A Learning Guide to R

Data, analytical, and programming skills

Remko Duursma

A Learning Guide to R: data, analytical, and programming skills.

Remko Duursma

2020-03-03

Contents

Preface **5**
About the author 5

1 Beginner skills **7**
1.1 Installing R and Rstudio 7
1.2 Basic operations 8
1.3 Working with vectors 9
1.4 Writing code in a script 10
1.5 Vectorized operations 11
1.6 Working with matrices 12
1.7 Objects in the workspace 14
1.8 Files in the working directory 14
1.9 Rstudio projects 15
1.10 Packages 15
1.11 Accessing the help files 18
1.12 Writing lots of code: rmarkdown or scripts? 19
1.13 Pay attention to detail: write clean code 21
1.14 Exercises 21

2 Data skills - Part 1 **27**
2.1 Introduction 27
2.2 Generating data 28
2.3 Reading data 29
2.4 Working with dataframes 32
2.5 Extracting data 35
2.6 Exporting data 42
2.7 Special data types 43
2.8 Exercises 62

3 Data skills - Part 2 **67**
3.1 Summarizing dataframes 67
3.2 Combining dataframes 80
3.3 Reshaping data 87
3.4 More complex `dplyr` examples 90
3.5 Exercises 93

4 Functions, lists and loops **97**
4.1 Introduction 97
4.2 Writing functions 97
4.3 Working with lists 99

4.4 Loops . . . 109
4.5 Functions : advanced concepts . . . 111
4.6 Robust functions: defensive programming . . . 117
4.7 Exercises . . . 122

5 R on the web 125
5.1 Introduction . . . 125
5.2 Reading data from remote databases . . . 125
5.3 Accessing an API . . . 127
5.4 Deploying an API with `plumber` . . . 128

6 Version control with git 131
6.1 Should I learn version control? . . . 131
6.2 Basics of git . . . 131
6.3 Local git repositories . . . 134
6.4 Remote git repositories . . . 138
6.5 Ignoring files . . . 142
6.6 Using the command line . . . 143

7 Project management and workflow 147
7.1 Tips on organizing your code . . . 147
7.2 Set up a project in Rstudio . . . 148
7.3 Directory structure . . . 149
7.4 The R scripts . . . 150
7.5 Archiving the output . . . 153
7.6 A logical structure for your scripts . . . 154

8 Mastering package dependencies 157
8.1 What version? . . . 157
8.2 All packages: loaded or installed . . . 158
8.3 Avoiding conflicts . . . 158
8.4 The search path and conflicts . . . 158
8.5 Bundling package dependencies with `renv` . . . 159
8.6 Minimizing package dependencies . . . 160

Preface

About the author

Remko Duursma is a Data Scientist and R Trainer at Shinto Labs. Previously, he was a scientist in the field of forestry and plant ecology, and studied the effects of climate change with the use of data and models. He has published nearly 70 articles in the scientific literature, and more than ten R packages.

You can contact Remko here:

- Email: remkoduursma@gmail.com
- LinkedIn: https://www.linkedin.com/in/remkoduursma/
- www.remkoduursma.com

Chapter 1

Beginner skills

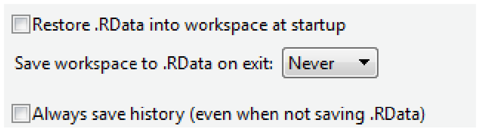
1.1 Installing R and Rstudio

We assume you have installed R and Rstudio. We occasionally give some tips on how to use Rstudio effectively, but since Rstudio is likely to change more quickly than R - we instead suggest you keep an eye on their documentation (link).

We assume that you have installed both R, and Rstudio. Please visit both websites (here for R, here for Rstudio) to download the latest version for your platform. Note that Rstudio simply runs R for us (and provides many, many other features) - you still need to install R itself.

Rstudio settings

We strongly recommend you change the following default settings in Rstudio. The default behaviour is to save all your objects to an 'RData' file when you exit, and loads the same objects when you open RStudio. This is very dangerous behaviour, and you **must** turn it off. For now, make sure you go to Tools > Global Options... and on the General tab, make sure the settings are like the figure below.

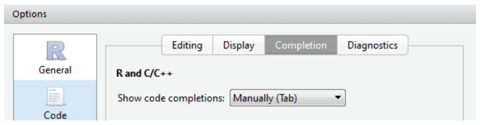


☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

☐ Always save history (even when not saving .RData)

Another feature you may want to turn off is the automatic code completion, which is now a standard feature in RStudio. This is a matter of taste, but we find it handier to use code completion only when requested. If you change the settings as shown in the screenshot below, you can engage code completion by typing part of a function and then pressing `Tab`.



R and C/C++

Show code completions: Manually (Tab)

Further reading This text is not a complete or even comprehensive guide to Rstudio - we focus on R. If you want to know more about all the options in the menu's, keyboard shortcuts, and various add-ons and features, please visit this link to get started.

1.1.1 Example data used throughout this book

All example datasets used throughout this book are provided in the R package `lgrdata`. Install it like any other package (see 1.10 for more information):

```
install.packages("lgrdata")
```

At the start of each script where you use an example dataset, load the package:

```
library(lgrdata)
```

Then, you can make available any dataset from the package with :

```
data(allometry)
```

Each dataset has help page, inspect it (`?allometry`) to learn a bit about the dataset, the meaning and units of the variables, and sometimes a simple example plot with the data.

Try this yourself To look at all datasets included in the `lgrdata` package, it is most convenient to use Rstudio's package explorer, under the `Packages` tab in the bottom-right hand menu. Just find the package and click on it.

1.2 Basic operations

In Rstudio, the R console is shown in one of the windows. The other windows include the 'Environment' window (with tabs 'Environment', 'History', etc.), a 'File management' window (where you can browse files, view plots, install packages, etc.), and a source window.

The source window (containing a script where you can write and save code) may not show up on your first use of Rstudio, press Ctrl-Shift-N to open a new script. We return to scripts in Section 1.12.2.

In the console, you can type R code and get immediate results, for example:

```
# I want to add two numbers:
1 + 1
```

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

Here, we typed `1 + 1`, hit Enter, and R produced `2`. The `[1]` means that the result only has one element (the number '2').

In this book, the R output is shown after `##`. Every example can be run by you, simply copy the section (use the text selection tool in Adobe reader), and paste it into the console (with Ctrl + Enter on a Windows machine, or Cmd + Enter on a Mac).

We can do all sorts of basic calculator operations. Consider the following examples:

```
# Arithmetic
12 * (10 + 1)
```

```
## [1] 132
```

```
# Scientific notation
3.5E03 + 4E-01
```

```
## [1] 3500.4
```

```r
# pi is a built-in constant
sin(pi/2)
```

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

```r
# Absolute value
abs(-10)
```

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

```r
# Yes, you can divide by zero
1001/0
```

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

```r
# Square root
sqrt(225)
```

```
## [1] 15
```

```r
# Exponents
15^2
```

```
## [1] 225
```

```r
# Round down to nearest integer (and ceiling() for up or round() for closest)
floor(3.1415)
```

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

Try typing `?Math` for description of more mathematical functions.

Also note the use of # for comments: anything after this symbol on the same line is *not* read by R.

Try this yourself When typing code directly into the console (other than a script, which is what you usually will do when developing more serious code), consider these tips:

- Press **UP** on your keyboard to repeat the previous command (and keep pressing UP to find all previous commands).
- Press **TAB** to auto-complete the name of a function (even those from loaded packages); very handy if you don't remember the name of the function, or you can't be bothered typing it in.
- Press **CTRL-L** (CMD-L on Mac) to 'clear' the console. This does not remove the history or any objects, just starts a fresh screen.
- Press **ESC** to cancel an operation (if R is taking long, and you don't want to wait).

1.3 Working with vectors

A very useful type of object is the `vector`, which is basically a string of numbers or bits of text (but not a combination of both). The power of R is that most functions can use a vector directly as input, which greatly simplifies coding in many applications.

Let's construct an example vector with 7 numbers:

```r
nums1 <- c(1,4,2,8,11,100,8)
```

We can now do basic arithmetic with this *numeric vector* :

```
# Get the sum of a vector:
sum(nums1)
```

```
## [1] 134
```

```
# Get mean, standard deviation, number of observations (length):
mean(nums1)
```

```
## [1] 19.14286
```

```
sd(nums1)
```

```
## [1] 35.83494
```

```
length(nums1)
```

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

```
# Some functions result in a new vector, for example:
rev(nums1)  # reverse elements
```

```
## [1]   8 100  11   8   2   4   1
```

```
cumsum(nums1)  # cumulative sum
```

```
## [1]   1   5   7  15  26 126 134
```

There are many more functions you can use directly on vectors. See the table below for a few useful ones.

	Function	What it does	Example
1	`length`	Length of the vector	`length(nums1)`
2	`rev`	Reverses the elements of a vector	`rev(nums1)`
3	`sort`	Sorts the elements of a vector	`sort(nums1, decreasing = TRUE)`
4	`order`	The order of elements in a vector	`order(nums1)`
5	`head`	The first few elements of a vector	`head(nums1, 5)`
6	`max`	The maximum value	`max(nums1)`
7	`min`	The minimum value	`min(nums1)`
8	`which.max`	Which element of the vector is the max?	`which.max(nums1)`
9	`which.min`	Which element of the vector is the min?	`which.min(nums1)`
10	`mean`	The average value	`mean(nums1)`
11	`median`	The median	`median(nums1)`
12	`var`	Variance	`var(nums1)`
13	`sd`	Standard deviation	`sd(nums1)`
14	`cumsum`	Cumulative sum (running total)	`cumsum(nums1)`
15	`diff`	Successive difference of a vector	`diff(nums1)`
16	`unique`	Unique values used in the vector	`unique(nums1)`
17	`round`	Rounds numbers to a specified number of decimal points	`round(nums1, 2)`

Table 1.1: A selection of useful built-in functions in R.

1.4 Writing code in a script

To continue, we are first going to open a script - simply a text file where you can type your code, and execute it immediately. To do so, click on the menu File/New File/R Script.

You now have an empty R script, where you can write code, and add comments.

A single line can be run by placing the cursor on that line, and clicking 'Run' in the menu just to the top-right of the window. Alternatively (and this is **recommended**), use the keyboard shortcut Ctrl-Enter (or Cmd-Enter on Mac) to run that line. If you make a text selection with the mouse, this selection can be executed with the same keyboard shortcut.

1.5 Vectorized operations

In the above section, we introduced a number of functions that you can use to do calculations on a vector of numbers. In R, a number of operations can be done on two vectors, and the result is a vector itself. Basically, R knows to apply these operations one element at a time. This is best illustrated by some examples:

```
# Make two vectors,
vec1 <- c(1,2,3,4,5)
vec2 <- c(11,12,13,14,15)

# Add a number, element-wise
vec1 + 10
```

```
## [1] 11 12 13 14 15
```

```
# Element-wise quadratic:
vec1^2
```

```
## [1]  1  4  9 16 25
```

```
# Pair-wise multiplication:
vec1 * vec2
```

```
## [1] 11 24 39 56 75
```

```
# Pair-wise sum:
vec1 + vec2
```

```
## [1] 12 14 16 18 20
```

```
# Compare the pair-wise sum to the sum of both vectors:
sum(vec1) + sum(vec2)
```

```
## [1] 80
```

In each of the above examples, the operators (like + and so on) 'know' to make the calculations one element at a time (if one vector), or pair-wise (when two vectors). Clearly, for all examples where two vectors were used, the two vectors need to be the same length (i.e., have the same number of elements).

1.5.1 Applying multiple functions at once

In R, we can apply functions and operators in combination. In the following examples, the *innermost* expressions are always evaluated first. This will make sense after some examples:

```
# Mean of the vector 'vec1', *after* squaring the values:
mean(vec1^2)
```

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

```r
# Mean of the vector, *then* square it:
mean(vec1)^2
```

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

```r
# Mean of the log of vec2:
mean(log(vec2))
```

```
## [1] 2.558972
```

```r
# Log of the mean of vec2:
log(mean(vec2))
```

```
## [1] 2.564949
```

Alternatively, we can use the `pipe` operator, which allows us to write the operators in order of application. For the pipe operator, we need to load the `magrittr` package (see 1.10 on how to install and load packages).

```r
library(magrittr)

# Identical to mean(log(vec2))
vec2 %>% log %>% mean
```

```
## [1] 2.558972
```

The above code can be written in at least two other equivalent ways: `log(vec2) %>% mean`, and even `log(vec2) %>% mean(.)`, where the `.` indicates explicitly the result from the previous operator.

In any of the following, we assume you have loaded either `magrittr`, or some package that loads it for you (for example, `dplyr`). If you have not, you will see the error message `could not find function "%>%"`.

The pipe operator has become very popular, and we use it when it greatly simplifies code, making it easier to follow what the code actually does (and in which order).

1.6 Working with matrices

Vectors are one-dimensional collections of data; all elements are of the same type (numeric, character, etc.). A matrix has two dimensions: rows and columns, and again each element (each cell) has the same type. We do not use matrices all that often in R, since a more general data storage type (the dataframe) is generally more useful (there, each column can contain different kind of data). However, sometimes working with matrices is faster, or built-in functions return a matrix - some basic skills are useful.

```r
# Construct a matrix with the matrix function:
mymat <- matrix(c(1,2,3,4,5,6,7,8,9), ncol=3)

# Note how the values are entered in the matrix column-by-column:
mymat
```

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

```
# Alternatively, add the values row-by-row:
mymat2 <- matrix(c(1,2,3,4,5,6,7,8,9), ncol=3,  byrow=TRUE)

# Note the difference:
mymat2
```

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

A few built-in functions are especially handy when dealing with a matrix:

```
# Transpose a matrix (flip it alongside the diagonal)
t(mymat)
```

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

```
# Row or column-wise sums
rowSums(mymat)
```

```
## [1] 12 15 18
```

```
# Row or column-wise means
colMeans(mymat)
```

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

```
# Extract the diagonal:
diag(mymat)
```

```
## [1] 1 5 9
```

Functions usually used for vectors often work with matrices, all values are simply treated as if stored in a vector:

```
# Standard deviation
sd(mymat)
```

```
## [1] 2.738613
```

```
# Number of observations
length(mymat)
```

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

```
# Is the same as number of columns * number of rows
nrow(mymat) * ncol(mymat)
```

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

As with vectors, we can calculate directly on all elements:

```
mymat * 10
```

```
##      [,1] [,2] [,3]
## [1,]   10   40   70
## [2,]   20   50   80
## [3,]   30   60   90
```

1.7 Objects in the workspace

In the examples above, we have created a few new objects. These objects are kept in memory for the remainder of your session (that is, until you close R).

In RStudio, you can browse all objects that are currently loaded in memory. Objects that are currently loaded in memory make up your *workspace*. Find the window that has the tab 'Environment'. Here you see a list of all the objects you have created in this R session. When you click on an object, a window opens that shows the contents of the object.

Alternatively, to see which objects you currently have in your workspace, use the following command:

```
ls()
```

```
## [1] "nums1"   "vec1"    "vec2"    "words"   "x"       "y"       "a"
## [8] "b"       "numbers"
```

To remove objects,

```
rm(nums1, nums2)
```

And to remove all objects that are currently loaded, use this command. **Note:** you want to use this wisely!

```
rm(list=ls())
```

Finally, when you are ending your R session, but you want to continue exactly at this point the next time, make sure to save the current workspace. In RStudio, find the menu `Session` (at the top). Here you can save the current workspace, and also load a previously saved one.

1.8 Files in the working directory

Each time you run R, it 'sees' one of your folders ('directories') and all the files in it. This folder is called the *working directory*. You can change the working directory in RStudio in a couple of ways.

The first option, in Rstudio, is to go to the menu `Session > Set Working Directory > Choose Directory...`.

Or, find the 'Files' tab in one of the RStudio windows (usually bottom-right). Here you can browse to the directory you want to use (by clicking on the small `...` button on the right), and click `More > Set as working directory`.

You can also set or query the current working directory by typing the following code:

```
# Set working directory to C:/myR
setwd("C:/myR")

# What is the current working directory?
getwd()
```

Note: For Windows users, use a forward slash (that is, /), not a back slash!

Finally, you can see which files are available in the current working directory, as well as in subdirectories, using the `dir` function (*Note*: a synonym for this function is `list.files`).

```
# Show files in the working directory:
dir()

# List files in some other directory:
dir("c:/work/projects/data/")

# Show files in the subdirectory "data":
dir("data")

# Show files in the working directory that end in csv.
# (The ignore.case=TRUE assures that we find files that end in 'CSV' as well as 'csv',
# and the '[]' is necessary to find the '.' in '.csv'.
dir(pattern="[.]csv", ignore.case=TRUE)
```

1.9 Rstudio projects

If you end up working primarily from Rstudio, we very strongly recommend the use of Projects in Rstudio to keep your work organized, and to help set the working directory.

Rstudio projects are really just small files added to a certain folder, which "tags" that folder as containing a particular project. This idea works best if you have organized your work into smaller subsections, where you keep your different projects in separate folders.

1.10 Packages

Throughout this tutorial, we will focus on the basic functionality of R, but we will also call on a number of add-on 'packages' that include additional functions. These packages need to be *installed* (downloaded) before first use, and *loaded* every time you start a new R session.

