dim(A1)
[1] 3 4 2
dimnames(A1) <- list(c("r1", "r2", "r3"), c("c1", "c2", "c3", "c4"),
                     c("a1", "a2"))
A1
, , a1

   c1 c2 c3 c4
r1  1  4  7 10
r2  2  5  8 11
r3  3  6  9 12

, , a2

   c1 c2 c3 c4
r1 13 16 19 22
r2 14 17 20 23
r3 15 18 21 24
```

**Useful functions in the context for matrices and arrays**

- The extension for `c` for matrices is `cbind` and `rbind`. Similarly the package `abind` provides the function `abind`.
- For transposing a matrix in R the function `t` is available and for the array counterpart the function `aperm`.
- To check if an object is a matrix / array the functions `is.matrix` / `is.array` can be used.
- Similarly coercion to matrices and arrays can be performed using `as.matrix` / `as.array`.

### 1.6.6 Data frames and tibbles

Data frames are in R the most common structures to store data. Internally they are the same as a **list of equal length vectors**. They have however also a similar structure as a matrix.

Hence it shares properties from both types.

For example the function `length` returns for a data frame the number of list components, which is the number of columns and hence the same as `ncol`. While `nrow` returns the the number of rows. Following the same reasoning, `names` gives the names of the vectors which is the same as `colnames`. `rownames` in turn gives the row names.

The function `data.frame` can be used to create data frames. Since R 4.0.0 it does not by default convert character vectors to factors anymore.

```
DF1 <- data.frame(V1 = 1:5,
                  V2 = c("a", "a", "b", "a", "d"))
str(DF1)
'data.frame':   5 obs. of  2 variables:
 $ V1: int  1 2 3 4 5
 $ V2: chr  "a" "a" "b" "a" ...
```

Most functions which read external data into R also return a data frame.

Note that argument `stringAsFactors = TRUE` provides the old behavior of automatic conversion.

```
DF2 <- data.frame(V1 = 1:5,
                  V2 = c("a", "a", "b", "a", "d"),
                  stringsAsFactors = TRUE)
str(DF2)
'data.frame':   5 obs. of  2 variables:
 $ V1: int  1 2 3 4 5
 $ V2: Factor w/ 3 levels "a","b","d": 1 1 2 1 3
```

This can also be controlled globally by using

```
options(stringAsFactors = TRUE)
```

**More on data frames** Basically a data frame is a list with (an S3) class attribute:

```
typeof(DF1)
[1] "list"
class(DF1)
[1] "data.frame"
is.data.frame(DF1)
[1] TRUE
```

**Coercion to data frames** Lists, vectors and matrices can be coerced to data frames if it is appropriate. For lists this means that all objects have the same "length."

```
V1 <- 1:5
L1 <- list(V1 = V1, V2 = letters[c(1, 2, 3, 2, 1)])
L2 <- list(V1 = V1, V2 = letters[c(1, 2, 3, 2, 1, 3)])
str(as.data.frame(V1))
'data.frame':   5 obs. of  1 variable:
 $ V1: int  1 2 3 4 5
```

```
str(as.data.frame(M1))
'data.frame':   2 obs. of  3 variables:
 $ A: int  1 2
 $ B: int  3 4
 $ C: int  5 6
str(as.data.frame(L1))
'data.frame':   5 obs. of  2 variables:
 $ V1: int  1 2 3 4 5
 $ V2: chr  "a" "b" "c" "b" ...
str(as.data.frame(L2))
Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE, : arguments imply di
```

### Combining data frames

- The basic functions to combine two data frames (works similar with matrices) are `cbind` and `rbind`.
- When combining column-wise, then the numbers of rows must match and row names are ignored (hence observations need to be in the same order).
- When combining row-wise the number of columns **and** their names must match.
- For more advanced combining see the function `merge`.

```
cbind(DF1, data.frame(new = 6:10))
  V1 V2 new
1  1  a   6
2  2  a   7
3  3  b   8
4  4  a   9
5  5  d  10
rbind(DF1, data.frame(V1 = 1, V2 = "c"))
  V1 V2
1  1  a
2  2  a
3  3  b
4  4  a
5  5  d
6  1  c
```

**More about cbind** Note that `cbind` (and `rbind`) try to make matrices when possible. Only if at least one of the elements to be combined is a data frame the results will be also a data.frame.

Hence vectors can't usually be combined into a data frame using `cbind`.

```
V1 <- 1:3
V2 <- c("a", "b", "a")
str(cbind(V1, V2))
 chr [1:3, 1:2] "1" "2" "3" "a" "b" "a"
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:2] "V1" "V2"
```

### Special columns in a data frame

- Since a data frame is list of vectors it is possible to have a list as a column.
- However, when a list is given to the `data.frame` function it usually fails as the function tries to put each list item into its own column.
- A workaround is to use the function `I` which is a protector function and says something should be treated **as is**.
- More common than adding a list is to add a matrix to a data frame - also here should the protector function `I` be used.

```
DF4 <- data.frame(a = 1:3)
# works:
DF4$b <- list(1:2,1:3,1:4)
DF4
  a          b
1 1       1, 2
2 2    1, 2, 3
3 3 1, 2, 3, 4
# does not work
DF5 <- data.frame(a = 1:3, b = list(1:2,1:3,1:4))
Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE, : arguments imply di
# does work
DF6 <- data.frame(a = 1:3, b = I(list(1:2,1:3,1:4)))
DF6
  a          b
1 1       1, 2
2 2    1, 2, 3
3 3 1, 2, 3, 4
```

```
DF6 <- data.frame(a = 1:3, b = I(matrix(1:6, nrow = 3)))
DF6
  a b.1 b.2
1 1   1   4
2 2   2   5
3 3   3   6
str(DF6)
'data.frame':   3 obs. of  2 variables:
 $ a: int  1 2 3
 $ b: 'AsIs' int [1:3, 1:2] 1 2 3 4 5 6
```

**Tibbles** Due to frustration over same old-school design choice in `data.frame`s, the **tibble** package (Müller and Wickham 2021) introduced a modified type of data frames, namely tibbles. They share the same structure as data frames but the class vector is longer:

```
attributes(DF1)
```

```
$names
[1] "V1" "V2"

$class
```

```
[1] "data.frame"

$row.names
[1] 1 2 3 4 5
```

```
library("tibble")
DF1t <- tibble(V1 = 1:5,
               V2 = c("a", "a", "b", "a", "d"))
attributes(DF1t)
```

```
$class
[1] "tbl_df"     "tbl"        "data.frame"

$row.names
[1] 1 2 3 4 5

$names
[1] "V1" "V2"
```

**Non-syntactic names** While data frames automatically transform non-syntactic names (unless `check.names = FALSE`), tibbles do not (although they do print non-syntactic names surrounded by “‘).

```
names(data.frame(`1` = 1))
```

```
[1] "X1"
```

```
names(tibble(`1` = 1))
```

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

**Recycling** While every element of a data frame (or tibble) must have the same length, both `data.frame()` and `tibble()` will recycle shorter inputs. However, while data frames automatically recycle columns that are an integer multiple of the longest column, tibbles will only recycle vectors of length one.

```
data.frame(x = 1:4, y = 1:2)
```

```
  x y
1 1 1
2 2 2
3 3 1
4 4 2
```

```
data.frame(x = 1:4, y = 1:3)
```

```
Error in data.frame(x = 1:4, y = 1:3): arguments imply differing number of rows: 4, 3
```

```
tibble(x = 1:4, y = 1)
```

```
# A tibble: 4 x 2
      x     y
  <int> <dbl>
1     1     1
2     2     1
3     3     1
4     4     1
```

```
tibble(x = 1:4, y = 1:2)
```

```
Error:
! Tibble columns must have compatible sizes.
* Size 4: Existing data.
* Size 2: Column `y`.
i Only values of size one are recycled.
```

**Variable reference** There is one final difference: `tibble()` allows you to refer to variables created during construction:

```
tibble(
  x = 1:3,
  y = x * 2
)
```

```
# A tibble: 3 x 2
      x     y
  <int> <dbl>
1     1     2
2     2     4
3     3     6
```

### 1.6.7 NULL

`NULL` is special data structure because it has a unique type, is always length zero, and can't have any attributes:

```
typeof(NULL)
```

```
[1] "NULL"
```

```
length(NULL)
```

```
[1] 0
```

```
x <- NULL
attr(x, "y") <- 1
```

```
Error in attr(x, "y") <- 1: attempt to set an attribute on NULL
```

It is used to represent an empty vector (a vector of length zero) of arbitrary type:

```
c()
```

```
NULL
```

```
c(1, NULL)
```

```
[1] 1
```

or to represent an absent vector (e.g., `NULL` is often used as a default function argument, when the argument is optional but the default value requires some computation).

### 1.6.8 Some more functions for dealing with different data structures

The functions `lapply()` can be useful when working with lists. It will apply a function `FUN` to each element of a list:

```
L1 <- list(1:5, "a", c(TRUE, FALSE))
lapply(L1, length)
```

```
[[1]]
[1] 5

[[2]]
[1] 1

[[3]]
[1] 2
```

Function `apply()` is important for working with data frames or with matrices. It applies a function`FUN` to each row (if `MARGIN = 1`) or each column (if `MARGIN = 2`).

```
df <- data.frame(Income = c(1000, 1100, 1200, 1300, 1400),
                 Age    = c(20, 19, 54, 45, 24))
```

```
## Apply to columns
apply(df, 2, mean)
```

```
Income    Age
1200.0   32.4
```

```
apply(df, 2, function(x) length(x))
```

```
Income    Age
     5      5
```

We add new column to `df`:

```
df$Group <- factor(c("A", "B", "A", "A", "B"))
```

The following command returns an error as we cannot compute the mean of a categorical variable.

```
apply(df, 2, mean)
```

```
Warning in mean.default(newX[, i], ...): argument is not numeric or logical:
returning NA

Warning in mean.default(newX[, i], ...): argument is not numeric or logical:
returning NA

Warning in mean.default(newX[, i], ...): argument is not numeric or logical:
returning NA

Income    Age  Group
    NA     NA     NA
```

We can select the first two columns of the data frame and then use the function `apply()`:

```
apply(df[, 1:2], 2, mean)
```

```
Income    Age
1200.0   32.4
```

Note that `apply()` internally coerces the data frame to a matrix, which means the types are also coerced. Therefore, a better option would be using the function `sapply()` which applies the specified function to each column.

```
sapply(df, mean)
```

```
Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
returning NA

Income    Age  Group
1200.0   32.4     NA
```

For example: we can use the `sapply()` to identify the numeric variables and then use the resulting boolean vector for selecting the columns for which the mean should be computed:

```
id_num <- sapply(df, is.numeric)
id_num
```

```
Income    Age  Group
  TRUE   TRUE  FALSE
```

```
sapply(df[, id_num], mean)
```

```
Income    Age
1200.0   32.4
```

Note that `sapply()` can also be used for lists. If the result of the functions applied on each element are of same length it will return a vector or a matrix (pay attention to coercion!). Otherwise it returns the list. For example,

```
sapply(L1, length) # returns a vector
```

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

```
sapply(L1, summary) # returns a list
```

```
[[1]]
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       2       3       3       4       5

[[2]]
   Length     Class      Mode
        1 character character

[[3]]
   Mode   FALSE    TRUE
logical       1       1
```

The function `tapply()` is also very useful when working with data frame. It applies a function for each factor variable in a vector.

```
tapply(df$Income, df$Group, mean)
```

```
   A    B
1167 1250
```

For computing summary statistics for each column of the data frame the function `summary()` can be used:

```
summary(df)
```

```
     Income          Age       Group
 Min.   :1000   Min.   :19.0   A:3
 1st Qu.:1100   1st Qu.:20.0   B:2
 Median :1200   Median :24.0
 Mean   :1200   Mean   :32.4
 3rd Qu.:1300   3rd Qu.:45.0
 Max.   :1400   Max.   :54.0
```

## 1.7 Subsetting data structures in R

To work with data **subsetting** is a key feature. R is really flexible in this regard and has many different ways to subset the different data structures.

### 1.7.1 Subsetting atomic vectors

We will start subsetting **atomic vectors** as subsetting other structures is quite similar.

There are **six** ways to subset an atomic vector:

1. **positive indexing** using positive integers.
2. **negative indexing** using negative integers.
3. **logical indexing** using logical vectors.
4. **named indexing** using character vectors.
5. **blank** indexing.
6. **zero** indexing.

#### 1.7.1.1 Positive indexing of atomic vectors

One can index an atomic vector by specifying in square brackets the position of the elements which should be selected.

```
V1 <- c(1, 3, 2.5, 7.2, -3.2)
# basic version
V1[c(1, 3)]
[1] 1.0 2.5
# same elements can be selected multiple times
V1[c(1, 3, 1, 3, 1, 3, 1)]
[1] 1.0 2.5 1.0 2.5 1.0 2.5 1.0
# double valued indices are truncated to integers
V1[c(1.1, 3.9)]
[1] 1.0 2.5
```

#### 1.7.1.2 Negative indexing of atomic vectors

One can index an atomic vector by specifying in square brackets the positions of the elements which should **not** be selected.

```
V1[-c(2, 4, 5)]
[1] 1.0 2.5
```

Note that positive and negative indexing cannot be combined:

```
V1[c(-1, 2)]
Error in V1[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

#### 1.7.1.3 Logical indexing of atomic vectors

Giving in square brackets a logical vector of the same length means that the elements with value `TRUE` will be selected.

```
# basic version
V1[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] 1.0 2.5
# if the logical vector is too short,
# it will be recycled.
V1[c(TRUE, FALSE, TRUE)]
[1] 1.0 2.5 7.2
# most common is to use expression
# which return a logical vector
V1[V1 < 3]
[1]  1.0  2.5 -3.2
```

#### 1.7.1.4 Named indexing of atomic vectors

Giving in square brackets a character vector of the names which should be selected.

```
names(V1) <- letters[1:5]
# basic version
V1[c("a", "c")]
  a   c
1.0 2.5
# same elements can be selected multiple times
V1[c("a", "c", "a", "c", "a", "c", "a")]
  a   c   a   c   a   c   a
1.0 2.5 1.0 2.5 1.0 2.5 1.0
# names are matched exactly
V1[c("a", "c", "ab", "z")]
   a    c <NA> <NA>
 1.0  2.5   NA   NA
```

#### 1.7.1.5 Blank and zero indexing of atomic vectors

**Blank indexing** is not useful for atomic vectors but will be relevant for higher dimensional objects. It returns in this case the original atomic vector.

```
V1[]
   a    b    c    d    e
 1.0  3.0  2.5  7.2 -3.2
```

**Zero indexing** returns in this case a zero length vector. It is often used when generating testing data.

```
V1[0]
named numeric(0)
```

### 1.7.2 Indexing lists

Lists are in general subset quite like atomic vectors. There are however more operators available for subsetting:

1. `[` (`[ ]`)
2. `[[` (`[[ ]]`)
3. `$`

The first one returns always a list, the other two options extract list components (details will follow later).

```
L1 <- list(a = 1:2, b = letters[1:3], c = c(TRUE, FALSE))
L1[1]
$a
[1] 1 2
L1[[1]]
[1] 1 2
L1$a
[1] 1 2
```

### 1.7.3 Indexing matrices and arrays

Subsetting of higher dimensional objects can be done in three ways:

1. using multiple vectors.
2. using a single vector.
3. using matrices

The most common way is to generalize the atomic vector subsetting to higher dimensions by using one of the six methods described earlier for each dimension.

Here especially the blank indexing becomes relevant. We will focus here on matrices, but arrays work basically the same.

**1.7.3.1 Subsetting matrices with two vectors** Using one vector for the rows and one vector for the columns:

```
M1 <- matrix(1:6, ncol = 3)
rownames(M1) <- LETTERS[1:2]
colnames(M1) <- letters[1:3]

M1[c(TRUE, FALSE), c("b", "c")]
b c
3 5

M1[ ,c(1, 1, 2)]
  a a b
A 1 1 3
B 2 2 4

M1[-2, ]
a b c
1 3 5
```

**1.7.3.2 Subsetting matrices with one vector** As matrices (arrays) are essentially vectors with a dimension attribute, also a single vector can be used to extract elements. For this it is important that matrices (arrays) filled in column major order.

```
M2 <- outer(1:5, 1:5, paste, sep = ",")
M2
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "1,1" "1,2" "1,3" "1,4" "1,5"
[2,] "2,1" "2,2" "2,3" "2,4" "2,5"
[3,] "3,1" "3,2" "3,3" "3,4" "3,5"
[4,] "4,1" "4,2" "4,3" "4,4" "4,5"
[5,] "5,1" "5,2" "5,3" "5,4" "5,5"

M2[c(3, 17)]
[1] "3,1" "2,4"
```

#### 1.7.3.3 Subsetting matrices with a matrix

This is rarely done but possible. To select elements from an $n$-dimensional object, a matrix with $n$ columns can be used. Each row of the matrix specifies one element. The result will always be a vector. The matrix can consist of integers or of characters (if the array is named).

```
M3 <- matrix(ncol = 2, byrow = TRUE,
             data = c(1, 4,
                      3, 3,
                      5, 1))
M2[M3]
[1] "1,4" "3,3" "5,1"
```

### 1.7.4 Subsetting data frames and tibbles

Recall that data frames are on the one side lists and on the other side similar to matrices.

If a data frame is subset with a single vector it behaves like a list. If subset with two vectors it behaves like a matrix.

```
DF1 <- data.frame(a = 4:6, b = 7:5, c = letters[15:17])
DF1[DF1$a <= 5, ]
  a b c
1 4 7 o
2 5 6 p
DF1[c(1,3), ]
  a b c
1 4 7 o
3 6 5 q
```

To select columns:

```
# like a matrix
DF1[, c("a","c")]
  a c
1 4 o
2 5 p
3 6 q
# like a list
DF1[c("a","c")]
  a c
1 4 o
2 5 p
3 6 q
```

The behavior differs, if only one column is selected:

```
# like a matrix
DF1[, "a"]
[1] 4 5 6
# like a list
DF1["a"]
  a
1 4
2 5
3 6
```

Subsetting a tibble with `[` always returns a tibble:

```
df <- tibble::tibble(a = 4:6, b = 7:5, c = letters[15:17])
str(df["a"])
```

```
tibble [3 x 1] (S3: tbl_df/tbl/data.frame)
 $ a: int [1:3] 4 5 6
```

```
str(df[, "a"])
```

```
tibble [3 x 1] (S3: tbl_df/tbl/data.frame)
 $ a: int [1:3] 4 5 6
```

### 1.7.5 Subsetting arbitrary S3 objects

In general S3 objects consist of atomic vectors, matrices, arrays, lists and so on. And they can be extracted from the S3 object using the same ways as described above.

Again, the initial step is to look at `str` to reveal the details of the object.

Here is an example

```
set.seed(1)
x <- runif(1:100)
y <- 3 + 0.5 * x + rnorm(100, sd = 0.1)
fit1 <- lm(y ~ x)
class(fit1)
[1] "lm"
```

Assume we want to extract individually the three parameters of the model.

```
str(fit1)
List of 12
 $ coefficients : Named num [1:2] 2.982 0.531
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "x"
 $ residuals    : Named num [1:100] 0.0495 -0.0549 0.0342 -0.1234 0.1549 ...
  ..- attr(*, "names")= chr [1:100] "1" "2" "3" "4" ...
 $ effects      : Named num [1:100] -32.572 1.414 0.028 -0.137 0.157 ...
  ..- attr(*, "names")= chr [1:100] "(Intercept)" "x" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:100] 3.12 3.18 3.29 3.46 3.09 ...
  ..- attr(*, "names")= chr [1:100] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
  ..$ qr   : num [1:100, 1:2] -10 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:100] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:2] "(Intercept)" "x"
  .. ..- attr(*, "assign")= int [1:2] 0 1
  ..$ qraux: num [1:2] 1.1 1.05
  ..$ pivot: int [1:2] 1 2
  ..$ tol  : num 1e-07
```

```
  ..$ rank : int 2
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 98
 $ xlevels      : Named list()
 $ call         : language lm(formula = y ~ x)
 $ terms        :Classes 'terms', 'formula'  language y ~ x
  .. ..- attr(*, "variables")= language list(y, x)
  .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "y" "x"
  .. .. .. ..$ : chr "x"
  .. ..- attr(*, "term.labels")= chr "x"
  .. ..- attr(*, "order")= int 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(y, x)
  .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
 $ model        :'data.frame':  100 obs. of  2 variables:
  ..$ y: num [1:100] 3.17 3.12 3.32 3.34 3.24 ...
  ..$ x: num [1:100] 0.266 0.372 0.573 0.908 0.202 ...
  ..- attr(*, "terms")=Classes 'terms', 'formula'  language y ~ x
  .. .. ..- attr(*, "variables")= language list(y, x)
  .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. .. ..$ : chr [1:2] "y" "x"
  .. .. .. .. ..$ : chr "x"
  .. .. ..- attr(*, "term.labels")= chr "x"
  .. .. ..- attr(*, "order")= int 1
  .. .. ..- attr(*, "intercept")= int 1
  .. .. ..- attr(*, "response")= int 1
  .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. .. ..- attr(*, "predvars")= language list(y, x)
  .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
 - attr(*, "class")= chr "lm"
```

```
# the intercept
fit1$coefficients[1]
(Intercept)
      2.982
# the slope
fit1$coefficients[2]
     x
0.5312
# sigma needs to be computed
sqrt(sum((fit1$residuals-mean(fit1$residuals))^2)
    /fit1$df.residual)
[1] 0.09411
```

The variance is actually output by `summary.lm`:

```
summary(fit1)

Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-0.18498 -0.05622 -0.00871  0.05243  0.25166

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.9821     0.0206     145   <2e-16 ***
x             0.5312     0.0353      15   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.0941 on 98 degrees of freedom
Multiple R-squared:  0.697, Adjusted R-squared:  0.694
F-statistic:  226 on 1 and 98 DF,  p-value: <2e-16
```

### 1.7.6 More on standard subsetting operators

We used already the operators `[[` and `$` which are frequently used when extracting parts from lists and other objects.

- `[[` is similar to `[`, but in can only extract single a value/component. Hence only positive integers or a strings can be used in combination with `[[`.
- `$` is a shorthand for `[[` when the component is named.

These operators are mainly used in the context of lists and the difference is that `[` returns **always** a list and `[[` gives the **content** of the list.

```
str(L1)
List of 3
 $ a: int [1:2] 1 2
 $ b: chr [1:3] "a" "b" "c"
 $ c: logi [1:2] TRUE FALSE
L1[[1]]
[1] 1 2
L1[1]
$a
[1] 1 2
L1$a
[1] 1 2
str(L1[[1]])
 int [1:2] 1 2
str(L1[1])
List of 1
 $ a: int [1:2] 1 2
str(L1$a)
 int [1:2] 1 2
```

If [[ is used with a vector of integers or characters then it is assuming nested list structures.

```
L2 <- list(a = list(A = list(aA = 1:3, bB = 4:6),
                    B = "this"),
           b = "that")
str(L2)
List of 2
 $ a:List of 2
  ..$ A:List of 2
  .. ..$ aA: int [1:3] 1 2 3
  .. ..$ bB: int [1:3] 4 5 6
  ..$ B: chr "this"
 $ b: chr "that"
L2[[c("a","A","aA")]]
[1] 1 2 3
```

**Simplification vs preservation** As the different subsetting operators have different properties **simplifying** or **preservation** needs to be remembered at all times as it can have huge impact in programming.

In doubt it is usually better not to simplify. As it is then better that an object is always of the type it was originally.

To prevent or force simplification, the argument `drop` can be specified in `[`.

| | Simplification | Preservation |
|---|---|---|
| vector | `x[[1]]` | `x[1]` |
| list | `x[[1]]` | `x[1]` |
| factor | `x[ind, drop=TRUE]` | `x[1]` |
| matrix | `x[1,]` or `x[,1]` | `x[ind, , drop = FALSE]` or `x[,ind, drop = FALSE]` |
| data frame | `x[,1]` or `x[[1]]` | `x[ , 1, drop = FALSE]` or `x[1]` |

here `ind` is an indexing vector of positive integers and naturally arrays behave the "same" as matrices.

What does simplification mean for atomic vectors? Simplification for atomic vectors concerns the loss of names.

```
V1 <- c(a = 1, b = 2, c = 3)
V1[1]
a
1
V1[[1]]
[1] 1
```

What does simplification mean for lists?

Simplification for lists concerns if the result has to be a list or can be of the type of the extracted object.

```
L1 <- list(a = 1, b = 2:3, c = "a")
str(L1[1])
List of 1
 $ a: num 1
str(V1[[1]])
 num 1
```

What does simplification mean for factors?

Simplification for factors mean that unused levels are dropped.

```
F1 <- factor(c("a", "b", "a"),
             levels = c("a","b","c"))
F1
[1] a b a
Levels: a b c
F1[1]
[1] a
Levels: a b c
F1[1, drop = TRUE]
[1] a
Levels: a
droplevels(F1)
[1] a b a
Levels: a b
```

What does simplification mean for matrices?

Simplification for matrices concerns the loss of a dimension.

```
M1 <- matrix(1:6, nrow=3)
M1[, 1 , drop = FALSE]
     [,1]
[1,]    1
[2,]    2
[3,]    3
M1[, 1]
[1] 1 2 3
```

```
A1 <- array(1:12, dim = c(2, 3, 2))
A1[ , , 1, drop = FALSE]
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
dim(A1[ , , 1, drop = FALSE])
[1] 2 3 1
A1[ , , 1]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
dim(A1[ , , 1])
[1] 2 3
```

What does simplification mean for data frames?

Simplification for data frames means single columns are returned as vectors and not as data frames.

```
DF1 <- data.frame(a = 1:2, b = letters[1:2])
str(DF1[1])
```

```
'data.frame':   2 obs. of  1 variable:
 $ a: int  1 2
str(DF1[[1]])
 int [1:2] 1 2
str(DF1[ , "a", drop=FALSE])
'data.frame':   2 obs. of  1 variable:
 $ a: int  1 2
str(DF1[ , "a"])
 int [1:2] 1 2
```

**More on `$`** Basically `x$n` is equivalent to `x[["n", exact = FALSE]]`.

It is usually used to extract variables from a data frame.

Note that cannot be used to extract variables using stored variable names:

```
DF1 <- data.frame(a = 1:2, b = letters[1:2])
name.a <- "a"
DF1$name.a
NULL
DF1[[name.a]]
[1] 1 2
```

Another difference between `$` and `[[` is that `$` does partial matching.

```
DF1 <- data.frame(aaa=1:2, bbb=letters[1:2])
DF1$a
[1] 1 2
DF1[["a"]]
NULL
```

**Missing and out-of-bounds indices** It is useful to understand what happens with `[[` when one uses an "invalid" index.

An invalid index can be:

- missing (NA)
- zero-length
- out of bounds (integer)
- out of bounds (character)

Some inconsistencies happen among the different data types:

```
x <- list(
  a = list(1, 2, 3),
  b = list(3, 4, 5)
)
y <- c(1, 2, 3)


## missing index
x[[NA]]
```

```
NULL
```

```
y[[NA]]
```

```
Error in y[[NA]]: subscript out of bounds
```

```
## zero length
x[[logical()]]
```

```
Error in x[[logical()]]: attempt to select less than one element in get1index
```

```
y[[logical()]]
```

```
Error in y[[logical()]]: attempt to select less than one element in get1index
```

```
## out of bounds (integer)
x[[5]]
```

```
Error in x[[5]]: subscript out of bounds
```

```
y[[5]]
```

```
Error in y[[5]]: subscript out of bounds
```

```
## out of bounds (character)
x[["e"]]
```

```
NULL
```

```
y[["a"]]
```

```
Error in y[["a"]]: subscript out of bounds
```

**Subsetting and assignment** All subsetting operators can be combined with assigning values to the selected parts.

```
x <- 1:6
x
[1] 1 2 3 4 5 6
x[1] <- 20
x
[1] 20  2  3  4  5  6
x[-1] <- 51:55
x
[1] 20 51 52 53 54 55
x[c(1,1)] <- c(-10,-20)
x
[1] -20  51  52  53  54  55
```

```
## Logical & NA indexing can be combined!
## It is be recycled:
x[c(TRUE, FALSE, NA)] <- 1
x
[1]  1 51 52  1 54 55
```

## 1.8 Flow control

R code is usually executed in a sequential way - one line after another. However, there are methods to control the flow of commands. These are called control statements and they can be categorized into **conditional control** used for branching statements (`if/else`, and `switch`) and **repetitive control** used for looping statements (`for`, `while`, `repeat`).

The `function` construct itself constitutes a different way to affect flow of commands and will be discussed in a later chapter.

### 1.8.1 Conditional control

**1.8.1.1 `if` statement** The `if` statement can be used to control whether and when certain statements are to be executed. The basic form of an `if` statement in R is as follows:

```
if (condition) {
    statements when condition is TRUE
}
```

or

```
if (condition){
    statements when TRUE
} else {
    statements when FALSE
}
```

Typically, the actions are compound statements contained within `{`:

```
x <- 3
if (x < 5) {
  print("'x' is smaller than 5")
} else if (x < 10) {
  print("'x' is at least 5 but less than 10")
} else {
  print("'x' is at least 10")
}
```

```
[1] "'x' is smaller than 5"
```

`if` actually returns a value so that you can assign the results:

```
## use this only if command fits on one line
x1 <- if (TRUE) 1 else 2
x2 <- if (FALSE) 1 else 2
c(x1, x2)
```

```
[1] 1 2
```

The condition should evaluate to a single TRUE or FALSE. Most other inputs will generate an error:

```
if ("x") 1
```

```
Error in if ("x") 1: argument is not interpretable as logical
```

```
if (logical()) 1
```

```
Error in if (logical()) 1: argument is of length zero
```

```
if (NA) 1
```

```
Error in if (NA) 1: missing value where TRUE/FALSE needed
```

Actually, the condition is coerced to a logical. This why running:

```
if (1) 1 else 2
```

```
[1] 1
```

```
if (0) 1 else 2
```

```
[1] 2
```

works.

---

**Exercise:** Check how the coercion to logicals works. What does `as.logical(0)`, `as.logical(0.1)` and `as.logical(10)` return? Why does the following work:

```
x <- 1:10
if (length(x)) "not empty" else "empty"
```

```
[1] "not empty"
```

```
x <- numeric()
if (length(x)) "not empty" else "empty"
```

```
[1] "empty"
```

---

A logical vector of length greater than 1 generates a warning:

```r
if (c(TRUE, FALSE)) 1
```

```
Warning in if (c(TRUE, FALSE)) 1: the condition has length > 1 and only the
first element will be used
```

```
[1] 1
```

In this example the TRUE condition is considered so the code outputs 1.

**1.8.1.2 Vectorized `if`** Statement `if` only works with a single TRUE or FALSE value. If one has a vector of logical values, then the `ifelse()` function can be used.

The basic syntax is:

```r
ifelse(EXPR, yes, no)
```

The result is a vector of same length as `EXPR` that has as corresponding entry the value of `yes` if `EXPR` is `TRUE`, of `no` if `EXPR` is `FALSE`.

```r
x <- 1:4
ifelse(x < 2.5, "yes", "no")
```

```
[1] "yes" "yes" "no"  "no"
```

Note that `ifelse` will try to coerce `EXPR` to logical if it is not.

```r
y <- c(1, 0, 1) # will be coerced to as.logical()
ifelse(y, "yes", "no")
```

```
[1] "yes" "no"  "yes"
```

Also the attributes from `EXPR` will be kept and only the entries replaced.

```r
x <- 1:4
names(x) <- c("x1", "x2", "x3", "x4")
ifelse(x < 2.5, "yes", "no") # keeps the names
```

```
   x1    x2    x3    x4
"yes" "yes"  "no"  "no"
```

Missing values in `EXPR` remain missing values.

```r
x <- 1:4
x[3] <- NA
ifelse(x < 2.5, "yes", "no")
```

```
[1] "yes" "yes" NA    "no"
```

#### 1.8.1.3 `switch` statement

Another possibility for conditional execution is the function `switch`. It is especially useful when the there are more than two possibilities or if the options are named. The basic syntax is:

```
switch(EXPR, options)
```

where `EXPR` can be an integer value which says which option should be chosen, alternatively it can be a character string if the options are named. The recommendation is to typically use `switch()` with character inputs.

For example, it can be used to replace the following code containing nested `if` statements

```
x <- "a"
if (x == "a") {
  "option 1"
} else if (x == "b") {
  "option 2"
} else if (x == "c") {
  "option 3"
} else {
  stop("Invalid `x` value")
}
```

```
[1] "option 1"
```

by

```
x <- "a"
switch(x,
       a = "option 1",
       b = "option 2",
       c = "option 3",
       stop("Invalid `x` value")
)
```

```
[1] "option 1"
```

The last component of a `switch()` should always throw an error, otherwise unmatched inputs will invisibly return `NULL`:

```
switch("c", a = 1, b = 2)
switch("c", a = 1, b = 2, stop("Input not found"))
```

```
Error in eval(expr, envir, enclos): Input not found
```

### 1.8.2 Repetitive control

#### 1.8.2.1 `for` loop

The `for()` statement in R specifies that certain statements are to be repeated a fixed number of times. The syntax looks like:

```
for (index in vector) {
  statements
}
```

This means that the variable `index` runs through all elements in vector. For each value then in vector the statements are executed.

```
for (i in 1:3) print(i)
```

```
[1] 1
[1] 2
[1] 3
```

Common pitfalls

If for each value a result is created which should be stored, then it is recommended to create first an object of the appropriate length which is used to store the results. The `numeric()` or `vector()` functions are helpful here.

```
out <- numeric(3)
for (i in 1:3) {
   out[i] <- i * 10 + 1
}
out
```

```
[1] 11 21 31
```

Pay attention to iterating over S3 vectors, as loops typically strip the attributes:

```
xs <- as.Date(c("2020-01-01", "2010-01-01"))
for (x in xs) {
  print(x)
}
```

```
[1] 18262
[1] 14610
```

Pay attention to iterating over `1:length(x)` when length of `x` is zero (the index can also run backwards):

```
out <- c()
for (i in 1:length(out)) {
  print(i)
}
```

```
[1] 1
[1] 0
```

In such cases, it is better to use `seq_along(x)` instead. It always returns a value the same length as `x`:

```
out <- c()
for (i in seq_along(out)) {
  print(i)
}
```

To compute in R the first 10 Fibonacci numbers we can use a `for` loop in the following way:

```
Fib <- numeric(10) ## create a vector which will store numeric elements
Fib[1] <- 1
Fib[2] <- 1
for (i in 3:10) {
  Fib[i] <- Fib[i-1] + Fib[i-2]
}
Fib
```

```
 [1]  1  1  2  3  5  8 13 21 34 55
```

**1.8.2.2 `while` loop** The `while` loop can be used when statements have to be repeated but is not known in advance how often exactly. The computations should be continued as long as a `condition` is fullfilled. The syntax looks like:

```
while (condition) {
    statements
}
```

Hence here `condition` is evaluated and if `FALSE` nothing will be done. If the condition is however `TRUE`, then the statements are executed. After the statements are executed, the condition is again evaluated. –>

To compute for example all Fibonacci numbers smaller than 100 we could use

```
Fib1 <- 1
Fib2 <- 1
Fibs <- c(Fib1)
while (Fib2 < 100) {
 Fibs <- c(Fibs, Fib2)
 oldFib2 <- Fib2
 Fib2 <- Fib1 + Fib2
 Fib1 <- oldFib2
}
Fibs
```

```
 [1]  1  1  2  3  5  8 13 21 34 55 89
```

Note: increasing the length of a vector can be costly for R! Avoid if possible.

**1.8.2.3 `repeat` loop** If a loop is needed which does not go through a prespecified number of iterations or should not have a condition check at the top the `repeat` loop can be used. The syntax looks like:

```
repeat {
    statements
}
```

This causes the statement to be repeated endlessly. Therefore a terminator called `break` needs to be included. It is usually included as:

```
if (condition) break
```

In general the `break` command can be used in any loop and it causes the loop to terminate immediately. Similarly, the command `next` can also be used in any loop and causes that the computations of the current iteration are terminated immediately and the next iteration is started from the top. The `repeat` loop and the functions `break` and `next` are rarely used since it is much easier to read and understand programs using the other looping methods.

