Microsoft

QUANTIDE
knowledge from data

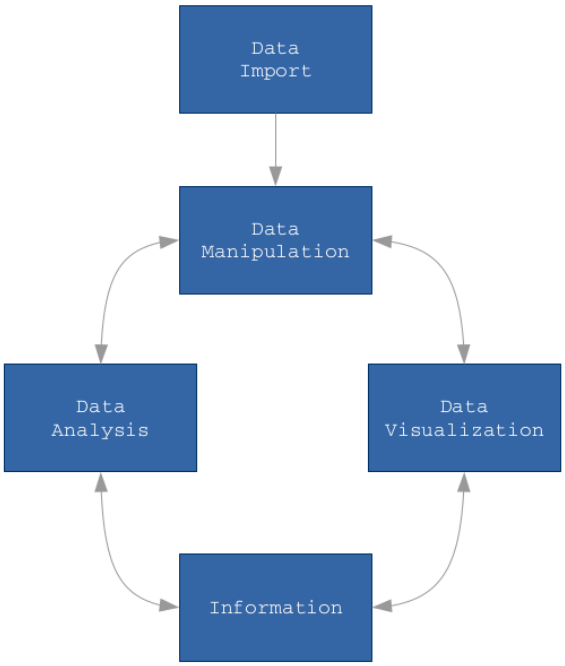
$R for Beginners$
$Manual$

Contents

Chapter 1

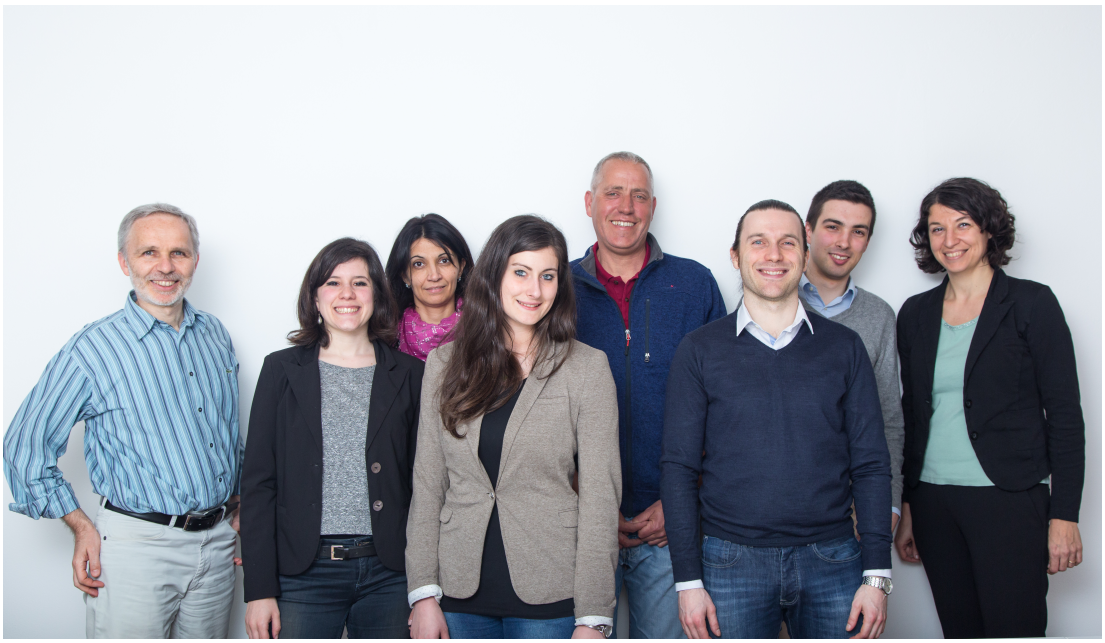
Introduction

This course offers the basics of R, and get an overview on methods for data import, data manipulation, data visualization and data analysis.



This web book is the result of several years of introductory R courses. It contains work of my colleagues:

- Daniela Manzato, who wrote the first version of this book,
- Enrico Pegoraro, who wrote chapters about statistical modeling,
- Veronica Giro and Nicola Sturaro, which reviewed and reassembled all these materials.



However, it would not have been possible without the sharing of knowledge, information, ideas, doubts and even criticisms of many individuals on the internet. I would like to extend my sincere thanks to all of them.

I am grateful to Bill Venables and John Chambers for their publications on R that provided solid foundations to my knowledge on this subject. Moreover Quick R often provides some useful ready-to-cook recipe.

I would finally like to express my special gratitude to Hadley Wickham for providing and sharing his research on the `R` packages: `dplyr` and `ggplot2`. Without his contribution, a part of this manual would never have been written.

I finally express my sincere excuses to all researchers and R enthusiasts I have borrowed any knowledge from without mentioning them. This was not intentional, simply I had not always tracked my sources. If this is the case, please contact me directly and I will be more than happy to include any appropriate reference in this manual.

Andrea Spanò,
Quantide s.r.l.

Chapter 2

About R

2.1 R History

R is a programming environment for data analysis, graphics and statistical computing. The R language is widely used among statisticians for developing statistical software and data analysis.

R was initially developed in early 90s by Robert Gentleman and Ross Ihaka at the Department of Statistics of the University of Auckland as a dialect of the S language.

The R name is partly based on the (first) names of the first two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of S.

2.1.1 What is S and a bit of history

S is a statistical programming language developed by John Chambers and others in Bell Laboratories.

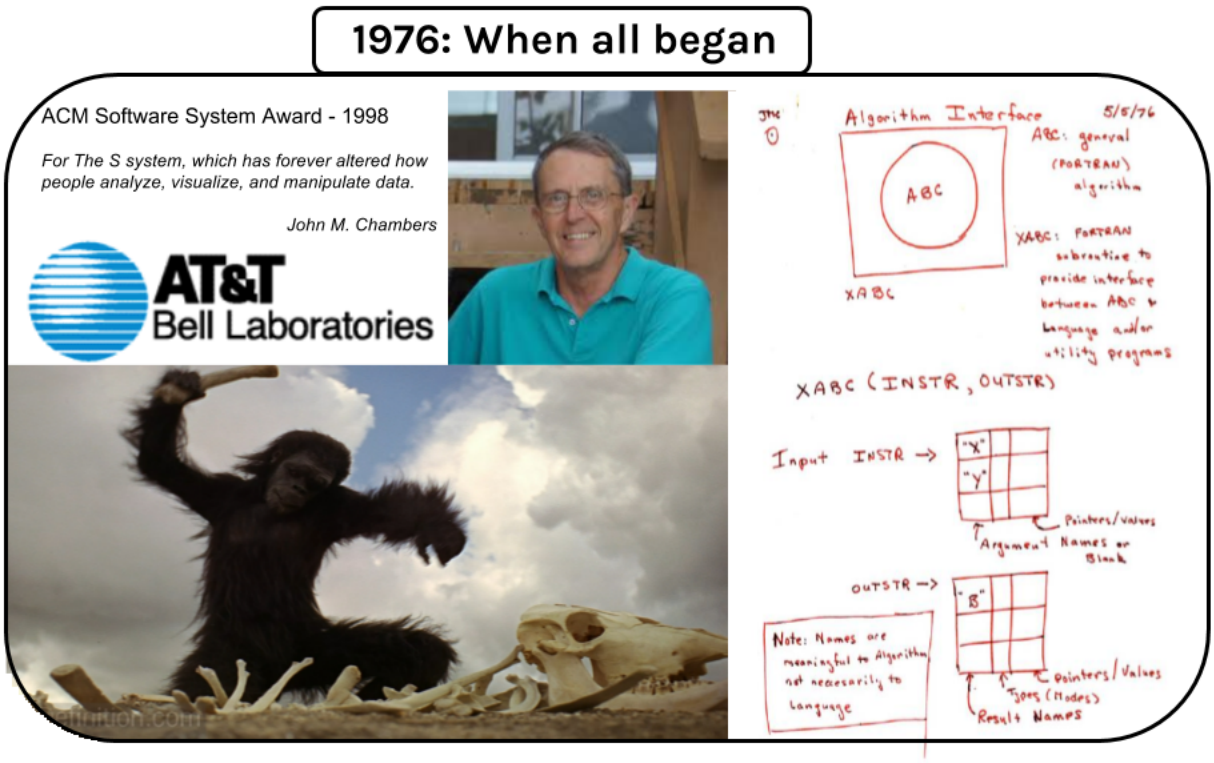


Figure 2.1:

A bit of history:

- 1976: the first version of S is developed as an internal statistical analysis environment. It is originally implemented as Fortran libraries.
- 1980: the first version of S is distributed outside of Bell Laboratories. In 1981, source version is made available.
- 1984: Richard A. Becker and John M. Chambers, "S. An Interactive Environment for Data Analysis and Graphics". (Brown Book). Historical interest only.
- 1988: Richard A. Becker, John M. Chambers and Allan R. Wilks, "The New S Language". London: Chapman & Hall. (Blue Book). It introduces what is now known as S version 2. The system is rewritten in C and begins to resemble the system that we have today.
- 1992: John M. Chambers and Trevor J. Hastie, "Statistical Models in S". (White Book). It introduces S version 3, often abbreviated S3, which adds structures to facilitate statistical modeling in S.
- 1998: John M. Chambers, "Programming with Data". (Green Book). It introduces S version 4, often abbreviated S4, which provides advanced object-oriented features. S4 classes differ markedly from S3 classes.

The S language itself has not changed dramatically since 1998.

2.1.2 What is S-PLUS and a bit of history

S-PLUS is a commercial implementation of the S programming language.

S-PLUS provides a number of fancy features (GUIs, mostly) on top of it, hence the "PLUS".

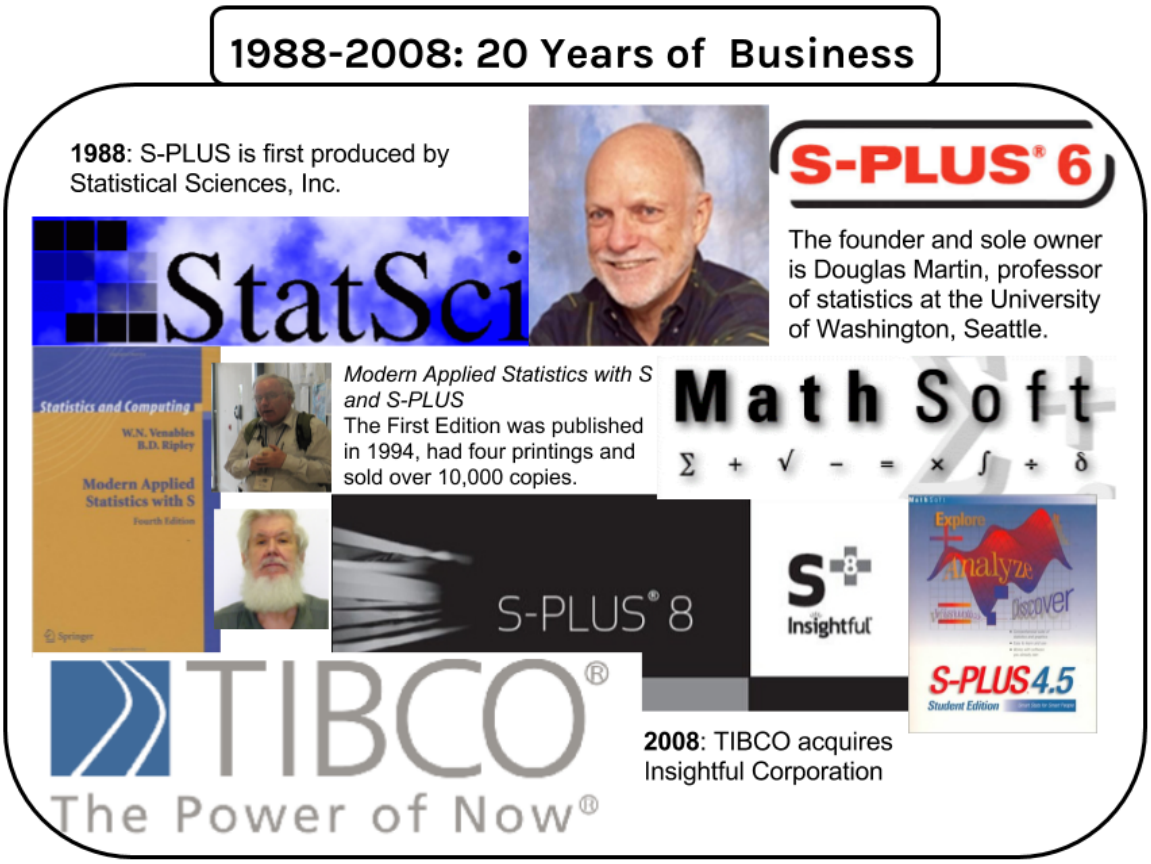


Figure 2.2:

A bit of history:

- 1993: Statistical Sciences, Inc. acquires the exclusive license to distribute S and merges with MathSoft.
- 2001: MathSoft sells its Cambridge-based Engineering and Education Products Division (EEPD). It changes name to Insightful Corporation.
- 2004: Insightful purchases the S language from Lucent Technologies for $2 million.
- 2008: TIBCO acquires Insightful Corporation.

2.1.3 R: a bit of history

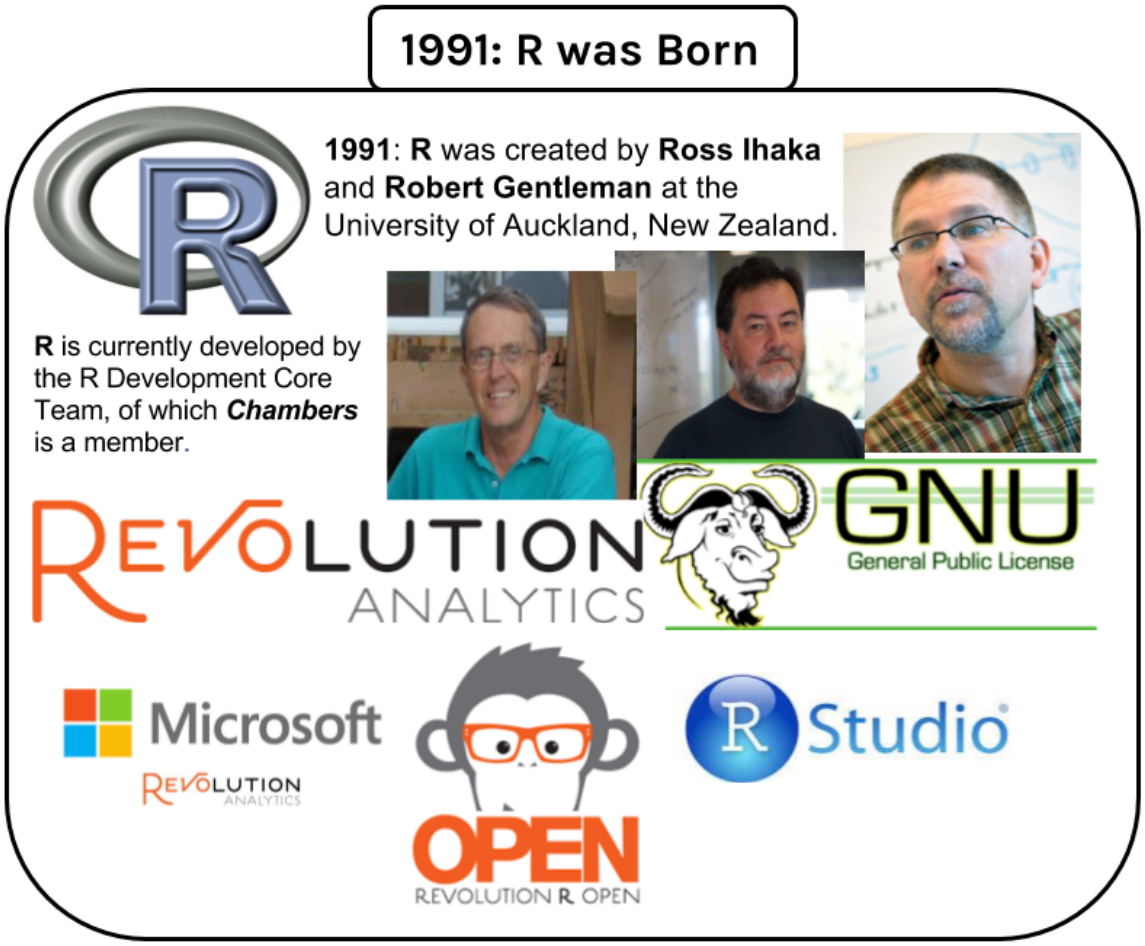


Figure 2.3:

- 1993: First announcement of R to the public.
- 1995: Martin Maechler convinces Ross Ihaka and Robert Gentleman to use the GNU General Public License to make R free software.
- 1997: The R Development Core Team is formed. The team controls the source code for R.
- 2000: R version 1.0.0 released. Developers consider R stable enough for production use.
- 2004: R version 2.0.0 released. Introduced lazy loading, which enables fast loading of data with minimal expense of system memory.
- 2013: R version 3.0.0 released. Introduced long vectors.

While R is an open source project supported by the community developing it, some companies strive to provide commercial support and/or extensions for their customers. R history is intertwined with that of its commercial support:

- 2007: Revolution Analytics was founded to provide commercial support for Revolution R, the distribution of R developed by Revolution Analytics which also includes components developed by the company.

- 2010: RStudio was founded. It is a company that develops free and open tools for the R community.
- 2014: Microsoft Corporation starts the release of Microsoft R Open, an enhanced distribution of R, formerly known as Revolution R Open (RRO).
- 2015: Microsoft Corporation completed the acquisition of Revolution Analytics.

We will deepen the commercial support tools mentioned in *R commercial support* paragraph.

In April 2016, R is in 18th place of TIOBE Programming Community Index, that is an indicator of the popularity of programming languages. R is above SAS that is in 23th place.

2.2 The R-project and R Licence

R is supported by a wide community of academic users, professors, companies and developers. This community composes the so-called "R-project". The "R-project" is supported by the "R Foundation". The R Foundation is a not for profit organisation.

R is an official part of the Free Software Foundation's GNU project. The R Foundation has similar goals to other open source software foundations like the Apache Foundation or the GNOME Foundation. R is free and open source software. It is released under the GPL (version 2) licence.

R is free:

- you can have R without paying for it (freeware);
- you can copy and re-use the software (free software);
- you can access source code and modify it (open source).

2.2.1 What R does?

R provides a suite of software facilities for:

- matrix algebra;
- hash tables and regular expressions;
- reading and manipulating data;
- computation;
- programming language: loops, subroutines, functions, etc.;
- conducting statistical analyses;
- graphics and tables;
- displaying the results.

On the contrary, R:

- it is not a database, but it connects to databases;
- it does not provide a graphical interface, but it uses Java, TclTk and, under Windows, COM to provide graphical interfaces;
- it is not a spreadsheet, but it connects to spreadsheets;
- it does not provide commercial support. Revolution R is a commercially supported distribution of R.

In conclusion, R is an interpreted computer language. R provides a platform for the development and implementation of new algorithms and technology transfer. Most user-visible functions are written in R itself, calling upon a smaller set of internal primitives. It is possible to interface procedures written in C, C+, or FORTRAN languages for efficiency, and to write additional primitives. System commands can be called from within R.

2.2.1.1 R Advantages and Disadvantages

Main R advantages are:

- Fast and free.
- State of the art: Statistical researchers provide their methods as R packages. SPSS and SAS are years behind R!
- Excellent for graphics.
- Mx, WinBugs, and other programs use or will use R.
- Active user community.
- Excellent for simulation, programming, computer intensive analyses, etc.
- Forces you to think about your analysis.
- Interfaces with database storage software (SQL).

Main R disadvantages are:

- Not user friendly at start: steep learning curve, minimal GUI.
- Sometimes, figuring out correct methods or how to use a function on your own can be frustrating.
- Easy to make mistakes and not know.
- Working with large datasets is limited by RAM.
- Data preparation and cleaning can be messier and more mistake prone in R vs SPSS or SAS.

2.3 R Online Resources

R strength is its community, which is distributed and keeps growing all over the World!

Figure 2.4:

Thanks to its community, R can boast a wide variety of online resources, free and otherwise, to learn more about it, ranging from websites, blogs and commercial support tools.

Let us have a look at the most important.

2.3.1 R-project Website

The R-project website (www.r-project.org) is the starting point for R materials.

The website contains:

- the software and packages;
- the search engine interface (the same queries can be submitted with the RSiteSearch ('query') function within R);
- the on-line documentation both in HTML and in PDF format. The HTML version can be accessed with the `help.start()` function within R;
- the R Journal. The R Journal is the open access, refereed journal of the R project. It features short to medium length articles covering topics that might be of interest to users or developers of R;
- the interface to the mailing list;
- the wiki, suggested books and many others.

The on-line documentation includes the following manuals. These manuals have been written by the R Development Core Team itself and contain precious information.

- *An Introduction to R* gives an introduction to the language and how to use R for doing statistical analysis and graphics.
- *Writing R Extensions* covers how to create your own packages, write R help files, and the foreign language (C, C++, Fortran, ...) interfaces.
- *R Data Import/Export* describes the import and export facilities available either in R itself or via packages which are available from CRAN.
- *R Installation and Administration*.

Other manuals and tutorials provided by R users can be downloaded from the R-project website (cran.r-project.org/other-docs.html).

Mailing lists is the most important tool to contact the R community. Mailing lists can be accessed from the R-project website (www.r-project.org/mail.html).

There are five general mailing lists devoted to R:

- *R-announce*: This list is for major announcements about the development of R and the availability of new code.
- *R-packages*: This list is for announcements as well, usually on the availability of new or enhanced contributed packages (on CRAN, typically).
- *R-help*: The "main" R mailing list, for discussion about problems and solutions using R, announcements about the availability of new functionality for R and documentation of R, comparison and compatibility with S-plus, and for the posting of nice examples and benchmarks.
- *R-devel*: This list is intended for questions and discussion about code development in R.
- *R-package-devel*: This list is to get help about package development in R.

2.3.2 Other Online Resources

It is very difficult estimate how many sites about R are on-line. However, Google returns 224.000.000 sites searching "R stat blog". Also if only the 0.1% of these sites talk about R, it means almost 220.000 sites about R.

R-bloggers (www.r-bloggers.com) is a blog aggregator of content collected from bloggers who write about R. R-bloggers contains R news and tutorials contributed by hundreds of R bloggers.

We suggest you to visit MilanoR (www.milanor.net), which is the blog of R users in the Milan Area. Its aim is to exchange knowledge, learn and share tricks and techniques and provide R beginners with an opportunity to meet more experienced users.

Other useful websites about R are:

- Stack Overflow (www.stackoverflow.com) is a website that features questions and answers on a wide range of topics in computer programming, among which r.
- Quick-R (www.statmethods.net) is a useful on-line guide to R. It provides many examples and useful tips.
- R seek (rseek.org) uses Google to search in a selected list of websites about R.

2.3.3 R Commercial Support

2.3.3.1 RStudio Inc.

RStudio, Inc. (www.rstudio.com) is a company that develops free and open tools for the R community, inspired by the innovations of R users in science, education, and industry. These include the RStudio development environment as well as the `shiny`, `ggvis`, and `dplyr` packages (among many others).

RStudio has a mission to provide the most widely used open source and enterprise-ready professional software for the R statistical computing environment. These tools will further the cause of expanding the use of R and the field of data science. It also offers open source and enterprise ready tools for the R computing environment. The flagship product of the RStudio team is an Integrated Development Environment (IDE) which makes it easy for analysts, scientists, data scientists and quants to perform their analyses. It also offer Shiny: a platform that allows you to take those analyses and share them with your team/organization by creating interactive web applications.

2.3.3.2 Microsoft Corporation

Microsoft Corporation provides Microsoft R Open www.mran.microsoft.com/open, an enhanced distribution of R, formerly known as Revolution R Open (RRO). It is a complete open source platform for statistical analysis and data science. The current version, Microsoft R Open 3.3.2, is based on (and 100% compatible with) R-3.3.2 and is therefore fully compatibility with all packages, scripts and applications that work with that version of R. It includes additional capabilities for improved performance, reproducibility, as well as support for Windows and Linux-based platforms. Like R, Microsoft R Open is open source and free to download, use, and share. It is available for Windows, Linux and Mac Os operative systems and you can download it here.

Chapter 3

R and RStudio Set Up

3.1 Installing and Updating R

3.1.1 Design of the R System

The R system is divided into two conceptual parts:

1. The "base" R system that you download from CRAN.
2. Everything else.

R functionality is divided into a number of packages. The "base" R system contains, among other things, the *base* package which is required to run R and contains the most fundamental functions. The "base" system contains also some other packages. Furthermore, every R installation contains "recommended" packages, that are not necessarily maintained by R Core.

"Everything else" point out CRAN "contributed" packages and packages that are not on CRAN. This does not mean that these packages are necessarily of lesser quality than the above, e.g., many contributed packages on CRAN are written and maintained by R Core members. The goal is simply to try to keep the base distribution as lean as possible. Beyond CRAN, many packages are avalaible in BioConductor project or in Github repositories.

3.1.2 Installing R

R is available for Windows, Mac Os and Linux.

The base R can be downloaded from the Comprehensive R Archive Network (CRAN) website. The CRAN is a collection of sites which carry identical material, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries.

Go to www.r-project.org. Here you can read about R and see what's new in the latest version.

Click on CRAN under Download in the left list (see Figure Screenshot of the R-project website). That'll take you to a list of servers (mirrors) in different countries where you can download R.

Now you can choose what operative system you have. Choose within the upper box with "Download and Install R". The box contains "Precompiled binary distributions"; sounds complicated, but means that the program is ready to use, with installation program and all.

Windows users click on "base" and then click "Download R X.X.X for Windows".

Mac users click on "R-X.X.X.pkg (latest version)".

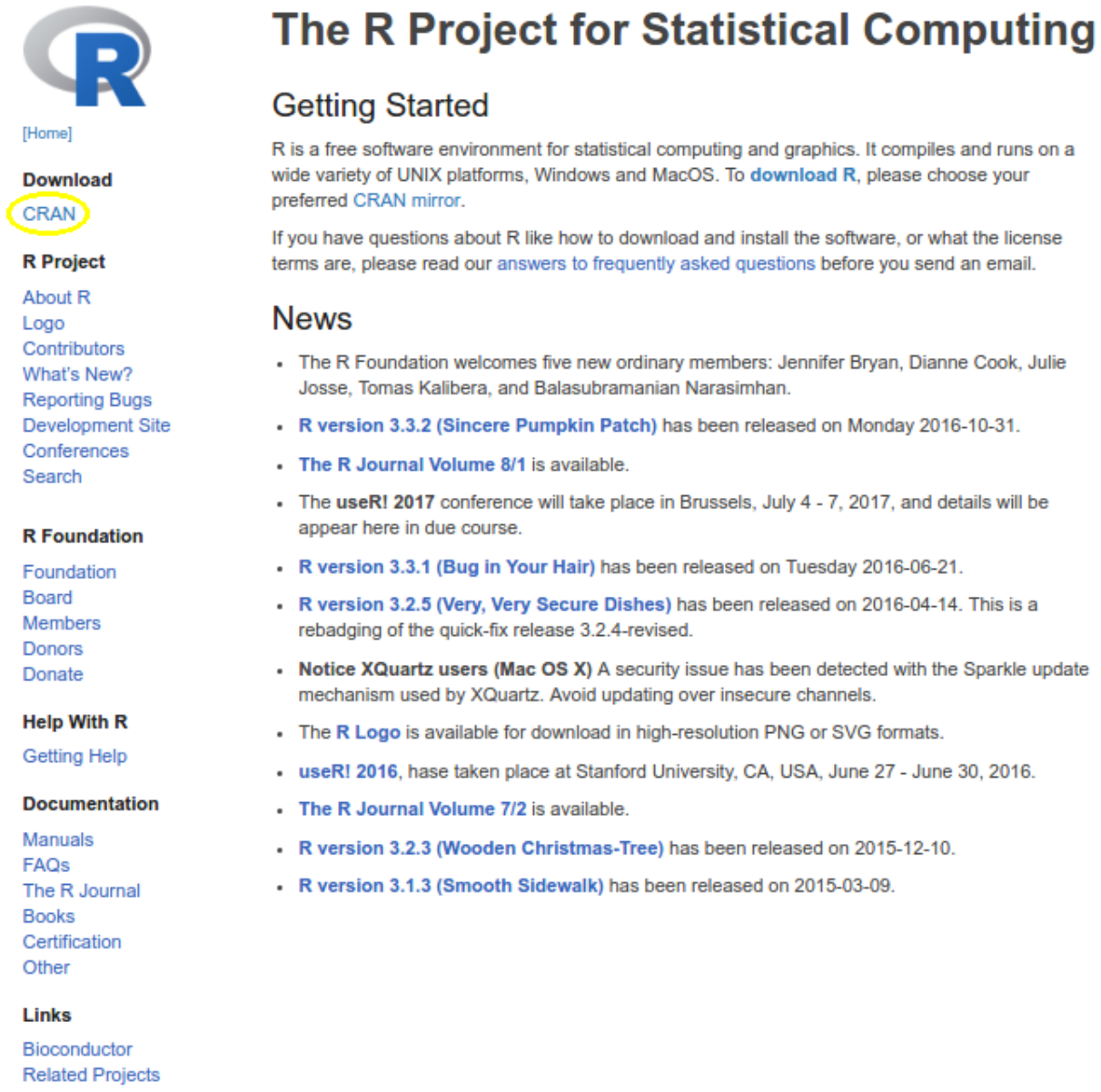


Figure 3.1: Screenshot of the R-project website

Linux users find download files and installation instructions for their Linux distribution in the website.

X.X.X identify the current version of R. In December 2016, the current version is the 3.3.2.

To install R, Windows and Mac users must just double click on the file and follow the installation instructions.

Linux users can download and install R using the precompiled R binary for their distribution. Alternatively, experienced Linux users can compile R from sources. Currently, precompiled R binary are available for Debian, Ubuntu, Suse and RedHat. The R installation package for RedHat downloadable from CRAN is obsolete. An R updated version for Red Hat Enterprise Linux 5 is available for free by Revolution Analytics.

The Figure Screenshot of a first try with R shows a first try with R, just after installation.

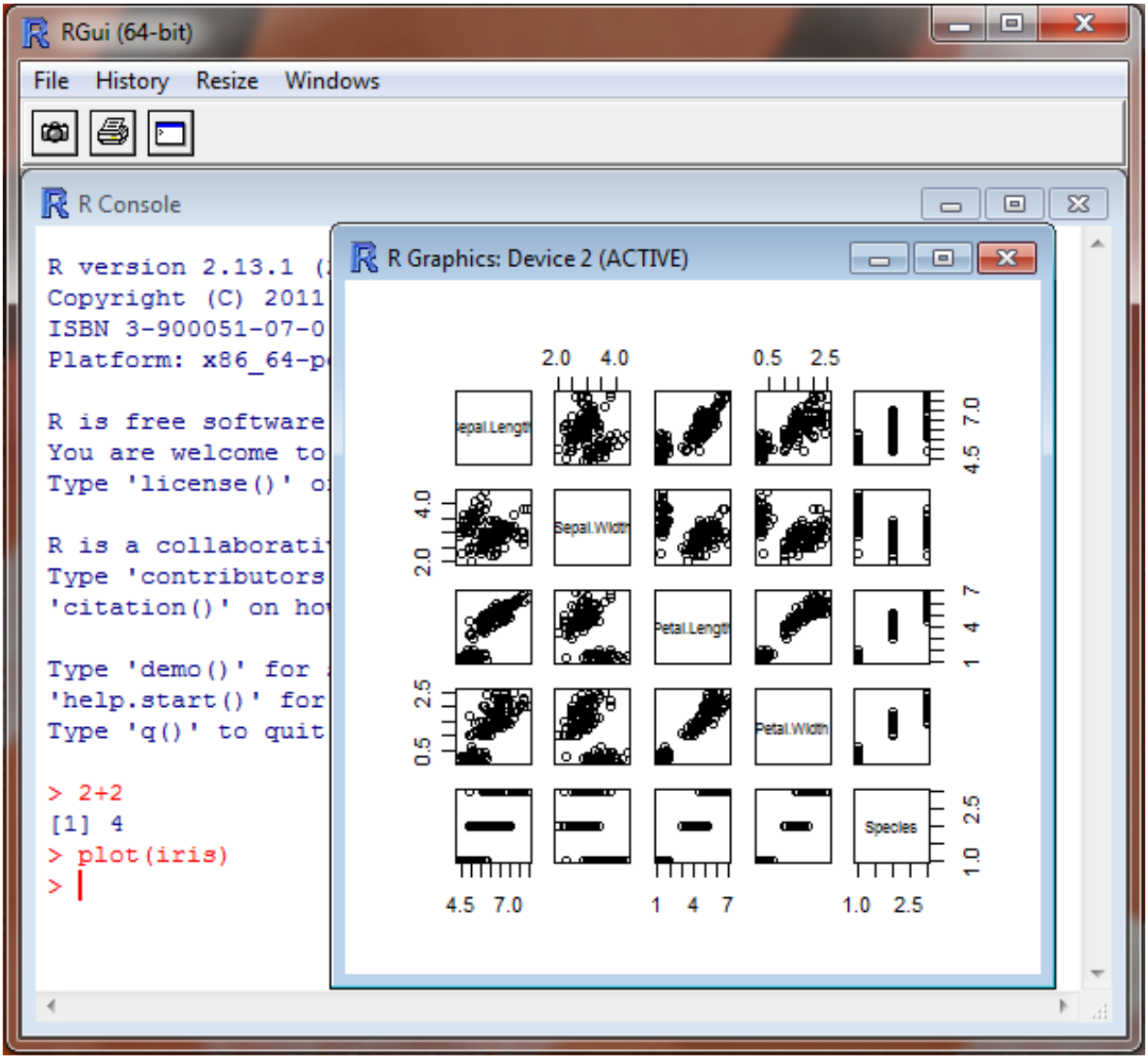


Figure 3.2: Screenshot of a first try with R

3.1.3 Updating R

In Windows and Mac, there is not an automatic way to update R. Two or more versions of R can coexist in the same machine, so a newer release of R can be installed beside the old release. Unfortunately, packages must be re-installed in the "new" R version. A copy-and-paste of package files from the directory containing the packages of the "old" R version to the directory containing the packages of the "new" R version may be useful when many packages are installed. Within R, the `.Library` variable shows the directory containing the packages. After the copy-and-paste, packages of the "new" R version should be updated.

In Linux, usually one R version at time can be installed. R can be updated following the instructions in the CRAN website.

3.2 Graphical User Interfaces

3.2.1 R Default Interfaces

R is provided with a Command Line Interface (CLI), which is the preferred user interface for power users because it allows direct control on calculations and it is flexible. However, good knowledge of the language is required. CLI is thus intimidating for beginners. The learning curve is typically longer than with a Graphical User Interface (GUI), although it is recognised that the effort is profitable and leads to better practice (finer understanding of the analysis, command easily saved and replayed).

R is available for many different operative systems so it does not provide the same graphical interface. When you use the R program interactively, it issues a prompt when it expects input commands. The default prompt is '>'. In Windows, RGui is the default GUI. Inside RGui there is the RConsole window. In Macintosh, RConsole is the default GUI. In Linux, R does not provide any graphical interface by default and it can be used through CLI.

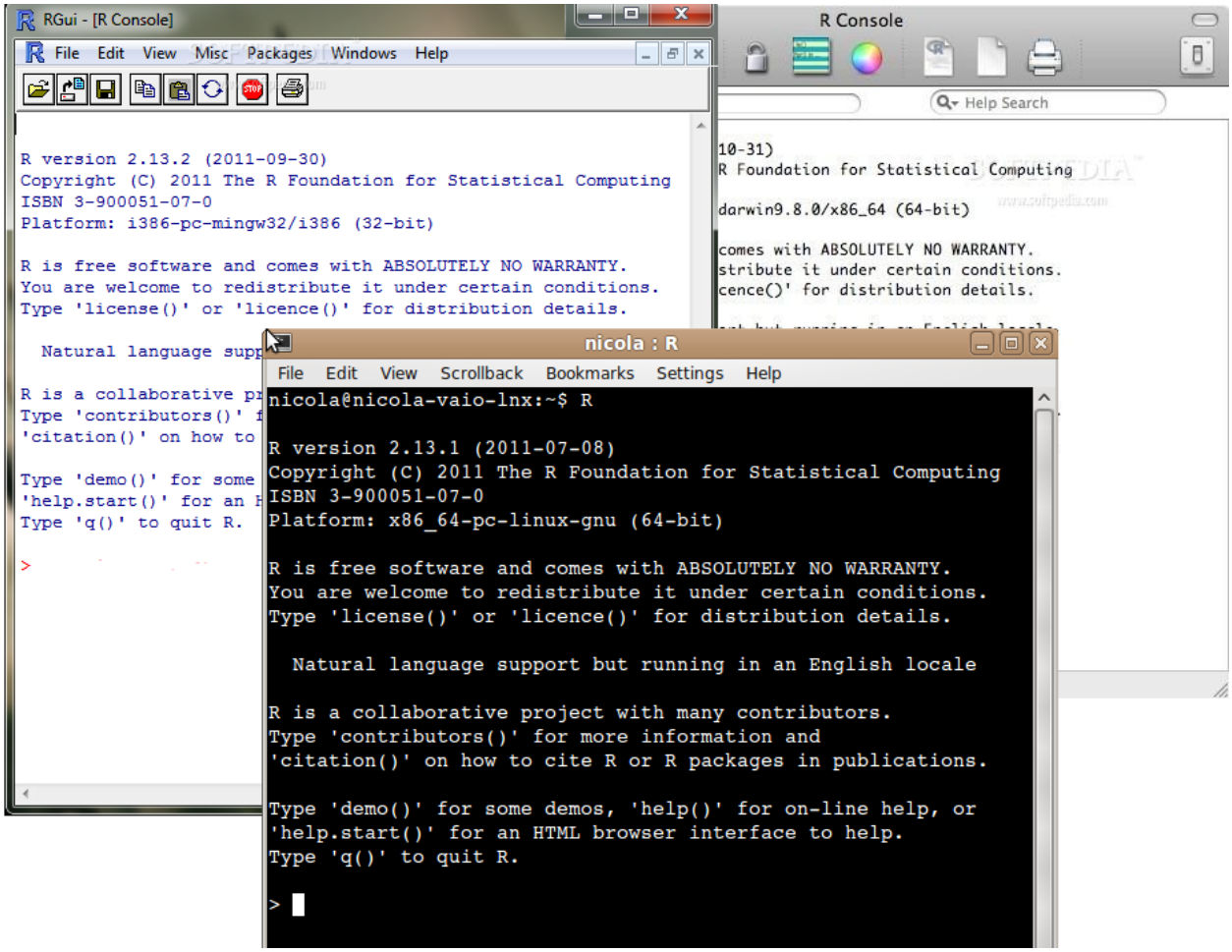

RGui - [R Console]
File Edit View Misc Packages Windows Help
R version 2.13.2 (2011-09-30)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-pc-mingw32/i386 (32-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R Console
Help Search
R Foundation for Statistical Computing
comes with ABSOLUTELY NO WARRANTY.
cence()' for distribution details.
nicola : R
File Edit View Scrollback Bookmarks Settings Help
nicola@nicola-vaio-lnx:~$ R
R version 2.13.1 (2011-07-08)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-linux-gnu (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Figure 3.3:

3.2.2 RStudio and Others R Alternate Interfaces

Several projects develop or offer the opportunity to develop alternate user interfaces. They are presented at www.sciviews.org/_rgui.

RStudio (www.rstudio.org) is a free and open source multi-platform integrated development environment (IDE) for R. It provides syntax highlighting, code completion and smart indentation. Moreover, it executes R code directly from the source editor and it manages easily multiple working directories using projects. It provides:

- workspace browser and data viewer;
- plot history, zooming, and flexible image and PDF export;
- integrated R help and documentation;
- Sweave authoring including one-click PDF preview;
- searchable command history.