1.10.1 Install packages from CRAN

You can find a full list of packages available for R on the CRAN site (http://cran.r-project.org/, navigate to 'Packages'), but be warned – it's a very long list, with over 10000 packages. If you really want to browse a long list of packages, you should prefer the website http://r-pkg.org/, which provides a nice interface.

In RStudio, click on the `Packages` tab in the lower-right hand panel. There you can see which packages are already installed, and you can install more by clicking on the `Install` button.

Alternatively, to install a particular package, simply type:

```
install.packages("gplots")
```

Caution Never use `install.packages` in an rmarkdown file (it simply does not work), and it is generally bad practice to add it to a script, since you do not want the packages to be installed every time the script is run. Instead, make it clear what the dependencies are in a README file, or use the approach outlined in "Loading many packages" further below, or in "Install missing packages".

The package needs to be installed only once (except when you need to update it, see Section 1.10.6).

1.10.2 Loading packages

To use the package in your current R session, type:

```
library(gplots)
```

This loads the `gplots` package from your local library (where all packages are stored) into working memory. You can now use the functions available in the `gplots` package. If you close and reopen R, you will need to load the package again (but you don't have to install it again).

To quickly learn which functions are included in a package, type:

```
library(help=gplots)
```

Try this yourself Type `.libPaths()` to see where your packages are installed. You can change the default directory permanently after setting the environment variable `R_LIBS` on your system.

1.10.3 Setting the CRAN mirror

CRAN is a central package for R packages, but it is hosted on many servers around the world. Back in the day, you always had to set the CRAN 'mirror' once for every R session. If you are using Rstudio, though, you won't have to worry about this as it is set automatically to Rstudio's own CRAN mirror, which is actually just a clever service that sends you to a nearby server.

If you use R from the command line, you might want to set the mirror directly:

```
# Not needed from Rstudio!
options(repos = c(CRAN = "https://cran.rstudio.com/"))
```

This avoids an annoying popup when attempting `install.packages`.

1.10.4 Install from git hosting sites

The previous section applies to packages that have been uploaded to CRAN (the central repository for R packages). It is however also quite common for R packages to be available on Github or Bitbucket - two large web services that host source code under *git* version control. For repositories that contain R packages, these cannot be installed via `install.packages`. Instead, do:

```
library(remotes)
install_bitbucket("remkoduursma/fitplc")
```

This installs an R package from www.bitbucket.org. Similarly, `install_github` installs the package from www.github.com, if it is hosted there.

For this command to work, your system needs to be able to compile R packages from source. Windows users will need to install Rtools.

1.10.5 Updating R and package locations

By default, `install.packages` places the packages in a subdirectory of your R installation, which means that when you update R, you have to reinstall all packages - unless you follow these instructions. To check where your packages are being installed to, type:

```
.libPaths()
```

You may see more than one path listed - packages will be installed to the first directory. It is generally a good idea to make a custom location for all your packages, for example under `c:/RLIBRARY`, or `~/rpackages`, or some other location independent of where R is installed.

To do this, you have to add an environment variable `R_LIBS`, that points to your custom directory. If you don't know how to do this, simply do a web search for 'add environment variable windows/mac'. The next time you open Rstudio, `.libPaths()` should show the custom directory.

1.10.6 Updating packages

Besides updating R and Rstudio itself (which is recommended at least every couple of months or so), it is also a very good idea to keep all packages up to date. If your workflow ends up depending on an older version of the package, your code may not work on a different system. To update all *installed* packages, you can run the following command - make sure to do this right after opening Rstudio:

```
update.packages(ask=FALSE)
```

It is usually best to type 'no' when the installation process asks if you want to compile some updated packages. Also, occasionally this process fails - usually stating e.g. 'package Rcpp not found'. Then first install those packages (the usual way), and retry `update.packages`.

1.10.7 Install missing packages

The `reinstallr` package can be used to install all packages that are 'mentioned' in any of the source files in your working directory. That is, all those that are loaded in an R script, rmarkdown file, and some others, using `library` or `require`. You can simply do,

```
library(reinstallr)
reinstallr()
```

1.10.8 Conflicts

It is fairly common that certain functions appear in multiple packages. For example, `summarize` is a function in `Hmisc` and in `dplyr`. If you have loaded `Hmisc` first, and then load `dplyr`, you will see (among others), the following message:

```
The following objects are masked from ‘package:Hmisc’:

    src, summarize
```

This means that if you now use `summarize()`, it will be used from the `dplyr` package. As a consequence of this, *package conflicts are resolved based on the order in which you load the packages*. This can be very confusing (and dangerous!). It is better to use the `::` operator, when you want to make sure to use a function from a certain package:

```r
# Use summarize from dplyr
dplyr::summarize(...)
```

In fact, it can be considered very good practice to *always* use the `::` (called a *namespace directive*) to make it clear in your code where the function is used from. We refrain from using the operator very often in this book though, as it makes the code a bit messy and longer.

1.10.9 Loading many packages

If you are using packages in a script file, it is usually considered a good idea to load any packages you will be using at the *start* of the script. Usually you will see a long list of `library(this)`, and `library(that)`, but we like to use the `pacman` package at the top of a script, like this:

```r
if(!require(pacman))install.packages("pacman")
pacman::p_load(gplots, geometry, rgl, remotes, svglite)
```

The first line checks whether the `pacman` package is available (`require` is like `library`, except it returns `TRUE` if the package could be loaded), and if not (`!`), it is installed. Next, `p_load` from `pacman` can be used to load many packages in one line, *installing the ones that are not available*. This approach is a very concise way to make your script reproducible.

1.11 Accessing the help files

Every function that you use in R has its own built-in help file. For example, to access the help file for the arithmetic mean, type `?mean`.

This opens up the help file in your default browser (but does not require an internet connection). Functions added by loading new packages also have built-in help files. For example, to read about the function `bandplot` in the `gplots` package, type:

```r
library(gplots)
?bandplot

# Alternatively:
?gplots::bandplot
```

Do not get overwhelmed when looking at the help files. Much of R's reputation for a steep learning curve has to do with the rather obscure and often confusing help files. A good tip for beginners is to *not read the help files*, but skip straight to the Example section at the bottom of the help file. The first line of the help file, which shows what kind of input, or arguments, the function takes, can also be helpful.

The built-in help files offer detailed documentation of built-in functions (and those offered by packages), but often you are looking for something, but don't quite know what exactly. This frustrating experience is well described by this quote, to R-help in 2006:

> "This is probably documented, but I cannot find the right words or expression for a search. My attempts failed."
>
> — Denis Chabot

Sometimes it is useful to search all help files (including all *installed* packages, loaded or not) for a keyword, for example `??ANOVA` will find everything that mentions ANOVAs.

The best advice, though, is simply to **just use Google**. As a search, simply type the question like you would ask someone else, for example "How to place legend outside plot with ggplot2?", "(Is there an) R package for 3D plots?". You usually end up on *Stackoverflow* for the best and most up-to-date answers (but be careful when reading very old posts!).

1.12 Writing lots of code: rmarkdown or scripts?

In this chapter you have written a little code directly to the Console, but for any more serious work you will use scripts or "rmarkdown" documents. In Chapter **??**, we will give more information on `rmarkdown` documents: an outstanding format to collect code, output, and text in a single, well formatted document. Scripts on the other hand only include code (and comments), and can be executed completely, or piece-by-piece by selecting code manually.

For both options, a brief summary is provided below. Overall, you want to use **scripts** if you:

- Want to be able to run all R code at once, producing some sort of output in files, or some other side-effect (perhaps figures as PDF, new datasets written to disk, objects produced and saved, data downloaded or uploaded, etc.).

On the other hand, you should use an **rmarkdown document** if you:

- Want to produce a document that includes all output from your analyses, including tables, figures, output from statistical analyses, and so on. The document can be well formatted and directly shared (hiding code, for example), and published on the web or simply emailed to your co-workers.

1.12.1 Reproducible documents with rmarkdown

- Documents are mix of normal text, with markup (headers, captions, images, tables, formatting), and blocks of code, **code chunks**.
- The document can be quickly converted to html, pdf, or an MS Word document, and includes output from code chunks (text output, figures), well formatted.
- Easy to learn syntax, and Rstudio provides many shortcuts and tools for working with `rmarkdown`.
- Output formats include HTML, PDF (if LaTeX is installed) and MS Word.
- Make not just simple documents, but also HTML presentations (in many formats), blogs (with `blogdown`), books (with `bookdown`), auto-generated websites documenting R packages (with `pkgdown`), and many more templates provided by add-on packages.

We give more information in Section **??**.

1.12.2 Using R scripts

Clearly, R markdown is the right solution if you are interested in writing fully reproducable reports or presentations. For everything else, we use R scripts - simply text files that include R code. A script is meant to be fully executable, so that you can run the entire script, producing the desired results, files, objects, or side-effects. In Rstudio, you can open a new R script with `File/New File/R script`, simply a text file that is saved with extension `.R`.

To run your script, you can either click 'Source' on the topright corner of the document, or execute it from R with:

```r
source("myscript.R")
```

This is *not useful* when you are interested in the output that you normally see in the console. If you are *printing* things to the screen in your script, the call to `source` will not show any of it. You can make these results visible by including `print` directly in your script, for example:

```r
# The output of this will not show up if you
# do source("somescript.R")
summary(mtcars)

# Here, the output will be printed to the screen,
# also with source()
print(summary(cars))
```

Although the trick with `print` works, usually you want to switch to `rmarkdown` if you are interested in viewing the outputs. Scripts are more useful for side effects, for example the creation of new datasets (that are saved to disk), production of figures as PDF files, up- or downloading of files, or even converting an rmarkdown file to HTML with various settings.

1.12.2.1 Running scripts from the command line

From outside R or Rstudio, you can run your script from the *command line* with `Rscript`, for example:

```
Rscript myscript.R
```

(Note that this only works on Windows if R is added to the search path).

You can also run bits of R code directly via,

```
Rscript -e "sqrt(9)"
```

1.12.2.2 Running scripts in the background

Suppose you have a script that takes a long time to complete. You want to run this script in a background process, so it does not tie up Rstudio for a long time. A nice feature in Rstudio (since late 2018) can be found on the Jobs tab, in the Console pane.

There, click 'Start Job', and select an R script you would like to run, and the working directory that should be used. The script will be run in the background, and the results will be shown in the Jobs tab. This way you can run an R script for every core you have available to you.

1.13 Pay attention to detail: write clean code

You may not want to spend the time making sure your code is neatly formatted, but it helps tremendously in a) spotting syntax errors while you write, and b) make your code vastly more readable by another person.

Unlike a language like Python, in R you are incredibly free to format your code (with spaces, tabs, newlines) pretty much however you want. Although this gives the programmer a lot of room for creative (and short) code, it can also be frustrating because it makes some code hard to read (when it is "poorly" formatted).

Further reading I find *Google's style guide* a very neat an readable way to indent and format R code: https://google.github.io/styleguide/Rguide.html

Try this yourself If you would like to use a tool that checks your code for various superficial (and some less superficial) problems, try using `lintr` : https://github.com/jimhester/lintr (I find it too restrictive and too opinionated, but you will certainly learn a lot by going through the process).

1.14 Exercises

1.14.1 Calculating

First inspect Section 1.2 and 1.3 ("Working with vectors").

Calculate the following quantities:

1. The sum of `100.1`, `234.9` and `12.01`

2. The square root of `256`

3. Calculate the 10-based logarithm of `100`, and multiply the result with the cosine of π. *Hint:* see `?log` and `?pi`.

4. Calculate the cumulative sum ('running total') of the numbers 2,3,4,5,6.

5. Calculate the cumulative sum of those numbers, but in reverse order. *Hint:* use the `rev` function.

6. Find 10 random numbers between 0 and 100, rounded to the nearest whole number (*Hint:* you can use either `sample` or a combination of `round` and `runif`).

1.14.2 Simple objects

Type the following code, which assigns numbers to objects `x` and `y`.

```
x <- 10
y <- 20
```

1. Calculate the product of `x` and `y`

2. Store the result in a new object called `z`

3. Inspect your workspace by typing `ls()`, and by clicking the `Environment` tab in Rstudio, and find the three objects you created.

4. Make a vector of the objects `x`, `y` and `z`. Use this command,

```
myvec <- c(x,y,z)
```

5. Find the minimum, maximum, length, and variance of `myvec`.

6. Remove the `myvec` object from your workspace.

1.14.3 Working with a single vector

If the last exercise was easy, you can skip this one.

1. The numbers below are the first ten days of rainfall amounts in 1996. Read them into a vector using the `c()` function.

```
 0.1  0.6 33.8  1.9  9.6  4.3 33.7  0.3  0.0  0.1
```

Inspect the table with functions in the Section 1.3 ("Working with vectors"), and answer the following questions:

2. Make a vector with the min, max and mean of rainfall. You can also name elements of a vector, for example `c(x = 1, y = 2)`.

3. Calculate the cumulative rainfall ('running total') over these ten days. Confirm that the last value of the vector that this produces is equal to the total sum of the rainfall.

4. Which day saw the highest rainfall (write code to get the answer)?

1.14.4 Scripts

This exercise will make sure you are able to make a 'reproducable script', that is, a script that will allow you to repeat an analysis without having to start over from scratch. First, set up an R script, and save it in your current working directory.

1. Find the `History` tab in Rstudio. Copy a few lines of history that you would like to keep to the script you just opened, by selecting the line with the mouse and clicking `To Source`.

2. Tidy up your R script by writing a few comments starting with `#`.

3. Now make sure your script works completely (that is, it is entirely *reproducible*). First clear the workspace (`rm(list=ls())` or click `Clear` from the `Environment` tab). Then, run the entire script (by clicking `Source` in the script window, top-right).

1.14.5 To quote or not to quote

This short exercise points out the use of quotes in R.

1. Run the following code, which makes two numeric objects.

```
one <- 1
two <- 2
```

2. Run the following two lines of code, and look at the resulting two vectors. The first line makes a character vector, the second line a numeric vector by recalling the objects you just constructed. Make sure you understand the difference.

```
vector1 <- c("one","two")
vector2 <- c(one, two)
```

3. The following lines of code contain some common errors that prevent them from being evaluated properly or result in error messages. Look at the code without running it and see if you can identify the errors and correct them all. Also execute the faulty code by copying and pasting the text into the console (not typing it, R studio will attempt to avoid these errors by default) so you get to know some common error messages (but not all of these result in errors!).

```
vector1 <- c('one', 'two', 'three', 'four, 'five', 'seven')

vec.var <- var(c(1, 3, 5, 3, 5, 1)
vec.mean <- mean(c(1, 3, 5, 3, 5, 1))

vec.Min <- Min(c(1, 3, 5, 3, 5, 1))

Vector2 <- c('a', 'b', 'f', 'g')
vector2
```

1.14.6 Working with two vectors

First make sure you understand Section 1.5.

1. You have measured five cylinders, their lengths are:

```
2.1, 3.4, 2.5, 2.7, 2.9
```

and the diameters are :

```
0.3, 0.5, 0.6, 0.9, 1.1
```

Read these data into two vectors (give the vectors appropriate names).

2. Calculate the correlation between lengths and diameters (use the `cor` function).

3. Calculate the volume of each cylinder (V = length * pi * (diameter / 2)^2).

4. Calculate the mean, standard deviation, and coefficient of variation of the volumes.

5. Assume your measurements are in centimetres. Recalculate the volumes so that their units are in cubic millimetres. Calculate the mean, standard deviation, and coefficient of variation of these new volumes.

1.14.7 Alphabet aerobics 1

For the second question, you need to know that the 26 letters of the Roman alphabet are conveniently accessible in R via `letters` and `LETTERS`. These are not functions, but vectors that are always loaded.

1. Using `c()` and `rep()` make this vector (in one line of code):

```
"A" "A" "A" "B" "B" "B" "C" "C" "C" "D" "D" "D"
```

and this:

```
"A" "B" "C" "D" "A" "B" "C" "D" "A" "B" "C" "D"
```

If you are unsure, look at `?rep`.

2. Draw 10 random letters from the lowercase alphabet, and sort them alphabetically (*Hint:* use `sample` and `sort`). The solution can be one line of code.

3. Draw 5 random letters from each of the lowercase and uppercase alphabets, incorporating both into a single vector, and sort it alphabetically.

4. Repeat the above exercise but sort the vector alphabetically in descending order.

1.14.8 Comparing and combining vectors

This question is **harder**! You learn three new functions and have to learn by experimenting.

Inspect the help page `union`, and note the useful functions `union`, `setdiff` and `intersect`. These can be used to compare and combine two vectors. Make two vectors :

```
x <- c(1,2,5,9,11)
y <- c(2,5,1,0,23)
```

Experiment with the three functions to find solutions to these questions.

1. Find values that are contained in both `x` and `y`

2. Find values that are in `x` but not `y` (and vice versa).

3. Construct a vector that contains all values contained in either `x` or `y`, and compare this vector to `c(x,y)`.

1.14.9 Into the matrix

In this exercise you will practice some basic skills with matrices. Recall Section 1.6.

1. Construct a matrix with 10 columns and 10 rows, all filled with random numbers between 0 and 1.

2. Calculate the row means of this matrix (*Hint:* use `rowMeans`). Also calculate the standard deviation across the row means (now also use `sd`).