To compute for example all Fibonacci numbers smaller than 100 we could use also

```
Fib1r <- 1
Fib2r <- 1
Fibsr <- c(Fib1r)

repeat {
 Fibsr <- c(Fibsr, Fib2r)
 oldFib2r <- Fib2r
 Fib2r <- Fib1r + Fib2r
 Fib1r <- oldFib2r
 if (Fib2r > 100) break
}
Fibsr
```

```
 [1]  1  1  2  3  5  8 13 21 34 55 89
```

## 1.9 Functions

R is a functional programming language. As John Chambers (creator of S) put it:

> Everything that exists is an object. Everything that happens is a function call.

Functions are **fundamental building blocks** in R and are self contained units of code with a well-defined purpose. To create a function `function()` is used. The parentheses enclose the arguments list. Then a single statement or multiple statements enclosed by `{}` are specified.

When R executes a function definition it produces an object with three parts:

1. `body`: the code inside the function.
2. `formals`: the list of arguments which specify how to call the function.
3. `environment`: the data structure that determines how the function finds the values associated with the names.

When printing the function it will display these parts (if the environment is not shown it is the global environment).

To reduce the burden for the user, one can give default values to some arguments:

```
f <- function(x, y = 1) {
  z <- x + y
  2 * z
}
f
```

```
function(x, y = 1) {
  z <- x + y
  2 * z
}
```

```
formals(f)
```

```
$x


$y
[1] 1
```

```
body(f)
```

```
{
    z <- x + y
    2 * z
}
```

```
environment(f)
```

```
<environment: R_GlobalEnv>
```

There is one exception of a group of functions which have not the three parts just described - these are called **primitive functions**.

All primitive functions are located in the **base** package. They call directly **C code** and do not contain any R code.

```
sum
function (..., na.rm = FALSE)  .Primitive("sum")
formals(sum)
NULL
body(sum)
NULL
environment(sum)
NULL
```

As mentioned, every operation in R is a function call. Really, **every operation in R is a function call**. So also `+`, `-`, `*`, `[`, `$`, `{`, `for`... are functions. To demonstrate how some operators are actually functions check the following code:

```
x <- 10
y <- 20

x + y
[1] 30

'+'(x, y)
[1] 30
```

### 1.9.1 Scope of variables

The scope of a variable tells us where the variable would be recognized. E.g. Variables defined within functions have local scope and are only recognized within the function. In R, scope is controlled by the environment of the functions. Variables defined in console have global scope. Variables defined in functions are visible in the function and in functions defined within in.

Using local variables instead of global ones is less prone to bugs. Also packages in R have their own environment (known as namespace).

```
f <- function(x, y = 1) {
  z <- x + y
  2 * z
}
z
```

```
Error in eval(expr, envir, enclos): object 'z' not found
```

The basic principle of lexical scoping is that names defined inside a function mask names defined outside a function. If a name isn't defined inside a function, R looks one level up.

```
x <- 10
y <- 20
g01 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g01()
```

```
[1] 1 2
```

```
x <- 10
y <- 20
g02 <- function() {
  y <- 2
  c(x, y)
}
g02()
```

```
[1] 10  2
```

The same rules apply if a function is defined inside another function. First, R looks inside the current function. Then, it looks where that function was defined (and so on, all the way up to the global environment). Finally, it looks in other loaded packages.

```
x <- 1
g03 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
```

```
  }
  i()
}
g03()
```

```
[1] 1 2 3
```

In R, functions are ordinary objects. This means the scoping rules described above also apply to functions:

```
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
```

```
[1] 110
```

However, when a function and a non-function share the same name (they must, of course, reside in different environments), applying these rules gets a little more complicated. When you use a name in a function call, R ignores non-function objects when looking for that value.

```
c <- 1
c(a = 1, c = c)
```

```
a c
1 1
```

---

**Exercise:** Check the following code and explain why `f1(3)` and `f2(3)` return different values.

```
y <- 10
f1 <- function(x) {
  g <- function(x) {
    x * y
  }
  y <- 2
  y ^ 2 + g(x)
}
f1(3)

f2 <- function(x) {
  y <- 2
  y ^ 2 + g(x)
}
g <- function(x) {
  x * y
}
f2(3)
```

---

### 1.9.2 Lazy evaluation

In the standard case, R arguments are lazy - they are only evaluated when they are actually used. To force an evaluation you have to use the function `force`.

This also allows us to specify default values in the header of the function for variables which are created locally.

```
f1 <- function(x) 10
f2 <- function(x) {
  force(x)
  10
}
f1(stop("You made an error!"))
[1] 10
f2(stop("You made an error!"))
Error in force(x): You made an error!
```

### 1.9.3 Calling functions

There are different ways to call functions:

1. Named argument call: Arguments are matched by exact names.
2. Partially named argument call: Arguments are matched using the shortest unique string.
3. Positioning argument call: using the position of the arguments in the function definition.

The three different ways can also be mixed in a function call.

Then R uses first named matching, then partial named matching and finally position matching.

```
f <- function(Position1, Pos2, Pos3) {
  list(pos1 = Position1, pos2 = Pos2, pos3 = Pos3)
}

str(f(Position1 = 1, Pos2 = 2, Pos3 = 3))
List of 3
 $ pos1: num 1
 $ pos2: num 2
 $ pos3: num 3
```

```
str(f(Pos2 = 2, Position1 = 1, Pos3 = 3))
List of 3
 $ pos1: num 1
 $ pos2: num 2
 $ pos3: num 3

str(f(1, 2, 3))
List of 3
 $ pos1: num 1
 $ pos2: num 2
 $ pos3: num 3
```

```
str(f(2, 3, Position1 = 1))
List of 3
 $ pos1: num 1
 $ pos2: num 2
 $ pos3: num 3

str(f(2, Posi = 1, 3))
List of 3
 $ pos1: num 1
 $ pos2: num 2
 $ pos3: num 3
```

```
str(f(2, 3, Position1 = 1))
List of 3
 $ pos1: num 1
 $ pos2: num 2
 $ pos3: num 3

str(f(1, Pos = 2, 3))
Error in f(1, Pos = 2, 3): argument 2 matches multiple formal arguments
```

```
# good calls
mean(1:10)
[1] 5.5
mean(1:10, trim=0.2)
[1] 5.5

# confusing calls
mean(1:10, n=T)
[1] 5.5
mean(1:10, , FALSE)
[1] 5.5
mean(1:10, 0.2)
[1] 5.5
mean(, , TRUE, x=1:10)
[1] 5.5
```

In programming you will often have the case, that the arguments to call a function are specified in list. How to use that then for the actual function call?

The solution is the function `do.call`.

```
L1 <- list(1:10, na.rm=TRUE)
do.call(mean, L1)
[1] 5.5
```

**Missing arguments** Sometimes reasonable default values might be complicated and lengthy expressions and difficult to add in the function definition.

Then basically in the inside of the function the **default** is computed only if the argument was not user specified. To check if an argument is missing, the function `missing` is used.

In R it is however more common to set in that case arguments to NULL as default and then check inside the function if the argument is NULL using `is.null`.

```
f1 <- function(a, b) {
  if (missing(a)) a <- "default"
  if (missing(b)) b <- "default"
  list(a = a, b = b)
}

str(f1())
List of 2
 $ a: chr "default"
 $ b: chr "default"
str(f1(1, 2))
List of 2
 $ a: num 1
 $ b: num 2
```

```
f2 <- function(a = NULL, b = NULL) {
  if (is.null(a)) a <- "default"
  if (is.null(b)) b <- "default"
  list(a=a, b=b)
}

str(f2())
List of 2
 $ a: chr "default"
 $ b: chr "default"
str(f2(1, 2))
List of 2
 $ a: num 1
 $ b: num 2
```

### 1.9.4 Function returns

Functions in general can return only **one object** as a rule. Which is however not a real restriction as all the desired output can be collected into a list.

The last expression evaluated in a function is by default the returned object.

Whenever the function `return(object)` is called within a function, the function is terminated and `object` is returned.

```
f1 <- function(x) {
  if (x < 0) return("not positive")
  if (x < 5) {
    "between 0 and 5"
  } else {
    "larger than 5"
  }
}
f1(-1)
[1] "not positive"
f1(1)
[1] "between 0 and 5"
f1(10)
[1] "larger than 5"
```

#### 1.9.4.1 Invisible return

It is possible to return objects from a function call which are not printed by default using the `invisible` function.

Invisible output can be assigned to an object and/or forced to be printed by putting the function call between round parentheses.

```
f1 <- function() 1
f2 <- function() invisible(1)

f1()
[1] 1
f2()

resf2 <- f2()
resf2
[1] 1
(f2())
[1] 1
```

#### 1.9.4.2 The pipe operator

The **magrittr** package defines the pipe operator `%>%` and many other packages also make use of it.

Rather than typing `f(x, y)` we type `x %>% f(y)` (*start with x then use f(y) to modify it*).

R 4.1.x contains a base R pipe `|>` with the same syntax:

```
x <- 1:4; y <- 4
sum(x, y)
```

```
[1] 14
```

```
x |> sum(y)
```

```
[1] 14
```

```
x |> mean()
```

```
[1] 2.5
```

## 1.10 Efficient coding, debugging, benchmarking

*This section has been adapted from lecture notes by Naim Rashid and Michael Love.* For more on this topic you can have a look at Efficient R Programming.

### 1.10.1 Readable code

**Readable code** for a high-level programming language will look similar across whatever language you use.

- *Visual space:* Code within a file is broken into meaningful vertical chunks, and code within a project is broken into meaningful files (this varies a bit from person to person). Use spaces between operators, e.g. `x <- 1` rather than `x<-1`.

- *Non-repeating:* Functions are used to define operations that will be repeated more than once. There should almost never be any code that looks as if it were *copy-pasted* within a project. Variations among similar code chunks can be turned into arguments of a function.
- *Inline documentation:* User-facing functions should have arguments that are documented above the function, along with a description of what the function returns. (We will discuss strategies for doing this in R using *roxygen2* and the *devtools* package.)
- *Comments:* Use lots of comments to describe the choices that are made in the code. It's difficult to actually provide *too many* comments. This helps others, and it will certainly help yourself in the future as you return to your code to make sense of what you were trying to do.
- *Meaningful names:* Function and variable naming is actually quite difficult. Simple and descriptive is good. If you have a function that estimates weights, e.g. `estimateWeights`. For the main variables/objects (such as the ones the user provides as an argument), short variable names are acceptable, especially when these align with standard notation, e.g. `y`, `x`, or abbreviations, `wts`. For intermediate variables/objects it is best to be more descriptive, e.g. `robustEstVar` for a robust estimate of variance. Some R developers use "camel-case" for functions and variables (`estimateWeights`), while others use underscores (`estimate_weights`) for functions or periods for variables names `robust.est.var`. This is not so important, but try to be consistent within a project.

It is also important to maintain a consistent coding style, as this consistency throughout your code helps with readability. As mentioned in class, Hadley Wickham's R style guide is a good place to start (there are others that exist such a the Google R style guide).

### 1.10.2 Efficient code

**Efficient code** is language specific, and here we will focus on efficiency in the R language. The most important factor below however is true also for the other high-level programming languages.

- *Use vectorized functions:* The most important thing to recognize about high-level programming languages is that they are built on top of fast routines written in Fortran, C, or C++ mostly. Iteration in the high-level language will always be slower than the *vectorized* function which iterates in the lower-level language. Use of row- or column-based operations over matrices, or matrix multiplication, that avoids iterating over rows or columns is one of the keys to efficient programming in R.
- *Allocate first:* Another pitfall for writing R code is any operation which grows the memory space required for an object at each iteration. You should almost never have a loop where the inside of the loop has `out <- c(out, new.thing)`. Concatenation is ok, but remember that it has a time cost, so you don't want to be doing it often. Loops in R are not as bad as you may have heard, as long as the space has been pre-allocated. Loops in R will likely be slower than using a vectorized function, or a loop in C or C++, but they don't need to be avoided at all cost.
- *Avoid copying large objects:* This goes along with the above point, but copying large objects takes up a lot of time in R. To the extent that you can avoid making copies of a data object, you will avoid unnecessary slowdowns.
- *Go to C or C++:* You can typically gain a lot of speed by moving a repetitive operation from R to C or C++.
- *Caching variables and memoization:* A straightforward method for speeding up code is to calculate objects once and reuse the value when necessary. This could be as simple as replacing `sd(x)` in multiple function calls with the object `sd_x <- sd(x)` that is defined once and reused. If you happen to be writing a function that will take input which has repeated elements, and it is very important for this function to be very fast, memoization can be a useful technique. Memoization entails storing the values of expensive function calls so that they don't have to be repeated. There is a small overhead in saving and looking up the precomputed values, but if the degree of repeated input is high, the savings of memoization can be large. The **memoise** package helps with this in R.

- *Parallelization:* One can think of the standard for loop as a serial operation: the $(i+1)$th iteration is always ran after the $(i)$th iteration has completed. On machines with more than one CPU available, say with $P$ CPUs, parallelization of iterations may allow for the execution of up to $P$ iterations of the for loop at the same time. This is particularly helpful when there are a large number of loop iterations that are non-recursive, meaning the next iteration of the loop does not depend on the prior one (simulations for example). One thing to note is that each parallel instance invoked in R requires additional memory, and therefore in memory-intensive operations this may quickly exhaust the available memory on your machine if you are not careful.

Two notes on avoiding making copies, from Section 2.5 in Advanced R:

> For loops have a reputation for being slow in R, but often that slowness is caused by every iteration of the loop creating a copy. Consider the following code. It subtracts the median from each column of a large data frame:

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
medians <- sapply(x, median)

for (i in seq_along(medians)) {
  x[[i]] <- x[[i]] - medians[[i]]
}
```

We can use the `tracemem` function to understand if and how many copies are being done in R for this for loop:

```
cat(tracemem(x), "\n")
```

```
<0x126085fd8>
```

```
for (i in seq_along(medians)) {
  x[[i]] <- x[[i]] - medians[[i]]
}
```

```
tracemem[0x126085fd8 -> 0x1261b0048]: eval eval eval_with_user_handlers withVisible withCallingHandlers
tracemem[0x1261b0048 -> 0x1261affd8]: [[<-.data.frame [[<- eval eval eval_with_user_handlers withVisibl
tracemem[0x1261affd8 -> 0x1261afef8]: eval eval eval_with_user_handlers withVisible withCallingHandlers
tracemem[0x1261afef8 -> 0x1261afe18]: [[<-.data.frame [[<- eval eval eval_with_user_handlers withVisibl
tracemem[0x1261afe18 -> 0x1261afda8]: eval eval eval_with_user_handlers withVisible withCallingHandlers
tracemem[0x1261afda8 -> 0x1261afcc8]: [[<-.data.frame [[<- eval eval eval_with_user_handlers withVisibl
tracemem[0x1261afcc8 -> 0x1261afb78]: eval eval eval_with_user_handlers withVisible withCallingHandlers
tracemem[0x1261afb78 -> 0x1261af408]: [[<-.data.frame [[<- eval eval eval_with_user_handlers withVisibl
tracemem[0x1261af408 -> 0x1261d8c38]: eval eval eval_with_user_handlers withVisible withCallingHandlers
tracemem[0x1261d8c38 -> 0x1261d8b58]: [[<-.data.frame [[<- eval eval eval_with_user_handlers withVisibl
```

```
untracemem(x)
```

In fact, each iteration copies the data frame not once, but twice! We can reduce the number of copies by using a list instead of a data frame. (modifying a list uses internal C code and deals better with counting bindings, see Section 2.5 in Advanced R for more details).

```
y <- as.list(x)
cat(tracemem(y), "\n")
```

```
<0x1262311e8>
```

```
for (i in 1:5) {
  y[[i]] <- y[[i]] - medians[[i]]
}
```

```
tracemem[0x1262311e8 -> 0x126247568]: eval eval eval_with_user_handlers withVisible withCallingHandlers
```

```
untracemem(y)
```

Now the object is only copied once.

> While it's not hard to determine when a copy is made, it is hard to prevent it. If you find yourself resorting to exotic tricks to avoid copies, it may be time to rewrite your function in C++.

### 1.10.3 Benchmarking and profiling R code

**1.10.3.1 Benchmarking** We will make use of **bench** package (Hester and Vaughan 2021) to assess efficiency of methods here and again in the course. It is not part of the core set of R packages, so you will need to install it with `install.packages`.

You can compare two implementations by passing them to the `mark` function:

```
library("bench")
slow.sqrt <- function(x) {
  ans <- numeric(0)
  for (i in seq_along(x)) {
    ans <- c(ans, sqrt(x[i]))
  }
  ans
}
mark(sqrt(1:1000), slow.sqrt(1:1000))
```

```
# A tibble: 2 x 6
  expression             min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>        <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 sqrt(1:1000)        2.34us   4.06us   242906.    15.72KB     72.9
2 slow.sqrt(1:1000)   2.22ms   2.49ms      400.     3.89MB     32.7
```

This benchmark indicates that the vectorized version of the square root is many orders of magnitude faster than a naive implementation. Let's compare `slow.sqrt` with a version where we preallocate the vector that stores the eventual output of the function:

```
pre.sqrt <- function(x) {
  ans <- numeric(length(x))
  for (i in seq_along(x)) {
    ans[i] <- sqrt(x[i])
```

```
  }
  ans
}
mark(pre.sqrt(1:1000), slow.sqrt(1:1000))
```

```
# A tibble: 2 x 6
  expression             min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>        <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 pre.sqrt(1:1000)   58.71us  60.72us    16168.   31.62KB     2.02
2 slow.sqrt(1:1000)   2.23ms   2.48ms      403.    3.86MB    35.9
```

So simply pre-allocating saves us about an order of magnitude in speed.

**1.10.3.2 Profiling** If you know that your function is correct but think it is slow you can do **profiling** which helps to identify the parts of the functions which are bottlenecks and then you can consider if these parts could be improved.

The idea in profiling is that the software checks in very short intervals which function is currently running. The main functions in R to do profiling are `Rprof` and `summaryRprof`. But there are also many other specialized packages for this purpose. We will look only at the package **profvis** (Chang, Luraschi, and Mastny 2020).

A function to profile

```
Stest <- function(n = 1000000, seed = 1) {
  set.seed(seed)
  normals <- rnorm(n*10)
  X <- matrix(normals, nrow=10)
  Y <- matrix(normals, ncol=10)
  XXt <-  X %*% t(X)
  XXcp <- tcrossprod(X)
  return(n)
}
system.time(Stest())
   user  system elapsed
  0.927   0.050   0.984
```

```
Rprof(interval = 0.01)
Stest()
[1] 1e+06
Rprof(NULL)
summaryRprof()$by.self
             self.time self.pct total.time total.pct
"rnorm"           0.40    52.63       0.40     52.63
"%*%"             0.15    19.74       0.15     19.74
"tcrossprod"      0.12    15.79       0.12     15.79
"matrix"          0.06     7.89       0.06      7.89
"t.default"       0.03     3.95       0.03      3.95
```

Run it on your own computer and look at the full output of `summaryRprof()`.

The **profvis** package yields interactive results displayed in html and therefore this again should be done in R (best RStudio) and cannot be visualized in the lecture notes.

```
library(profvis)
profvis({
  Stest <- function(n=1000000, seed=1){
    set.seed(seed)
    normals <- rnorm(n*10)
    X <- matrix(normals, nrow=10)
    Y <- matrix(normals, ncol=10)
    XXt <-  X %*% t(X)
    XXcp <- tcrossprod(X)
    return(n)
  }
  Stest()
})
```

### 1.10.4 Debugging

There are two commonly referred to claims:

1. Programmers spent more time on debugging their own code that actually programming it.
2. In every 20 lines of code is at least one bug.

Hence debugging is an essential part of programming and there are strategies and tools available in R to do this well in R. In the following we introduce several strategies.

**Top-down programming** The general agreement is that good code is written in a modular manner. This means when you have a procedure to implement, you decompose it into small parts where each part will become an own function. Then the main function is "short" and will consist mainly of calling these subfunctions. Naturally also within these functions the same approach is to be taken (see Section on *Readable code*).

Then same approach is followed in debugging. First the top level function is debugged and all subfunctions are assumed correct. If this does not yield a solution, then the next level is debugged and so on.

**Small start strategy** The **small start** strategy in debugging suggests to start using small test cases for the debugging. Once these work fine, then consider larger testing cases. At that stage also extreme cases should be tested.

**Antibugging** Also some **antibugging** strategies are useful in this context. Assume that at line $n$ in your code you know that variable or vector `x` must have some specific property, like being positive or sum up to 1.

Then you can add in that line in the code for debugging purposes for example

```
stopifnot(x > 0)
```

or

```
stopifnot(sum(x) == 1)
```

This might help to narrow done where the bug occurred.

**R functions for debugging** R provides many functions to help in the debugging process. To name some:

- `browser`
- `debug` and `undebug`
- `debugger`
- `dump.frames`
- `recover`
- `trace` and `untrace`

For details about these functions see their help pages. In the following we will look only at `debug` and `traceback`.

Note that also Rstudio offers special debugging tools, see https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio for details.

**traceback** Often when using functions and error occurs it is not really clear where the actually error occurs, which (sub)function caused the error. One strategy then is to use the `traceback` function, which returns when called directly after the erroneous call the sequence of function calls which lead to the error.

```
f1 <- function(x)  f2(x)^2
f2 <- function(x)  log(x) + "x"
mainf <- function(x) {
  x <- f1(x)
  y <- mean(x)
  y
}
mainf(1:3)
traceback()
```

```
Error in log(x) + "x": non-numeric argument to binary operator
3: mainf(1:3)
2: f1(x)
1: f2(x)
```

**debug** Assume you have a function `foo` you assume faulty. Using then

```
debug(foo)
```

will open whenever the function is called the "browser" until either the function is changed or the debugging mode terminated using

```
undebug(foo)
```

In the "browser" the function will be executed line by line where always the **next to be executed line** will be shown.

In the browsing mode the following commands have a special meaning:

- `n` (or just hitting enter) will execute the line shown and then present the next line to be executed.
- `c` this is almost like `n` just that it might execute several lines of code at once. For example if you are in a loop then c will jump to the next iteration of the loop.

- `where` this prints a stack trace, the sequence of function calls which led the execution to the current location
- `Q` this quits the browser.

And in the browser mode any other R command can be used. However to see for example the value of a variable `n`the variable needs then to explicitly printed using `print n`. In a demo we will go through the following function in debugging mode

```
SimuMeans <- function(m, n = 100, seed = 1) {
  set.seed(seed)
  RES <- matrix(0, nrow = m, ncol = 3)
  for (i in 1:m){
    X <- cbind(rnorm(n), rt(n,2), rexp(n))
    for (j in 1:3){
      RES[i,j] <- mean(X[,j])
    }
    print(paste(i,Sys.time()))
  }
  return(RES)
}
debug(SimuMeans)
SimuMeans(5)
```

**Capturing errors** Especially in simulations it is often desired that when an error occurs that not the whole process is terminated but that the error is **caught** and an appropriate record made but otherwise the simulations should continue. R has for this purpose the function `try` and `tryCatch` where we will consider only `tryCatch`.

The idea of `tryCatch` is to run the "risky" part where errors might occur within the `tryCatch` call and tell `tryCatch` what to return in the case of an error.

Consider a modified version of our previous simulation function:

```
my.mean <- function(x){
  na.fail(x)
  mean(x)
}
SimuMeans2 <- function(m, n=100, seed=1) {
  set.seed(seed)
  RES <- matrix(0, nrow=m, ncol=3)
  for (i in 1:m){
    X <- cbind(rnorm(n), rt(n,2), rexp(n))
    if (i==3) X[1,1] <- NA
    for (j in 1:3){
      RES[i,j] <- my.mean(X[,j])
    }
  }
  return(RES)
}
SimuMeans2(5)
Error in na.fail.default(x): missing values in object
```

Using `tryCatch`:

```
SimuMeans3 <- function(m, n=100, seed=1) {
  set.seed(seed)
  RES <- matrix(0, nrow=m, ncol=3)
  for (i in 1:m){
    X <- cbind(rnorm(n), rt(n,2), rexp(n))
    if (i==3) X[1,1] <- NA
    for (j in 1:3){
      RES[i,j] <- tryCatch(my.mean(X[,j]), error = function(e) NA)
    }
  }
  return(RES)
}
SimuMeans3(5)
```

```
         [,1]     [,2]   [,3]
[1,]  0.10889 -0.29099 1.1103
[2,] -0.04921 -0.17200 0.8624
[3,]       NA -0.02305 1.0302
[4,] -0.09209 -0.27303 1.0814
[5,] -0.05374  0.13526 1.0200
```

# 2 Part 2: Data manipulation using the tidyverse

**Data manipulation** (also refereed to as **data wrangling**) usually consists of several steps and includes: loading the data into the workspace, reformatting and restructuring the data and perhaps exporting the data.

Before discussing how data can be imported into R, it is worth mentioning that base R and a lot of add on packages have built-in datasets (i.e., `data.frame` objects) to demonstrate the usage of functions. Those datasets can be loaded using the function

```
data("foo", package = "pkg")
```

This function searches following the search path for a dataset with the corresponding name. A list of all datasets currently available can retrieved submitting only

```
data()
```

Detailed information for a dataset is given in the **datasets** helpfile.

Also, if you are not familiar with what a **working directory** is, here is some information on this topic. The **working directory** is the path where R will search by default for files to read or where R will by default save files. The current working directory can be obtained or changed using functions `getwd()` and `setwd()` or from the RStudio Session menu.

Here is my working directory for compiling these lecture notes.

```
getwd()
```

```
[1] "/Users/lauravanagur/Documents/Teaching/Statistical Computing/Script"
```

When opening a file with RStudio, it automatically sets the working directory to the location of the file. In R Markdown it is automatically the location of the .Rmd.

Note that paths can be specified in an **absolute** way or in a **relative** way (i.e., relative to the working directory). Assume I want to change my working directory to `/Users/lauravanagur/Documents/Teaching/Statistical Computing/` (one level above the current working directory):

```
## absolute
setwd("/Users/lauravanagur/Documents/Teaching/Statistical Computing/")
## relative
setwd("../")
```

where `../` means folder one level up. Note that in an .Rmd it is not advisable to use such commands to change the working directory.

Relative paths are useful for reproducibility. They ensure that if another user has the same folder structure, they can use the same paths. We will see some more examples later.

## 2.1 Data import

The first step in any data analysis is to load the data into your workspace. This seemingly easy task can be quite challenging since

- data come in different formats
- data come in different sizes (and can exceed your memory)

The format of the data does not only include the kind of file the data is saved in but also how the data is encoded, etc. Most data sets come in tabular or spreadsheet-like form, which means that each row stands for an observation and each column for a variable. This kind of data is often referred to as flat data and can come in different file formats, e.g., .csv, .txt, .xlsx. R can also import data from other statistical software platforms such as STATA, SPSS, SAS. Other forms include XML or HTML data, which is highly structured but not flat data.

An important criterion for the choice of the data import function is the size of the data. If the data set exceeds the size of your memory you cannot load the entire data set into your workspace and will have to work with databases. This section is supposed to give a basic overview over the different kind of functions in R used to import data into the workspace.

### 2.1.1 `read.table()` and derivatives

The function `read.table` is the most convenient way to read-in a rectangular grid of data. It belongs to the **utils** package and is loaded by default. Because of the many possibilities, there are several other functions that call `read.table` but change a group of default arguments. These variants are `read.csv()`, `read.csv2()`, `read.delim()`, `read.delim2()`.

The number of options can be seen from its help page

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
```

```
           fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)

read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)

read.csv2(file, header = TRUE, sep = ";", quote = "\"",
          dec = ",", fill = TRUE, comment.char = "", ...)

read.delim(file, header = TRUE, sep = "\t", quote = "\"",
           dec = ".", fill = TRUE, comment.char = "", ...)

read.delim2(file, header = TRUE, sep = "\t", quote = "\"",
            dec = ",", fill = TRUE, comment.char = "", ...)
```

See `?read.table` for a description of each argument.

Assume there is a file called `iris.csv` in a folder called `data` in your working directory.

```
# getwd()
iris <- read.csv("data/iris.csv")
str(iris)
```

```
'data.frame':   150 obs. of  6 variables:
 $ X           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : chr  "setosa" "setosa" "setosa" "setosa" ...
```

Import worked well!

### 2.1.2 `readr` of the tidyverse

The tidyverse contains its own implementation of these functions, which are bundled in the package **readr** (Wickham, Hester, and Bryan 2021).

```
library("readr")
```

The functionality of the function `read_csv` and `read_tsv` corresponds more or less to the functions `read.csv` and `read.delim` of the **utils** package, however, it is

- faster
- the argument `stringsAsFactors` is set to FALSE by default (also the case in the **utils** package since R.4.x.x).
- output is a tibble

```
read_delim(file, delim, quote = "\"", escape_backslash = FALSE,
  escape_double = TRUE, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  comment = "", trim_ws = FALSE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max), progress = show_progress())

read_csv(file, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  quote = "\"", comment = "", trim_ws = TRUE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max), progress = show_progress())

read_csv2(file, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  quote = "\"", comment = "", trim_ws = TRUE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max), progress = show_progress())

read_tsv(file, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  quote = "\"", comment = "", trim_ws = TRUE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max), progress = show_progress())
```

```
iris2 <- read_csv("data/iris.csv")
```

```
New names:
* `` -> ...1

Rows: 150 Columns: 6
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Species
dbl (5): ...1, Sepal.Length, Sepal.Width, Petal.Length, Petal.Width

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
str(iris2)
```

```
spec_tbl_df [150 x 6] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ ...1        : num [1:150] 1 2 3 4 5 6 7 8 9 10 ...
 $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : chr [1:150] "setosa" "setosa" "setosa" "setosa" ...
 - attr(*, "spec")=
  .. cols(
  ..   ...1 = col_double(),
  ..   Sepal.Length = col_double(),
  ..   Sepal.Width = col_double(),
  ..   Petal.Length = col_double(),
  ..   Petal.Width = col_double(),
  ..   Species = col_character()
```

```
  .. )
 - attr(*, "problems")=<externalptr>
```

The functions in `readr` print more output to the console which help the user verify that the data has been correctly imported. Notice for example that the `Species` column was not saved imported as a but remained a character string.

### 2.1.3 `data.table`

The **data.table** package (Dowle and Srinivasan 2021) has one objective: to increase the speed of importing data in R. It is

- very fast and
- produces objects of class `data.table`, a special kind of `data.frame`.

```
library("data.table")
iris3 <- fread("data/iris.csv")
str(iris3)
```

```
Classes 'data.table' and 'data.frame':  150 obs. of  6 variables:
 $ V1          : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : chr  "setosa" "setosa" "setosa" "setosa" ...
 - attr(*, ".internal.selfref")=<externalptr>
```

The output of this function is of class `data.table`, a child of `data.frame`. There is much more to learn about the **data.table** package which will not be covered in this class. Interested students can find more information in the package vignette or in a separate DataCamp course.

### 2.1.4 Read in excel files `xlsx`

There are a few packages that import data from .xlsx files. My package of choice would be the **readxl** package (Wickham and Bryan 2022):

```
library("readxl")
```

The first handy function lists all sheets in an Excel spreadsheet.

```
excel_sheets(path)
```

The actual reading of the file is implemented in function `read_excel`:

```
read_excel(path, sheet = NULL, range = NULL, col_names = TRUE,
  col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max))
```

See the documentation in `?read_excel` for details on the arguments.

```
str(read_excel("data/iris.xlsx"))
```

```
New names:
* `` -> ...1

tibble [150 x 6] (S3: tbl_df/tbl/data.frame)
 $ ...1        : num [1:150] 1 2 3 4 5 6 7 8 9 10 ...
 $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : chr [1:150] "setosa" "setosa" "setosa" "setosa" ...
```

The example file `iris.xlsx` has only one sheet:

```
excel_sheets("data/iris.xlsx")
```

```
[1] "iris"
```

The easiest way to import all spreadsheets from an excel file and save them into a list is by using `lapply()`:

```
my_workbook <- lapply(excel_sheets("data/iris.xlsx"), read_excel,
                      path = "data/iris.xlsx")
```

Currently, there is no possibility to write .xlsx files from the R console with this package. Other packages, however, provide this function.

### 2.1.5 Other formats

The `haven` package of the **tidyverse** (Wickham and Miller 2021) provides functions to import and export data from/for SPSS, Stata and SAS.

```
library("haven")
```

The necessary functions are

```
read_sav(file, user_na = FALSE)

read_por(file, user_na = FALSE)

read_dta(file, encoding = NULL)

read_sas(data_file, catalog_file = NULL, encoding = NULL,
  catalog_encoding = encoding, cols_only = NULL)
```

### 2.1.6 Import XML Data

When reading data from text files, it is the responsibility of the user to know and to specify the conventions used to create that file, e.g. the comment character, whether a header line is present, the value separator, the representation for missing values etc. A markup language which can be used to describe not only content but also the structure of the content can make a file self-describing, so that one does not need to provide these details to the software reading the data.

The eXtensible Markup Language - more commonly known simply as XML - can be used to provide such structure, not only for standard datasets but also for more complex data structures. XML is becoming extremely popular and is emerging as a standard for general data markup and exchange. It is being used by different communities to describe geographical data such as maps, graphical displays, mathematics etc. The function necessary to parse an XML document, `xmlInternatTreeParse()` or `xmlTreeParse()`, are in the **XML** package (Temple Lang 2021) and can be installed via `install.packages("XML")`

```
library("XML")
```

### 2.1.7 Importing from databases

Instead of loading all the data into RAM, as R does, databases query data from the hard-disk. This can allow a subset of a very large dataset to be defined and read into R quickly, without having to load it first. However, this depends of the kind of database that we want to load from. There are many different kinds of databases. An excellent overview over the different packages and functions can be found under https://db.rstudio.com/databases/.

As an example, let us have a look at the **RODBC** package (Ripley and Lapsley 2021). It is one the most mature packages of this kind and sets up links to external databases using the Open Database Connectivity (ODBC) API. **RODBC** connects to "traditional" databases such as MySQL, PostgreSQL, Oracle and SQLite.

The function used to set-up a connection to an external database with **RODBC** is `odbcConnect`, which takes Data Source Name (`dsn`), User ID (`uid`) and password (`pwd`) as required arguments.

```
library("RODBC")

#open the ODBC connection
ch <- odbcConnect("ODBCDriverName")

##### Alternative ODBC connection for Microsoft SQL Server
ch <- odbcDriverConnect(
  "Driver=SQL Server; Server=servername\\instance; Database=databasename; UID=username; Pwd=password"
)

#run the query, store in a data frame
sqlResult <- sqlQuery(ch, "SELECT ...
                FROM ...
                WHERE ...
                ;")

#close the ODBC connection
odbcClose(ch)
```

A new development is the ability to interact with databases using exactly the same syntax used to interact with R objects stored in RAM. This innovation was made possible by **dplyr** (Wickham et al. 2022), an R

library for data processing that aims to provide a unified "front end" to perform a wide range of analysis tasks on datasets using a variety of "back ends" which do the number crunching. This is one of the main advantages of **dplyr**. It translates the data wrangling function from the **dplyr** package to `SQL` queries. Thus, your R code is translated into SQL and executed in the database, not in R. When working with databases, **dplyr** tries to be as lazy as possible:

- It never pulls data into R unless you explicitly ask for it.
- It delays doing any work until the last possible moment: it collects together everything you want to do and then sends it to the database in one step.

## 2.2 Introduction to tidy data with tidyr

> "Happy families are all alike; every unhappy family is unhappy in its own way."
> —- Leo Tolstoy

> "Tidy datasets are all alike, but every messy dataset is messy in its own way."
> —- Hadley Wickham

(excerpt from R for Data Science, Chapter 12)

The content of this chapter is based on the 2014 paper Tidy Data by Hadley Wickham published in the Journal of Statistical Software.

It is often said that 80% of data analysis is spent on the process of cleaning and preparing the data (Dasu and Johnson 2003). This process is often referred to as data wrangling. It includes many steps, such as outlier checking and data imputation, reformatting variables – especially dates, etc. The **tidyr** package (Wickham and Girlich 2022) provides functions for one of these tasks: *tidying* i.e., structuring datasets to facilitate analysis.

The principles of tidy data provide a standard way to organize data values within a dataset. A standard makes initial data cleaning easier because you do not need to start from scratch and reinvent the wheel every time. The tidy data standard has been designed to facilitate initial exploration and analysis of the data, and to simplify the development of data analysis tools that work well together.