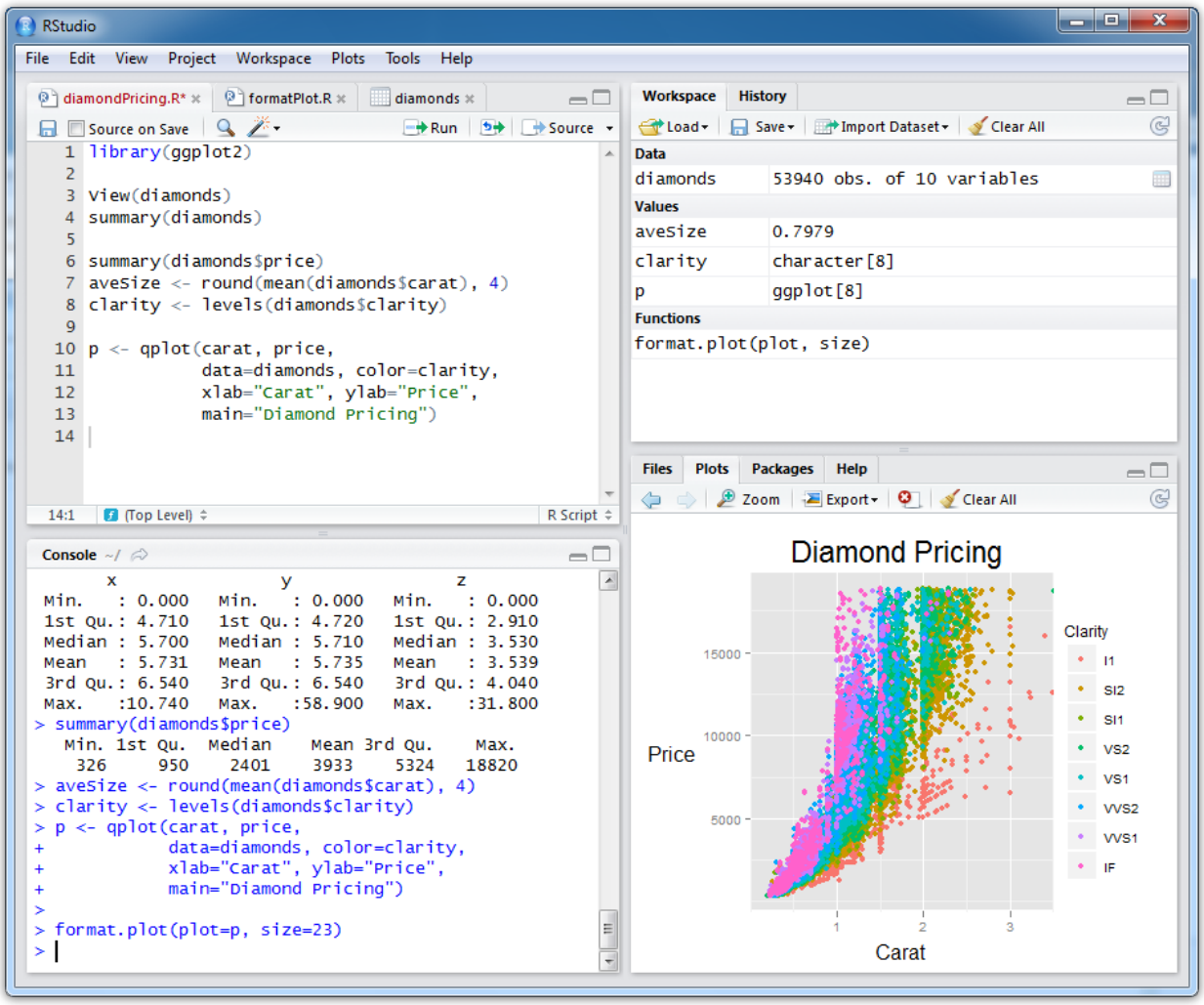


Figure 3.4:

RStudio is available for Windows, Mac Os and Linux and you can install it from its website (www.rstudio.org), clicking on *Download RStudio*. Next, click *Download RStudio Desktop*. At this stage, click the link to the version of RStudio appropriate for your system in *Installers for Supported Platforms* section. Clicking this link downloads RStudio to your computer. Run the installation file and RStudio will be installed on your system.

Similarly to RStudio, **JGR** (rforge.net/JGR) and **Rkward** (rkward.sourceforge.net) are multi-platform IDE. JGR is an R console replacement, it provides also a very decent R code editor with syntax highlighting, calltips, completion and submission of code to R. Rkward has interesting features too to edit R script files and submit code to R.

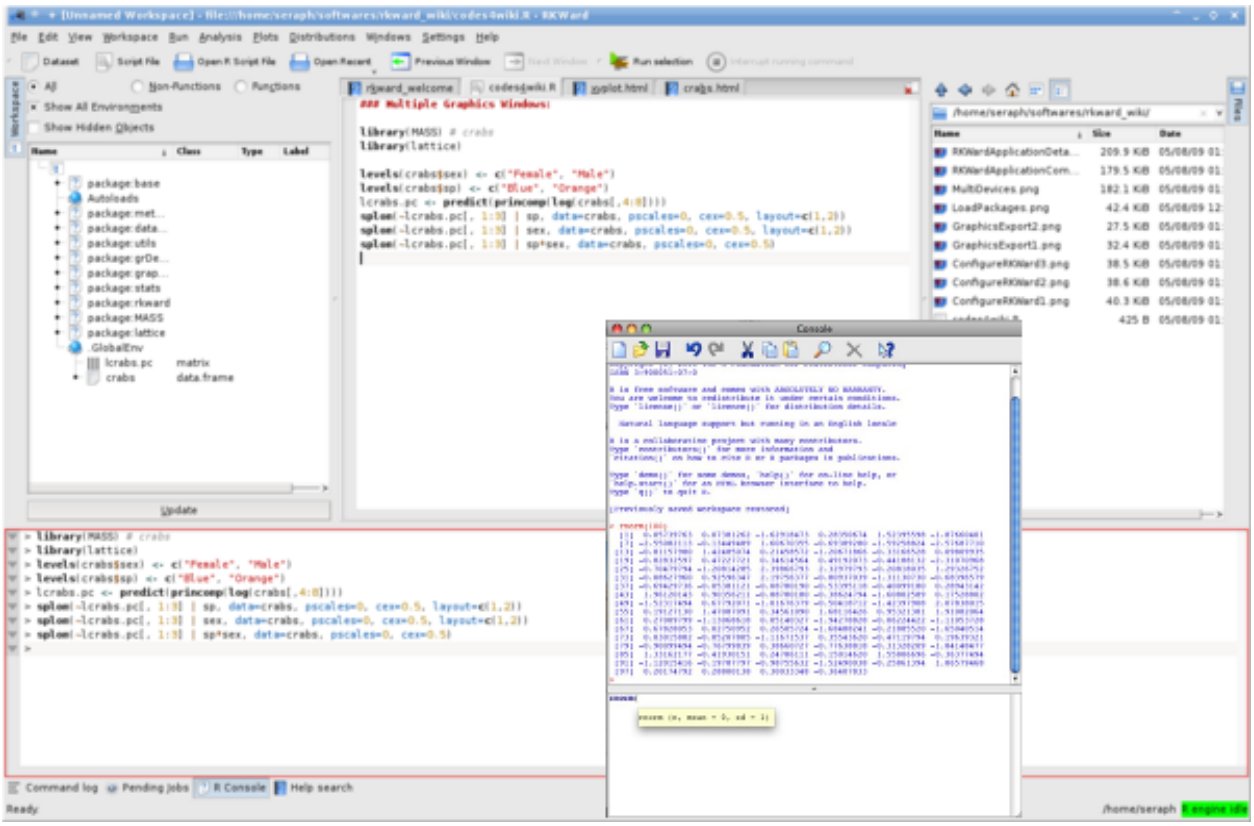


Figure 3.5:

Tinn-R (www.sciviews.org/Tinn-R) is one of the most used R editor by Windows users. It includes R syntax highlighting, help on R functions syntax while you type them, direct submission of R code and many other tools to control R from within the editor.

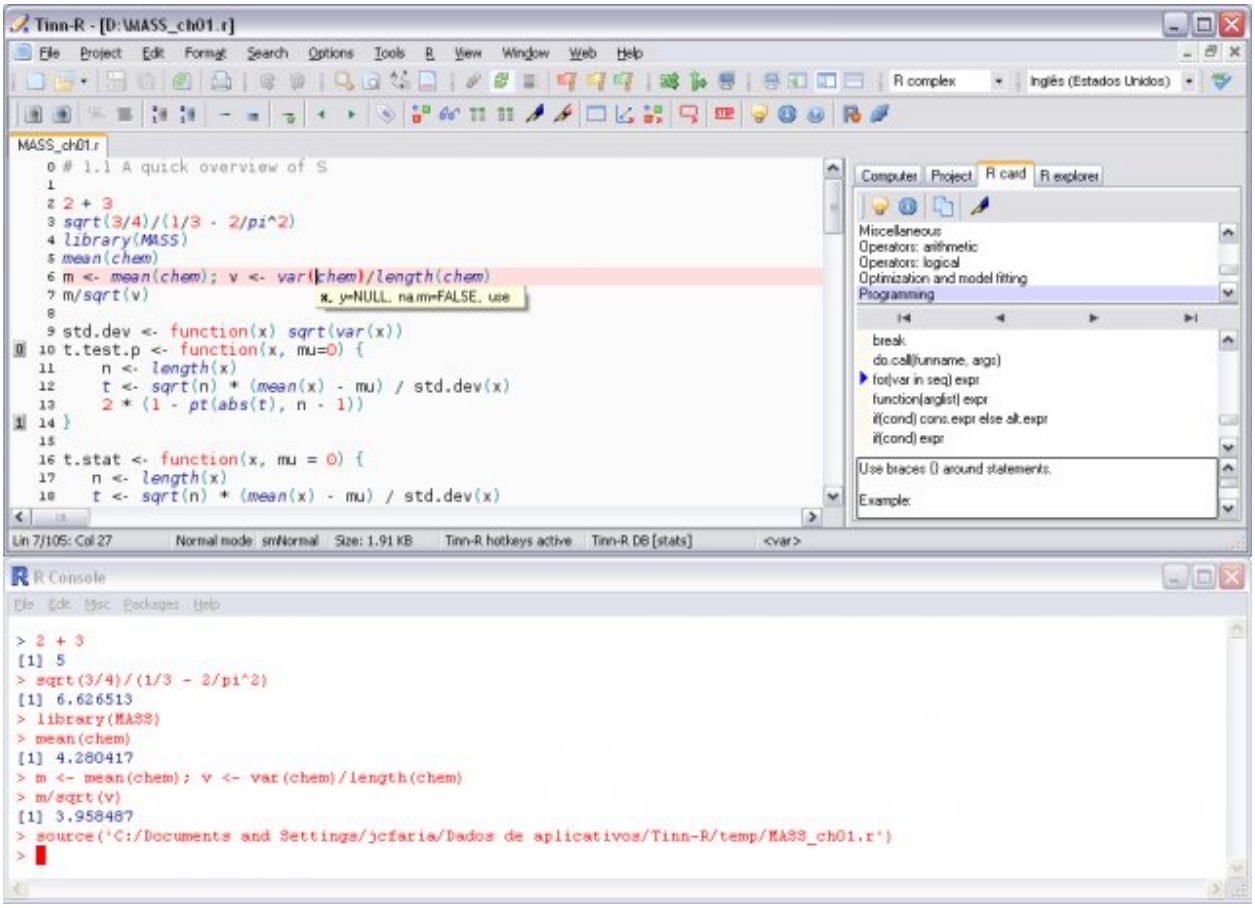


Figure 3.6:

Emacs, **VIM**, **Eclipse** and **Komodo** are powerful multiplatform editors well known by software developers. They support also R, directly or through plugins.

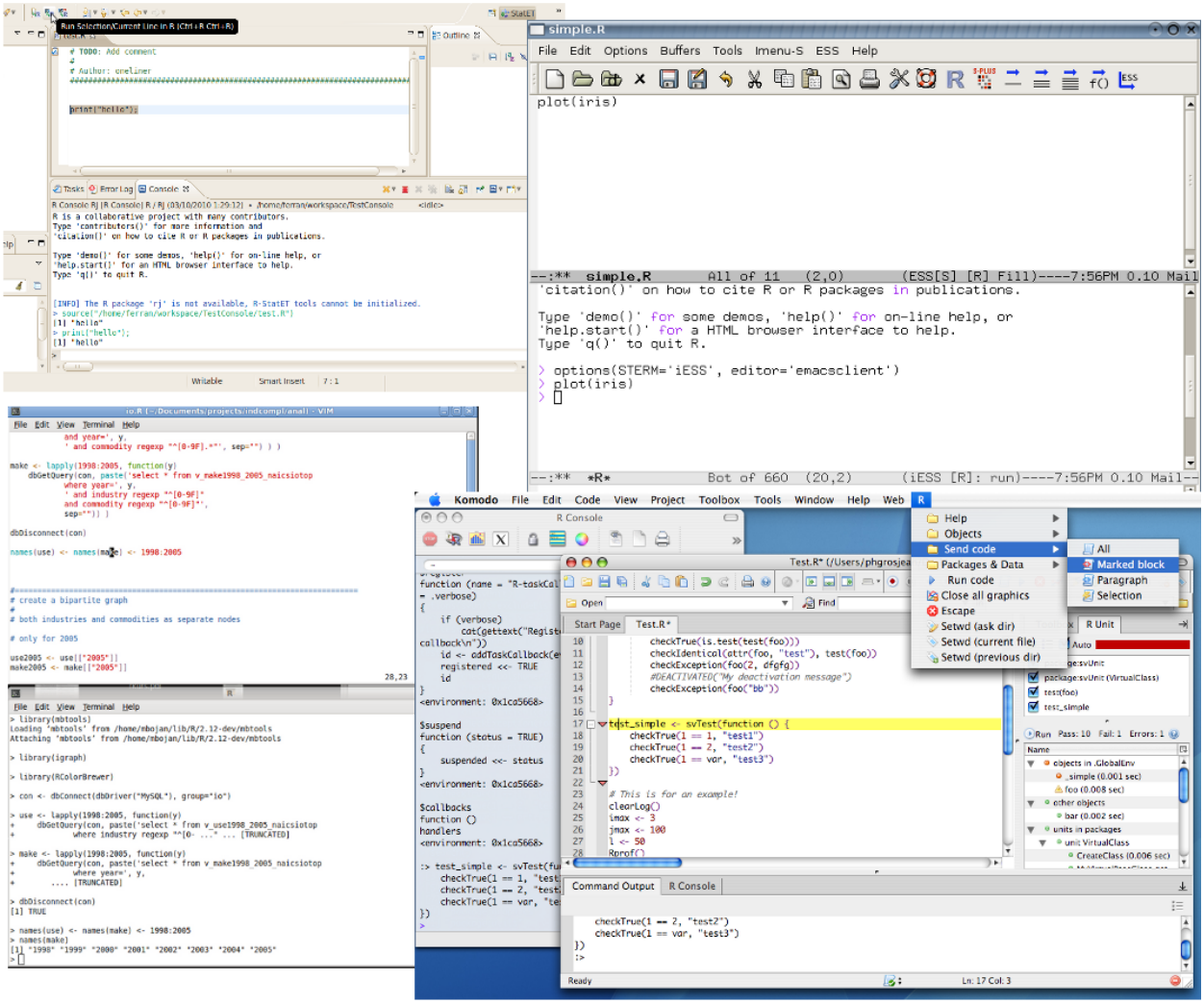


Figure 3.7:

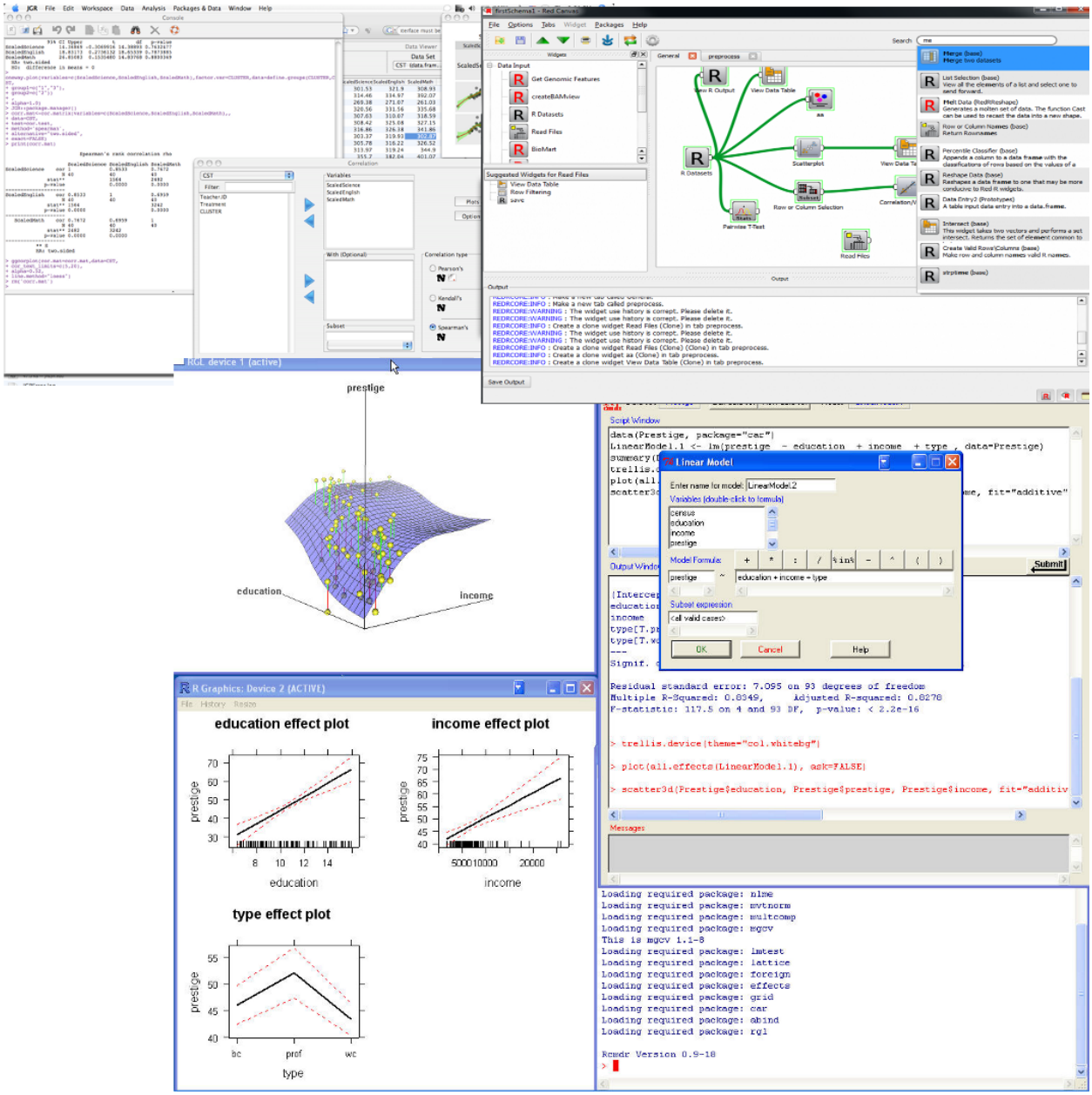

prestige
education
income
education effect plot
income effect plot
type effect plot
prestige
education
income
type
bc
prof
wc
Linear Model
Enter name for model:
LinearModel.2
Variables (double-click to formula)
census
education
income
prestige
Model Formula:
prestige ~ education + income + type
Subset expression
OK
Cancel
Help
Submit
Residual standard error: 7.095 on 93 degrees of freedom
Multiple R-Squared: 0.8349, Adjusted R-squared: 0.8276
F-statistic: 117.5 on 4 and 93 DF, p-value: < 2.2e-16
Loading required package: nlme
Loading required package: mvtnorm
Loading required package: multcomp
Loading required package: mgcv
This is mgcv 1.1-8
Loading required package: lmtest
Loading required package: lattice
Loading required package: foreign
Loading required package: effects
Loading required package: grid
Loading required package: car
Loading required package: abind
Loading required package: rgl
Rcmdr Version 0.9-18

Figure 3.8:

GUIs and IDEs are designed to facilitate programming. Several projects try to provide a graphical interface to R, with menu and buttons:

- **Deducer** (www.deducer.org) is designed to be a free easy to use alternative to proprietary data analysis software. It has a menu system to do common data manipulation and analysis tasks, and an excel-like spreadsheet in which to view and edit data frames. It provides an intuitive graphical user interface (GUI) for R, encouraging non-technical users to learn and perform analyses without programming getting in their way. Deducer is designed to be used with the Java based R console JGR.
- **R-Commander** (socserv.mcmaster.ca/jfox/Misc/Rcmdr), or Rcmdr, provides also access to the most frequently used statistical tools through the mouse.
- **Red-R** (www.red-r.org) is a dataflow programming interface for R designed to bring the power of the R statistical environment to the general researcher or user.

3.3 R Packages

3.3.1 Installing R Packages

R functions are collected in packages. Packages that are not contained in the "base" R systems can be downloaded from the CRAN website. A list of R packages accompanied by a brief description can be found on the CRAN website where there are more than 3700 packages available. Many of these packages are very useful; however, there are some packages in prerelease, incomplete packages, "abandoned" packages (i.e. not more updated) and/or packages containing functions with errors or compatibility troubles.

This is the R package universe!

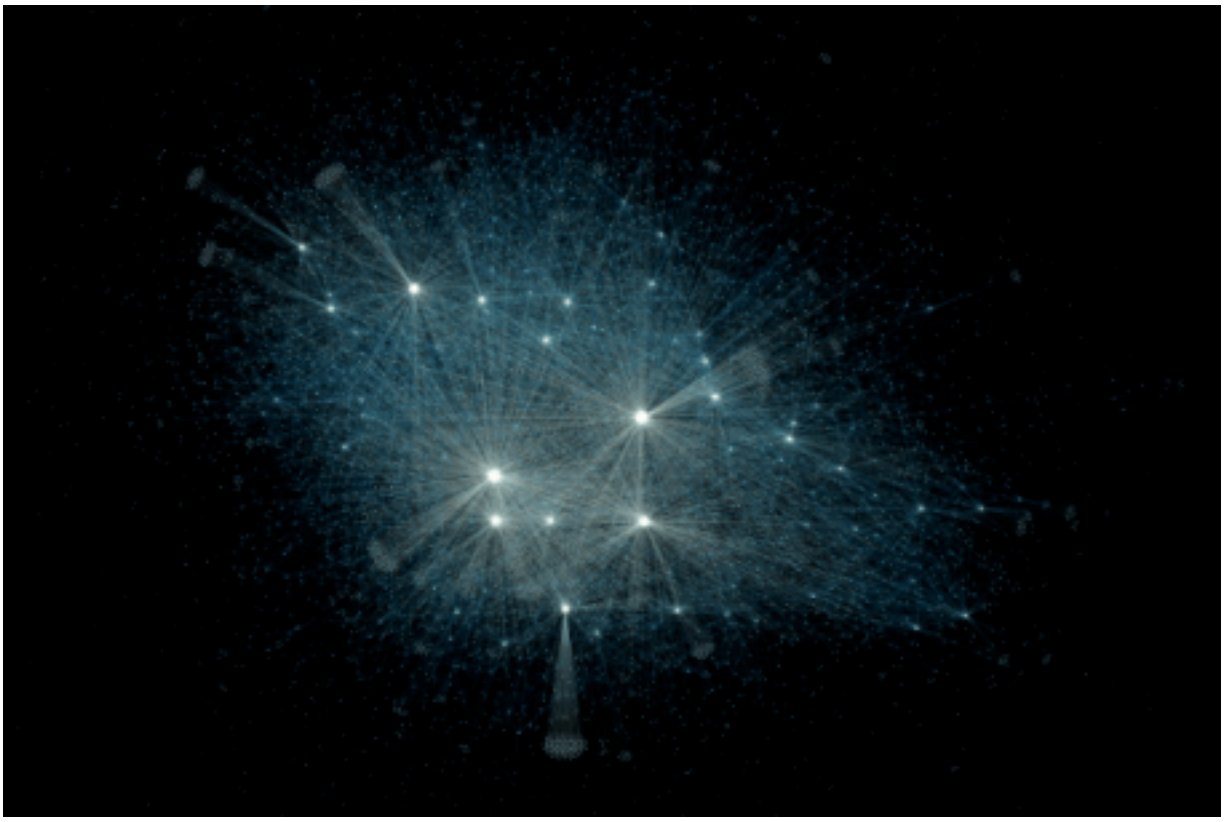


Figure 3.9:

The simple way to install an R package is through the `install.packages()` function, directly from R. In Linux, R must be executed as administrator to install a package. Installation must be executed before the first use of a library.

When a function of a package that is not contained in the "base" R systems is required than the package must be loaded. The `require(pkg)` function load the *pkg* package.

3.3.2 Updating R Packages

R packages can be updated typing `update.packages()` within R. In Linux, R must be executed as administrator to update a package. Packages should be updated regularly.

Chapter 4

A First Look to R Session

4.1 The First R Session

Start the R system, the cursor is waiting for you to type in some R commands. For example, use R as a simple calculator:

```
6 + 3
```

```
## [1] 9
```

```
5 - 9
```

```
## [1] -4
```

```
4 * 6
```

```
## [1] 24
```

```
8 / 3
```

```
## [1] 2.666667
```

```
5 ^ 2
```

```
## [1] 25
```

```
(1 + 0.05)^8
```

```
## [1] 1.477455
```

```
exp(3)
```

```
## [1] 20.08554
```

```
log(14)
```

```
## [1] 2.639057
```

```
23.76 * log(8)/(23 + atan(9))
```

```
## [1] 2.01992
```

4.2 Assignment

Results of calculations can be stored in objects using the assignment operators:

- an arrow (`<-`) formed by a smaller than character and a hyphen without a space;
- the equal character (`=`).

These objects can then be used in other calculations.

There are some restrictions when giving an object a name:

- Object names cannot contain "strange" symbols like `!`, `+`, `-`, `#`.
- A dot (`.`) and an underscore (`_`) are allowed, also a name starting with a dot.
- Object names can contain a number but cannot start with a number.
- R is case sensitive: `X` and `x` are two different objects, as well as `temp` and `temP`.

```
x <- log(14)
y <- 23.76 * log(8)/(23 + atan(9))
z <- x + y
```

To print the object just enter the name of the object or use `print()` function.

```
x
```

```
## [1] 2.639057
```

```
y
```

```
## [1] 2.01992
```

```
print(z)
```

```
## [1] 4.658978
```

4.3 The R Workspace

The workspace is your current R working environment and includes any user-defined objects. It is also known as global environment.

4.3.1 Objects listing

Objects created during an R session are hold in memory. To list the objects in the current R session, the function `ls()` or the function `objects()` may be used.

```
ls()
```

```
## [1] "x" "y" "z"
```

```
objects()
```

```
## [1] "x" "y" "z"
```

So to run the function `ls` the name followed by an opening and a closing bracket must be entered. Entering only `ls` will just print the object, i.e. the underlying R code of the function.

Most functions in R accept certain arguments. For example, one of the arguments of the function `ls()` is pattern. To list all objects starting with the letter `x`:

```
x2 <- log(9)
ls(pattern = "x")
```

```
## [1] "x"  "x2"
```

4.3.2 Removing objects

If a value to an object that already exists is assigned then the contents of the object will be overwritten with the new value (without a warning!). The function `rm()` ought be used to remove one or more objects from your session.

```
rm(x2)
ls()
```

```
## [1] "x" "y" "z"
```

4.3.3 Garbage collection

When objects are no longer used, and this clearly happens when objects are deleted, `R` releases immediately the memory they filled in the system. This is done automatically by the garbage collector `gc()`.

We can call `gc()` also to see how much memory `R` is using for allocating objects

```
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 412003 22.1     750400 40.1   592000 31.7
## Vcells 590170  4.6    1308461 10.0   812817  6.3
```

4.4 The Working Directory

If you want to read files from a specific location or write files to a specific location without typing (on Windows, usually very long) path names you will need to set working directory in R.

You can set a working directory with `setwd()` function, using absolute or relative paths. The following example shows how to set the working directory in R using an absolute path. The working directory is set to the folder "Data" within the folder "Documents and Settings" on the C drive.

```
setwd("C:/User/Andrea/Documents and Settings/Data")
```

Remember that you must use the forward slash / or double backslash \\ in R! The Windows format of single backslash will not work.

It is a nice approach, but things get complicated if you move files to different computers, say from home to your office, and have different directory structures, disk names etc. Of course you can change it every time. Using relative paths solves this problem. Relative paths are more convenient than absolute paths as they makes your R script or RStudio project independent of the directory structure in which it resides thus facilitating the reproducibility of your work on a different PC.

So, you can use `getwd()` function to get the path of the current working directory:

```
getwd()
```

```
## [1] "C:/User/Andrea/Documents and Settings/Data"
```

Suppose that your current directory is the previous one ("Data" folder) and you want to set the directory on "Statistics" folder inside "Data" folder:

```
setwd("./Statistics")
```

```
## [1] "C:/User/Andrea/Documents and Settings/Data/Statistics"
```

`.` represents your current directory.

Suppose that your current directory is the previous one ("Statistics" folder) and you want to set the directory on "Maths" folder inside "Data" folder:

```
setwd("./../Maths")
```

```
## [1] "C:/User/Andrea/Documents and Settings/Data/Maths"
```

`..` are useful to move up the directory hierarchy relative to the current working directory. In particular, you move out from "Statistics" folder, and up into the "Maths" folder.

4.5 R help

Within R, the following functions provide help about R itself:

- The HTML version of R's online documentation can be printed on-screen by typing `help.start()`;
- Online documentation for most of the functions and variables in R can be printed on-screen by typing `help(name)` (or `?name`), where *name* is the name of the topic help is sought for;
- Online documentation for finding help pages on a vague topic can be printed on-screen by typing `help.search('topic')`;
- A list of function containing *topic* in the name can be printed on-screen by typing `apropos('topic')`;
- A research in the website can be performed by typing `RSiteSearch('query')`, where *query* is the search query.

To get help about the `mean()` function, the `help()` function can be used.

```
help(mean)
```

The `help()` function can be called using the `?`.

```
?mean
```

To get a list of functions concerning the mean, the `help.search()` function can be used.

```
help.search("mean")
```

To get a list of function containing "mean" in the name, the `apropos()` function can be used.

```
apropos("mean")
```

Chapter 5

Introduction to R Objects

There are different types of objects in R, which can be divided in:

- data objects
- function object

Data objects are specific types of data structures by which is possible to organize data. The most important are:

- vectors
- matrices
- lists
- factors
- data frames

Function object refers to operation done on data.

We will deepen these two categories of R objetcs in the following chapters.

5.1 Data Objects

The structures of data objects are represented in the following figure:

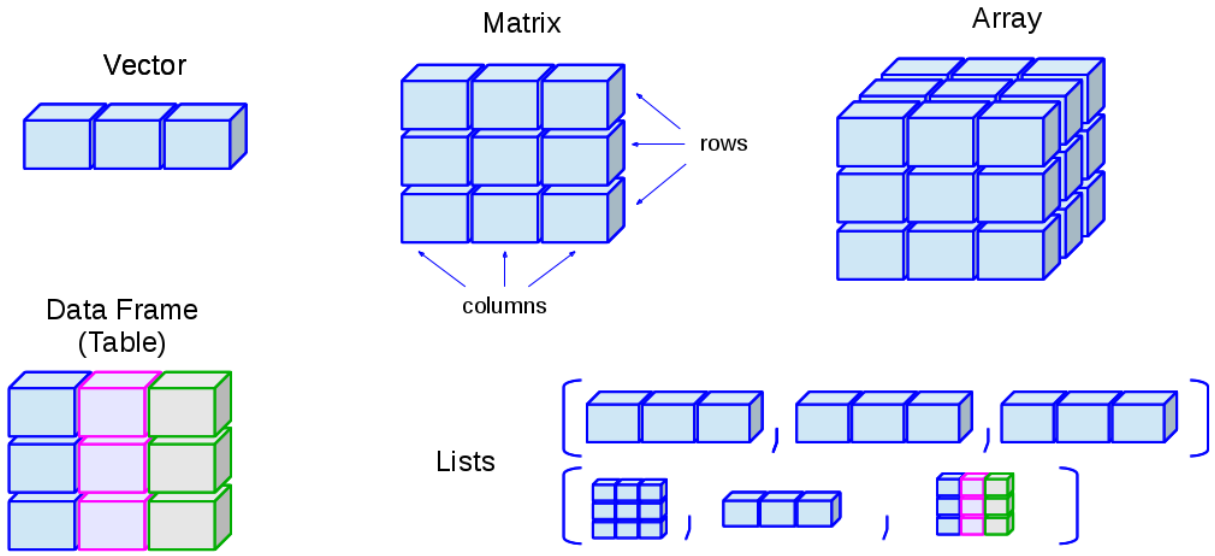


Figure 5.1:

5.1.1 Vectors

In R the simplest data structure is a vector. A vector is defined as an ordered sequence of elements of the same kind.

A vector can be defined according to the data type it contains. Therefore, there are:

- numeric vectors;
- logical (or Boolean) vectors;
- character (or string) vectors.

The most common method to define a vector is the `c()` function.

```
# Numeric vector
num <- c(1, 2, 5.3, 6, -2, 4)
num
```

```
## [1]  1.0  2.0  5.3  6.0 -2.0  4.0
```

```
# Character vector
char <- c("one", "two", "three")
char
```

```
## [1] "one"   "two"   "three"
```

Logical vectors are often defined as the result of control actions on numerical or character vectors.

```
# Logical vector
logic1 <- num > 3
logic1
```

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

Of course, a logical vector can be created using the `c()` function.

```
# Logical vector
logic2 <- c(TRUE, FALSE, TRUE)
logic2
```

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

If a vector mixes different data types, R will store it as a character vector.

```
mixed <- c("foo", 1, TRUE)
mixed
```

```
## [1] "foo"  "1"    "TRUE"
```

The `c()` function can be used to create a vector combining several vectors.

```
vec1 <- c(11, 12, 13)
vec2 <- c(21, 22, 23)
vec3 <- c(31, 32, 33)
comb <- c(vec1, vec2, vec3)
comb
```

```
## [1] 11 12 13 21 22 23 31 32 33
```

A vector can be created using sequences. The easiest method is using the operator `:`. The inputs of the operator `:` are the first number on the left and the last number on the right. The vector will be composed of numbers comprised between the first and the last number (by one unit).

```
go <- 1:20
go
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
back <- 20:1
back
```

```
##  [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

Other sequences can be created using the `seq()` function. The parameters `from` and `to` of the `seq()` function refer to the starting and end values of the sequence, respectively. The parameter `by` indicates the number by which the sequence has to be incremented. Alternatively, the `length.out` parameter refers to the desired length of the sequence.

```
seqby <- seq(from = 1, to = 5, by = 0.5)
seqby
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seqlength <- seq(from = 1, to = 5, length.out = 13)
seqlength
```

```
##  [1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667 3.000000 3.333333 3.666667 4.000000 4.333333
## [12] 4.666667 5.000000
```

Finally, vectors can be created with the `rep()` function which repeats the elements of a vector. The first parameter of the `rep()` function, `x`, is the value or vector to be repeated and the second parameter, `times`, represents the number of repetitions to be made. Alternatively, the `each` parameter enables the repetition of each element of the vector as many times as indicated by the number.

```
rep(x = 1 , times = 10)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1
```

```
rep(x = 1:5, times = 3)
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
rep(x = 1:5 , each = 3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

```
rep(c("ALI", "IPERLANDO"), times=4)
```

```
## [1] "ALI"     "IPERLANDO" "ALI"     "IPERLANDO" "ALI"     "IPERLANDO" "ALI"     "IPERLANDO"
```

```
rep(x = 1:5 , times = 2 , each = 3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

```
rep(x = "no",  times = 5)
```

```
## [1] "no" "no" "no" "no" "no"
```

```
rep(x = c("a", "b", "c"), times = c(3, 2, 1))
```

```
## [1] "a" "a" "a" "b" "b" "c"
```

```
rep(x = c("a", "b", "c"), times = rep(4,3))
```

```
##  [1] "a" "a" "a" "a" "b" "b" "b" "b" "c" "c" "c" "c"
```

```
mydata <- c("a", "b", "c")
myrep <- rep(4,3)
rep(x = mydata, times = myrep)
```

```
##  [1] "a" "a" "a" "a" "b" "b" "b" "b" "c" "c" "c" "c"
```

A subset of the vector `x` can be extracted with `x[subscripts]`. The selection can be done in three ways.

1. A vector of positive integers indicating the elements of the vector to be extracted.

```
x <- c(1, 4, 2, 5, 6, 8, 6, 9, 10)
x[3]
```

```
## [1] 2
```

```
x[1:3]
```

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

```
x[c(2,4)]
```

```
## [1] 4 5
```

2. A vector of negative integers indicating the elements which must not be extracted.

```
x <- c(1, 4, 2, 5, 6, 8, 6, 9, 10)
x[-2]
```

```
## [1]  1  2  5  6  8  6  9 10
```

```
x[-c(2, 5, 7)]
```

```
## [1]  1  2  5  8  9 10
```

3. A Boolean vector indicating the elements to be extracted (`TRUE`) or to be left (`FALSE`).

```
x <- c(1, 4, 2, 5, 6)
x[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
```

```
## [1] 1 4 6
```

```
y <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
x[y]
```

```
## [1] 1 4 6
```

```
x[!y]
```

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

The `!` symbol identifies the "not" logical operator. The logical operator "not" reverses the logical value of a condition on which it operates.

In this way elements satisfying a logical condition can be extracted. Usually, the logical vectors are obtained as the result of logical expressions.

```
x <- c(1, 4, 2, 5, 6, 8, 6, 9, 10)
y <- x > 5
x[y]
```

```
## [1]  6  8  6  9 10
```

The logical expression can be defined inside the square brackets, directly.

```
x[x > 5]
```

```
## [1]  6  8  6  9 10
```

The `&` symbol identifies the "and" logical operator. The "and" logical operator compare two (or more) logical expression and return TRUE if both are TRUE. The following example returns the `x` values greater or equal than 2 but less or equal than 8.

```
x[x >= 2 & x <= 8]
```

```
## [1] 4 2 5 6 8 6
```

The `|` symbol identifies the "or" logical operator. The "or" logical operator compare two (or more) logical expression and return TRUE if at least one is TRUE. The following example returns the `x` values less than 2 or greater than 8.

```
x[x < 2 | x > 8]
```

```
## [1]  1  9 10
```

Extracting unique values contained in a vector can sometimes be useful. This can be done with the `unique()` function.

```
x <- c(1, 2, 1, 1, 2, 3, 3, 2, 1, 2, 2, 3)
unique(x)
```

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

5.1.2 Matrices

Matrices are generalizations of vectors. Like vectors, matrices need to contain elements of the same kind. This Paragraph introduces numeric matrices.

A matrix can be created using the `matrix()` function.

```
matrix(1:8, nrow = 2, ncol = 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

By default, a matrix is filled by columns. The `byrow = TRUE` argument of the `matrix()` function fills the matrix by rows.

```
matrix(1:8, nrow = 2, ncol = 4, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

Alternatively, a matrix can be created by applying the `dim()` function to a vector.

```
x <- 1:8
x
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
dim(x) <- c(2, 4)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

Finally, a matrix can be created by joining two or more vectors, both as column vectors (`cbind()` function) and row vectors (`rbind()` function).

```
cmat <- cbind(1:3, 4:6, 7:9)
cmat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
rmat <- rbind(1:3, 4:6, 7:9)
rmat
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

The `cbind()` and `rbind()` functions can be used to join two (ore more) matrices or vectors and matrices.

```
cbind(cmat, 10:12)
```

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

```
rbind(cmat, rmat, cmat)
```

```
##       [,1] [,2] [,3]
##  [1,]    1    4    7
```

```
##  [2,]    2    5    8
##  [3,]    3    6    9
##  [4,]    1    2    3
##  [5,]    4    5    6
##  [6,]    7    8    9
##  [7,]    1    4    7
##  [8,]    2    5    8
##  [9,]    3    6    9
```

Like vectors, a subset of the matrix `x` can be extracted with `x[subscripts]`. Subscripts can be:

1. a set `[rows, cols]`, where `rows` is a vector of row numbers and `cols` is a vector of column numbers. Numbers are negative when they indicate a row or column to be excluded.
2. A number, a vector of numbers or a logical condition. In this case, the matrix is treated as if it were a single vector created by stacked matrix columns.

```
x <- matrix(1:24, nrow = 4, ncol = 6, byrow = TRUE)
x[1:2, ]
```

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

```
x[, c(1, 3, 5)]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    7    9   11
## [3,]   13   15   17
## [4,]   19   21   23
```

```
x[c(1,3), c(1, 4)]
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]   13   16
```

```
x[-1, ]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    8    9   10   11   12
## [2,]   13   14   15   16   17   18
## [3,]   19   20   21   22   23   24
```

```
x[1:18]
```

```
##  [1]  1  7 13 19  2  8 14 20  3  9 15 21  4 10 16 22  5 11
```

```
x[x >= 3 & x < 12]
```

```
## [1]  7  8  3  9  4 10  5 11  6
```

5.1.3 Array

They are similar to matrices but they can be multi-dimensional (more than two dimensions)

```
z <- array(1:24, dim=c(2,3,4))
z
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

5.1.4 Lists

A list is an ordered collection of objects. Each object is a component of the list. Each element of the list can have a different structure. It can be a list itself, a vector, a matrix, an array, a factor or a data frame. A list allows you to gather a variety of (possibly unrelated) objects under one name.

Lists are not usually created by users but are the result of statistical procedures in R.

For example, in its simplest form, the `lsfit()` function estimates a least-squares regression.

```
x <- 1:5
y <- c(-0.0921, 0.4543, -0.1473, -0.0235, -0.3923)
out <- lsfit(x, y)
out
```

```
## $coefficients
## Intercept         X
##   0.28328  -0.10782
##
## $residuals
## [1] -0.26756  0.38666 -0.10712  0.12450 -0.13648
##
## $intercept
## [1] TRUE
##
## $qr
## $qt
## [1]  0.08984521 -0.34095678  0.05740271  0.42144605  0.29288939
##
## $qr
##       Intercept          X
```

```
## [1,] -2.2360680 -6.7082039
## [2,]  0.4472136  3.1622777
## [3,]  0.4472136 -0.1954395
## [4,]  0.4472136 -0.5116673
## [5,]  0.4472136 -0.8278950
##
## $qraux
## [1] 1.447214 1.120788
##
## $rank
## [1] 2
##
## $pivot
## [1] 1 2
##
## $tol
## [1] 1e-07
##
## attr(,"class")
## [1] "qr"
```

The output of the function is a list made of four objects called “coefficients”, “residuals”, “intercept” e “qr”. The first element of the list is a vector with intercept and slope. The second element is a vector with the residuals of the model. The third element is a Boolean vector of length one indicating if the model contains the intercept. The fourth element is a list containing the QR matrix decomposition of the independent variables.