3. Now remake the above matrix with 100 columns, and 10 rows. Then calculate the column means (using, of course, `colMeans`), and plot a frequency diagram (a 'histogram') using `hist`. We will see this function in more detail in a later chapter, but it is easy enough to use as you just do `hist(myvector)`, where `myvector` is any numeric vector (like the column means). What sort of shape of the histogram do you expect? Now repeat the above with more rows, and more columns.

1.14.10 Packages

This exercise makes sure you know how to install packages, and load them. First, read the first subsections of 1.10, on installing and loading packages.

1. Install the `car` package (you only have to do this once for any computer).

2. Load the `car` package (you have to do this every time you open Rstudio).

3. Look at the help file for `densityPlot`.

4. Run the example for `densityPlot` (at the bottom of the help file), by copy-pasting the example into a script, and then executing it.

5. Run the example for `densityPlot` again, but this time use the `example` function:

```
example(densityPlot)
```

Follow the instructions to cycle through the different steps.

6. Explore the contents of the `car` package by clicking first on the `Packages` tab, then finding the `car` package, and clicking on that. This way, you can find out about all functions a package contains (which, normally, you hope to avoid, but sometimes it is the only way to find what you are looking for). The same list can be obtained with the command `library(help=car)`, but that displays a list that is not clickable, so probably not as useful.

Chapter 2

Data skills - Part 1

2.1 Introduction

Analysing data is much more than applying the right statistics at the right time. A lot of effort and time is spent on reading, filtering, reshaping, bending and twisting your data before you can actually use the data for visualization and analysis.

In this chapter we learn many skills for working with data in R. All of these skills can be seen as mandatory skills before you learn new statistical techniques or fancy new visualizations. We first look at reading datasets into *dataframes*, filtering data when certain conditions are met, and learn in detail about the most important data types that can be stored in dataframes.

We will also look at various ways to summarize data, from simple summaries of what is contained in the original data, to more complex tables of statistics by grouping variables. Finally we will look at merging (joining) dataframes by one or more key-variables, and reshaping datasets (from long to wide, and back).

Packages used in this chapter

The examples will generally let you know which packages are used, but for your convenience, here is a complete list of the packages used.

For plotting, we often use `ggplot2` and `ggthemes` (these are usually omitted from the examples):

Otherwise:

- `lgrdata` (for the example datasets, see 1.1.1)
- `lubridate` (for dates and times)
- `dplyr` (for various data skills)
- `padr` (for aggregating timeseries data)
- `doBy` (for `summaryBy`, handy for making summary tables)
- `Hmisc` (for `contents` and `describe`, both summarize dataframes)
- `tidyr` (for reshaping dataframes)
- `reshape2` (for reshaping dataframes)
- `data.table` (for `fread`, fast reading of dataframes)
- `readxl` (for reading Excel files)
- `stringr` (for working with text)

2.2 Generating data

2.2.1 Sequences of numbers

Let's look at a few ways to generate sequences of numbers that we can use in the examples and exercises. There are also a number of real-world situations where you want to use these functions.

First, as we saw already, we can use c() to 'concatenate' (link together) a series of numbers. We can also combine existing vectors in this way, for example:

```
a <- c(1,2,3)
b <- c(4,5,6)
c(a,b)
```

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

We can generate sequences of numbers using `:`, `seq` and `rep`, like so:

```
# Sequences of integer numbers using the ":" operator:
1:10   # Numbers 1 through 10
```

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

```
5:-5   # From 5 to -5, backwards
```

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

```
# Examples using seq()
seq(from=10, to=100, by=10)
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

```
seq(from=23, by=2, length=12)
```

```
##  [1] 23 25 27 29 31 33 35 37 39 41 43 45
```

```
# Replicate numbers:
rep(2, times = 10)
```

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

```
rep(c(4,5), each=3)
```

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

The `rep` function works with any type of vector. For example, character vectors:

```
# Simple replication
rep("a", times = 3)
```

```
## [1] "a" "a" "a"
```

```
# Repeat elements of a vector
rep(c("E. tereticornis","E. saligna"), each=3)
```

```
## [1] "E. tereticornis" "E. tereticornis" "E. tereticornis" "E. saligna"
## [5] "E. saligna"      "E. saligna"
```

2.2.2 Random numbers

We can draw random numbers using the `runif` function. The `runif` function draws from a uniform distribution, meaning there is an equal probability of any number being chosen.

```
# Ten random numbers between 0 and 1
runif(10)
```

```
##  [1] 0.02675224 0.72146312 0.76453995 0.87358691 0.68313202 0.51843063
##  [7] 0.58088118 0.71956897 0.11767692 0.09467767
```

```
# Five random numbers between 100 and 1000
runif(5, 100, 1000)
```

```
## [1] 763.6927 136.6795 878.2090 513.9121 840.6874
```

Try this yourself The `runif` function is part of a much larger class of functions, each of which returns numbers from a different probability distribution. Inspect the help pages of the functions `rnorm` for the normal distribution, and `rexp` for the exponential distribution. Try generating some data from a normal distribution with a mean of 100, and a standard deviation of 10.

Next, we will `sample` numbers from an existing vector.

```
numbers <- 1:15
sample(numbers, size=20, replace=TRUE)
```

```
##  [1] 13  5  8  3  5  8  4 11  4  7 15 12  4 12 15  8  4  1  3 14
```

This command samples 20 numbers from the `numbers` vector, with replacement.

2.3 Reading data

There are many ways to read data into R, but we are going to keep things simple and show only a couple of options to read data from text files, and from databases (hosted online).

Throughout this book we focus on 'rectangular data', that is, data that can be organized in a table with columns and rows. A data table like this, with individual observations in rows, and various data fields in columns, is called a *data frame*.

We first show a few options to read text files into dataframes in R, including comma, tab-delimited, and JSON formats, and how to read data from Excel. Reading data from remote (No)SQL databases is included in Chapter 5.

2.3.1 Reading CSV files

A very common text format for data is 'Comma-Separated Values', or CSV. You can use `read.csv` to read these files. Suppose you have the file `Allometry.csv` in your working directory, you can read in the file into a dataframe with,

```
allometry <- read.csv("Allometry.csv")
```

Here, `read.csv` assumes that your system uses a point ('.') to separate digits, but in many parts of the world the default is a comma - in which case values are separated by ';'. In this case, use `read.csv2` instead of `read.csv`.

Make sure you fully understand the concept of a working directory (see Section 1.8) before continuing.

If the file is stored elsewhere, you can specify the entire path (this is known as an *absolute* path).

```
allometry <- read.csv("c:/projects/data/Allometry.csv")
```

It is generally *not recommended* to use absolute paths, because the script will then depend on an exact location of a datafile. But sometimes you want to refer to very large files or databases that are stored in a central location.

If the file is stored in a sub-directory of your working directory, you can specify the *relative* path.

```
allometry <- read.csv("data/Allometry.csv")
```

The latter option is probably useful to keep your data files separate from your scripts and outputs in your working directory.

The function `read.csv` has many options, let's look at some of them. We can skip a number of rows from being read, and only read a fixed number of rows. For example, use this command to read rows 10-15, skipping the header line (which is in the first line of the file) and the next 9 lines. *Note:* you have to skip 10 rows to read rows 10-15, because the header line (which is ignored) counts as a row in the text file!

```
allomsmall <- read.csv("Allometry.csv", skip=10, nrows=5, header=FALSE)
```

2.3.2 Reading large CSV files

We used the built-in `read.csv` function above, but note that for large files, it is rather slow. The best alternative is to use `fread` from the `data.table` package. The `data.table` package is an excellent resource if you need to do data manipulations on large files (though the syntax takes some getting used to).

```
library(data.table)
allom <- fread("Allometry.csv")
```

Of course, you could use `fread` even for small files, but I generally recommend to keep dependencies to a minimum. In other words, use built-in ('base') functions when you can, to develop more robust and reproducable code.

2.3.3 Reading Tab-delimited text files

Sometimes, data files are provided as text files that are TAB-delimited. To read these files, use the following command:

```
mydata <- read.table("sometabdelimdata.txt", header=TRUE)
```

In fact, `read.table` is the more general function - `read.csv` is a specific case for comma-delimited files. When using `read.table`, you must specify whether a header (i.e., a row with column names) is present in the dataset (unlike `read.csv`, it is the default to not read the header). If you have a text file with some other delimiter, for example `;`, use the `sep` argument:

```
mydata <- read.table("somedelimdata.txt", header=TRUE, sep=";")
```

2.3.4 Including data in a script

You can also write the dataset in a text file, and read it as in the following examples. This is useful if you have (found) a small dataset that you typed in by hand, or for making reproducible code snippets that include the dataset.

```
read.table(header=TRUE, text="
a b
1 2
3 4
")
```

```
##   a b
## 1 1 2
## 2 3 4
```

2.3.5 JSON

A very popular format for data, especially on the web, is JSON - a text-based format that can represent not just rectangular data (dataframes), but any complex nested data structure.

We can use the `jsonlite` package to read JSON data, and convert it to a dataframe if the data allows it, as in the following example.

```
# Fake data can be read fro this site.
# Visit the link to see what the original data looks like.
url <- "https://jsonplaceholder.typicode.com/posts/1/comments"

# Package to read JSON; it is very fast.
library(jsonlite)

# fromJSON reads and simplifies the data into a dataframe (if possible)
comment_data <- fromJSON(url)

# We end up with a dataframe.
Hmisc::contents(comment_data)
```

```
##
## Data frame:comment_data  5 observations and 5 variables    Maximum # NAs:0
##
##
##          Storage
## postId   integer
## id       integer
## name   character
## email  character
## body   character
```

2.3.6 (No)SQL databases

We show examples of how to obtain data from SQL and NoSQL databases in Chapter 5.

2.3.7 Excel spreadsheets

Excel spreadsheets are *not* a recommended way to store data, but often you don't make that choice yourself. If you do need to read an XLS or XLSX file, the `readxl` package works very well. *Note*: avoid older implementations like the `xlsx` package and `read.xls` in the `gtools` package, which are less reliable.

```r
library(readxl)
mydata <- read_excel("mspreadsheet.xlsx", sheet = 2)
```

Here, `sheet` will specify the sheet by number, alternatively you can refer to the sheet by name (e.g., `sheet = 'rawdata'`).

2.3.8 Proprietary formats

Many statistical software packages store data in their own format, not just text files. For data from **SPSS**, **SAS** or **Stata**, we recommend the `haven` package for reading the data into dataframes, and the `foreign` package provides further support for Minitab, **Systat**, and **Weka.**

2.3.9 Web services

Nowadays many data are available via a 'RESTful' API, which is now by far the most common way to download publicly available (open) data (and many other services as well). We discuss reading data from a REST service, as well as setting up our REST service in Chapter 5.

2.4 Working with dataframes

As mentioned, this book focuses heavily on dataframes, because this is the object you will use most of the time in data analysis. The following sections provide a brief introduction, but we will see many examples using dataframes throughout this manual.

Like matrices, dataframes have two dimensions: they contain both columns and rows. Unlike matrices, each column can hold a different type of data. This is very useful for keeping track of different types of information about data points. For example, you might use one column to hold height measurements, and another to hold the matching species IDs. When you read in a file using `read.csv`, the data is automatically stored in a dataframe. Dataframes can also be created from scratch using the function `data.frame` (see Section 2.4.1), but usually we start with data read from a file.

Info **What is a tibble?** Sometimes you will notice that some functions return objects that are very similar to dataframes, but are actually called 'tibbles' (for example, `read_excel` returns one). A tibble is a newer format for dataframes, and is internally very nearly the same. You can pretty much always assume that dataframes and tibbles behave in the same way.

2.4.1 Convert vectors into a dataframe

Suppose you have two or more vectors (of the same length), and you want to include these in a new dataframe. You can use the function `data.frame`. Here is a simple example:

```
vec1 <- c(9,10,1,2,45)
vec2 <- 1:5

data.frame(x = vec1, y = vec2)
```

```
##    x y
## 1  9 1
## 2 10 2
## 3  1 3
## 4  2 4
## 5 45 5
```

Here, we made a dataframe with columns named `x` and `y`. *Note*: take care to ensure that the vectors have the same length, otherwise it won't work!

Try this yourself Modify the previous example so that the two vectors are *not* the same length. Then, attempt to combine them in a dataframe and inspect the resulting error message.

2.4.2 Variables in the dataframe

We read the `allometry` data (make sure the `lgrdata` package is loaded) to practice basic dataframe skills.

```
library(lgrdata)
data(allometry)
```

After reading the dataframe, it is good practice to always quickly inspect the dataframe to see if anything went wrong. I routinely look at the first few rows with `head`. Then, to check the types of variables in the dataframe, use the `str` function (short for 'structure'). This function is useful for other objects as well, to view in detail what the object contains.

```
head(allometry)
```

```
##   species diameter height   leafarea branchmass
## 1    PSME    54.61  27.04 338.485622  410.24638
## 2    PSME    34.80  27.42 122.157864   83.65030
## 3    PSME    24.89  21.23   3.958274    3.51270
## 4    PSME    28.70  24.96  86.350653   73.13027
## 5    PSME    34.80  29.99  63.350906   62.39044
## 6    PSME    37.85  28.07  61.372765   53.86594
```

```
str(allometry)
```

```
## 'data.frame':    63 obs. of  5 variables:
##  $ species   : Factor w/ 3 levels "PIMO","PIPO",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ diameter  : num  54.6 34.8 24.9 28.7 34.8 ...
##  $ height    : num  27 27.4 21.2 25 30 ...
##  $ leafarea  : num  338.49 122.16 3.96 86.35 63.35 ...
##  $ branchmass: num  410.25 83.65 3.51 73.13 62.39 ...
```

Individual variables in a dataframe can be extracted using the dollar $ sign. Let's print all the tree diameters here, after rounding to one decimal point:

```
round(allometry$diameter,1)
```

```
##  [1] 54.6 34.8 24.9 28.7 34.8 37.9 22.6 39.4 39.9 26.2 43.7 69.8 44.5 56.6
## [15] 54.6  5.3  6.1  7.4  8.3 13.5 51.3 22.4 69.6 58.4 33.3 44.2 30.5 27.4
## [29] 43.2 38.9 52.6 20.8 24.1 24.9 46.0 35.0 23.9 60.2 12.4  4.8 70.6 11.4
## [43] 11.9 60.2 60.7 70.6 57.7 43.1 18.3 43.4 18.5 12.9 37.9 26.9 38.6  6.5
## [57] 31.8 73.7 28.2 61.5 51.6 18.3  8.4
```

It is also straightforward to add new variables to a dataframe. Let's convert the tree diameter to inches, and add it to the dataframe as a new variable:

```
allometry$diameterInch <- allometry$diameter / 2.54
```

Instead of using the $-notation every time (which can result in lengthy, messy code, especially when your variable names are long) you can use `with` to indicate where the variables are stored. Let's add a new variable called `volindex`, a volume index defined as the square of tree diameter times height:

```
allometry$volindex <- with(allometry, diameter^2 * height)
```

The `with` function allows for more readable code, while at the same time making sure that the variables `diameter` and `height` are read from the dataframe `allometry`.

An even better approach is to use `mutate` from the `dplyr` package (similar to base R's `transform`, but with a useful advantage).

```
library(dplyr)
allometry <- mutate(allometry,
                    diameterInch = diameter / 2.54,
                    volindex = diameterInch^2 * height)
```

Where `mutate` adds new variables to the dataframe, and (unlike `transform`) it is able to use variables used just before in the same call (note `diameterInch` was created in the same call).

Try this yourself The above examples are important - many times throughout this book you will be required to perform similar operations. As an exercise, also add the ratio of height to diameter to the dataframe.

A simple summary of the dataframe can be printed with the `summary` function; where we use indexing (`[,1:3]`) to use the first three columns of `allometry` only to save space.

```
summary(allometry[,1:3])
```

```
##   species      diameter          height
##   PIMO:19   Min.   : 4.83   Min.   : 3.57
##   PIPO:22   1st Qu.:21.59   1st Qu.:21.26
##   PSME:22   Median :34.80   Median :28.40
##             Mean   :35.56   Mean   :26.01
##             3rd Qu.:51.44   3rd Qu.:33.93
##             Max.   :73.66   Max.   :44.99
```

For the numeric variables, the minimum, 1st quantile, median, mean, 3rd quantile, and maximum values are printed. For so-called 'factor' variables (i.e., categorical variables), a simple table is printed (in this case, for the `species` variable). We will come back to factors in Section sec:workingfactors. If the variables have missing values, the number of missing values is printed as well (see Section 2.7.3).

To see how many rows and columns your dataframe contains (handy for double-checking you read the data correctly), use `nrow` and `ncol`. Alternatively, `dim` gives the 'dimension' of the dataframe (rows x columns).

```
nrow(allometry)
```

```
## [1] 63
```

```
ncol(allometry)
```

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

2.4.3 Changing column names in dataframes

To access the names of a dataframe as a vector, use the `names` function. You can also use this to change the names. Consider this example:

```
# read names:
names(allometry)
```

```
## [1] "species"      "diameter"     "height"       "leafarea"
## [5] "branchmass"   "diameterInch" "volindex"
```

```
# rename all (make sure vector is same length as number of columns!)
names(allometry) <- c("spec","diam","ht","leafarea","branchm")
```

We can also change some of the names, using simple indexing (see Section 2.5.1).