The paper provides a set of principles of how to structure data, that is the fundamental approach for all packages in the **tidyverse**. We will also use the **dplyr** and the **ggplot2** (Wickham 2016) packages which share these common ideas.

To load all packages in the **tidyverse** you can use:

```
library("tidyverse")
```

However, you can also load the individual packages where and when you need them.

---

**NOTE**
Before we get started, remember the *pipe operator* introduced in 1.9.4.2.
The **tidyverse** packages make use the **magrittr** pipe `%>%`
so make sure you familiarize yourself with it!

---

### 2.2.1 Tidy data

Let us first define a few terms:

- A dataset is a collection of *values*, usually either numbers (if quantitative) or strings (if qualitative).
- Values are organized in two ways. Every value belongs to a *variable* and an *observation*.
- A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units.
- An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

Table 12: Typical presentation of experiment design dataset.

| | treatmenta | treatmentb |
|---|---|---|
| John Smith | - | 2 |
| Jane Doe | 16 | 11 |
| Mary Johnson | 3 | 1 |

Table 13: The same information as in Table 12 but structured differently.

| | John Smith | Jane Doe | Mary Johnson |
|---|---|---|---|
| treatmenta | - | 16 | 3 |
| treatmentb | 2 | 11 | 1 |

Tidy data is a standard way of mapping the *meaning* of a dataset to its *structure*. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Table 14: Tidy data version of experimental design data in Table 12.

| person | treatment | result |
|---|---|---|
| John Smith | a | - |
| Jane Doe | a | 16 |
| Mary Johnson | a | 3 |
| John Smith | b | 2 |
| Jane Doe | b | 11 |
| Mary Johnson | b | 1 |

Table 14 is the tidy version of Table 12. Each row represents an observation, the result of one treatment on one person, and each column is a variable. Note that in this experiment, the missing value represents an observation that should have been made, but was not, so it is important to keep it. Structural missing values, which represent measurements that cannot be made (e.g., the count of pregnant males) can be safely removed.

For a given dataset, it is usually easy to figure out what are observations and what are variables, but it is surprisingly difficult to precisely define variables and observations in general. For example, if the columns in the Table 12 were height and weight we would have been happy to call them variables. If the columns were height and width, it would be less clear cut, as we might think of height and width as values of a dimension variable.

Tidy data makes it easy for an analyst or a computer to extract needed variables because it provides a standard way of structuring a dataset. Compare Table 14 to Table 12: in Table 12 you need to use different strategies to extract different variables. This slows down the analysis and invites errors. If you consider how many data analysis operations involve all of the values in a variable (every aggregation function), you can see how important it is to extract these values in a simple, standard way. Tidy data is particularly well suited for vectorized programming languages like R, because the layout ensures that values of different variables from the same observation are always paired.

While the order of variables and observations does not affect analysis, a good ordering makes it easier to scan the raw values. One way of organizing variables is by their role in the analysis:

- are values fixed by the design of the data collection? Fixed variables describe the experimental design and are known in advance (such as `person` and `treatment` in Table 3). Computer scientists often call fixed variables dimensions, and statisticians usually denote them with subscripts on random variables.
- are they measured during the course of the experiment? Measured variables are what we actually measure in the study (such as `result` in Table 3).

Fixed variables should come first, followed by measured variables, each ordered so that related variables are contiguous. Rows can then be ordered by the first variable, breaking ties with the second and subsequent (fixed) variables.

### 2.2.2 Untidy data

In order to understand the structure of tidy data better it makes sense to look at examples of untidy data.

According to Hadley Wickham, the five most common problems with untidy datasets are:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

Let us have a look at the same underlying data but with different structures (example from the book R for Data Science). The four variables contained in this data are country, year, population, and cases We can notice that the different representations are not equally easy to use. One dataset, the tidy dataset `table1`, will be much easier to work with inside the **tidyverse**.

```
table1
#> # A tibble: 6 × 4
#>       country  year  cases population
#>         <chr> <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3      Brazil  1999  37737  172006362
#> 4      Brazil  2000  80488  174504898
#> 5       China  1999 212258 1272915272
#> 6       China  2000 213766 1280428583
```

```
table2
#> # A tibble: 12 × 4
#>       country  year       type     count
#>         <chr> <int>      <chr>     <int>
#> 1 Afghanistan  1999      cases       745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000      cases      2666
#> 4 Afghanistan  2000 population  20595360
#> 5      Brazil  1999      cases     37737
#> 6      Brazil  1999 population 172006362
#> # ... with 6 more rows

table3
#> # A tibble: 6 × 3
#>       country  year              rate
#> *       <chr> <int>             <chr>
#> 1 Afghanistan  1999      745/19987071
#> 2 Afghanistan  2000     2666/20595360
#> 3      Brazil  1999   37737/172006362
#> 4      Brazil  2000   80488/174504898
#> 5       China  1999 212258/1272915272
#> 6       China  2000 213766/1280428583
```

Or spread across two tibbles:

```
table4a
#> # A tibble: 3 × 3
#>       country `1999` `2000`
#> *       <chr>  <int>  <int>
#> 1 Afghanistan    745   2666
#> 2      Brazil  37737  80488
#> 3       China 212258 213766

table4b
#> # A tibble: 3 × 3
#>       country     `1999`     `2000`
#> *       <chr>      <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2      Brazil  172006362  174504898
#> 3       China 1272915272 1280428583
```

### 2.2.3 Tidying and reshaping data

The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

- One variable might be spread across multiple columns.
- One observation might be scattered across multiple rows.

For these purposes there are two functions in **tidyr**: `pivot_longer()` and `pivot_wider()` (these newer functions replace the functions `gather()` and `spread()`, which can still be used but are less flexible and no longer actively maintained).

- Two variables might be recorded in one column
- One variable might be recorded in two columns

For these purposes there are two functions in **tidyr**: `separate()` and `unite()`

**2.2.3.1 `pivot_longer()`** The function `pivot_longer()` pivots or "lengthens" data by taking multiple columns and collapsing them into key-value pairs, duplicating all other columns as needed. You use `pivot_longer()` when you notice that you have columns that are not variables. This operation is often described as making *wide* datasets *long* or *tall*.

```
pivot_longer(data, cols, names_to = "name", values_to = "value", ...)
```

In order to understand this function one must understand the following arguments:

- `data` A data frame to pivot.
- `cols`: Columns to pivot into longer format.
- `names_to`: A character vector specifying the new column or columns to create from the information stored in the column names of data specified by cols.
- `values_to`: A string specifying the name of the column to create from the data stored in cell values.

Consider the following example data set. The column names actually would correspond to the values of variable `year`:

```
table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

We can use `pivot_longer()` to make the data tidy

```
table4a %>%
  pivot_longer(col = c(`1999`, `2000`),
               names_to = "year", values_to = "cases")
```

```
# A tibble: 6 x 3
  country     year   cases
  <chr>       <chr>  <int>
1 Afghanistan 1999     745
2 Afghanistan 2000    2666
3 Brazil      1999   37737
4 Brazil      2000   80488
5 China       1999  212258
6 China       2000  213766
```

---

**Exercise**

The layout of the following data frame called `stocks` may be accessible to the human eye, however it is not tidy according to the rules laid out earlier.

```
# From http://stackoverflow.com/questions/1181060
stocks <- tibble(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)
```

In this example X, Y, and Z are not different variables but values of a variable called `stock`. How can we tidy the data?

```
stocks %>%
  pivot_longer(cols = c("X", "Y", "Z"),
               names_to = "stock", values_to = "price")
```

```
# A tibble: 30 x 3
   time       stock  price
   <date>     <chr>  <dbl>
 1 2009-01-01 X      2.02
 2 2009-01-01 Y      2.38
 3 2009-01-01 Z      1.03
 4 2009-01-02 X      0.738
 5 2009-01-02 Y     -1.79
 6 2009-01-02 Z      6.53
 7 2009-01-03 X     -1.69
 8 2009-01-03 Y     -1.53
 9 2009-01-03 Z      3.48
10 2009-01-04 X      0.478
# ... with 20 more rows
```

---

**2.2.3.2 `pivot_wider()`** Function `pivot_wider()` "widens" or spreads data, increasing the number of columns and decreasing the number of rows. It is the inverse transformation of `pivot_longer()`. You use it when an observation is scattered across multiple rows. This operation is often described as making *long* datasets *wide*.

```
pivot_wider(data, id_cols = NULL, id_expand = FALSE,
            names_from = "name", values_from = "value", ...)
```

The most important arguments are

- `data`: A data frame to pivot

- `names_from`, `values_from`: A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`).

Consider the data set:

```
table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

An observation is a country in a year, but each observation is spread across two rows.

```
table2 %>%
  pivot_wider(names_from = "type", values_from = "count")
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

---

**Exercise**

Why does this code fail?

```
table4a %>%
  pivot_longer(c(1999, 2000),
               names_to = "year", values_to = "cases")
```

---

**Exercise**

Consider the stocks data in long format from last exercise.

```
stocksm <- stocks %>%
  pivot_longer(cols = c("X", "Y", "Z"),
               names_to = "stock", values_to = "price")
```

In our analysis we want to consider a stock an observation and the price at a given point in time is considered a variable. How can we use the `pivot_wider()` function to create the proper formatting?

```
stocksm %>%
  pivot_wider(names_from = "time", values_from = "price")
```

```
# A tibble: 3 x 11
  stock `2009-01-01` `2009-01-02` `2009-01-03` `2009-01-04` `2009-01-05`
  <chr>        <dbl>        <dbl>        <dbl>        <dbl>        <dbl>
1 X             2.02        0.738        -1.69        0.478       0.0747
2 Y             2.38       -1.79         -1.53        1.97        1.83
3 Z             1.03        6.53          3.48        4.87        6.88
# ... with 5 more variables: `2009-01-06` <dbl>, `2009-01-07` <dbl>,
#   `2009-01-08` <dbl>, `2009-01-09` <dbl>, `2009-01-10` <dbl>
```

---

**2.2.3.3 `separate()`** This function pulls apart one column into multiple columns by splitting wherever a separator character appears.

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE,
  convert = FALSE, extra = "warn", fill = "warn", ...)
```

The most important arguments are

- `data` the data frame
- `col` the column name or position
- `into` the names of the new variables to create as character vector
- `sep` the separator between columns.

Consider `table3` above:

```
table3
```

```
# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

The rate contains both `cases` and `population` variables. We, thus, need to split it into two variables.

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep  = "/")
```

```
# A tibble: 6 x 4
  country      year cases  population
  <chr>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

#### 2.2.3.4 `unite()`

This is the inverse of `separate()`. It combines multiple columns into a single column.

```
unite(data, col, ..., sep = "_", remove = TRUE)
```

Its most important arguments are

- `data` the data frame
- `col` the name of the new column, as a string or symbol.
- `...` A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with x:z, exclude y with -y.
- `sep` Separator to use between values.

Let's assume we want to change `table1` into `table3`:

```
table1
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

The cases and population columns can be united with the `unite()` function.

```
table1 %>%
  unite(rate, cases, population, sep="/")
```

```
# A tibble: 6 x 3
  country      year rate
  <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

**2.2.3.5 Missing values** Changing the representation of a dataset (from wide to long and viceversa) brings up the issue of how missings values should be dealt with. As mentioned before, a value can be missing

- Explicitly, i.e. flagged with NA.
- Implicitly, i.e. simply not present in the data.

Let's again look at one example:

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

We have an *explicit* missing value for year 2015 in quarter 4. But at a closer inspection we actually see that we have one more *implicit* missing value 2016 quarter 1.

A function for making missing values explicit in tidy data is `complete()`.

```
stocks %>%
  complete(year, qtr)
```

```
# A tibble: 8 x 3
   year   qtr return
  <dbl> <dbl>  <dbl>
1  2015     1   1.88
2  2015     2   0.59
3  2015     3   0.35
4  2015     4  NA
5  2016     1  NA
6  2016     2   0.92
7  2016     3   0.17
8  2016     4   2.66
```

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

```
stocks_wide <- pivot_wider(stocks, names_from = "qtr", values_from = "return")
stocks_wide
```

```
# A tibble: 2 x 5
   year   `1`   `2`   `3`   `4`
  <dbl> <dbl> <dbl> <dbl> <dbl>
1  2015  1.88  0.59  0.35 NA
2  2016 NA     0.92  0.17  2.66
```

If we are not interesting in keeping any missing values in the long data format, we can set the argument `values_drop_na = TRUE` in `pivot_longer()`:

```
pivot_longer(stocks_wide, cols = c(`1`, `2`, `3`, `4`),
             names_to = "qtr", values_to = "return",
             values_drop_na = TRUE)
```

```
# A tibble: 6 x 3
   year qtr   return
  <dbl> <chr>  <dbl>
1  2015 1       1.88
2  2015 2       0.59
3  2015 3       0.35
4  2016 2       0.92
5  2016 3       0.17
6  2016 4       2.66
```

## 2.3 Basic data transformation with dplyr

This lesson is a selection of the 5th chapter of the book R for Data Science.

The **dplyr** package contains a number of functions that help in transforming data. Most important functions perform the following operations:

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarise()`).

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the *verbs* for a language of data manipulation.

All verbs work similarly:

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
- The result is a new data frame.

Together, these properties make it easy to chain together multiple simple steps to achieve a complex result.

### 2.3.1 Data: nycflights13

To explore the basic data manipulation verbs of **dplyr**, we will use the data `nycflights13::flights`. This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics and is documented in `?flights`.

```
library("nycflights13")
str(flights)
```

```
tibble [336,776 x 19] (S3: tbl_df/tbl/data.frame)
 $ year          : int [1:336776] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
 $ month         : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
 $ day           : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
 $ dep_time      : int [1:336776] 517 533 542 544 554 554 555 557 557 558 ...
 $ sched_dep_time: int [1:336776] 515 529 540 545 600 558 600 600 600 600 ...
 $ dep_delay     : num [1:336776] 2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
 $ arr_time      : int [1:336776] 830 850 923 1004 812 740 913 709 838 753 ...
 $ sched_arr_time: int [1:336776] 819 830 850 1022 837 728 854 723 846 745 ...
 $ arr_delay     : num [1:336776] 11 20 33 -18 -25 12 19 -14 -8 8 ...
 $ carrier       : chr [1:336776] "UA" "UA" "AA" "B6" ...
 $ flight        : int [1:336776] 1545 1714 1141 725 461 1696 507 5708 79 301 ...
 $ tailnum       : chr [1:336776] "N14228" "N24211" "N619AA" "N804JB" ...
 $ origin        : chr [1:336776] "EWR" "LGA" "JFK" "JFK" ...
 $ dest          : chr [1:336776] "IAH" "IAH" "MIA" "BQN" ...
 $ air_time      : num [1:336776] 227 227 160 183 116 150 158 53 140 138 ...
 $ distance      : num [1:336776] 1400 1416 1089 1576 762 ...
 $ hour          : num [1:336776] 5 5 5 5 6 5 6 6 6 6 ...
 $ minute        : num [1:336776] 15 29 40 45 0 58 0 0 0 0 ...
 $ time_hour     : POSIXct[1:336776], format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

### 2.3.2 Filter rows with `filter()`

Function `filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
```

```
# A tibble: 842 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
```

```
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

or with the pipe operator:

```
flights %>%
  filter(month == 1, day == 1)
```

```
# A tibble: 842 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

---

**Exercises**

1. Find all flights that:

a. Had an arrival delay of two or more hours
b. Flew to Houston (IAH or HOU)
c. Were operated by United (UA), American (AA), or Delta (DL)
d. Departed in summer (July, August, and September)
e. Arrived more than two hours late, but didn't leave late
f. Were delayed by at least an hour, but made up over 30 minutes in flight
g. Departed between midnight and 6am (included)

2. Another useful dplyr filtering helper is `between()`. What does it do? Can you use it to simplify the code needed to answer the previous challenges?

3. How many flights have a missing `dep_time`? What other variables are missing? What might these rows represent?

---

### 2.3.3 Arrange rows with `arrange()`

Function `arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, year, month, day)
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Use `desc()` to re-order by a column in descending order:

```
arrange(flights, desc(arr_delay))
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     9      641            900      1301     1242           1530
 2  2013     6    15     1432           1935      1137     1607           2120
 3  2013     1    10     1121           1635      1126     1239           1810
 4  2013     9    20     1139           1845      1014     1457           2210
 5  2013     7    22      845           1600      1005     1044           1815
 6  2013     4    10     1100           1900       960     1342           2211
 7  2013     3    17     2321            810       911      135           1020
 8  2013     7    22     2257            759       898      121           1026
 9  2013    12     5      756           1700       896     1058           2020
10  2013     5     3     1133           2055       878     1250           2215
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Missing values are always sorted at the end:

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
# A tibble: 3 x 1
      x
  <dbl>
1     2
2     5
3    NA
arrange(df, desc(x))
# A tibble: 3 x 1
```

```
      x
  <dbl>
1     5
2     2
3    NA
```

---

**Exercises**

1. How could you use `arrange()` to sort all missing values to the top? (Hint: use `is.na()`).
2. Sort flights to find the most delayed flights. Find the flights that left earliest.
3. Sort flights to find the fastest flights.
4. Which flights travelled the longest? Which travelled the shortest?

---

### 2.3.4 Select columns with `select()`

It is not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

```
select(flights, year, month, day)
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# ... with 336,766 more rows
select(flights, year:day)
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
```

```
 9  2013     1     1
10  2013     1     1
# ... with 336,766 more rows
select(flights, -(year:day))
# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
      <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
 1      517            515         2      830            819        11 UA
 2      533            529         4      850            830        20 UA
 3      542            540         2      923            850        33 AA
 4      544            545        -1     1004           1022       -18 B6
 5      554            600        -6      812            837       -25 DL
 6      554            558        -4      740            728        12 UA
 7      555            600        -5      913            854        19 B6
 8      557            600        -3      709            723       -14 EV
 9      557            600        -3      838            846        -8 B6
10      558            600        -2      753            745         8 AA
# ... with 336,766 more rows, and 9 more variables: flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")`: matches names that begin with "abc."
- `ends_with("xyz")`: matches names that end with "xyz."
- `contains("ijk")`: matches names that contain "ijk."
- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters.
- `num_range("x", 1:3)` matches x1, x2 and x3.

See `?select` for more details.

`select()` can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use `rename()`, which is a variant of `select()` that keeps all the variables that aren't explicitly mentioned:

```
rename(flights, tail_num = tailnum)
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tail_num <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame.

```
select(flights, time_hour, air_time, everything())
```

```
# A tibble: 336,776 x 19
   time_hour           air_time  year month   day dep_time sched_dep_time
   <dttm>                 <dbl> <int> <int> <int>    <int>          <int>
 1 2013-01-01 05:00:00      227  2013     1     1      517            515
 2 2013-01-01 05:00:00      227  2013     1     1      533            529
 3 2013-01-01 05:00:00      160  2013     1     1      542            540
 4 2013-01-01 05:00:00      183  2013     1     1      544            545
 5 2013-01-01 06:00:00      116  2013     1     1      554            600
 6 2013-01-01 05:00:00      150  2013     1     1      554            558
 7 2013-01-01 06:00:00      158  2013     1     1      555            600
 8 2013-01-01 06:00:00       53  2013     1     1      557            600
 9 2013-01-01 06:00:00      140  2013     1     1      557            600
10 2013-01-01 06:00:00      138  2013     1     1      558            600
# ... with 336,766 more rows, and 12 more variables: dep_delay <dbl>,
#   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, distance <dbl>,
#   hour <dbl>, minute <dbl>
```

---

**Exercises**

1. Use different options to select `dep_time`, `dep_delay`, `arr_time`, and `arr_delay` from `flights`.
2. What happens if you include the name of a variable multiple times in a `select()` call? What about if we select it twice using named indexing i.e., `flights[, c("dep_time", "dep_time")]`?
3. Does the result of running the following code surprise you?

```
select(flights, contains("TIME"))
```

```
# A tibble: 336,776 x 6
   dep_time sched_dep_time arr_time sched_arr_time air_time time_hour
      <int>          <int>    <int>          <int>    <dbl> <dttm>
 1      517            515      830            819      227 2013-01-01 05:00:00
 2      533            529      850            830      227 2013-01-01 05:00:00
 3      542            540      923            850      160 2013-01-01 05:00:00
 4      544            545     1004           1022      183 2013-01-01 05:00:00
 5      554            600      812            837      116 2013-01-01 06:00:00
 6      554            558      740            728      150 2013-01-01 05:00:00
 7      555            600      913            854      158 2013-01-01 06:00:00
 8      557            600      709            723       53 2013-01-01 06:00:00
 9      557            600      838            846      140 2013-01-01 06:00:00
10      558            600      753            745      138 2013-01-01 06:00:00
# ... with 336,766 more rows
```

How do the `select` helpers deal with upper case by default? How can you change that default?

```
select(flights, contains("TIME", ignore.case = FALSE))
```

### 2.3.5 Add new variables with `mutate()`

Besides selecting sets of existing columns, it is often useful to add new columns that are functions of existing columns. Function `mutate()` can be used for this purpose:

```
flights_sml <- flights %>%
  select(
    year:day,
    ends_with("delay"),
    distance,
    air_time
  )
flights_sml %>%
  mutate(
    gain = arr_delay - dep_delay,
    speed = distance / air_time * 60
  )
```

```
# A tibble: 336,776 x 9
    year month   day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9  370.
 2  2013     1     1         4        20     1416      227    16  374.
 3  2013     1     1         2        33     1089      160    31  408.
 4  2013     1     1        -1       -18     1576      183   -17  517.
 5  2013     1     1        -6       -25      762      116   -19  394.
 6  2013     1     1        -4        12      719      150    16  288.
 7  2013     1     1        -5        19     1065      158    24  404.
 8  2013     1     1        -3       -14      229       53   -11  259.
 9  2013     1     1        -3        -8      944      140    -5  405.
10  2013     1     1        -2         8      733      138    10  319.
# ... with 336,766 more rows
```

Note that you can refer to columns that you've just created (given that we are worling with tibbles):

```
flights_sml %>%
  mutate(
    gain = arr_delay - dep_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours
  )
```

```
# A tibble: 336,776 x 10
    year month   day dep_delay arr_delay distance air_time  gain hours
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9 3.78
 2  2013     1     1         4        20     1416      227    16 3.78
 3  2013     1     1         2        33     1089      160    31 2.67
 4  2013     1     1        -1       -18     1576      183   -17 3.05
```

```
 5  2013     1     1        -6       -25      762      116   -19 1.93
 6  2013     1     1        -4        12      719      150    16 2.5
 7  2013     1     1        -5        19     1065      158    24 2.63
 8  2013     1     1        -3       -14      229       53   -11 0.883
 9  2013     1     1        -3        -8      944      140    -5 2.33
10  2013     1     1        -2         8      733      138    10 2.3
# ... with 336,766 more rows, and 1 more variable: gain_per_hour <dbl>
```

If you only want to keep the new variables, use `transmute()`:

```
flights %>%
  transmute(
    gain = arr_delay - dep_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours
  )
```

```
# A tibble: 336,776 x 3
    gain hours gain_per_hour
   <dbl> <dbl>         <dbl>
 1     9 3.78           2.38
 2    16 3.78           4.23
 3    31 2.67          11.6
 4   -17 3.05          -5.57
 5   -19 1.93          -9.83
 6    16 2.5            6.4
 7    24 2.63           9.11
 8   -11 0.883        -12.5
 9    -5 2.33          -2.14
10    10 2.3            4.35
# ... with 336,766 more rows
```

There are many functions for creating new variables that you can use with `mutate()`. The key property is that the function must be vectorised: it must take a vector of values as input, return a vector with the same number of values as output.

---

**Exercises**

1. Currently, `dep_time` and `sched_dep_time` are convenient to look at, but hard to compute with because they're not really continuous variables. Convert them to a more convenient representation of number of minutes since midnight.

2. Compare `air_time` with `arr_time - dep_time`. What do you see?

3. Compare dep_time, sched_dep_time, and dep_delay. How would you expect those three numbers to be related?

---

### 2.3.6 Grouped summaries with `summarise()`

The last key verb is `summarise()`. It collapses a data frame to a single row:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

or with the pipe operator:

```
flights %>%
  summarise(delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

`summarise()` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the **dplyr** verbs on a grouped data frame they will be automatically applied group-wise. For example, if we applied exactly the same code to a data frame grouped by date, we get the average delay per date:

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
`summarise()` has grouped output by 'year', 'month'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

Together `group_by()` and `summarise()` provide one of the tools that you'll use most commonly when working with dplyr: grouped summaries.

### 2.3.7 Combining multiple operations with the pipe

The elegance of these transformation functions manifests when used in combination with the pipe operator:

```
flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

```
# A tibble: 96 x 4
   dest  count  dist delay
   <chr> <int> <dbl> <dbl>
 1 ABQ     254 1826   4.38
 2 ACK     265  199   4.85
 3 ALB     439  143  14.4
 4 ATL   17215  757. 11.3
 5 AUS    2439 1514.  6.02
 6 AVL     275  584.  8.00
 7 BDL     443  116   7.05
 8 BGR     375  378   8.03
 9 BHM     297  866. 16.9
10 BNA    6333  758. 11.8
# ... with 86 more rows
```

### 2.3.8 Useful summary functions

- Measures of location: `mean()` and `median()`

```
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average positive delay
  )
```

```
`summarise()` has grouped output by 'year', 'month'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day avg_delay1 avg_delay2
   <int> <int> <int>      <dbl>      <dbl>
 1  2013     1     1     12.7         32.5
 2  2013     1     2     12.7         32.0
 3  2013     1     3      5.73        27.7
 4  2013     1     4     -1.93        28.3
 5  2013     1     5     -1.53        22.6
```

```
 6  2013     1     6      4.24        24.4
 7  2013     1     7     -4.95        27.8
 8  2013     1     8     -3.23        20.8
 9  2013     1     9     -0.264       25.6
10  2013     1    10     -5.90        27.3
# ... with 355 more rows
```

- Measure of spread: `sd(x)`, `IQR(x)`, `mad(x)`. The empirical standard deviation or sd for short, is the standard measure of spread. The interquartile range `IQR()` and median absolute deviation `mad(x)` are robust equivalents that may be more useful if you have outliers.

```
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(dest) %>%
  summarise(distance_sd = sd(distance),
            distance_iqr = IQR(distance),
            distance_mad = mad(distance)) %>%
  arrange(desc(distance_sd))
```

```
# A tibble: 104 x 4
   dest  distance_sd distance_iqr distance_mad
   <chr>       <dbl>        <dbl>        <dbl>
 1 EGE         10.5            21         1.48
 2 SAN         10.4            21         0
 3 SFO         10.2            21         0
 4 HNL         10.0            20         0
 5 SEA          9.98           20         0
 6 LAS          9.91           21         0
 7 PDX          9.87           20         0
 8 PHX          9.86           20         0
 9 LAX          9.66           21         0
10 IND          9.46           20         0
# ... with 94 more rows
```

- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`. For example, we can find the first and last departure for each day:

```
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(year, month, day) %>%
  summarise(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )
```

```
`summarise()` has grouped output by 'year', 'month'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day first_dep last_dep
   <int> <int> <int>     <int>    <int>
```

```
 1  2013     1     1       517     2356
 2  2013     1     2        42     2354
 3  2013     1     3        32     2349
 4  2013     1     4        25     2358
 5  2013     1     5        14     2357
 6  2013     1     6        16     2355
 7  2013     1     7        49     2359
 8  2013     1     8       454     2351
 9  2013     1     9         2     2252
10  2013     1    10         3     2320
# ... with 355 more rows
```

- counts: `n()` takes no arguments, and returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct (unique) values, use `n_distinct(x)`.

```
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
```

```
# A tibble: 104 x 2
   dest  carriers
   <chr>    <int>
 1 ATL          7
 2 BOS          7
 3 CLT          7
 4 ORD          7
 5 TPA          7
 6 AUS          6
 7 DCA          6
 8 DTW          6
 9 IAD          6
10 MSP          6
# ... with 94 more rows
```

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll up a dataset:

```
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
(per_month <- summarise(per_day, flights = sum(flights)))
(per_year  <- summarise(per_month, flights = sum(flights)))
```

---

**Exercise:**

1. Which carrier has the biggest delays?
2. Which flight is always at least 10 minutes late?

---

### 2.3.9 Relational data: merging data sets

It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

To work with relational data you need verbs that work with pairs of tables. There are two main families of verbs in **dplyr** designed to work with relational data:

- Mutating joins, which add new variables to one data frame from matching observations in another.
- Filtering joins, which filter observations from one data frame based on whether or not they match an observation in the other table.

The datasets used here are from the **nycflights13** package, which contains four tibbles that are related to the `flights` table we used previously.

```
flights
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
airlines
```

```
# A tibble: 16 x 2
   carrier name
   <chr>   <chr>
 1 9E      Endeavor Air Inc.
 2 AA      American Airlines Inc.
 3 AS      Alaska Airlines Inc.
 4 B6      JetBlue Airways
 5 DL      Delta Air Lines Inc.
 6 EV      ExpressJet Airlines Inc.
 7 F9      Frontier Airlines Inc.
 8 FL      AirTran Airways Corporation
 9 HA      Hawaiian Airlines Inc.
10 MQ      Envoy Air
11 OO      SkyWest Airlines Inc.
12 UA      United Air Lines Inc.
```

```
13 US      US Airways Inc.
14 VX      Virgin America
15 WN      Southwest Airlines Co.
16 YV      Mesa Airlines Inc.
```

```
airports
```

```
# A tibble: 1,458 x 8
   faa   name                            lat    lon   alt    tz dst   tzone
   <chr> <chr>                         <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
 1 04G   Lansdowne Airport              41.1  -80.6  1044    -5 A     America/~
 2 06A   Moton Field Municipal Airport  32.5  -85.7   264    -6 A     America/~
 3 06C   Schaumburg Regional            42.0  -88.1   801    -6 A     America/~
 4 06N   Randall Airport                41.4  -74.4   523    -5 A     America/~
 5 09J   Jekyll Island Airport          31.1  -81.4    11    -5 A     America/~
 6 0A9   Elizabethton Municipal Airport 36.4  -82.2  1593    -5 A     America/~
 7 0G6   Williams County Airport        41.5  -84.5   730    -5 A     America/~
 8 0G7   Finger Lakes Regional Airport  42.9  -76.8   492    -5 A     America/~
 9 0P2   Shoestring Aviation Airfield   39.8  -76.6  1000    -5 U     America/~
10 0S9   Jefferson County Intl          48.1 -123.    108    -8 A     America/~
# ... with 1,448 more rows
```

```
planes
```

```
# A tibble: 3,322 x 9
   tailnum  year type              manufacturer model engines seats speed engine
   <chr>   <int> <chr>             <chr>        <chr>   <int> <int> <int> <chr>
 1 N10156   2004 Fixed wing multi~ EMBRAER      EMB-~       2    55    NA Turbo~
 2 N102UW   1998 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
 3 N103US   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
 4 N104UW   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
 5 N10575   2002 Fixed wing multi~ EMBRAER      EMB-~       2    55    NA Turbo~
 6 N105UW   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
 7 N107US   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
 8 N108UW   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
 9 N109UW   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
10 N110UW   1999 Fixed wing multi~ AIRBUS INDU~ A320~       2   182    NA Turbo~
# ... with 3,312 more rows
```

```
weather
```

```
# A tibble: 26,115 x 15
   origin  year month   day  hour  temp  dewp humid wind_dir wind_speed
   <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>
 1 EWR     2013     1     1     1  39.0  26.1  59.4      270      10.4
 2 EWR     2013     1     1     2  39.0  27.0  61.6      250       8.06
 3 EWR     2013     1     1     3  39.0  28.0  64.4      240      11.5
 4 EWR     2013     1     1     4  39.9  28.0  62.2      250      12.7
 5 EWR     2013     1     1     5  39.0  28.0  64.4      260      12.7
 6 EWR     2013     1     1     6  37.9  28.0  67.2      240      11.5
 7 EWR     2013     1     1     7  39.0  28.0  64.4      240      15.0
 8 EWR     2013     1     1     8  39.9  28.0  62.2      250      10.4
```

```
 9 EWR     2013     1     1     9  39.9  28.0  62.2      260      15.0
10 EWR     2013     1     1    10  41    28.0  59.6      260      13.8
# ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>,
#   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dttm>
```

---

**Exercise**

Inspect how the 5 tables are connected.

---

**2.3.9.1 Keys** The variables used to connect each pair of tables are called **keys**. A **key** is a variable (or set of variables) that uniquely identifies an observation. For example, * each plane in `planes` is uniquely identified by its `tailnum`. * to identify an observation in `weather` you need five variables: `year`, `month`, `day`, `hour`, and `origin`, so the key consists to 5 variables.

To check whether the identified primary keys are indeed correct, there should be only one observation for each key in the table:

```
planes %>%
  count(tailnum) %>% # how many rows per key
  filter(n > 1) # are there any with more rows per key?
```

```
# A tibble: 0 x 2
# ... with 2 variables: tailnum <chr>, n <int>
```

```
weather %>%
  count(year, month, day, hour, origin) %>% # how many rows per key
  filter(n > 1) # are there any with more rows per key?
```

```
# A tibble: 3 x 6
   year month   day  hour origin     n
  <int> <int> <int> <int> <chr>  <int>
1  2013    11     3     1 EWR        2
2  2013    11     3     1 JFK        2
3  2013    11     3     1 LGA        2
```

For `weather`, one would need to further dig into these 3 cases where `year`, `month`, `day`, `hour`, and `origin` are not primary keys.

There are two types of keys:

- A primary key uniquely identifies an observation in its own table. For example, `planes$tailnum` is a primary key because it uniquely identifies each plane in the planes table.
- A foreign key uniquely identifies an observation in another table. For example, `flights$tailnum` is a foreign key because it appears in the `flights` table where it matches each flight to a unique plane.

**2.3.9.2 Mutating joins** A mutating join allows you to combine variables from two tables. It first matches observations by their keys, then copies across variables from one table to the other.

Types of joins:

- inner join: keeps observations that appear in both tables
- outer join: outer join keeps observations that appear in at least one of the tables. There are three types of outer joins:
  - left join keeps all observations in x.
  - right join keeps all observations in y.
  - full join keeps all observations in x and y

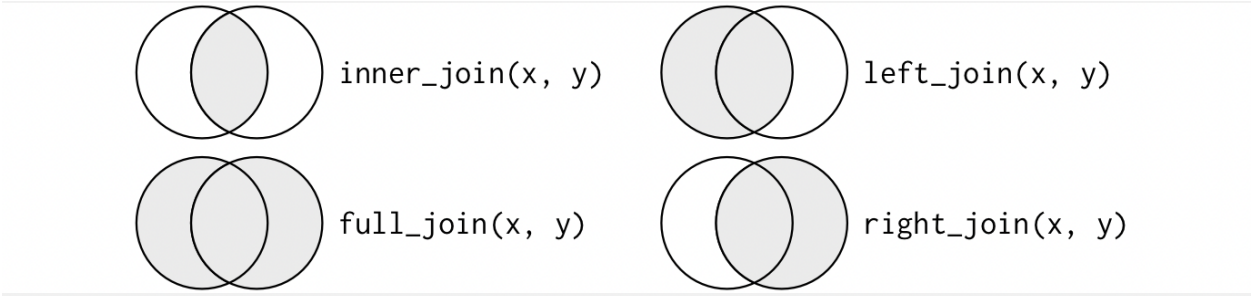



Figure 2: Source: https://r4ds.had.co.nz/relational-data.html

Function `base::merge()` can perform all four types of mutating join:

| **dplyr** merge | `base::merge()` |
|---|---|
| `inner_join(x, y)` | `merge(x, y)` |
| `left_join(x, y)` | `merge(x, y, all.x = TRUE)` |
| `right_join(x, y)` | `merge(x, y, all.y = TRUE)` |
| `full_join(x, y)` | `merge(x, y, all.x = TRUE, all.y = TRUE)` |

As an example consider two tibbles which contain grades for students in two different courses. The key for both tables in the student names:

```
x <- tibble(student = c("Jim","Sarah","Mike"),
            grade_x = c(1, 4, 2))
y <- tibble(student = c("Jim","Sarah","Julia"),
            grade_y = c(1, 5, 1),
            majorS = c("Epi","PH","Epi"))
```

Inner join will contain students in both tables only (i.e., Jim and Sarah):

```
x %>%
  inner_join(y, by = "student")
```

```
# A tibble: 2 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Sarah         4       5 PH
```

Left join will keep all students in `x`:

```
left_join(x, y, by = "student")
```

```
# A tibble: 3 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Sarah         4       5 PH
3 Mike          2      NA <NA>
```

Right join will keep all students in `y`:

```
right_join(x, y, by = "student")
```

```
# A tibble: 3 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Sarah         4       5 PH
3 Julia        NA       1 Epi
```

Full join will keep all students in `x` and all students in `y`:

```
full_join(x, y, by = "student")
```

```
# A tibble: 4 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Sarah         4       5 PH
3 Mike          2      NA <NA>
4 Julia        NA       1 Epi
```

Assume the key column in table `y` would be named `student_name`.