Even if rarely used, `list()` is the basic function to create a list. Its arguments are the elements of the list.

```
my_list <- list(vec = 1:7, mat = matrix(1:12, ncol = 3),
  lis = list(a = 1, b = letters[1:4]))
my_list
```

```
## $vec
## [1] 1 2 3 4 5 6 7
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## $lis
## $lis$a
## [1] 1
##
## $lis$b
## [1] "a" "b" "c" "d"
```

The elements of a list can be extracted in three different ways:

1. with square brackets;
2. with double square brackets;
3. with the name of the object inside the list.

In the first case, square brackets can be used to extract a list made of one or more objects. As for vectors, the position of the elements to be included or excluded ought to be specified.

```
my_list[1:2]
```

```
## $vec
## [1] 1 2 3 4 5 6 7
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
my_list[-3]
```

```
## $vec
## [1] 1 2 3 4 5 6 7
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
ml1 <- my_list[1]
ml1
```

```
## $vec
## [1] 1 2 3 4 5 6 7
```

In the second case, double square brackets can be used to extract one object (only) from the list.

```
ml2 <- my_list[[1]]
ml2
```

```
## [1] 1 2 3 4 5 6 7
```

```
my_list[[2]]
```

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

```
my_list[[3]]
```

```
## $a
## [1] 1
##
## $b
## [1] "a" "b" "c" "d"
```

Please note the difference between `my_list[1]` and `my_list[[1]]`. The first argument extracts a list with only the first object contained in `my_list`; in our case a vector. On the other hand, the second argument extracts the vector, which is the content of the first object of the list.

The third way enables the extraction of the content of an object in the list. The use of the object position in the list, as for double square brackets, is replaced by the use of the object name preceded by the symbol $.

```
my_list$vec
```

```
## [1] 1 2 3 4 5 6 7
```

```
my_list$mat
```

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

```
my_list$lis
```

```
## $a
## [1] 1
##
## $b
## [1] "a" "b" "c" "d"
```

Indices for the selection can be combined to extract elements in an object of the list using the above-mentioned methods.

```
my_list[[1]][1:3]
```

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

```
my_list$mat[3:4, c(1, 3)]
```

```
##      [,1] [,2]
## [1,]    3   11
## [2,]    4   12
```

```
my_list$lis[1]
```

```
## $a
## [1] 1
```

```
my_list$lis[[1]]
```

```
## [1] 1
```

```
my_list$lis$a
```

```
## [1] 1
```

5.1.5 Factors

A factor is a vector-like object used to specify a discrete classification (grouping) of the components of other vectors of the same length. R provides both *ordered* and *unordered* factors.

Factor variables are categorical variables that can be either numeric or string variables. There are a number of advantages to converting categorical variables to factor variables. Perhaps the most important advantage is that they can be used in statistical modeling where they will be implemented correctly, e.g., they will then be assigned the correct number of degrees of freedom. Factor variables are also very useful in many different types of graphics. Furthermore, storing string variables as factor variables is a more efficient use of memory.

Vectors, matrices and lists contain numerical data, characters or logics and are basic objects in R. Factors, on the other hand, are a more complex structure, as they contain both the numerical data vector and the labels associated with each level.

To create a factor variable the `factor()` function is used. The only required argument is a vector of values which can be either string or numeric. Optional arguments include the `levels` argument, which determines the categories of the factor variable, and the default is the sorted list of all the distinct values of the data vector. The `labels` argument is another optional argument which is a vector of values that will be the labels of the categories in the `levels` argument. The `exclude` argument is also optional; it defines which levels will be classified as NA in any output using the factor variable.

```
grade <- c(3, 4, 2, 2, 4, 1, 1, 4, 2, 2)
factor(grade)
```

```
##  [1] 3 4 2 2 4 1 1 4 2 2
## Levels: 1 2 3 4
```

```
gender <- c(rep("male", 3), rep("female", 4))
gender <- factor(gender, levels=c("male", "female", "trans"))
gender
```

```
## [1] male   male   male   female female female female
## Levels: male female trans
```

Once a vector has been defined, it is always possible to modify the labels of the vector's levels. This can be done with the `levels()` function.

```
size <- factor(c(2, 3, 1, 1, 1, 2, 3, 3), levels = c(1, 2, 3),
  labels = c("small", "medium", "large"))
levels(size)
```

```
## [1] "small"  "medium" "large"
```

```
levels(size) <- c("S", "M", "L")
size
```

```
## [1] M L S S S M L L
## Levels: S M L
```

The `order` parameter of the `factor()` function creates a factor with ordered levels.

```
mark <- sample(c("D", "C", "B", "A"), 12, replace = T)
mark1 <- factor(mark)
mark2 <- factor(mark, levels = c("D", "C", "B", "A"), order = T)
mark1
```

```
##  [1] C A D A A A A A C D B A
## Levels: A B C D
```

```
mark2
```

```
##  [1] C A D A A A A A C D B A
## Levels: D < C < B < A
```

The `as.numeric()` and `as.character()` functions transform a factor into a numeric vector or into a vector whose elements are the levels' labels.

```
as.numeric(size)
```

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

```
as.character(size)
```

```
## [1] "M" "L" "S" "S" "S" "M" "L" "L"
```

The elements of a factor can be extracted in the same way as the elements of a vector. The logic conditions on the elements of a factor are referred to the factor's levels which can be obtained with the`levels()` function.

```
size <- factor(c(2, 3, 1, 1, 1, 2, 3, 3), levels = c(1, 2, 3),
  labels = c("small", "medium", "large"))
size
```

```
## [1] medium large  small  small  small  medium large  large
## Levels: small medium large
```

```
size[1:5]
```

```
## [1] medium large  small  small  small
## Levels: small medium large
```

```
size[-4]
```

```
## [1] medium large  small  small  medium large  large
## Levels: small medium large
```

```
levels(size)
```

```
## [1] "small"  "medium" "large"
```

```
size[size == "medium" | size == "large"]
```

```
## [1] medium large  medium large  large
## Levels: small medium large
```

5.1.6 Data Frames

A data frame in R can be thought of as:

- a generalization of a matrix;
- a list of particular kind.

In the first case a data frame can be thought of as a matrix whose columns can be both factors and vectors of the same length but (possibly) of different types (numeric, character, Boolean).

In the second case, the data frame is a list completely made of either vectors (of any kind) or factors, all with the same length.

From a formal point of view, data frames are not a new type of objects. They are objects of list type and data frame class. From a practical point of view, a data frame is a very well-know structure in statistics. Its different kinds of information are organized in columns, whereas rows represent different types of observational units.

Furthermore, data imported in R from external sources, such as text files, Excel files or databases, is saved in R as data frame-like objects.

To sum up, unless we have specific needs, data frames in R are the ideal tool for data filing and management.

Data frames are usually imported from external sources but the creation of a data frame object in R might sometimes be needed. The most widespread method to define a data frame is the `data.frame()` function. Its inputs are a series of vectors or factors of the same length. The generated object is made of as many columns as input elements.

```
name <- c("James", "Stevie", "Otis", "Bob", "Levon", "Patti", "Karen")
height <- c(180, 170, 175, 190, 168, 160, 165)
gender <- factor(c("M", "M", "M", "M", "M", "F", "F"))
df <- data.frame(name, height, gender, stringsAsFactors = FALSE)
df
```

```
##     name height gender
## 1  James    180      M
## 2 Stevie    170      M
## 3   Otis    175      M
## 4    Bob    190      M
## 5  Levon    168      M
## 6  Patti    160      F
## 7  Karen    165      F
```

Alternatively, the vectors composing the data frame can be defined inside the `data.frame()` function itself.

```
df <- data.frame(
  height = c(180, 170, 175, 190, 168, 160, 165),
  name = c("James", "Stevie", "Otis", "Bob", "Levon", "Patti", "Karen"),
  gender = factor(c("M", "M", "M", "M", "M", "F", "F")),
  stringsAsFactors = FALSE
)
df
```

```
##   height   name gender
## 1    180  James      M
## 2    170 Stevie      M
## 3    175   Otis      M
## 4    190    Bob      M
## 5    168  Levon      M
## 6    160  Patti      F
## 7    165  Karen      F
```

The management of character vectors in R requires a detailed explanation. By default, numeric vectors become part of data frames as such, whereas character vectors are transformed into factors whose levels correspond to the vector's unique values.

This behaviour is surely effective when character vectors represent categorical variables with a definite number of modes, such as education qualification or job.

A character vector can also represent a set of strings which are not necessarily referable to a definite number of modes (e.g. person's proper names) or to numerous and/or unique modes (e.g. Italian municipalities are about 8,000 and have different names). In this case R's behaviour becomes a disturbing factor because it tends to transform variables of a different nature into factors.

Unfortunately, there is not an optimal solution to this problem. Much depends on the most used types of variables dealt with by a single user.

This behaviour is managed by the `stringsAsFactors` logical parameter.

As already mentioned, the default setting is `stringsAsFactors = TRUE` which tells R to transform character vectors into factors inside a data frame; `stringsAsFactors = FALSE` does not change character vectors. This parameter can be set both in "local" option with the `stringsAsFactors` parameter inside the `data.frame()` function and in "global" option with `stringsAsFactors` inside the `options()` function. Clearly, the "global" option of the `stringsAsFactors` lasts for the whole session of work but can be locally modified in any moment in a single call to a function.

There are several ways to extract a subset from a data frame. Data management is not among topics of this introductory manual. Like matrices, data can be extracted using `x[subscripts]` where subscripts is a set `[rows, cols]`, where `rows` is a vector of row numbers and `cols` is a vector of column numbers. Numbers are negative when they indicate a row or column to be excluded.

The following code chunk shows how to extract the first, the third and the seventh row from the `df` dataframe.

```
df[c(1, 3, 7), ]
```

```
##   height  name gender
## 1    180 James      M
## 3    175  Otis      M
## 7    165 Karen      F
```

The code below shows how to extract the first and the second column from the `df` dataframe.

```
df[, c(1, 2)]
```

```
##   height   name
## 1    180  James
## 2    170 Stevie
## 3    175   Otis
## 4    190    Bob
## 5    168  Levon
## 6    160  Patti
## 7    165  Karen
```

Like lists, data can be extracted using `x$column` where column is the column name.

```
df$height
```

```
## [1] 180 170 175 190 168 160 165
```

```
is.vector(df$height)
```

```
## [1] TRUE
```

The above code returns a vector. To get a data frame, the `data.frame()` function can be used.

```
data.frame(height = df$height)
```

```
##   height
## 1    180
## 2    170
## 3    175
## 4    190
## 5    168
## 6    160
## 7    165
```

```
data.frame(height = df$height, name = df$name)
```

```
##   height   name
## 1    180  James
## 2    170 Stevie
## 3    175   Otis
## 4    190    Bob
## 5    168  Levon
## 6    160  Patti
## 7    165  Karen
```

The `dim()` function can be used to know the number of rows and of columns of a data frame. The same information can be obtained using `nrow()` and `ncol()` functions, respectively.

```
dim(df)
```

```
## [1] 7 3
```

```
nrow(df)
```

```
## [1] 7
```

```
ncol(df)
```

```
## [1] 3
```

The `str()` function returns the structure of a dataframe. For each variable (column), it shows: the name, the type (numeric, character, factor etc.) and first elements.

```
df
```

```
##   height   name gender
## 1    180  James      M
## 2    170 Stevie      M
## 3    175   Otis      M
## 4    190    Bob      M
## 5    168  Levon      M
## 6    160  Patti      F
## 7    165  Karen      F
```

```
str(df)
```

```
## 'data.frame':    7 obs. of  3 variables:
##  $ height: num  180 170 175 190 168 160 165
##  $ name  : chr  "James" "Stevie" "Otis" "Bob" ...
##  $ gender: Factor w/ 2 levels "F","M": 2 2 2 2 2 1 1
```

The `head()` function show the first rows of a data frame. Its use is particularly convenient when data sets are long. `iris` is a built in R data set. It contains the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris.

```
nrow(iris)
```

```
## [1] 150
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

5.1.7 Missing Values, Null Objects and Infinite

In R missing values are represented by the symbol `NA` (Not Available).

```
x <- c(4, 1, "a")
y <- as.integer(x)
```

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

```
y
```

```
## [1]  4  1 NA
```

The above chunk create a vector `x`. The vector contains three values: 4, 1 and the character string "a". Then, elements of the vector are transformed into integer values.

The character string "a" cannot be transformed, so a `NA` value is returned.

To check if data is missing, the function `is.na()` can be used.

```
is.na(y)
```

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

The `NaN` symbol (Not a Number) represents a missing value obtained as a result of an impossible numerical operation. `NaN` can be detected with the function is.nan.

```
log(-1)
```

```
## Warning in log(-1): NaNs produced
```

```
## [1] NaN
```

```
sqrt(-4)
```

```
## Warning in sqrt(-4): NaNs produced
```

```
## [1] NaN
```

```
x <- log(c(-1, 1, 2))
```

```
## Warning in log(c(-1, 1, 2)): NaNs produced
```

```
x
```

```
## [1]       NaN 0.0000000 0.6931472
```

```
is.nan(x)
```

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

The `NULL` symbol represents the null object in R. `NULL` is often returned by expressions and functions whose value is undefined. The `is.null()` function returns `TRUE` if its argument is `NULL` and FALSE otherwise.

```
x <- c()
x
```

```
## NULL
```

```
is.null(x)
```

```
## [1] TRUE
```

Infinite values are represented by the `+Inf` and `-Inf` symbols in R.

```
log(0)
```

```
## [1] -Inf
```

In the above example, R calculates the limit of the function $log(x)$ as x approaches zero.

5.2 Function Object

Functions are one of the most important objects in R as when working with R we all make constant use of functions.

A function is characterised by an input and an output. The input of the function, i.e. the set of the arguments, can be either null or made of one or more R objects. The output of the function can be either null or a single R object. If the function has to return more than one object, the objects are to be inserted in a list.

Let us see some examples:

The function `ls()` doesn't require any input and returns an output.

```
ls()
```

```
##  [1] "back"     "char"     "cmat"     "comb"     "df"       "gender"   "go"       "grade"
##  [9] "height"   "logic1"   "logic2"   "mark"     "mark1"    "mark2"    "mixed"    "ml1"
## [17] "ml2"      "mydata"   "my_list"  "myrep"    "name"     "num"      "out"      "rmat"
## [25] "seqby"    "seqlength" "size"     "vec1"     "vec2"     "vec3"     "x"        "y"
## [33] "z"
```

The function `rm()` requires an input and doesn't return an output.

```
foo <- 3
rm(foo)
```

Usually, functions require an input and returns an output.

```
sum(1:10)
```

```
## [1] 55
```

It is advisable to use the help command of each function to understand not only the arguments which can be used as function inputs, but also the output:

```
help(read.table)
```

or

```
?read.table
```

Let us deeply explore function structure.

5.2.1 Function structure

We can create and assign functions to a variable name as we do with any other object:

```
f <- function(x, y = 0) {
  z <- x + y
  z
}
```

Eventually, we can delete any function with the usual call to `rm()` or `remove()`

Function structure is made up by three basic components:

- a formal arguments list
- a body
- an environment.

```
formals(f)
```

```
## $x
##
##
## $y
## [1] 0
```

```
body(f)
```

```
## {
##     z <- x + y
##     z
## }
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

5.2.1.1 Formals

Formals are the formal arguments of a function.

When we call a function, formals arguments can be specified by position or by name and we can mix positional matching with matching by name so that the following are equivalent:

```
mean(x = 1:5, trim = 0.1)
```

```
## [1] 3
```

```
mean(1:5, trim = 0.1)
```

```
## [1] 3
```

```
mean(x = 1:5, 0.1)
```

```
## [1] 3
```

```
mean(1:5, 0.1)
```

```
## [1] 3
```

```
mean(trim = 0.1, x = 1:5)
```

```
## [1] 3
```

Along with position and name, we can also specify formals by partial matching so that:

```
mean(1:5, tr = 0.1)
```

```
## [1] 3
```

```
mean(tr = 0.1, x = 1:5)
```

```
## [1] 3
```

would work anyway.

Functions formals may also have the construct `symbol = default`, that unless differently specified, forces any argument to be used with its default value.

Specifically, function `mean()` also have a third argument `na.rm` that defaults to `FALSE` and, as a result passing vectors with `NA` values to `mean()` returns `NA`

```
mean(c(1, 2, NA))
```

```
## [1] NA
```

While, by specifying `na.rm=TRUE` we get the mean of all non missing elements of vector `x`.

```
mean(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] 1.5
```

The order `R` uses for matching formals against value is:

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

5.2.1.2 Body of a function

The body of a function is a parsed R statement. In practice, this implies that the body of a function needs to be correct from a formal point of view but no evaluation of the body of a function occurred yet.

As a result, this function would return an error:

```
wrong <- function(x) {x =}
```

as its body is not a correct `R` statement.

While this function:

```
right <- function(x){x+h}
```

is accepted by `R` as is formally correct even thought, except under specific circumstances, will always return an error:

```
right(x = 2)
```

```
## Error in right(x = 2): object 'h' not found
```

The body of a function, is usually a collection of statements in braces but it can be a single statement, a symbol or even a constant.

5.2.1.3 Environment of a function

Environments are a fundamental concept in R and their knowledge is essential for advanced programming.

Environments have the following fundamental concepts:

- The environment of a function is the environment that was active at the time that the function was created. Generally, for user defined function, the Global environment:

  ```
  f <- function(x){x+1}
  environment(f)
  ```

  ```
  ## <environment: R_GlobalEnv>
  ```

  or, when a function is defined within a package, the environment associated to that package:

  ```
  environment(mean)
  ```

  ```
  ## <environment: namespace:base>
  ```

- Objects defined within a function, exists in the environment of the function itself.

```
f <- function(x){x+1}
x
```

```
## NULL
```

5.2.2 Function Examples

There are lots of function types, let us see some examples.

5.2.2.1 Mathematical Functions

The names of the basic functions and mathematical operators in R follow the standards of programming languages. In this paragraph the functions and operators enabling basic mathematical operations to be performed will be dealt with. R can also be used to perform more complex calculations, such as matrix operations or calculations with complex numbers.

Functions are usually applied to one or more vectors. In this case, operations are performed on each element of each vector. Vectors have to be the same length.

```
x <- 1:10
y <- 11:20
z <- -4:5
x + y + z
```

```
##  [1]  8 11 14 17 20 23 26 29 32 35
```

```
exp(x)
```

```
## [1]    2.718282    7.389056   20.085537   54.598150  148.413159  403.428793 1096.633158 2980.957987
##  [9] 8103.083928 22026.465795
```

```
log(z)
```

```
## Warning in log(z): NaNs produced
```

```
## [1]     NaN     NaN     NaN     NaN    -Inf 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

```
abs(z)
```

```
##  [1] 4 3 2 1 0 1 2 3 4 5
```

```
sqrt(x)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000 3.162278
```

The `sum()` function calculates the sum of all the elements of a vector.

```
sum(x)
```

```
## [1] 55
```

The `floor`, `ceiling`, `trunc` and `round` functions can be used to round a number. `floor()` returns a numeric vector containing the largest integers not greater than the corresponding input elements. `ceiling()` returns a numeric vector containing the smallest integers not less than the corresponding input elements. `trunc()` returns a numeric vector containing the integers formed by truncating the input values toward 0. `round()` rounds the values in its first argument to the specified number of decimal places (default 0).

```
floor(3.14)
```

```
## [1] 3
```

```
floor(3.67)
```

```
## [1] 3
```

```
ceiling(3.14)
```

```
## [1] 4
```

```
ceiling(3.67)
```

```
## [1] 4
```

```
trunc(3.14)
```

```
## [1] 3
```

```
trunc(3.67)
```

```
## [1] 3
```

```
round(3.14, digits = 1)
```

```
## [1] 3.1
```

```
round(3.19, digits = 1)
```

```
## [1] 3.2
```

5.2.2.2 Probabilistic Functions

Probabilistic functions in R fall into four categories:

1. `r*` functions for generating random numbers,
2. `d*` functions for calculating the value of the density function in a point,
3. `p*` functions for calculating the cumulative distribution function,
4. `q*` functions for calculating quantiles.

The asterisk indicates the distribution which is used: `norm` for normal distribution, `t` for Student's t-distribution, `binom` for binomial distribution, `gamma` distribution, `beta` distribution, `weibull` distribution, etc. R integrates numerous statistical distributions. The list of all the probability distributions included in the base R can be obtained by typing `help(Distributions)`. Other probability distributions become available when loading additional packages.

The following functions:

```
rnorm(n = 10)
```

```
##  [1]  0.02337421 -1.44410066 -0.43434366 -0.04781239 -0.34635967  0.11256413  0.07641944  0.42482086
##  [9]  0.53604628  0.31538745
```

```
rnorm(n = 20, mean = 3, sd = 5)
```

```
##  [1]  5.3239927  3.3400441  4.4148980  0.5964838 19.1105153  5.3903611  3.6090670  4.4191646 -5.1875807
## [10]  7.0353226 -4.4877227  3.4662121  5.2164005 -5.6929351  9.8080872  7.2742448 -2.4244734  9.8496841
## [19]  6.3544117  4.6772858
```

```
rbinom(n = 50, size = 20, prob = 0.8)
```

```
##  [1] 16 11 20 16 13 17 14 18 15 17 14 17 15 16 18 16 16 17 13 14 13 15 15 16 19 16 16 14 20 14 16 13 16 14
## [35] 15 18 15 17 19 16 16 14 18 14 18 19 14 18 16 19
```

```
rweibull(n = 30, shape = 5, scale = 3)
```

```
##  [1] 1.743297 2.161191 3.681191 2.646652 1.756540 2.784472 3.198930 1.682767 2.309938 2.361880 2.541066
## [12] 2.460120 1.928519 2.473901 1.855068 2.857655 2.785827 3.155953 2.253992 2.801523 2.753235 2.993436
## [23] 3.200559 2.797460 1.371322 2.415364 3.553419 2.900355 2.251122 2.618322
```

generate, respectively:

- 10 pseudorandom values from a normal distribution with parameters (0, 1);
- 20 pseudorandom values from a normal distribution with parameters (3, 5);
- 50 pseudorandom values from a binomial distribution with $n = 20$ and $\pi = 0.8$;
- 50 pseudorandom values from a Weibull distribution with parameters (5, 3).

The following functions:

```
dbinom(x = 20, size = 20, prob = 0.8)
```

```
## [1] 0.01152922
```

```
dnorm(x = -5:5, mean = 0, sd = 1)
```

```
##  [1] 1.486720e-06 1.338302e-04 4.431848e-03 5.399097e-02 2.419707e-01 3.989423e-01 2.419707e-01 5.399097
##  [9] 4.431848e-03 1.338302e-04 1.486720e-06
```

calculate, respectively:

- the probability that `x` is equal to 20, if `x` is distributed as a binomial distribution with $n = 20$ and $\pi = 0.8$;
- thee values of the density function of a standard normal for integer values comprised between -5 and 5. As expected, the highest value is obtained with 0.

The following functions:

```
pnorm(q = 0, mean = 0, sd = 1)
```

```
## [1] 0.5
```

```
pbinom(q = 20, size = 20, prob = 0.8)
```

```
## [1] 1
```

calculate, respectively:

- the value of the cumulative distribution function of a standard normal distribution at zero; as expected the result is 0.5.
- the value of a cumulative distribution function of a binomial distribution with parameters $n = 20$ and $\pi = 0.8$ at 20; as expected, the result is 1.

The following functions:

```
qnorm(p = 0.5, mean = 0, sd = 1)
```

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

```
qbinom(p = 0.5, size = 20, prob = 0.8)
```

```
## [1] 16
```

calculate, respectively:

- the quantile with which a 0.5 probability on the left is obtained in a standard normal distribution;
- The quantile with which a 0.5 probability on the left is obtained in a binomial distribution with parameters $n = 20$ and $\pi = 0.8$.

5.2.2.3 Statistical Functions

Any kind of statistical analysis can be performed in R thanks to the built-in functions of the base version or the numerous additional packages. The functions enabling the calculation of the main descriptive statistical analyses are explained below.

The `mean()`, `median()`, `sd()` and `var()` functions are used to calculate the mean, the median, the sample standard deviation and the sample variance of a numeric vector.

```
x <- mtcars$mpg
mean(x)
```

```
## [1] 20.09062
```

```
median(x)
```

```
## [1] 19.2
```

```
sd(x)
```

```
## [1] 6.026948
```

```
var(x)
```

```
## [1] 36.3241
```

The `quantile()` function calculates one or more quantiles.

```
quantile(x, .9)
```

```
##   90%
## 30.09
```

```
quantile(x, c(.3, .84))
```

```
##    30%    84%
## 15.980 26.052
```

```
quantile(x, c(.25, .50, .75))
```

```
##    25%    50%    75%
## 15.425 19.200 22.800
```

The `min()` and `max()` functions return the minimum and maximum value respectively.

```
min(x)
```

```
## [1] 10.4
```

```
max(x)
```

```
## [1] 33.9
```

The `summary()` generic function applied to a numeric vector returns minimum, maximum, quartiles and arithmetic mean.

```
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.42   19.20   20.09   22.80   33.90
```

Correlation and covariance can be calculated with the `cor()` and `cov()` functions, respectively.

```
data <- mtcars[, c(1, 3, 4, 5, 6)]
cor(data)
```

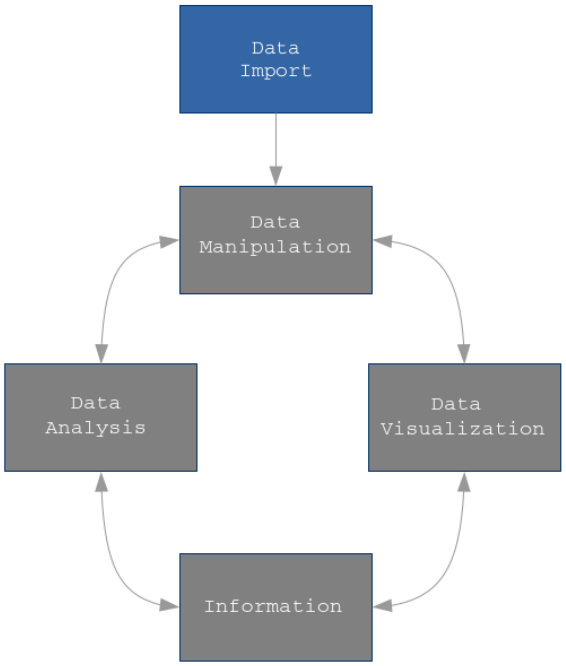
```
##             mpg       disp         hp       drat         wt
## mpg   1.0000000 -0.8475514 -0.7761684  0.6811719 -0.8676594
## disp -0.8475514  1.0000000  0.7909486 -0.7102139  0.8879799
## hp   -0.7761684  0.7909486  1.0000000 -0.4487591  0.6587479
## drat  0.6811719 -0.7102139 -0.4487591  1.0000000 -0.7124406
## wt   -0.8676594  0.8879799  0.6587479 -0.7124406  1.0000000
```

```
cov(data)
```

```
##              mpg        disp         hp        drat          wt
## mpg    36.324103  -633.09721 -320.73206   2.1950635  -5.1166847
## disp -633.097208 15360.79983 6721.15867 -47.0640192 107.6842040
## hp   -320.732056  6721.15867 4700.86694 -16.4511089  44.1926613
## drat    2.195064   -47.06402  -16.45111   0.2858814  -0.3727207
## wt     -5.116685   107.68420   44.19266  -0.3727207   0.9573790
```

Chapter 6

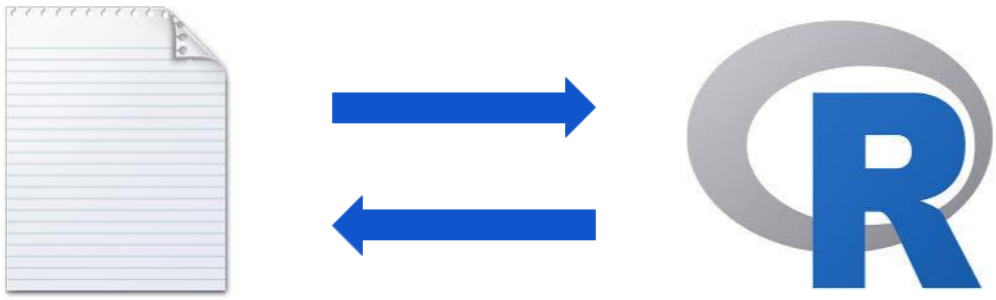
Introduction to Data Import and Export



In the following chapters we will explore data import and export methods for:

- Text files
- Microsoft Excel files
- Databases
- R data files

6.1 Text Files



6.1.1 Data Import

The `read.table()` function imports a text file (ASCII) with a table structure where each row represents a case.

A full path can be provided, but it must be modified by each user, otherwise it fails:

```
df <- read.table("C:/Users/UserName/Documents/data/tennis.txt", header = TRUE, sep = "", dec = ".")
```

```
## Warning in file(file, "rt"): cannot open file 'C:/Users/UserName/Documents/data/tennis.txt': No such fil
## directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

The path uses the slash ("/") as delimiting character, in the UNIX-like style. Under Windows, can be used both a slash character or a doubled backslash character ("\\").

So, it is strongly suggested to set the working directory to the directory containing the data.

`getwd()` function allows you to view the current working directory:

```
getwd()
```

```
## [1] "C:/Users/Andrea/Documents"
```

and `setwd()` function allows you to set the working directory on "data" folder, in this way:

```
setwd("./data")
```

```
## [1] "C:/Users/Andrea/Documents/data"
```

Now, the the text file can be imported just providing its filename:

```
df <- read.table("tennis.txt", header = TRUE)
```

```
head(df)
```

```
##         Name First.Name Age Sex Rank Slams Won Lost Earnings Citizen
## 1    Sampras       Pete  23   M    1     2  74   11 3607.812      US
## 2     Agassi      Andre  24   M    2     1  51   13 1941.667      US
## 3     Becker      Boris  27   M    3     0  48   16 2029.756 Germany
## 4   Bruguera      Sergi  24   M    4     1  65   23 3031.874   Spain
## 5 Ivanisevic      Goran  23   M    5     0  63   26 2060.278 Croatia
## 6       Graf     Steffi  25   F    1     1  58    6 1487.980 Germany
```

The `header = TRUE` option tells R that the first row of the file contains column headings and it is used to assign the name of variables. If the first row contains the first case the `header = FALSE` ought to be used and the names of the variables are automatically assigned. R assumes a default value for the `header` parameter according to the file format, which is why specifying the correct option is advisable. Alternatively, the names of the columns can be specified using the `col.names` parameter. This parameter requires a character vector with the same length as the number of the data frame columns.

The `sep` argument specifies the separator between different cases. The default value for the `read.table()` function is `sep = ""` which takes into consideration the fields delimited by a white space, be it one or more spaces or tabulations.

The `dec` argument specifies the decimal separator. The argument usually assumes the `dec = "."` (default) or `dec = ","` values.

The `nrows` argument specifies the maximum number of rows to read in.

```
read.table("tennis.txt", header = TRUE, sep = "", dec = ".", nrows = 3)
```

```
##      Name First.Name Age Sex Rank Slams Won Lost Earnings Citizen
## 1 Sampras       Pete  23   M    1     2  74   11 3607.812      US
## 2  Agassi      Andre  24   M    2     1  51   13 1941.667      US
## 3  Becker      Boris  27   M    3     0  48   16 2029.756 Germany
```

The `skip` argument specifies the number of lines of the data file to skip before beginning to read data. If the first line contains the header and it is ignored, than `header = FALSE` should be set.

```
read.table("tennis.txt", header = FALSE, sep = "", dec = ".", skip = 2)
```

```
##                V1       V2 V3 V4 V5 V6 V7 V8       V9            V10
## 1          Agassi    Andre 24  M  2  1 51 13 1941.667             US
## 2          Becker    Boris 27  M  3  0 48 16 2029.756        Germany
## 3        Bruguera    Sergi 24  M  4  1 65 23 3031.874          Spain
## 4      Ivanisevic    Goran 23  M  5  0 63 26 2060.278        Croatia
## 5            Graf   Steffi 25  F  1  1 58  6 1487.980        Germany
## 6 Sanchez Vicario  Arantxa 23  F  2  2 74  9 2943.665          Spain
## 7        Martinez Conchita 22  F  3  1 55 15 1540.167          Spain
## 8         Novotna     Jana 26  F  4  0 43 11  876.119 Czech Republic
## 9          Pierce     Mary 20  F  5  0 45 18  768.614         France
```

The `nrows` and `skip` arguments can be mixed. The following example read the second and the third rows of the data frame.

```
read.table("tennis.txt", header = FALSE, sep = "", dec = ".", nrows = 2, skip = 2)
```

```
##       V1    V2 V3 V4 V5 V6 V7 V8       V9     V10
## 1 Agassi Andre 24  M  2  1 51 13 1941.667      US
## 2 Becker Boris 27  M  3  0 48 16 2029.756 Germany
```

Variables containing text are set as character variables with the `stringsAsFactors = FALSE` option, whereas by default they are set as factors. This setting can be modified with the "global" option for it to be applied until the end of the work session. This can be done with the `options(stringsAsFactors = FALSE)` instruction.

```
df <- read.table("tennis.txt", header = TRUE, sep = "", dec = ".", stringsAsFactors = FALSE)
head(df)
```

```
##         Name First.Name Age Sex Rank Slams Won Lost Earnings Citizen
## 1    Sampras       Pete  23   M    1     2  74   11 3607.812      US
## 2     Agassi      Andre  24   M    2     1  51   13 1941.667      US
## 3     Becker      Boris  27   M    3     0  48   16 2029.756 Germany
```

```
## 4   Bruguera      Sergi  24   M    4     1  65   23 3031.874   Spain
## 5 Ivanisevic      Goran  23   M    5     0  63   26 2060.278 Croatia
## 6       Graf     Steffi  25   F    1     1  58    6 1487.980 Germany
```

When there are missing values the `na.strings` can be used to indicate which string is referred to them. The `na.string` argument can be a character vector. R indicates missing values with the `NA` (Not Available) symbol.

```
# Data frame imported without na.strings parameter
df <- read.table("tennis.NA.txt", header = TRUE, sep = "", dec = ".", stringsAsFactors = FALSE)
head(df)
```

```
##         Name First.Name Age Sex Rank Slams Won Lost Earnings Citizen
## 1    Sampras       Pete  23   M    1     2  74   11 3607.812      US
## 2     Agassi      Andre  24   M    2     1  51   13 1941.667      US
## 3     Becker      Boris  MC   M    3     0  48   16 2029.756 Germany
## 4   Bruguera      Sergi  24   M    4     1  65   23 3031.874   Spain
## 5 Ivanisevic      Goran  ND   M    5     0  63   26 2060.278 Croatia
## 6       Graf     Steffi  25   F    1     1  58    6 1487.980 Germany
```

```
# Data frame imported considering also na.strings parameter
df <- read.table("tennis.NA.txt", header = TRUE, sep = "", dec = ".", na.strings = c("MC", "ND"), string
head(df)
```

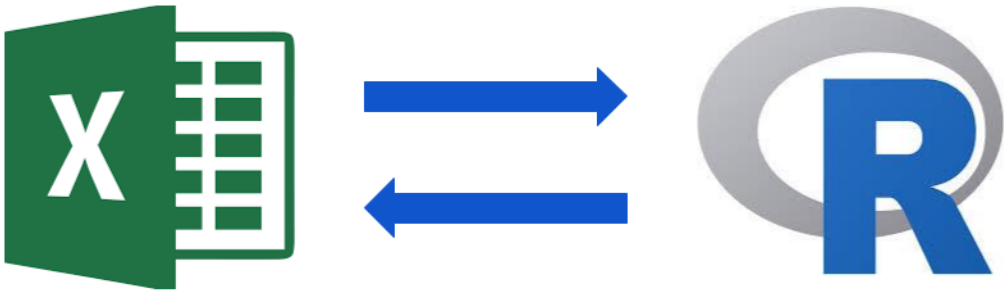
```
##         Name First.Name Age Sex Rank Slams Won Lost Earnings Citizen
## 1    Sampras       Pete  23   M    1     2  74   11 3607.812      US
## 2     Agassi      Andre  24   M    2     1  51   13 1941.667      US
## 3     Becker      Boris  NA   M    3     0  48   16 2029.756 Germany
## 4   Bruguera      Sergi  24   M    4     1  65   23 3031.874   Spain
## 5 Ivanisevic      Goran  NA   M    5     0  63   26 2060.278 Croatia
## 6       Graf     Steffi  25   F    1     1  58    6 1487.980 Germany
```

6.1.2 Data Export

To save a data frame in a text file in R use the `write.table()` function.

```
# It creates a df_write.txt file in the current directory containing df data frame
df <- data.frame(a1 = rnorm(10), a2 = rnorm(10), a3 = rnorm(10))
write.table(df, file = "df_write.txt")
```

6.2 Interacting with Microsoft Excel Files



6.2.1 XLConnect

The R package `XLConnect` permits to create a formatted spreadsheet usable as a dynamic report of the R analisys and it allows one to read existing xlsx files and to modify them from R.
Let us see how `XLConnect` works.

```
require(XLConnect)
```

6.2.1.1 Create a new file xlsx

To create a new empty file xlsx with one empty sheet named *Input* the syntax is:

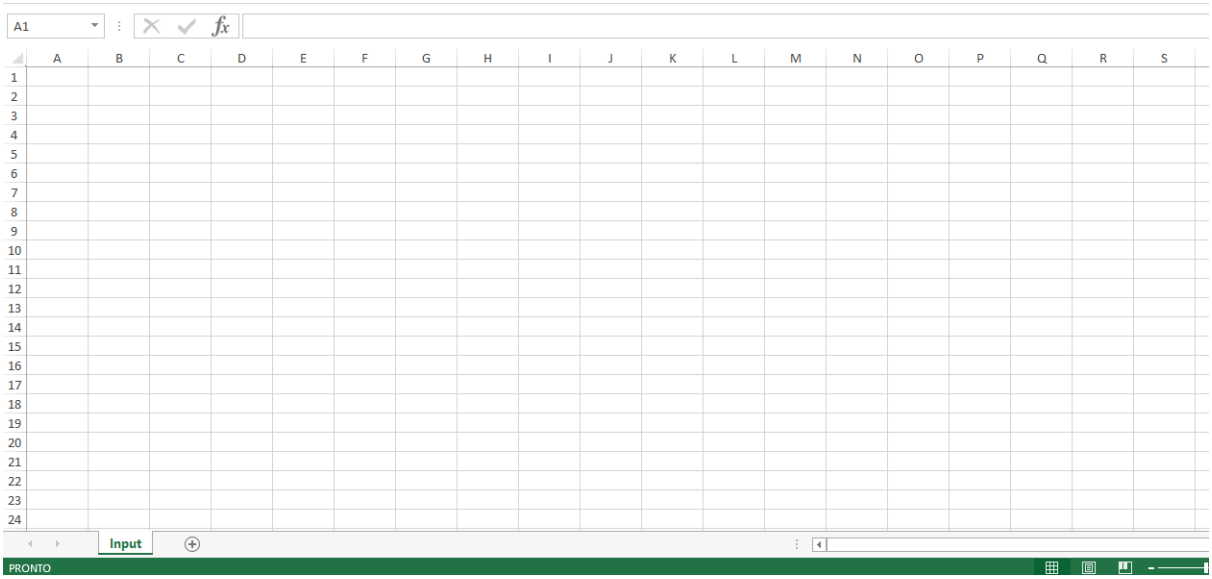
```
# Set up output directory and output file name
outDir <- "./xlsx"
```

```
# File path string
file_xls <- paste(outDir,"newFile.xlsx",sep='/')
# Delete file_xls if it already exists
unlink(file_xls, recursive = FALSE, force = FALSE)
```

```
exc <- loadWorkbook(filename = file_xls, create = TRUE)
createSheet(object = exc, name = 'Input')
saveWorkbook(exc)
```

`loadWorkbook()` function creates an R workbook object in the path and with the name specified by `filename` argument. It creates it ex-novo, as `create` argument is set as `TRUE`. An R workbook object represents a Microsoft Excel workbook.

The function `createSheet()` creates the worksheet *Input* in R object and `saveWorkbook()` function fisically save the R object in a file xlsx. Remember to call this function every time you modified the R object in order to save the changes also in xlsx file.