```
# rename Second one to 'Diam'
names(allometry)[2] <- "Diam"

# rename 1st and 2nd:
names(allometry)[1:2] <- c("SP","D")
```

Better yet is to use `rename` from the `dplyr` package, which makes sure you change the right column names (indexing as above can be dangerous if the order of columns has changed!).

```
library(dplyr)

# The names on the right are the original names, on the left are the new ones.
allometry <- rename(allometry,
                    spec = species,
                    diam = diameter)
```

2.5 Extracting data

2.5.1 Vectors

Let's look at reordering or taking subsets of a vector, or `indexing` as it is commonly called. This is an important skill to learn, so we will look at several examples.

Let's define two `numeric vectors`:

```
nums1 <- c(1,4,2,8,11,100,8)
nums2 <- c(3.3,8.1,2.5,9.8,21.2,13.8,0.9)
```

Individual elements of a vector can be extracted using square brackets, `[ ]`. For example, to extract the first and then the fifth element of a vector:

```
nums1[1]
```

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

```
nums1[5]
```

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

You can also use another object to do the indexing, as long as it contains a integer number. For example,

```
# Get last element:
nelements <- length(nums1)
nums1[nelements]
```

```
## [1] 8
```

This last example extracts the last element of a vector. To do this, we first found the length of the vector, and used that to *index* the vector to extract the last element.

We can also select multiple elements, by *indexing* the vector with another vector. Recall how to construct sequences of numbers, explained in Section 2.2.1.

```
# Select the first 3:
nums1[1:3]
```

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

```
# Select a few elements of a vector:
selectthese <- c(1,5,2)
nums1[selectthese]
```

```
## [1]  1 11  4
```

```
# Select every other element:
everyother <- seq(1,7,by=2)
nums1[everyother]
```

```
## [1]  1  2 11  8
```

```
# Select five random elements:
ranels <- sample(1:length(nums2), 5)
nums2[ranels]
```

```
## [1]  3.3  8.1  2.5 13.8 21.2
```

```
# Remove the first element:
nums1[-1]
```

```
## [1]   4   2   8  11 100   8
```

```
# Remove the first and last element:
nums1[-c(1, length(nums1))]
```

```
## [1]   4   2   8  11 100
```

Next, we can look at selecting elements of a vector based on the values in that vector. Suppose we want to make a new vector, based on vector `nums2` but only where the value within certain bounds. We can use simple logical statements to index a vector.

```
# Subset of nums2, where value is at least 10 :
nums2[nums2 > 10]
```

```
## [1] 21.2 13.8
```

```
# Subset of nums2, where value is between 5 and 10:
nums2[nums2 > 5 & nums2 < 10]
```

```
## [1] 8.1 9.8
```

```
# Subset of nums2, where value is smaller than 1, or larger than 20:
nums2[nums2 < 1 | nums2 > 20]
```

```
## [1] 21.2  0.9
```

```
# Subset of nums1, where value is exactly 8:
nums1[nums1 == 8]
```

```
## [1] 8 8
```

```
# Subset nums1 where number is NOT equal to 100
nums1[nums1 != 100]
```

```
## [1]  1  4  2  8 11  8
```

```
# Subset of nums1, where value is one of 1,4 or 11:
nums1[nums1 %in% c(1,4,11)]
```

```
## [1]  1  4 11
```

```
# Subset of nums1, where value is NOT 1,4 or 11:
nums1[!(nums1 %in% c(1,4,11))]
```

```
## [1]   2   8 100   8
```

These examples showed you several new logical operators (<, >, ==, &). See the help page `?Logic` for more details on logical operators. We will return to logical data in Section sec:workinglogic.

2.5.1.1 Assigning new values to subsets

All of this becomes very useful if we realize that new values can be easily assigned to subsets. This works for any of the examples above. For instance,

```
# Where nums1 was 100, make it -100
nums1[nums1 == 100] <- -100

# Where nums2 was less than 5, make it zero
nums2[nums2 < 5] <- 0
```

Try this yourself Using the first set of examples in this section, practice assigning new values to subsets of vectors.

2.5.2 Dataframes

In base R, there are two ways to take a subset of a dataframe: using the square bracket notation (`[]`) as in the above examples, or using the `filter` function from the `dplyr` package. We will learn both, as they are both useful from time to time.

Similar to vectors, dataframes can be indexed with row and column numbers using `mydataframe[row,column]`.

Here, `row` refers to the row number (which can be a vector of any length), and `column` to the column number (which can also be a vector). You can also refer to the column by its *name* rather than its number, which can be very useful. All this will become clearer after some examples.

Let's look at a few examples using the Allometry dataset.

```
# Load data
data(allometry)

# Extract tree diameters: take the 4th observation of the 2nd variable:
allometry[4,2]
```

```
## [1] 28.7
```

```
# We can also index the dataframe by its variable name:
allometry[4,"diameter"]
```

```
## [1] 28.7
```

```
# Extract the first 3 rows of 'height':
allometry[1:3, "height"]
```

```
## [1] 27.04 27.42 21.23
```

```
# Extract the first 5 rows, of ALL variables
# Note the use of the comma followed by nothing
# This means 'every column' and is very useful!
allometry[1:5,]
```

```
##   species diameter height   leafarea branchmass
## 1    PSME    54.61  27.04 338.485622  410.24638
## 2    PSME    34.80  27.42 122.157864   83.65030
## 3    PSME    24.89  21.23   3.958274    3.51270
## 4    PSME    28.70  24.96  86.350653   73.13027
## 5    PSME    34.80  29.99  63.350906   62.39044
```

```
# Extract the fourth column
# Here we use nothing, followed by a comma,
# to indicate 'every row'
allometry[,4]
```

```
##  [1] 338.485622 122.157864   3.958274  86.350653  63.350906  61.372765
##  [7]  32.077794 147.270523 141.787332  45.020041 145.809802 349.057010
## [13] 176.029213 319.507115 234.368784   4.851567   7.595163  11.502851
## [19]  25.380647  65.698749 160.839174  31.780702 189.733007 253.308265
## [25]  91.538428  90.453658  99.736790  34.464685  68.150309  46.326060
## [31] 160.993131   9.806496  20.743280  21.649603  66.633675  54.267994
## [37]  19.844680 131.727303  22.365837   2.636336 411.160376  15.476022
## [43]  14.493428 169.053644 139.651156 376.308256 417.209436 103.672633
## [49]  33.713580 116.154916  44.934469  18.855867 154.078625  70.602797
## [55] 169.163842   7.650902  93.072006 277.494360 131.856837 121.428976
## [61] 212.443589  82.093031   6.551044
```

```
# Select only 'height' and 'diameter', store in new dataframe:
allomhd <- allometry[,c("height", "diameter")]
```

As we saw when working with vectors (see Section 2.5.1), we can use expressions to extract data. Because each column in a dataframe is a vector, we can apply the same techniques to dataframes, as in the following examples.

We can also use one vector in a dataframe to find subsets of another. For example, what if we want to find the value of one vector, if another vector has a particular value?

```
# Extract diameters larger than 60
allometry$diameter[allometry$diameter > 60]
```

```
## [1] 69.85 69.60 60.20 70.61 60.20 60.71 70.61 73.66 61.47
```

```
# Extract all rows of allom where diameter is larger than 60.
# Make sure you understand the difference with the above example!
allometry[allometry$diameter > 60,]
```

```
##    species diameter height leafarea branchmass
## 12    PSME    69.85  31.35 349.0570   543.9731
## 23    PIPO    69.60  39.37 189.7330   452.4246
## 38    PIPO    60.20  31.73 131.7273   408.3383
## 41    PIPO    70.61  31.93 411.1604  1182.4222
## 44    PIPO    60.20  35.14 169.0536   658.2397
## 45    PIMO    60.71  39.84 139.6512   139.2559
## 46    PIMO    70.61  40.66 376.3083   541.3062
## 58    PIMO    73.66  44.64 277.4944   275.7165
## 60    PIMO    61.47  44.99 121.4290   199.8634
```

```
# We can use one vector to index another. For example, find the height of the tree
# that has the largest diameter, we can do:
allometry$height[which.max(allometry$diameter)]
```

```
## [1] 44.64
```

```
# Recalling the previous section, this is identical to:
allometry[which.max(allometry$diameter), "height"]
```

```
## [1] 44.64
```

```
# Get 10 random observations of 'leafarea'. Here, we make a new vector
# on the fly with sample(), which we use to index the dataframe.
allometry[sample(1:nrow(allometry),10),"leafarea"]
```

```
##  [1]  25.380647 349.057010 176.029213  31.780702   4.851567 121.428976
##  [7]  22.365837 169.053644  63.350906 139.651156
```

```
# As we did with vectors, we can also use %in% to select a subset.
# This example selects only two species in the dataframe.
allometry[allometry$species %in% c("PIMO","PIPO"),]
```

```
##    species diameter    height   leafarea branchmass
## 23    PIPO    69.60 39.369999 189.733007  452.42455
## 24    PIPO    58.42 35.810000 253.308265  595.64015
## 25    PIPO    33.27 20.800001  91.538428  160.44416
## 26    PIPO    44.20 29.110001  90.453658  149.72883
## 27    PIPO    30.48 22.399999  99.736790   44.13532
## 28    PIPO    27.43 27.690001  34.464685   22.98360
## 29    PIPO    43.18 35.580000  68.150309  106.40410
## 30    PIPO    38.86 33.120001  46.326060   58.24071
## 31    PIPO    52.58 41.160003 160.993131  214.34109
## 32    PIPO    20.83 23.340000   9.806496    8.25614
## 33    PIPO    24.13 25.940001  20.743280   22.60111
## 34    PIPO    24.89 25.110000  21.649603   16.77015
## 35    PIPO    45.97 30.389999  66.633675   87.36908
## 36    PIPO    35.05 28.399999  54.267994   51.09006
## 37    PIPO    23.88 23.380001  19.844680   13.98343
## 38    PIPO    60.20 31.729999 131.727303  408.33826
## 39    PIPO    12.45  7.360001  22.365837   16.98648
## 40    PIPO     4.83  3.570000   2.636336    1.77810
## 41    PIPO    70.61 31.929997 411.160376 1182.42222
## 42    PIPO    11.43  6.920000  15.476022    9.25151
## 43    PIPO    11.94  5.849999  14.493428    7.55701
## 44    PIPO    60.20 35.139998 169.053644  658.23971
## 45    PIMO    60.71 39.840003 139.651156  139.25590
## 46    PIMO    70.61 40.659999 376.308256  541.30618
## 47    PIMO    57.66 38.889998 417.209436  310.56688
## 48    PIMO    43.13 36.240000 103.672633  100.40178
## 49    PIMO    18.29 23.130000  33.713580   14.48567
## 50    PIMO    43.43 37.589998 116.154916  108.44781
## 51    PIMO    18.54 21.289999  44.934469   16.54457
## 52    PIMO    12.95 13.440000  18.855867    8.71068
## 53    PIMO    37.85 36.590000 154.078625   72.02907
## 54    PIMO    26.92 29.049999  70.602797   35.87333
## 55    PIMO    38.61 35.519999 169.163842  114.06445
## 56    PIMO     6.48  5.420000   7.650902    3.50621
## 57    PIMO    31.75 34.559999  93.072006   44.72725
## 58    PIMO    73.66 44.640000 277.494360  275.71655
## 59    PIMO    28.19 22.590000 131.856837   91.76231
## 60    PIMO    61.47 44.989998 121.428976  199.86339
## 61    PIMO    51.56 40.229999 212.443589  220.55688
## 62    PIMO    18.29 12.980000  82.093031   28.04785
## 63    PIMO     8.38  4.950000   6.551044    4.36969
```

```r
# Extract tree diameters for the PIMO species, as long as diameter > 50
allometry$diameter[allometry$species == "PIMO" & allometry$diameter > 50]
```

```
## [1] 60.71 70.61 57.66 73.66 61.47 51.56
```

```r
# (not all output shown)
```

Try this yourself As with vectors, we can quickly assign new values to subsets of data using the <- operator. Try this on some of the examples above.

2.5.3 A faster method

While the above method to index dataframes is very flexible and concise, sometimes it leads to code that is difficult to understand. It is also easy to make mistakes when you subset dataframes by the column or row number (imagine the situation where the dataset has changed and you redo the analysis). Consider the `filter` function as a convenient and safe alternative, from the `dplyr` package. (Note that base R provides a nearly identical function, `subset`, but `filter` is much faster).

With `filter`, you can select rows that meet a certain criterion, and columns as well. This example uses the pupae data. The last example shows the use of `select` from `dplyr` to keep only certain columns, conveniently with the pipe operator.

```r
# Read data
data(pupae)

# For filter(), select().
library(dplyr)

# Take subset of pupae, ambient temperature treatment and CO2 is 280.
# Note: statements separated by commas are interpreted as AND by filter()
filter(pupae,
       T_treatment == "ambient",
       CO2_treatment == 280,
       Gender == 0)
```

```
##   T_treatment CO2_treatment Gender PupalWeight Frass
## 1     ambient           280      0       0.244 1.900
## 2     ambient           280      0       0.221    NA
## 3     ambient           280      0       0.280 1.996
## 4     ambient           280      0       0.257 1.069
## 5     ambient           280      0       0.275 2.198
## 6     ambient           280      0       0.254 2.220
## 7     ambient           280      0       0.258 1.877
## 8     ambient           280      0       0.224 1.488
```

```r
# (not all output shown)

# Take subset where Frass is larger than 2.9.
# Also, keep only variables 'PupalWeight' and 'Frass'.
filter(pupae, Frass > 2.6) %>%
  select(PupalWeight, Frass)
```

```
##   PupalWeight Frass
## 1       0.319 2.770
## 2       0.384 2.672
## 3       0.385 2.603
## 4       0.405 3.117
## 5       0.473 2.631
## 6       0.469 2.747
```

2.5.4 Deleting columns

It is rarely necessary to delete columns from a dataframe, unless you want to save a copy of the dataframe to disk (see Section 2.6). Instead of deleting columns, you can take a subset and make a new dataframe to continue with. Also, it should not be necessary to delete columns from the dataframe that you have accidentally created in a reproducible script: when things go wrong, simply clear the workspace and run the entire script again.

That aside, you have the following options to delete a column from a dataframe.

```
# A simple example dataframe
dfr <- data.frame(a=-5:0, b=10:15)

# Delete the second column (make a new dataframe 'dfr2' that does not include that column)
# This only works if you know the column index (here, 2) for sure,
# it does not work with a column name!
dfr2 <- dfr[,-2]

# Using select() from the dplyr, we can drop columns by name:
dfr2 <- select(dfr, -a)
```

2.6 Exporting data

To write a dataframe to a comma-separated values (CSV) file, use the `write.csv` function. For example,

```
# Some data
dfr <- data.frame(x=1:3, y=2:4)

# Write to disk (row names are generally not wanted in the CSV file).
write.csv(dfr,"somedata.csv", row.names=FALSE)
```

If you want more options, or a different delimiter (such as TAB), look at the `write.table` function. Note that if you write a dataset to a text file, and read it back in with `read.csv` (or `read.table`), the dataset will not be exactly the same. One reason is the number of digits used in the text file, or that you have converted some columns to character, numeric, or whatever.

If you want to store a dataframe to disk, and later read it back into R exactly as it was before, it is preferred to use a **binary format**. This idea works for any R object, not just dataframes:

```
# Much faster than writing text files, resulting files are much smaller,
# and objects are saved exactly as they were in R.
saveRDS(dfr, "somefile.rds")

# To read the object back in, do:
dfr <- readRDS("somefile.rds")
```

It is also possible to save all objects that are currently loaded in memory (everything that shows up with `ls()`) with the command `save.image`. However, we strongly urge against it, as it is easy to lose track of which objects are important, and it is too easy to make a mess of things. A fresh installation of Rstudio saves all your objects after you quit Rstudio, but as we mentioned in Section 1.1, we suggest you switch that behaviour off.

2.7 Special data types

Now that we know how to read in dataframes, it is time we take a closer look at the types of data that can be contained in a dataframe. For the purpose of this book, a dataframe can contain six types of data. These are summarized in the table below:

```
## Warning in print.xtable(xtab, tabular.environment = "longtable"): Attempt
## to use "longtable" with floating = TRUE. Changing to FALSE.
```

% latex table generated in R 3.6.2 by xtable 1.8-4 package % Tue Mar 03 22:32:16 2020

	Data type	Description
1	numeric	Any number, including 'double' (double precision floating point) and 'integer' (whole numbers). In R you
2	character	Strings of text
3	factor	Categorial variable. Preferred over character when few unique levels (values) present in the data. Must
4	logical	Either TRUE or FALSE. Internally stored as 0 (FALSE) or 1 (TRUE).
5	Date	Special Date class. Internally stored as number of days since 1970-1-1.
6	POSIXct	Special Date-time class. Internally stored as number of seconds since 1970-1-1, and may have timezone

Also, R has a very useful built-in data type to represent **missing values**. This is represented by `NA` (Not Available) (see Section 2.7.3).

We will show how to convert between data types at the end of this chapter (Section 2.7.6).

2.7.1 Working with factors

The *factor* data type is used to represent qualitative, categorical data.

When reading data from file, for example with `read.csv`, R will automatically convert any variable to a factor if it is unable to convert it to a numeric variable. If a variable is actually numeric, but you want to treat it as a factor, you can use `as.factor` to convert it, as in the following example.