```
y <- tibble(student_name = c("Jim","Sarah","Julia"),
            grade_y = c(1, 5, 1),
            majorS = c("Epi","PH","Epi"))
```

Then the join call would be

```
x %>%
  inner_join(y, by = c("student" = "student_name"))
```

```
# A tibble: 2 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Sarah         4       5 PH
```

**Duplicate keys** So far we assumed that the keys are unique. But that's not always the case. There are two possibilities:

1. One table has duplicate keys. This is useful when you want to add in additional information as there is typically a one-to-many relationship. Assume that table `x` contains grades in subject `x` for both mid-terms and final-terms.

```
x <- tibble(student = c("Jim","Jim","Sarah","Sarah", "Mike"),
            grade_x = c(1, 2, 4, 3, 2))
y <- tibble(student = c("Jim","Sarah","Julia"),
            grade_y = c(1, 5, 1),
            majorS = c("Epi","PH","Epi"))
left_join(x, y, by = "student")
```

```
# A tibble: 5 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Jim           2       1 Epi
3 Sarah         4       5 PH
4 Sarah         3       5 PH
5 Mike          2      NA <NA>
```

2.Both tables have duplicate keys. This is usually an error because in neither table do the keys uniquely identify an observation. When you join duplicated keys, you get all possible combinations, the Cartesian product:

```
x <- tibble(student = c("Jim","Jim","Sarah","Sarah", "Mike"),
            grade_x = c(1, 2, 4, 3, 2))
y <- tibble(student = c("Jim","Sarah","Sarah","Julia"),
            grade_y = c(1, 5, 4, 1),
            majorS = c("Epi","PH", "PH", "Epi"))
left_join(x, y, by = "student")
```

```
# A tibble: 7 x 4
  student grade_x grade_y majorS
  <chr>     <dbl>   <dbl> <chr>
1 Jim           1       1 Epi
2 Jim           2       1 Epi
3 Sarah         4       5 PH
4 Sarah         4       4 PH
5 Sarah         3       5 PH
6 Sarah         3       4 PH
7 Mike          2      NA <NA>
```

**2.3.9.3 Filtering joins** Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two types:

- `semi_join(x, y)` keeps all observations in `x` that have a match in `y`.
- `anti_join(x, y)` drops all observations in `x` that have a match in `y`.

Semi-joins are useful for matching filtered summary tables back to the original rows. For example, imagine you have found the top ten most popular destinations in flights:

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest
```

```
# A tibble: 10 x 2
   dest      n
   <chr> <int>
 1 ORD   17283
 2 ATL   17215
 3 LAX   16174
 4 BOS   15508
 5 MCO   14082
 6 CLT   14064
 7 SFO   13331
 8 FLL   12055
 9 MIA   11728
10 DCA    9705
```

If you want to find each flight that went to one of those destinations you can use `semi_join`

```
flights %>%
  semi_join(top_dest)
```

```
Joining, by = "dest"
```

```
# A tibble: 141,145 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      542            540         2      923            850
 2  2013     1     1      554            600        -6      812            837
 3  2013     1     1      554            558        -4      740            728
 4  2013     1     1      555            600        -5      913            854
 5  2013     1     1      557            600        -3      838            846
 6  2013     1     1      558            600        -2      753            745
 7  2013     1     1      558            600        -2      924            917
 8  2013     1     1      558            600        -2      923            937
 9  2013     1     1      559            559         0      702            706
10  2013     1     1      600            600         0      851            858
# ... with 141,135 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Note that filtering joins never duplicate rows like mutating joins do. This is the main difference between `semi_join` and `inner_join`.

The inverse of a semi-join is an anti-join. An anti-join keeps the rows that don't have a match:

```
flights %>%
  anti_join(top_dest) # contains all flights to dest other than top 10
```

```
Joining, by = "dest"
```

```
# A tibble: 195,631 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      544            545        -1     1004           1022
 4  2013     1     1      557            600        -3      709            723
 5  2013     1     1      558            600        -2      849            851
 6  2013     1     1      558            600        -2      853            856
 7  2013     1     1      559            600        -1      941            910
 8  2013     1     1      559            600        -1      854            902
 9  2013     1     1      601            600         1      844            850
10  2013     1     1      602            610        -8      812            820
# ... with 195,621 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

## 2.4 More on data transformation

Further steps, in addition to what we have seen so far, are often needed to bring data into a usable format. Such steps involve transforming character or numeric columns to dates, reordering or collapsing factor levels or dealing with strings. Here we will look into factors and date-times. For more information on how to handle strings with package **stringr** (Wickham 2019) and on regular expressions see Chapter 14, R for Data Science.

### 2.4.1 `case_when`

The `case_when` function allows you to vectorize multiple if and if else statements. It is the **tidyverse** equivalent of `switch` but has a more "robust" behavior (remember, using `switch` is recommended with character inputs).

As an example, let's make a new variable in data set `flights` which categorizes departures into morning (05:01-11:00), noon (11:01-13:00), afternoon (13:01-18:00), evening (18:01-23:00), night (23:01-05:00).

```
flights_new <- flights %>%
  mutate(new_var = case_when(
    dep_time > 500  & dep_time <= 1100 ~ "morning",
    dep_time > 1100 & dep_time <= 1300 ~ "noon",
    dep_time > 1300 & dep_time <= 1800 ~ "afternoon",
    dep_time > 1800 & dep_time <= 2300 ~ "evening",
    dep_time > 2300 | dep_time <=  500 ~ "night"
  ))
```

### 2.4.2 Factors and package forcats

To cast a (character) variable to a factor, you can use the `as.factor` function we learned in Chapter 1.

```
flights_new <- flights_new %>%
  mutate(new_var = as.factor(new_var))

levels(flights_new$new_var)
```

```
[1] "afternoon" "evening"   "morning"   "night"     "noon"
```

We can see that the levels of the factor are ordered in a alphanumeric fashion.

For more functionalities on dealing with factors, the **tidyverse** contains the **forcats** package (Wickham 2021).

Let's look at the average arrival and departure delay for the five times of the day.

```
flights_new %>%
  group_by(new_var) %>%
  summarise(
    delay_departure = mean(dep_delay, na.rm = TRUE),
    delay_arrival   = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
```

```
# A tibble: 6 x 4
  new_var   delay_departure delay_arrival      n
  <fct>               <dbl>         <dbl>  <int>
1 afternoon           11.9           6.58 104679
2 evening             28.8          22.5   73776
3 morning              1.24         -4.58 113075
4 night              105.           97.8    4068
5 noon                 6.58          1.15  32923
6 <NA>               NaN           NaN      8255
```

Ideally, we would like the table to be ordered by time of day not alphanumerically but chronologically. For this, one would need to reorder the levels of the factor variable. For this `fct_relevel` can be used.

```
library("forcats")
flights_summary <- flights_new %>%
  mutate(new_var = fct_relevel(new_var, "morning", "noon")) %>%
  group_by(new_var) %>%
  summarise(
    delay_departure = mean(dep_delay, na.rm = TRUE),
    delay_arrival   = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
flights_summary
```

```
# A tibble: 6 x 4
  new_var   delay_departure delay_arrival      n
  <fct>               <dbl>         <dbl>  <int>
1 morning              1.24         -4.58 113075
2 noon                 6.58          1.15  32923
3 afternoon           11.9           6.58 104679
```

```
4 evening             28.8          22.5   73776
5 night              105.           97.8    4068
6 <NA>               NaN           NaN      8255
```

One can also order a summary statistic. This is useful in some cases for plotting. For example, we want the levels to be ordered by their frequency, `fct_infreq()` can be used.

```
flights_new %>%
  mutate(new_var = fct_infreq(new_var))  %>%
  count(new_var)
```

```
# A tibble: 6 x 2
  new_var        n
  <fct>      <int>
1 morning   113075
2 afternoon 104679
3 evening    73776
4 noon       32923
5 night       4068
6 <NA>        8255
```

One can also order the levels by a summary statistic. This is useful in some cases for plotting. For example, we want the time of day to be ordered by the average or the maximum distance of the flights departing in the that time frame. Function `fct_reorder(f, x)` orders the levels of `f` based on the median value of `x` (by default) and `fct_reorder2(f, x)` reorders the factor by values associated with the largest `x` values

```
flights_new %>%
  mutate(new_var = fct_reorder(new_var, distance))  %>%
  group_by(new_var) %>%
  summarise(avg_dist = mean(distance, na.rm = TRUE),
            med_dist = median(distance, na.rm = TRUE),
            n = n())
```

```
# A tibble: 6 x 4
  new_var   avg_dist med_dist      n
  <fct>        <dbl>    <dbl>  <int>
1 noon          952.      762  32923
2 night         942.      764   4068
3 evening      1052.      866  73776
4 afternoon    1040.      872 104679
5 morning      1086.      944 113075
6 <NA>          695.      583   8255
```

Note that the `NA` level is always last.

More powerful than changing the orders of the levels is changing their values. The most useful function for this purpose is `fct_recode()`.

```
flights_new %>%
  mutate(new_var = fct_recode(new_var,
    "Noon (starting with 11:01)" = "noon",
    "Afternoon (starting with 13:01)" = "afternoon",
```

```
    "Morning (starting with 05:01)" = "morning",
    "Evening (starting with 18:01)" = "evening",
    "Night (starting with 23:01)" = "night"
  )) %>%
  count(new_var)
```

```
# A tibble: 6 x 2
  new_var                              n
  <fct>                            <int>
1 Afternoon (starting with 13:01) 104679
2 Evening (starting with 18:01)    73776
3 Morning (starting with 05:01)   113075
4 Night (starting with 23:01)       4068
5 Noon (starting with 11:01)       32923
6 <NA>                              8255
```

For recoding one can of course use also base R functions. Here is an example, though it seems more cumbersome.

```
table(flights_new$new_var)
```

```
afternoon   evening   morning     night      noon
   104679     73776    113075      4068     32923
```

```
# a function to capitalize the word ...
# One can use regular expressions, if familiar with them.
capitalize <- function(s) {
  paste0(toupper(substring(s, 1, 1)),
         substring(s, 2))
}
levels(flights_new$new_var) <-
  paste0(capitalize(levels(flights_new$new_var)),
       " (starting with ", c("13", "18", "5", "23", "11"), ":01)")
```

To combine groups, you can assign multiple old levels to the same new level:

```
flights_new %>%
  mutate(new_var = fct_recode(new_var,
    "Noon/Afternoon (starting with 11:01)" = "noon",
     "Noon/Afternoon (starting with 11:01)" = "afternoon",
    "Morning (starting with 05:01)" = "morning",
    "Evening (starting with 18:01)" = "evening",
    "Night (starting with 23:01)" = "night"
  )) %>%
  count(new_var)
```

```
Warning: Unknown levels in `f`: noon, afternoon, morning, evening, night
```

```
# A tibble: 6 x 2
  new_var                              n
  <fct>                            <int>
```

```
1 Afternoon (starting with 13:01) 104679
2 Evening (starting with 18:01)    73776
3 Morning (starting with 5:01)    113075
4 Night (starting with 23:01)       4068
5 Noon (starting with 11:01)       32923
6 <NA>                              8255
```

If you want to collapse a lot of levels, `fct_collapse()` is a useful variant of `fct_recode()`.

```
flights_new %>%
  mutate(new_var = fct_collapse(new_var,
    "Noon/Afternoon (starting with 11:01)" = c("noon", "afternoon"),
    "Night/Morning (starting with 23:01)" = c("morning", "night"),
    "Evening (starting with 18:01)" = "evening"
  )) %>%
  count(new_var)
```

```
Warning: Unknown levels in `f`: noon, afternoon, morning, night, evening
```

```
# A tibble: 6 x 2
  new_var                            n
  <fct>                          <int>
1 Afternoon (starting with 13:01) 104679
2 Evening (starting with 18:01)    73776
3 Morning (starting with 5:01)    113075
4 Night (starting with 23:01)       4068
5 Noon (starting with 11:01)       32923
6 <NA>                              8255
```

Finally, if ou just want to lump together all the small groups to make a plot or table simpler, you can use `fct_lump()`:

```
flights_new %>%
  mutate(new_var = fct_lump(new_var)) %>%
  count(new_var)
```

```
# A tibble: 5 x 2
  new_var                            n
  <fct>                          <int>
1 Afternoon (starting with 13:01) 104679
2 Evening (starting with 18:01)    73776
3 Morning (starting with 5:01)    113075
4 Other                            36991
5 <NA>                              8255
```

The default behavior is to progressively lump together the smallest groups, ensuring that the aggregate is still the smallest group. One can choose the group size for which levels should be lumped together by the setting the `n` argument to an integer:

```
flights_new %>%
  mutate(new_var = fct_lump(new_var, n = 5000)) %>%
  count(new_var)
```

```
# A tibble: 6 x 2
  new_var                             n
  <fct>                           <int>
1 Afternoon (starting with 13:01) 104679
2 Evening (starting with 18:01)    73776
3 Morning (starting with 5:01)    113075
4 Night (starting with 23:01)       4068
5 Noon (starting with 11:01)       32923
6 <NA>                              8255
```

---

**Exercise**

What are the `NA`s in `new_var`? Can you assign them a meaning?

---

### 2.4.3 Dates and times with lubridate

For handlingdates, the **tidyverse** relies on the **lubridate** package (Grolemund and Wickham 2011), which makes it easier to work with dates and times in R. However, **lubridate** is not part of core tidyverse because you only need it if or when you are working with dates/times.

```
library("lubridate")
```

```
Attaching package: 'lubridate'

The following objects are masked from 'package:data.table':

    hour, isoweek, mday, minute, month, quarter, second, wday, week,
    yday, year

The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

```
today()
```

```
[1] "2022-06-01"
```

```
now()
```

```
[1] "2022-06-01 16:55:10 CEST"
```

Often the dates are read into R as strings (one can also specify the column type in package **readr** so the transformation could be tackled at data import). hey automatically work out the format once you specify the order of the component. To use the helper functions in **lubridate**, you must identify the order in which year, month, and day appear in your dates, then arrange "y," "m," and "d" in the same order. That gives you the name of the **lubridate** function that will parse your date. For example:

```
ymd("2021-02-02")
```

```
[1] "2021-02-02"
```

```
mdy("May 10, 2022")
```

```
[1] "2022-05-10"
```

```
dmy("31-Jan-2017")
```

```
[1] "2017-01-31"
```

To create a date-time, add an underscore and one or more of "h," "m," and "s" to the name of the parsing function:

```
dmy_hm("31-Jan-2017 08:01")
```

```
[1] "2017-01-31 08:01:00 UTC"
```

```
dmy_hms("31-Jan-2017 08:01:30")
```

```
[1] "2017-01-31 08:01:30 UTC"
```

Going back the the `flights` data, there are also helper functions which make a date or date-time from multiple columns. For example, to create a date of scheduled departure we have

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
# A tibble: 336,776 x 6
    year month   day  hour minute departure
   <int> <int> <int> <dbl>  <dbl> <dttm>
 1  2013     1     1     5     15 2013-01-01 05:15:00
 2  2013     1     1     5     29 2013-01-01 05:29:00
 3  2013     1     1     5     40 2013-01-01 05:40:00
 4  2013     1     1     5     45 2013-01-01 05:45:00
 5  2013     1     1     6      0 2013-01-01 06:00:00
 6  2013     1     1     5     58 2013-01-01 05:58:00
 7  2013     1     1     6      0 2013-01-01 06:00:00
 8  2013     1     1     6      0 2013-01-01 06:00:00
 9  2013     1     1     6      0 2013-01-01 06:00:00
10  2013     1     1     6      0 2013-01-01 06:00:00
# ... with 336,766 more rows
```

The departure and arrival times are in a rather weird format. To create proper data-times we would need something like:

```
flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime(year, month, day, dep_time %/% 100, dep_time %% 100),
    arr_time =  make_datetime(year, month, day, arr_time %/% 100, arr_time %% 100)
  ) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"))
flights_dt
```

```
# A tibble: 328,063 x 9
   origin dest  dep_delay arr_delay dep_time            sched_dep_time
   <chr>  <chr>     <dbl>     <dbl> <dttm>                       <int>
 1 EWR    IAH           2        11 2013-01-01 05:17:00            515
 2 LGA    IAH           4        20 2013-01-01 05:33:00            529
 3 JFK    MIA           2        33 2013-01-01 05:42:00            540
 4 JFK    BQN          -1       -18 2013-01-01 05:44:00            545
 5 LGA    ATL          -6       -25 2013-01-01 05:54:00            600
 6 EWR    ORD          -4        12 2013-01-01 05:54:00            558
 7 EWR    FLL          -5        19 2013-01-01 05:55:00            600
 8 LGA    IAD          -3       -14 2013-01-01 05:57:00            600
 9 JFK    MCO          -3        -8 2013-01-01 05:57:00            600
10 LGA    ORD          -2         8 2013-01-01 05:58:00            600
# ... with 328,053 more rows, and 3 more variables: arr_time <dttm>,
#   sched_arr_time <int>, air_time <dbl>
```

You may want to switch between a date-time and a date. That is the job of `as_datetime()` and `as_date()`:

```
as_datetime(today())
```

```
[1] "2022-06-01 UTC"
```

```
as_date(now())
```

```
[1] "2022-06-01"
```

Sometimes you will get date/times as numeric offsets from the "Unix Epoch," 1970-01-01. If the offset is in seconds, use `as_datetime()`; if it is in days, use `as_date()`.

```
as_datetime(60 * 60 * 10)
```

```
[1] "1970-01-01 10:00:00 UTC"
```

```
as_date(365 * 10 + 2)
```

```
[1] "1980-01-01"
```

One can also use the accessor functions that let us get individual components.

```r
datetime <- ymd_hms("2016-07-08 12:34:56")
year(datetime)
```

```
[1] 2016
```

```r
month(datetime)
```

```
[1] 7
```

```r
month(datetime, label = TRUE)
```

```
[1] Jul
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```r
month(datetime, label = TRUE, abbr = FALSE)
```

```
[1] July
12 Levels: January < February < March < April < May < June < ... < December
```

```r
mday(datetime)
```

```
[1] 8
```

```r
yday(datetime)
```

```
[1] 190
```

```r
wday(datetime)
```

```
[1] 6
```

```r
wday(datetime, label = TRUE)
```

```
[1] Fri
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```r
wday(datetime, label = TRUE, abbr = FALSE)
```

```
[1] Friday
7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

```r
hour(datetime)
```

```
[1] 12
```

```
minute(datetime)
```

```
[1] 34
```

```
second(datetime)
```

```
[1] 56
```

One can also perform operations on dates. For example, if we want to add one year to the scheduled departure day in the `flights` data we can simply do:

```
flights_dt %>%
  select(dep_time) %>%
  mutate(dep_time_1_year = dep_time + years(1))
```

```
# A tibble: 328,063 x 2
   dep_time            dep_time_1_year
   <dttm>              <dttm>
 1 2013-01-01 05:17:00 2014-01-01 05:17:00
 2 2013-01-01 05:33:00 2014-01-01 05:33:00
 3 2013-01-01 05:42:00 2014-01-01 05:42:00
 4 2013-01-01 05:44:00 2014-01-01 05:44:00
 5 2013-01-01 05:54:00 2014-01-01 05:54:00
 6 2013-01-01 05:54:00 2014-01-01 05:54:00
 7 2013-01-01 05:55:00 2014-01-01 05:55:00
 8 2013-01-01 05:57:00 2014-01-01 05:57:00
 9 2013-01-01 05:57:00 2014-01-01 05:57:00
10 2013-01-01 05:58:00 2014-01-01 05:58:00
# ... with 328,053 more rows
```

Functions `seconds()`, `minutes()`, `hours`, `days()`, `weeks()` `months()` can be used similarly.

For more information consult Chapter 16, R for Data Science.

## 2.5 Data export

The functions presented in Section 2.1 all have a counterpart for saving data from R into different formats.

If data is in a tabular format we have several options:

- If a data frame should be saved as an ASCII file the function `write.table` can be used. It works basically as the function `read.table`. Additional arguments allow you to exclude row and column names, specify what to use for missing values, add or remove quotations around character strings, etc.

```
write.table(iris, file = "data/iris_copy.csv")
write.table(iris, file = "data/iris_copy.csv", sep = "\t",
          row.names = FALSE, na = "MISSING!")
```

- If you have a very large data frame you want to export, then the function `write.matrix` of the package `MASS` might be more suitable since it requires much less memory.
- **readr** write functions are about twice as fast and they do not write row names

```
# as csv
write_csv(iris, path = "data/iris_copy.csv")
# as excel
write_excel_csv(df, path = "df", col_names = FALSE)
```

- **haven** can be used to save other types of files such as `.dta`, `.sav` etc.

However, it is often of interest to export R object files (data or other types of objects such as lists, model outputs etc.). There are three primary ways that people tend to save R data/objects: as .RData, .rda, or as .rds files. .rda is just short for .RData, therefore, these file extensions represent the same underlying object type. You use the .rda or .RData file types when you want to save several, or all, objects and functions that exist in your global environment. On the other hand, if you only want to save a single R object such as a data frame, function, or statistical model results its best to use .rds file type.

The following illustrates how you save R objects with each type.

```
# save() can be used to save multiple objects in you global environment,
# in this case I save two objects to a .RData file
x <- stats::runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.RData")

# save a single object to a .RData file
save(x, file = "x.RData")

# save a single object to file
saveRDS(x, "x.rds")

# write rds file readr
readr::write_rds(x, "x.rds")
```

As you can see, you can also use .rda or .RData to save a single object but the benefit of .rds is that it only saves a representation of the object and not the name whereas .rda and .RData save the both the object and its name. As a result, with .rds the saved object can be loaded into a named object within R that is different from the name it had when originally saved.

```
# x <- load("xy.RData") this will not work
load("x.RData") # this will load x.
str(x) # I need to know the name of the saved object
```

```
 num [1:20] 0.59 0.299 0.239 0.929 0.638 ...
```

```
x <- readRDS("x.rds")
str(x) # I need to know the name of the saved object
```

```
 num [1:20] 0.59 0.299 0.239 0.929 0.638 ...
```

# 3 Part 3: Visualization

Data visualization is an integral part of data analysis. It is used both for exploratory purposes as well as an explanatory purposes. The first gives you insight about the data, lets you generate hypothesis about

the data, and allows human intuition to participate in the selection of a suitable model for your data. The second purpose for graphical tools is to communicate the results of your analysis to others.

Chosing the right graph for your data and illustrating it in the most informative way is a form of art and requires a lot of experience. For inspiration one can look at the following resources

- the R graph gallery https://www.r-graph-gallery.com/ (for all kinds of graphs)
- the `vcd` package for visualizing categorical data.

R has several graphic systems to produce graphs which follow different philosophies.

1. Base R graphics
2. Trellis plots (package **lattice**)
3. Grammar of graphics plots (package **ggplot2**)
4. Grid graphics

Here we will cover 1. and 3.

## 3.1 Base R Graphics

This is the standard graphics system of R. A small overview of the different functions can be accessed by the demo

```
demo("graphics")
```

Base R graphics functions can be divided into three groups:

- **High-level** plotting functions (to create a new plot in the graphics window.)
- **Low-level** plotting functions (to add information to an existing plot.)
- **Interactive graphics** functions (to add or extract information to / from an existing plot using devices such as a mouse.)

### 3.1.1 High-level plotting commands

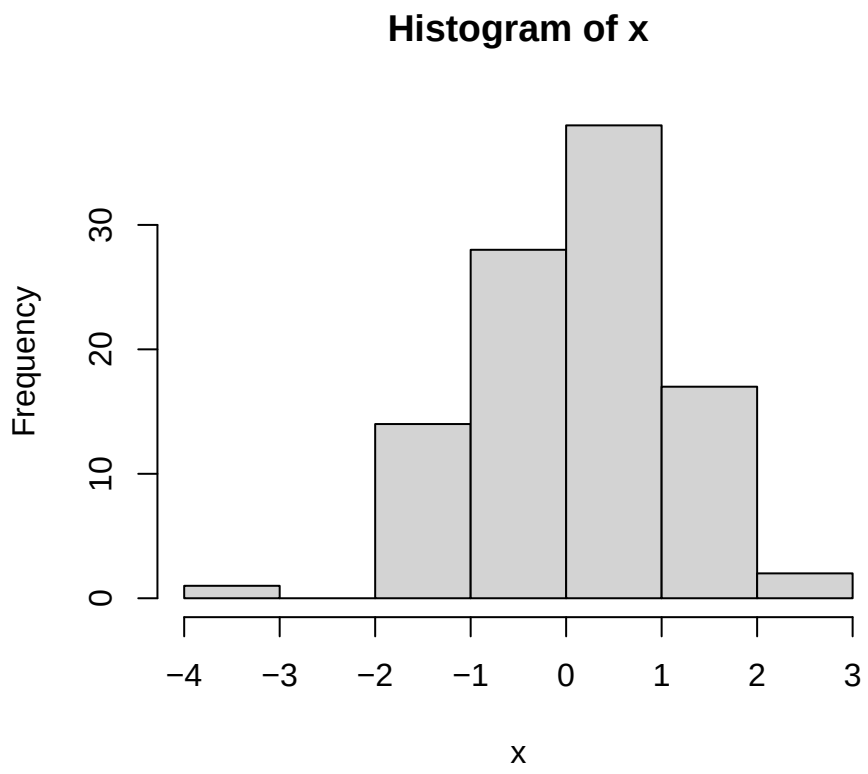
High-level plotting commands create a complete plot for the data passed on. They create always a new plot and overwrite if necessary the last one. High plotting commands are generally generic functions which produce different types of plots depending on the class of the data to plot. High-level plotting commands have default values for axes, labels, etc., depending on the type of plot.

The main plotting function is `plot`. It has methods for most classes. To get an overview see `methods(plot)`. However, there are many other high-level plotting commands. Let us have a look at these ordered by the characteristics of the data.

#### 3.1.1.1 Univariate continuous variables

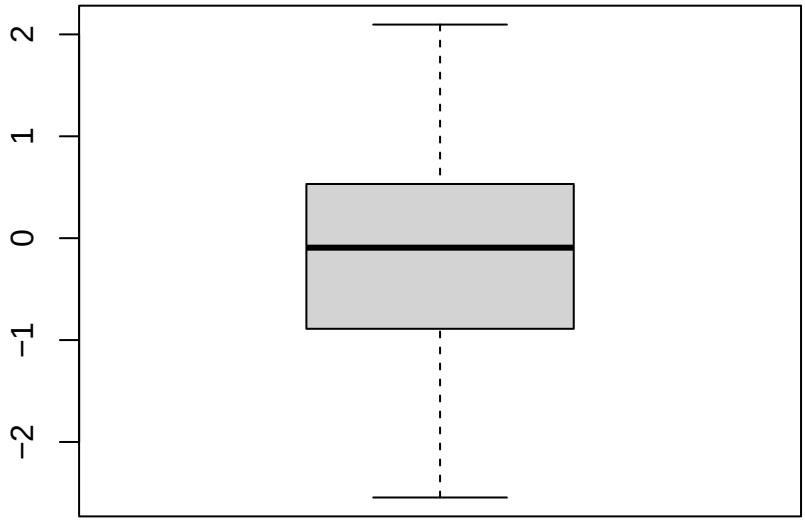
`hist(x)` plots a histogram.

```
x <- rnorm(100)
hist(x)
```




`boxplot(x)` produces a boxplot.

```
x <- rnorm(100)
boxplot(x)
```

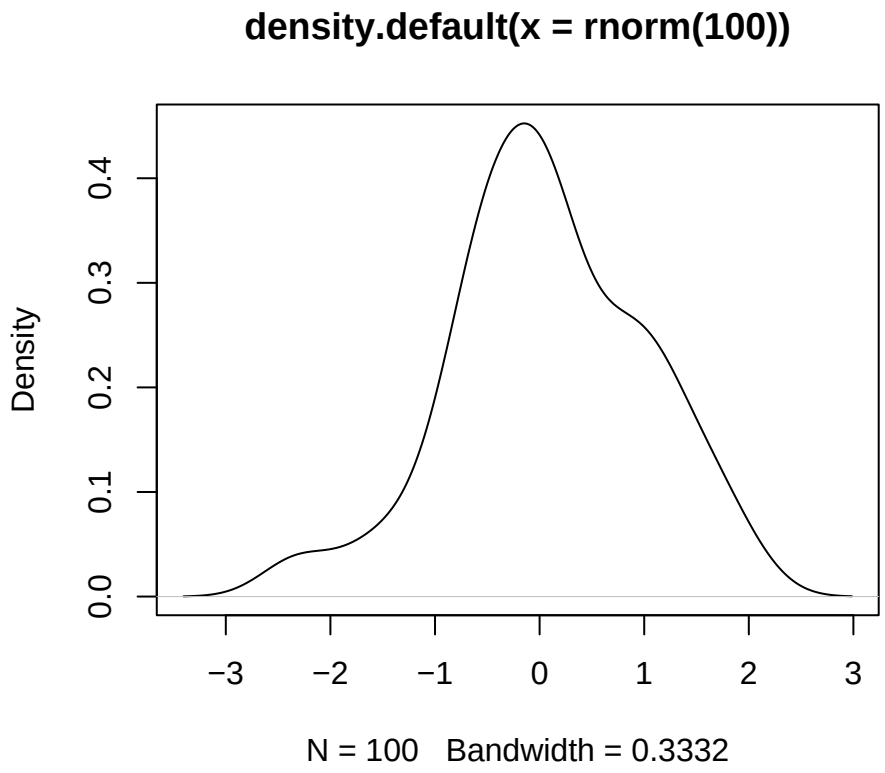



`plot(x)` produces a plot of the density estimate if `x` is of class `density`.

```
x <- density(rnorm(100))
class(x)
```

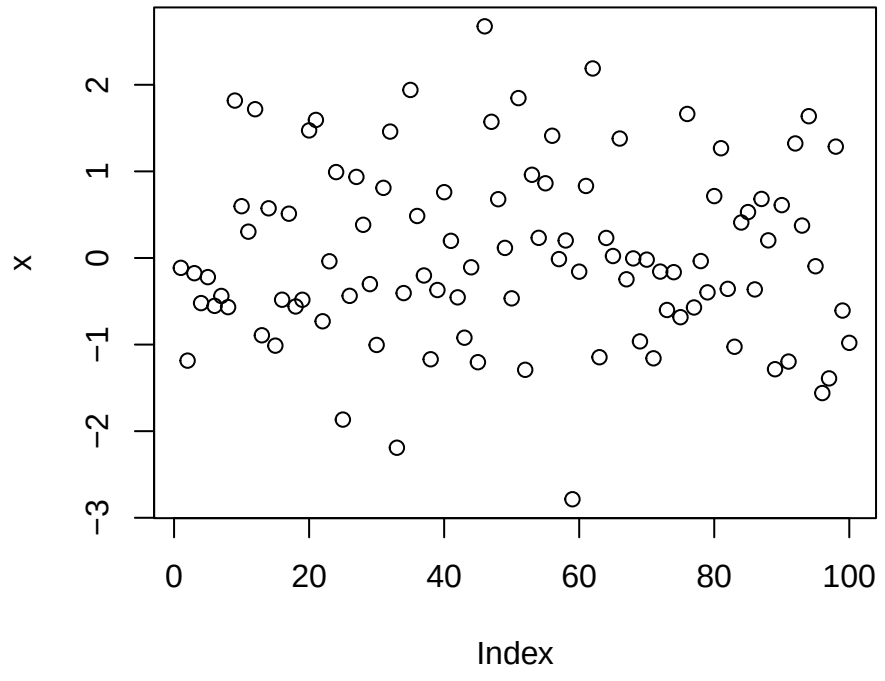
```
[1] "density"
```

```
plot(x)
```



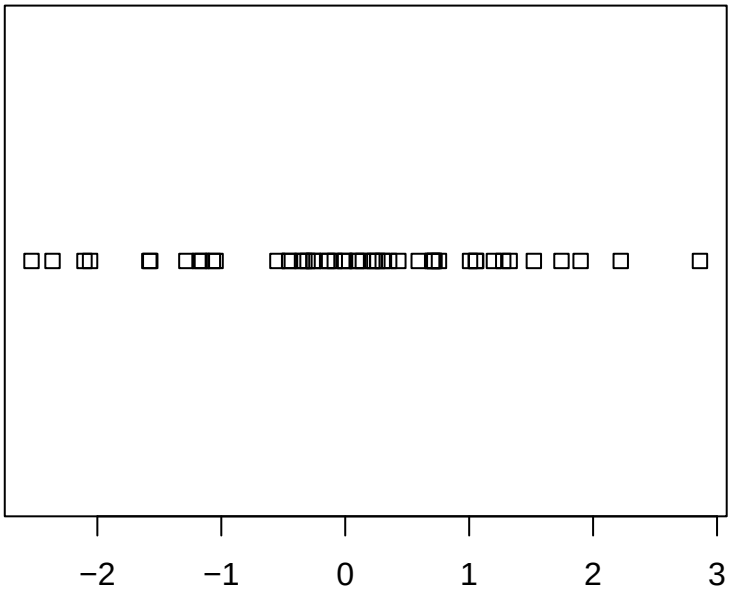

For a univariate continuous variable `x`, `plot(x)` produces a scatter plot of `x` against its index vector.

```
x <- rnorm(100)
plot(x)
```



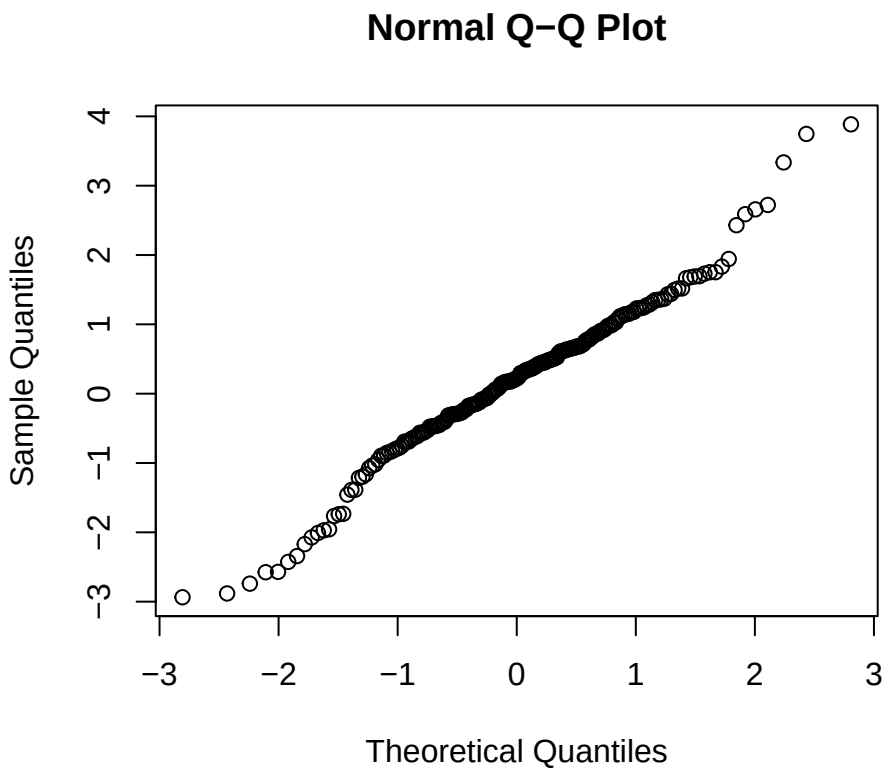
`stripchart(x)` produces a 1D scatterplot.