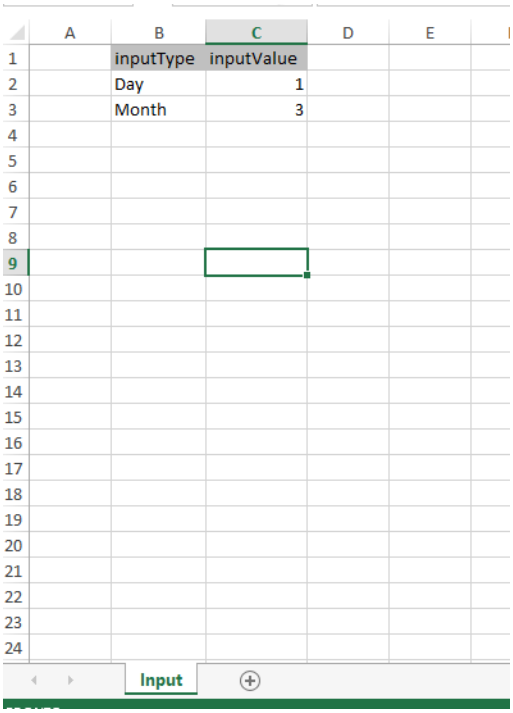


6.2.1.2 Populate a sheet

To add something to an empty sheet use `writeWorkbook` function:

```
df <- data.frame('inputType'=c('Day','Month'),'inputValue'=c(1,3))
writeWorksheet(object = exc, data = df, sheet = "Input", startRow = 1, startCol = 2)
saveWorkbook(exc)
```

The `df` data frame with 2 rows and 2 column is created and `writeWorkbook()` function write the content of this data frame in the sheet *Input* starting from the cell (1,2).



6.2.1.3 Create multiple sheets

To add other sheets to an R object workbook, use `createSheet()` function:

```
# Add a sheet named Airquality to exc object
createSheet(exc,'Airquality')
saveWorkbook(exc)
```

Suppose we want to add a dataset to the sheet just created.
We want to add `airquality` dataset available in `datasets` package, which reports daily air quality measurements in New York, from May to September 1973.

```
# Add an empty column to airquality dataset before add it to 'Airquality' sheet
airquality$isCurrent<-NA
# Add airquality dataset to the sheet Airquality
createName(exc, name='Airquality',formula='Airquality!$A$1')
writeNamedRegion(exc, airquality, name = 'Airquality', header = TRUE)
saveWorkbook(exc)
```

In particular, `createName()` function creates a named region 'Airquality' starting from the cell $A$1 of sheet *Airquality*. In Excel, a named region/range represents cells, a range of cells, a constant value, or a formula with a defined name which make easier to work. It is useful for navigation, to quickly select the named range, for reusing it when referencing it in such things as charts and formulas, …
`writeNamedRegion()` function writes `airquality` data frame with headers `(header=TRUE)` in the named region *Airquality*.

	A	B	C	D	E	F	G
1	Ozone	Solar.R	Wind	Temp	Month	Day	isCurrent
2	41	190	7,4	67	5	1	
3	36	118	8	72	5	2	
4	12	149	12,6	74	5	3	
5	18	313	11,5	62	5	4	
6			14,3	56	5	5	
7	28		14,9	66	5	6	
8	23	299	8,6	65	5	7	
9	19	99	13,8	59	5	8	
10	8	19	20,1	61	5	9	
11		194	8,6	69	5	10	
12	7		6,9	74	5	11	
13	16	256	9,7	69	5	12	
14	11	290	9,2	66	5	13	
15	14	274	10,9	68	5	14	
16	18	65	13,2	58	5	15	
17	14	334	11,5	64	5	16	
18	34	307	12	66	5	17	
19	6	78	18,4	57	5	18	
20	30	322	11,5	68	5	19	
21	11	44	9,7	62	5	20	
22	1	8	9,7	59	5	21	
23	11	320	16,6	73	5	22	
24	4	25	9,7	61	5	23	

Input Airquality

PRONTO

6.2.1.4 Add a formula

Use `setCellFormula()` function to set cell formulas for specific cells in a workbook.
The empty column *isCurrent* in *Airquality* sheet could be populate with a formula that lies *Input* sheet with *Airquality* sheet.

```
# Define the column index of the cell to edit
col_index <- which(names(airquality) == 'isCurrent')
# Define the excel letter for the column 'Day' and 'Month' needed by the formula
letter_day <- idx2col(which(names(airquality) == 'Day'))
letter_month <- idx2col(which(names(airquality) == 'Month'))
```

The function `idx2col()` returns the correspondig excel letter for the index column. With the syntax:

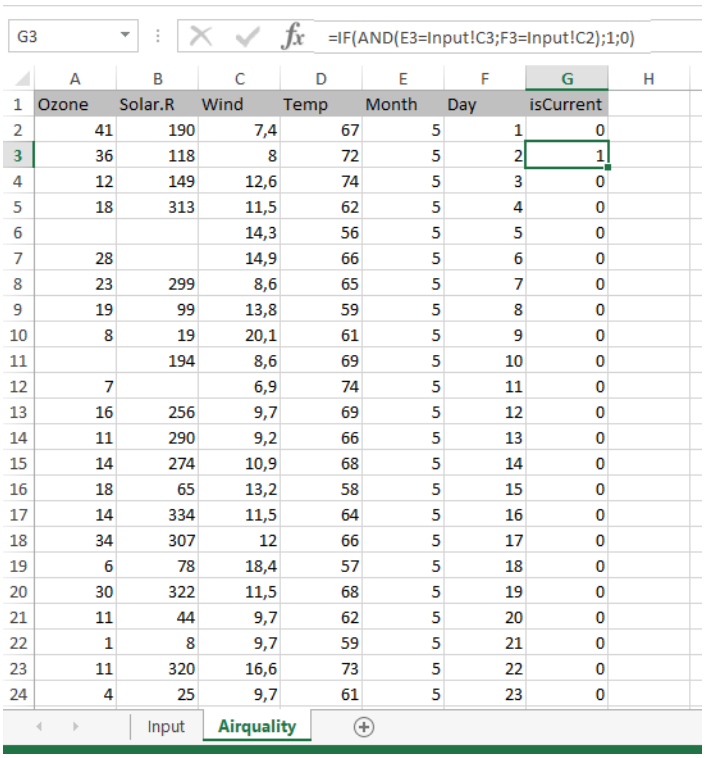
```
letter_day <- idx2col(which(names(airquality) == 'Day'))
```

the variable `letter_day` contains the excel letter for the column *Day*

```
## letter_day= F
```

```
# Define the formula to apply to the cell
formula_xls <- paste('IF(AND(',
                     letter_month,
                     2:(nrow(airquality)+1),
                     '=Input!C3,',
                     letter_day,
                     2:(nrow(airquality)+1),
                     '=Input!C2)',
                     ',1,0)',sep='')
setCellFormula(exc, sheet='Airquality', row = 2:(nrow(airquality)+1), col = col_index, formula = formula
saveWorkbook(exc)
```

The function `setCellFormula()` apply the formula specified by the argument `formula` to the rows specified by the `row` argument of the column specified by the `col` argument of the sheet of the R object specified by the `sheet` argument.



	A	B	C	D	E	F	G	H
1	Ozone	Solar.R	Wind	Temp	Month	Day	isCurrent	
2	41	190	7,4	67	5	1	0	
3	36	118	8	72	5	2	1	
4	12	149	12,6	74	5	3	0	
5	18	313	11,5	62	5	4	0	
6			14,3	56	5	5	0	
7	28		14,9	66	5	6	0	
8	23	299	8,6	65	5	7	0	
9	19	99	13,8	59	5	8	0	
10	8	19	20,1	61	5	9	0	
11		194	8,6	69	5	10	0	
12	7		6,9	74	5	11	0	
13	16	256	9,7	69	5	12	0	
14	11	290	9,2	66	5	13	0	
15	14	274	10,9	68	5	14	0	
16	18	65	13,2	58	5	15	0	
17	14	334	11,5	64	5	16	0	
18	34	307	12	66	5	17	0	
19	6	78	18,4	57	5	18	0	
20	30	322	11,5	68	5	19	0	
21	11	44	9,7	62	5	20	0	
22	1	8	9,7	59	5	21	0	
23	11	320	16,6	73	5	22	0	
24	4	25	9,7	61	5	23	0	

6.2.1.5 Read an existing xlsx file

To read an existing excel file, the syntax is:

```
# Excel file (with path) to be loaded into R
file_xls <- "./xlsx/newFile.xlsx"
```

```
exc2 <- loadWorkbook(file_xls)
dt_air <- readWorksheet(exc2, sheet = 'Airquality')
head(dt_air)
```

```
##   Ozone Solar.R Wind Temp Month Day isCurrent
## 1    41     190  7.4   67     5   1         0
## 2    36     118  8.0   72     5   2         0
## 3    12     149 12.6   74     5   3         0
## 4    18     313 11.5   62     5   4         0
## 5    NA      NA 14.3   56     5   5         0
## 6    28      NA 14.9   66     5   6         0
```

`loadWorkbook()` function loads a Microsoft Excel workbook, in this case "newFile.xlsx", into R creating a R workbook object, `exc2`.
`readWorksheet()` function reads data from *Airquality* sheet of `exc2` object (the workbook that has been previously loaded).

6.2.1.6 Modify an existing xlsx file

Suppose we want to create another sheet named *OzonePlot*, with a named region *OzonePlot*:

```
createSheet(exc2, name = "OzonePlot")
createName(exc2, name='OzonePlot',formula='OzonePlot!$A$1')
saveWorkbook(exc2)
```

`createSheet()` function adds the new sheet *OzonePlot* to `exc2` object and `createName()` function creates a new named region *OzonePlot* starting from *OzonePlot!$A$1* cell. `saveWorkbook()` function fisically save the change done to R object also in the corresponding xlsx file, in this case "newFile.xlsx".

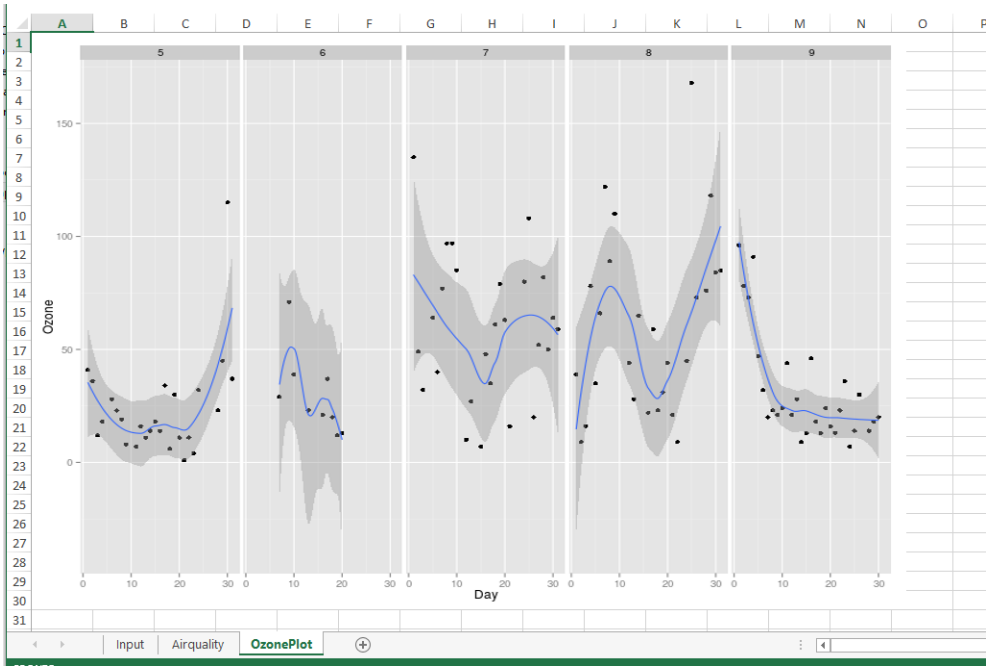
6.2.1.7 Adding a plot (image)

After creating a new sheet it is possible to put in this sheet a picture of a graph created in R with the function `addImage()`:

```
require(ggplot2)
# Generate a graph and save it in png format
fileGraph <- paste(outDir,'graph.png',sep='/')
png(filename = fileGraph, width = 800, height = 600)
ozone_plot <- ggplot(dt_air, aes(x=Day, y=Ozone)) +
geom_point() +
geom_smooth()+
facet_wrap(~Month, nrow=1)
print(ozone_plot)
invisible(dev.off())
# Add image file created to 'OzonePlot' named region with its original size
addImage(exc2, filename =  fileGraph, name = 'OzonePlot', originalSize = TRUE)
saveWorkbook(exc2)
```

```
# Remove the graph file created
file.remove(fileGraph)
```

```
FALSE [1] TRUE
```



6.2.2 readxl

Another R package for importing excel files into R is `readxl`. Let us see how `readxl` works.

```
require(readxl)
```

6.2.2.1 Read an existing xlsx file

To read an existing excel file, the syntax is:

```
# Excel file (with path) to be loaded into R
file_xls <- "./xlsx/newFile.xlsx"
```

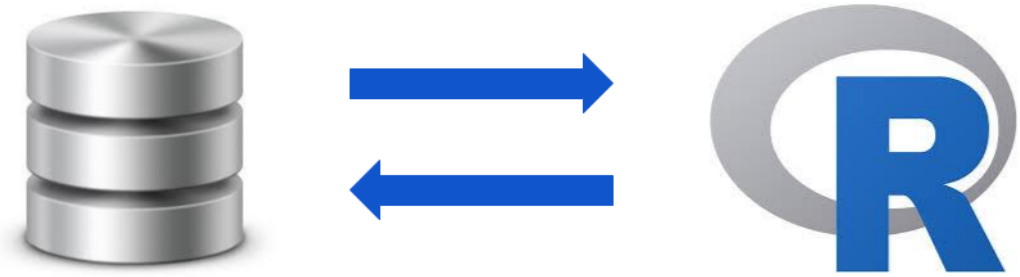
```
ds <-read_excel(path = file_xls, sheet = 'Airquality', col_names = TRUE)
head(ds)
```

```
## # A tibble: 6 × 7
##   Ozone Solar.R  Wind  Temp Month   Day isCurrent
##   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>     <dbl>
## 1    41     190   7.4    67     5     1         0
## 2    36     118   8.0    72     5     2         0
## 3    12     149  12.6    74     5     3         0
## 4    18     313  11.5    62     5     4         0
## 5    NA      NA  14.3    56     5     5         0
## 6    28      NA  14.9    66     5     6         0
```

`read_excel()` function allows us to read xls and xlsx files, specified in `path` argument. `sheet` argument specifies the sheet to read and `col_names` indicates if the first row has to be used as column names (set as

`TRUE`).

6.3 Working with databases



6.3.1 ODBC

Open Database Connectivity (ODBC) is a standard programming language interface for accessing database management systems (DBMS). ODBC is independent from database systems and operating systems. An application can use ODBC to query data from a DBMS, regardless of the operating system or DBMS it uses. ODBC accomplishes DBMS independence by using an ODBC driver as a translation layer between the application and the DBMS.

With the `RODBC` package R enables the use of ODBC for interacting with databases. This solution is particularly useful when data occupies much space, is frequently updated or shared by two or more users. In this case, data is kept in the database. With R it is possible to make a query in the database, load data in the R workspace and carry out analyses.

The following code shows some examples of how to use ODBC in a MySQL database. For the following examples to work, it is necessary to modify the following functions with the parameters related to the available MySQL database.

The `odbcConnect()` function establishes the connection to the MySQL database. Its main arguments are: `dsn`, a string containing the name of the data source, `uid` and `pwd`, i.e. the user name and the password for the login.
The `sqlQuery()` function performs queries to the MySQL database. The use of single and double quotation marks require attention. In the following example the query is contained in a string and is delimited by double quotation marks. The strings belonging to the query are delimited by single quotation marks.

Finally, the `odbcClose()` function closes the connection to the database.

```
# RODBC driver ought be configured to work properly
require(RODBC)
conn = odbcConnect(dsn = "test", uid = "user", pwd = "pass")
sqlQuery(conn, "select * from tbl where gender = 'F'")
odbcClose(conn)
```

6.3.2 SQLlite

`RSQLite` package embeds the SQLite database engine in R, providing a DBI-compliant interface. SQLite is a public-domain, single-user, very light-weight database engine that implements a decent subset of the SQL 92 standard, including the core table creation, updating, insertion, and selection operations, plus transaction management.

```
require(RSQLite)
```

The function `dbConnect` connect to a SQLite database, or creates it if it doesn't exist, as in this case:

```
con <- dbConnect(RSQLite::SQLite(), "mtcars.sqlite")
```

To write a local data frame to the database, `dbWriteTable` is required:

```
dbWriteTable(con, "mtcars", mtcars)
```

```
## [1] TRUE
```

`dbDisconnect` disconnects from the database:

```
dbDisconnect(con)
```

```
## [1] TRUE
```

Now `mtcars.sqlite` exists and `dbConnect` connects us to it:

```
con <- dbConnect(RSQLite::SQLite(), "mtcars.sqlite")
```

To see a list of available SQLite tables:

```
dbListTables(con)
```

```
## [1] "mtcars"
```

or a list of fields in specified table:

```
dbListFields(con, "mtcars")
```

```
##  [1] "row_names" "mpg"      "cyl"      "disp"     "hp"       "drat"     "wt"       "qsec"
##  [9] "vs"        "am"        "gear"      "carb"
```

The next function mimic their R/S-Plus counterpart get, assign, exists, remove, and objects, except that they generate code that gets remotely executed in a database engine:

```
dbReadTable(con, "mtcars")
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
```

```
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

The function `dbGetQuery` send query, retrieve results and then clear result set:

```
dbGetQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
```

```
##         row_names  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1      Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## 2       Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## 3        Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## 4        Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 5     Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## 6  Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## 7   Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## 8       Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## 9   Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## 10   Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## 11     Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

And finally disconnect from the database:

```
dbDisconnect(con)
```

```
## [1] TRUE
```

6.4 R Data Files

6.4.1 Save R Data Files

Statistical packages often provide the opportunity to save the working environment with all the objects it contains in their own formats. Even if rarely used, this function is available in R as well. The format used by R is called `Rdata` (or `Rda`).

In this way, different objects can be saved in a single file. Moreover, all the features of a data frame which cannot be saved in a text file, such as the levels of a factor, can be kept in the file.

To save an object of the R workspace in a file use the `save()` function. The first argument of the function is the object to be saved, whereas the file name is defined in the `file` argument. If the position is not specified, R saves the file in the current directory.

```
# It creates a mtcars.Rda file in the current directory
save(mtcars, file = "mtcars.Rda")
```

To save more than one object list their names.

```
# It creates a datasets.Rda file in the current directory
save(mtcars, iris, file = "datasets.Rda")
```

An alternative method to save more than one object is provided by the `list` argument. The names of the objects to be saved in a vector can be inserted with the `list` argument. This method is advisable when the list of the files to be saved is contained in a vector.

```
# It creates a datasets.Rda file in the current directory
datalist = c("mtcars", "iris")
save(list = datalist, file = "datasets.Rda")
```

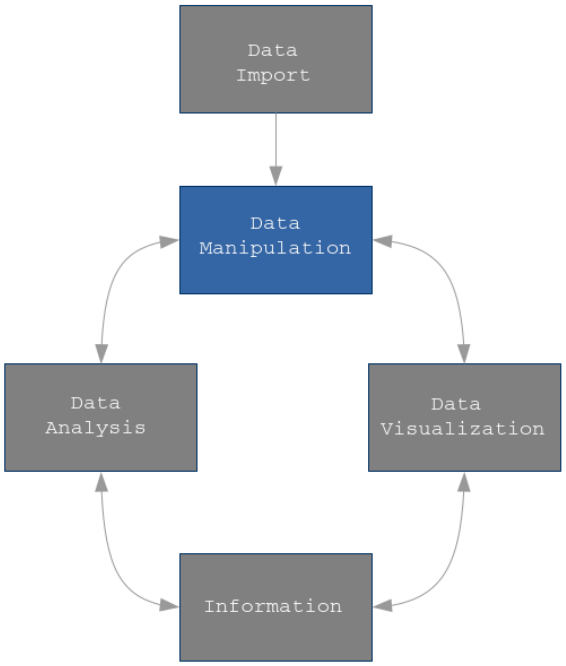
6.4.2 Load R Data Files

To upload `Rda` files in R use the `load()` function.

```
# It reads the datasets.Rda file previously created in the current directory
load("datasets.Rda")
```

Chapter 7

Introduction to dplyr



This chapter provides an overview of data management with `R` through the `dplyr`, `tidyr` and `lubridate` packages.

The `dplyr` package for `R` is very powerful for data management since:

- it simplifies how you can think about common data manipulation tasks;
- it provides simple "verbs", functions that correspond to the most common data manipulation tasks;
- it uses efficient data storage backends, so you spend less time waiting for the computer.

```
require(dplyr)
```

`tidyr` is a modern `R` package which provides a standard way to organise data values within a dataset.

```
require(tidyr)
```

`lubridate` is an R packages that deals with dates and times in an intuitive and coherent way.

```
require(lubridate)
```

In the following chapters, we will explore the innovations introduced by `dplyr`, `tidyr` and `lubridate` to make our lifes easier when dealing with dataframes manipulation tasks.
In particular:

- pipe operator (`%>%`)
- `tbl_df` data frame class
- `dplyr` verbs for data manipulation
- `dplyr` verbs for combining data
- `dplyr` with backend databases
- reshaping data with `tidyr`
- managing dates with `lubridate`

In the following two paragraphs we will explore two important `dplyr` innovations: pipe operator (`%>%`) and `tbl_df` data frame class

7.1 Pipe operator

`dplyr` pipe operator (`%>%`) allows us to pipe the output from one function to the input of another function. The idea of piping is to read the functions from left to right. It is particularly useful with nested functions (reading from the inside to the outside) or with multiple operations.

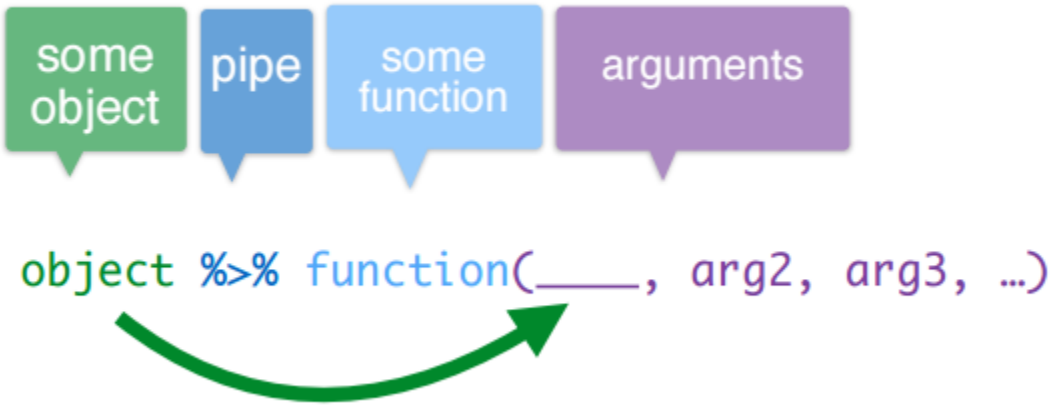


Figure 7.1: Source: www.datacamp.com

Pipes can work with nearly any functions (`dplyr` and not-`dplyr` functions), let us see an example.

Let us consider `bank` data set, included in `qdata` package, which contains information about a direct marketing campaigns of a Portuguese banking institution based on phone calls.

```
require(qdata)
```

```
## Loading required package: qdata
```

```
data(bank)
```

Suppose we want to visualize the first rows of `bank` dataframe, by using `head()` function.

Usually we write:

```
head(bank)
```

```
## # A tibble: 6 × 20
##    id  age       job marital education default balance housing  loan contact  day  month  year
##  <int> <int>     <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1    1   58  management married  tertiary     no   2143    yes    no unknown    5   may  2008
## 2    2   44  technician  single secondary     no     29    yes    no unknown    5   may  2008
## 3    3   33 entrepreneur married secondary     no      2    yes   yes unknown    5   may  2008
## 4    4   47  blue-collar married   unknown     no   1506    yes    no unknown    5   may  2008
## 5    5   33     unknown  single   unknown     no      1     no    no unknown    5   may  2008
## 6    6   35  management married  tertiary     no    231    yes    no unknown    5   may  2008
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

By using %>%, the code becomes:

```
bank %>% head()
```

```
## # A tibble: 6 × 20
##    id  age       job marital education default balance housing  loan contact  day  month  year
##  <int> <int>     <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1    1   58  management married  tertiary     no   2143    yes    no unknown    5   may  2008
## 2    2   44  technician  single secondary     no     29    yes    no unknown    5   may  2008
## 3    3   33 entrepreneur married secondary     no      2    yes   yes unknown    5   may  2008
## 4    4   47  blue-collar married   unknown     no   1506    yes    no unknown    5   may  2008
## 5    5   33     unknown  single   unknown     no      1     no    no unknown    5   may  2008
## 6    6   35  management married  tertiary     no    231    yes    no unknown    5   may  2008
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

Pipe takes the argument on the left (`bank`) and passes it to the function on the right (`head()`). So you don't need to write the first argument of the function.

Other arguments of the function must be added to the function itself, as usually done. By default `head()` prints the first 6 rows of the dataframe. Suppose we want to print 10 rows, by setting `n` argument to 10:

```
bank %>% head(n=10)
```

```
## # A tibble: 10 × 20
##     id  age       job  marital education default balance housing  loan contact  day  month  year
##   <int> <int>     <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1     1   58  management  married  tertiary     no   2143    yes    no unknown    5   may  2008
## 2     2   44  technician   single secondary     no     29    yes    no unknown    5   may  2008
## 3     3   33 entrepreneur  married secondary     no      2    yes   yes unknown    5   may  2008
## 4     4   47  blue-collar  married   unknown     no   1506    yes    no unknown    5   may  2008
## 5     5   33     unknown   single   unknown     no      1     no    no unknown    5   may  2008
## 6     6   35  management  married  tertiary     no    231    yes    no unknown    5   may  2008
## 7     7   28  management   single  tertiary     no    447    yes   yes unknown    5   may  2008
## 8     8   42 entrepreneur divorced  tertiary    yes      2    yes    no unknown    5   may  2008
## 9     9   58     retired  married   primary     no    121    yes    no unknown    5   may  2008
## 10   10   43  technician   single secondary     no    593    yes    no unknown    5   may  2008
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
```

```
## #   poutcome <fctr>, y <fctr>
```

7.1.1 tbl_df: the dplyr Data Frame Class

Sometimes data frames have large dimensions. `dplyr` package provide `tbl_df`, which is a wrapper around a data frame that will not accidentally print a lot of data to the screen; indeed tbl objects only print a few rows and all the columns that fit on one screen, describing the rest of it as text.

When the class of data object is not tbl, `tbl_df()` function should be used.
Let us consider `mtcars`, a dataset included in `datasets` package (automatically loaded at the start of an R session):

```
# Example of data frame
class(mtcars)
```

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

```
# If we do not convert it as a tbl_df, all mtcars rows and columns will be printed when calling mtcars
dim(mtcars)
```

```
## [1] 32 11
```

```
mtcars
```

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

```
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
# dplyr version of the same data frame (tbl_df conversion)
mtcars_tbl <- tbl_df(mtcars)
class(mtcars_tbl)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
mtcars_tbl
```

```
## # A tibble: 32 × 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
## *  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1   21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
## 2   21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
## 3   22.8     4 108.0    93  3.85 2.320 18.61     1     1     4     1
## 4   21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
## 5   18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
## 6   18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
## 7   14.3     8 360.0   245  3.21 3.570 15.84     0     0     3     4
## 8   24.4     4 146.7    62  3.69 3.190 20.00     1     0     4     2
## 9   22.8     4 140.8    95  3.92 3.150 22.90     1     0     4     2
## 10  19.2     6 167.6   123  3.92 3.440 18.30     1     0     4     4
## # ... with 22 more rows
```

7.2 Verb Functions

```
require(dplyr)
require(qdata)
```

`dplyr` aims to provide a function for each basic verb of data manipulating.

All these functions are very similar:

- the first argument is a data frame;
- the subsequent arguments describe what to do with it, and you can refer to columns in the data frame directly without using $. Note that the column names must be unquoted;
- the result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

These five functions provide the basis of a language of data manipulation. At the most basic level, you can only alter a tidy data frame in five useful ways:

1. select variables of interest: `select()`;
2. filter records of interest: `filter()`;
3. reorder the rows: `arrange()`;
4. add new variables that are functions of existing variables: `mutate()`;
5. collapse many values to a summary: `summarise()`.

In the following examples we will refer to `bank` data set which contains information about a direct marketing campaigns of a Portuguese banking institution based on phone calls.

```
data(bank)
```

7.2.1 select()

Often you work with large datasets with many columns where only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset of columns.

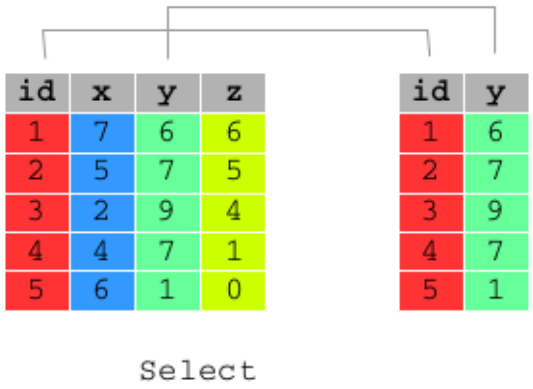


Figure 7.2:

The first argument is the name of the data frame, and the second and subsequent are the name of column/s of that data frame you want to select:

```
# Select columns: year, month and day of bank data frame
select(bank, year, month, day)
```

```
## # A tibble: 45,211 × 3
##     year  month   day
##    <int> <fctr> <int>
## 1   2008    may     5
## 2   2008    may     5
## 3   2008    may     5
## 4   2008    may     5
## 5   2008    may     5
## 6   2008    may     5
## 7   2008    may     5
## 8   2008    may     5
## 9   2008    may     5
## 10  2008    may     5
## # ... with 45,201 more rows
```

```
# Select columns: year, month and day of bank data frame
bank %>% select(year:day)
```

```
## # A tibble: 45,211 × 3
##     year  month   day
```

```
##    <int> <fctr> <int>
## 1   2008    may     5
## 2   2008    may     5
## 3   2008    may     5
## 4   2008    may     5
## 5   2008    may     5
## 6   2008    may     5
## 7   2008    may     5
## 8   2008    may     5
## 9   2008    may     5
## 10  2008    may     5
## # ... with 45,201 more rows
```

```
# Select all columns of bank data frame apart from: year, month and day
bank %>% select(-(year:day))
```

```
## # A tibble: 45,211 × 17
##     id  age       job marital education default balance housing  loan contact     date duration
##   <int> <int>     <fctr>  <fctr>   <fctr> <fctr>  <int> <fctr> <fctr>  <fctr>    <dttm>   <int>
## 1     1   58  management married  tertiary     no   2143    yes    no unknown 2008-05-05      261
## 2     2   44  technician  single secondary     no     29    yes    no unknown 2008-05-05      151
## 3     3   33 entrepreneur married secondary     no      2    yes   yes unknown 2008-05-05       76
## 4     4   47 blue-collar married   unknown     no   1506    yes    no unknown 2008-05-05       92
## 5     5   33     unknown  single   unknown     no      1     no    no unknown 2008-05-05      198
## 6     6   35  management married  tertiary     no    231    yes    no unknown 2008-05-05      139
## 7     7   28  management  single  tertiary     no    447    yes   yes unknown 2008-05-05      217
## 8     8   42 entrepreneur divorced  tertiary    yes      2    yes    no unknown 2008-05-05      380
## 9     9   58     retired married   primary     no    121    yes    no unknown 2008-05-05       50
## 10   10   43  technician  single secondary     no    593    yes    no unknown 2008-05-05       55
## # ... with 45,201 more rows, and 5 more variables: campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

You can rename variables with `select()` by using named arguments:

```
# Rename id variable as ID
bank %>% select(ID = id)
```

```
## # A tibble: 45,211 × 1
##       ID
##    <int>
## 1      1
## 2      2
## 3      3
## 4      4
## 5      5
## 6      6
## 7      7
## 8      8
## 9      9
## 10    10
## # ... with 45,201 more rows
```

7.2.2 filter()

`filter()` allows you to select a subset of the rows of a data frame.

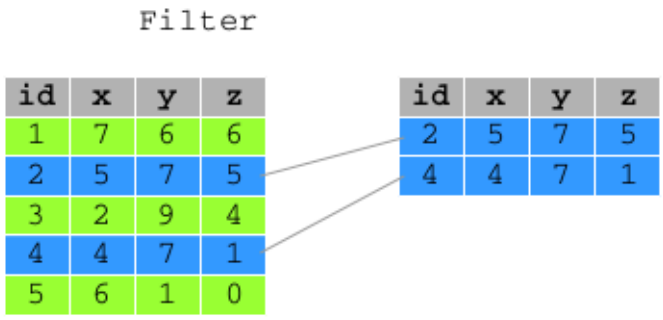


Figure 7.3:

The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame.

For example, you can select all calls made to students whit balance above 20,000:

```
filter(bank, job == "student", balance > 20000)
```

```
## # A tibble: 3 × 20
##    id  age    job marital education default balance housing  loan  contact  day  month  year
##  <int> <int>  <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1 31125   24 student  single secondary     no   23878     no    no cellular   18   feb  2009
## 2 39536   24 student  single secondary     no   23878     no    no cellular   26   may  2009
## 3 41923   27 student  single  tertiary     no   24025     no    no cellular   21   oct  2009
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

`filter()` allows you to give it any number of filtering conditions which are joined together with `&` and/or the other operators.

```
# Select all calls made to student of 18 years
bank %>% filter(age == 18 & job == "student")
```

```
## # A tibble: 12 × 20
##     id  age    job marital education default balance housing  loan   contact  day  month  year
##   <int> <int>  <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>   <fctr> <int> <fctr> <int>
## 1  40737   18 student  single  primary     no   1944     no    no telephone   10   aug  2009
## 2  40745   18 student  single  unknown     no    108     no    no  cellular   10   aug  2009
## 3  40888   18 student  single  primary     no    608     no    no  cellular   12   aug  2009
## 4  41223   18 student  single  unknown     no     35     no    no telephone   21   aug  2009
## 5  41253   18 student  single secondary     no      5     no    no  cellular   24   aug  2009
## 6  41274   18 student  single  unknown     no      3     no    no  cellular   25   aug  2009
## 7  41488   18 student  single  unknown     no    108     no    no  cellular    8   sep  2009
```

```
## 8  42147   18 student  single secondary     no    156     no    no  cellular    4   nov  2009
## 9  42275   18 student  single   primary     no    608     no    no  cellular   13   nov  2009
## 10 42955   18 student  single   unknown     no    108     no    no  cellular    9   feb  2010
## 11 43638   18 student  single   unknown     no    348     no    no  cellular    5   may  2010
## 12 44645   18 student  single   unknown     no    438     no    no  cellular    1   sep  2010
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

```
# Select all calls made to people of 18 or 95 years
bank %>% filter(age == 18 | age == 95)
```

```
## # A tibble: 14 × 20
##     id  age    job  marital education default balance housing  loan  contact  day  month  year
##   <int> <int>  <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>   <fctr> <int> <fctr> <int>
## 1  33700   95 retired divorced   primary     no   2282     no    no telephone   21   apr  2009
## 2  40737   18 student   single   primary     no   1944     no    no telephone   10   aug  2009
## 3  40745   18 student   single   unknown     no    108     no    no  cellular   10   aug  2009
## 4  40888   18 student   single   primary     no    608     no    no  cellular   12   aug  2009
## 5  41223   18 student   single   unknown     no     35     no    no telephone   21   aug  2009
## 6  41253   18 student   single secondary     no      5     no    no  cellular   24   aug  2009
## 7  41274   18 student   single   unknown     no      3     no    no  cellular   25   aug  2009
## 8  41488   18 student   single   unknown     no    108     no    no  cellular    8   sep  2009
## 9  41664   95 retired  married secondary     no      0     no    no telephone    1   oct  2009
## 10 42147   18 student   single secondary     no    156     no    no  cellular    4   nov  2009
## 11 42275   18 student   single   primary     no    608     no    no  cellular   13   nov  2009
## 12 42955   18 student   single   unknown     no    108     no    no  cellular    9   feb  2010
## 13 43638   18 student   single   unknown     no    348     no    no  cellular    5   may  2010
## 14 44645   18 student   single   unknown     no    438     no    no  cellular    1   sep  2010
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

`filter()` can be used also with `%in%` to establish conditions under which filter:

```
# Select all calls made to people of 18 or 95 years
bank %>% filter(age %in% c(18,95))
```

```
## # A tibble: 14 × 20
##     id  age    job  marital education default balance housing  loan  contact  day  month  year
##   <int> <int>  <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>   <fctr> <int> <fctr> <int>
## 1  33700   95 retired divorced   primary     no   2282     no    no telephone   21   apr  2009
## 2  40737   18 student   single   primary     no   1944     no    no telephone   10   aug  2009
## 3  40745   18 student   single   unknown     no    108     no    no  cellular   10   aug  2009
## 4  40888   18 student   single   primary     no    608     no    no  cellular   12   aug  2009
## 5  41223   18 student   single   unknown     no     35     no    no telephone   21   aug  2009
## 6  41253   18 student   single secondary     no      5     no    no  cellular   24   aug  2009
## 7  41274   18 student   single   unknown     no      3     no    no  cellular   25   aug  2009
## 8  41488   18 student   single   unknown     no    108     no    no  cellular    8   sep  2009
## 9  41664   95 retired  married secondary     no      0     no    no telephone    1   oct  2009
## 10 42147   18 student   single secondary     no    156     no    no  cellular    4   nov  2009
## 11 42275   18 student   single   primary     no    608     no    no  cellular   13   nov  2009
## 12 42955   18 student   single   unknown     no    108     no    no  cellular    9   feb  2010
## 13 43638   18 student   single   unknown     no    348     no    no  cellular    5   may  2010
## 14 44645   18 student   single   unknown     no    438     no    no  cellular    1   sep  2010
## # ... with 7 more variables: date <dttm>, duration <int>, campaign <int>, pdays <int>, previous <int>,
## #   poutcome <fctr>, y <fctr>
```

An other example is:

```
# Select all calls made to people whose job is admin. or technician
bank %>% filter(job %in% c("admin.","technician"))
```

```
## # A tibble: 12,768 × 20
##     id  age      job  marital education default balance housing  loan contact  day  month  year
##   <int> <int>    <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1     2   44 technician  single secondary     no     29    yes    no unknown    5   may  2008
## 2    10   43 technician  single secondary     no    593    yes    no unknown    5   may  2008
## 3    11   41    admin. divorced secondary     no    270    yes    no unknown    5   may  2008
## 4    12   29    admin.   single secondary     no    390    yes    no unknown    5   may  2008
## 5    13   53 technician  married secondary     no      6    yes    no unknown    5   may  2008
## 6    14   58 technician  married   unknown     no     71    yes    no unknown    5   may  2008
## 7    17   45    admin.   single   unknown     no     13    yes    no unknown    5   may  2008
## 8    26   44    admin.  married secondary     no   -372    yes    no unknown    5   may  2008
## 9    30   36 technician  single secondary     no    265    yes   yes unknown    5   may  2008
## 10   31   57 technician  married secondary     no    839     no   yes unknown    5   may  2008
## # ... with 12,758 more rows, and 7 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>
```

```
# Select all calls made to people whose job is admin. or technician
bank %>% filter(job == "admin." | job == "technician")
```

```
## # A tibble: 12,768 × 20
##     id  age      job  marital education default balance housing  loan contact  day  month  year
##   <int> <int>    <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1     2   44 technician  single secondary     no     29    yes    no unknown    5   may  2008
## 2    10   43 technician  single secondary     no    593    yes    no unknown    5   may  2008
## 3    11   41    admin. divorced secondary     no    270    yes    no unknown    5   may  2008
## 4    12   29    admin.   single secondary     no    390    yes    no unknown    5   may  2008
## 5    13   53 technician  married secondary     no      6    yes    no unknown    5   may  2008
## 6    14   58 technician  married   unknown     no     71    yes    no unknown    5   may  2008
## 7    17   45    admin.   single   unknown     no     13    yes    no unknown    5   may  2008
## 8    26   44    admin.  married secondary     no   -372    yes    no unknown    5   may  2008
## 9    30   36 technician  single secondary     no    265    yes   yes unknown    5   may  2008
## 10   31   57 technician  married secondary     no    839     no   yes unknown    5   may  2008
## # ... with 12,758 more rows, and 7 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>
```

7.2.3 arrange()

Function `arrange()` reorders a data frame by one or more variables. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

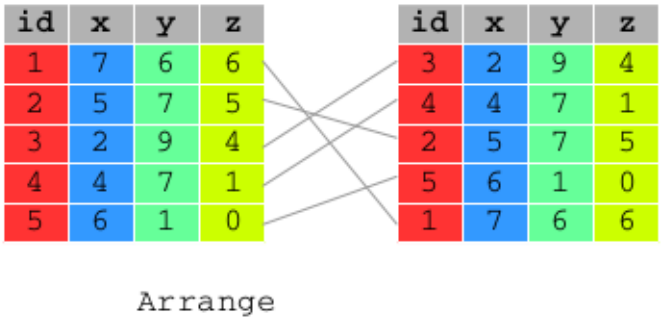


Figure 7.4:

The first argument is the name of the data frame, and the second and subsequent are the name of columns of that data frame you want to order by.
You may want to order the `bank` data frame by the balance of the account in ascending order:

```
arrange(bank, balance)
```

```
## # A tibble: 45,211 × 20
##      id   age         job  marital education default balance housing   loan  contact   day  month  year
##   <int> <int>       <fctr>   <fctr>   <fctr>  <fctr>   <int>  <fctr> <fctr>   <fctr> <int> <fctr> <int>
## 1  12910    26  blue-collar   single secondary     yes   -8019      no    yes cellular     7    jul  2008
## 2  15683    49   management  married  tertiary     yes   -6847      no    yes cellular    21    jul  2008
## 3  38737    60   management divorced  tertiary      no   -4057     yes     no cellular    18    may  2009
## 4   7414    43   management  married  tertiary     yes   -3372     yes     no unknown    29    may  2008
## 5   1897    57 self-employed  married  tertiary     yes   -3313     yes    yes unknown     9    may  2008
## 6  32714    39 self-employed  married  tertiary      no   -3058     yes    yes cellular    17    apr  2009
## 7  18574    40   technician  married  tertiary     yes   -2827     yes    yes cellular    31    jul  2008
## 8  31510    52   management  married  tertiary      no   -2712     yes    yes cellular     2    apr  2009
## 9  25120    49  blue-collar   single   primary     yes   -2604     yes     no cellular    18    nov  2008
## 10 14435    51   management divorced  tertiary      no   -2282     yes    yes cellular    14    jul  2008
## # ... with 45,201 more rows, and 7 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>
```

or in descending order by using function `desc()` within `arrange()`:

```
bank %>% arrange(desc(balance))
```

```
## # A tibble: 45,211 × 20
##      id   age         job  marital education default balance housing   loan   contact   day  month  year
##   <int> <int>       <fctr>   <fctr>   <fctr>  <fctr>   <int>  <fctr> <fctr>    <fctr> <int> <fctr> <int>
## 1  39990    51  management   single  tertiary      no  102127      no     no  cellular     3    jun  2009
## 2  26228    59  management  married  tertiary      no   98417      no     no telephone    20    nov  2008
```

```
## 3  42559   84    retired  married secondary    no  81204    no    no telephone   28   dec  2009
## 4  43394   84    retired  married secondary    no  81204    no    no telephone    1   apr  2010
## 5  41694   60    retired  married  primary    no  71188    no    no  cellular    6   oct  2009
## 6  19786   56 management divorced  tertiary    no  66721    no    no  cellular    8   aug  2008
## 7  21193   52 blue-collar  married  primary    no  66653    no    no  cellular   14   aug  2008
## 8  19421   59     admin.  married  unknown    no  64343    no    no  cellular    6   aug  2008
## 9  41375   32 entrepreneur  single  tertiary    no  59649    no    no  cellular    1   sep  2009
## 10 12927   56 blue-collar  married secondary    no  58932    no    no telephone    7   jul  2008
## # ... with 45,201 more rows, and 7 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>
```

You can order a data frame by one or more than one variables.