```
# Make sure you have loaded the lgrdata package
data(pupae)

# This dataset contains a temperature (T_treatment) and CO2 treatment (CO2_treatment).
# Both should logically be factors, however, CO2_treatment is read as numeric:
str(pupae)
```

```
## 'data.frame':    84 obs. of  5 variables:
##  $ T_treatment  : Factor w/ 2 levels "ambient","elevated": 1 1 1 1 1 1 1 1 1 1 ...
##  $ CO2_treatment: int  280 280 280 280 280 280 280 280 280 280 ...
##  $ Gender       : int  0 1 0 0 0 1 0 1 0 1 ...
##  $ PupalWeight  : num  0.244 0.319 0.221 0.28 0.257 0.333 0.275 0.312 0.254 0.356 ...
##  $ Frass        : num  1.9 2.77 NA 2 1.07 ...
```

```
# To convert it to a factor, we use:
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)

# Compare with the above,
str(pupae)
```

```
## 'data.frame':    84 obs. of  5 variables:
```

```
##  $ T_treatment  : Factor w/ 2 levels "ambient","elevated": 1 1 1 1 1 1 1 1 1 1 ...
##  $ CO2_treatment: Factor w/ 2 levels "280","400": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Gender       : int  0 1 0 0 0 1 0 1 0 1 ...
##  $ PupalWeight  : num  0.244 0.319 0.221 0.28 0.257 0.333 0.275 0.312 0.254 0.356 ...
##  $ Frass        : num  1.9 2.77 NA 2 1.07 ...
```

In the `allometry` example dataset, the `species` variable is a good example of a factor. A factor variable has a number of 'levels', which are the text values that the variable has in the dataset. Factors can also represent treatments of an experimental study. For example,

```
data(allometry)
levels(allometry$species)
```

```
## [1] "PIMO" "PIPO" "PSME"
```

Shows the three species in this dataset. We can also count the number of rows in the dataframe for each species, like this:

```
table(allometry$species)
```

```
##
## PIMO PIPO PSME
##   19   22   22
```

Note that the three species are always shown in the order of the **levels of the factor**: when the dataframe was read, these levels were assigned based on alphabetical order. Often, this is not a very logical order, and you may want to rearrange the levels to get more meaningful results.

In our example, let's shuffle the levels around, using `factor`.

```
allometry$species <- factor(allometry$species, levels=c("PSME","PIMO","PIPO"))
```

Now revisit the commands above, and note that the results are the same, but the order of the `levels` of the `factor` is different. You can also reorder the levels of a factor by the values of another variable, see the example in Section 3.1.4.

2.7.1.1 Turn numeric data into factors

We can also generate new factors, and add them to the dataframe. This is a common application:

```
# Add a new variable to allom:  'small' when diameter is less than 10, 'large' otherwise.
allometry$treeSizeClass <- factor(ifelse(allometry$diameter < 10, "small", "large"))

# Now, look how many trees fall in each class.
# Note that somewhat confusingly, 'large' is printed before 'small'.
# Once again, this is because the order of the factor levels is alphabetical by default.
table(allometry$treeSizeClass)
```

```
##
## large small
##    56     7
```

What if we want to add a new factor based on a numeric variable with more than two levels? In that case, we cannot use `ifelse`. We must find a different method. Look at this example using `cut`.

```
# The cut function takes a numeric vectors and cuts it into a categorical variable.
# Continuing the example above, let's make 'small','medium' and 'large' tree size classes:
```

```
allometry$treeSizeClass <- cut(allometry$diameter, breaks=c(0,25,50,75),
                               labels=c("small","medium","large"))

# And the results,
table(allometry$treeSizeClass)
```

```
##
##  small medium  large
##     22     24     17
```

2.7.1.2 Empty factor levels

It is important to understand how factors are used in R: they are not simply text variables, or 'character strings'. Each unique value of a factor variable is assigned a *level*, which is used every time you summarize your data by the factor variable.

Crucially, even when you delete data, the original factor *level* is still present. Although this behaviour might seem strange, it makes a lot of sense in many cases (zero observations for a particular factor level can be quite informative!).

Sometimes it is more convenient to drop empty factor levels with the `droplevels` function. Consider this example:

```
# Note that 'T_treatment' (temperature treatment) is a factor with two levels,
# with 37 and 47 observations in total:
table(pupae$T_treatment)
```

```
##
##  ambient elevated
##       37       47
```

```
# Suppose we decide to keep only the ambient treatment:
pupae_amb <- dplyr::filter(pupae, T_treatment == "ambient")

# Now, the level is still present, although empty:
table(pupae_amb$T_treatment)
```

```
##
##  ambient elevated
##       37        0
```

```
# In this case, we don't want to keep the empty factor level.
# Use droplevels to get rid of any empty levels:
pupae_amb2 <- droplevels(pupae_amb)
```

Try this yourself Compare the `summary` of `pupae_amb` and `pupae_amb2`, and note the differences.

2.7.1.3 Changing the levels of a factor

Sometimes you may want to change the levels of a factor, for example to replace abbreviations with more readable labels. To do this, we can assign new values with the `levels` function, as in the following example using the pupae data:

```
# Change the levels of T_treatment by assigning a character vector to the levels.
levels(pupae$T_treatment) <- c("Ambient","Ambient + 3C")

# Or only change the first level, using subscripting.
levels(pupae$T_treatment)[1] <- "Control"
```

Try this yourself Using the method above, you can also merge levels of a factor, simply by assigning the same new level to both old levels. Try this on a dataset of your choice (for example, in the `allom` data, you can assign new species levels, 'Douglas-fir' for PSME, and 'Pine' for both PIMO and PIPO). Then check the results with `levels()`.

2.7.2 Working with logical data

Some data can only take two values: true, or false. For data like these, R has the *logical* data type.

The following examples use various logical operators, and all return TRUE or FALSE, or a vector of them. The help page `?Syntax` has a comprehensive list of operators in R (including the logical operators).

```
# Answers to (in)equalities are always logical:
10 > 5
```

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

```
101 == 100 + 1
```

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

```
# ... or use objects for comparison:
apple <- 2
pear <- 3
apple == pear
```

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

```
# NOT equal to.
apple != pear
```

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

```
# Logical comparisons like these also work for vectors, for example:
nums <- c(10,21,5,6,0,1,12)
nums > 5
```

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

The functions `which`, `any` and `all` are very useful to know when working with logical data:

```
# Find which of the numbers are larger than 5:
which(nums > 5)
```

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

```
# Other useful functions are 'any' and 'all':
# Are any numbers larger than 25?
any(nums > 25)
```

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

```r
# Are all numbers less than or equal to 10?
all(nums <= 10)
```

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

You have already been using logical data when we filtered dataframes and vectors:

```r
# Use & for AND, for example to take subsets where two conditions are met:
subset(pupae, PupalWeight > 0.4 & Frass > 3)
```

```
##    T_treatment CO2_treatment Gender PupalWeight Frass
## 25     ambient           400      1       0.405 3.117
```

```r
# Use | for OR
nums[nums < 2 | nums > 20]
```

```
## [1] 21  0  1
```

Logical data are coded by integer numbers (0 = `FALSE`, 1 = `TRUE`), but normally you don't see this, since R will only *print* `TRUE` and `FALSE` 'labels'. However, once you know this, some analyses become even easier.

```r
# How many numbers are larger than 5?
#- Short solution
sum(nums > 5)
```

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

```r
#- Long solution
length(nums[nums > 5])
```

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

```r
# What fraction of data is larger than some value?
mean(pupae$PupalWeight > 0.3)
```

```
## [1] 0.4880952
```

2.7.3 Working with missing values

In R, missing values are represented with `NA`, a special data type that indicates the data is simply *Not Available*.

Many functions can handle missing data, usually in different ways. For example, suppose we have the following vector:

```r
myvec1 <- c(11,13,5,6,NA,9)
```

In order to calculate the mean, we might want to either exclude the missing value (and calculate the mean of the remaining five numbers), or we might want `mean(myvec1)` to fail (produce an error). This last case is useful if we don't expect missing values, and want R to only calculate the mean when there are no `NA`'s in the dataset.

These two options are shown in this example:

```r
# Calculate mean: this fails if there are missing values
mean(myvec1)
```

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

```r
# Calculate mean after removing the missing values
mean(myvec1, na.rm=TRUE)
```

```
## [1] 8.8
```

Many functions have an argument `na.rm`, or similar. Refer to the help page of the function to learn about the various options (if any) for dealing with missing values. For example, see the help pages `?lm` and `?sd`.

The function `is.na` returns `TRUE` when a value is missing, which can be useful to see which values are missing, or how many,

```r
# Is a value missing? (TRUE or FALSE)
is.na(myvec1)
```

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

```r
# Which of the elements of a vector is missing?
which(is.na(myvec1))
```

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

```r
# How many values in a vector are NA?
sum(is.na(myvec1))
```

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

2.7.3.1 Making missing values

In many cases it is useful to change some bad data values to `NA`. We can use our indexing skills to do so,

```r
# Some vector that contains bad values coded as -9999
datavec <- c(2,-9999,100,3,-9999,5)

# Assign NA to the values that were -9999
datavec[datavec == -9999] <- NA
```

In other cases, missing values arise when certain operations did not produce the desired result. Consider this example,

```r
# A character vector, some of these look like numbers:
myvec <- c("101","289","12.3","abc","99")

# Convert the vector to numeric:
as.numeric(myvec)
```

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

```
## [1] 101.0 289.0  12.3    NA  99.0
```

The warning message **NAs introduced by coercion** means that missing values were produced by when we tried to turn one data type (character) to another (numeric).

2.7.3.2 Not A Number

Another type of missing value is the result of calculations that went wrong, for example:

```
# Attempt to take the logarithm of a negative number:
log(-1)
```

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

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

The result is `NaN`, short for *Not A Number*.

Dividing by zero is not usually meaningful, but R does not produce a missing value:

```
1000/0
```

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

It produces 'Infinity' instead.

2.7.3.3 Missing values in dataframes

When working with dataframes, you often want to remove missing values for a particular analysis. We'll use the `pupae` dataset for the following examples. Note we use the `dplyr` package for `filter`.

```
# (re-)load data
data(pupae)

# Look at a summary to see if there are missing values:
summary(pupae)
```

```
##    T_treatment  CO2_treatment       Gender        PupalWeight
##  ambient :37   Min.    :280.0   Min.    :0.0000   Min.    :0.1720
##  elevated:47   1st Qu.:280.0   1st Qu.:0.0000   1st Qu.:0.2562
##                Median :400.0   Median :0.0000   Median :0.2975
##                Mean    :344.3   Mean    :0.4487   Mean    :0.3110
##                3rd Qu.:400.0   3rd Qu.:1.0000   3rd Qu.:0.3560
##                Max.    :400.0   Max.    :1.0000   Max.    :0.4730
##                                 NA's    :6
##      Frass
##  Min.    :0.986
##  1st Qu.:1.515
##  Median :1.818
##  Mean    :1.846
##  3rd Qu.:2.095
##  Max.    :3.117
##  NA's    :1
```

```
# Notice there are 6 NA's (missing values) for Gender, and 1 for Frass.

# Option 1: take subset of data where Gender is not missing:
pupae_subs1 <- filter(pupae, !is.na(Gender))

# Option 2: take subset of data where Frass AND Gender are not missing
```

```
pupae_subs2 <- filter(pupae,
                      !is.na(Frass),
                      !is.na(Gender))

# A more rigorous subset: remove all rows from a dataset where ANY variable
# has a missing value:
pupae_nona <- pupae[complete.cases(pupae),]

# This was an example where indexing (using square brackets) is a bit more
# convenient than the dplyr approach, where we have to write the following.
# A number of similar looking statements do not work!
pupae_nona <- pupae %>% filter(complete.cases(.))
```

2.7.3.4 Subsetting when there are missing values

When there are missing values in a vector, and you take a subset (for example all data larger than some value), should the missing values be included or dropped? There is no one answer to this, but it is important to know that `subset` drops them, but the square bracket method (`[]`) keeps them.

Consider this example, and especially the use of `which` to drop missing values when subsetting.

```
# A small dataframe
dfr <- data.frame(a=1:4, b=c(4,NA,6,NA))

# subset drops all missing values
# Note: dplyr::filter also drops NA
subset(dfr, b > 4, select=b)
```

```
##   b
## 3 6
```

```
# square bracket notation keeps them
dfr[dfr$b > 4,"b"]
```

```
## [1] NA  6 NA
```

```
# ... but drops them when we use 'which'
dfr[which(dfr$b > 4),"b"]
```

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

2.7.4 Working with text

Many datasets include variables that are text only (think of comments, species names, locations, sample codes, and so on), it is useful to learn how to modify, extract, and analyse text-based ('character') variables.

Consider the following simple examples when working with a single character string:

```
# Count number of characters in a bit of text:
sentence <- "Not a very long sentence."
nchar(sentence)
```

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

```
# Extract the first 3 characters:
substr(sentence, 1, 3)
```

```
## [1] "Not"
```

We can also apply these functions to a vector:

```
# Substring all elements of a vector
substr(c("good","good riddance","good on ya"),1,4)
```

```
## [1] "good" "good" "good"
```

```
# Number of characters of all elements of a vector
nchar(c("hey","hi","how","ya","doin"))
```

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

2.7.4.1 Combining text

To glue bits of text together, use the `paste` function, like so:

```
# Add a suffix to each text element of a vector:
txt <- c("apple","pear","banana")
paste(txt, "fruit", sep = "-")
```

```
## [1] "apple-fruit"  "pear-fruit"   "banana-fruit"
```

```
# Glue them all together into a single string using the collapse argument
paste(txt, collapse = "-")
```

```
## [1] "apple-pear-banana"
```

```
# Combine numbers and text:
paste("Question", 1:3)
```

```
## [1] "Question 1" "Question 2" "Question 3"
```

```
# This can be of use to make new variables in a dataframe,
# as in this example where we combine two factors to create a new one:
pupae$T_CO2 <- with(pupae, paste(T_treatment, CO2_treatment, sep="-"))
head(pupae$T_CO2)
```

```
## [1] "ambient-280" "ambient-280" "ambient-280" "ambient-280" "ambient-280"
## [6] "ambient-280"
```

Try this yourself Run the final example above, and inspect the variable `T_CO2` (with `str`) that we added to the dataframe. Make it into a factor variable using `as.factor`, and inspect the variable again.

2.7.4.2 Text in dataframes and `grep`

When you read in a dataset (with `read.csv`, `read.table` or similar), any variable that R cannot convert to numeric *is automatically converted to a factor*. This means that if a column has even just one value that is text (or some garble that does not represent a number), the column cannot be numeric.

When you want non-numeric columns to be read in as *character* instead of *factor*, you must do:

```
cereals <- read.csv("cereals.csv", stringsAsFactors = FALSE)
```

Here, the argument `stringsAsFactors=FALSE` avoided the automatic conversion of character variables to factors.

While we know that factors are very useful, sometimes we want a variable to be treated like text. For example, if we plan to analyse text directly, or extract numbers or other information from bits of text. Let's look at a few examples using the `titanic` dataset.

```
# Load dataset
data(titanic)

# Is passenger name stored as a character?
is.character(titanic$Name)
```

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

```
# Evidently not. We can convert it to character:
titanic$Name <- as.character(titanic$Name)

# ... or as part of a dplyr chain:
titanic <- titanic %>% mutate(Name = as.character(Name))
```

The following example uses `grep`, a very powerful function. This function can make use of *regular expressions*, a flexible tool for text processing.

```
# Extract cereal names (for convenience).
data(cereals)
cerealnames <- cereals$Cereal.name

# Find the cereals that have 'Raisin' in them.
# grep() returns the index of values that contain Raisin
grep("Raisin",cerealnames)
```

```
##  [1] 23 45 46 50 52 53 59 60 61 71
```

```
# grepl() returns TRUE or FALSE
grepl("Raisin",cerealnames)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE
## [56] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [67] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# That result just gives you the indices of the vector that have 'Raisin' in them.
# these are the corresponding names:
cerealnames[grep("Raisin",cerealnames)]
```

```
##  [1] "Crispy_Wheat_&_Raisins"
##  [2] "Muesli_Raisins,_Dates,_&_Almonds"
##  [3] "Muesli_Raisins,_Peaches,_&_Pecans"
##  [4] "Nutri-Grain_Almond-Raisin"
##  [5] "Oatmeal_Raisin_Crisp"
##  [6] "Post_Nat._Raisin_Bran"
##  [7] "Raisin_Bran"
##  [8] "Raisin_Nut_Bran"
##  [9] "Raisin_Squares"
## [10] "Total_Raisin_Bran"
```

```
# Now find the cereals whose name starts with Raisin.
# The ^ symbol is part of a 'regular expression', it indicates 'starts with':
grep("^Raisin",cerealnames)
```

```
## [1] 59 60 61
```

```
# Or end with 'Bran'
# The $ symbol is part of a 'regular expression', and indicates 'ends with':
grep("Bran$", cerealnames)
```

```
##  [1]  1  2  3 20 29 53 59 60 65 71
```

As mentioned, `grep` can do a *lot* of different things, so don't be alarmed if you find the help page *a bit overwhelming*. However, there are a few options worth knowing about. One very useful option is to turn off the case-sensitivity, for example:

```
grep("bran", cerealnames, ignore.case=TRUE)
```

```
##  [1]  1  2  3  4  9 10 20 29 53 59 60 65 71
```

finds `Bran` and `bran` and `BRAN`.

Finally, using the above tools, let's add a new variable to the `cereal` dataset that is `TRUE` when the name of the cereal ends in 'Bran', otherwise it is `FALSE`. For this example, the `grepl` function is more useful (because it returns TRUE and FALSE).