```
x <- rnorm(50)
stripchart(x)
```



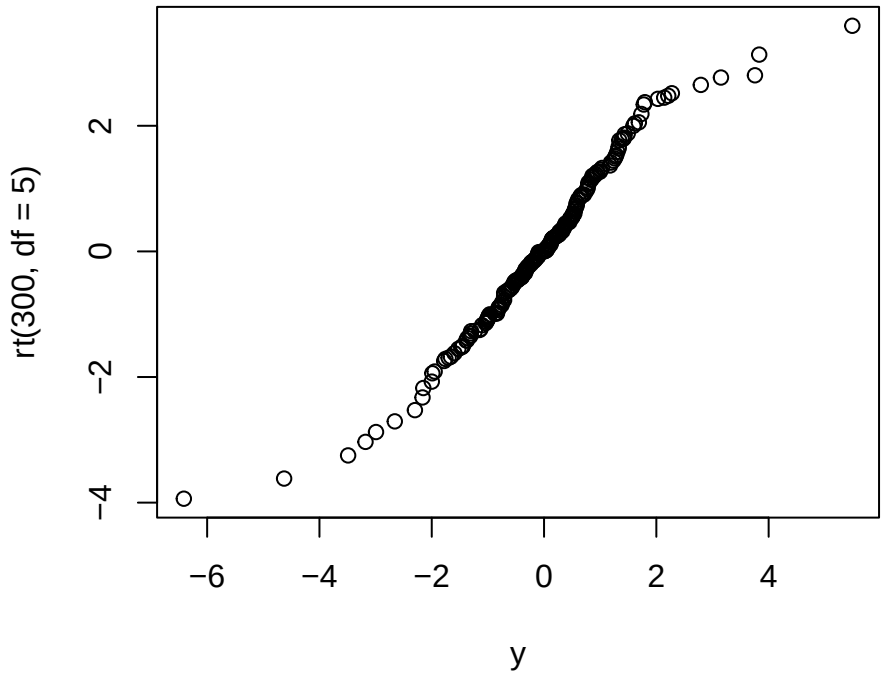

In order to compare the distribution with a normal or any other given distribution one can use `qqnorm(x)` which plots the quantiles of `x` against the ones of the normal distribution.

```
y <- rt(200, df = 5)
qqnorm(y)
```



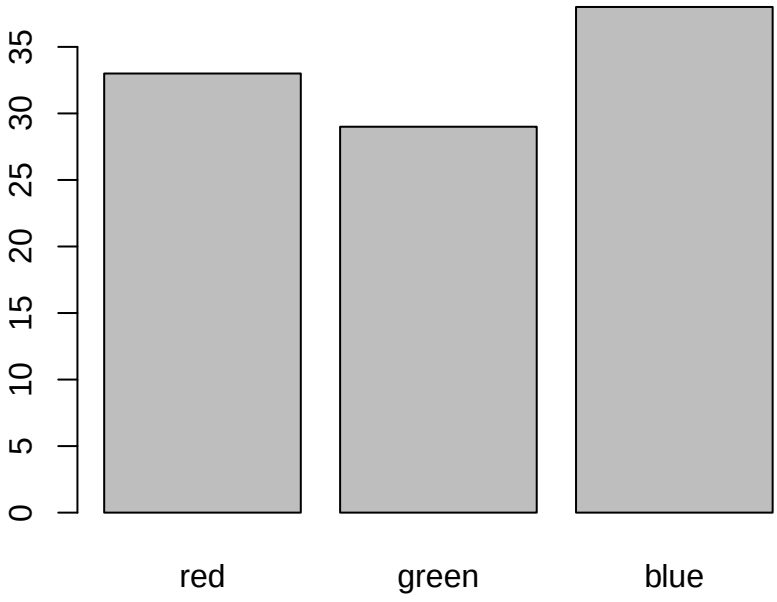
`qqplot(x,y)` plots the quantiles of `x` against the quantiles of `y`.

```
y <- rt(200, df = 5)
qqplot(y, rt(300, df = 5))
```



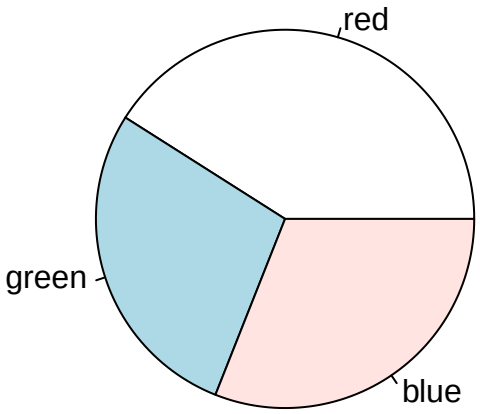

**3.1.1.2 Univariate categorical variables** `barplot(x)` produces a bar plot if `x` is a factor. This yields the same plot as `plot(x)`.

```
fac <- factor(sample(1:3, 100, replace = TRUE), labels=c("red", "green", "blue"))
plot(fac)
```



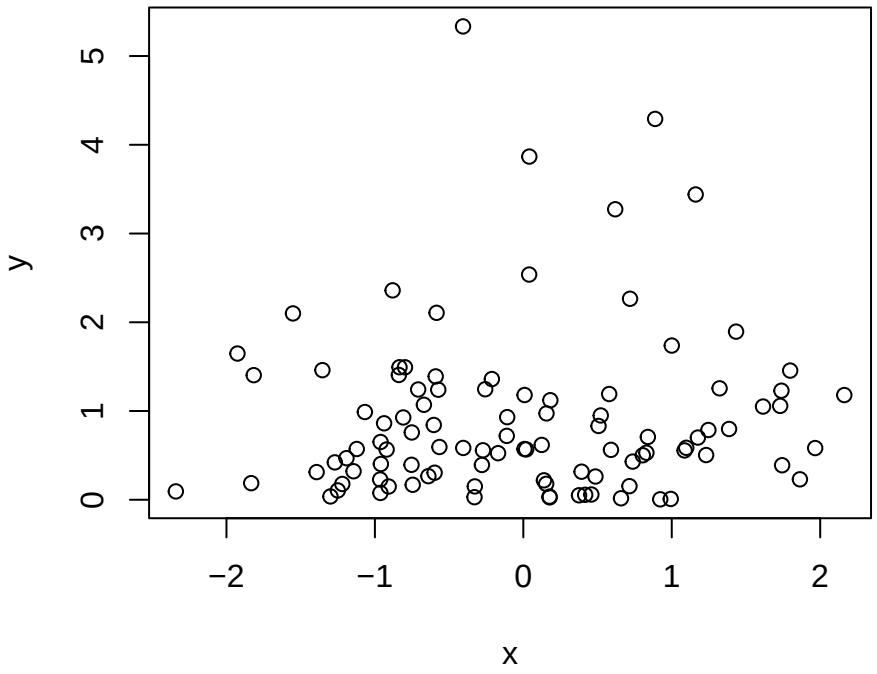
`pie(x)` produces a pie chart.

```
fac <- factor(sample(1:3, 100, replace = TRUE), labels=c("red", "green", "blue"))
pie(table(fac))
```




**3.1.1.3 Covariation of two continuous variables** `plot(x,y)` produces a scatter plot if `x` and `y` are numeric.

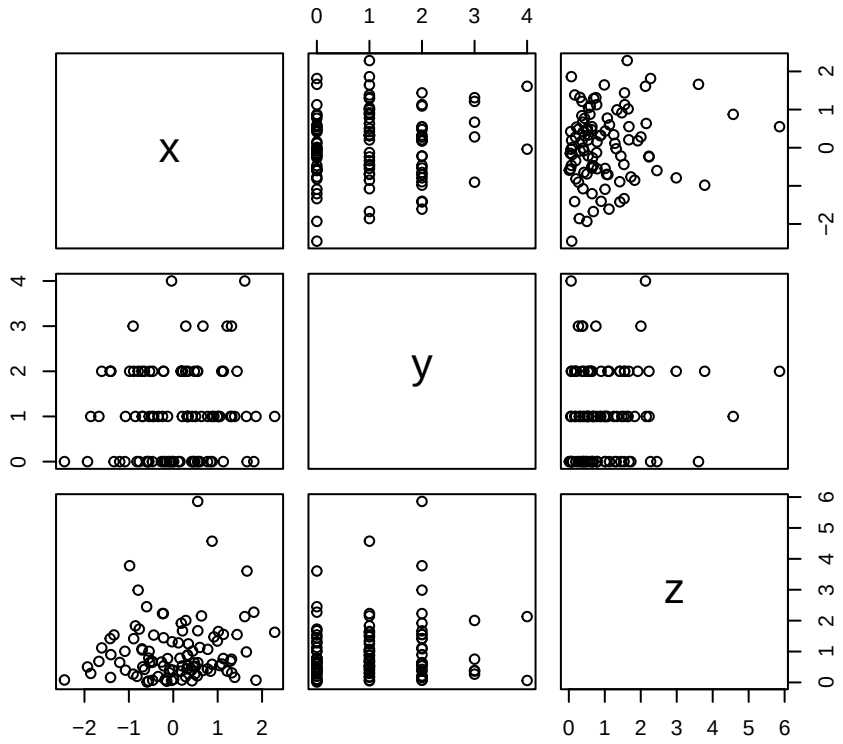
```
x <- rnorm(100)
y <- rexp(100)
plot(x,y)
```




For several continuous variables we can plot pairwise scatterplots of all pairs of variables.

`pairs(X)` produces a scatter plot matrix if `X` is a matrix or data frame. This is equivalent to `plot(X)`.

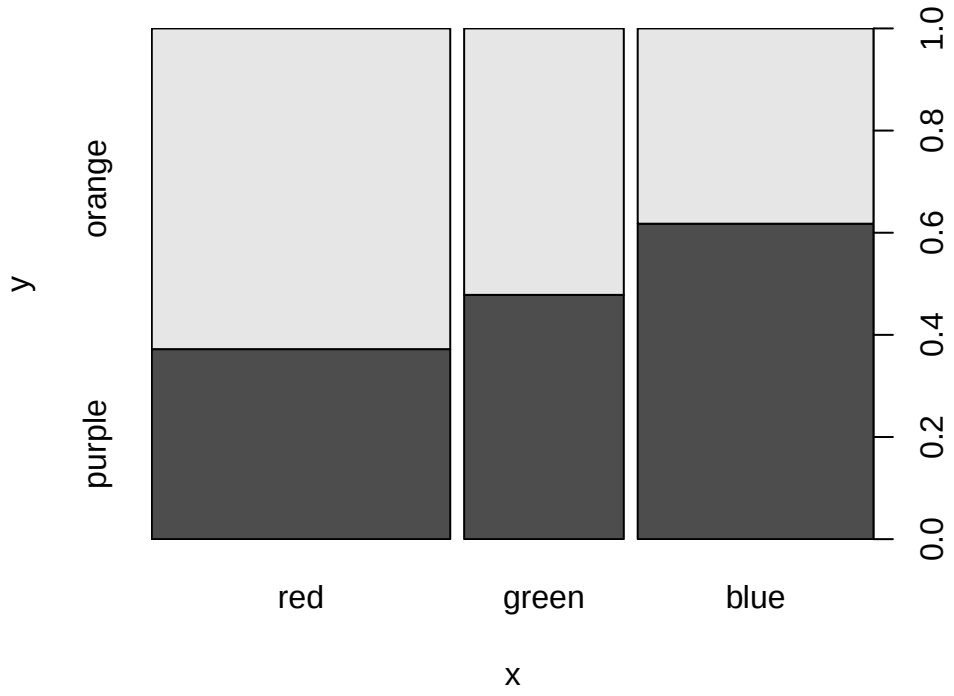
```
df <- data.frame("x" = rnorm(100), y = rpois(100, lambda = 1), z = rexp(100))
pairs(df)
```




#### 3.1.1.4 Covariation of two discrete variables

`plot(x,y)` produces a spine plot if `x` and `y` are factors.

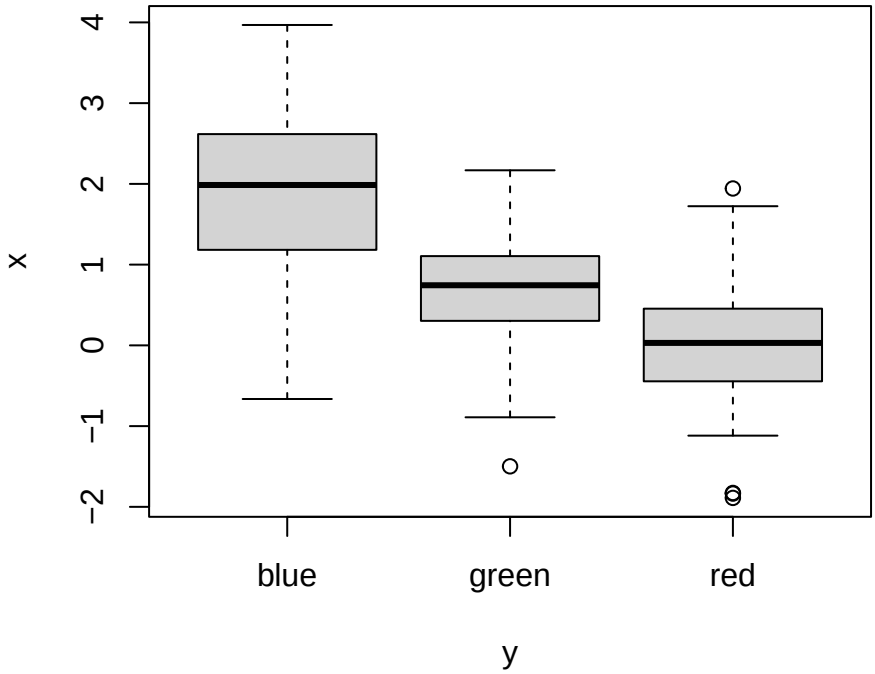
```
fac1 <- factor(sample(1:3, 100, replace = TRUE), labels=c("red", "green", "blue"))
fac2 <- factor(sample(1:2, 100, replace = TRUE), labels=c("orange", "purple"))
plot(fac1, fac2)
```



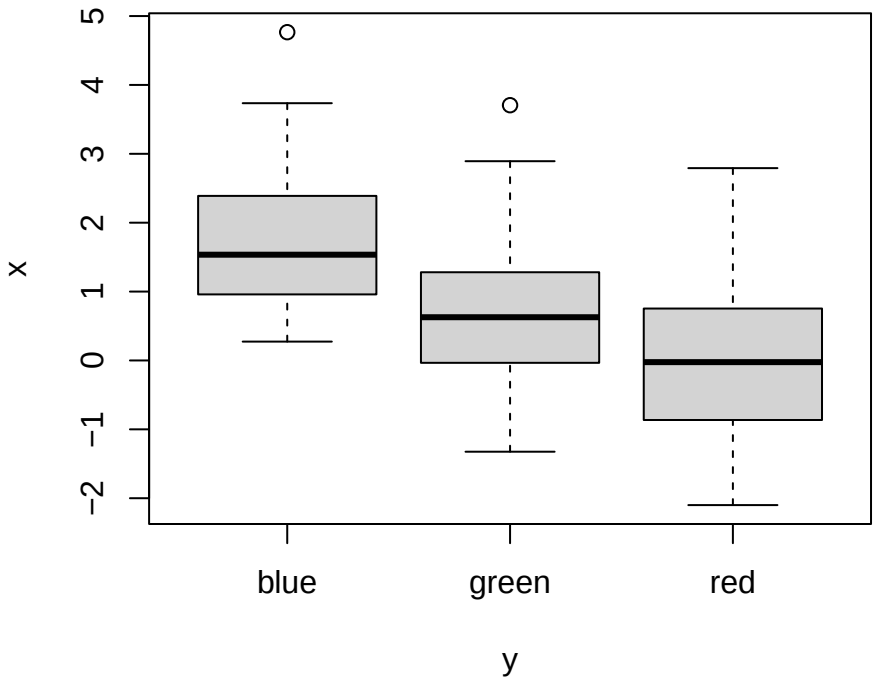

**3.1.1.5 Covariation of one continuous and one discrete or categorical variable** In order to compare several distributions one can do the following

`boxplot(x ~ y)` produces a boxplot of x for each level of y if `x` is numeric and `y` is a factor. This yields the same plot as `plot(x~y)`.

```
y <- factor(c(rep("red", 30), rep("blue", 30), rep("green", 30)))
x <- c(rnorm(30, mean = 0), rnorm(30, mean = 2), rnorm(30, mean=1))
boxplot(x ~ y)
```

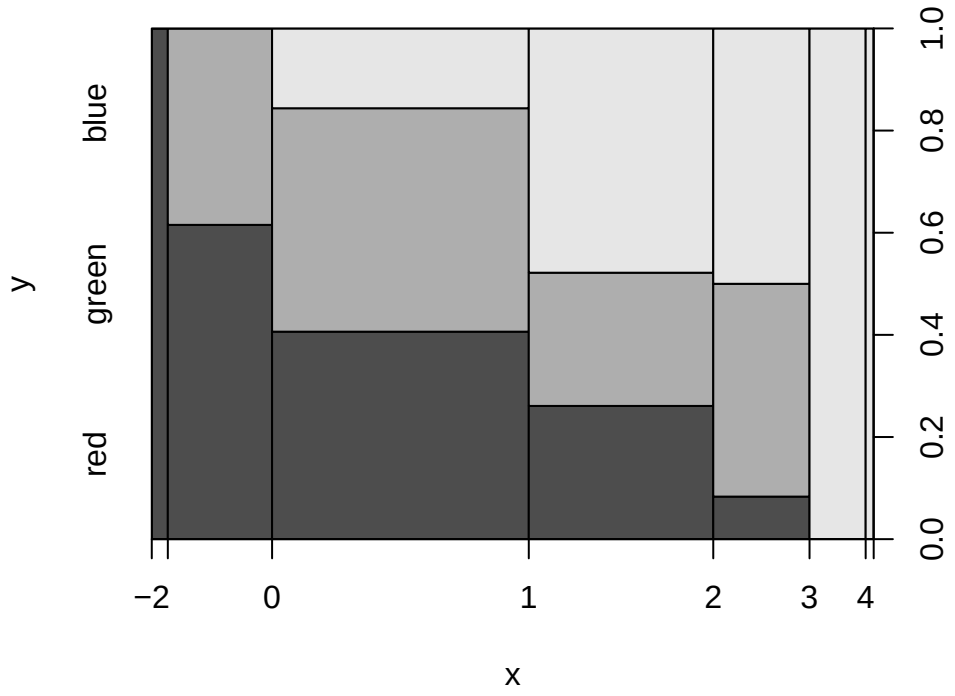


```
y <- factor(c(rep("red", 30), rep("blue", 30), rep("green", 30)))
x <- c(rnorm(30, mean = 0), rnorm(30, mean = 2), rnorm(30, mean=1))
plot(x ~ y)
```




If the categorical variable is to be plotted on the y axis (and should be seen as a dependent variable in the further analysis), R plots by default a spineplot:

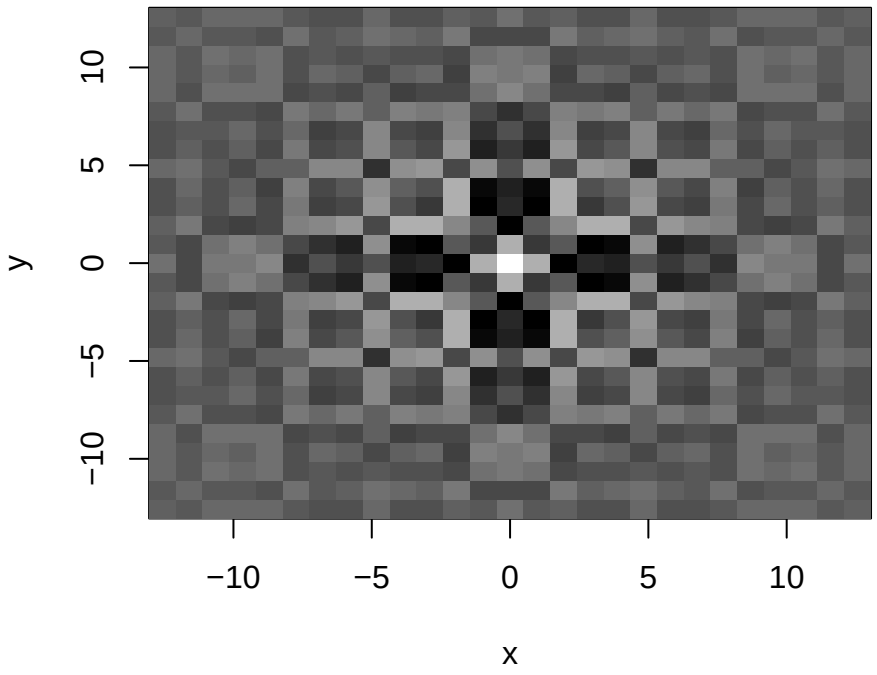
```
y <- factor(c(rep("red", 30), rep("blue", 30), rep("green", 30)))
x <- c(rnorm(30, mean = 0), rnorm(30, mean = 2), rnorm(30, mean=1))
plot(y ~ x)
```




**3.1.1.6 Three dimensional data** For three dimensional data, the base package provides `image()`, `contour()`, and `persp()`.

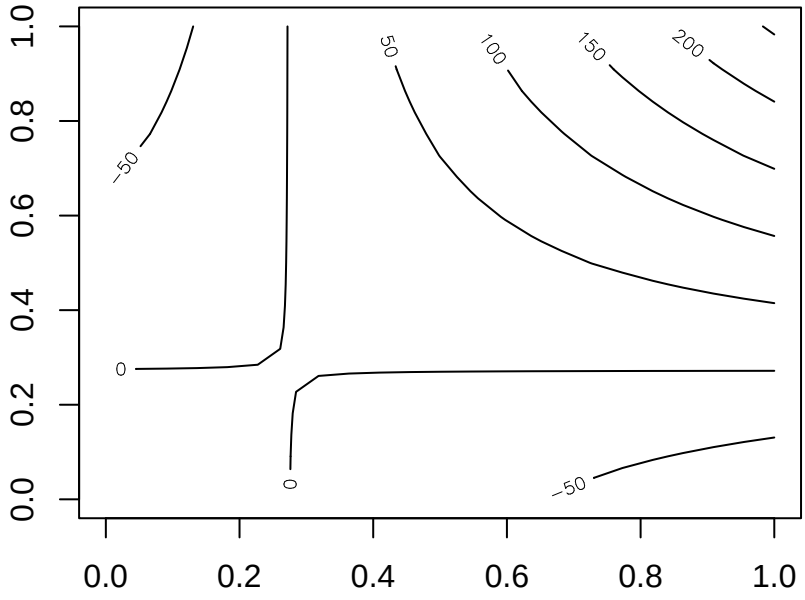
`image(x,y,z)` plots a grid of rectangles along the ascending x, y values and fills them with different colours to represent the values of z.

```
require(grDevices)
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
z <- cos(r^2)*exp(-r/6)
image(x, y, z, col  = gray((0:32)/32))
```




`contour(x,y,z)` draws a contour plot for z.

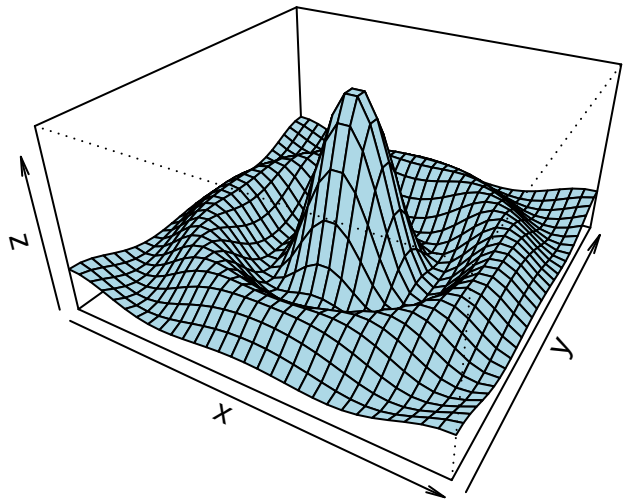
```
x <- -6:16
contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
```




`persp(x,y,z)` draws a 3D surface for z.

```
x <- seq(-10, 10, length= 30)
y <- x
f <- function(x, y) {
  r <- sqrt(x^2+y^2)
```

```
  10 * sin(r)/r
}
z <- outer(x, y, f)
z[is.na(z)] <- 1
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
```




### 3.1.2 Arguments for high-level plotting commands

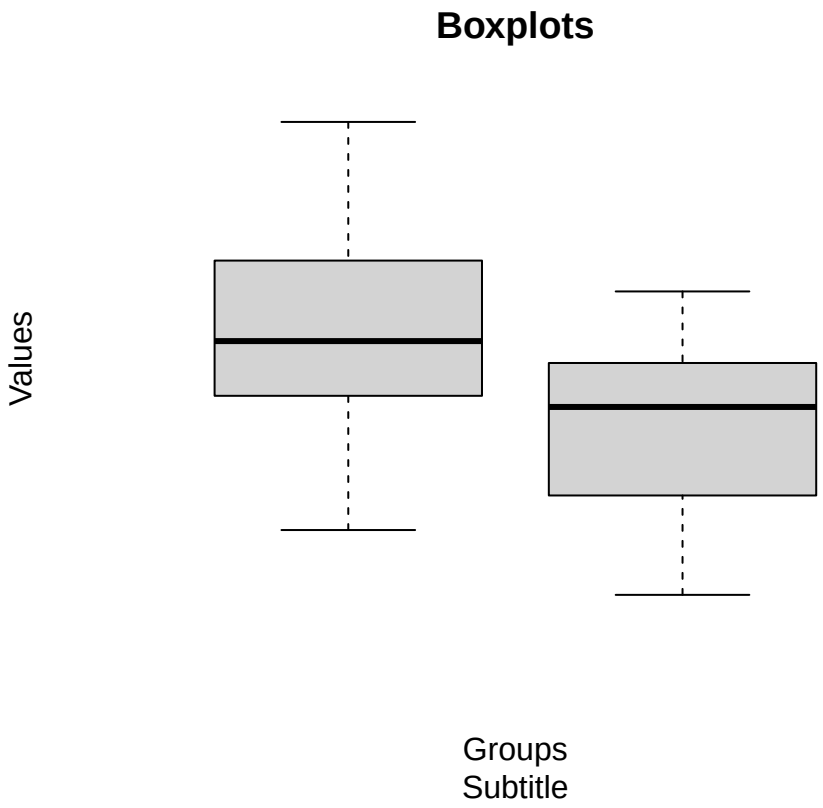
R plots look most times already pretty good by default, though one normally needs to customize some settings. There are a lot of arguments for high-level plotting commands to do so. Here is a selection of some of them. However, they don't work for all the functions.

- `add = TRUE`
  forces the function to act like a low-level plotting command, "adds" the plot to an already existing one.
- `axes = FALSE`
  suppresses axis, useful when custom axes are added.
- `log = "x", "y" or "xy"`
  Logarithmic transformation of x, y or both axes.
- `xlab = "text"`
  changes x-axis label (default usually object name).
- `ylab = "text"`
  changes y-axis label (default usually object name).
- `main = "text"`
  adds a title at the top of the plot.
- `sub = "text"`
  adds a subtitle just below the x-axis.
- `type =`
  controls the type of the plot, the default is points `p`. Other options for this argument are:
- `"l"`: lines between the points.
- `"b"`: plots points and connects them with lines.

- `"o"`: points overlaid by lines.
- `"h"`: draws vertical lines to the zero axis.
- `"s"`: step function, top of the vertical defines the point.
- `"S"`: step function, bottom of the vertical defines the point.
- `"n"`: no plotting, plots however the default axes and coordinates according to the data (might be needed in order to continue with low-level plotting commands).

```
df <- data.frame("x"=c(rnorm(50, mean=1), rnorm(50)), "y"=c(rep("A", 50), rep("B", 50)))

boxplot(x~y, data = df, axes = FALSE,
        xlab = "Groups", ylab = "Values", main = "Boxplots", sub = "Subtitle")
```




### 3.1.3 Low-level plotting functions

Low-level plotting commands add additional information (like extra points, lines, legend, ...) to an existing plot. A non-exhaustive list includes:

**3.1.3.1 Points and lines** The functions `points` and `lines` add points or lines to the current plot. The different types can also be specified using the `type=` argument.
The function `abline` is however often more convenient to add straight mathematical lines. It can be used in the following ways:

- `abline(a,b)` adds a line with intercept a and slope b.

- `abline(h=y)` y defines the y coordinate for a horizontal line.
- `abline(v=x)` x defines the x coordinate for a vertical line.
- `abline(lm.object)` if lm.object is a list of length 2 it adds a line using the first value as intercept and the second as slope.
- `polygon` adds a polygon to the existing plot.

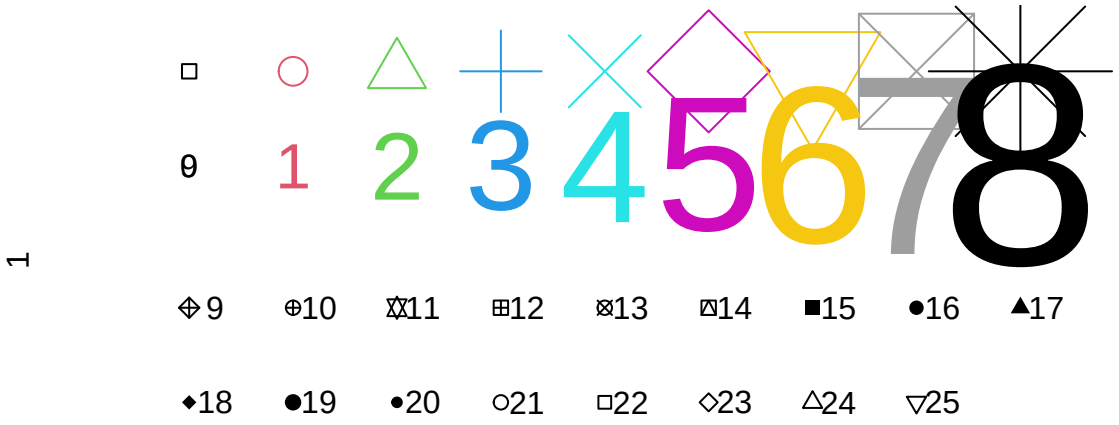
#### 3.1.3.2 Text and legend

- `title` can be used to add titles and subtitles to an existing plot. The positions will be the same as when using the corresponding arguments of the high-level plotting commands.

- `axis(side,...)` can be used to modify all aspects (position, label, tickmarks, ...) of the axis. This function is mainly used when in the high-level plotting function the argument `axes` was set to `FALSE`.
- `text(x,y,labels,...)` adds text to a plot at specified coordinates, which means that $label_i$ is put to the position $(x_i, y_i)$.
- `legend(x,y,legend,...)` adds a legend to a specified position in the plot.
- `fill = v` colours of filled boxes.
- `lty = v` line styles.
- `lwd = v` line widths.
- `col = v` colours of points or lines.
- `pch = v` plotting characters.

The standard colors and plotting characters can be seen in the following figure:

```
plot(1,1,xlim=c(1,10),ylim=c(0,5),type="n", axes=FALSE)
points(1:9,rep(4.5,9),cex=1:9,col=1:9,pch=0:8)
text(1:9,rep(3.5,9),labels=paste(0:9),cex=1:9,col=1:9)
points(1:9,rep(2,9),pch=9:17)
text((1:9)+0.25,rep(2,9),paste(9:17))
points(1:8,rep(1,8),pch=18:25)
text((1:8)+0.25,rep(1,8),paste(18:25))
```



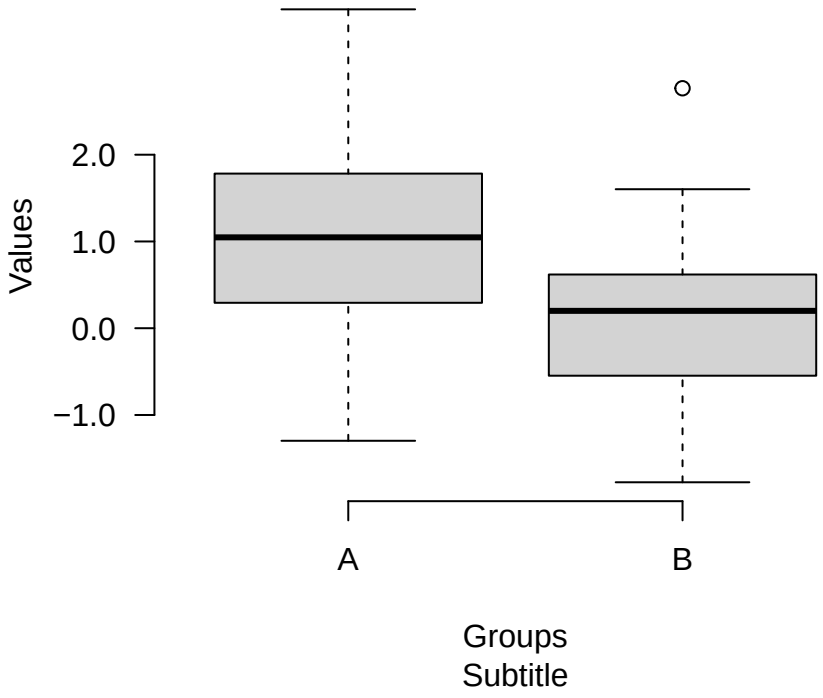

1

Note:
Also mathematical symbols and formulae can be added as text, then the labels are rather expressions. For details see help for `plotmath`.

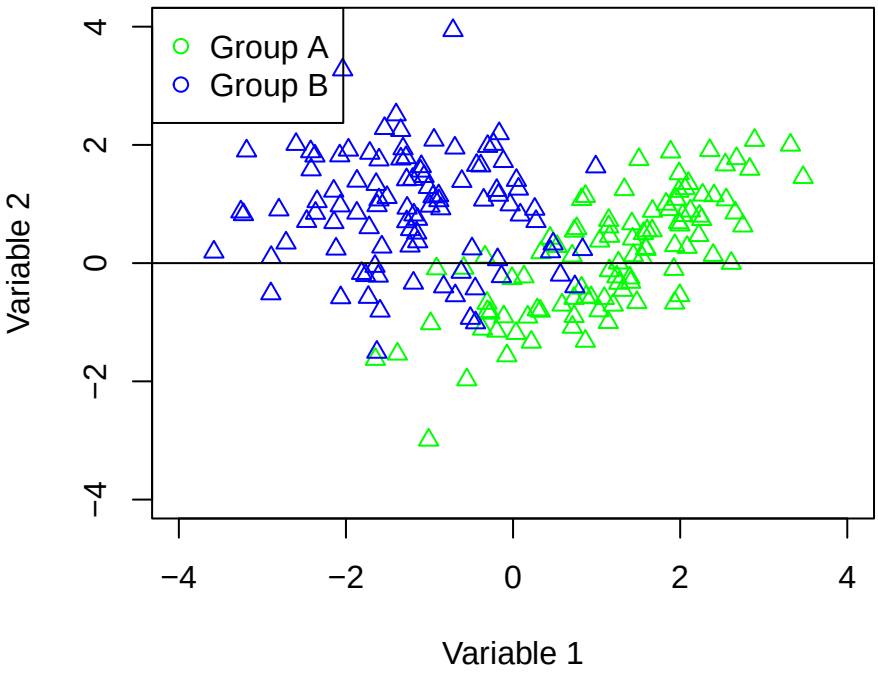
```
df <- data.frame("x"=c(rnorm(50, mean=1), rnorm(50)), "y"=c(rep("A", 50), rep("B", 50)))

boxplot(x~y, data = df, axes = FALSE,
        xlab = "Groups", ylab = "Values", main = "Boxplots", sub = "Subtitle")

axis(side=1, at = c(1,2), labels = c("A", "B"))
axis(side=2, at = c(-1,0,1,2), labels = c("-1.0", "0.0", "1.0", "2.0"), las = 2)
```




```
df <- data.frame(rbind(MASS::mvrnorm(100, mu = c(1,0), Sigma = matrix(c(1,0.7,0.7,1), ncol=2)),
                       MASS::mvrnorm(100, mu = c(-1,1), Sigma = matrix(c(1,0.0,0.0,1), ncol=2))),
                 "group" = c(rep("A", 100), rep("B", 100)))
plot(df$X1, df$X2, xlim = c(-4,4), ylim = c(-4,4),
     col=c(rep("green", 100), rep("blue", 100)),
     pch = 2, xlab = "Variable 1", ylab="Variable 2")
abline(h=0, col="black")
legend("topleft", legend=c("Group A", "Group B"), col = c("green", "blue"), pch = c(1,1))
```




### 3.1.4 Interactive plotting functions

To position text, labels, legends and so on is often a laborious task since it is difficult to choose the right values for the coordinates x and y. One problem here is also, that those values change with the data used - those coordinates are related to the scale of the observations. With the help of interactive functions the user has the possibility to place those items using the mouse.

But one can also use interactive functions not only to add something to the plot but one can also extract information from the plot.

- `locator` can be used to add "points" to an existing plot. This function has two arguments, `n`, the number of points to add, and `type`, the type of the points. The points or positions can be chosen by pressing the left mouse button. More often the `locator` function is used to position text on a plot. Then the usage is for example within the `text` function:
  `text(locator(1),"text")`
- `identify()` can be used to highlight a certain point in a plot

```
plot(x,y)
identify(x,y)
```

one could click the left mouse button and the index of the x value of the point closest to the cursor would appear on the plot. Given a vector of the same length of x and y containing labels for the observation one could highlight the labels with

```
identify(x,y, labels)
```

This would end as soon as all points are identified or pressing the right mouse button and choosing then stop.

### 3.1.5 Graphic parameters

Always when a graphic device gets activated, a list of graphical parameters is activated. This list has certain default settings. Those default settings are often however not satisfying and should be changed. Changes can be done permanently and for all plots using the `par` function or within the call of a high-level plotting function using the same arguments, e.g.