Ordering data frame bank by `age` first and descending `balance` afterward requires:

```
bank %>% arrange(age, desc(balance))
```

```
## # A tibble: 45,211 × 20
##     id  age    job marital education default balance housing  loan  contact  day month  year
##   <int> <int>  <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>   <fctr> <int> <fctr> <int>
## 1  40737   18 student  single  primary     no   1944     no    no telephone   10   aug  2009
## 2  40888   18 student  single  primary     no    608     no    no  cellular   12   aug  2009
## 3  42275   18 student  single  primary     no    608     no    no  cellular   13   nov  2009
## 4  44645   18 student  single  unknown     no    438     no    no  cellular    1   sep  2010
## 5  43638   18 student  single  unknown     no    348     no    no  cellular    5   may  2010
## 6  42147   18 student  single secondary     no    156     no    no  cellular    4   nov  2009
## 7  40745   18 student  single  unknown     no    108     no    no  cellular   10   aug  2009
## 8  41488   18 student  single  unknown     no    108     no    no  cellular    8   sep  2009
## 9  42955   18 student  single  unknown     no    108     no    no  cellular    9   feb  2010
## 10 41223   18 student  single  unknown     no     35     no    no telephone   21   aug  2009
## # ... with 45,201 more rows, and 7 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>
```

7.2.4 mutate()

As well as selecting from the set of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

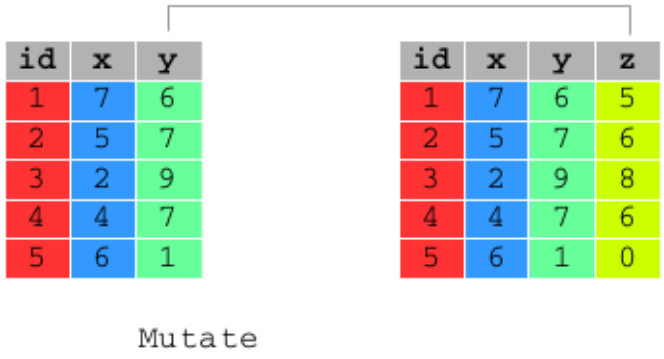


Figure 7.5:

The first argument is the name of the data frame, and the second and subsequent are expressions for creating new columns to add to that data frame or for modifying the existing ones:

```
df <- data.frame(x = 1:3, y = 3:1)

mutate(df, x1 = x+1)
```

```
##   x y x1
## 1 1 3  2
## 2 2 2  3
## 3 3 1  4
```

```
mutate(df, x = x+1)
```

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

```
mutate(df, x = x+1, y = x+1)
```

```
##   x y
## 1 2 3
## 2 3 4
## 3 4 5
```

```
mutate(df, x1 = x+1, y1 = x1+1)
```

```
##   x y x1 y1
## 1 1 3  2  3
## 2 2 2  3  4
## 3 3 1  4  5
```

```
mutate(df, y1 = x+1, x1 = x+1)
```

```
##   x y y1 x1
## 1 1 3  2  2
## 2 2 2  3  3
## 3 3 1  4  4
```

```
mutate(df, xx = x)
```

```
##   x y xx
## 1 1 3  1
## 2 2 2  2
## 3 3 1  3
```

```
# generate a variable indicating the total number of times each person has been contacted
# during this campaign and during the previous ones
mutate(bank, contacts_n = campaign + previous)
```

```
## # A tibble: 45,211 × 21
##     id  age       job marital education default balance housing  loan contact  day month  year
##   <int> <int>     <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1     1   58  management married  tertiary      no   2143     yes     no unknown     5   may  2008
## 2     2   44  technician  single secondary      no     29     yes     no unknown     5   may  2008
## 3     3   33 entrepreneur married secondary      no      2     yes    yes unknown     5   may  2008
## 4     4   47 blue-collar married   unknown      no   1506     yes     no unknown     5   may  2008
## 5     5   33     unknown  single   unknown      no      1      no     no unknown     5   may  2008
## 6     6   35  management married  tertiary      no    231     yes     no unknown     5   may  2008
## 7     7   28  management  single  tertiary      no    447     yes    yes unknown     5   may  2008
## 8     8   42 entrepreneur divorced  tertiary     yes      2     yes     no unknown     5   may  2008
## 9     9   58     retired married   primary      no    121     yes     no unknown     5   may  2008
## 10   10   43  technician  single secondary      no    593     yes     no unknown     5   may  2008
## # ... with 45,201 more rows, and 8 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>, contacts_n <int>
```

`mutate()` allows you to refer to columns that you just created:

```
# generate two variable: one indicating the year of birth and one the year of birth without century
bank %>% mutate(year_of_birth = year - age, year_of_birth_no_century = year_of_birth - 1900)
```

```
## # A tibble: 45,211 × 22
##     id  age       job marital education default balance housing  loan contact  day month  year
##   <int> <int>     <fctr>  <fctr>   <fctr>  <fctr>  <int>  <fctr> <fctr>  <fctr> <int> <fctr> <int>
## 1     1   58  management married  tertiary      no   2143     yes     no unknown     5   may  2008
## 2     2   44  technician  single secondary      no     29     yes     no unknown     5   may  2008
## 3     3   33 entrepreneur married secondary      no      2     yes    yes unknown     5   may  2008
## 4     4   47 blue-collar married   unknown      no   1506     yes     no unknown     5   may  2008
## 5     5   33     unknown  single   unknown      no      1      no     no unknown     5   may  2008
## 6     6   35  management married  tertiary      no    231     yes     no unknown     5   may  2008
## 7     7   28  management  single  tertiary      no    447     yes    yes unknown     5   may  2008
## 8     8   42 entrepreneur divorced  tertiary     yes      2     yes     no unknown     5   may  2008
## 9     9   58     retired married   primary      no    121     yes     no unknown     5   may  2008
## 10   10   43  technician  single secondary      no    593     yes     no unknown     5   may  2008
## # ... with 45,201 more rows, and 9 more variables: date <dttm>, duration <int>, campaign <int>,
## #   pdays <int>, previous <int>, poutcome <fctr>, y <fctr>, year_of_birth <int>,
## #   year_of_birth_no_century <dbl>
```

7.2.5 summarise()

The last verb is `summarise()`, which collapses a data frame to a single row.

The first argument is the name of the data frame, and the second and subsequent are summarising expressions.

```
# Compute the mean of balance variable of bank data frame
bank %>% summarise(mean_balance = mean(balance, na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##   mean_balance
##          <dbl>
## 1     1362.272
```

```
# Compute the minimum and the maximum value of balance of bank data frame
bank %>% summarise(max_balance = max(balance, na.rm = TRUE), min_balance = min(balance, na.rm = TRUE))
```

```
## # A tibble: 1 × 2
##   max_balance min_balance
##         <int>       <int>
## 1      102127       -8019
```

7.3 Group Data and Chain Verbs

```
require(dplyr)
require(qdata)
data(bank)
```

7.3.1 group_by()

The verb functions are useful, but they become really powerful when you combine them with the idea of "group by", repeating the operation individually on groups of observations within the dataset. In `dplyr`, you use the `group_by()` function to describe how to break a dataset down into groups of rows. You can then use the resulting object in the verbs functions; they'll automatically work "by group" when the input is a grouped. Let us see some examples (pay attention to objects class).

```
# Example data frame
df <- data.frame(x = 1:6, f = rep(1:2, each = 3))

# Grouped data frame
dff <- group_by(df, f)
dff
```

```
## Source: local data frame [6 x 2]
## Groups: f [2]
##
##       x     f
##   <int> <int>
## 1     1     1
## 2     2     1
## 3     3     1
## 4     4     2
## 5     5     2
## 6     6     2
```

```
class(dff)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

```
# Use dff (grouped data frame) as .data argument value in mutate()
dffn <- mutate(.data = dff, n = n())
dffn
```

```
## Source: local data frame [6 x 3]
## Groups: f [2]
##
##       x     f     n
##   <int> <int> <int>
## 1     1     1     3
## 2     2     1     3
## 3     3     1     3
## 4     4     2     3
## 5     5     2     3
## 6     6     2     3
```

```
class(dffn)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

```
# Use dff (grouped data frame) as .data argument value in arrange()
dffa <- arrange(.data = dff, desc(x))
dffa
```

```
## Source: local data frame [6 x 2]
## Groups: f [2]
##
##       x     f
##   <int> <int>
## 1     6     2
## 2     5     2
## 3     4     2
## 4     3     1
## 5     2     1
## 6     1     1
```

```
class(dffa)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

```
# Use dff (grouped data frame) as data argument value in summarise()
dfg <- summarise(.data = dff, x_avg = mean(x))
dfg
```

```
## # A tibble: 2 × 2
##       f x_avg
##   <int> <dbl>
## 1     1     2
## 2     2     5
```

```
class(dfg)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

In the following example, you split the complete dataset into years and then summarise each year by counting the number of phone calls (`count = n()`) and computing the average duration call (`dist = mean(duration, na.rm = TRUE)`) and balance (`delay = mean(balance, na.rm = TRUE)`):

```
# Split the complete dataset (bank) into years (group the df)
by_year <- group_by(bank, year)
# Summarise each year applying summarise() verb to the grouped df (by_year)
summarise(by_year,
          count = n(),
          mean_duration = mean(duration, na.rm = TRUE),
          mean_balance = mean(balance, na.rm = TRUE))
```

```
## # A tibble: 3 × 4
##    year count mean_duration mean_balance
##   <int> <int>         <dbl>        <dbl>
## 1  2008 27729      252.1936     1315.211
## 2  2009 14862      262.9644     1362.140
## 3  2010  2620      294.1065     1861.092
```

Then, you search for the number of days covered per year. You report also the number of phone calls per year:

```
summarise(by_year,
          days = n_distinct(date),
```

```
          count = n())
```

```
## # A tibble: 3 × 3
##    year  days count
##   <int> <int> <int>
## 1  2008   122 27729
## 2  2009   212 14862
## 3  2010   227  2620
```

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(bank, year, month, day)
groups(daily)
```

```
## [[1]]
## year
##
## [[2]]
## month
##
## [[3]]
## day
```

```
per_day <- summarise(daily, calls = n())
groups(per_day)
```

```
## [[1]]
## year
##
## [[2]]
## month
```

```
per_month <- summarise(per_day, calls = sum(calls))
groups(per_month)
```

```
## [[1]]
## year
```

```
per_year <- summarise(per_month, calls = sum(calls))
groups(per_year)
```

```
## NULL
```

```
df <- data.frame(year = rep(c(2010, 2011, 2012), each = 3),
                 month = rep(1:3, each = 3),
                 day = rep(20:22, 3),
                 x = 1:9)

df
```

```
##   year month day x
## 1 2010     1  20 1
## 2 2010     1  21 2
## 3 2010     1  22 3
## 4 2011     2  20 4
## 5 2011     2  21 5
## 6 2011     2  22 6
```

```
## 7 2012     3  20 7
## 8 2012     3  21 8
## 9 2012     3  22 9
```

```
df1 <- df %>% group_by(year, month, day)

groups(df1)
```

```
## [[1]]
## year
##
## [[2]]
## month
##
## [[3]]
## day
```

```
df2 <-  df1 %>%
  summarise(x_avg = mean(x), n = n())

df2
```

```
## Source: local data frame [9 x 5]
## Groups: year, month [?]
##
##    year month   day x_avg     n
##   <dbl> <int> <int> <dbl> <int>
## 1  2010     1    20     1     1
## 2  2010     1    21     2     1
## 3  2010     1    22     3     1
## 4  2011     2    20     4     1
## 5  2011     2    21     5     1
## 6  2011     2    22     6     1
## 7  2012     3    20     7     1
## 8  2012     3    21     8     1
## 9  2012     3    22     9     1
```

```
groups(df2)
```

```
## [[1]]
## year
##
## [[2]]
## month
```

```
summarise(df2, n())
```

```
## Source: local data frame [3 x 3]
## Groups: year [?]
##
##    year month `n()`
##   <dbl> <int> <int>
## 1  2010     1     3
## 2  2011     2     3
## 3  2012     3     3
```

```
ungroup(df2) %>% summarise(n())
```

```
## # A tibble: 1 × 1
##   `n()`
##   <int>
## 1     9
```

7.3.2 Chain Together Multiple Operations

The `dplyr` API (Application Program Interface) is functional in the sense that function calls don't have side-effects, and you must always save their results. This doesn't lead to particularly elegant code if you want to do many operations at once. You either have to do it step-by-step:

```
a1 <- group_by(bank, date)
a2 <- select(a1, age, balance)
```

```
## Adding missing grouping variables: `date`
```

```
a3 <- summarise(a2,
                mean_age = mean(age, na.rm = TRUE),
                mean_balance = mean(balance, na.rm = TRUE)
                )
(a4 <- filter(a3, mean_age < 40 & mean_balance > 5000))
```

```
## # A tibble: 5 × 3
##         date mean_age mean_balance
##       <dttm>    <dbl>        <dbl>
## 1 2008-10-18   37.500      5502.75
## 2 2008-12-27   28.000      6100.00
## 3 2009-03-20   28.000      5916.00
## 4 2009-10-02   39.875      5088.00
## 5 2009-12-31   32.000     14533.00
```

Or if you don't want to save the intermediate results, you need to wrap the function calls inside each other:

```
filter(
  summarise(
    select(
      group_by(bank, date), age, balance
    ),
    mean_age = mean(age, na.rm = TRUE),
    mean_balance = mean(balance, na.rm = TRUE)
  ),
  mean_age < 40 & mean_balance > 5000
)
```

```
## Adding missing grouping variables: `date`
```

```
## # A tibble: 5 × 3
##         date mean_age mean_balance
##       <dttm>    <dbl>        <dbl>
## 1 2008-10-18   37.500      5502.75
## 2 2008-12-27   28.000      6100.00
## 3 2009-03-20   28.000      5916.00
## 4 2009-10-02   39.875      5088.00
## 5 2009-12-31   32.000     14533.00
```

This is difficult to read because the order of the operations is from inside to out, and the arguments are a long way away from the function. To get around this problem, use the `%>%` operator.

```
bank %>%
  group_by(date) %>%
  select(age, balance) %>%
  summarise(
    mean_age = mean(age, na.rm = TRUE),
    mean_balance = mean(balance, na.rm = TRUE)
  ) %>%
  filter(mean_age < 40 & mean_balance > 5000)
```

```
## Adding missing grouping variables: `date`
```

```
## # A tibble: 5 × 3
##         date mean_age mean_balance
##       <dttm>    <dbl>        <dbl>
## 1 2008-10-18   37.500      5502.75
## 2 2008-12-27   28.000      6100.00
## 3 2009-03-20   28.000      5916.00
## 4 2009-10-02   39.875      5088.00
## 5 2009-12-31   32.000     14533.00
```

7.4 dplyr verbs for combining data: join

```
require(dplyr)
```

Very often you will have to deal with many tables that contribute to the analysis you are performing and you need flexible tools to combine them. Supposing that the two tables are already in a tidy form: the rows are observations and the columns are variables, `dplyr` provides **mutating joins**, which add new variables to one table from matching rows in another.

There are four types of mutating join, which differ in their behaviour when a match is not found.

- `inner_join(x, y)`
- `left_join(x, y)`
- `right_join(x, y)`
- `outer_join(x, y)`

All these verbs work similarly:

- the first two arguments, `x` and `y`, provide the tables to combine
- the output is always a new table with the same type as `x`

For the next examples we will consider these two small data frames:

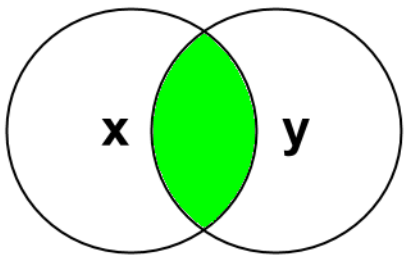
```
df1 <- data.frame(id = 1:4, x1 = letters[1:4])
df1
```

```
##   id x1
## 1  1  a
## 2  2  b
## 3  3  c
## 4  4  d
```

```
df2 <- data.frame(id = 3:5, x2 = letters[3:5])
df2
```

```
##   id x2
## 1  3  c
## 2  4  d
## 3  5  e
```

7.4.1 inner_join(x, y)



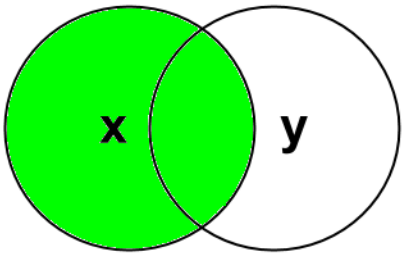
`inner_join(x, y)` only includes observations that match in both x and y:

```
inner_join(df1, df2)
```

```
## Joining, by = "id"
```

```
##   id x1 x2
## 1  3  c  c
## 2  4  d  d
```

7.4.2 left_join(x, y)



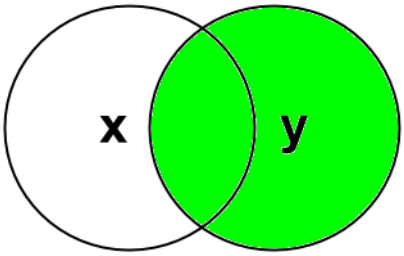
`left_join(x, y)` includes all observations in `x`, regardless of whether they match or not. This is the most commonly used join because it ensures that you don't lose observations from your primary table:

```
left_join(df1, df2)
```

```
## Joining, by = "id"
```

```
##   id x1   x2
## 1  1  a <NA>
## 2  2  b <NA>
## 3  3  c    c
## 4  4  d    d
```

7.4.3 right_join(x, y)



`right_join(x, y)` includes all observations in `y`. It's equivalent to `left_join(y, x)`, but the columns will be ordered differently:

```
right_join(df1, df2)
```

```
## Joining, by = "id"
```

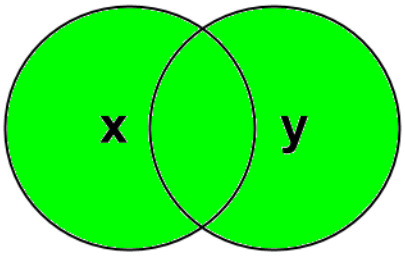
```
##   id   x1 x2
## 1  3    c  c
## 2  4    d  d
## 3  5 <NA>  e
```

```
left_join(df2, df1)
```

```
## Joining, by = "id"
```

```
##   id x2   x1
## 1  3  c    c
## 2  4  d    d
## 3  5  e <NA>
```

7.4.4 full_join()



`full_join()` includes all observations from `x` and `y`:

```
full_join(df1, df2)
```

```
## Joining, by = "id"
```

```
##   id   x1   x2
## 1  1    a <NA>
## 2  2    b <NA>
## 3  3    c    c
## 4  4    d    d
## 5  5 <NA>    e
```

The left, right and full joins are collectively know as outer joins. When a row doesn't match in an outer join, the new variables are filled in with missing values.

7.5 dplyr with backend databases

```
require(dplyr)
require(ggplot2)
require(DBI)
require(RSQLite)
```

7.5.1 Getting Started with Large Data

To experimement with large files you may want to use data from http://stat-computing.org/dataexpo/2009/the-data.html.

The data consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. This is a large dataset: there are nearly 120 million records in total, and takes up 1.6 gigabytes of space compressed and 12Gb when uncompressed. Only four of these files will be considered in the next example for a 29 millions records data frame.

This is clearly a case for SQLite: let us load the `RSQLite` package.

```
require(RSQLite)
```

As these files might have different encodings you may need to define a simple function that returns the encoding of the file. Note that this function uses unix system commands and may not work under windows based computers.

```
charset <- function (file){
  info <- system(paste("file -i" ,file ), intern = TRUE)
  info_split <- strsplit(info, split = "charset=")[[1]][2]
  info_split
}
```

You then create an empty SQLite database with a little trick that ensures the database is removed in case it already exists:

```
path <- "./"
db <- "ontime.sqlite"
path_db <- paste(path, db, sep = "/")
db_exists <- list.files(path, pattern = db)
if(length(db_exists) != 0) system(paste("rm", path_db))
con <- dbConnect(RSQLite::SQLite(), path_db)
```

Now it's time to get the list of files you want to load:

```
files <- list.files("./../data", pattern = ".csv", full.names = TRUE)
```

and by using a for loop you can load all the files into the newly created data base:

```
for (i in 1:length(files)){
  head <- ifelse (i == 1, TRUE, FALSE)
  skip <- ifelse (i == 1, 0, 1)
  append <- ifelse (i == 1, FALSE, TRUE)
  this_file <- files[i]
  encoding <- charset(this_file)
  df <- read.table(this_file, sep = ",", head = head, encoding = encoding, skip = skip, nrows = 100)
  dbWriteTable(conn = con, name = "ontime", value = df ,append = append)
  cat(date() , this_file, "loaded", "\n")
```

```
  rm(df)
}
```

finally disconnect from the database:

```
dbDisconnect(con)
```

You can now start using the data base with dplyr interface:

```
con_dplyr <- src_sqlite(path_db)
ontime <- tbl(con_dplyr, "ontime")
class(ontime)
dim(ontime)
```

When working with databases, `dplyr` tries to be as lazy as possible. It's lazy in two ways:

- It never pulls data back to `R` unless you explicitly ask for it;
- It delays doing any work until the last possible minute, collecting together everything you want to do then sending that to the database in one step.

For example, take the following code:

```
ontime_stat <- ontime %>% group_by(Year, Month) %>%
   summarise(avg = mean(DepDelay), min = min(DepDelay), max = max(DepDelay))
```

Suprisingly, this sequence of operations never actually touches the database. It's not until you ask for the data (e.g. by printing `ontime_stat`) that `dplyr` generates the SQL and requests the results from the database, and even then it only pulls down 10 rows:

```
ontime_stat
class(ontime_stat)
```

To pull down all the results use `collect()`, which returns a `tbl` class object:

```
ontime_dep_delay <- ontime %>%
  select(year = Year, dep_delay = DepDelay, arr_delay = ArrDelay) %>%
  filter(dep_delay > 0) %>%
  collect()
```

Once data are collected you can use it within `R`:

```
pl <- ggplot(ontime_dep_delay , aes(dep_delay, arr_delay))
pl <- pl + stat_binhex(bins = 30) + facet_wrap(~year, ncol = 2)
print(pl)
```

Chapter 8

Reshaping data with tidyr

```
require(dplyr)
require(tidyr)
require(qdata)
require(ggplot2)
```

8.1 Introduction

As a data scientist you collect data from different sources and, very often you will have very little control on the original structure of this data.

Such a variety in data structure implies long time spent at manipulating your data prior any analysis or visualization step.

On the other hand, you may think about a well defined and standard way storing your data so that data are ready to be pushed into any next step.

This *standard* format has been defined as **tidy**.

At the same time, analysis of visualization procedures require to be ready to accept as input *tidy* data structure. This is not always the case but it is becoming more and more a standard in the `R` community.

As a simple example let's consider the `people` data frame from package `qdata`:

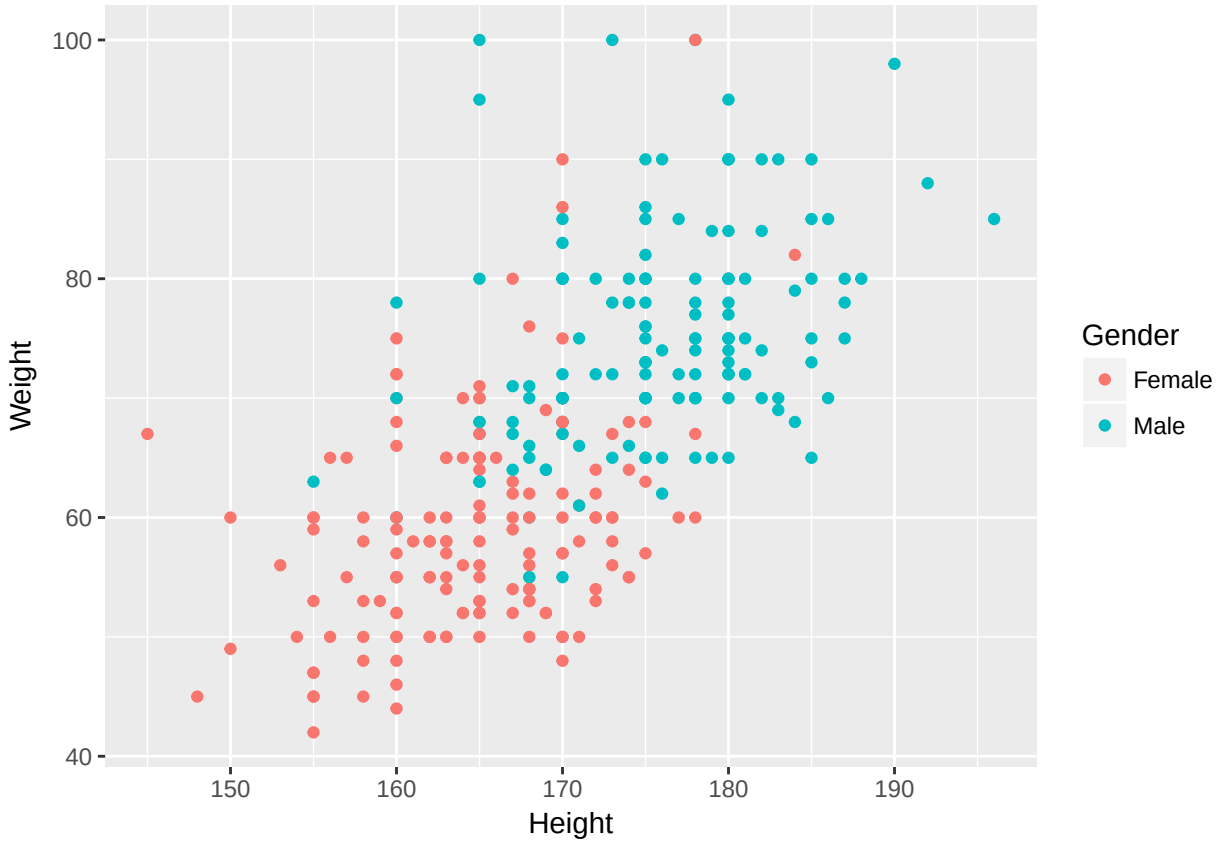
```
data(people)
head(people)
```

```
## # A tibble: 6 × 4
##   Gender   Area Weight Height
##   <fctr> <fctr>  <int>  <int>
## 1 Female  Isole     54    168
## 2 Female   Nord     61    171
## 3   Male    Sud     68    170
## 4 Female   Nord     52    164
## 5   Male   Nord     75    181
## 6   Male   Nord     77    178
```

This data frame can be considered as a tidy data frame.

As a result, plotting `Weight` as a function of `Height` with different colours for `Gender`, using package `ggplot2`, is very straightforward as `ggplot2` is a set of visualization tools ready for tidy data

```
ggplot(people, aes(x = Height, y = Weight, colour = Gender)) + geom_point()
```

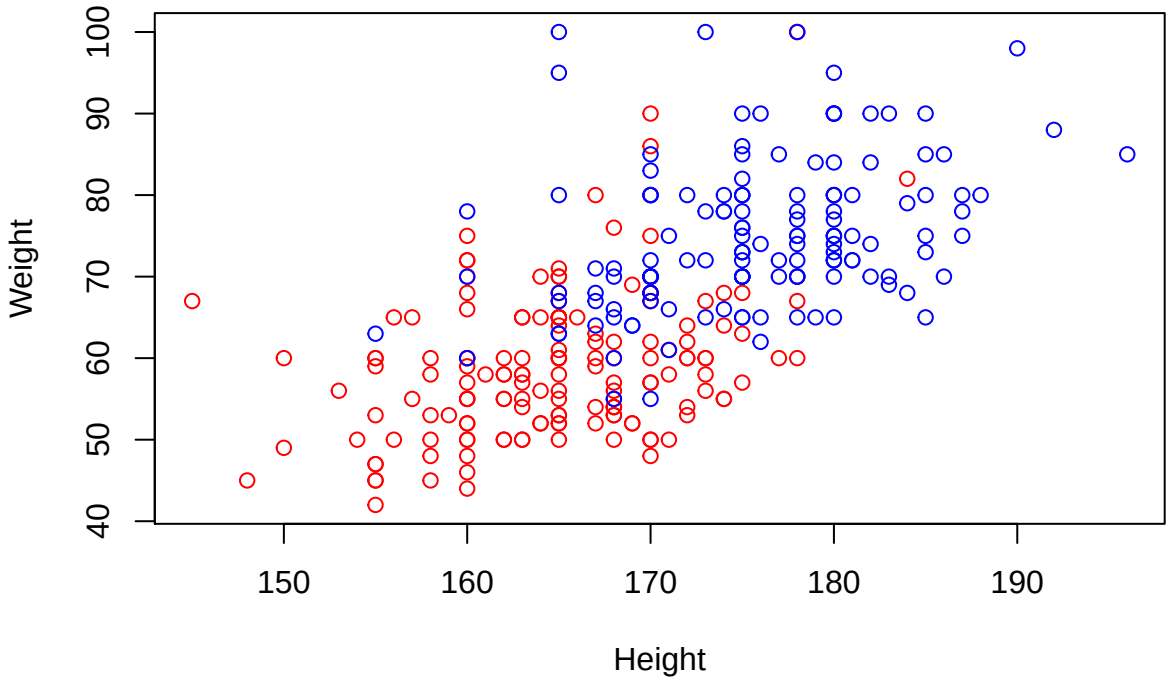


A similar result, with the standard R plotting tools, may require first a data transformation effort:

```
female <- people[people$Gender == 'Female',]
male <- people[people$Gender == 'Male',]
```

followed by the plotting instruction:

```
with(people, plot(Height, Weight, type = "n"))
with(female, points(Height, Weight, col = "red"))
with(male, points(Height, Weight, col = "blue"))
```



As a conclusion, as *Hadley Wickam* writes: *Tidy datasets and tidy tools work hand in hand to make data analysis easier, allowing you to focus on the interesting domain problem, not on the uninteresting logistics of data.*

8.2 Defining Tidy Data

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:

- each variable forms a column;
- each observation forms a row;
- each type of observational unit forms a table.

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. 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.

8.3 Tidying Messy Datasets

Real datasets can, and often do, violate the three precepts of tidy data in almost every way imaginable. These are the five most common problems with messy datasets:

1. column headers are values, not variable names;
2. multiple variables are stored in one column;
3. variables are stored in both rows and columns;
4. multiple types of observational units are stored in the same table;

5. a single observational unit is stored in multiple tables.

8.4 Tools

Surprisingly, most messy datasets, including types of messiness not explicitly described above, can be tidied with a small set of tools:

- `gather()` and `spread()`

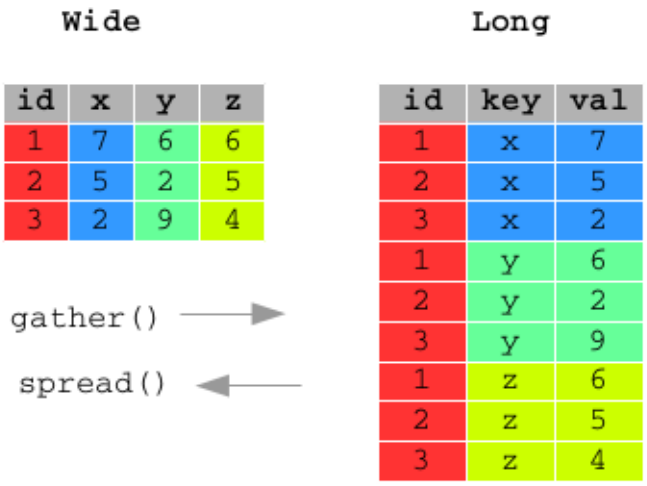


Figure 8.1:

- `separate()` and `unite()`

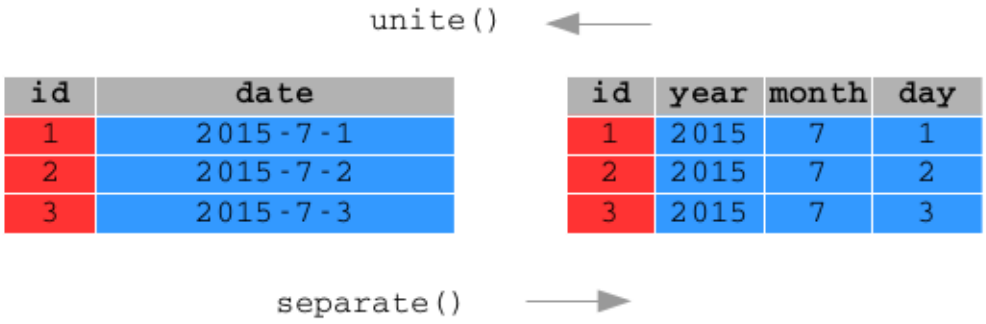


Figure 8.2:

8.4.1 1. column headers are values, not variable names

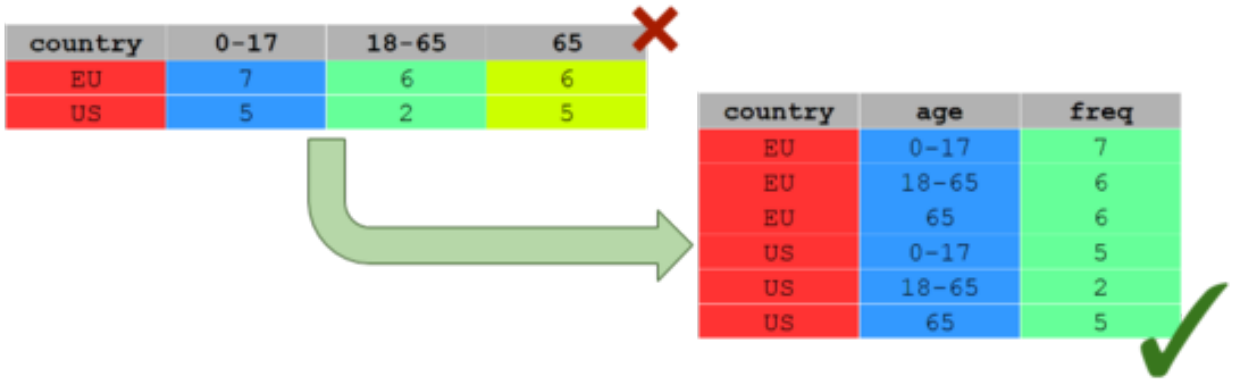


Figure 8.3:

```
df <- setNames(data.frame(c("EU", "US"), c(7,5), c(6,2), c(6,5)),
               c("country","0-17" , "18-65", "66+"))

df
```

```
##   country 0-17 18-65 66+
## 1      EU    7     6   6
## 2      US    5     2   5
```

This data frame can be classified as *messy* because variables: `0-17`, `18-65`, `66+` are values of a variable, say `age`, not column headers.

The unusual definition of column header by using function `setNames()` is due to the non standard naming convention used in this case. A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number.

The structure of `df` does not make clear:

- what column headers refer to
- the meaning of each observational unit

A simple instruction to tidy this data is:

```
gather(df, key = "age", value = "freq" , 2:4)
```

```
##   country   age freq
## 1      EU  0-17    7
## 2      US  0-17    5
## 3      EU 18-65    6
## 4      US 18-65    2
## 5      EU   66+    6
## 6      US   66+    5
```

where `key` and `value` are the names for the newly created key-value pair and `2:4` are the columns to stack. Again, instead of passing the columns to stack by position it would be better to use column names but, the unusual definition of column names does not allow this.