```
# grepl will return FALSE when Bran is not found, TRUE otherwise
cereals$BranOrNot <- grepl("Bran$", cerealnames)

# Quick summary:
table(cereals$BranOrNot)
```

```
##
## FALSE  TRUE
##    67    10
```

Further reading A clear explanation of regular expressions is on the Wikipedia page http://en.wikipedia.org/wiki/Regular_expressions. Also worth a look is Robin Lovelace's introduction specifically for R and RStudio at https://www.r-bloggers.com/regular-expressions-in-r-vs-rstudio/.

2.7.4.3 More control with the `stringr` package

In the previous section we used `grep` and `grepl` to find text in strings, which we can use to subset dataframes, or simply to find data. We can do a lot more with strings, but things get a bit difficult quickly when we stick to base R. Instead, the `stringr` package provides an easy-to-use interface to many common text operations. The following examples give a few very handy tips, but the package provides much more (see `library(help=stringr)`).

```r
library(stringr)

# Extract all numbers from a string
# Note that the result is a character!
str_extract("I have 5 apples", "[0-9]")
```

```
## [1] "5"
```

```r
# Extract the nth word from text.
word("Dantchoff, Mr Khristo", 2)
```

```
## [1] "Mr"
```

```r
# Replace text that matches a pattern.
# (sub or gsub from base R also work OK)
str_replace("Dantchoff, Mr Khristo", "Mr", "Mrs")
```

```
## [1] "Dantchoff, Mrs Khristo"
```

2.7.4.4 Combining text and R objects with `glue`

When controlling output, we often want to combine results stored in R objects, and text. As we saw in Section 2.7.4.1, we can use `paste` to achieve this, but there's a better way:

```r
library(glue)

data(titanic)
glue("The titanic dataset has {nrow(titanic)} rows. ",
     "For {percmissing}% of the data, Age is missing.",
     percmissing = round(100 * mean(is.na(titanic$Age)), 1))
```

```
## The titanic dataset has 1313 rows. For 42.4% of the data, Age is missing.
```

With `glue`, we can include both R code directly within `{}` (as we did with `nrow(titanic)`), and also use temporary variables (here: `percmissing`) that are given as extra arguments. Also note that strings separated by `,` are pasted together by `glue`.

2.7.5 Working with dates and times

Admittedly, working with dates and times in R is somewhat annoying at first. The built-in help files on this subject describe all aspects of this special data type, but do not offer much for the beginning R user. This section covers basic operations that you may need when analysing and formatting datasets.

For working with dates, we use the `lubridate` package, which simplifies it tremendously.

2.7.5.1 Reading dates

The built-in `Date` class in R is encoded as an integer number representing the number of days since 1-1-1970 (but this actual origin does not matter for the user). Converting a character string to a date with `as.Date` is straightforward if you use the standard order of dates: `YYYY-MM-DD`. So, for example,

```r
as.Date("2008-5-22")
```

```
## [1] "2008-05-22"
```

The output here is not interesting, R simply prints the date. Because dates are represented as numbers in R, we can do basic arithmetic:

```r
# First load lubridate when working with dates or date-time combinations.
# (as.Date and difftime are part of the base package, but lubridate provides
# years(), months(), and many more functions used below).
library(lubridate)

# A date, 7 days later:
as.Date("2011-5-12") + 7
```

```
## [1] "2011-05-19"
```

```r
# Difference between dates.
as.Date("2009-7-1") - as.Date("2008-12-1")
```

```
## Time difference of 212 days
```

```r
# With difftime, you can specify the units:
difftime(as.Date("2009-7-1"), as.Date("2008-12-1"), units = "weeks")
```

```
## Time difference of 30.28571 weeks
```

```r
# To add other timespans, use functions months(), years() or weeks() to
# avoid problems with leap years
as.Date("2013-8-18") + years(10) + months(1)
```

```
## [1] "2023-09-18"
```

Try this yourself The previous example showed a very useful trick for adding numbers to a `Date` to get a new `Date` a few days later. Confirm for yourself that this method accounts for leap years. That is, the day before `2011-3-1` should be `2011-2-28` (2011 is not a leap year). But what about `2012-3-1`?

Often, text strings representing the date are not in the standard format. Fortunately, it is possible to convert any reasonable sequence to a `Date` object in R. All we have to do is provide a character string to `as.Date` and tell the function the order of the fields.

To convert any format to a Date, we can use the `lubridate` package, which contains the functions `ymd`, `mdy`, and all other combinations of y, m, and d. These functions are pretty smart, as can be seen in these examples:

```r
# Day / month / year
as.Date(dmy("31/12/1991"))
```

```
## [1] "1991-12-31"
```

```r
# Month - day - year (note, only two digits for the year)
as.Date(mdy("4-17-92"))
```

```
## [1] "1992-04-17"
```

```r
# Year month day
as.Date(ymd("1976-5-22"))
```

```
## [1] "1976-05-22"
```

```r
#-- Unusual formatting can be read in with the 'format' argument
# in as.Date. See ?strptime for a list of codes.
# For example, Year and day of year ("%j" stands for 'Julian date')
as.Date("2011 121", format="%Y %j")
```

```
## [1] "2011-05-01"
```

Another method to construct date objects is when you do not have a character string as in the above example, but separate numeric variables for year, month and day. In this case, use the `ISOdate` function:

```r
as.Date(ISOdate(2008,12,2))
```

```
## [1] "2008-12-02"
```

Finally, here is a simple way to find the number of days since you were born, using `today` from the lubridate package.

```r
# Today's date (and time) can be with the today() function
today()
```

```
## [1] "2020-03-03"
```

```r
# We can now simply subtract your birthday from today's date.
today() - as.Date("1976-5-22")
```

```
## Time difference of 15991 days
```

2.7.5.2 Example: using dates in a dataframe

The `as.Date` function that we met in the previous section also works with vectors of dates, and the `Date` class can also be part of a dataframe. Let's take a look at the Hydro data to practice working with dates.

```r
# Load the hydro dataset
data(hydro)

# Now convert this to a Date variable.
# If you first inspect head(hydro$Date), you will see the format is DD/MM/YYYY
hydro$Date <- as.Date(dmy(hydro$Date))
```

If any of the date conversions go wrong, the `dmy` function (or its equivalents) should print a message letting you know. You can double check if `any` of the converted dates is `NA` like this:

```r
any(is.na(hydro$Date))
```

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

We now have successfully read in the date variable. The `min` and `max` functions are useful to check the range of dates in the dataset:

```r
# Minimum and maximum date (that is, oldest and most recent),
min(hydro$Date)
```

```
## [1] "2005-08-08"
```

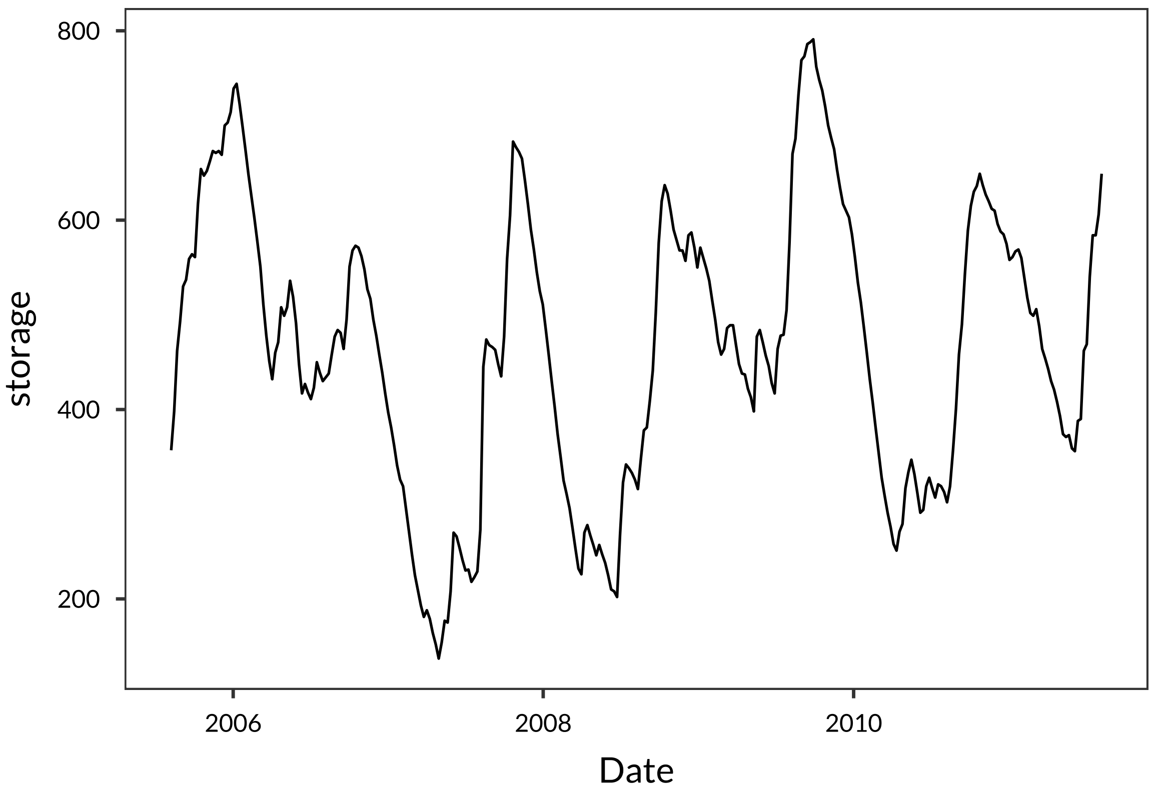


Figure 2.1: A simple plot of the hydro data.

```r
max(hydro$Date)
```

```
## [1] "2011-08-08"
```

```r
#... and length of measurement period:
max(hydro$Date) - min(hydro$Date)
```

```
## Time difference of 2191 days
```

Finally, the `Date` class is very handy when plotting. Let's make a simple graph of the Hydro dataset. The following code produces Fig. 2.1. Note how the `X` axis is automatically formatted to display the date in a (fairly) pretty way.

```r
ggplot(hydro, aes(x = Date, y = storage)) +
  geom_line()
```

2.7.5.3 Date-Time combinations

For dates that include the time, R has a special class called `POSIXct`. The `lubridate` package makes it easy to work with this class.

Internally, a date-time is represented as the number of seconds since the 1st of January, 1970. Time zones are also supported, but we will not use this functionality in this book (as it can be quite confusing).

From the lubridate package, we can use any combination of (y)ear,(m)onth,(d)ay, (h)our, (m)inutes, (s)econds. For example `ymd_hms` converts a character string in that order.

Let's look at some examples,

```
# Load lubridate
library(lubridate)

# The standard format is YYYY-MM-DD HH:MM:SS
ymd_hms("2012-9-16 13:05:00")
```

```
## [1] "2012-09-16 13:05:00 UTC"
```

```
# Read two times (note the first has no seconds, so we can use ymd_hm)
time1 <- ymd_hm("2008-5-21 9:05")
time2 <- ymd_hms("2012-9-16 13:05:00")

# Time difference:
time2 - time1
```

```
## Time difference of 1579.167 days
```

```
# And an example with a different format, DD/M/YY H:MM
dmy_hm("23-1-89 4:30")
```

```
## [1] "1989-01-23 04:30:00 UTC"
```

```
# To convert a date-time to a Date, you can also use the as.Date function,
# which will simply drop the time.
as.Date(time1)
```

```
## [1] "2008-05-21"
```

As with `Date` objects, we can calculate timespans using a few handy functions.

```
# What time is it 3 hours and 15 minutes from now?
now() + hours(3) + minutes(15)
```

```
## [1] "2020-03-04 01:47:18 CET"
```

Try this yourself The 2012 Sydney marathon started at 7:20AM on September 16th. The winner completed the race in 2 hours, 11 minutes and 50 seconds. What was the time when the racer crossed the finish line? Using the `weekdays` function, which day was the race held?

2.7.5.4 Example: date-times in a dataframe

Now let's use a real dataset to practice the use of date-times. We also introduce the functions `month`, `yday`, `hour` and `minute` to conveniently extract components of date-time objects.

The last command produces Fig. 2.2.

```
# Read the 2008 met dataset from the HFE.
data(hfemet2008)

# Convert 'DateTime' to POSIXct class.
# The order of the original data is MM/DD/YYYY HH:MM
```

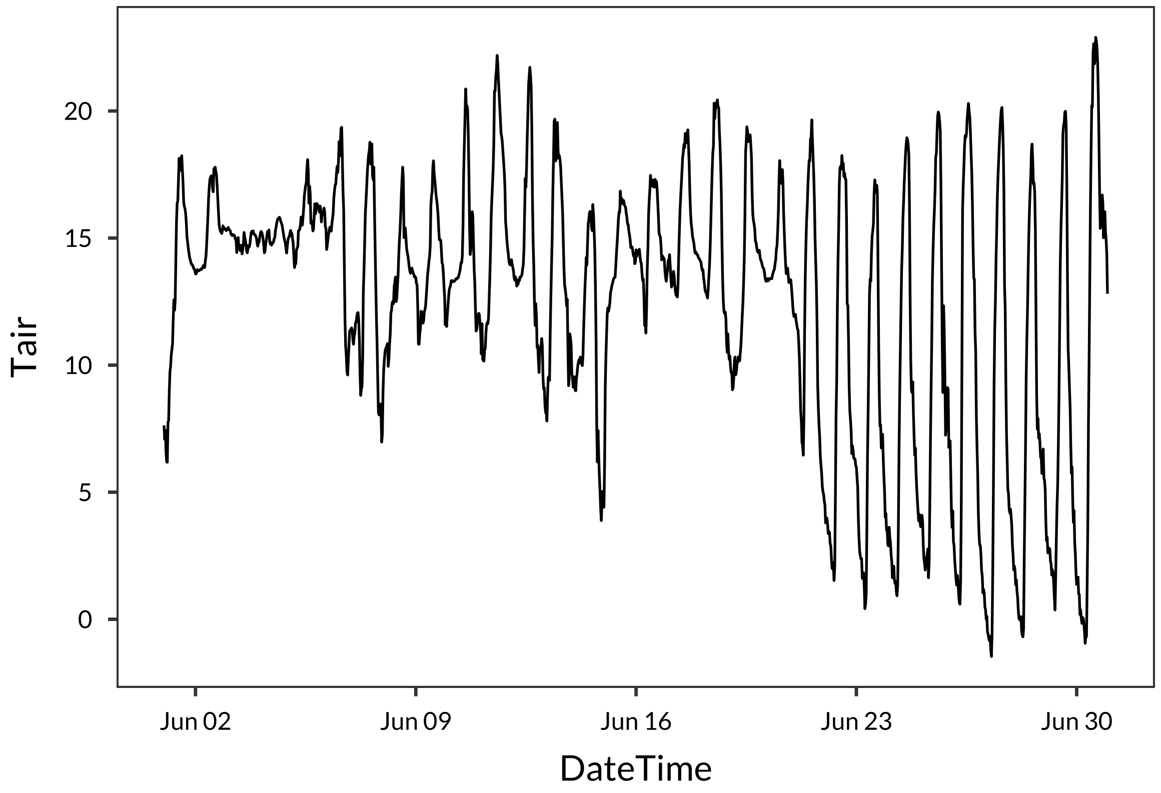


Figure 2.2: Air temperature for June at the HFE

```r
# We also add various date and time fields to the dataframe.
hfemet <- hfemet2008 %>%
  mutate(DateTime = mdy_hm(DateTime),
         Date = as.Date(DateTime),
         DOY = yday(DateTime),
         hour = hour(DateTime),
         minute = minute(DateTime),
         month = month(DateTime))

# Make sure all datetimes were converted OK (if not, NAs would be produced)
any(is.na(hfemet$DateTime))
```

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

```r
# FALSE is good here!

# Make a simple line plot for data from one month.
filter(hfemet, month ==  6) %>%
  ggplot(aes(x = DateTime, y =  Tair)) +
  geom_line()
```

```r
# We can also take a subset of just one day, using the Date variable we added:
hfemet_oneday <- filter(hfemet, Date == as.Date("2008-11-1"))
```

2.7.5.5 Sequences of dates and times

It is often useful to generate sequences of dates. We can use `seq` as we do for numeric variables (as we already saw in Section 2.2.1).

```
# A series of dates, by day:
seq(from=as.Date("2011-1-1"), to=as.Date("2011-1-5"), by="day")
```

```
## [1] "2011-01-01" "2011-01-02" "2011-01-03" "2011-01-04" "2011-01-05"
```

```
# Two-weekly dates:
seq(from=as.Date("2011-1-1"), length=4, by="2 weeks")
```

```
## [1] "2011-01-01" "2011-01-15" "2011-01-29" "2011-02-12"
```

```
# Monthly:
seq(from=as.Date("2011-12-13"), length=4, by="months")
```

```
## [1] "2011-12-13" "2012-01-13" "2012-02-13" "2012-03-13"
```

Similarly, you can generate sequences of date-times.