```
plot(x,y, pch="*")
```

produces a scatter plot with * as plotting character instead of using a point.

With graphical parameters one can change almost every aspect of a graphic. Submitting only `par()` gives a list with all graphical parameters and their current settings. If one want to change the parameters for a number of plots and wants to return to the default setting afterwards, one can do the following:

```
def.par <- par(no.readonly = TRUE) # save default, for resetting...
# block of commands
par(def.par)  #- reset to default
```

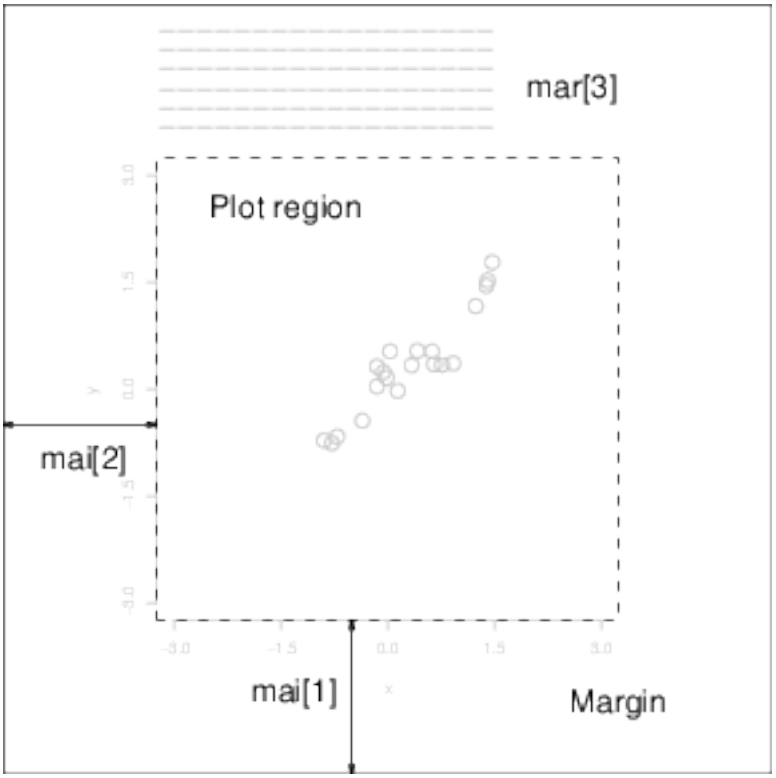
The most important graphic parameters are:

- `pch` specifies the character used for the plotting. The character can be directly specified submitting it in quotes or indirectly by providing an integer between 0 and 18.
- `lty` specifies the line type with an integer from 1 onwards.
- `lwd` specifies the line width in multiples of the default width.
- `col` specifies the colour of the symbols, text and so on. For every graphic element exits a list with the possible colours. The value needed here is the index of the colour in the list.
- `cex` specifies the character expansions in multiples of the default.

#### 3.1.5.1 Parameters concerning the axis

- `lab = c(x,y,n)` `x` specifies the number of ticks at the x-axis, `y` at the y-axis, `n` the length of the tick labels in characters (including decimal point).
- `las` determins the orientation of axis labels (0=parallel, 1=horizontal, 2=perpendicular, 3=vertical).
- `mgp = c(d1,d2,d3)` positions of axis components (details see manual).
- `tck` length of the tick marks.
- `xaxs` style of the x-axis (possible settings, “s,” “e,” “i,” “r,” “d”) y-axis analogous.

**3.1.5.2 Parameters concerning the figure margins** There are two arguments to control the margins. The argument `mai` sets the margins measured in inches, whereas the argument `mar` measures them in number of text lines. The margins itself are divided into four parts: the bottom is part 1, left part 2, top part 3 and right part 4. The different parts are addressed with the corresponding index of the margin vector.
For instance:
`mai=c(1,2,3,4)` (1 inch bottom, 2 inches left, 3 inches top, 4 inches right)
`mar=c(1,2,3,4)` (1 line bottom, 2 lines left, 3 lines top, 4 lines right)

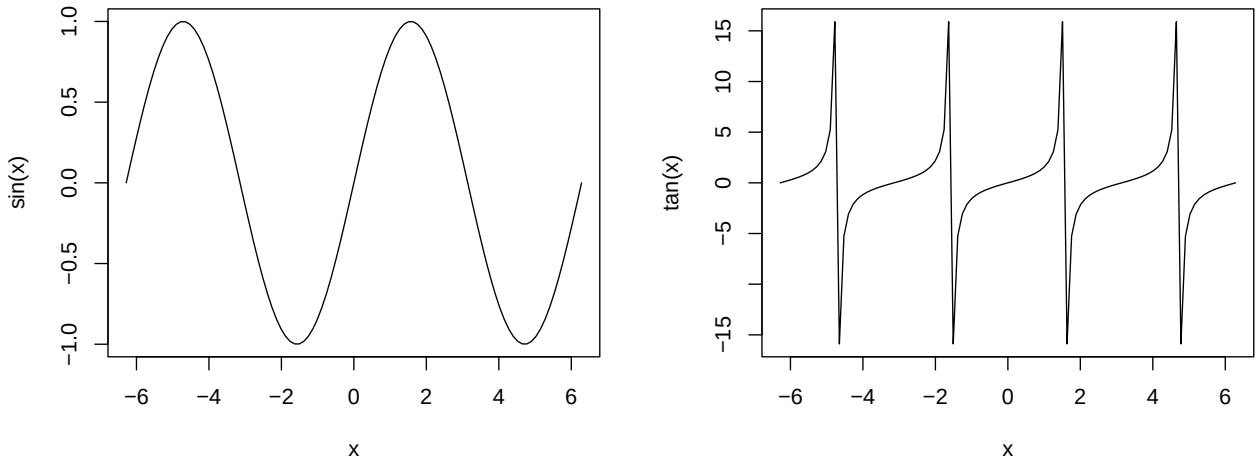



The outer margins (by default 0) can be set using the `oma` and `omi` arguments analogous as the `mar` and `mai` arguments. Text to the outer margins can be added using the `mtext` function.

**3.1.5.3 Multiple plots in one figure** There are two possibilities to place several plots in one figure in base R. The first is by simply setting the `mfrow` or `mfcol` parameters, the second is using the function `layout()`.

`mfrow` and `mfcol` allow to put several figures into one window. Each figure still has its own plotting area and margins, but in addition one can add optionally a surrounding overall outer margin. To do that one has to define an array which sets the size of the multiple figures environment. The two functions `mfcol` and `mfrow` define such an environment, the only difference is, that `mfcol` fills the array by columns and `mfrows` by rows.

```
par(mfrow=c(1,2))
curve(sin, -2*pi, 2*pi)
curve(tan, -2*pi, 2*pi)
```

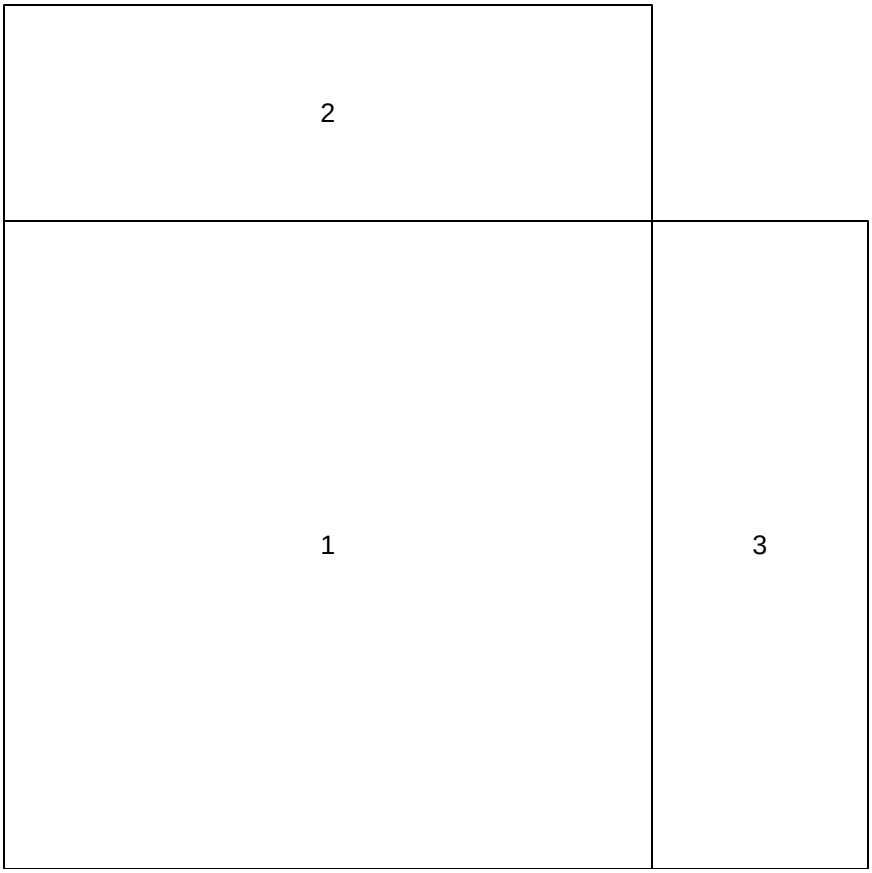



There exists a special function to specify complex plot arrangements: `layout()`.

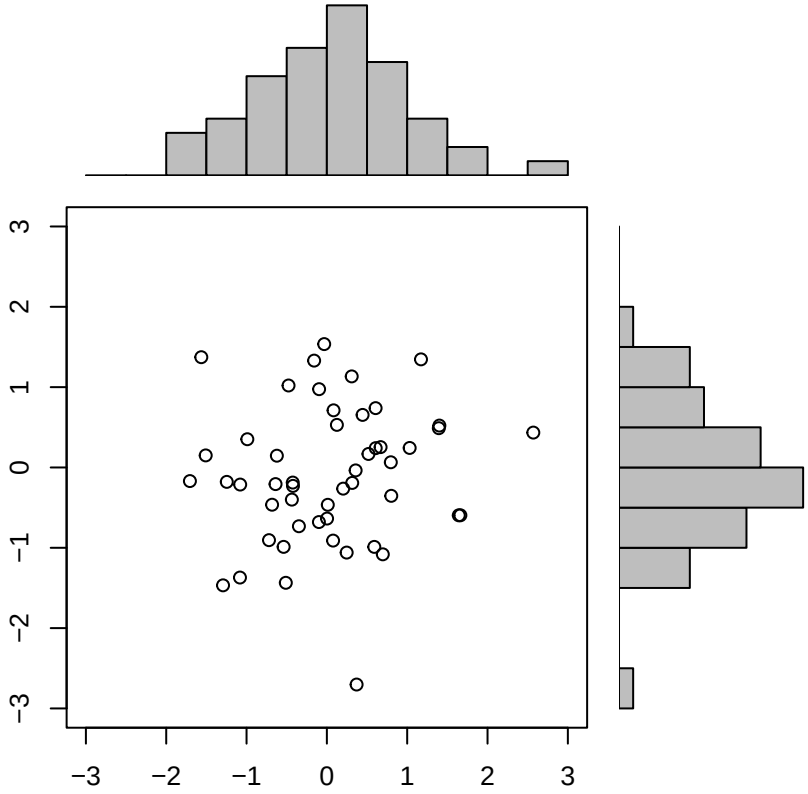
```
layout(mat, widths = rep.int(1, ncol(mat)),
       heights = rep.int(1, nrow(mat)), respect = FALSE)
```

It divides the device up into as many rows and columns as there are in matrix `mat`, with the column-widths and the row-heights specified in the respective arguments.

```
x <- pmin(3, pmax(-3, stats::rnorm(50)))
y <- pmin(3, pmax(-3, stats::rnorm(50)))
xhist <- hist(x, breaks = seq(-3,3,0.5), plot = FALSE)
yhist <- hist(y, breaks = seq(-3,3,0.5), plot = FALSE)
top <- max(c(xhist$counts, yhist$counts))
xrange <- c(-3, 3)
yrange <- c(-3, 3)
nf <- layout(matrix(c(2,0,1,3),2,2,byrow = TRUE), c(3,1), c(1,3), TRUE)
layout.show(nf)
```




```
par(mar = c(3,3,1,1))
plot(x, y, xlim = xrange, ylim = yrange, xlab = "", ylab = "")
par(mar = c(0,3,1,1))
barplot(xhist$counts, axes = FALSE, ylim = c(0, top), space = 0)
par(mar = c(3,0,1,1))
barplot(yhist$counts, axes = FALSE, xlim = c(0, top), space = 0,
        horiz = TRUE)
```




### 3.1.6 Device drivers

R can create for almost all types of driver display or printing devices graphics. However, R has to be told before making the figure, which device should be applied - therefore the device driver has to be specified.

`help(Devices)` provides a list with all possible devices. The special device of interest is activated by calling its name and specifying the necessary options in the parentheses.
For instance:

```
jpeg(file="C:/Temp/figure.jpg", width=5, height=4, bg="white")
```

produces a .jpg file. To finish with a device, one should submit

```
dev.off()
```

In R several graphic devices can be used at the same time. To start a new device one calls its name. E.g. `windows()` opens a new graphic windows when running R under windows. Always the last opened device is the active one. To reactivate an older window one has to use the function `dev.set`. `dev.set(1)` would for example reactivate the first device. Plotting commands affect only the active device.

## 3.2 ggplots

According to Wickham (2010), a grammar of graphics is a tool that enables us to concisely describe the components of a graphic. Such a grammar allows us to move beyond named graphics (e.g., the “scatterplot” or “lineplot”) and gain insight into the deep structure that underlies statistical graphics. This article proposes a parameterization of the grammar of graphics, based around the idea of building up a graphic from multiple layers of data.

Consider a simple data set:

Table 16: Simple data set to plot.

| A | B | C | D |
|---|---|---|---|
| 2 | 3 | a | 1 |
| 1 | 2 | a | 2 |
| 4 | 5 | b | 6 |
| 9 | 10 | b | 10 |

Assume we want to create a scatterplot of $A$ and $B$ where variable $C$ is also mapped to the shape of the points.

The steps we would do to create the plot are the following:

- We would create another data set that reflects the mapping of the $x$-position in the graph to $A$, $y$-position in the graph to $B$ and shape parameter to $C$. $x$-position, $y$-position, and shape are examples of aesthetics. All other variables are not needed and would be removed. See Table~17.
- We could create different plots with these aesthetics (we could use bars, lines, points etc). These are examples of geometric objects.
- The next thing we need to do is to convert these numbers measured in data units to numbers measured in physical units, things that the computer can display. To do that we need to know that we are going to use linear scales and a Cartesian coordinate system. We can then convert the data units to aesthetic units, which have meaning to the underlying drawing system. Also categories are mapped to shapes such as circles and squares These transformations are the responsibility of scales
- In general, there is another step that one often uses: a statistical transformation. Here we are using the identity transformation, but there are many others that are useful, such as binning or aggregating.
- We might want to annotate the plot by a title.

Table 17: Simple data set with variables named as aesthetics.

| $x$ | $y$ | shape |
|---|---|---|
| 2 | 3 | a |
| 1 | 2 | a |
| 4 | 5 | b |
| 9 | 10 | b |

### 3.2.1 Components of the layered grammar of graphics

Components of the layered grammar of graphics:

- Layer
  - Data
  - Mapping
  - Statistical transformation (stat)
  - Geometric object (geom)

  - Position adjustment (position)

- Scale
- Coordinate system (coord)
- Faceting (facet)

#### 3.2.1.1 Layers

**3.2.1.1.1 Data and mapping** Data are obviously a critical part of the plot, but it is important to remember that they are independent from the other components: we can construct a graphic that can be applied to multiple datasets. Data are what turns an abstract graphic into a concrete graphic. Along with the data, we need a specification of which variables are mapped to which aesthetics. For example, we might map weight to x position, height to y position, and age to size.

In general there are many aesthetics to map to:

- x and y axis
- color (and fill color)
- style (line style, shapes in scatter plots)
- size
- alpha (transparency used when plotting)

**3.2.1.1.2 Statistical transformation** A statistical transformation, or stat, transforms the data, typically by summarizing them in some manner. For example, a useful stat is the smoother, which calculates the mean of y, conditional on x, subject to some restriction that ensures smoothness.

**3.2.1.1.3 Geometric object** Geometric objects, or geoms for short, control the type of plot that you create. For example, using a point geom will create a scatterplot, whereas using a line geom will create a line plot. We can classify geoms by their dimensionality:

- 0d: point, text,
- 1d: path, line (ordered path),
- 2d: polygon, interval.

Geoms are mostly general purpose, but do require certain outputs from a statistic. For example, the boxplot geom requires the position of the upper and lower fences, upper and lower hinges, the middle bar, and the outliers. Any statistic used with the boxplot needs to provide these values. Every geom has a default statistic, and every statistic a default geom. For example, the bin statistic defaults to using the bar geom to produce a histogram. Overriding these defaults will still produce a valid plot, but it may violate graphical conventions.

**3.2.1.1.4 Position adjustment** Sometimes we need to tweak the position of the geometric elements on the plot, when otherwise they would obscure each other.

#### 3.2.1.2 Scales

A scale controls the mapping from data to aesthetic attributes, and so we need one scale for each aesthetic property used in a layer. Scales are common across layers to ensure a consistent mapping from data to aesthetics. For example, the color gradient scale maps a segment of the real line to a path through a color space.

#### 3.2.1.3 Coordinate system

A coordinate system maps the position of objects onto the plane of the plot. Position is often specified by two coordinates (x, y), but could be any number of coordinates. The Cartesian coordinate system is the most common coordinate system for two dimensions, whereas polar coordinates and various map projections are used less frequently. For higher dimensions, we have parallel coordinates (a projective geometry), mosaic plots (a hierarchical coordinate system), and linear projections onto the plane. Coordinate systems affect all position variables simultaneously and differ from scales in that they also change the appearance of the geometric objects. For example, in polar coordinates, bar geoms look like segments of a circle. Additionally, scaling is performed before statistical transformation, whereas coordinate transformations occur afterward.

#### 3.2.1.4 Faceting

The faceting specification describes which variables should be used to split up the data, and how they should be arranged.

### 3.2.2 Examples

These examples have been adapted from materials by Ursula Laa.

As an example we will use the package **palmerpenguins**.

```
install.packages("palmerpenguins")
```

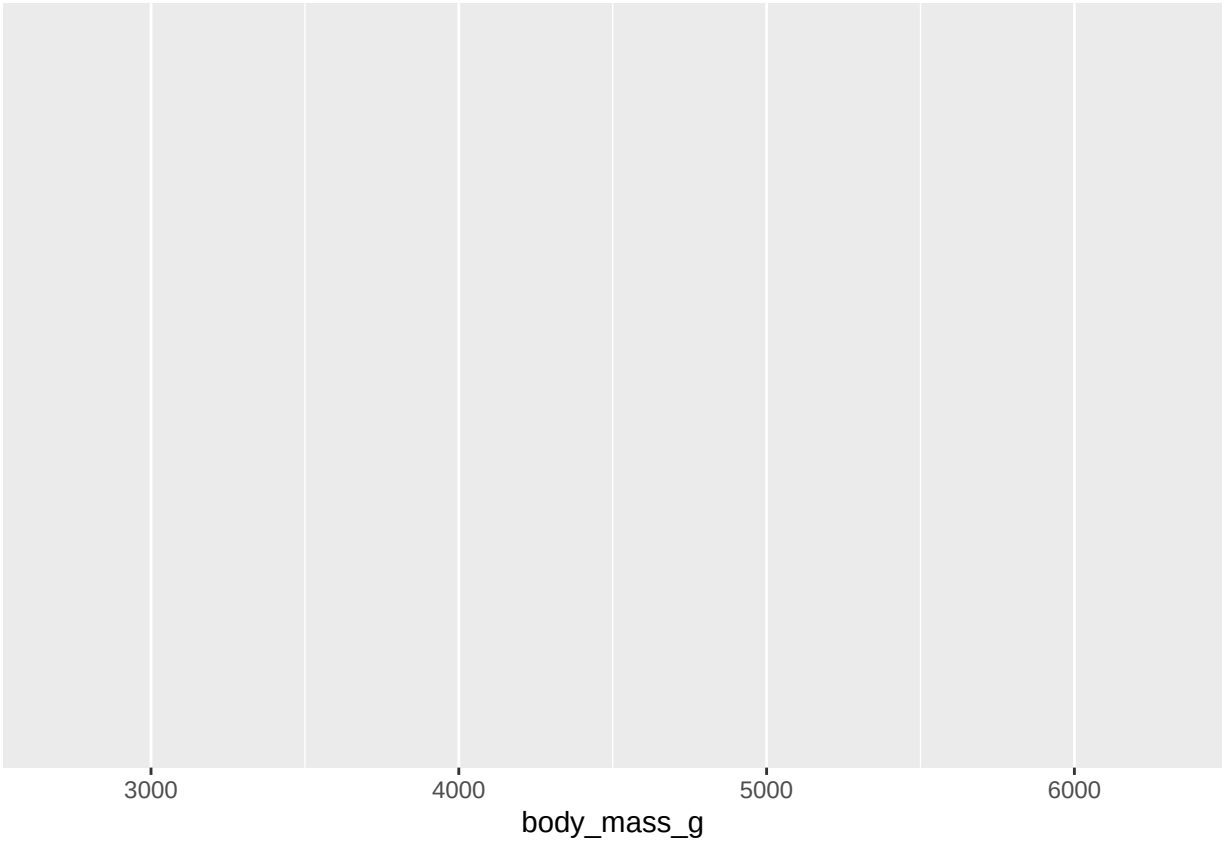
```
library("ggplot2")
library("palmerpenguins")
```

We start by passing our data set to the ggplot function

```
ggplot(penguins)
```

We next define the aesthetic mapping (or variable mapping)

```
ggplot(penguins) +
  aes(x = body_mass_g)
```

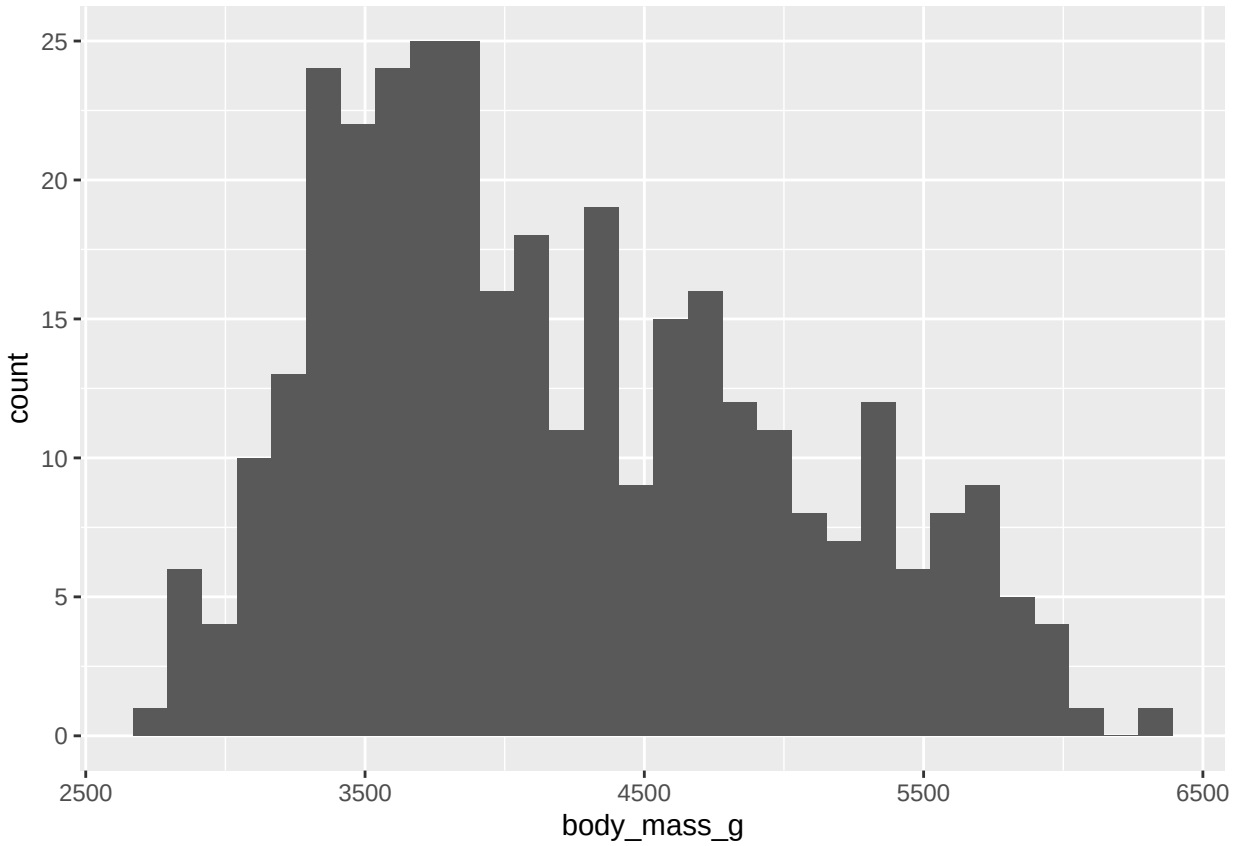



Finally we specify the geom we want to draw

```
ggplot(penguins) +
  aes(x = body_mass_g) +
  geom_histogram()
```

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

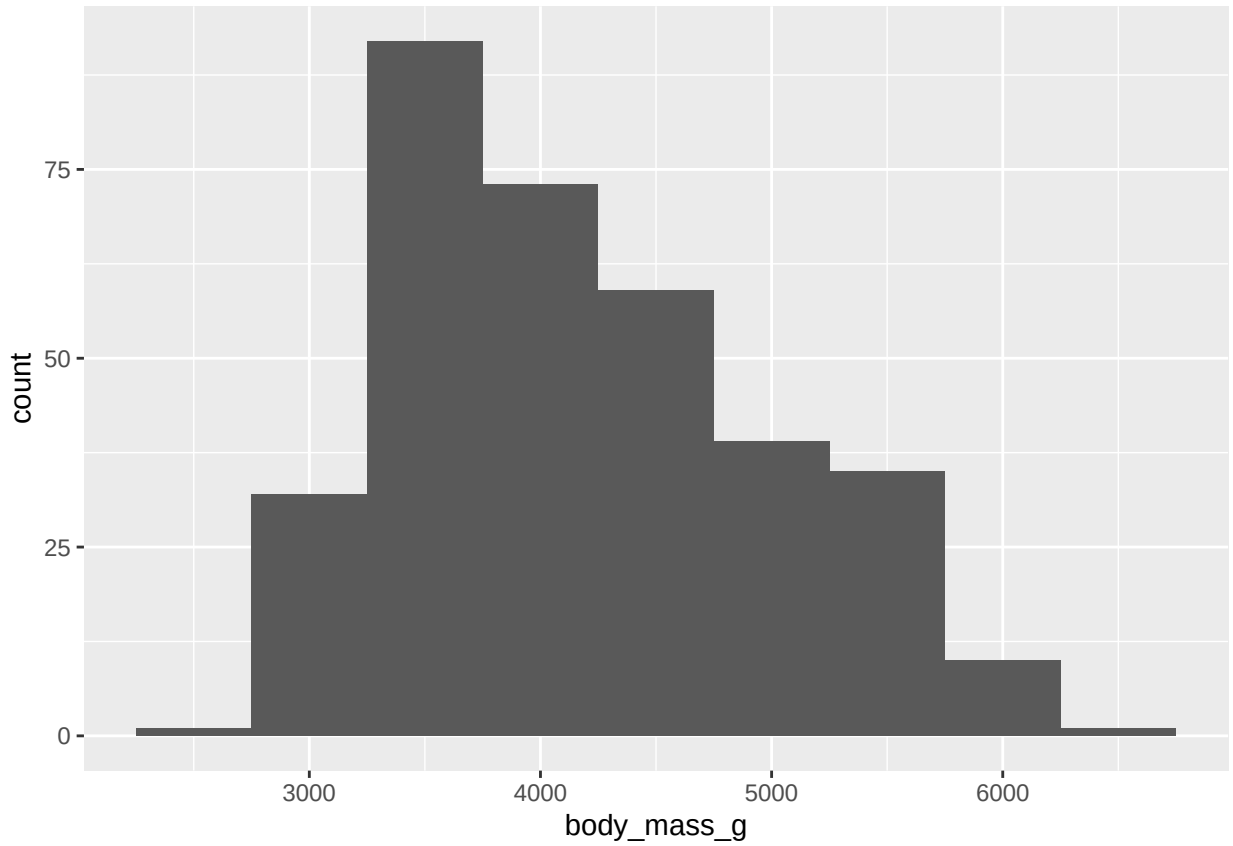
```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```




We can adjust the graph, e.g. to get wider bins

```
ggplot(penguins) +
  aes(x = body_mass_g) +
  geom_histogram(binwidth=500)
```

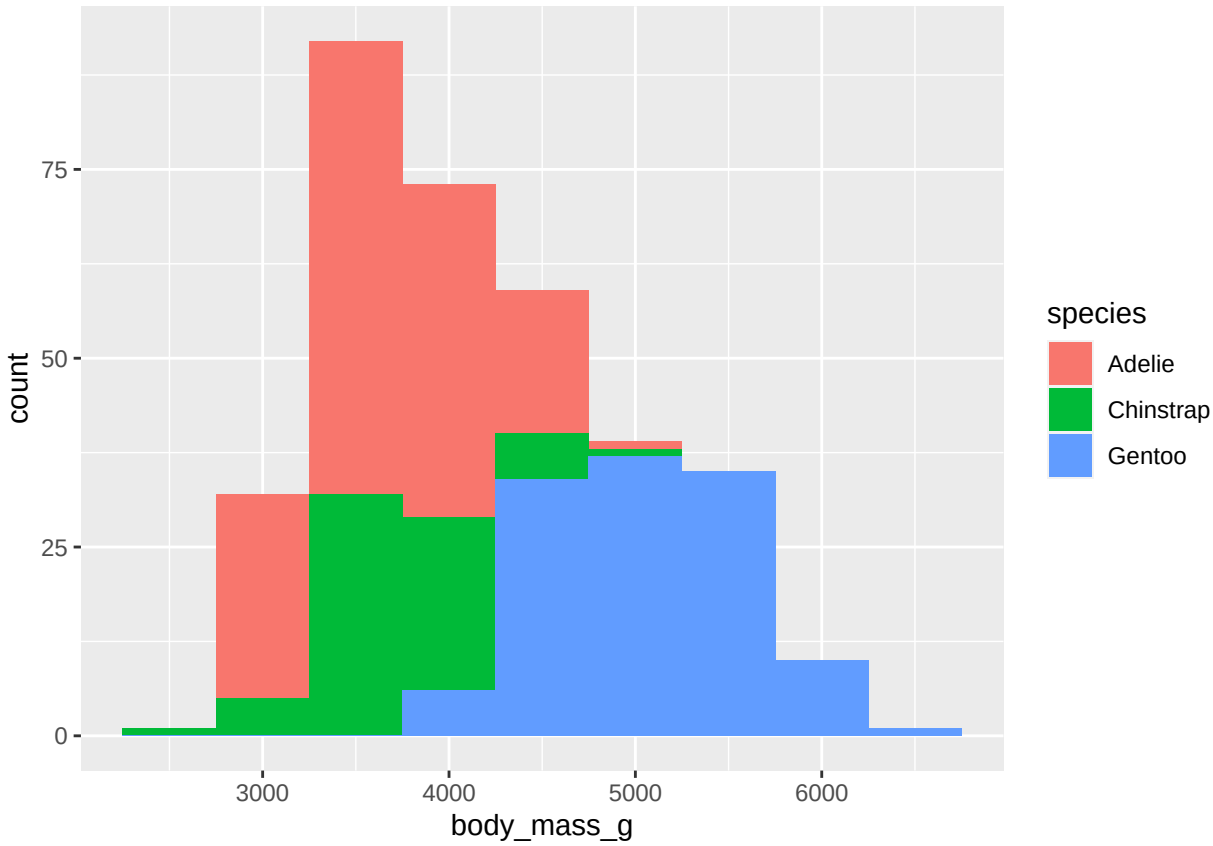
```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```




We can map the species to fill color

```
ggplot(penguins, mapping = aes(x = body_mass_g, fill = species)) +
  geom_histogram(binwidth = 500)
```

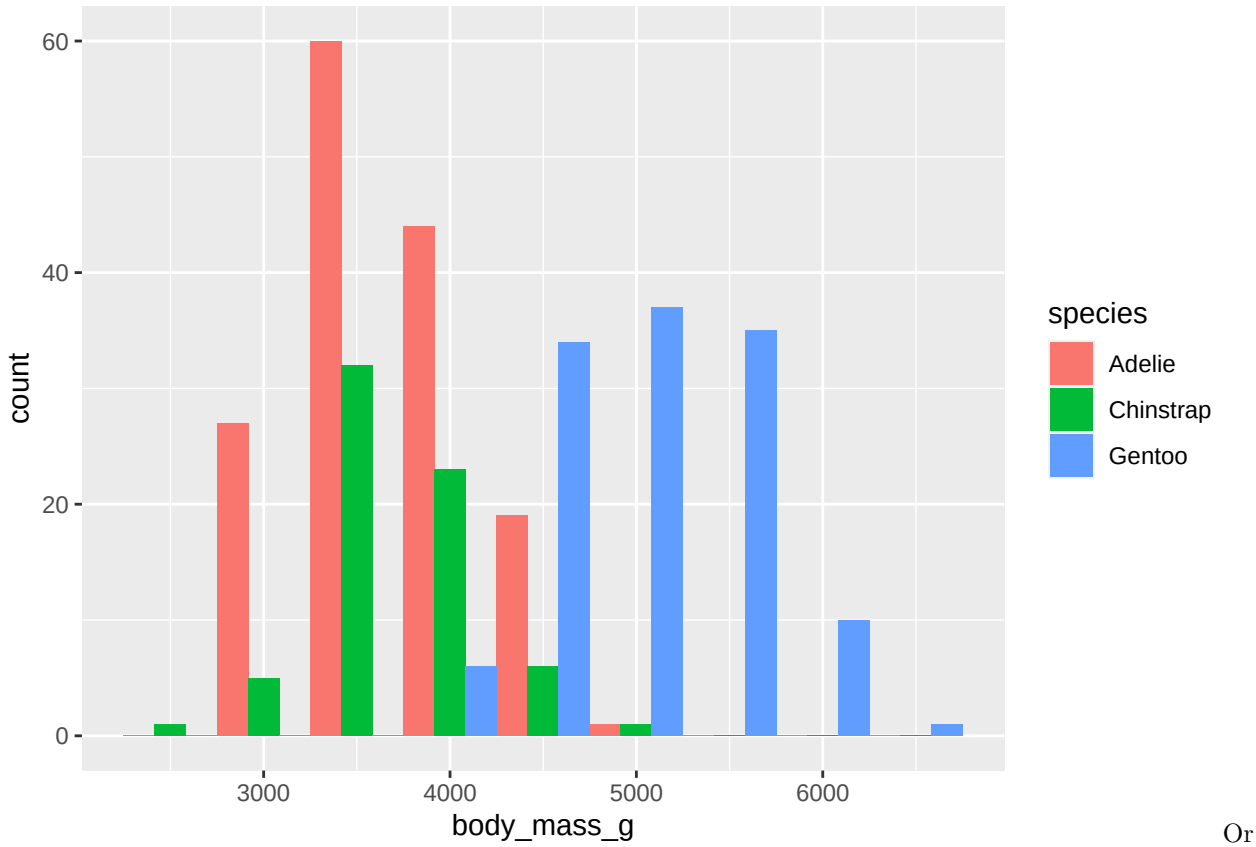
```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```