```
df <- data.frame(country = c("EU", "US"),
                 age_0_17 = c(7,5),
                 age_18_65 = c(6,2),
```

```
                age_65 = c(6,5))

df <- gather(df, key = "age", value = "freq" , age_0_17, age_18_65, age_65)

subx <- function(x, pattern, replacement , ...) sub(pattern, replacement, x, ...)

df %>% mutate(age = subx(age, "age_", ""))
```

```
##   country   age freq
## 1      EU  0_17    7
## 2      US  0_17    5
## 3      EU 18_65    6
## 4      US 18_65    2
## 5      EU    65    6
## 6      US    65    5
```

8.4.2 2. multiple variables are stored in one column

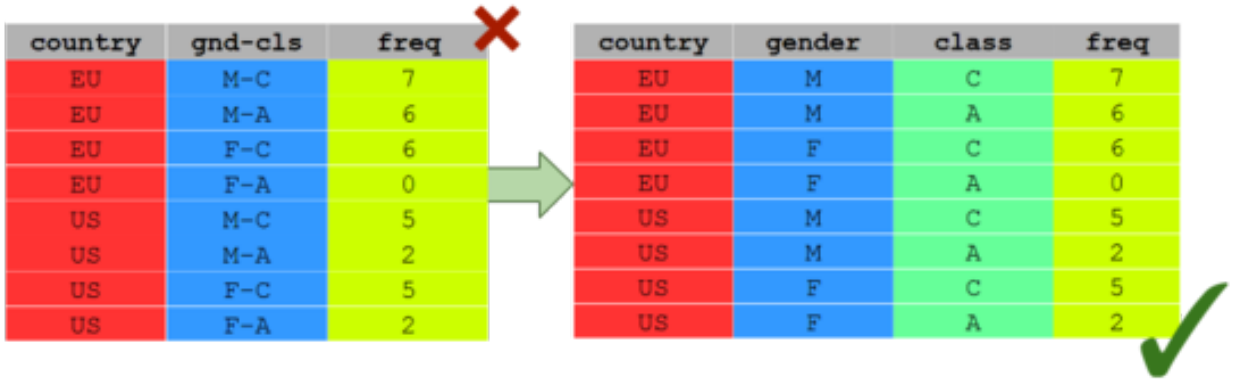


Figure 8.4:

```
df <- data.frame(
  country = c("EU", "EU", "EU", "EU", "US", "US", "US", "US"),
  gnd_cls = c("M-C", "M-A", "F-C", "F-A", "M-C", "M-A", "F-C", "F-A"),
  freq = c(7,6,6,0,5,2,5,2))

df
```

```
##   country gnd_cls freq
## 1      EU     M-C    7
## 2      EU     M-A    6
## 3      EU     F-C    6
## 4      EU     F-A    0
## 5      US     M-C    5
## 6      US     M-A    2
## 7      US     F-C    5
## 8      US     F-A    2
```

This data frame can be classified as messy because variable `gnd_cls` stores information about two variables: `gender` and `class`.

Function `separate()`, whenever you can find a way to do it, separate the content of a column into two columns:

```
separate(df, col = "gnd_cls", into = c("gender", "class"), sep = "-")
```

```
##   country gender class freq
## 1      EU      M     C    7
## 2      EU      M     A    6
## 3      EU      F     C    6
## 4      EU      F     A    0
## 5      US      M     C    5
## 6      US      M     A    2
## 7      US      F     C    5
## 8      US      F     A    2
```

8.4.3 3. variables are stored in both rows and columns;

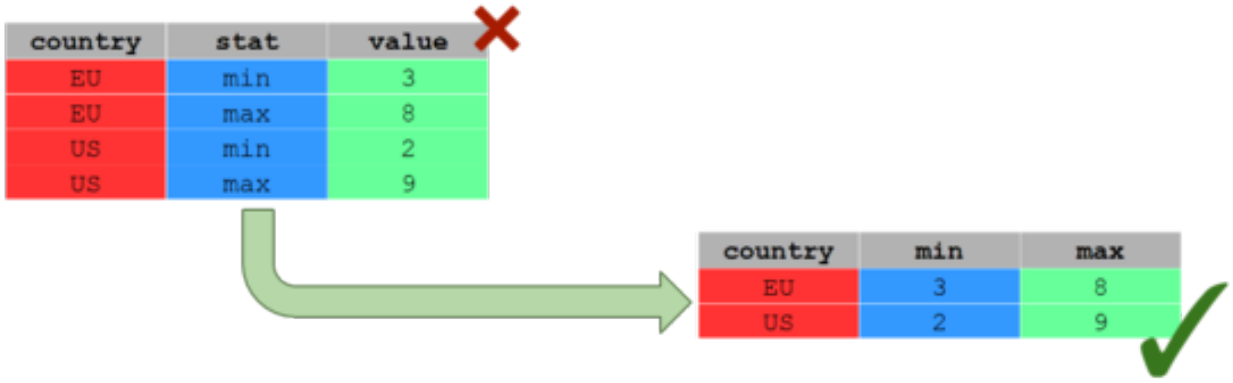


Figure 8.5:

```
df <- data.frame(country = c("EU", "EU", "US", "US"),
                 stat = c("min", "max", "min", "max"),
                 value = c(3, 8 , 2, 9)
                 )
df
```

```
##   country stat value
## 1      EU  min     3
## 2      EU  max     8
## 3      US  min     2
## 4      US  max     9
```

This data frame is *messy* because observational units of variable `stat` are variables by themselves.

Function `spread()` helps to structure this data in a tidy form:

```
spread(df, key = stat, value = value)
```

```
##   country max min
## 1      EU   8   3
## 2      US   9   2
```

8.4.4 4. multiple types of observational units are stored in the same table

Figure 8.6:

```
df <- data.frame(
  country =  c("EU", "EU", "EU", "US", "US", "EU", "EU", "EU", "US", "US"),
  state = c("UK", "FR", "CH", "WA", "CA",   "UK",   "FR",   "CH",   "WA",   "CA"),
  km3 = c(244820, 643801, 41290,    184665, 403933, 244820, 643801, 41290, 184665, 403933),
  year = c(2015,    2015,   2015,   2016,   2016,   2015,   2015,   2015,   2016,   2016),
  event = c(0, 4,   5, 6, 3, 4, 5, 3,   2, 1))

df
```

```
##    country state    km3 year event
## 1       EU    UK 244820 2015     0
```

```
## 2       EU    FR 643801 2015     4
## 3       EU    CH  41290 2015     5
## 4       US    WA 184665 2016     6
## 5       US    CA 403933 2016     3
## 6       EU    UK 244820 2015     4
## 7       EU    FR 643801 2015     5
## 8       EU    CH  41290 2015     3
## 9       US    WA 184665 2016     2
## 10      US    CA 403933 2016     1
```

This data frame stores two types of observational units in the same table. Variables `country`, `state` and `km3` refer to the first type of observational unit. The number of events: `event` per year: `year` by `state` refer to a second type of observational unit.

You can tidy this data by splitting the original data frame into two data frame: `df1` and `df2`.

You first define `df1` by selecting the distinct values of the type one columns:

```
df1 <-  df %>%
  select(country, state, km3) %>%
  distinct()
df1
```

```
##   country state    km3
## 1      EU    UK 244820
## 2      EU    FR 643801
## 3      EU    CH  41290
## 4      US    WA 184665
## 5      US    CA 403933
```

After that you define `df2` as:

```
df2 <- df %>%  select(state, year, event)
```

8.4.5 5. a single observational unit is stored in multiple tables.

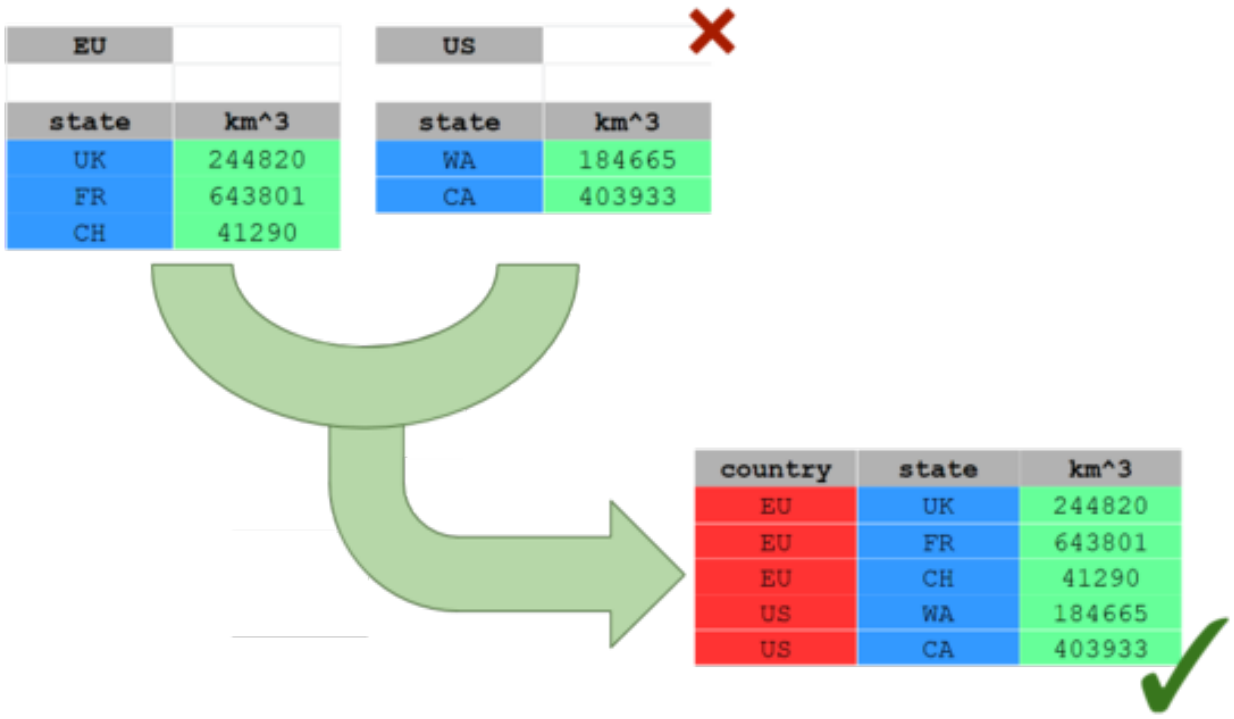


Figure 8.7:

```
EU <- data.frame(state = c("UK", "FR", "CH"), km3 = c(244820, 643801, 41290))
US <- data.frame(state = c("WA", "CA"), km3 = c(184664, 403933))

EU
```

```
##   state    km3
## 1    UK 244820
## 2    FR 643801
## 3    CH  41290
```

```
US
```

```
##   state    km3
## 1    WA 184664
## 2    CA 403933
```

This case often happens when you are loading data from different data sources.

The name of the data frame, in this case, pays a crucial role as it has to be stored into a variable:

Tidying this data requires more than one step:

First save the data frame name into a variable name:

```
EU <- EU %>% mutate(country = "EU")
US <- US %>% mutate(country = "US")
```

the combine by rows the two data frames

```
all <- Reduce(rbind , list(EU, US))
all
```

```
##   state    km3 country
## 1    UK 244820      EU
## 2    FR 643801      EU
## 3    CH  41290      EU
## 4    WA 184664      US
## 5    CA 403933      US
```

8.5 Example: Phone Time

In this example about some measurements of how much time people spend on their phones, measured at two locations (work and home), at two times. Each person has been randomly assigned to either treatment or control.

```
messy <- data.frame(
  id = 1:4,
  trt = rep(c('control', 'treatment'), each = 2),
  work.T1 = c(7, 6, 4, 3),
  home.T1 = c(5, 4, 1, 3),
  work.T2 = c(22, 27, 20, 16),
  home.T2 = c(11, 7, 10, 6)
)
messy
```

```
##   id       trt work.T1 home.T1 work.T2 home.T2
## 1  1   control       7       5      22      11
## 2  2   control       6       4      27       7
## 3  3 treatment       4       1      20      10
## 4  4 treatment       3       3      16       6
```

To tidy this data, before using `gather()` it is necessary to turn columns `work.T1`, `home.T1`, `work.T2` and `home.T2` into a key-value pair of key and time.

```
tidier <- gather(messy, key, time, -id, -trt)
tidier
```

```
##    id       trt     key time
## 1   1   control work.T1    7
## 2   2   control work.T1    6
## 3   3 treatment work.T1    4
## 4   4 treatment work.T1    3
## 5   1   control home.T1    5
## 6   2   control home.T1    4
## 7   3 treatment home.T1    1
## 8   4 treatment home.T1    3
## 9   1   control work.T2   22
## 10  2   control work.T2   27
## 11  3 treatment work.T2   20
## 12  4 treatment work.T2   16
## 13  1   control home.T2   11
## 14  2   control home.T2    7
## 15  3 treatment home.T2   10
```

```
## 16  4 treatment home.T2    6
```

Next `separate()` splits the key into `location` and `time`, using a regular expression to describe the character that separates them.

```
tidy <- separate(tidier, col = key, into = c("location", "time"), sep = "\\.")
tidy
```

```
##    id       trt location time time
## 1   1   control     work   T1    7
## 2   2   control     work   T1    6
## 3   3 treatment     work   T1    4
## 4   4 treatment     work   T1    3
## 5   1   control     home   T1    5
## 6   2   control     home   T1    4
## 7   3 treatment     home   T1    1
## 8   4 treatment     home   T1    3
## 9   1   control     work   T2   22
## 10  2   control     work   T2   27
## 11  3 treatment     work   T2   20
## 12  4 treatment     work   T2   16
## 13  1   control     home   T2   11
## 14  2   control     home   T2    7
## 15  3 treatment     home   T2   10
## 16  4 treatment     home   T2    6
```

8.6 Example: Stock Market data

Supposing you have the following packages, load them into `R`:

```
require(quantmod)
require(lubridate)
```

Then, download these data from Google Finance:

```
invisible(Sys.setlocale("LC_MESSAGES", "C"))
invisible(Sys.setlocale("LC_TIME", "C"))
getSymbols.google(Symbols = "IBM",
                  env = globalenv(),
                  return.class = 'data.frame',
                  from = "2015-01-01",
                  to = Sys.Date())
```

```
## [1] "IBM"
```

```
head(IBM)
```

```
##            IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume
## 2015-01-02   161.31   163.31  161.00    162.06    5525466
## 2015-01-05   161.27   161.27  159.19    159.51    4880389
## 2015-01-06   159.67   159.96  155.17    156.07    6146712
## 2015-01-07   157.20   157.20  154.03    155.05    4701839
## 2015-01-08   156.24   159.04  155.55    158.42    4241113
## 2015-01-09   158.42   160.34  157.25    159.11    4488347
```

In the following code chunks you will find also some `dplyr` functions. To better understand them, see the next chapters of the course, in particular the `dplyr` parts about `mutate()` and `summarise()`.

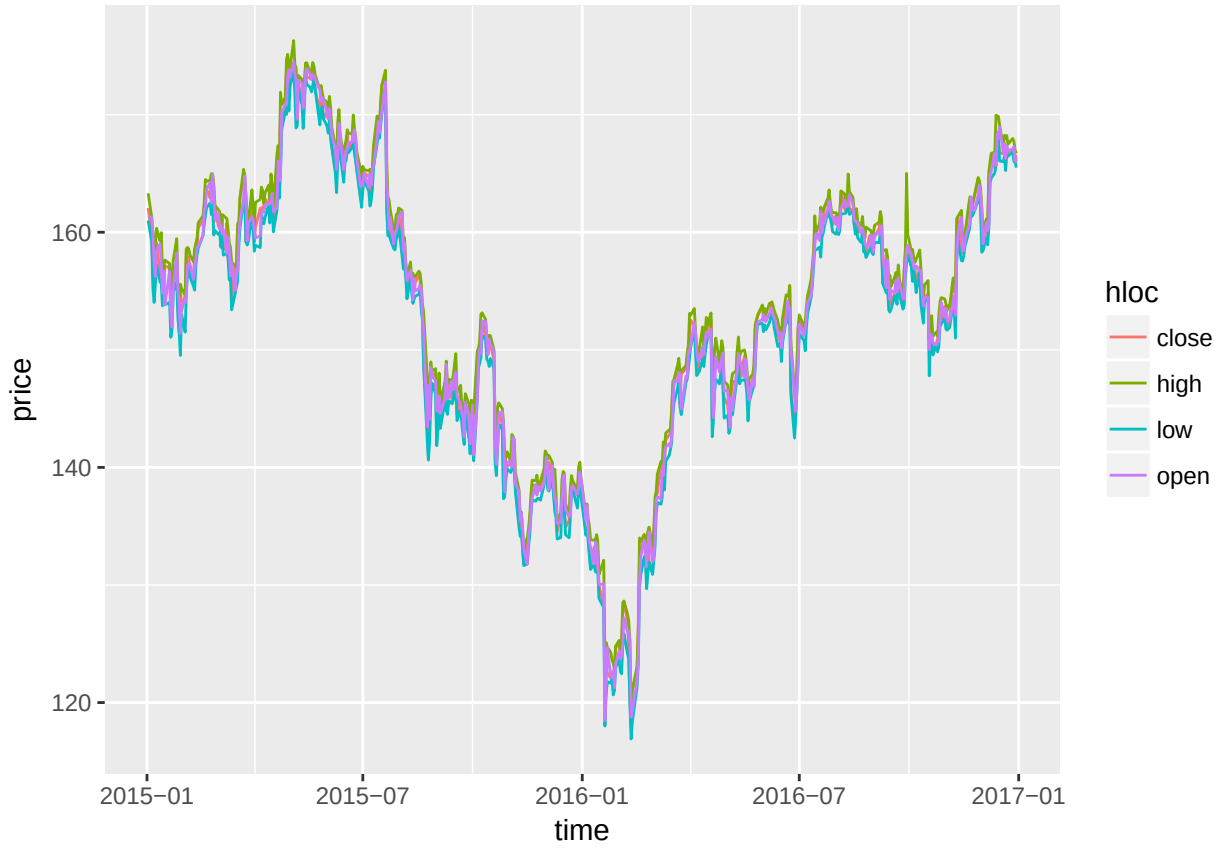
Manipulate data from wide to long:

```
df  <- IBM  %>%
  transmute(time = ymd(row.names(.)), open = IBM.Open, high = IBM.High, low = IBM.Low,
            close = IBM.Close) %>%
  gather(key = hloc, value = price, -time) %>%
  mutate(hloc = factor(hloc)) %>%
  tbl_df()
head(df)
```

```
## # A tibble: 6 × 3
##         time   hloc  price
##       <date> <fctr>  <dbl>
## 1 2015-01-02   open 161.31
## 2 2015-01-05   open 161.27
## 3 2015-01-06   open 159.67
## 4 2015-01-07   open 157.20
## 5 2015-01-08   open 156.24
## 6 2015-01-09   open 158.42
```

Plot data with ggplot:

```
ggplot(data = df, aes(x = time, y = price, colour = hloc)) + geom_line()
```



Compute returns:

```
df <- df %>% group_by(hloc) %>% mutate(returns = (price - lag(price))/lag(price))
head(df)
```

```
## Source: local data frame [6 x 4]
## Groups: hloc [1]
##
##         time   hloc  price       returns
##       <date> <fctr>  <dbl>         <dbl>
## 1 2015-01-02   open 161.31            NA
## 2 2015-01-05   open 161.27 -0.0002479697
## 3 2015-01-06   open 159.67 -0.0099212501
## 4 2015-01-07   open 157.20 -0.0154694056
## 5 2015-01-08   open 156.24 -0.0061068702
## 6 2015-01-09   open 158.42  0.0139528930
```

And summarise them:

```
df %>%
  group_by(hloc) %>%
  summarise(n = n(), avg = mean(returns, na.rm = TRUE), sd = sd(returns, na.rm = TRUE))
```

```
## # A tibble: 4 × 4
##     hloc     n          avg         sd
##   <fctr> <int>        <dbl>      <dbl>
## 1  close   504 0.0001317910 0.01294992
## 2   high   504 0.0001055665 0.01137312
## 3    low   504 0.0001330205 0.01245763
## 4   open   504 0.0001409560 0.01250028
```

Chapter 9

Managing dates with lubridate

```
require(lubridate)
require(dplyr)
require(ggplot2)
require(tidyr)
require(qdata)
```

9.1 A First Look to `lubridate`

Among packages that deals with dates and times in R, `lubridate` provides a set of intuitive and coherent functions. Intuitive since each function usually perform a single task described by its name; coherent since `lubridate` is well integrated with other data management tools, like `dplyr`.

Supposing `lubridate` is currently installed, it must be loaded in order to use its functions.

```
require(lubridate)
```

The starting point is a character string.

```
chr <- "01-02-12"
class(chr)
```

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

The above string is an ambiguous date:

- in Europe, it can be easily interpreted as 1st February 2012,
- in the US, it can be read as January 2, 2012,
- IT developers write in this way the day 12 February 2001.

`lubridate` provides a single function for each one of these cases: `dmy()`, `mdy()` and `ymd()`. These functions parse dates according to the order provided, where `y` is the *year*, `m` is the *month* and `d` is the *day*.

```
dmy(chr)
```

```
## [1] "2012-02-01"
```

```
mdy(chr)
```

```
## [1] "2012-01-02"
```

```
ymd(chr)
```

```
## [1] "2001-02-12"
```

```
class(dmy(chr))
```

```
## [1] "Date"
```

Parsed objects have now two new classes: `POSIXct` and `POSIXt`. As Wikipedia states, *the Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.*

The `POSIXct` format stores dates as the number of seconds since the start of January 1, 1970, as shown by `as.numeric()` below; `POSIXt` is a virtual class common to `POSIXct` and other `POSIX*` classes that are not used here.

```
as.numeric(dmy(chr))
```

```
## [1] 15371
```

The `lubridate` package provides other parsing functions, to deal with less frequent cases `myd()`, `dym()` and `ydm()`.

```
myd(chr)
```

```
## [1] "2002-01-12"
```

```
dym(chr)
```

```
## [1] "2002-12-01"
```

```
ydm(chr)
```

```
## [1] "2001-12-02"
```

Accordingly to the help, `ymd()`-style *functions recognize arbitrary non-digit separators as well as no separator. As long as the order of formats is correct, these functions will parse dates correctly even when the input vectors contain differently formatted dates.*

```
x <- c(20160101, "2016-01-02", "2016 01 03", "2016-1-4", "2016-1, 5",
       "Created on 2016 1 6", "201601 !!! 07")
ymd(x)
```

```
## [1] "2016-01-01" "2016-01-02" "2016-01-03" "2016-01-04" "2016-01-05" "2016-01-06" "2016-01-07"
```

9.2 Extract Date Components

The following function returns an `IBM` data frame with IBM quotation data from Yahoo! Finance. Notice that the function refer to the `quantmod` package, so it should be installed.

```
quantmod::getSymbols("IBM", return.class = 'data.frame')
```

```
## [1] "IBM"
```

```
head(IBM)
```

```
##            IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume IBM.Adjusted
## 2007-01-03    97.18    98.40   96.26     97.27    9196800     78.35465
## 2007-01-04    97.25    98.79   96.88     98.31   10524500     79.19241
## 2007-01-05    97.60    97.95   96.91     97.42    7221300     78.47548
```

```
## 2007-01-08    98.50    99.50   98.35     98.90   10340000     79.66768
## 2007-01-09    99.08   100.33   99.07    100.07   11108200     80.61016
## 2007-01-10    98.50    99.05   97.93     98.89    8744800     79.65962
```

The data frame contains dates as row names but data manipulation can be performed better on a variable. With a bit of `dplyr` code the new column can be added.

```
IBM <- IBM %>% mutate(date = row.names(.)) %>% tbl_df
IBM
```

```
## # A tibble: 2,518 × 7
##    IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume IBM.Adjusted       date
##       <dbl>    <dbl>   <dbl>     <dbl>      <dbl>        <dbl>      <chr>
## 1     97.18    98.40   96.26     97.27    9196800     78.35465 2007-01-03
## 2     97.25    98.79   96.88     98.31   10524500     79.19241 2007-01-04
## 3     97.60    97.95   96.91     97.42    7221300     78.47548 2007-01-05
## 4     98.50    99.50   98.35     98.90   10340000     79.66768 2007-01-08
## 5     99.08   100.33   99.07    100.07   11108200     80.61016 2007-01-09
## 6     98.50    99.05   97.93     98.89    8744800     79.65962 2007-01-10
## 7     99.00    99.90   98.50     98.65    8000700     79.46630 2007-01-11
## 8     98.99    99.69   98.50     99.34    6636500     80.02211 2007-01-12
## 9     99.40   100.84   99.30    100.82    9602200     81.21431 2007-01-16
## 10   100.69   100.90   99.90    100.02    8200700     80.56988 2007-01-17
## # ... with 2,508 more rows
```

At this point, the new `date` column is a string and it must be parsed to be interpreted by R as a date.

```
IBM <- IBM %>% mutate(date = ymd(date))
IBM
```

```
## # A tibble: 2,518 × 7
##    IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume IBM.Adjusted       date
##       <dbl>    <dbl>   <dbl>     <dbl>      <dbl>        <dbl>     <date>
## 1     97.18    98.40   96.26     97.27    9196800     78.35465 2007-01-03
## 2     97.25    98.79   96.88     98.31   10524500     79.19241 2007-01-04
## 3     97.60    97.95   96.91     97.42    7221300     78.47548 2007-01-05
## 4     98.50    99.50   98.35     98.90   10340000     79.66768 2007-01-08
## 5     99.08   100.33   99.07    100.07   11108200     80.61016 2007-01-09
## 6     98.50    99.05   97.93     98.89    8744800     79.65962 2007-01-10
## 7     99.00    99.90   98.50     98.65    8000700     79.46630 2007-01-11
## 8     98.99    99.69   98.50     99.34    6636500     80.02211 2007-01-12
## 9     99.40   100.84   99.30    100.82    9602200     81.21431 2007-01-16
## 10   100.69   100.90   99.90    100.02    8200700     80.56988 2007-01-17
## # ... with 2,508 more rows
```

The `lubridate` package provide a set of function to extract single components of a date: `month()`, `day()` and `year()`. Using just `lubridate` and `dplyr` the components of a date can be added as new variables to a data frame.

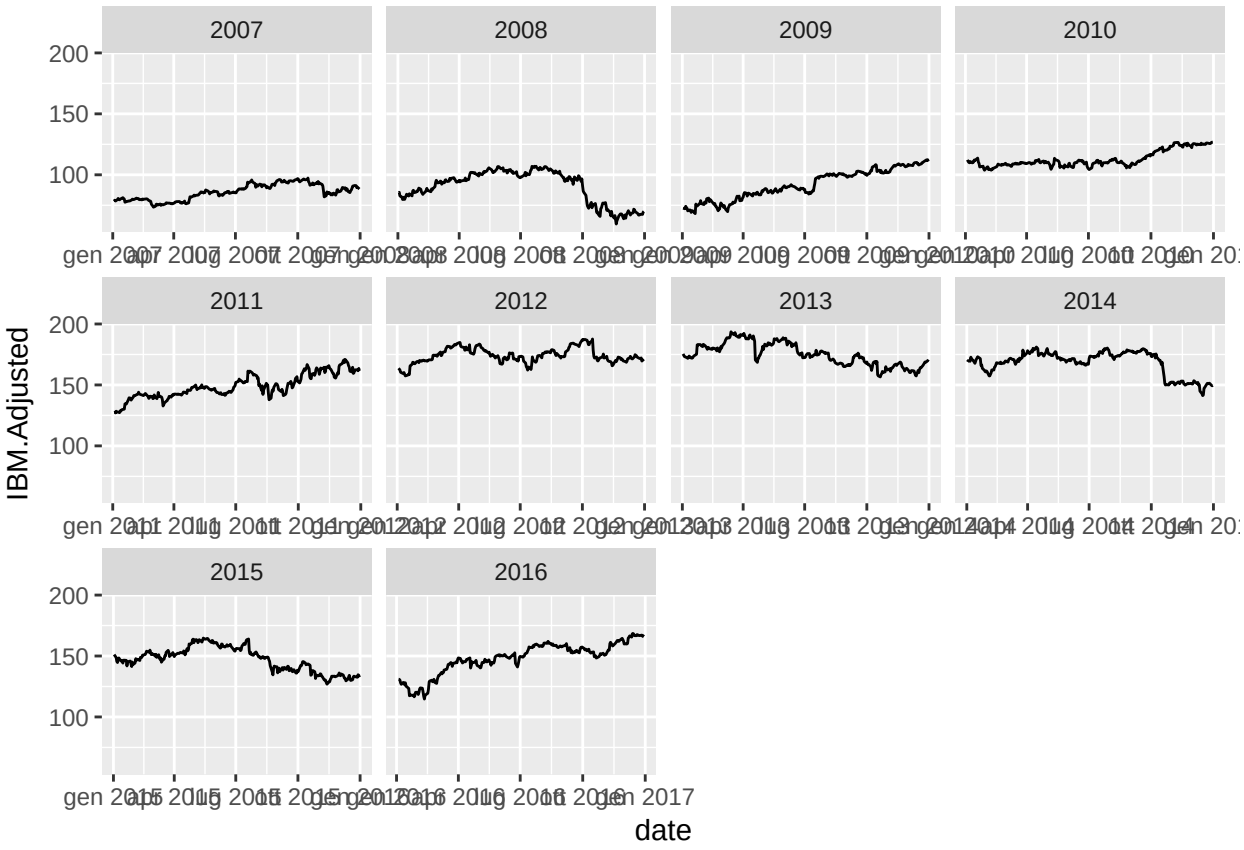
```
IBM <- IBM %>% mutate(month = month(date), day = day(date), year = year(date))
IBM
```

```
## # A tibble: 2,518 × 10
##    IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume IBM.Adjusted       date month   day  year
##       <dbl>    <dbl>   <dbl>     <dbl>      <dbl>        <dbl>     <date> <dbl> <int> <dbl>
## 1     97.18    98.40   96.26     97.27    9196800     78.35465 2007-01-03     1     3  2007
## 2     97.25    98.79   96.88     98.31   10524500     79.19241 2007-01-04     1     4  2007
## 3     97.60    97.95   96.91     97.42    7221300     78.47548 2007-01-05     1     5  2007
```

```
## 4     98.50    99.50   98.35    98.90   10340000     79.66768 2007-01-08     1     8  2007
## 5     99.08   100.33   99.07   100.07   11108200     80.61016 2007-01-09     1     9  2007
## 6     98.50    99.05   97.93    98.89    8744800     79.65962 2007-01-10     1    10  2007
## 7     99.00    99.90   98.50    98.65    8000700     79.46630 2007-01-11     1    11  2007
## 8     98.99    99.69   98.50    99.34    6636500     80.02211 2007-01-12     1    12  2007
## 9     99.40   100.84   99.30   100.82    9602200     81.21431 2007-01-16     1    16  2007
## 10   100.69   100.90   99.90   100.02    8200700     80.56988 2007-01-17     1    17  2007
## # ... with 2,508 more rows
```

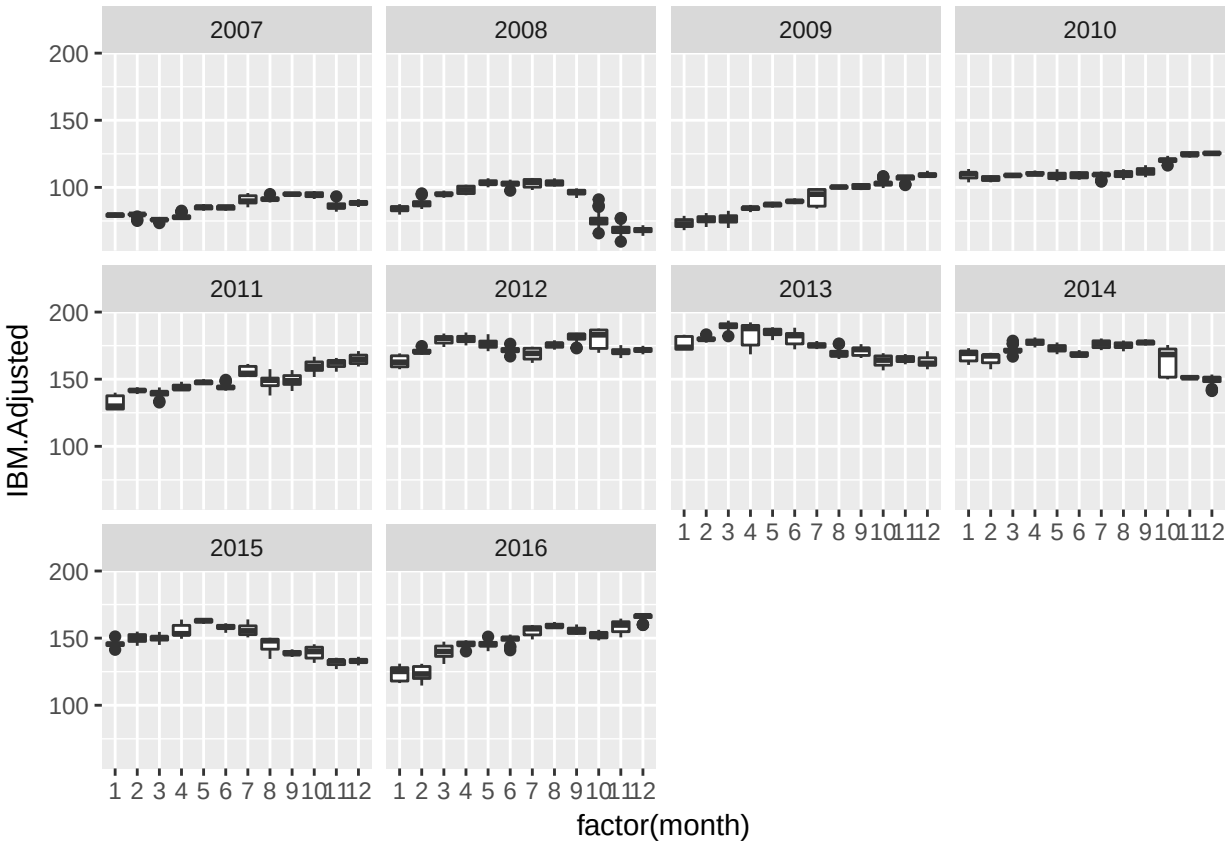
At this point, you can easily draw a time series plot with adjusted quotations for the IBM, with a facet for each year.

```
ggplot(data=IBM, mapping=aes(x=date)) +
  geom_line(aes(y=IBM.Adjusted)) +
  facet_wrap(~ year, scales="free_x")
```



Similarly, you can explore differences among months with a box plot for each month.

```
ggplot(data=IBM, mapping=aes(x=factor(month), y=IBM.Adjusted, group=month)) +
  geom_boxplot() +
  facet_wrap(~ year)
```



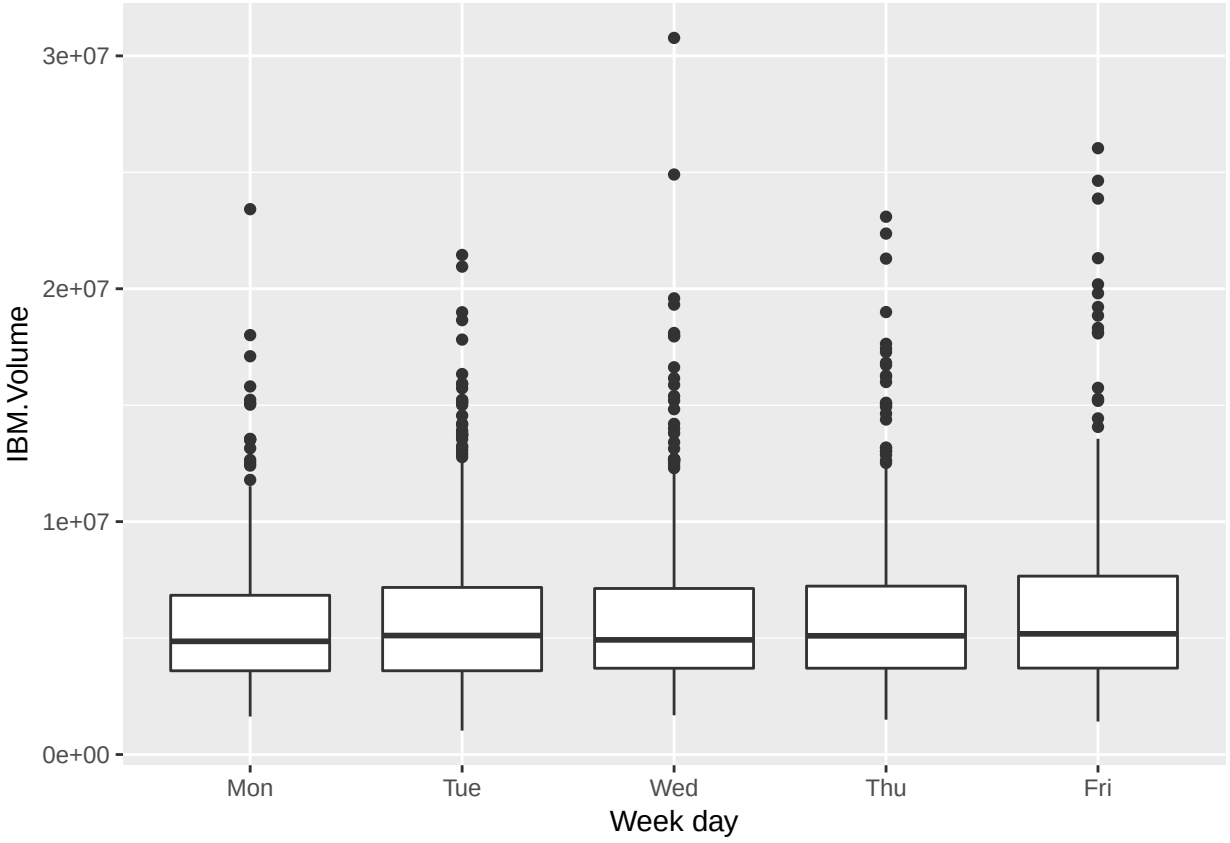
The last function shown here is `wday()`: it extracts the week day of a date, returning a number from Sunday (1) to Saturday (7).

```
IBM <- IBM %>% mutate(weekday = wday(date))
IBM
```

```
## # A tibble: 2,518 × 11
##   IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume IBM.Adjusted     date month   day  year weekday
##      <dbl>    <dbl>   <dbl>     <dbl>      <dbl>       <dbl>    <date> <dbl> <int> <dbl>   <dbl>
## 1    97.18    98.40   96.26     97.27    9196800     78.35465 2007-01-03     1     3  2007       4
## 2    97.25    98.79   96.88     98.31   10524500     79.19241 2007-01-04     1     4  2007       5
## 3    97.60    97.95   96.91     97.42    7221300     78.47548 2007-01-05     1     5  2007       6
## 4    98.50    99.50   98.35     98.90   10340000     79.66768 2007-01-08     1     8  2007       2
## 5    99.08   100.33   99.07    100.07   11108200     80.61016 2007-01-09     1     9  2007       3
## 6    98.50    99.05   97.93     98.89    8744800     79.65962 2007-01-10     1    10  2007       4
## 7    99.00    99.90   98.50     98.65    8000700     79.46630 2007-01-11     1    11  2007       5
## 8    98.99    99.69   98.50     99.34    6636500     80.02211 2007-01-12     1    12  2007       6
## 9    99.40   100.84   99.30    100.82    9602200     81.21431 2007-01-16     1    16  2007       3
## 10  100.69   100.90   99.90    100.02    8200700     80.56988 2007-01-17     1    17  2007       4
## # ... with 2,508 more rows
```

Of course, also the new variable can be used as grouping variable for box plots.

```
ggplot(data=IBM, mapping=aes(
    x=factor(weekday, labels=c("Mon", "Tue", "Wed", "Thu", "Fri")), y=IBM.Volume,
    group=weekday)
  ) + geom_boxplot() + xlab("Week day")
```



9.3 Dealing with Years: Leap Year and Date Differences

A leap year (also known as a bissextile year) is a year containing 366 days instead of the usual 365, by extending February to 29 days rather than the common 28.

Since 2016 is a leap year, the difference between March 17, 2016 and March 17, 2015 is 366 days.

```
ymd("2016-03-17") - ymd("2015-03-17")
```

```
## Time difference of 366 days
```

By the way, if you add an year to a date may be you expect the same day/month of the following year. The function `years()` add a number of years, ignoring leap years. Notice the spelling difference among this function, `years()` and the function to extract the `year()` component of a date, that have the singular form of the units as a name.

```
ymd("2015-03-17") + years(1)
```

```
## [1] "2016-03-17"
```

In some cases, you may be interested in the exact time differences. The function `dyears()` adds an year (i.e. 365 days) according to leap years.

```
ymd("2015-03-17") + dyears(1)
```

```
## [1] "2016-03-16"
```

9.4 Dealing with Months and Days

Similarly to `years()`, the `months()` function return the same day shifted by a specified number of months.

```
ymd("2016-06-01") + months(1)
```

```
## [1] "2016-07-01"
```

```
ymd("2016-06-01") - months(6)
```

```
## [1] "2015-12-01"
```

If you try to add a month to January 31, you get a `NA` value. This is because February does *not* have 31 days.

```
ymd("2016-01-31") + months(1)
```

```
## [1] NA
```

In these cases you may want to add to January 31 a "standard" month, that is 30 days. You can solve the issue with the `days()` function.

```
ymd("2016-01-31") + days(30)
```

```
## [1] "2016-03-01"
```

Otherwise, you may want a month later January 31, without exceeding the following month. The package `lubridate` allows you to deal with this case, using the special operator `%m+%`.

```
ymd("2016-01-31") %m+% months(1)
```

```
## [1] "2016-02-29"
```

As shown above, you can shift a date by a given number of `days()`:

```
ymd("2016-01-31") + days(7)
```

```
## [1] "2016-02-07"
```

```
ymd("2016-03-26") - days(360)
```

```
## [1] "2015-04-01"
```

9.5 Time Parsing

Package `lubridate` allow you to deal with times, too. Times have no mean without dates, so you will parse the whole date-time string.

When data has date and time in two different variables, you can merge it an a single variable with a bit of `tidyr`. The following example requires data frame `diabetes` from `qdata` package.

```
data(diabetes)
names(diabetes)
```

```
## [1] "patient" "date"    "time"    "code"    "value"
```

```
diabetes <- diabetes %>% unite(datetime, date, time)
diabetes
```

```
## # A tibble: 29,297 × 4
##    patient         datetime  code value
```

```
## *    <chr>            <chr> <int> <chr>
## 1       01  04-21-1991_9:09    58   100
## 2       01  04-21-1991_9:09    33     9
## 3       01  04-21-1991_9:09    34    13
## 4       01 04-21-1991_17:08    62   119
## 5       01 04-21-1991_17:08    33     7
## 6       01 04-21-1991_22:51    48   123
## 7       01  04-22-1991_7:35    58   216
## 8       01  04-22-1991_7:35    33    10
## 9       01  04-22-1991_7:35    34    13
## 10      01 04-22-1991_13:40    33     2
## # ... with 29,287 more rows
```

As for dates, you can use the parsing functions to tell R that `datetime` is a date-time object.

```
diabetes <- diabetes %>% mutate(datetime = mdy_hm(datetime))
```

```
## Warning: 7 failed to parse.
```

```
diabetes
```

```
## # A tibble: 29,297 × 4
##    patient            datetime  code value
##      <chr>              <dttm> <int> <chr>
## 1       01 1991-04-21 09:09:00    58   100
## 2       01 1991-04-21 09:09:00    33     9
## 3       01 1991-04-21 09:09:00    34    13
## 4       01 1991-04-21 17:08:00    62   119
## 5       01 1991-04-21 17:08:00    33     7
## 6       01 1991-04-21 22:51:00    48   123
## 7       01 1991-04-22 07:35:00    58   216
## 8       01 1991-04-22 07:35:00    33    10
## 9       01 1991-04-22 07:35:00    34    13
## 10      01 1991-04-22 13:40:00    33     2
## # ... with 29,287 more rows
```

The `lubridate` package provides three types of functions that deal with date-time strings:

- `ymd_h()` when time has only hours
- `ymd_hm()` when time has hours and minutes
- `ymd_hms()` when time has hours, minutes and seconds.

Notice that `ymd()` can be replaced by any date style (e.g. `mdy`, `dmy` etc.) as seen above. Also in this case, functions recognize arbitrary separators.

```
ymd_h("2016-03-10 08")
```

```
## [1] "2016-03-10 08:00:00 UTC"
```

```
ymd_hm("2016-03-10 08:15")
```

```
## [1] "2016-03-10 08:15:00 UTC"
```

```
ymd_hms("2016-03-10 08:15:40")
```

```
## [1] "2016-03-10 08:15:40 UTC"
```

```
ymd_hms("16+03/10*08.15,40")
```

```
## [1] "2016-03-10 08:15:40 UTC"
```

```
ymd_hms("160310:081540")
```

```
## [1] "2016-03-10 08:15:40 UTC"
```

The function `now` returns the current date-time.