```
# Generate a sequence with 30 min timestep:
# Here, the 'by' field specifies the timestep in seconds.
fromtime <- ymd_hm("2012-6-1 14:30")
seq(from=fromtime, length=3, by=30*60)
```

```
## [1] "2012-06-01 14:30:00 UTC" "2012-06-01 15:00:00 UTC"
## [3] "2012-06-01 15:30:00 UTC"
```

2.7.6 Converting between data types

It is often useful, or even necessary, to convert from one data type to another. For example, when you read in data with `read.csv` or `read.table`, any column that contains some non-numeric values (that is, values that cannot be converted to a number) will be converted to a factor variable. Sometimes you actually want to convert it to numeric, which will result in some missing values (`NA`) when the value could not be converted to a number.

Another common example is when one of your variables should be read in as a factor variable (for example, a column with treatment codes), but because all the values are numeric, R will simply assume it is a numeric column.

Before we learn how to convert, it is useful to make sure you know what type of data you have to begin with. To find out what type of data a particular vector is, we use `str` (this is also useful for any other object in R).

```
# Numeric
y <- c(1,100,10)
str(y)
```

```
##  num [1:3] 1 100 10
```

```
# This example also shows the dimension of the vector ([1:3]).

# Character
```

```r
x <- "sometext"
str(x)
```

```
##  chr "sometext"
```

```r
# Factor
p <- factor(c("apple","banana"))
str(p)
```

```
##  Factor w/ 2 levels "apple","banana": 1 2
```

```r
# Logical
z <- c(TRUE,FALSE)
str(z)
```

```
##  logi [1:2] TRUE FALSE
```

```r
# Date
sometime <- as.Date("1979-9-16")
str(sometime)
```

```
##  Date[1:1], format: "1979-09-16"
```

```r
# Date-Time
library(lubridate)
onceupon <- ymd_hm("1969-8-18 09:00")
str(onceupon)
```

```
##  POSIXct[1:1], format: "1969-08-18 09:00:00"
```

To test for a particular type of data, use the `is.*something*` functions, which give `TRUE` if the object is of that type, for example:

```r
# Test for numeric data type:
is.numeric(c(10,189))
```

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

```r
# Test for character:
is.character("HIE")
```

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

We can convert between types with the `as.*something*` class of functions.

```r
# First we make six example values that we will use to convert
mynum <- 1001
mychar <- c("1001","100 apples")
myfac <- factor(c("280","400","650"))
mylog <- c(TRUE,FALSE,FALSE,TRUE)
mydate <- as.Date("2015-03-18")
mydatetime <- ymd_hm("2011-8-11 16:00")

# A few examples:

# Convert to character
as.character(mynum)
```

```
## [1] "1001"
```

```r
as.character(myfac)
```

```
## [1] "280" "400" "650"
```

```r
# Convert to numeric
# Note that one missing value is created
as.numeric(mychar)
```

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

```
## [1] 1001   NA
```

```r
# Warning!!!
# When converting from a factor to numeric, first convert to character
# !!!
as.numeric(as.character(myfac))
```

```
## [1] 280 400 650
```

```r
# Convert to Date
as.Date(mydatetime)
```

```
## [1] "2011-08-11"
```

```r
# Convert to factor
as.factor(mylog)
```

```
## [1] TRUE  FALSE FALSE TRUE
## Levels: FALSE TRUE
```

Caution When converting a factor to numeric (i.e. when the factor labels can be converted to numeric values), first convert it to character: `x <- as.numeric(as.character(somefactor))`. The reason for this is that factor variables are internally stored as integer numbers, referring to the *levels* of the factor.

2.8 Exercises

2.8.1 Working with a single vector - indexing

Recall Exercise 1.14.3. Load the `rainfall` data once more.

We now practice subsetting a vector (see Section 2.5.1).

1. Take a subset of the rainfall data where rain is larger than 20.

2. What is the mean rainfall for days where the rainfall was at least 4?

3. Subset the vector where it is either exactly zero, or exactly 0.6.

2.8.2 Alphabet aerobics 2

The 26 letters of the Roman alphabet are conveniently accessible in R via `letters` and `LETTERS`. These are not functions, but vectors that are always loaded.

1. What is the 18th letter of the alphabet?

2. What is the last letter of the alphabet (pretend you don't know the alphabet has 26 letters)?

3. Use `?sample` to figure out how to sample with replacement. Generate a random subset of fifteen letters. Are any letters in the subset duplicated? *Hint:* use the `any` and `duplicated` functions. Which letters?

2.8.3 Basic operations with the Cereals data

First read or otherwise understand Section 2.4.2.

For this exercise, we will use the `cereals` dataset (from the `lgrdata` package, use `data(cereals)` to load it).

1. Read in the dataset, look at the first few rows with `head` and inspect the data types of the variables in the dataframe with `str`.

2. Add a new variable to the dataset called `totalcarb`, which is the sum of `carbo` and `sugars`. You can do this is in (at least) three ways! (One of which is `mutate` from `dplyr` - a good idea to start using this powerful function).

3. How many cereals in the dataframe are 'hot' cereals? Either make a subset and count the rows, or subset the `Cold.or.Hot` vector directly, and determine its `length` (do not use `table` yet, or a similar function!).

4. How many unique manufacturers are included in the dataset? *Hint:* use `length` and `unique`.

5. Now try the `n_distinct` function from `dplyr`.

6. Take a subset of the dataframe of all cereals that have less than 80 calories, AND have more than 20 units of vitamins.

7. Take a subset of the dataframe containing cereals that contain at least 1 unit of sugar, and keep only the variables 'Cereal.name', 'calories' and 'vitamins'. Then inspect the first few rows of the dataframe with `head`.

8. For one of the above subsets, write a new CSV file to disk using `write.csv` (see Section 2.6).

9. Rename the column 'Manufacturer' to 'Producer' (see Section 2.4.3).

2.8.4 A short dataset

1. Read the following data into R (number of cuckoos seen in a week by an avid birdwatcher). Give the resulting dataframe a reasonable name. *Hint:*To read this dataset, look at Section 2.3.4 for a possibility (there are at least two ways to read this dataset, or you can type it into Excel and save as a CSV file if you prefer).

```
Day nrbirds
sunday 3
monday 2
tuesday 5
wednesday 0
thursday 8
friday 1
saturday 2
```

2. Add a day number to the dataset you read in above (sunday=1, saturday=7). Recall the `seq` function (Section 2.2.1).

3. Delete the 'Day' variable (to only keep the `daynumber` that you added above).

4. On which `daynumber` did you observe the most honeyeaters? *Hint:* use `which.max`, in combination with indexing.

5. Sort the dataset by number of birds seen. *Hint:* use the `order` function to find the order of the number of birds, then use this vector to index the dataframe.

2.8.5 Titanic - Part 1

1. Read the `titanic` data from the `lgrdata` package.

2. Make two new dataframes : a subset of male survivors, and a subset of female survivors. Recall Section 2.5.2. Use `filter` from `dplyr`.

3. Based on the previous question, what was the name of the oldest surviving male? In what class was the youngest surviving female? *Hint:* use `which.max`, `which.min` on the subsets you just created.

The easiest solution here is to use square bracket notation, but you can also solve this question in steps.

4. Take 15 random names of passengers from the Titanic, and sort them alphabetically. *Hint:* use `sort`.

2.8.6 Titanic - Part 2

1. Convert the 'Name' (passenger name) variable to a 'character' variable, and store it in the dataframe. See Section 2.7.4.2.

2. How many observations of 'Age' are missing from the dataframe? See examples in Section 2.7.3.

3. Make a new variable called 'Status', based on the 'Survived' variable already in the dataset. For passengers that did not survive, Status should be 'dead', for those who did, Status should be 'alive'. Make sure this new variable is a factor. See the example with the `ifelse` function in Section 2.7.1.

4. Count the number of passengers in each class (1st, 2nd, 3rd). *Hint:* use `table` as shown in Section 2.7.1.

5. Using `grep`, find the six passengers with the last name 'Fortune'. Make this subset into a new dataframe. Did they all survive? *Hint:* to do this, make sure you recall how to use one vector to index a dataframe (see Section 2.5.2). Also, the `all` function might be useful here (see Section 2.7.2).

6. As in *2.*, for what proportion of the passengers is the age unknown? Was this proportion higher for 3rd class than 1st and 2nd? *Hint:* First make a subset of the dataframe where age is missing (see Section 2.7.3.3), and then use `table`, as well as `nrow`.

2.8.7 Hydro dam

Use the `hydro` dam data (used in Section 2.7.5.2).

1. Start by reading in the data. Change the first variable to a `Date` class (see Section 2.7.5.1).

2. Are the successive measurements in the dataset always exactly one week apart? *Hint:* use `diff`.

3. Assume that a dangerously low level of the dam is 235 Gwh. How many weeks was the dam level equal to or lower than this value?

4. (**Hard question**). For question *2.*, how many times did `storage` decrease below 235 (regardless of how long it remained below 235)? *Hint:* use `diff` and `subset`).

2.8.8 HFE tree measurements

Use the data for an experiment where trees were irrigated and/or fertilized (the `hfeplotmeans` dataset).

1. Read the data, write a copy to a new object called `trees` (easier to type!) and look at various summaries of the dataset. Use `summary`, `str` and `describe` (the latter is in the `Hmisc` package).

2. From these summaries, find out how many missing values there are for `height` and `diameter`. Also count the number of missing values as shown in Section 2.7.3.

3. Inspect the levels of the treatment (`treat`), with the `levels` function. Also count the number of levels with the `nlevels` function. Now assign new levels to the factor, replacing the abbreviations with a more informative label. Follow the example in Section 2.7.1.3.

4. Using `table`, count the number of observations by `treat`, to check if the dataset is balanced. Be aware that `table` simply counts the number of rows, regardless of missing values. Now take a subset of the dataset where `height` is not missing, and check the number of observations again.

5. For which dates do missing values occur in `height` in this dataset? *Hint:* use a combination of `is.na` and `unique`.

2.8.9 Flux data

In this exercise, you will practice working with Dates and Date-Time combinations, with a timeseries dataset (the `fluxtower` dataset from the `lgrdata` package).

In this dataset, a new row was produced every 30min - of various meteorological measurements above a forest in Spain. For example, `FCO2` is the flux of carbon dioxide out of forest, so that negative values indicate photosynthesis.

1. Read the dataframe. Rename the first column to 'DateTime' (recall Section 2.4.3).

2. Convert DateTime to a `POSIXct` class. Beware of the formatting (recall Section 2.7.5.3).

3. Did the above action produce any missing values? Were these already missing in the original dataset?

4. Add a variable to the dataset called 'Quality'. This variable should be 'bad' when the variable 'ustar' is less than 0.15, and 'good' otherwise. Recall the example in Section 2.7.1.

5. Add a 'month' column to the dataset, as well as 'year'.

6. Look at the 'Rain' column. There are some problems; re-read the data or find another way to display `NA` whenever the data have an invalid value. *Hint:* look at the argument `na.strings` in `read.table`.

Did it rain on this forest in Spain?

2.8.10 DNA Aerobics

DNA sequences can also be represented using text strings. In this exercise, you will make an artificial DNA sequence.

1. Make a random DNA sequence, consisting of a 100 random selections of the letters `C`,`A`,`G`,`T`, and paste the result together into one character string (*Hint* : use `sample` as in Section 2.2.2 with replacement, and use `paste` as shown in Section 2.7.4. Write it in one line of R code.

Chapter 3

Data skills - Part 2

3.1 Summarizing dataframes

There are a few useful functions to print general summaries of a dataframe, to see which variables are included, what types of data they contain, and so on. We already looked at `summary` and `str` in Section 2.4.

Two more very useful functions are from the `Hmisc` package. The first, `describe`, is much like `summary`, but offers slightly more sophisticated statistics. The second, `contents`, is similar to `str`, but does a very nice job of summarizing the `factor` variables in your dataframe, prints the number of missing variables, the number of rows, and so on.

```
# Load data
data(pupae)

# Make sure CO2_treatment is a factor (it will be read as a number)
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)

# Show contents:
library(Hmisc)
contents(pupae)
```

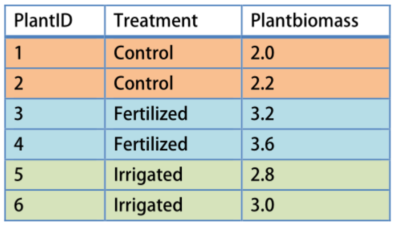
```
##
## Data frame:pupae 84 observations and 5 variables    Maximum # NAs:6
##
##
##                Levels Storage NAs
## T_treatment         2 integer   0
## CO2_treatment       2 integer   0
## Gender                integer   6
## PupalWeight            double   0
## Frass                  double   1
##
## +-------------+----------------+
## |Variable     |Levels          |
## +-------------+----------------+
## |T_treatment  |ambient,elevated|
## +-------------+----------------+
## |CO2_treatment|280,400         |
## +-------------+----------------+
```

Here, `storage` refers to the internal storage type of the variable: note that the factor variables are stored as 'integer', and other numbers as 'double' (this refers to the precision of the number).

3.1.1 Making summary tables

3.1.1.1 Summarizing vectors with `tapply`

If we have the following dataset called `plantdat`,

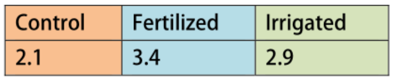


PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and execute the command

```
with(plantdat, tapply(Plantbiomass, Treatment, mean))
```

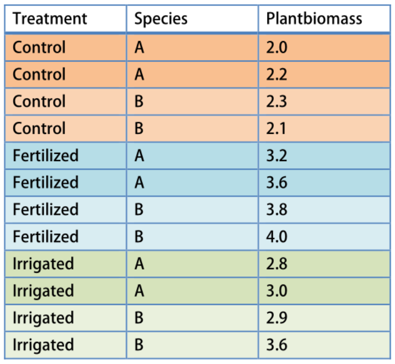
we get the result



Control	Fertilized	Irrigated
2.1	3.4	2.9

Note that the result is a `vector` (elements of a vector can have names, like columns of a dataframe).

If we have the following dataset called `plantdat2`,

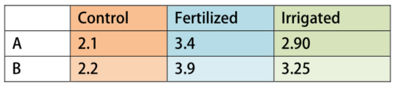


Treatment	Species	Plantbiomass
Control	A	2.0
Control	A	2.2
Control	B	2.3
Control	B	2.1
Fertilized	A	3.2
Fertilized	A	3.6
Fertilized	B	3.8
Fertilized	B	4.0
Irrigated	A	2.8
Irrigated	A	3.0
Irrigated	B	2.9
Irrigated	B	3.6

and execute the command

```
with(plantdat2, tapply(Plantbiomass, list(Species, Treatment), mean))
```

we get the result



	Control	Fertilized	Irrigated
A	2.1	3.4	2.90
B	2.2	3.9	3.25

Note that the result here is a `matrix`, where `A` and `B`, the species codes, are the rownames of this matrix.

Often, you want to summarize a variable by the levels of another variable. For example, in the `rain` data, the `Rain` variable gives daily values, but we might want to calculate annual sums,

```
# Read data
data(rain)

# Annual rain totals.
with(rain, tapply(Rain, Year, FUN=sum))
```

```
##   1996   1997   1998   1999   2000   2001   2002   2003   2004   2005
##  717.2  640.4  905.4 1021.3  693.5  791.5  645.9  691.8  709.5  678.2
```

The `tapply` function applies a function (`sum`) to a vector (`Rain`), that is split into chunks depending on another variable (`Year`).

We can also use the `tapply` function on more than one variable at a time. Consider these examples on the `pupae` data.