By default this is stacking the histograms of the different species. It might be easier to read if they are side-by-side instead using the position argument to better compare the distributions.

```
ggplot(penguins, mapping = aes(x = body_mass_g, fill = species)) +
  geom_histogram(binwidth = 500, position = "dodge")
```

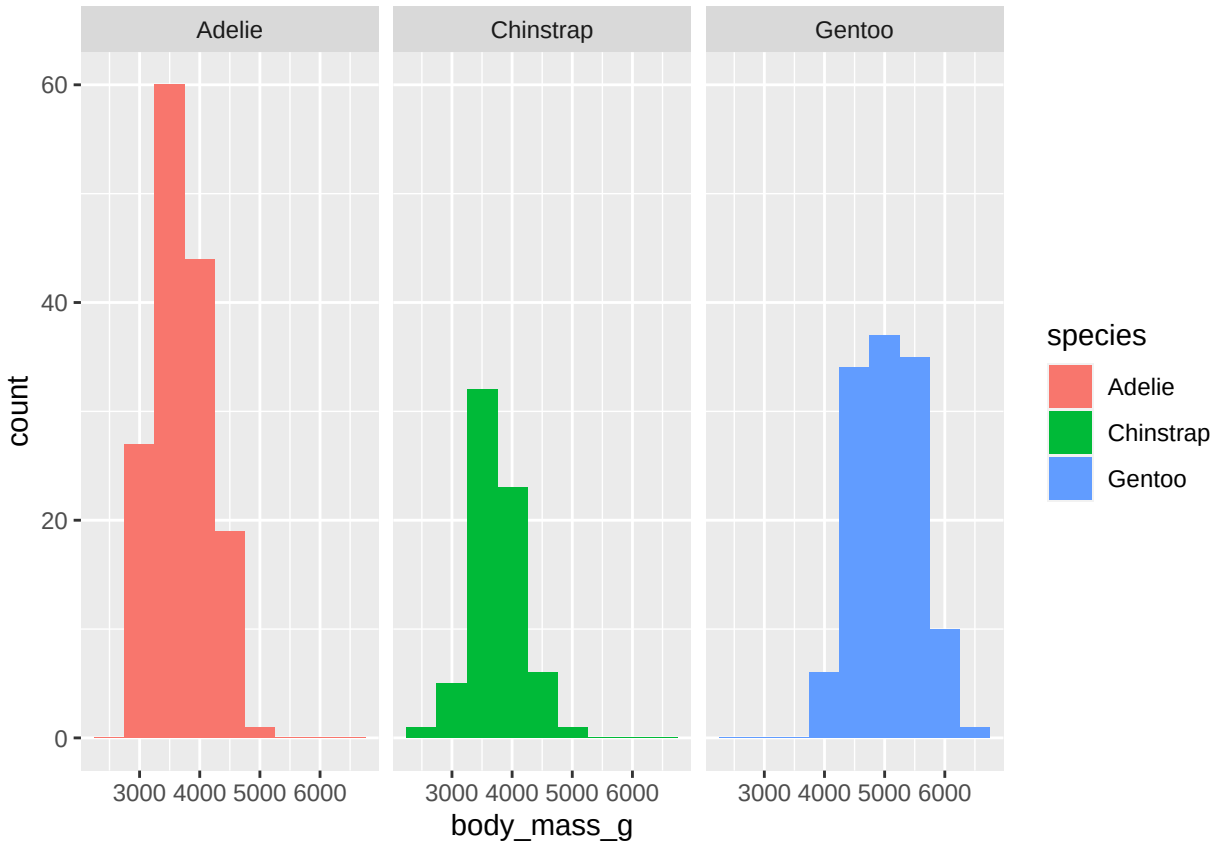
```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```




Or we can use faceting to split up the histograms instead:

```
ggplot(penguins, mapping = aes(x = body_mass_g, fill = species)) +
  geom_histogram(binwidth = 500) +
  facet_wrap(~species)
```

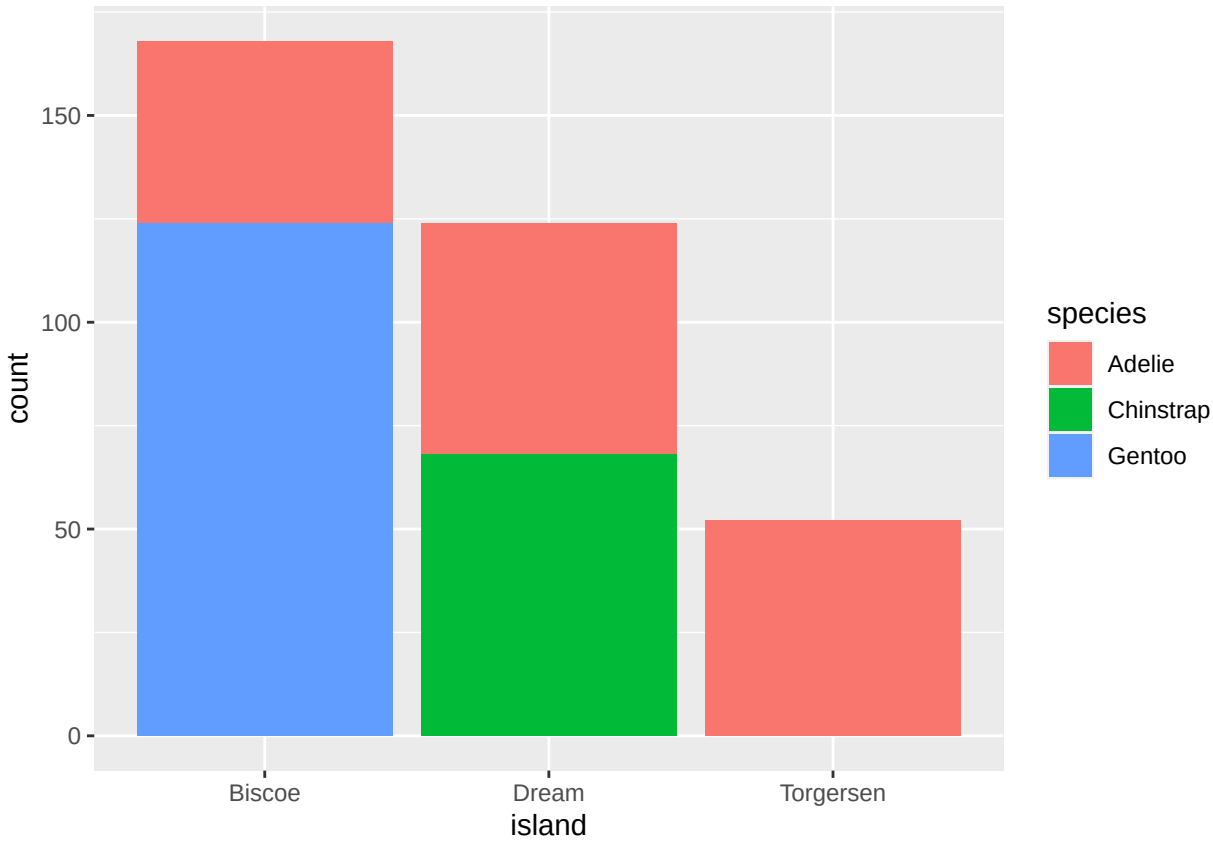
```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```




Let's now have a look at geoms other than `geom_hist`.

**Bar charts:**

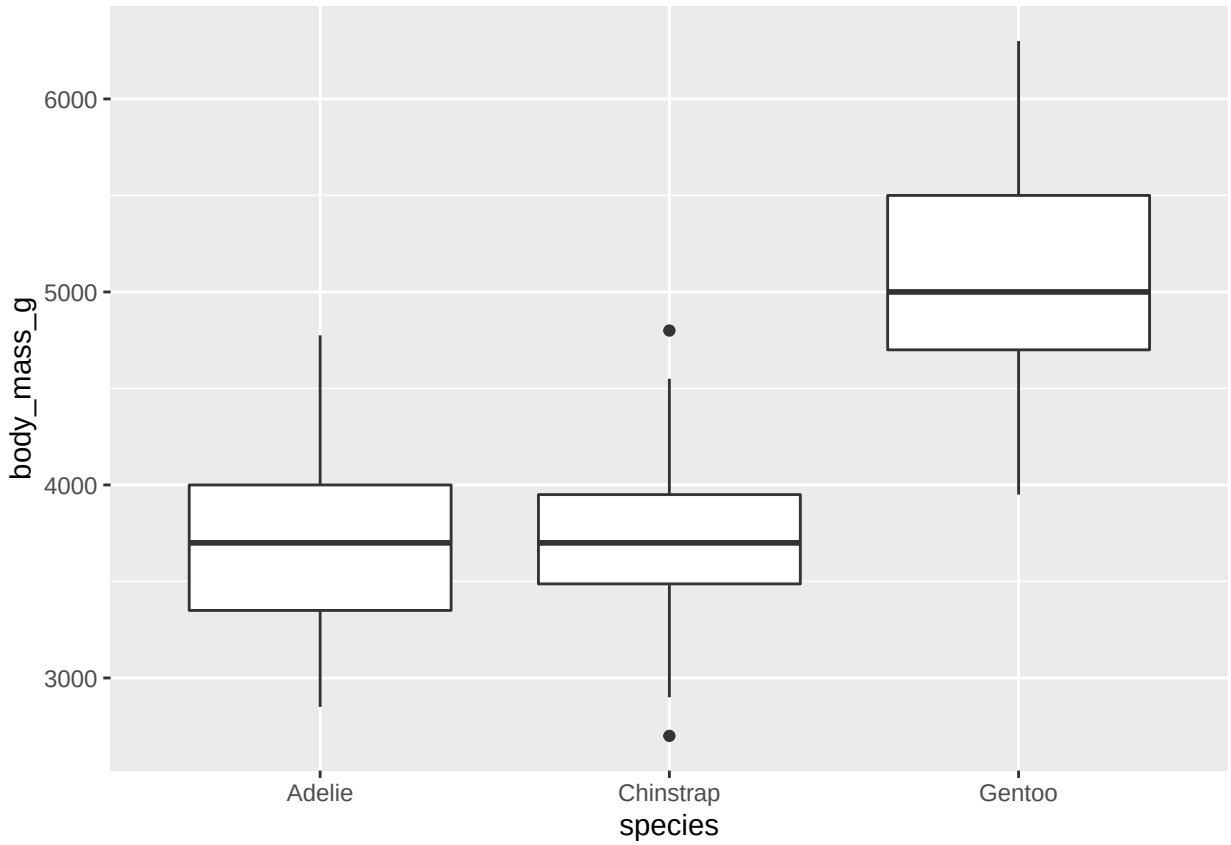
```
ggplot(penguins, mapping = aes(x = island, fill = species)) +
  geom_bar()
```




**Boxplots:**

```
ggplot(penguins, mapping = aes(x=species, y=body_mass_g)) +
  geom_boxplot()
```

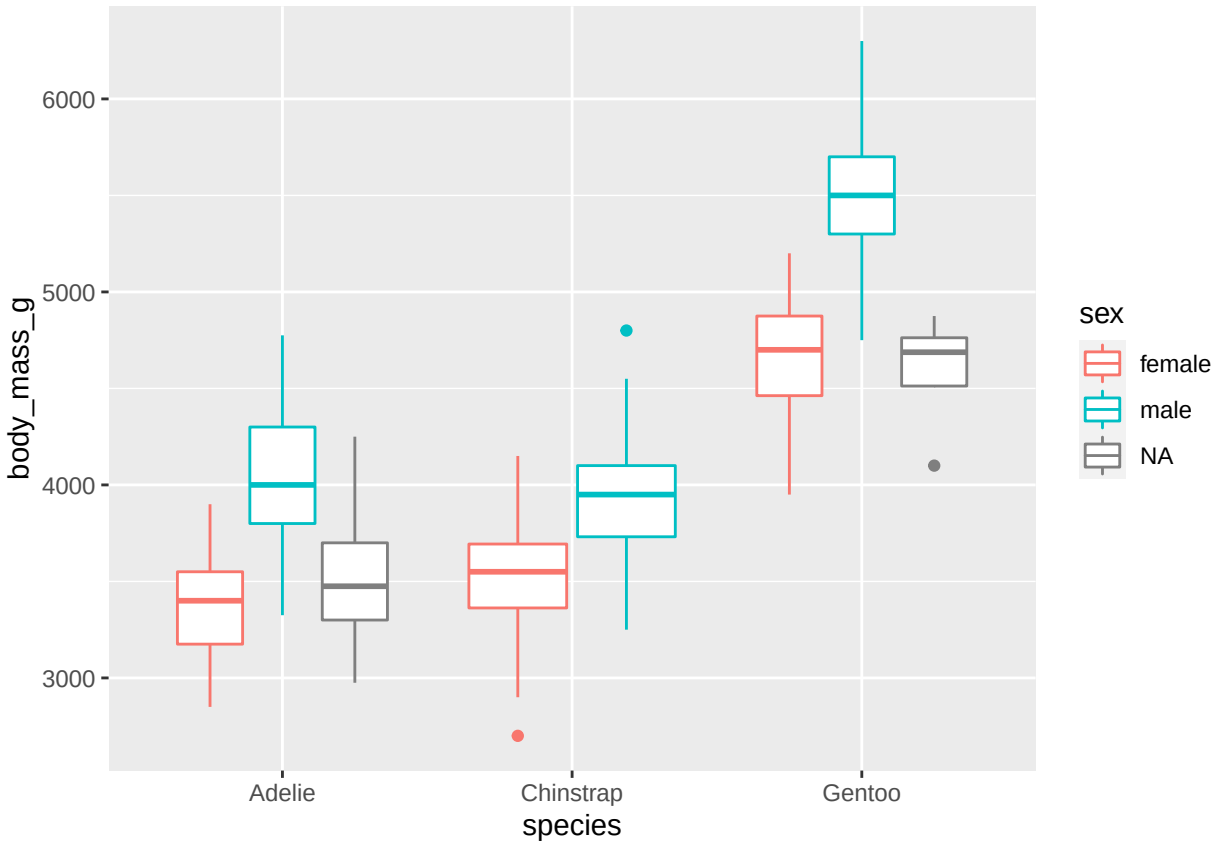
```
Warning: Removed 2 rows containing non-finite values (stat_boxplot).
```




Again, it is easy to change aspects of this plot to learn something new, for example

```
ggplot(penguins, mapping = aes(x=species, y=body_mass_g, color=sex)) +
  geom_boxplot()
```

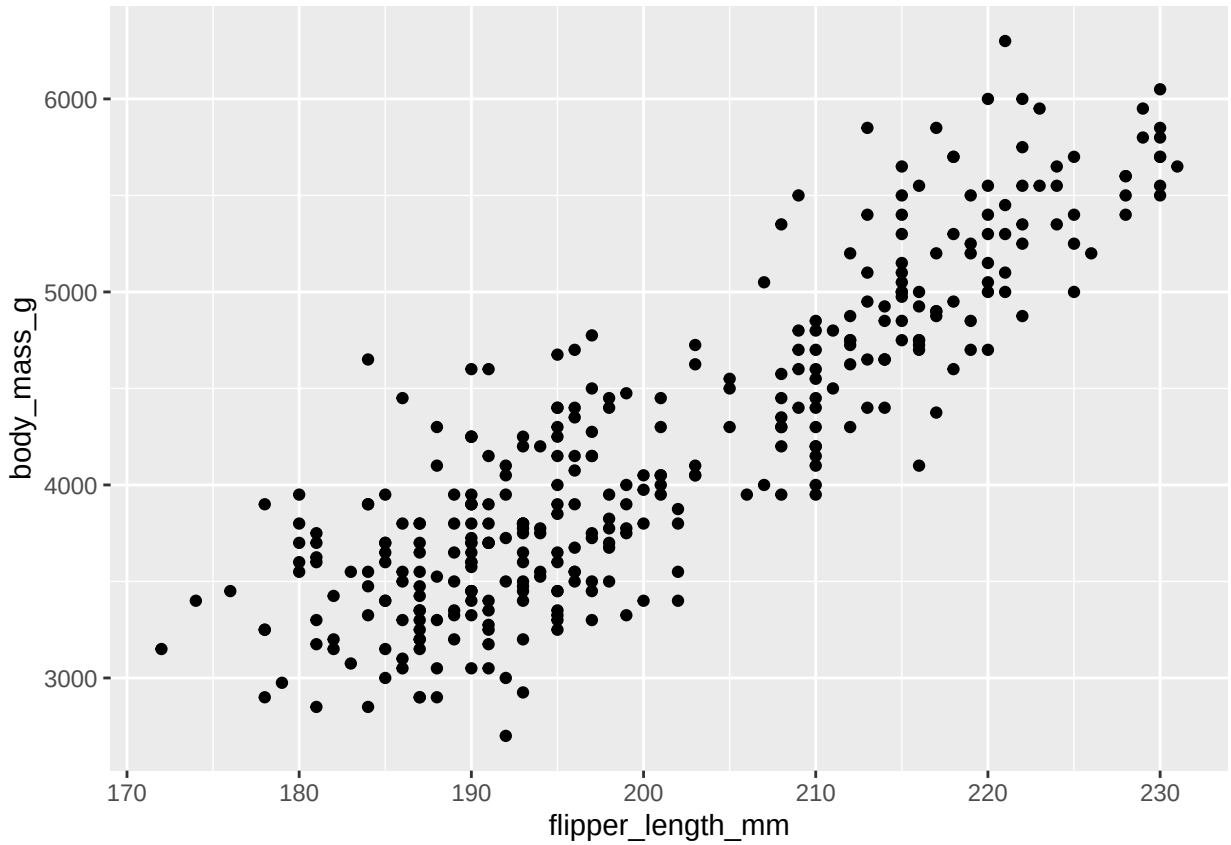
```
Warning: Removed 2 rows containing non-finite values (stat_boxplot).
```




**Scatter plots:** for example we would expect that the flipper length and the body mass are related

```
ggplot(penguins, mapping = aes(x=flipper_length_mm, y=body_mass_g)) +
  geom_point()
```

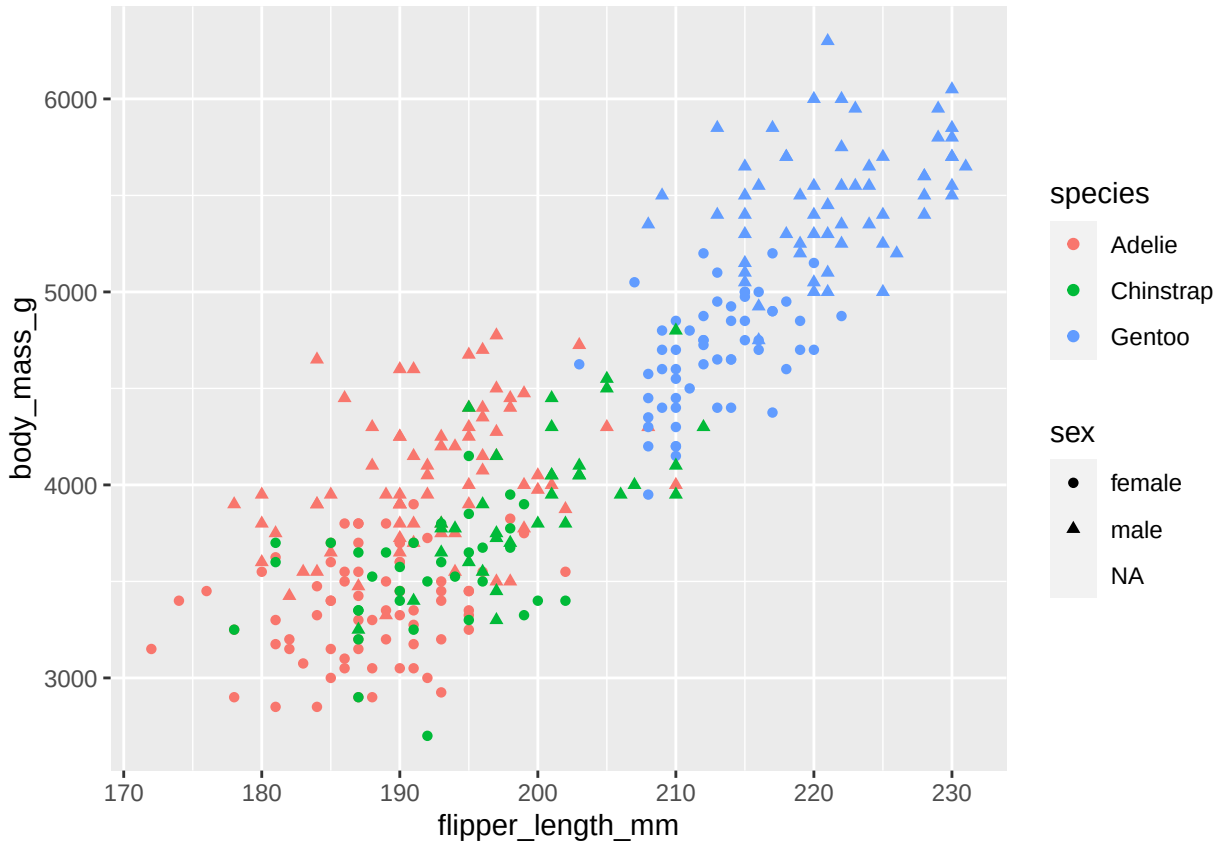
```
Warning: Removed 2 rows containing missing values (geom_point).
```




How does this depend on the species and sex of the penguins?

```
ggplot(penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g,
                               color = species, shape = sex)) +
  geom_point()
```

```
Warning: Removed 11 rows containing missing values (geom_point).
```




Building layers:

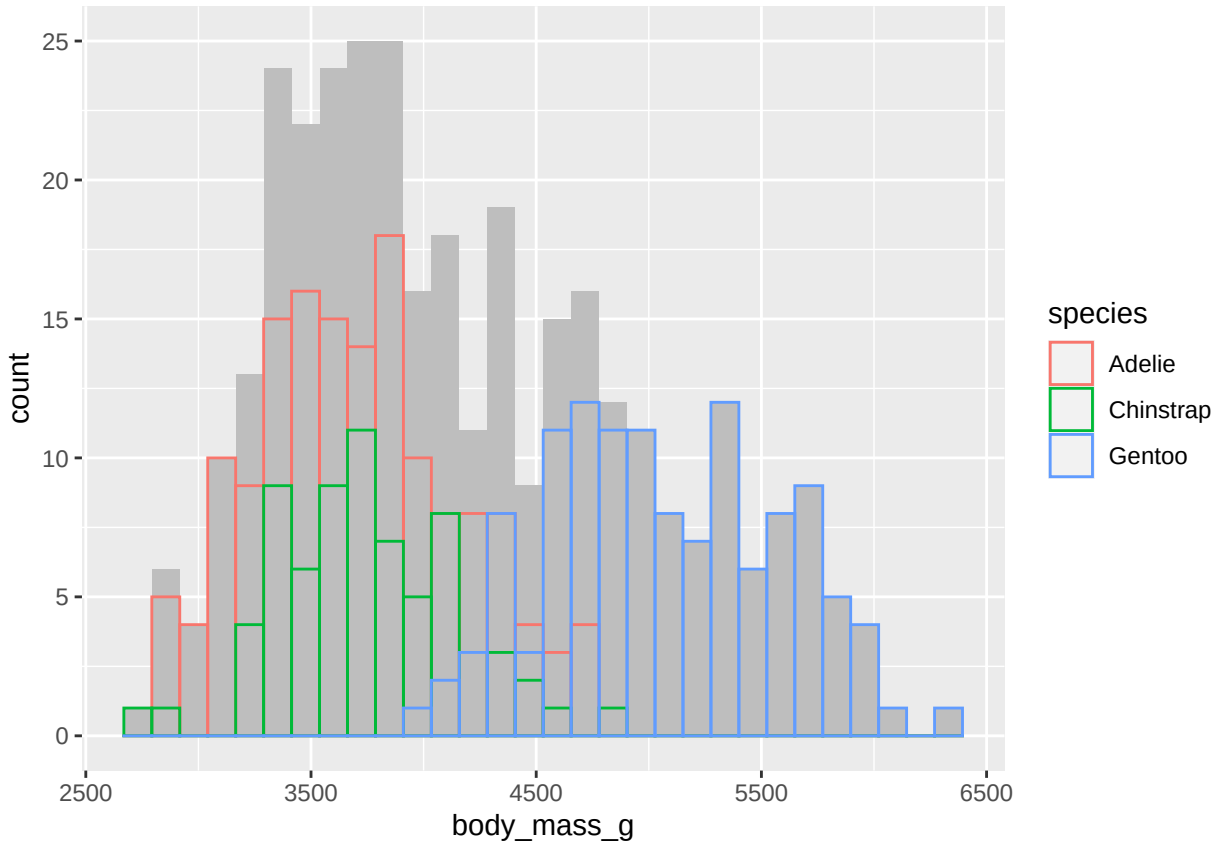
```
ggplot(penguins,
       aes(x=body_mass_g)) +
  geom_histogram(fill = "grey") +
  geom_histogram(aes(color = species),
                 position = "identity",
                 fill = NA)
```

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

```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```

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

```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```

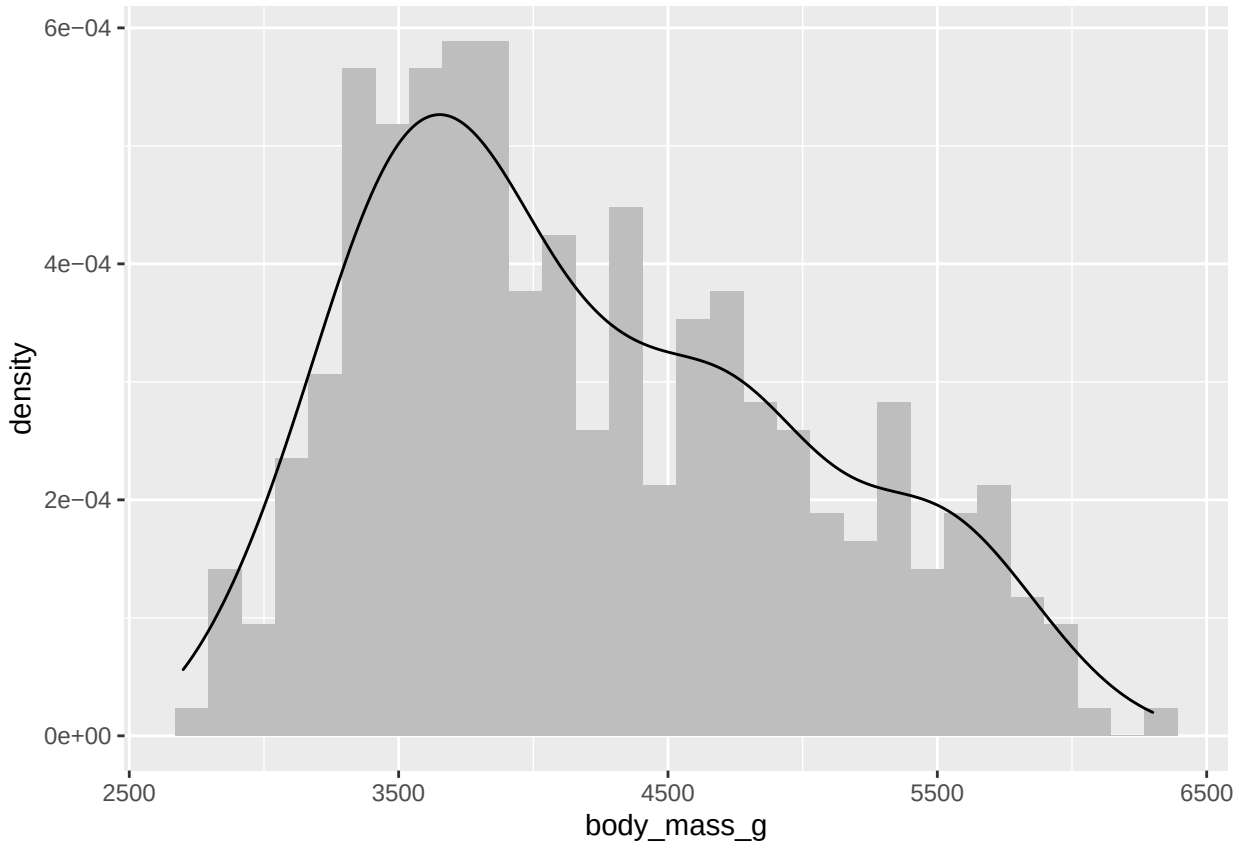



```
ggplot(penguins,
       aes(x = body_mass_g, y = stat(density))) +
  geom_histogram(fill = "grey") +
  geom_density()
```

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

```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```

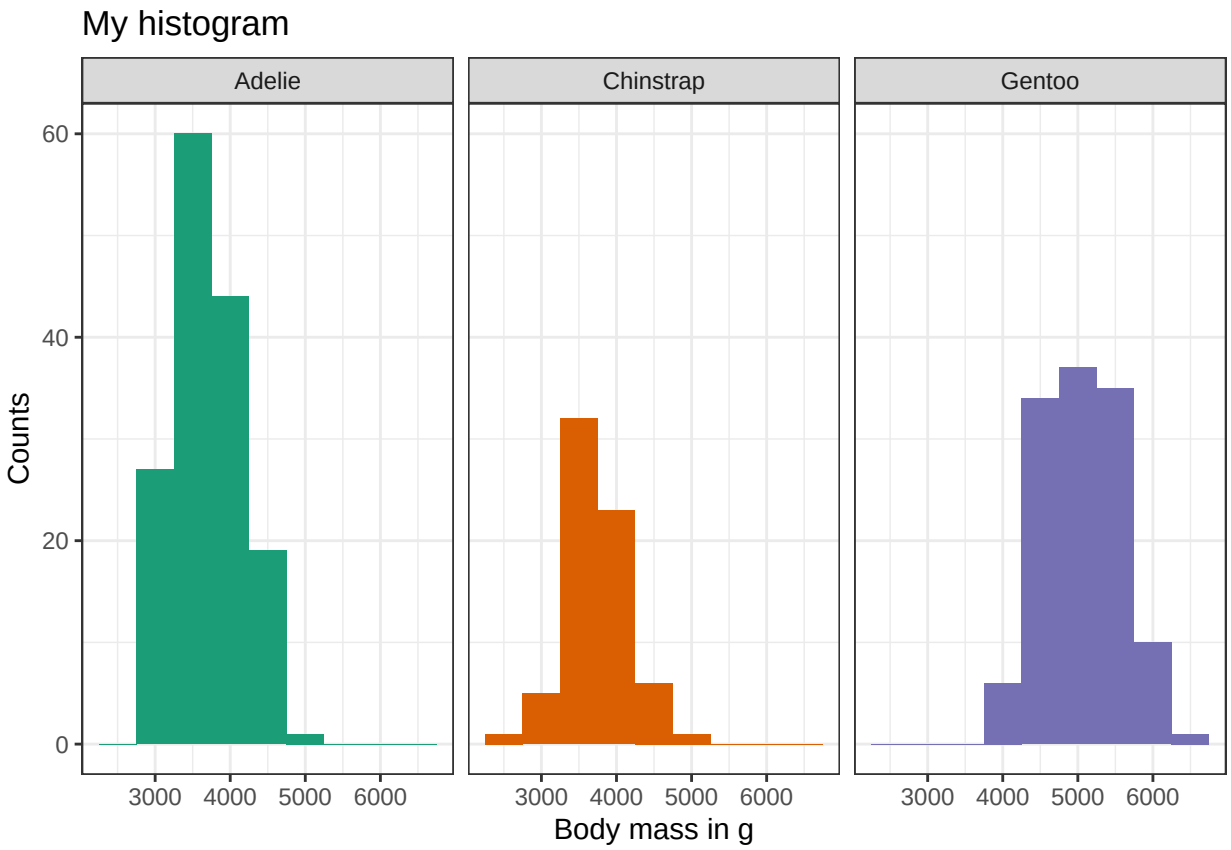
```
Warning: Removed 2 rows containing non-finite values (stat_density).
```




Customizing the graph: * setting a title with `ggtitle()` * setting axis labels with `xlab()` and `ylab()` * modifying the theme with `theme()` or using pre-defined themes like `theme_bw()` * changing color scales with the `scale_color_` and `scale_fill_` functions * change to using polar coordinates with `coord_polar()`

```
ggplot(penguins) +
  aes(x = body_mass_g, fill = species) +
  geom_histogram(binwidth = 500) +
  facet_wrap(~ species) +
  theme_bw() +
  ggtitle("My histogram") +
  xlab("Body mass in g") +
  ylab("Counts") +
  scale_fill_brewer(palette = "Dark2") +
  theme(legend.position = "none")
```

```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```

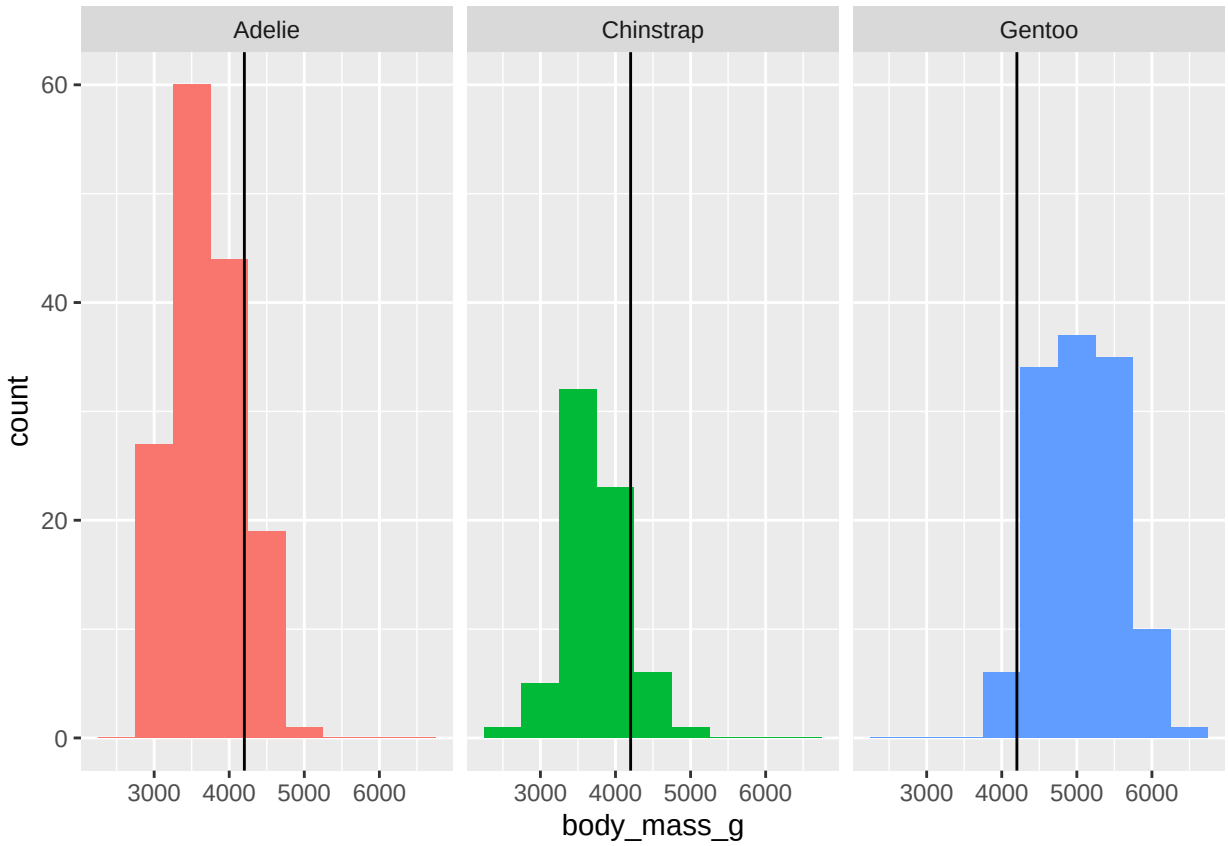



### Common elements in facets

We might want to have a common element across all facets. For example we might want to draw a vertical line in the histograms in all facets corresponding to the overall mean in the data set. For this, we need to create a new data frame which will contain a row for each species with the average body mass for the whole data set repeated for all species. The vertical line can be drawn using `geom_vline()`. For `geom_vline()` this new data set will be used.

```
avg_body_mass <- mean(penguins$body_mass_g, na.rm=TRUE)
dat_mean <- data.frame(body_mass_g = rep(avg_body_mass, 3),
                       species = c("Adelie", "Chinstrap", "Gentoo"))
ggplot(penguins) +
  aes(x = body_mass_g, fill = species) +
  geom_histogram(binwidth = 500) +
  geom_vline(data = dat_mean, aes(xintercept = body_mass_g)) +
  facet_wrap(~ species) +
  theme(legend.position = "none")
```

```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```



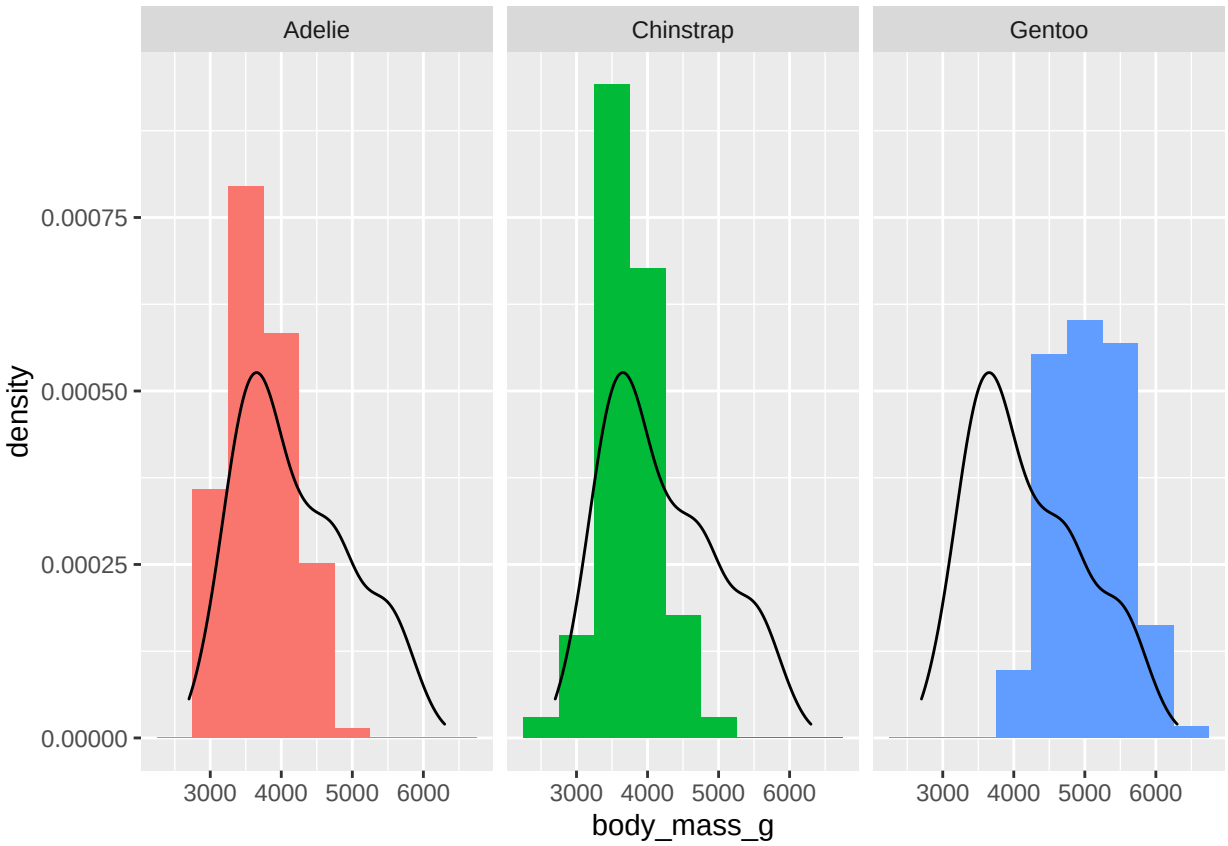

Note that one can use different data sets in the different geoms.

Let's consider another example the density over the whole sample is plotted in each facet. Again, we need to create a data set which will repeat the whole vector of the body mass three times, for each species.