```
now()
```

```
## [1] "2017-01-03 13:26:58 CET"
```

9.6 Time Zones

Date and times may vary with refer to timezone. When you define a `POSIXct` class object with parsing function seen above, you can refer to a specified timezone.

For example, you can set March 1, 2016 at 2.00 PM in the Central European Time (CET).

```
ymd_hms("2016-03-01 14:00:00", tz="CET")
```

```
## [1] "2016-03-01 14:00:00 CET"
```

In the same way, you can set April 1, 2016 at 2.00 PM.

```
ymd_hms("2016-04-01 14:00:00", tz="CET")
```

```
## [1] "2016-04-01 14:00:00 CEST"
```

The parsed object return `CEST` instead of `CET`. This is because April is during Daylight Saving Time, and CET area move to Central European *Summer* Time (CEST).

The next examples show how to set timezone when parsing date.

Function `force_tz()` forces a date-time to a new time zone

Function `with_tz()` returns a date-time as it would appear in a different time zone

```
d <- ymd_hms("2016-03-01 14.00.00")
d
```

```
## [1] "2016-03-01 14:00:00 UTC"
```

```
force_tz(d, tz="CET")
```

```
## [1] "2016-03-01 14:00:00 CET"
```

```
with_tz(d, tz="CET")
```

```
## [1] "2016-03-01 15:00:00 CET"
```

9.7 Dealing with Hours, Minutes and Seconds

Similarly to functions `years()` and `dyears`, `lubridate` provides functions for time duration:

- `hours()`
- `minutes()`
- `seconds()`

for relative time spans and:

- `dhours()`
- `dminutes()`
- `dseconds()`

for exact time spans.

Relative time spans shift times according to the input (e.g. one hour more), while exact time spans return the exact time after n seconds, being n the function input. Relative and exact time spans are identical, unless there are "external" adjustments of time, like Daylight Saving Time.

As an example, you can substract an hour to October 30, 2016 at 2.00 AM. The `hours()` function returns 1.00 AM, since it just subtract an hour; the `dhours()` function returns 2.00 AM again, because in this night DST ends and clocks are set back, as if to repeat one hour.

```
ymd_hms("2016-10-30 02:00:00", tz="CET") - hours(1)
```

```
## [1] "2016-10-30 01:00:00 CEST"
```

```
ymd_hms("2016-10-30 02:00:00", tz="CET") - dhours(1)
```

```
## [1] "2016-10-30 02:00:00 CEST"
```

If you want to go crazy, take a look to this example. If you add an hour to 1.30 AM of March 27, 2016 using `hours()` you get a `NA` value, because the 2.59 AM does not exists in this night: DST begins and clocks skip an hour. If you add an hour to 1.30 AM of March 27, 2016 using `dhours()` you get 3.30 AM, the exact time an hour later.

```
ymd_hms("2016-03-27 01:30:00", tz="CET") + hours(1)
```

```
## [1] NA
```

```
ymd_hms("2016-03-27 01:30:00", tz="CET") + dhours(1)
```

```
## [1] "2016-03-27 03:30:00 CEST"
```

9.8 Rounding Date and Times

The `lubridate` package provides a set of functions to round date and times, whose names recall numeric rounding functions of base R:

- `ceiling_date()` rounds a date-time up to the nearest integer value;
- `floor_date()` rounds a date-time down to the nearest integer value;
- `round_date()` rounds a date-time to the nearest integer value.

Value refers to the specified time unit: you can specify whether to round to the nearest second, minute, hour, day, week, month, or year using the `unit` argument.

```
dttm <- c(ymd_hms("2016-03-27 14:15:00"), ymd_hms("2016-03-27 14:30:00"),
          ymd_hms("2016-03-27 14:45:00"))
dttm
```

```
## [1] "2016-03-27 16:15:00 CEST" "2016-03-27 16:30:00 CEST" "2016-03-27 16:45:00 CEST"
```

```
ceiling_date(dttm, unit="hour")
```

```
## [1] "2016-03-27 17:00:00 CEST" "2016-03-27 17:00:00 CEST" "2016-03-27 17:00:00 CEST"
```

```
floor_date(dttm, unit="hour")
```

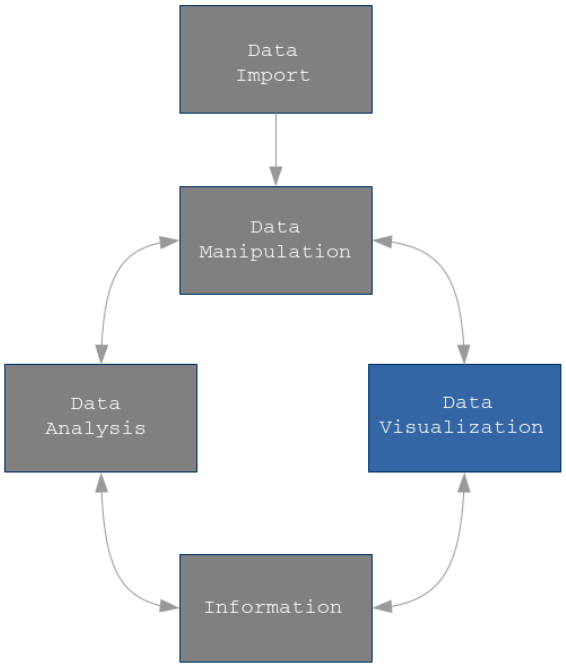
```
## [1] "2016-03-27 16:00:00 CEST" "2016-03-27 16:00:00 CEST" "2016-03-27 16:00:00 CEST"
```

```
round_date(dttm, unit="hour")
```

```
## [1] "2016-03-27 16:00:00 CEST" "2016-03-27 17:00:00 CEST" "2016-03-27 17:00:00 CEST"
```

Chapter 10

Data Visualization with R



Visualizing data is crucial in today's world. Without powerful visualizations, it is almost impossible to create and narrate stories on data. These stories help us build strategies and make intelligent business decisions.

`ggplot2` is a data visualization package which has become a synonym for data visualization in R.

```
require(ggplot2)
```

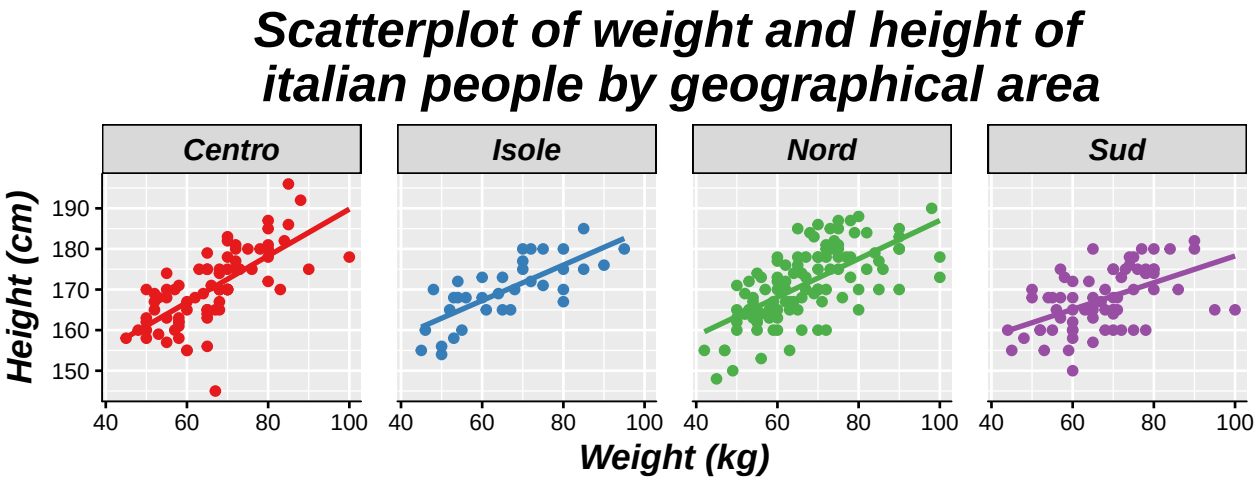
Created by Hadley Wickham in 2005, `ggplot2` is an implementation of Leland Wilkinson's Grammar of Graphics, a general scheme for data visualization which breaks up graphs into semantic independent components, such as scales and layers, that can be composed in many different ways. This makes `ggplot2` very powerful, because there are no limitations due to a set of pre-specified graphics, so it is possible to create new graphics that are precisely tailored for the problem in analysis.

10.1 An overview of ggplot2 grammar

Let us consider **people** dataset, included in **qdata** package. **people** dataset contains informations about weight, height, gender and geographical area of 300 italian people.

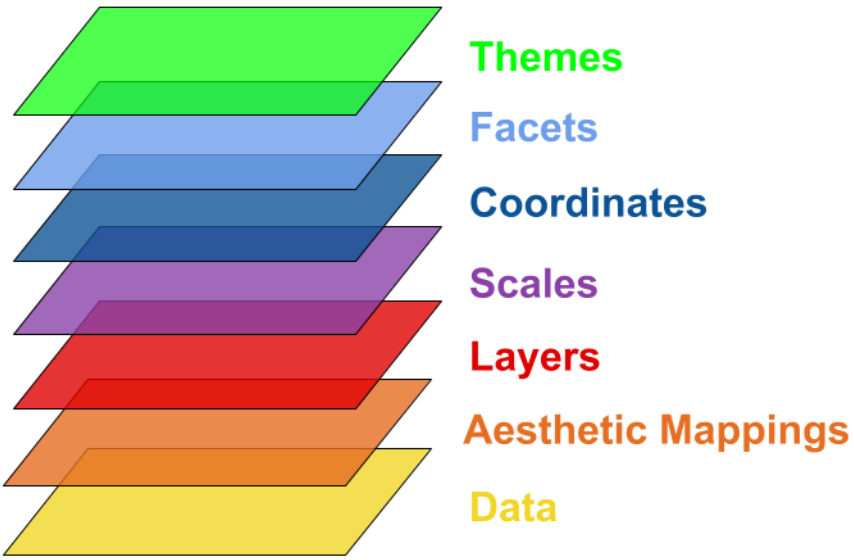
```
require(qdata)
```

Suppose we want to visualize the relationship between weight and height of italian people according to the different geographical area, by using a **scatterplot**:



The previous plot is composed of building blocks that are added to the plot one after the other.

The complete scheme of the most important building blocks of **ggplot2** grammar is displayed in the following figure:



The scheme must be read from bottom to top. Starting from bottom, the first three building blocks (Data , Aestethic Mappings and Layers) are fundamental to build a simple plot, indeed they are called **"key"** building blocks. The remaining building blocks (Scales , Coordinates , Facets and Themes) allow us to build a complex plot and to customize it; their use and order is not compulsory.

Let us briefly describe the task of each element of the scheme and how it helps build the previous plot:

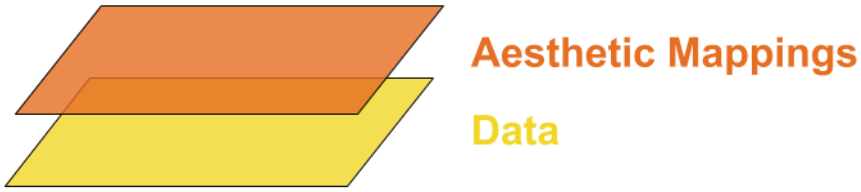
1. Data : the dataset that we want to visualize


```
# people dataset
data(people)
```

```
head(people)
```

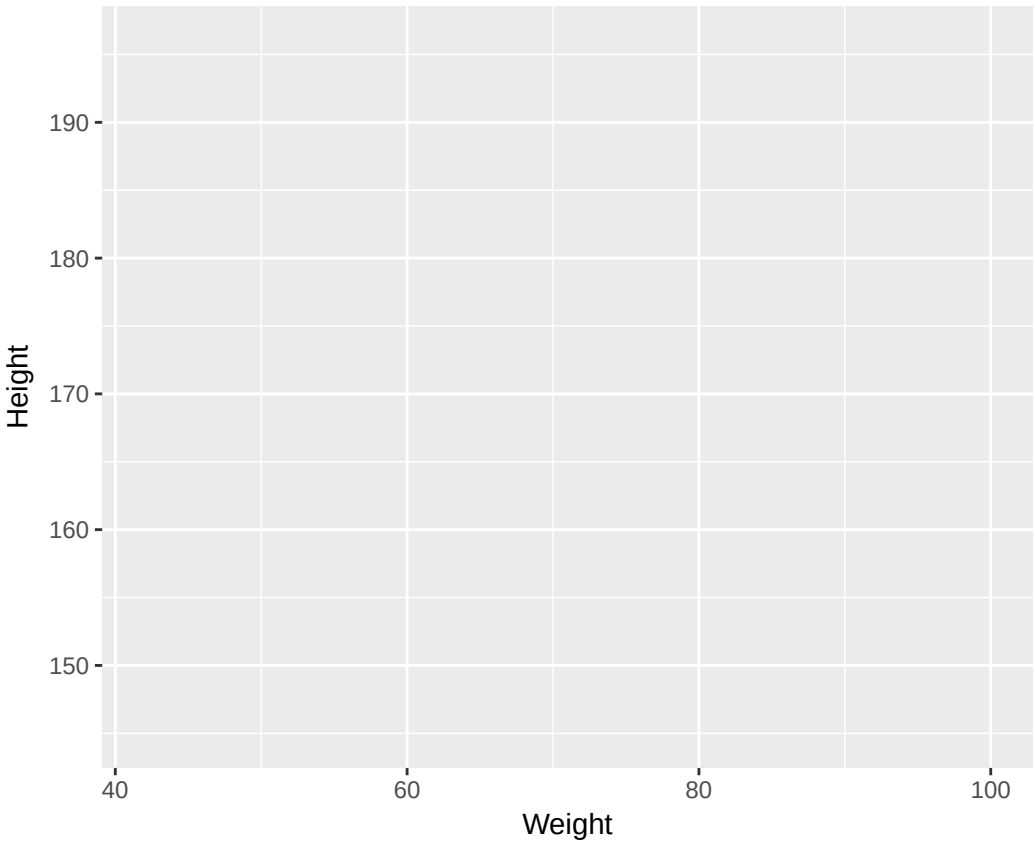
```
## # A tibble: 6 × 4
##   Gender   Area Weight Height
##   <fctr> <fctr>  <int>  <int>
## 1 Female  Isole     54    168
## 2 Female   Nord     61    171
## 3   Male    Sud     68    170
## 4 Female   Nord     52    164
## 5   Male   Nord     75    181
## 6   Male   Nord     77    178
```

2. Aestethic Mappings : describes how variables in the data are mapped to aestethic attributes that you can perceive

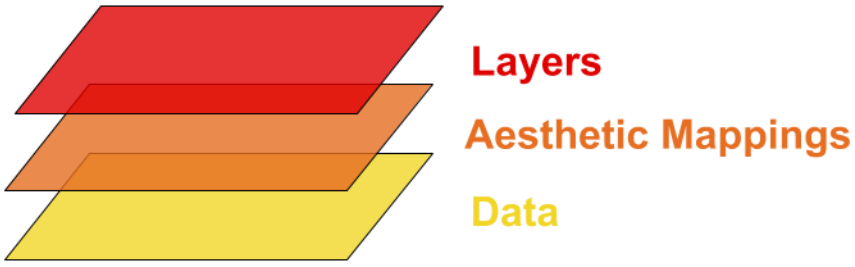


Gender	Area	Weight	Height
		x	y

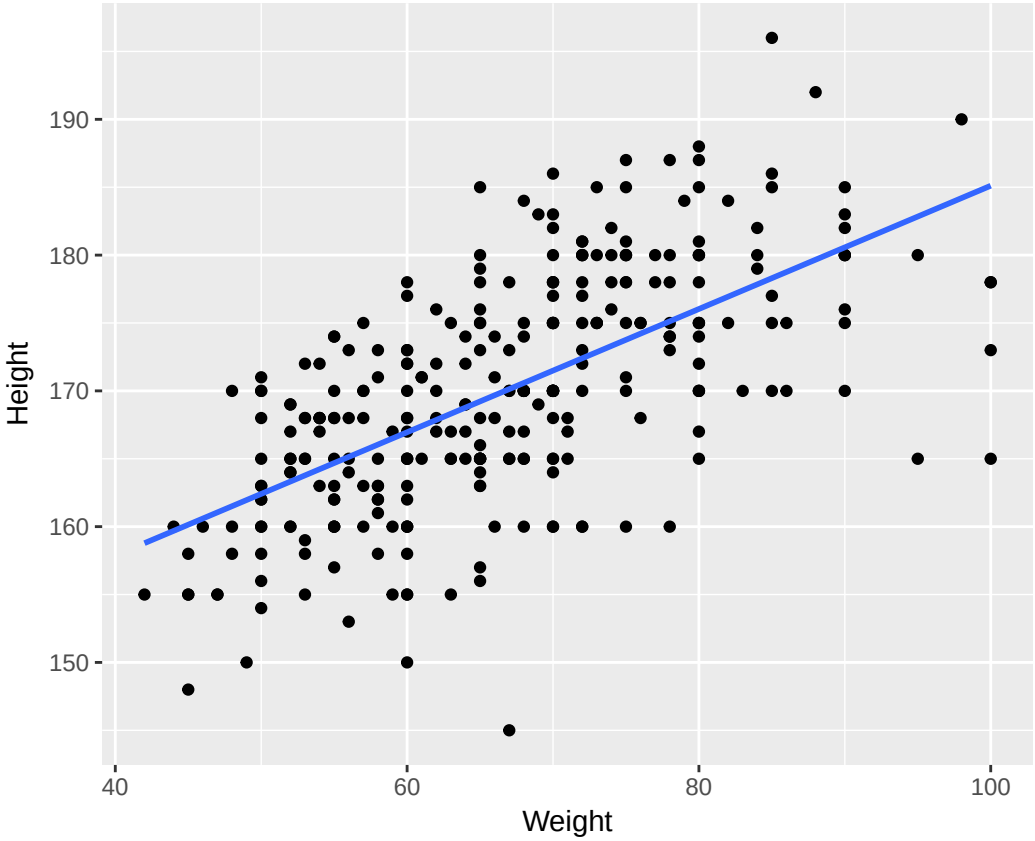
```
ggplot(data = people, aes(x = Weight, y = Height))
```



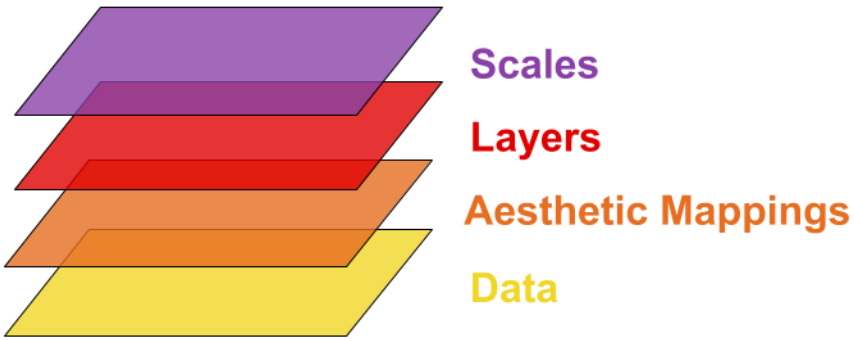
3. Layers : are made up by geometric elements and statistical transformations. In details, geometric objects (`geoms`) represent what we actually see on the plot: points, lines, polygons, etc. Statistical transformations (`stats`) summarise data in many useful ways. For example, binning and counting observations to create an histogram, or summarising a 2d relationship with a linear model.



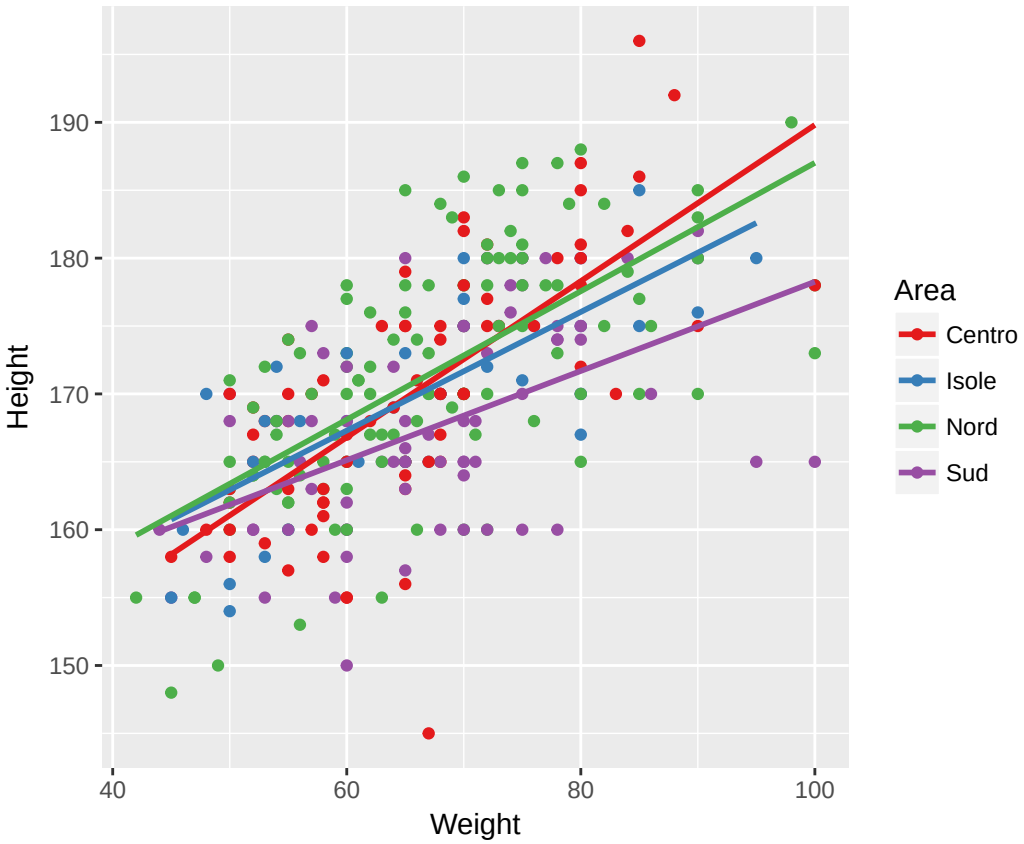

```
# Scatterplot of the relationship between weight and height with regression line
ggplot(people, aes(x = Weight, y = Height)) +
  geom_point() + # layer 1 (draw points)
  stat_smooth(method = "lm", se = FALSE) # layer 2 (draw regression line)
```



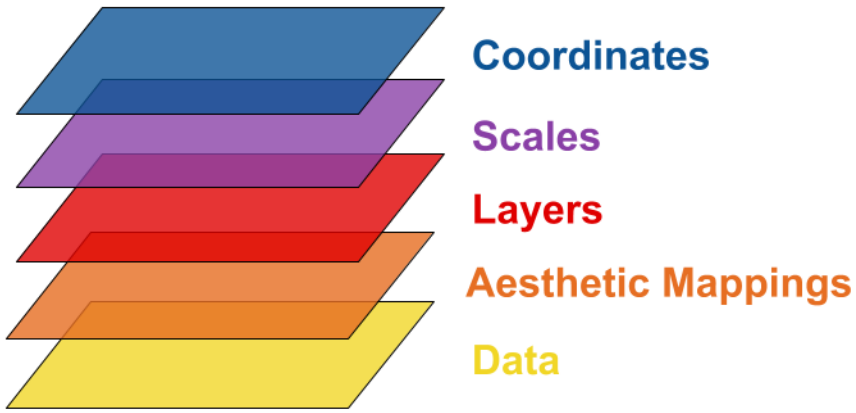
4. Scales : map values in the data space to values in an aesthetic space, whether it be colour, or size or shape. Scales draw a legend on axes, which provide an inverse mapping to make it possible to read the original data values from the graph. Scales are closely related to aestethics mapped.



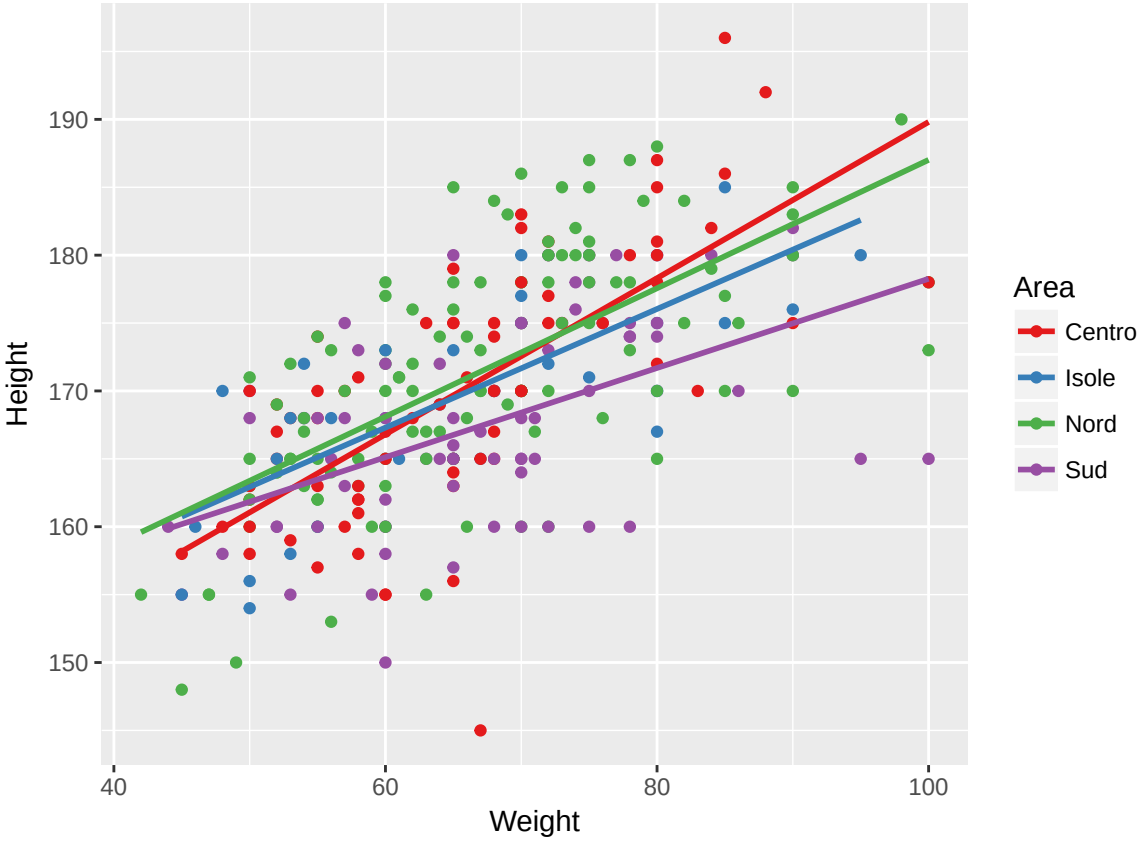

```
# map Area to colour in aes() and change the default colours of colour scale
ggplot(people, aes(x = Weight, y = Height, colour = Area)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1")
```



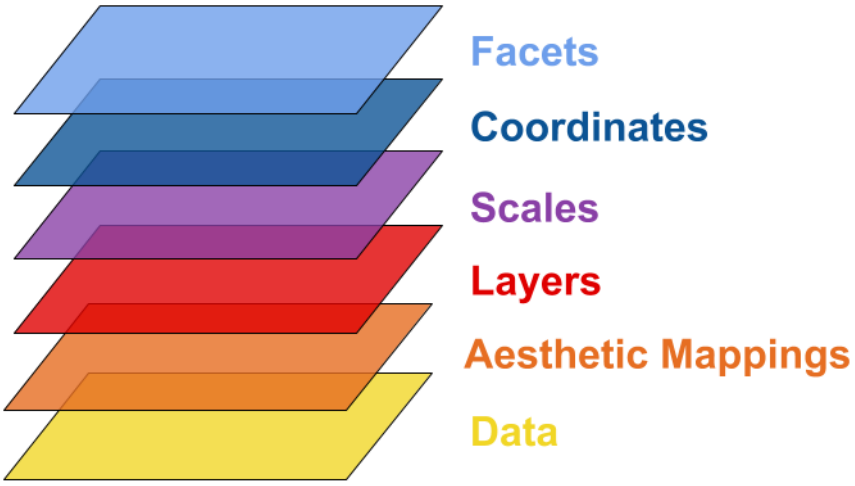
5. Coordinates : or coordinate system (coord) describe how data coordinates are mapped to the plane of the graphic. They also provides axes and gridlines to make it possible to read the graph. We normally use Cartesian coordinate system, but a number of others are available.



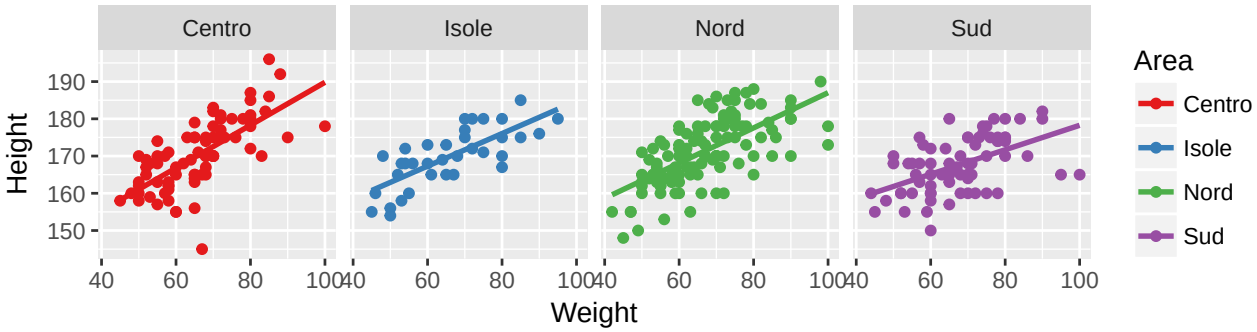

```
# Generate a plot for each geographical area
ggplot(people, aes(x = Weight, y = Height, colour = Area)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1") +
  coord_equal()
```



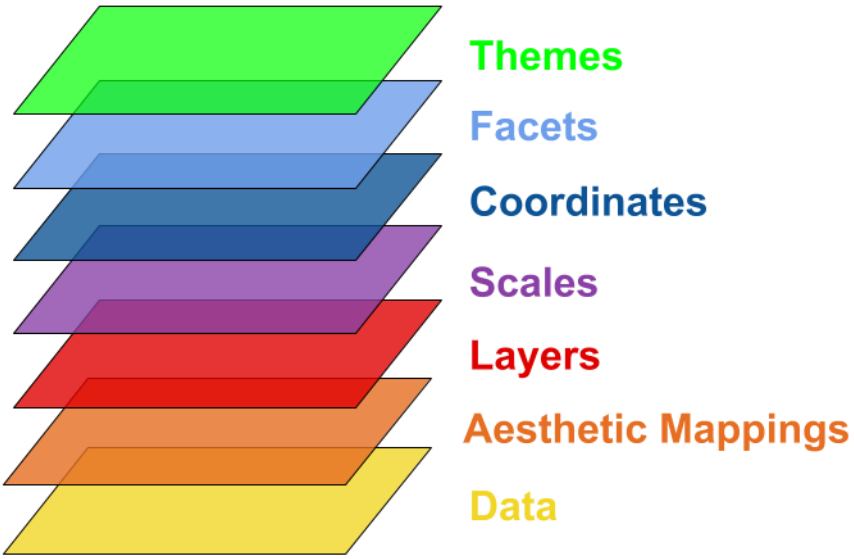
6. Facets : describes how to break up the data into subset and how to display those subsets as small multiples.




```
# Generate a plot for each geographical area
ggplot(people, aes(x = Weight, y = Height, colour = Area)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1") +
  coord_equal() +
  facet_grid(. ~ Area)
```

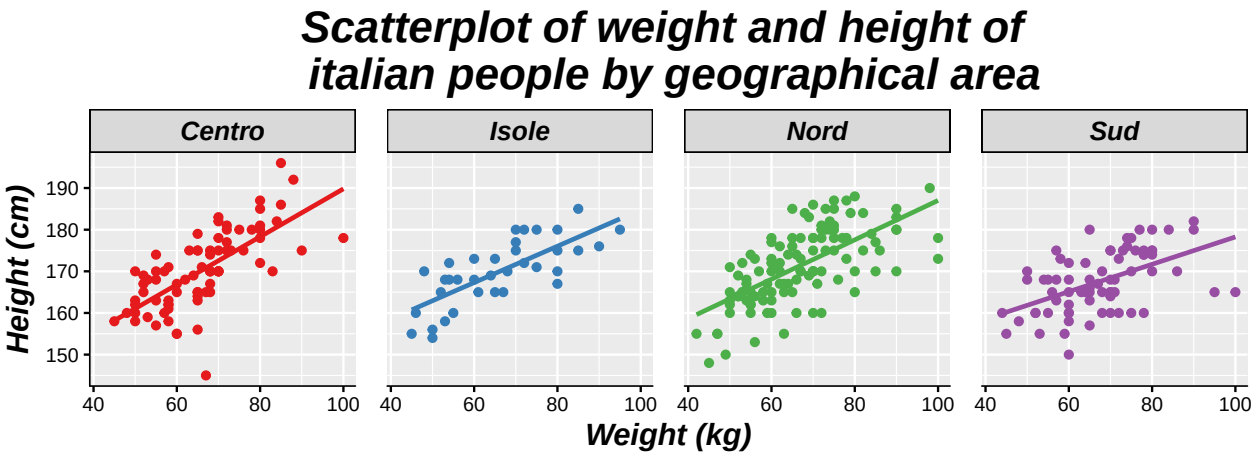


7. Themes : controls all non-data elements of the plot, like the font size, background colour, etc.




```
# Customize the appearance of the plot
ggplot(people, aes(x = Weight, y = Height, colour = Area)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1") +
  coord_equal() +
  facet_grid(. ~ Area) +
  ggtitle("Scatterplot of weight and height of \n italian people by geographical area") +
    xlab("Weight (kg)") +
    ylab("Height (cm)") +
  theme(plot.background = element_blank(),
    axis.text = element_text(colour = "black"),
    axis.ticks = element_line(colour = "black"),
    axis.line.x = element_line(colour = "black"),
    axis.line.y = element_line(colour = "black"),
    axis.title = element_text(colour = "black", size = 14, face = "bold.italic"),
    strip.background = element_rect(colour = "black"),
    strip.text = element_text(colour = "black", face = "bold.italic", size = 12),
    plot.title = element_text(colour = "black", size = 20, face = "bold.italic", hjust = 0.5),
    panel.spacing = unit(1, "lines"),
```

```
    legend.position = "none")
```

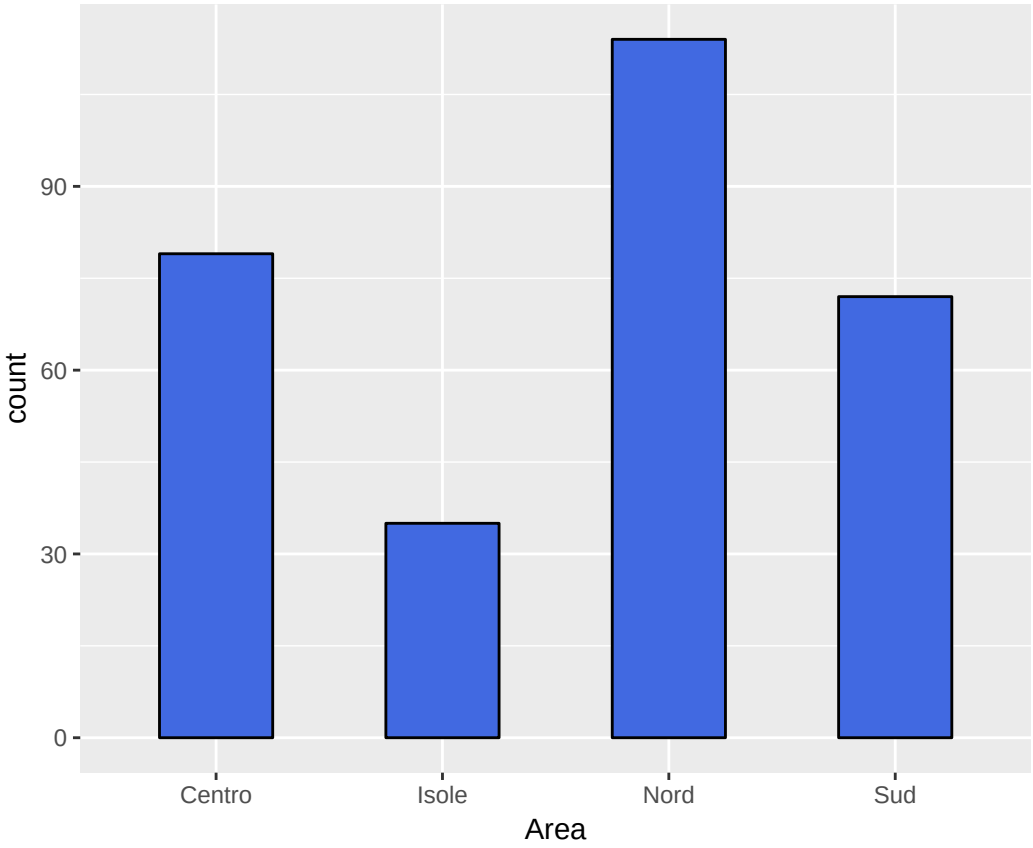


10.2 Other plot types

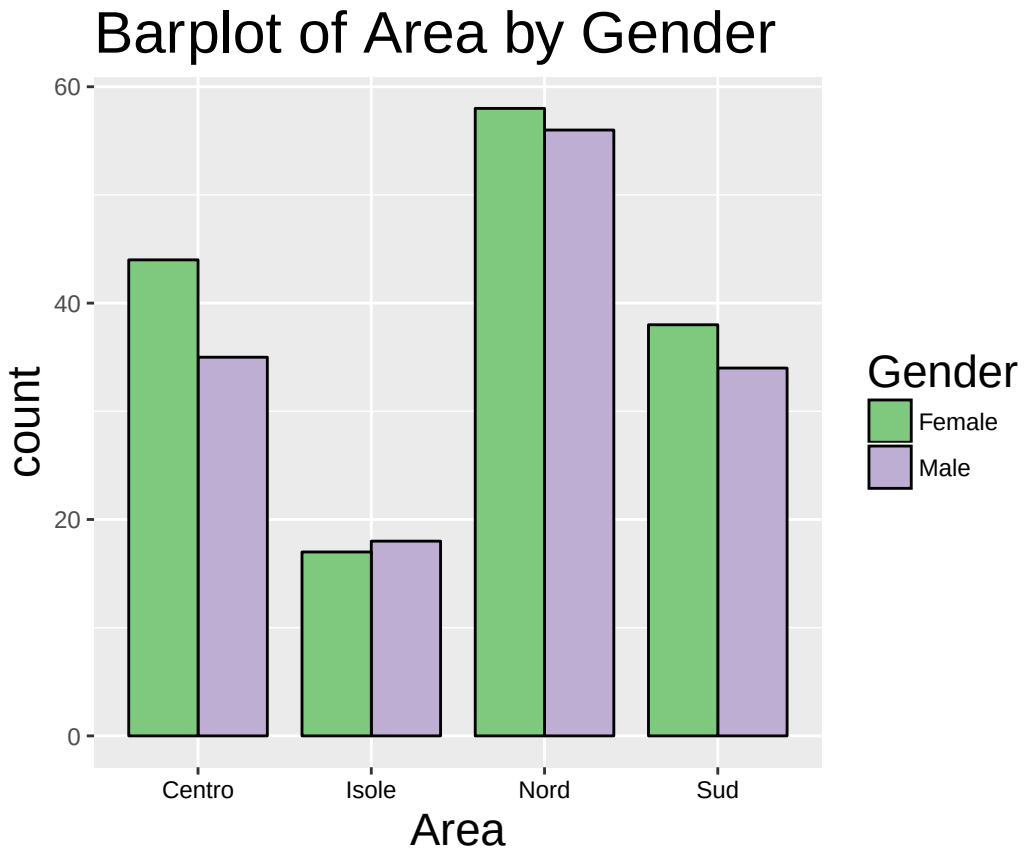
The building block scheme previously seen can help us understanding how to build other plot types. We will focus on the first three building blocks of the scheme (data, aestethic mappings and layers).

- **Barplot**, which is used to show the count of each case of a categorial variable. Suppose we want to analyze the count of people by geographical area:

```
# base plot: key building blocks (data, aes, layer)
ggplot(data = people, mapping = aes(x = Area)) +
  geom_bar(fill = "royalblue", colour = "black", width = 0.5)
```

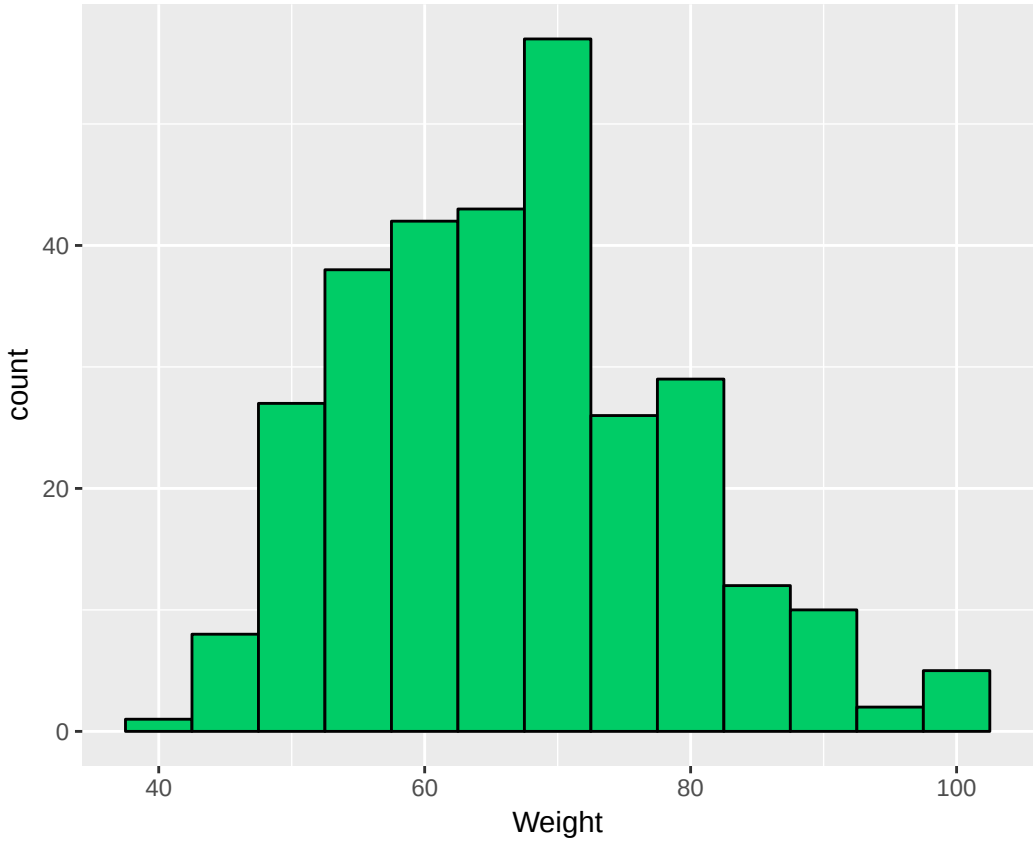


```
# customized plot: key building blocks + scales + theme
ggplot(data = people, mapping = aes(x = Area, fill = Gender)) +
  geom_bar(position = "dodge", width = 0.8, colour="black") +
       scale_fill_brewer(palette = "Accent") +
       ggtitle("Barplot of Area by Gender") +
       theme(axis.title.y = element_text(size = rel(1.5), angle = 90),
             axis.title.x = element_text(size = rel(1.5)),
             axis.text.x = element_text(colour="black"),
             plot.title = element_text(size = rel(2)),
             legend.title = element_text(size = rel(1.5)))
```

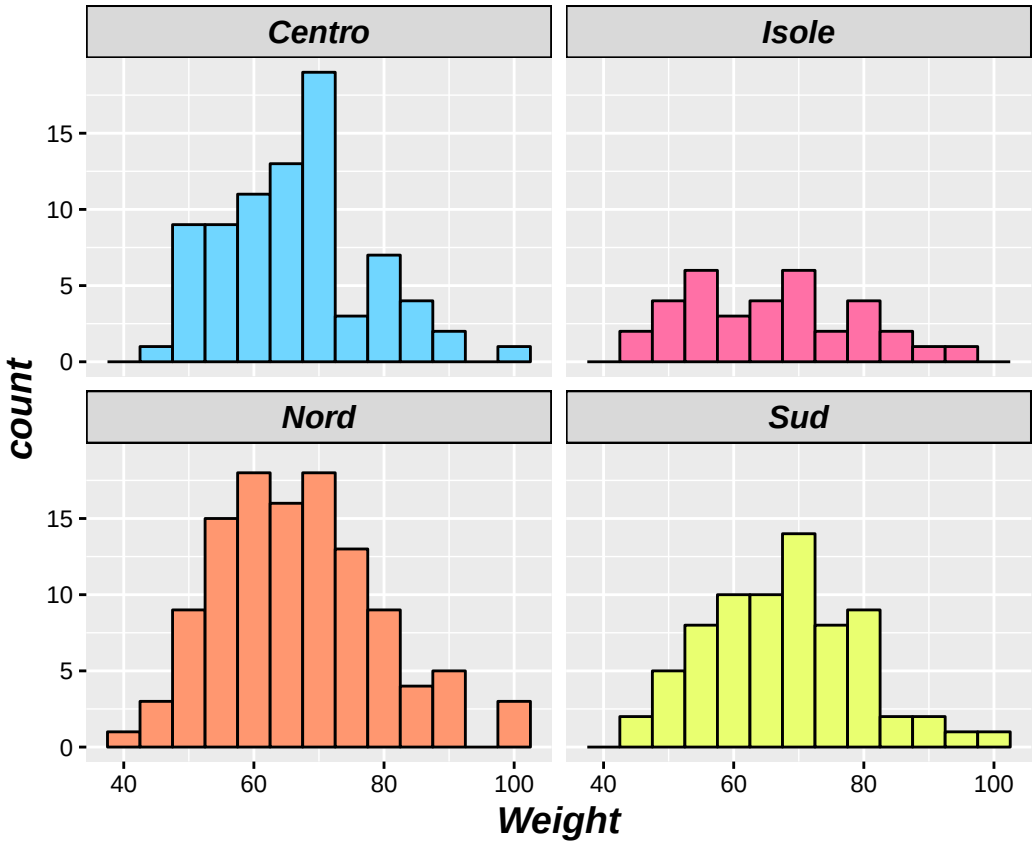


- **Histogram**, which is used to summarize a continuous variable into classes. Suppose we want to analyze the distribution of people weight:

```
# base plot: key building blocks (data, aes, layer)
ggplot(data=people, mapping=aes(x=Weight)) +
  geom_histogram(fill="#00cc66", colour= "#000000", binwidth=5)
```

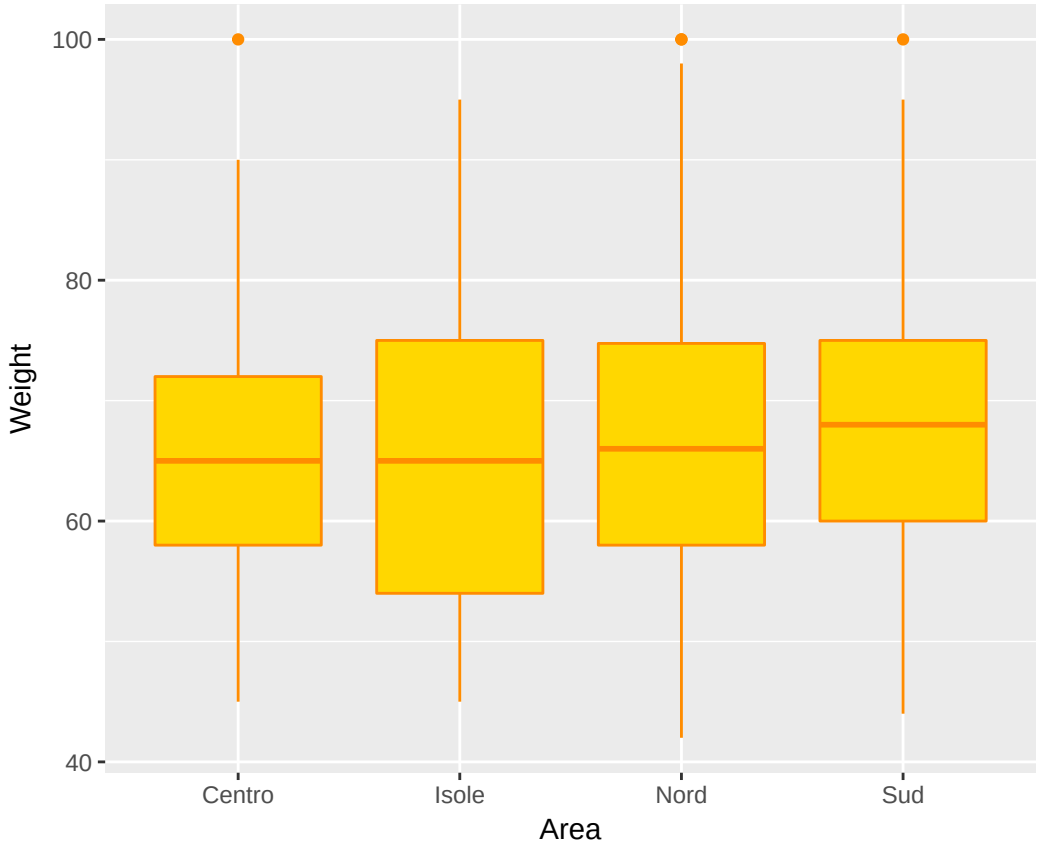


```
# customized plot: key building blocks + scales + facet + theme
ggplot(data=people, mapping=aes(x=Weight)) +
  geom_histogram(mapping=aes(fill=Area), binwidth=5, colour="black") +
  scale_fill_manual(values = c("#70D6FF", "#FF70A6", "#FF9770", "#E9FF70")) +
  facet_wrap( ~ Area) +
  theme(axis.text = element_text(colour = "black"),
    axis.ticks = element_line(colour = "black"),
    axis.title = element_text(colour = "black", size = 14, face = "bold.italic"),
    strip.background = element_rect(colour = "black", fill=),
    strip.text = element_text(colour = "black", face = "bold.italic", size = 12),
    plot.title = element_text(colour = "black", size = 20, face = "bold.italic"),
    legend.position = "none")
```

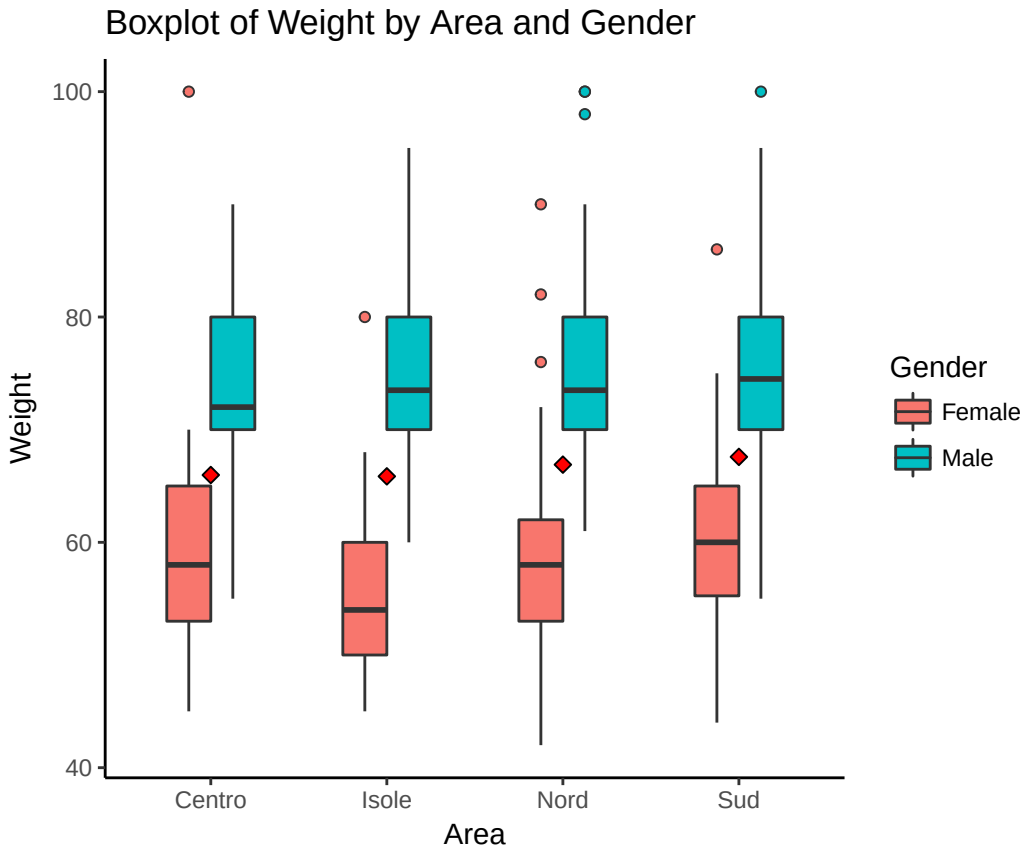


- **Boxplot**, which is used to draw a data distribution. Supposing you are interested in the differences of weight accordingly to geographical area:

```
# base plot: key building blocks (data, aes, layer)
ggplot(data=people, aes(x=Area, y=Weight)) +
  geom_boxplot(fill="gold", colour="darkorange")
```



```
# customized plot: key building blocks + scale + layer (stat) + theme
ggplot(data=people, aes(x=Area, y=Weight, fill=Gender)) +
  geom_boxplot(outlier.size = 1.5, outlier.shape = 21, width = .5) +
  stat_summary(fun.y = "mean", geom = "point", shape = 23, size = 2, fill = "red") +
  ggtitle("Boxplot of Weight by Area and Gender") +
  theme_classic()
```



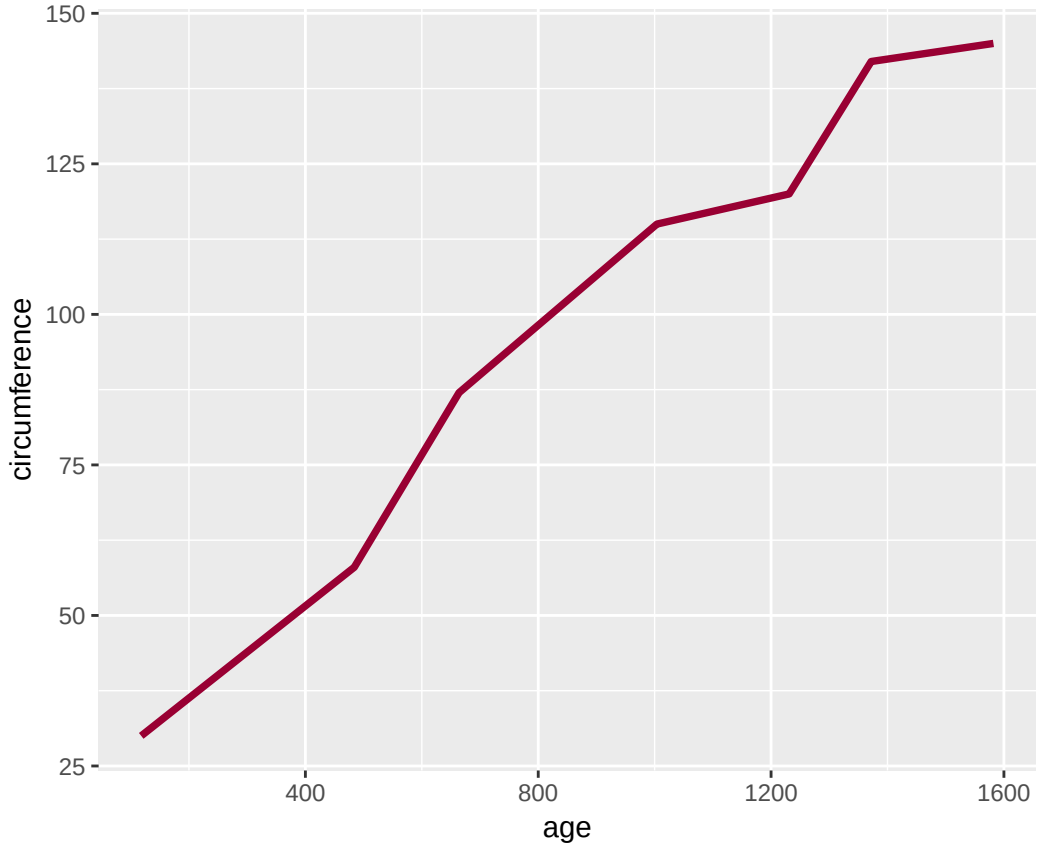
- **Lineplot**, used to display how one continuous variable, on the y-axis, changes in relation to another continuous variable, on the x-axis. For this example we consider `orange` data, included in `qdata` package. `orange` contains information about the growth of 5 Orange Trees, according to their trunk circumferences.

```
data(orange)
head(orange)
```

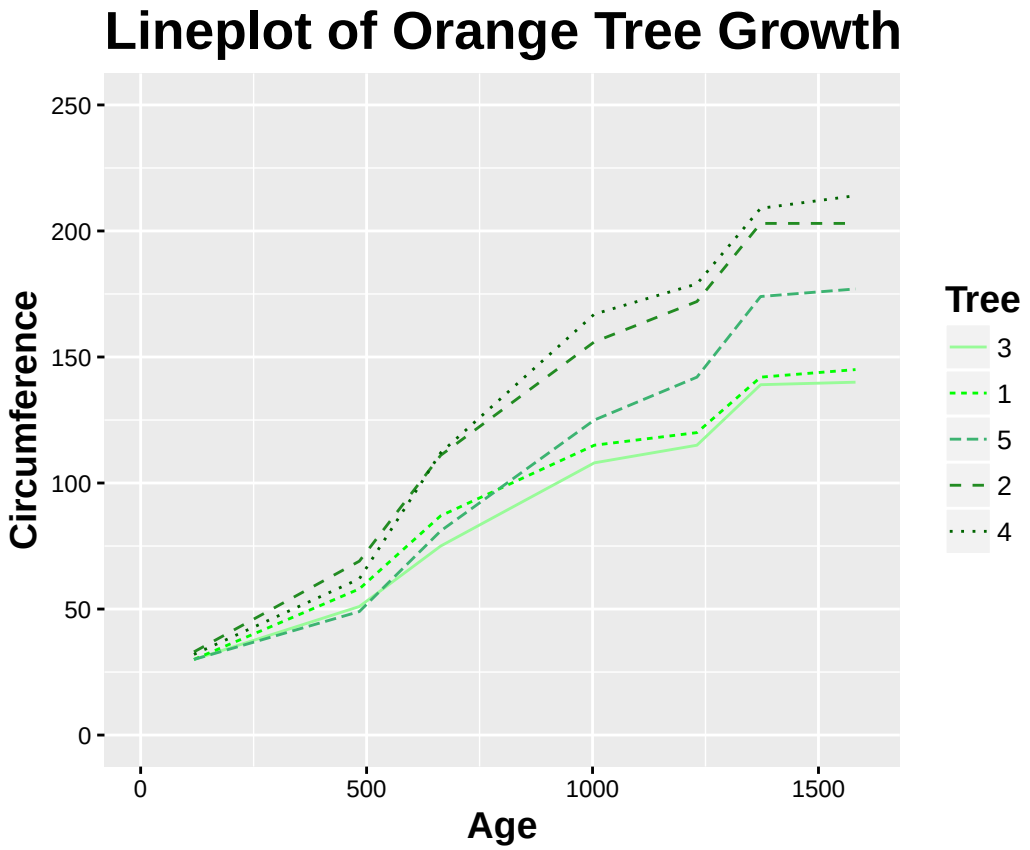
```
##   Tree  age circumference
## 1    1  118            30
## 2    1  484            58
## 3    1  664            87
## 4    1 1004           115
## 5    1 1231           120
## 6    1 1372           142
```

Suppose we want to represent the growth of one tree first and then of all 5 trees:

```
require(dplyr)
# base plot of 1 tree: key components(data, aes, layer)
ggplot(data=orange %>% filter(Tree==1), mapping=aes(x=age, y=circumference)) +
  geom_line(colour= "#990033", size=1.3)
```



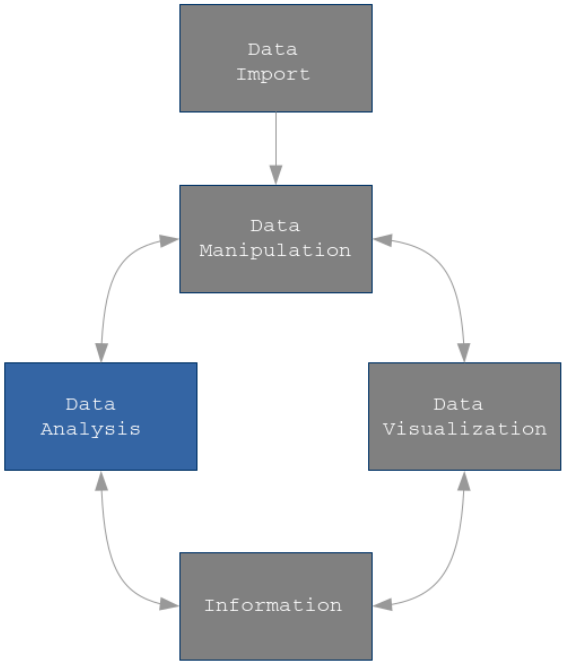
```
# customized plot of 5 trees: key components(data, aes, layer) + scales + theme
ggplot(data=Orange, mapping=aes(x=age, y=circumference, colour=Tree)) +
  geom_line(mapping=aes(linetype=Tree)) +
  scale_colour_manual(values = c("palegreen", "green", "mediumseagreen", "forestgreen" ,"darkgreen")) +
  ylim(0,250) + xlim(0,1600) +
  ggtitle("Lineplot of Orange Tree Growth") +
  xlab("Age") + ylab("Circumference") +
  theme(axis.text = element_text(colour = "black"),
  axis.ticks = element_line(colour = "black"),
  axis.title = element_text(colour = "black", size = 14, face = "bold"),
  plot.title = element_text(colour = "black", size = 20, face = "bold"),
  legend.text = element_text(colour="black", size=10),
  legend.title = element_text(colour = "black", size = 14, face = "bold"))
```



Chapter 11

Models with R

```
require(qdata)
require(ggplot2)
require(dplyr)
```



Data can be analysed by regression models with R. Regression is an analysis that attempts to determine the strength of the relationship between one dependent variable, y, and a series of other changing variables, x.

There are lots of different types of regression models in statistics but R mantains a coherent syntax for the estimation of all of them. Indeed, the common interface to fit a model in R is made of a call to the corresponding function with arguments `formula` and `data`.

The `lm` and `aov` functions are used in R to fit respectively linear regression and analysis of variance model and their syntax is:

```
linear_model <- lm(formula, data)
anova_model <- aov(formula, data)
```

`formula` argument is a symbolic description of the model to be fitted, which has the form: `response variable predictor variables`. The variables involved in `formula` should be columns of a dataframe specied in `data` argument.

The resulting object (`linear_model` or `anova_model`) is a list of elements containing information about regression results. This information can be investigated by the following functions:

Expression	Description
`coef(obj)`	regression coefficients
`resid(obj)`	residuals
`fitted(obj)`	fitted values
`summary(obj)`	analysis summary
`predict(obj, newdata = ndat)`	predict for newdata
`deviance(obj)`	residual sum of squares

Let us see some examples of regression analisys.

11.0.1 Drug Dosage and Reaction Time

In an experiment to investigate the effect of a depressant drug, the reaction times of ten males rats to a certain stimulus were measured after a specified dose of the drug had been administer to each rat. The results were as follows:

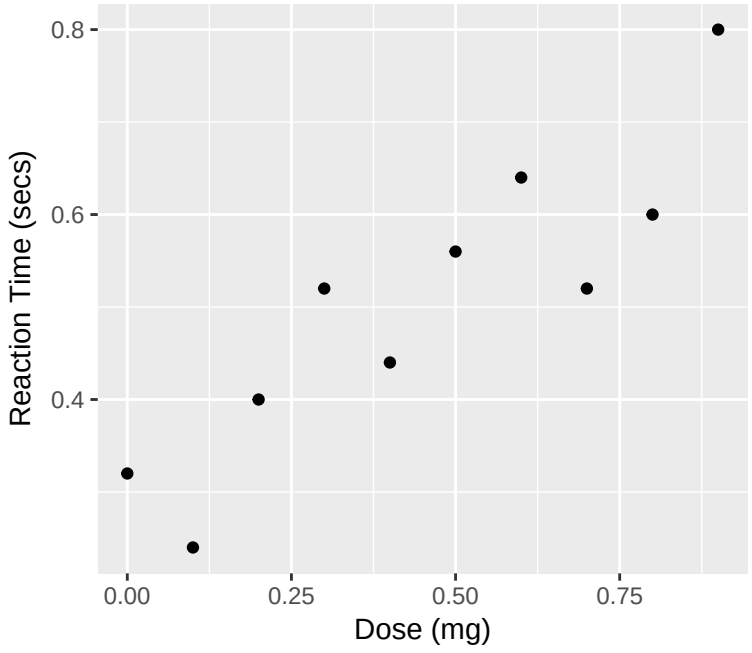
```
data(drug)
str(drug)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    10 obs. of  3 variables:
##  $ rat : int  1 2 3 4 5 6 7 8 9 10
##  $ dose: num  0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
##  $ time: num  0.32 0.24 0.4 0.52 0.44 0.56 0.64 0.52 0.6 0.8
```

Basic graphical data exploration may be achieved with:

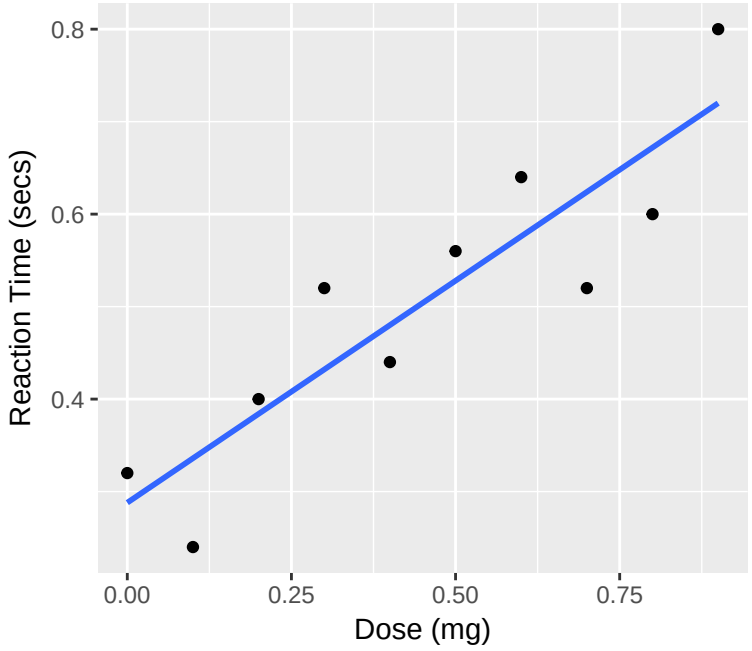
```
pl_1 <- ggplot(data = drug, mapping = aes(x = dose, y=time)) +
  geom_point() +
  xlab(label = "Dose (mg)") +
  ylab(label = "Reaction Time (secs)")

pl_1
```



A simple model for these data might be a straight line, which can be easy superposed to the data scatterplot by:

```
pl_1 + geom_smooth(method="lm", se=FALSE)
```



The R command to fit a simple linear model is:

```
fm <- lm(formula = time ~ dose, data = drug)
```

As we said, `fm` object is a list of elements containing information about regression results.

Regression results can be investigated by generic function `summary()`, which includes the most important elements for model interpretation.

```
summary(fm)
```

```
##
## Call:
## lm(formula = time ~ dose, data = drug)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -0.104 -0.064  0.024  0.056  0.088
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.28800    0.04522   6.368 0.000216 ***
## dose         0.48000    0.08471   5.666 0.000472 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.07694 on 8 degrees of freedom
## Multiple R-squared:  0.8005, Adjusted R-squared:  0.7756
## F-statistic: 32.11 on 1 and 8 DF,  p-value: 0.0004724
```

Small p-values for t-test on coefficients lead to consider both coefficients as significant. Adjust R-squared equal to 0.78 shows an acceptable portion of total variation explained by the regression model.

Small p-value for F-statistic confirm fitted model significance when compared with null model (model only with intercept).

You can also visualize single elements of `fm` list.
For example:

```
# regression coefficients
coef(fm)
```

```
## (Intercept)        dose
##       0.288       0.480
```

```
# residuals
resid(fm)
```

```
##      1      2      3      4      5      6      7      8      9     10
##  0.032 -0.096  0.016  0.088 -0.040  0.032  0.064 -0.104 -0.072  0.080
```

```
# fitted values
fitted(fm)
```

```
##     1     2     3     4     5     6     7     8     9    10
## 0.288 0.336 0.384 0.432 0.480 0.528 0.576 0.624 0.672 0.720
```

Prediction for response variable at specific values of the explanatories variable can be gained using `predict()` function with the following sintax:

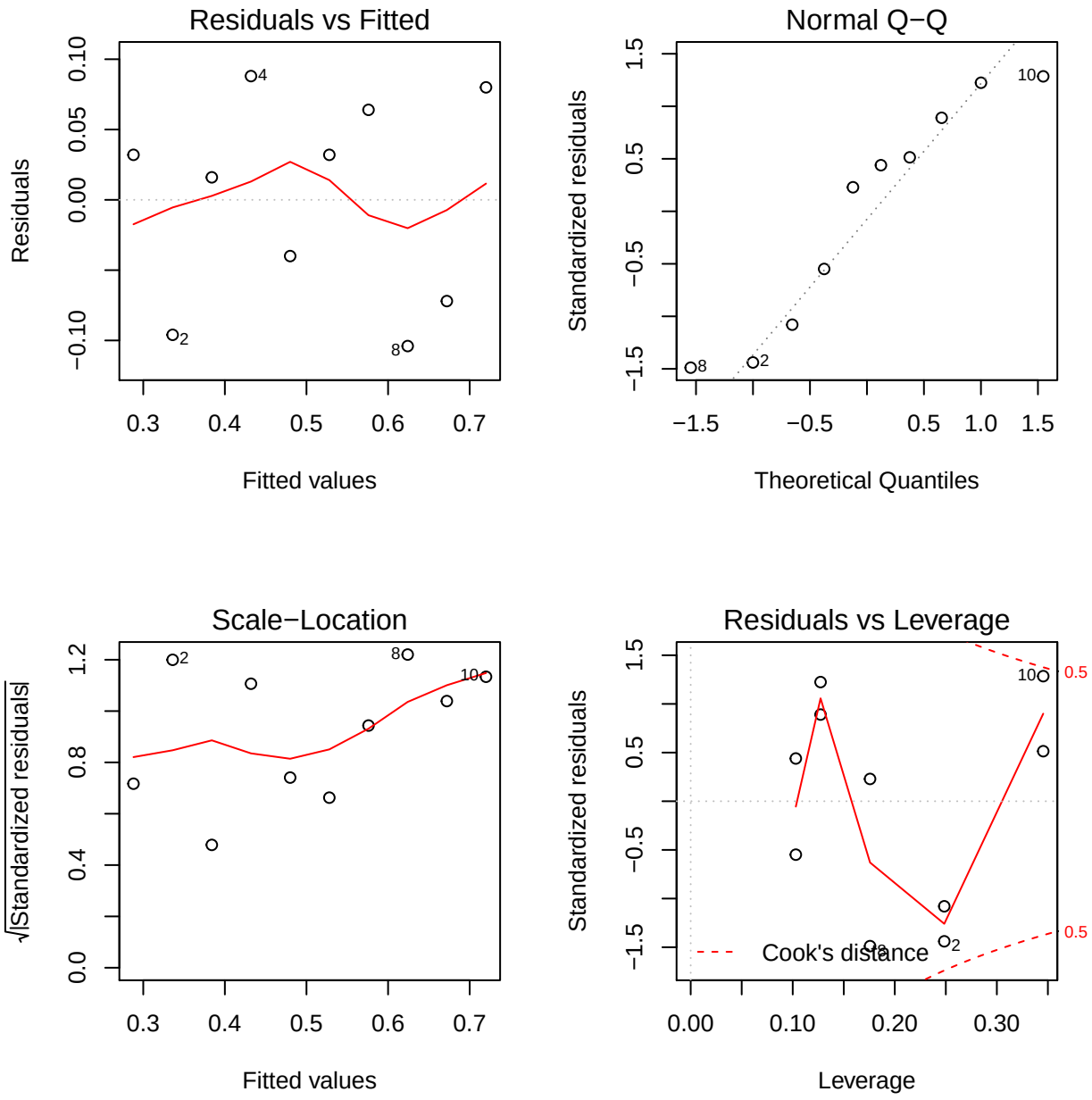
```
newdata <- data.frame(dose = c(0.2, 0.4, 0.6 ))
predict (fm , newdata = newdata)
```

```
##     1     2     3
## 0.384 0.480 0.576
```

Note how the `newdata` argument within `predict()` requires an object of class `data.frame`, whose column names have to be equal to the explicative variables names of the model.

For a visual inspection of the suite of residuals plots of the model, the sintax is:

```
par(mfrow = c(2,2))
plot(fm)
```



The "Residuals vs Fitted" plot does not show, in this example, any particular pattern. The presence of patterns may suggest that the model is inadequate. The "Normal Q-Q" plot shows points close to the straight line. If the normal assumption of residuals is not satisfied, points are far from the straight line. The "Normal Q-Q plot" is less reliable on the distribution tails, i.e. points ought be very far from the straight line to suggest that residuals follow a non-normal distribution. The "Scale location" is similar to the residuals versus fitted values plot, but it uses the square root of the standardized residuals in order to diminish skewness. Like the first plot, there should be no discernable pattern to the plot. The "Residuals vs Leverage" shows leverage points. Leverage points are those observations, if any, made at extreme or outlying values of the independent variables such that the lack of neighboring observations means that the fitted regression model will pass close to that particular observation. Leverage points fall out the dotted lines.

11.0.2 Car Seat

Three quality inspectors are studying the reproducibility of a measurement method aiming to test resistance of a specific material used to cover car seats. As a result, an experiments involving 75 samples of material from the same batch is set up. Three operators: Kevin, Michelle and Rob are assigned to test 25 samples each.

Comparison of operators average measurements and within operators variations are the key points of the analysis.

```
data(carseat)
str(carseat)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    75 obs. of  2 variables:
##  $ Operator: Factor w/ 3 levels "Kevin","Michelle",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ Strength: num  11.3 10.6 10.4 10.2 10.4 ...
```

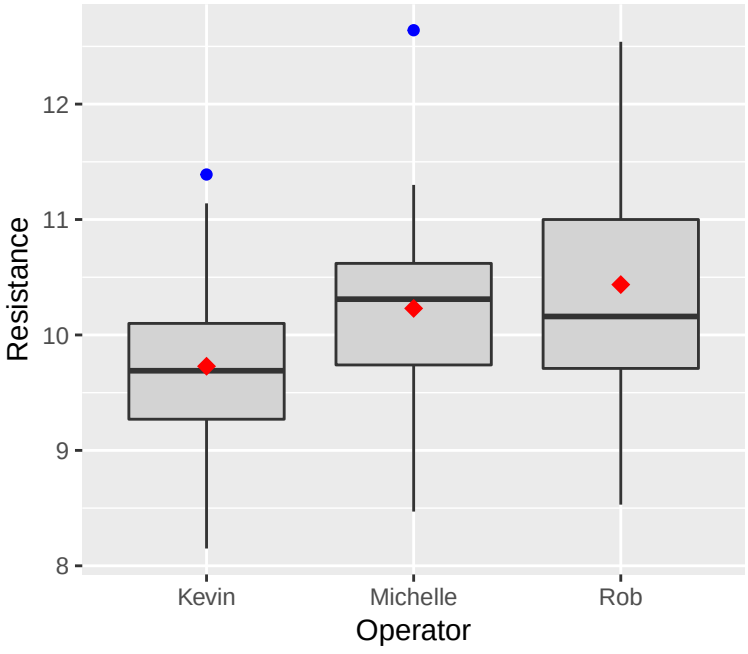
Boxplot of *Strength by Operator* along with within operators averages can be display by:

```
require(dplyr)
```

```
# Data frame of Strength means for each Operator
means_df <- carseat %>% group_by(Operator) %>% summarize(mean=mean(Strength))

pl_2 <- ggplot(data = carseat, mapping = aes(x=Operator, y=Strength)) +
  geom_boxplot(fill = "lightgray", outlier.colour = "blue") +
  geom_point(data=means_df, mapping = aes(x=Operator, y=mean), colour="red", shape=18, size=3) +
  xlab(label = "Operator") + ylab(label = "Resistance")

pl_2
```



At first glance, difference between operators is hard to detect as within variation seems to be quite large.

A simple one way analysis of variance model is computed with:

```
fm <- aov(Strength~Operator, data = carseat)
```

Results are shown with the usual `summary()` method.

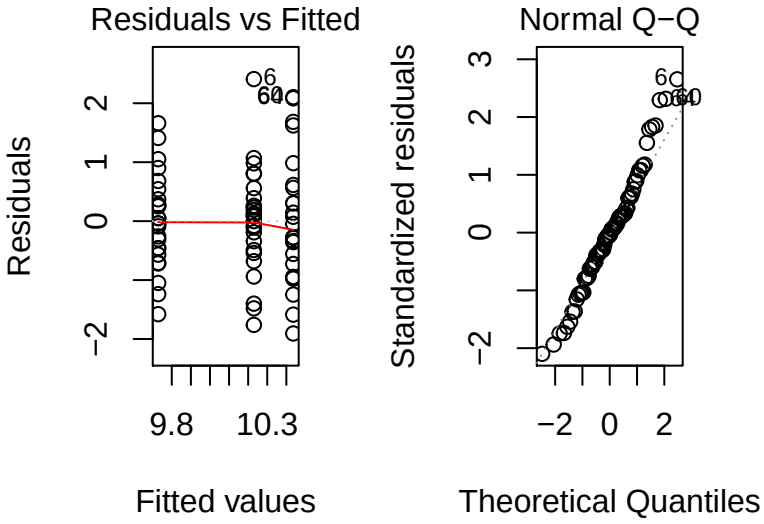
```
summary(fm)
```

```
##             Df Sum Sq Mean Sq F value Pr(>F)
## Operator     2   6.62    3.31   3.851 0.0258 *
## Residuals   72  61.90    0.86
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Operators appears to be borderline significant as its F test p-value is 0.0258. That is, not a real difference exists between operators measurements methods.

Finally, normality of residuals can be checked by:

```
par(mfrow = c(1,2))
plot(fm, which = 1:2)
```



Despite a small departure from normality on the rigth side of the residuals distribution, model assumptions seem to confirm.

11.0.3 Boiling Time

In an attempt to resolve a domestic dispute about which of two pans was the quicker pan for cooking, the following data were obtained:

```
data(boiling)
str(boiling)
```

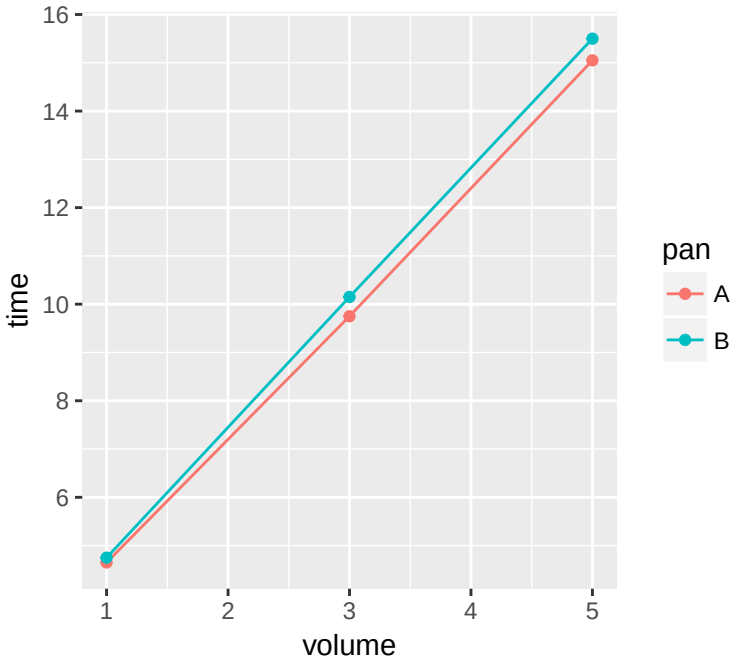
```
## Classes 'tbl_df', 'tbl' and 'data.frame':    6 obs. of  3 variables:
##  $ time  : num  4.65 9.75 15.05 4.75 10.15 ...
##  $ pan   : Factor w/ 2 levels "A","B": 1 1 1 2 2 2
##  $ volume: num  1 3 5 1 3 5
```

Various measured volumes (pints) of cold water were put into each pan and heated using the same setting of the cooker. The response variable was the time in minutes until the water boiled.

From a first visual investigation, it's easy to understand that differences in boling time means difference either in intercept or slope between the two lines:

```
pl_3 <- ggplot(data = boiling, mapping = aes(x=volume, y=time, colour=pan)) +
  geom_line() + geom_point()

pl_3
```



The natural candidate for this problem is the Gamma distibution as it doesn't assume negative value (boiling time cannot be negative) and it respect the increasing relationship between mean and variance.

A generalized linear model with family Gamma and link identity can be used:

```
fm_5 <- glm(time ~ pan*volume, data = boiling,
  family = Gamma(link = "identity"))
summary(fm_5)
```

```
##
## Call:
## glm(formula = time ~ pan * volume, family = Gamma(link = "identity"),
##     data = boiling)
```

```
##
## Deviance Residuals:
##          1           2           3           4           5           6
##  0.0014711  -0.0062328   0.0047405  -0.0003524   0.0015024  -0.0011512
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.0592378  0.0394407  52.211 0.000367 ***
## panB        0.0008908  0.0565096   0.016 0.988854
## volume      2.5839284  0.0194681 132.726 5.68e-05 ***
## panB:volume 0.1076174  0.0279858   3.845 0.061457 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Gamma family taken to be 3.355215e-05)
##
##     Null deviance: 1.3170e+00  on 5  degrees of freedom
## Residual deviance: 6.7191e-05  on 2  degrees of freedom
## AIC: -15.087
##
## Number of Fisher Scoring iterations: 3
```