```
dat_density <- data.frame(body_mass_g = rep(penguins$body_mass_g, 3),
                          species = rep(c("Adelie", "Chinstrap", "Gentoo"),
                                        each = nrow(penguins)))
ggplot(penguins) +
  aes(x = body_mass_g, y = stat(density)) +
  geom_histogram(binwidth = 500, aes(fill = species)) +
  geom_density(data = dat_density) +
  facet_wrap(~ species) +
  theme(legend.position = "none")
```

```
Warning: Removed 2 rows containing non-finite values (stat_bin).
```

```
Warning: Removed 6 rows containing non-finite values (stat_density).
```




## 3.3 Interactive graphics using gganimate and ggplotly

Once you have build your graphic with **ggplot2**, turning it into an animation or an interactive graph can be just one extra line of code.

Run the following chunks of code on your computer to see how the plots look like.

**3.3.0.1 gganimate** Using **gganimate** you can add `transition_` functions to generate an animation from your ggplot

For example, `transition_states()` splits up plot data by a discrete variable and animates between the different states.

```
# install.packages("gganimate")
library("gganimate")
anim <- ggplot(penguins, mapping = aes(x = flipper_length_mm,
                                       y = body_mass_g,
                                       color = species)) +
  geom_point() +
  transition_states(species)
anim
```

It might also be useful to show what each frame in the animation represents.

```
anim +
  ggtitle('Now showing {closest_state}',
          subtitle = 'Frame {frame} of {nframes}')
```

Using the `enter_` and `exit_` functions you can also fine tune how the data appears and disappears from the plot.

```
anim +
  enter_fade() +
  exit_shrink()
```

As another example consider the gapminder data:

```
# install.packages("gapminder")
library("gapminder")

ggplot(gapminder,aes(x = gdpPercap, y=lifeExp, size = pop, colour = country)) +
  geom_point(show.legend = FALSE, alpha = 0.7) +
  scale_color_viridis_d() +
  scale_size(range = c(2, 12)) +
  scale_x_log10() +
  labs(x = "GDP per capita", y = "Life expectancy")+
  transition_time(year) +
  labs(title = "Year: {frame_time}")
```

**3.3.0.2 plotly** The **plotly** R package serializes **ggplot2** figures into Plotly's universal graph JSON. Using the `ggplotly()` function from the **plotly** package you can generate an interactive graph, e.g. with tooltip text. The function `ggplotly()` will crawl the ggplot2 figure, extract and translate all of the attributes of the ggplot2 figure into JSON (the colors, the axes, the chart type, etc), and draw the graph with plotly.js.

```
# install.packages("plotly")
library("plotly")
p <- ggplot(penguins,
            mapping = aes(x = flipper_length_mm,
                          y = body_mass_g,
                          color = species)) +
  geom_point()

pltly <- ggplotly(p, width=600, height=400)
```

You can customize the plots by using `layout()`, which specifies interaction modes, and `style()`, for modifying data-level attributes.

With `layout()` you can modify aspects of the layout such as change default hovermode behavior, stylizing hover labels (hoverlabel), changing click+drag mode (dragmode) and/or constraining rectangular selections (dragmode='select') vertically or horizontally (selectdirection), as well as add dropdowns https://plot.ly/r/dropdowns/, sliders https://plot.ly/r/sliders/, and rangesliders.

```
pltly <- pltly %>% layout(dragmode = "pan")
pltly
```

With `style()` you can control the tooltip content, turn hovering on/off or add marker points to lines.

In this example, information will be provided when hovering over the red group, but not the other two:

```
pltly <- pltly %>% style(p, hoverinfo = "none", traces = 2:3)
```

# 4 Part 4: Shiny apps

Shiny is a framework for creating web applications using R code. It is designed primarily with data scientists in mind, and to that end, one can create pretty complicated Shiny apps with no knowledge of HTML, CSS, or JavaScript.

It is used in almost as many niches and industries as R itself is, and it is seen as an integral part in the communication of statistical concepts and results. It's used in academia as a teaching tool for statistical concepts and also by companies who want to set up realtime metrics dashboards that incorporate advanced analytics.

**shiny** (Chang et al. 2021) is an R package which can be installed as usual:

```
install.packages("shiny")
```

```
library("shiny")
```

A variety of templates and examples can be found at https://shiny.rstudio.com/gallery/. In this chapter we will start with a simple example and then move on to a more complex app.

## 4.1 Simple interactive app

There are several ways to create a Shiny app. The simplest is to create a new directory for your app, and put a single file called `app.R` in it. This `app.R` file will be used to tell Shiny both how your app should look, and how it should behave. A convenient way in RStudio is to create a new directory and an app.R file containing a basic app in one step by clicking `File | New Project`, then selecting New Directory and Shiny Web Application. This will create a template file which can be modified.

Try it out and add the following code to `app.R`:

```
library(shiny)
ui <- fluidPage(
  "Hello, world!"
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

The code has four parts:

- It calls `library(shiny)` to load the **shiny** package.
- It defines the user interface (UI), the HTML webpage that humans interact with. In this case, it's a page containing the words "Hello, world!" This is also referred to as *front-end*.
- It specifies the behavior of the app by defining a server function. This is also referred to as *back-end*. It's currently empty, so our app doesn't do anything for now.
- It executes `shinyApp(ui, server)` to construct and start a Shiny application from UI and server.

There are a few ways you can run this app:

- Click the Run App button in the document toolbar.
- Use a keyboard shortcut: Cmd/Ctrl + Shift + Enter.
- If you are not using RStudio, you can `source()` the whole document, or call `shiny::runApp()` with the path to the directory containing `app.R`.

Before you close the app, go back to RStudio and look at the R console. It says something like

```
Listening on http://127.0.0.1:3827
```

This tells you the URL where your app can be found. You can enter that URL into any compatible2 web browser to open another copy of your app.

Also notice that R is busy. While a Shiny app is running, you can't run new commands at the R console until the Shiny app stops.

You can stop the app and return access to the console using any one of these options:

- Click the stop sign icon on the R console toolbar.
- Click on the console, then press Esc (or press Ctrl + C if you're not using RStudio).
- Close the Shiny app window.

Now let us make a slightly less trivial app. The UI contains two parts: one where the user can choose an existing data set in R and one where, for the chosen data set, the summary is shown. The server takes the input from the user and generates the summary to be shown in the UI.

```
library(shiny)
ui <- fluidPage(
  selectInput("dataset", "Choose an existing dataset in R:",
              choices = ls("package:datasets")),
  verbatimTextOutput("summary")
)
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })
}
shinyApp(ui, server)
```

```
PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, please ma
```

## 4.2 Basic UI

The basic UI is built from input and output functions.

- The input functions are used to insert inputs from the user into your app.
- The output functions are used create placeholders in the UI that are later filled by the server function.

### 4.2.1 Inputs

All input functions have the same first argument: `inputId`. This is the identifier used to connect the front end with the back end: if your UI has an input with ID `"name"`, the server function will access it with `input$name`.

The `inputId` must be a simple string that contains only letters, numbers, and underscores (no spaces, dashes, periods, or other special characters allowed!). Name it like you would name a variable in R. Also, it must be unique. Otherwise there is no way to refer to this control in the server function.

Most input functions have a second parameter called `label`. This is used to create a human-readable label for the control. The third parameter is typically `value`, which, where possible, lets you set the default value. The remaining parameters are unique to the control.

The following contains an overview on the main input functions implemented in **shiny**.

#### 4.2.1.1 Numeric inputs

To collect numeric values, create a constrained text box with `numericInput()` or a slider with `sliderInput()`:

```
ui <- fluidPage(
  numericInput("num", "Number one", value = 0, min = 0, max = 100),
  sliderInput("num2", "Number two", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```

Sliders are extremely customisable. See `?sliderInput` and https://shiny.rstudio.com/articles/sliders.html for more details.

#### 4.2.1.2 Limited choices

There are two different approaches to allow the user to choose from a prespecified set of options: `selectInput()` and `radioButtons()`.

```
animals <- c("dog", "cat", "mouse", "bird", "other", "I do not like animals")

ui <- fluidPage(
  selectInput("state", "What's your favourite state?", state.name),
  radioButtons("animal", "What's your favourite animal?", animals)
)
```

Dropdowns created with `selectInput()` are more suitable for longer lists, while `radioButtons()` will show all possible choices.

You can also set `multiple = TRUE` in `selectInput()` to allow the user to select multiple elements.

```
ui <- fluidPage(
  selectInput(
    "state", "What's your favourite state?", state.name,
    multiple = TRUE
  )
)
```

There is no way to select multiple values with radio buttons, but for this purpose you can use `checkboxGroupInput()`.

```
ui <- fluidPage(
  checkboxGroupInput("animal", "What animals do you like?", animals)
)
```

#### 4.2.1.3 Dates

Collect a single day with `dateInput()` or a range of two days with `dateRangeInput()`:

```
ui <- fluidPage(
  dateInput("dob", "When were you born?"),
  dateRangeInput("holiday", "When do you want to go on vacation next?")
)
```

#### 4.2.1.4 Free text

Collect small amounts of text with `textInput()`, passwords with `passwordInput()`, and paragraphs of text with `textAreaInput()`.

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "What's your password?"),
  textAreaInput("story", "Tell me about yourself", rows = 3)
)
```

#### 4.2.1.5 File upload

Allow the user to upload a file with `fileInput()`:

```
ui <- fluidPage(
  fileInput("upload", NULL)
)
```

`fileInput()` requires special handling on the server side, and is discussed later in the chapter.

#### 4.2.1.6 Action buttons

The user can perform an action with `actionButton()` or `actionLink()`:

```
ui <- fluidPage(
  actionButton("click", "Click me!"),
  actionButton("drink", "Drink me!", icon = icon("cocktail"))
)
```

Actions links and buttons are paired with `observeEvent()` or `eventReactive()` in the server function. We will learn more about them in the next part.

### 4.2.2 Outputs

Like inputs, outputs take a unique ID as their first argument: if the UI specification creates an output with ID `"plot"`, you can access it in the server function with `output$plot`.

Each output function on the front end is coupled with a `render` function in the back end. There are three main types of output, corresponding to the three things you usually include in a report:

- text,
- tables, and

- plots.

The following sections show you the basics of the output functions on the front end, along with the corresponding render functions in the back end.

**4.2.2.1 Text** Output regular text with `textOutput()` and fixed code and console output with `verbatimTextOutput()`.

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("code")
)
server <- function(input, output, session) {
  output$text <- renderText({
    "Hello friend!"
  })
  output$code <- renderPrint({
    summary(1:10)
  })
}
```

**4.2.2.2 Tables** There are two options for displaying data frames in tables:

- `tableOutput()` and `renderTable()` render a static table of data, showing all the data at once.
- `dataTableOutput()` and `renderDataTable()` render a dynamic table, showing a fixed number of rows along with controls to change which rows are visible.

```
ui <- fluidPage(
  tableOutput("static"),
  dataTableOutput("dynamic")
)
server <- function(input, output, session) {
  output$static  <- renderTable(head(mtcars))
  output$dynamic <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

**4.2.2.3 Plots** You can display any type of R graphic (base, ggplot2, or otherwise) with `plotOutput()` and `renderPlot()`:

```
ui <- fluidPage(
  plotOutput("plot", width = "400px")
)
server <- function(input, output, session) {
  output$plot <- renderPlot(plot(1:5), res = 96)
}
```

If you have an interactive graph you can use `plotlyOutput()` with `renderPlotly()`:

```
library(shiny)
library(plotly)
ui <- fluidPage(
  plotlyOutput("interactive_plot", width = "400px")
)
server <- function(input, output, session) {
  output$interactive_plot <- renderPlotly({
    p <- ggplot(mtcars, aes(x = hp, y = mpg, col = factor(am))) +
      geom_point()
    ggplotly(p)
  })
}
```

**4.2.2.4 Downloads** One can let the user download a file with `downloadButton()` or `downloadLink()`. We will not go into details here but if interested, check https://mastering-shiny.org/action-transfer.html#action-transfer

## 4.3 Basic reactivity

In Shiny, you express your server logic using reactive programming. The key idea of reactive programming is to specify a graph of dependencies so that when an input changes, all related outputs are automatically updated. This makes the flow of an app considerably simpler.

### 4.3.1 Server function

Shiny invokes the `server()` function each time a new session starts. Just like any other R function, when the `server` function is called it creates a new local environment that is independent of every other invocation of the function. This allows each session to have a unique state, as well as isolating the variables created inside the function.

Server functions take three parameters: `input`, `output`, and `session`.

**4.3.1.1 Input argument** The input argument is a list-like object that contains all the input data sent from the browser, named according to the `inputID`. For example, if the UI contains a numeric input control with an `inputID` of `count`:

```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)
```

then you can access the value of that input with `input$count`. It will initially contain the value 100, and it will be automatically updated as the user changes the value in the browser.

Unlike a typical list, input objects are **read-only**. If you attempt to modify an input inside the server function, you'll get an error (run this on your computer to see the error message):

```
server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
```

One more important thing about input: it's selective about who is allowed to read it. To read from an input, you must be in a reactive context created by a function like `renderText()` or `reactive()`.

This code illustrates the error you will get see if you make this mistake (run this on your computer to see the error message):

```
server <- function(input, output, session) {
  message("The value of input$count is ", input$count)
}

shinyApp(ui, server)
```

**4.3.1.2 Output argument** The argument `output` is very similar to `input`: it is a list-like object named according to the `outputID`. The main difference is that we use it for sending output instead of receiving input. The `output` object should be used together with a render function, as in the following simple example:

```
ui <- fluidPage(
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText("Hello human!")
}
```

The render function sets up a special reactive context that automatically tracks what inputs the output uses. It also converts the output of the R code into HTML suitable for display on a web page.

### 4.3.2 Reactive programming

Let's look at a simple app:

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
```

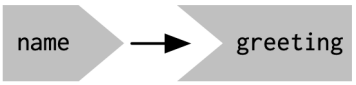
If we run the app and type in the name box, we see that the greeting updates automatically as you type (try it out). Note that one does not need to tell an output when to update, because Shiny does this automatically.

The style of programming used in Shiny is declarative, that means the programmer expresses higher-level goals or describes important constraints, and relies on someone else or on the software to decide how and/or when to translate that into action.

One of the strengths of declarative programming in Shiny is that it allows apps to be extremely lazy. A Shiny app will only ever do the minimal amount of work needed to update the output controls that you can currently see in the UI. Shiny's laziness has another important property. In most R code, you can understand the order of execution by reading the code from top to bottom. That doesn't work in Shiny, because code

is only run when needed. To understand the order of execution you need to instead look at the **reactive graph**, which describes how inputs and outputs are connected. The reactive graph contains one symbol for every input and output, and an input is connected to an output whenever the output accesses the input.

For the greeting app above the reactive graph is very simple an looks like this:




One more important component in the reactive graph are the **reactive expressions**. One can think of them as a tool that reduces duplication in the reactive code by introducing additional nodes into the reactive graph. For the greeting app we don't necessarily need a reactive expression, but let's see how we can use one:

```
server <- function(input, output, session) {
  string <- reactive(paste0("Hello ", input$name, "!"))
  output$greeting <- renderText(string())
}
```

The reactive graph becomes


Finally, note that due to laziness, the order of the lines in the code does not matter (might not be intuitive given the way code is usually run in R). So the following code yields the same reaction as the previous one:

```
server <- function(input, output, session) {
  output$greeting <- renderText(string())
  string <- reactive(paste0("Hello ", input$name, "!"))
}
```

## 4.4 Layout

Layout functions provide the high-level visual structure of an app. Layouts are created by a hierarchy of function calls, where the hierarchy in R matches the hierarchy in the generated HTML. For example, look at the following code for UI. It should be intuitively clear what it does.

```
fluidPage(
  titlePanel("Hello Shiny!"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Observations:", min = 0, max = 1000, value = 500)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

It will generate a classic app design: a title bar at top, followed by a sidebar (containing a slider) and main panel (containing a plot).

This section contains a brief introduction to the main layout functions in Shiny. For more information on the customization of Shiny apps see https://mastering-shiny.org/action-layout.html#action-layout and references therein.

### 4.4.1 Page functions

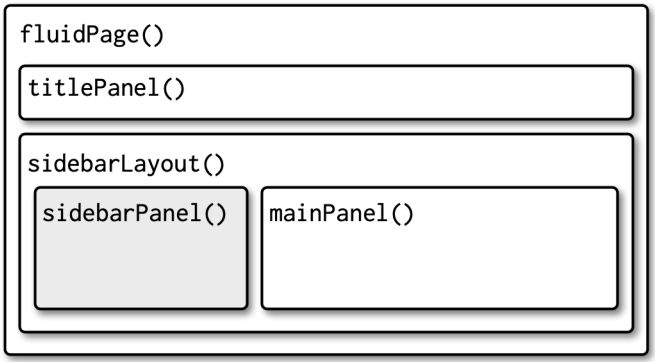
Function `fluidPage()`, which we have seen in the examples so far, sets up all the HTML, CSS, and JavaScript that Shiny needs.

In addition to `fluidPage()` Shiny provides a couple of other page functions that can come in handy in more specialised situations: `fixedPage()` (has a fixed maximum width, which stops your apps from becoming unreasonable wide on bigger screens) and `fillPage()` (fills the full height of the browser and is useful if you want to make a plot that occupies the whole screen).

### 4.4.2 Page with sidebar

For more complex layouts, layout functions should be called inside of `fluidPage()`. For example, to make a two-column layout with inputs on the left and outputs on the right you can use `sidebarLayout()` (along with `titlePanel()`, `sidebarPanel()`, and `mainPanel()`).

```
fluidPage(
  titlePanel(
    # app title/description
  ),
  sidebarLayout(
    sidebarPanel(
      # inputs
    ),
    mainPanel(
      # outputs
    )
  )
)
```

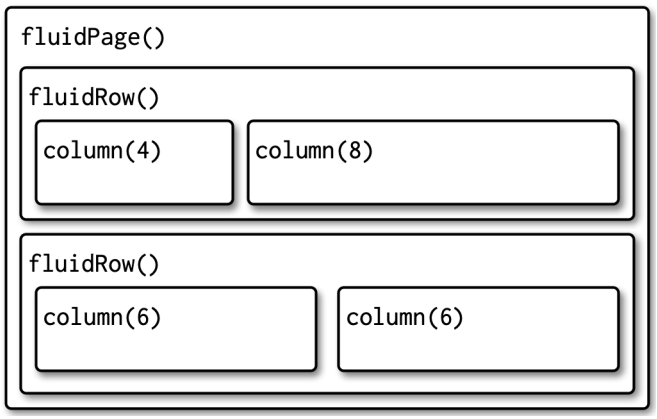



### 4.4.3 Multi-row

As usual, you start with `fluidPage()`. Then you create rows with `fluidRow()`, and columns with `column()`.

```
fluidPage(
  fluidRow(
    column(4,
      ...
    ),
    column(8,
```

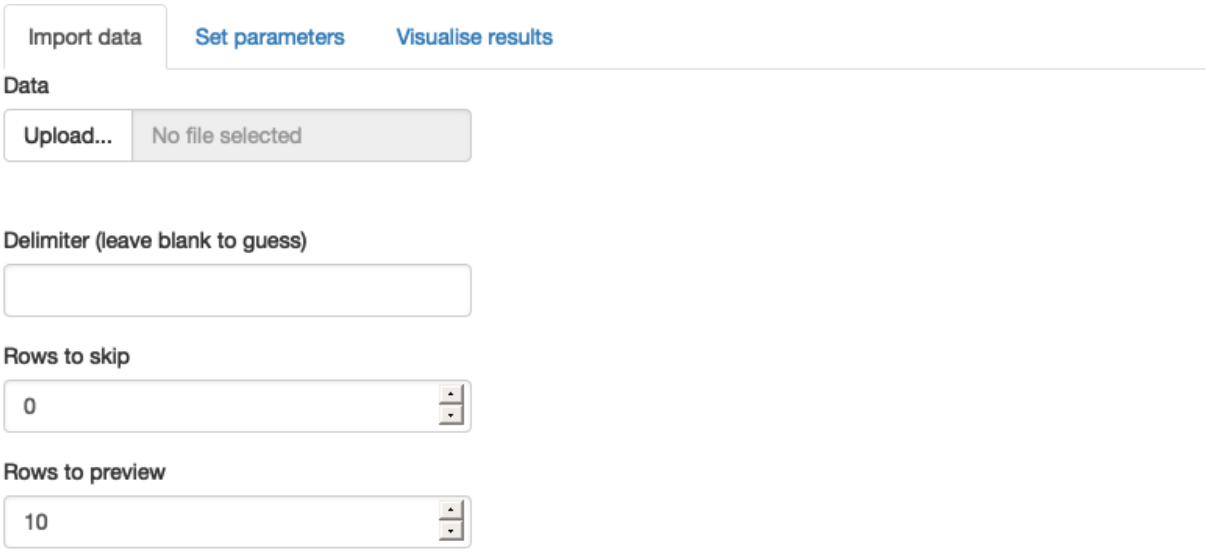
```
      ...
    )
  ),
  fluidRow(
    column(6,
      ...
    ),
    column(6,
      ...
    )
  )
)
```




### 4.4.4 Multi-page layouts

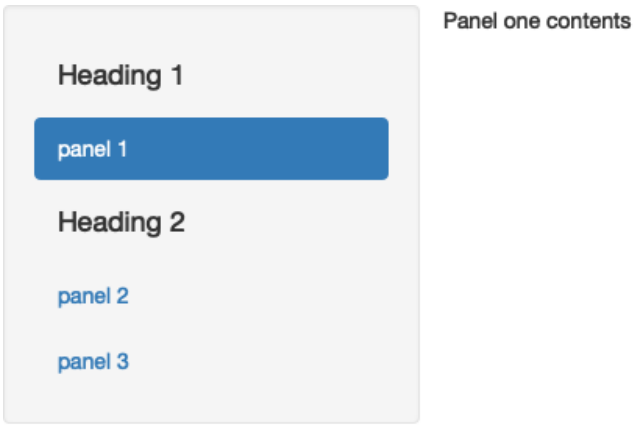
**4.4.4.1 Tabsets** The simple way to break up a page into pieces is to use `tabsetPanel()` and `tabPanel()`.

```
ui <- fluidPage(
  tabsetPanel(
    tabPanel("Import data",
      fileInput("file", "Data", buttonLabel = "Upload..."),
      textInput("delim", "Delimiter (leave blank to guess)", ""),
      numericInput("skip", "Rows to skip", 0, min = 0),
      numericInput("rows", "Rows to preview", 10, min = 1)
    ),
    tabPanel("Set parameters"),
    tabPanel("Visualise results")
  )
)
```




**4.4.4.2 Navlists and navbars** Because tabs are displayed horizontally, there is a fundamental limit to how many tabs you can use, particularly if they have long titles. `navbarPage()` and `navbarMenu()` provide two alternative layouts that let you use more tabs with longer titles. `navlistPanel()` is similar to tabsetPanel() but instead of running the tab titles horizontally, it shows them vertically in a sidebar.

```
ui <- fluidPage(
  navlistPanel(
    id = "tabset",
    "Heading 1",
    tabPanel("panel 1", "Panel one contents"),
    "Heading 2",
    tabPanel("panel 2", "Panel two contents"),
    tabPanel("panel 3", "Panel three contents")
  )
)
```



## 4.5 Dynamic UI

So far, the user interface has been defined statically when the app is launched so it can't respond to anything that happens in the app. However, we can also create dynamic user interfaces, changing the UI using code run in the server function.

There are three key techniques for creating dynamic user interfaces:

- Using the `update` family of functions to modify parameters of input controls.
- Using `tabsetPanel()` to conditionally show and hide parts of the user interface (for examples see https://mastering-shiny.org/action-dynamic.html).
- Using `uiOutput()` and `renderUI()` to generate selected parts of the user interface with code.

Here is a simple example, where once a minimum and maximum has been specified by the user, the ends of the slider change accordingly:

```
ui <- fluidPage(
  numericInput("min", "Minimum", 0),
  numericInput("max", "Maximum", 3),
  sliderInput("n", "n", min = 0, max = 3, value = 1)
)
server <- function(input, output, session) {
  observeEvent(input$min, {
    updateSliderInput(inputId = "n", min = input$min)
  })
  observeEvent(input$max, {
    updateSliderInput(inputId = "n", max = input$max)
  })
}
```

Another simple example resets the slider value to zero every time the reset action button is clicked.

```
ui <- fluidPage(
  sliderInput("x1", "x1", 0, min = -10, max = 10),
  sliderInput("x2", "x2", 0, min = -10, max = 10),
  sliderInput("x3", "x3", 0, min = -10, max = 10),
  actionButton("reset", "Reset")
)

server <- function(input, output, session) {
  observeEvent(input$reset, {
    updateSliderInput(inputId = "x1", value = 0)
    updateSliderInput(inputId = "x2", value = 0)
    updateSliderInput(inputId = "x3", value = 0)
  })
}
```

If we want to create a specific type of inputs depending on other inputs, we can use `uiOutput()` and `renderUI()`. Here is an example that dynamically creates an input control, with the type and label control by two other inputs:

```
ui <- fluidPage(
  textInput("label", "label"),
  selectInput("type", "type", c("slider", "numeric")),
  uiOutput("numeric")
)
server <- function(input, output, session) {
  output$numeric <- renderUI({
    if (input$type == "slider") {
      sliderInput("dynamic", input$label, value = 0, min = 0, max = 10)
    } else {
      numericInput("dynamic", input$label, value = 0, min = 0, max = 10)
    }
  })
}
```

## 4.6 Validation

The most important feedback you can give to the user is whether they have given a bad input. Thinking through how the user might misuse your app allows you to provide informative messages in the UI, rather than allowing errors to trickle through into the R code and generate uninformative errors. A great way to give additional feedback to the user is via the **shinyFeedback** package (Merlino and Howard 2021).

First, you add `useShinyFeedback()` to the `ui`

```
ui <- fluidPage(
  shinyFeedback::useShinyFeedback(),
  numericInput("n", "n", value = 10),
  textOutput("half")
)
```

Then in the `server()` function, you call one of the feedback functions: `feedback()`, `feedbackWarning()`, `feedbackDanger()`, and `feedbackSuccess()`. They all have three key arguments:

- `inputId`, the id of the input where the feedback should be placed.
- `show`, a logical determining whether or not to show the feedback.
- `text`, the text to display.

They also have color and icon arguments that you can use to further customise the appearance. See the documentation for more details.

```
server <- function(input, output, session) {
  half <- reactive({
    even <- input$n %% 2 == 0
    shinyFeedback::feedbackWarning("n", !even, "Please select an even number")
    input$n / 2
  })

  output$half <- renderText(half())
}
```

The error message is displayed, but the output is still updated. To stop inputs from triggering reactive changes, you need a new tool: `req()`, short for "required."

```
server <- function(input, output, session) {
  half <- reactive({
    even <- input$n %% 2 == 0
    shinyFeedback::feedbackWarning("n", !even, "Please select an even number")
    req(even)
    input$n / 2
  })

  output$half <- renderText(half())
}
```

**shinyFeedback** is great when the problem is related to a single input. But sometimes the invalid state is a result of a combination of inputs. In this case it doesn't really make sense to put the error next to an input (which one would you put it beside?) and instead it makes more sense to put it in the output. You can do so with `validate()`. When called inside a reactive or an output, `validate(message)` stops execution of the rest of the code and instead displays message in any downstream outputs. The following code shows a simple example where we don't want to log or square-root negative values.

```
ui <- fluidPage(
  numericInput("x", "x", value = 0),
  selectInput("trans", "transformation",
    choices = c("square", "log", "square-root")
  ),
  textOutput("out")
)

server <- function(input, output, session) {
  output$out <- renderText({
    if (input$x < 0 && input$trans %in% c("log", "square-root")) {
      validate("x can not be negative for this transformation")
    }

    switch(input$trans,
      square = input$x ^ 2,
      "square-root" = sqrt(input$x),
      log = log(input$x)
    )
  })
}
```

For more information on user feedback see https://mastering-shiny.org/action-feedback.html

# References


Chang, Winston, Joe Cheng, JJ Allaire, Carson Sievert, Barret Schloerke, Yihui Xie, Jeff Allen, Jonathan McPherson, Alan Dipert, and Barbara Borges. 2021. *Shiny: Web Application Framework for r*. https://CRAN.R-project.org/package=shiny.

Chang, Winston, Javier Luraschi, and Timothy Mastny. 2020. *Profvis: Interactive Visualizations for Profiling r Code*. https://CRAN.R-project.org/package=profvis.

Dahl, David B., David Scott, Charles Roosen, Arni Magnusson, and Jonathan Swinton. 2019. *Xtable: Export Tables to LaTeX or HTML*. https://CRAN.R-project.org/package=xtable.

Dowle, Matt, and Arun Srinivasan. 2021. *Data.table: Extension of 'Data.frame'*. https://CRAN.R-project.org/package=data.table.

Grolemund, Garrett, and Hadley Wickham. 2011. "Dates and Times Made Easy with lubridate." *Journal of Statistical Software* 40 (3): 1–25. https://www.jstatsoft.org/v40/i03/.

Hester, Jim, and Davis Vaughan. 2021. *bench: High Precision Timing of r Expressions*. https://CRAN.R-project.org/package=bench.

Leifeld, Philip. 2013. "texreg: Conversion of Statistical Model Output in R to LaTeX and HTML Tables." *Journal of Statistical Software* 55 (8): 1–24. http://dx.doi.org/10.18637/jss.v055.i08.

Merlino, Andy, and Patrick Howard. 2021. *shinyFeedback: Display User Feedback in Shiny Apps*. https://CRAN.R-project.org/package=shinyFeedback.

Müller, Kirill, and Hadley Wickham. 2021. *Tibble: Simple Data Frames*. https://CRAN.R-project.org/package=tibble.

Ripley, Brian, and Michael Lapsley. 2021. *RODBC: ODBC Database Access*. https://CRAN.R-project.org/package=RODBC.

Temple Lang, Duncan. 2021. *XML: Tools for Parsing and Generating XML Within r and s-Plus*. https://CRAN.R-project.org/package=XML.

Wickham, Hadley. 2010. "A Layered Grammar of Graphics." https://byrneslab.net/classes/biol607/readings/wickham_layered-grammar.pdf.

———. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org.

———. 2019. *Stringr: Simple, Consistent Wrappers for Common String Operations*. https://CRAN.R-project.org/package=stringr.

———. 2021. *Forcats: Tools for Working with Categorical Variables (Factors)*. https://CRAN.R-project.org/package=forcats.

Wickham, Hadley, and Jennifer Bryan. 2022. *Readxl: Read Excel Files*. https://CRAN.R-project.org/package=readxl.

Wickham, Hadley, Romain François, Lionel Henry, and Kirill Müller. 2022. *Dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr.

Wickham, Hadley, and Maximilian Girlich. 2022. *Tidyr: Tidy Messy Data*. https://CRAN.R-project.org/package=tidyr.

Wickham, Hadley, Jim Hester, and Jennifer Bryan. 2021. *Readr: Read Rectangular Text Data*. https://CRAN.R-project.org/package=readr.

Wickham, Hadley, and Evan Miller. 2021. *Haven: Import and Export 'SPSS', 'Stata' and 'SAS' Files*. https://CRAN.R-project.org/package=haven.