```
# Average pupal weight by CO2 and T treatment:
with(pupae, tapply(PupalWeight, list(CO2_treatment, T_treatment), FUN=mean))
```

```
##       ambient elevated
## 280 0.2900000  0.30492
## 400 0.3419565  0.29900
```

```
# Further split the averages, by gender of the pupae.
with(pupae, tapply(PupalWeight, list(CO2_treatment, T_treatment, Gender), FUN=mean))
```

```
## , , 0
##
##      ambient  elevated
## 280 0.251625 0.2700000
## 400 0.304000 0.2687143
##
## , , 1
##
##       ambient  elevated
## 280 0.3406000 0.3386364
## 400 0.3568333 0.3692857
```

As the examples show, the `tapply` function produces summary tables by one or more factors. The resulting object is either a vector (when using one factor), or a matrix (as in the examples using the pupae data).

The limitations of `tapply` are that you can only summarize one variable at a time, and that the result is not a dataframe.

The main advantage of `tapply` is that we can use it as input to `barplot`, as the following example demonstrates (Fig. fig:pupgroupedbar})

```
# Pupal weight by CO2 and Gender. Result is a matrix.
pupm <- with(pupae, tapply(PupalWeight, list(CO2_treatment,Gender),
                           mean, na.rm=TRUE))

# When barplot is provided a matrix, it makes a grouped barplot.
```

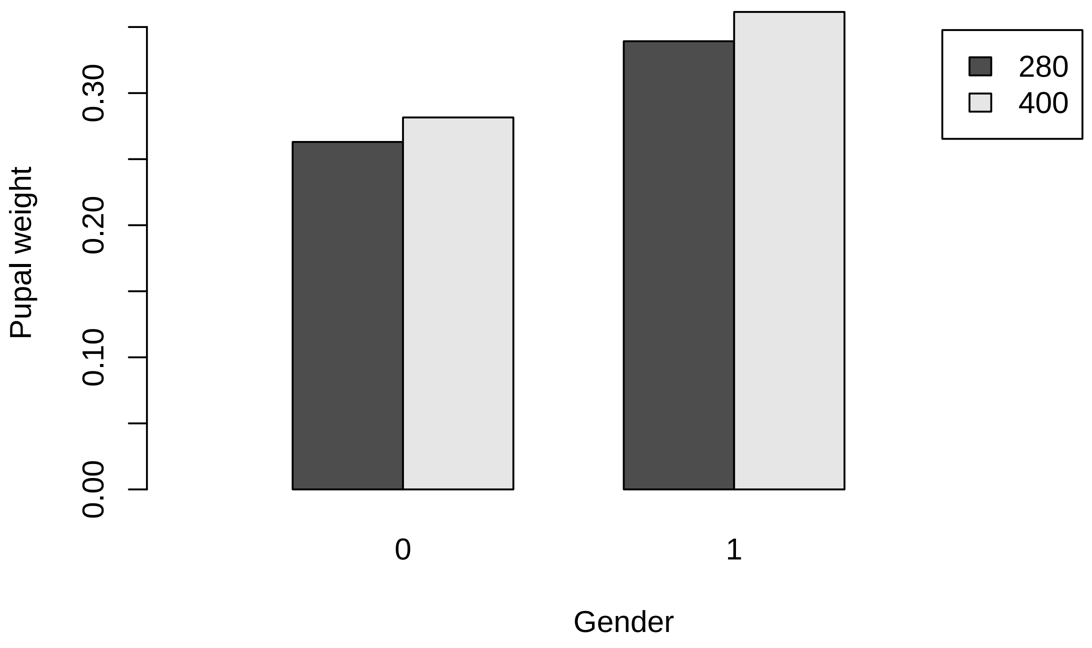
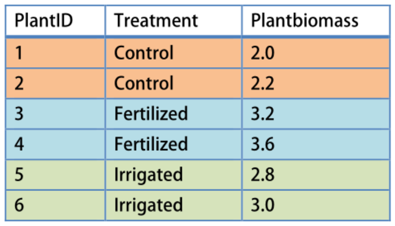


Figure 3.1: A grouped barplot of average pupal weight by CO2 and Gender for the pupae dataset. This is easily achieved via the use of tapply.

```r
# We specify xlim to make some room for the legend.
barplot(pupm, beside=TRUE, legend.text=TRUE, xlim=c(0,8),
        xlab="Gender", ylab="Pupal weight")
```

3.1.1.2 Quick summary tables with `summaryBy`

If we have the following dataset called `plantdat`,

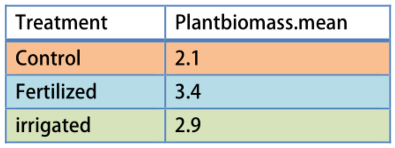


PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and execute the command

```r
library(doBy)
summaryBy(Plantbiomass ~ treatment, FUN=mean, data=plantdat)
```

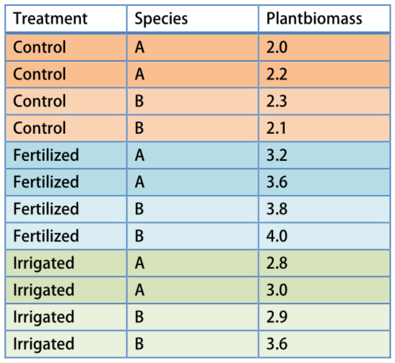
we get the result



Treatment	Plantbiomass.mean
Control	2.1
Fertilized	3.4
irrigated	2.9

Note that the result here is a `dataframe`.

If we have the following dataset called `plantdat2`,

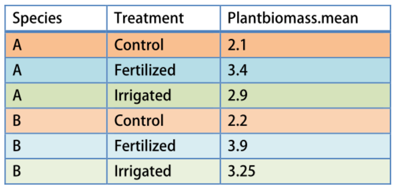


Treatment	Species	Plantbiomass
Control	A	2.0
Control	A	2.2
Control	B	2.3
Control	B	2.1
Fertilized	A	3.2
Fertilized	A	3.6
Fertilized	B	3.8
Fertilized	B	4.0
Irrigated	A	2.8
Irrigated	A	3.0
Irrigated	B	2.9
Irrigated	B	3.6

and execute the command

```
summaryBy(Plantbiomass ~ Species + Treatment, FUN=mean, data=dfr)
```

we get the result



Species	Treatment	Plantbiomass.mean
A	Control	2.1
A	Fertilized	3.4
A	Irrigated	2.9
B	Control	2.2
B	Fertilized	3.9
B	Irrigated	3.25

Note that the result here is a `dataframe`.

In practice, it is often useful to make summary tables of multiple variables at once, and to end up with a dataframe. In this book we first use `summaryBy`, from the `doBy` package, to achieve this.

With `summaryBy`, we can generate multiple summaries (mean, standard deviation, etc.) on more than one variable in a dataframe at once. We can use a convenient formula interface for this. It is of the form,

```
summaryBy(Yvar1 + Yvar2 ~ Groupvar1 + Groupvar2, FUN=c(mean,sd), data=mydata)
```

where we summarize the (numeric) variables `Yvar1` and `Yvar2` by all combinations of the (factor) variables `Groupvar1` and `Groupvar2`.

```
# Load the doBy package
library(doBy)

# read pupae data if you have not already
data(pupae)

# Get mean and standard deviation of Frass by CO2 and T treatments
```

```
summaryBy(Frass ~ CO2_treatment + T_treatment,
          data=pupae, FUN=c(mean,sd))
```

```
##   CO2_treatment T_treatment Frass.mean  Frass.sd
## 1           280     ambient         NA        NA
## 2           280    elevated   1.479520 0.2387150
## 3           400     ambient   2.121783 0.4145402
## 4           400    elevated   1.912045 0.3597471
```

```
# Note that there is a missing value. We can specify na.rm=TRUE,
# which will be passed to both mean() and sd(). It works because those
# functions recognize that argument (i.e. na.rm is NOT an argument of
# summaryBy itself!)
summaryBy(Frass ~ CO2_treatment + T_treatment,
          data=pupae, FUN=c(mean,sd), na.rm=TRUE)
```

```
##   CO2_treatment T_treatment Frass.mean  Frass.sd
## 1           280     ambient   1.953923 0.4015635
## 2           280    elevated   1.479520 0.2387150
## 3           400     ambient   2.121783 0.4145402
## 4           400    elevated   1.912045 0.3597471
```

```
# However, if we use a function that does not recognize it, we first have to
# exclude all missing values before making a summary table, like this:
pupae_nona <- pupae[complete.cases(pupae),]

# Get mean and standard deviation for
# the pupae data (Pupal weight and Frass), by CO2 and T treatment.
# Note that length() does not recognize na.rm (see ?length), which is
# why we have excluded any NA from pupae first.
summaryBy(PupalWeight+Frass ~ CO2_treatment + T_treatment,
          data=pupae_nona,
          FUN=c(mean,sd,length))
```

```
##   CO2_treatment T_treatment PupalWeight.mean Frass.mean PupalWeight.sd
## 1           280     ambient        0.2912500   1.957333     0.04895847
## 2           280    elevated        0.3014583   1.473167     0.05921000
## 3           400     ambient        0.3357000   2.103250     0.05886479
## 4           400    elevated        0.3022381   1.931000     0.06602189
##    Frass.sd PupalWeight.length Frass.length
## 1 0.4192227                 12           12
## 2 0.2416805                 24           24
## 3 0.4186310                 20           20
## 4 0.3571969                 21           21
```

You can also use any function that returns a vector of results. In the following example we calculate the 5% and 95% quantiles of all numeric variables in the allometry dataset. To do this, use `.` for the left-hand side of the formula.

```
# . ~ species means 'all numeric variables by species'.
# Extra arguments to the function used (in this case quantile) can be set here as well,
# they will be passed to that function (see ?quantile).
data(allometry)
summaryBy(. ~ species, data=allometry, FUN=quantile, probs=c(0.05, 0.95))
```

```
##   species diameter.5% diameter.95% height.5% height.95% leafarea.5%
## 1    PIMO      8.1900      70.9150  5.373000     44.675    7.540916
## 2    PIPO     11.4555      69.1300  5.903499     39.192   10.040843
## 3    PSME      6.1635      56.5385  5.276000     32.602    4.988747
##   leafarea.95% branchmass.5% branchmass.95%
## 1     380.3984      4.283342       333.6408
## 2     250.1295      7.591966       655.1097
## 3     337.5367      3.515638       403.7902
```

3.1.1.3 Summarizing dataframes with `dplyr`

We started with the `summaryBy` package, since it is so easy to use. A more modern and popular approach is to use `dplyr` for all your dataframe summarizing needs. The main advantage is that `dplyr` is very, very fast. For datasets with > hundreds of thousands of rows, you will notice an incredible speed increase. For millions of rows, you really have to use `dplyr` (or `data.table`, but we don't cover that package in this book).

Making summary tables as we did in the above examples requires two steps:

1. Group your dataframe by one or more factor variables
2. Apply summarizing functions to each of the groups

As is the norm, we use the pipe operator (%>%) to keep the steps apart. Here is a simple example to get started.

```
library(dplyr)

group_by(pupae, CO2_treatment, T_treatment) %>%
  summarize(Frass_mean = mean(Frass, na.rm=TRUE),
            Frass_sd = sd(Frass, na.rm=TRUE))
```

```
## # A tibble: 4 x 4
## # Groups:   CO2_treatment [2]
##   CO2_treatment T_treatment Frass_mean Frass_sd
##           <int> <fct>            <dbl>    <dbl>
## 1           280 ambient           1.95    0.402
## 2           280 elevated          1.48    0.239
## 3           400 ambient           2.12    0.415
## 4           400 elevated          1.91    0.360
```

I used `as.data.frame` at the end, so we arrive at an actual dataframe, not a tibble (only for the reason so you can compare the output to the previous examples). The `dplyr` package (and others) always produce tibbles, which are really just dataframes but with some adjusted printing methods.

In `summarize` you can specify each of the new variables that should be produced, in this case giving mean and standard deviation of Frass. If we want to apply a number of functions over many variables, we can use `summarize_at`, like so:

```
group_by(pupae, CO2_treatment, T_treatment) %>%
  summarize_at(.vars = c("Frass", "PupalWeight"),
               .funs = c("mean", "sd"))
```

```
## # A tibble: 4 x 6
## # Groups:   CO2_treatment [2]
##   CO2_treatment T_treatment Frass_mean PupalWeight_mean Frass_sd
##           <int> <fct>           <dbl>           <dbl>    <dbl>
## 1           280 ambient         NA              0.290   NaN
## 2           280 elevated         1.48           0.305    0.239
## 3           400 ambient          2.12           0.342    0.415
## 4           400 elevated         1.91           0.299    0.360
## # ... with 1 more variable: PupalWeight_sd <dbl>
```

The result is identical to our last example with `summaryBy`.

Let's look at a more advanced example using weather data collected at the Hawkesbury Forest Experiment in 2008. The data given are in half-hourly time steps. It is a reasonable request to provide data as daily averages (for temperature) and daily sums (for precipitation).

The following code produces a daily weather dataset, and Fig. 3.2.

```
# Read data, convert DateTime field to a proper Datetime class using
# lubridate's mdy_hm function, and add a Date column with as.Date.
# Instead of loading packages, we use :: in the example below to make sure
# the functions are used from the right package.
data(hfemet2008)

hfemet_agg <- hfemet2008 %>%
  mutate(DateTime = lubridate::mdy_hm(DateTime),
         Date = as.Date(DateTime)) %>%
  group_by(Date) %>%
  summarize(Rain = sum(Rain),
            Tair = mean(Tair))

# A simple plot of daily rainfall.
library(ggplot2)
ggplot(hfemet_agg, aes(x = Date, y = Rain)) +
  geom_bar(stat="identity")
```

3.1.1.4 Using `padr` to aggregate timeseries data

In the previous example we saw a quick and concise methods to aggregate and filter a dataframe that includes a single date-time column (here roughly referred to as timeseries data). In the example, we calculated daily totals and averages, but what if we want to aggregate by other timespans, like two-week intervals, 6 months, or 15 minutes? The `padr` package is very convenient for this sort of mutation.

The following example uses `padr` in combination with several functions from `dplyr` to make a table of total rainfall in 4 hour increments for one day, using the `hfemet2008` dataset.

```
library(dplyr)
library(padr)
library(lubridate)

# From the lgrdata package
data(hfemet2008)

mutate(hfemet2008, DateTime = mdy_hm(DateTime)) %>% # convert to proper DateTime
  filter(as.Date(DateTime) == "2008-6-3") %>%       # select a single day
```

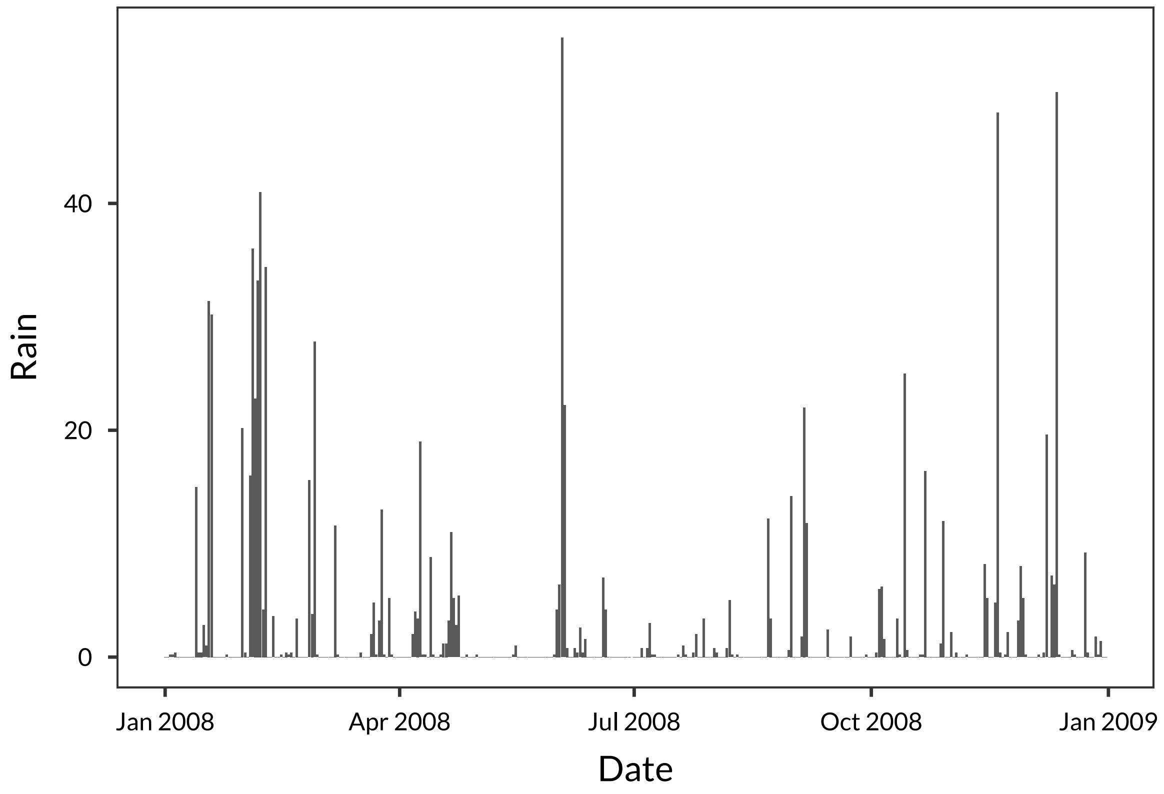


Figure 3.2: Daily rainfall at the HFE in 2008

```
  thicken("4 hours", round="down") %>%           # add datetime in 4hour steps
  group_by(DateTime_4_hour) %>%                  # set grouping variable
  summarize(Rain = sum(Rain)) %>%                # sum over the grouping variable
  select(DateTime = DateTime_4_hour, Rain)       # show two variables
```

```
## # A tibble: 6 x 2
##   DateTime             Rain
##   <dttm>              <dbl>
## 1 2008-06-03 00:00:00   2.4
## 2 2008-06-03 04:00:00   1.8
## 3 2008-06-03 08:00:00  11.8
## 4 2008-06-03 12:00:00   5.6
## 5 2008-06-03 16:00:00   6.6
## 6 2008-06-03 20:00:00  26.4
```

3.1.2 Tables of counts

It is often useful to count the number of observations by one or more multiple factors. One option is to use `tapply` or `summaryBy` in combination with the `length` function. A much better alternative is to use the `xtabs` and `ftable` functions, in addition to the simple use of `table`. Alternatively, `dplyr` provides a `count` function. We will look at both options.

Consider these examples using the Titanic data.

```
# Read titanic data
data(titanic)

# Count observations by passenger class
table(titanic$PClass)
```

```
##
## 1st 2nd 3rd
## 322 280 711
```

```
# With more grouping variables, it is more convenient to use xtabs.
# Count observations by combinations of passenger class, sex, and whether they survived:
xtabs( ~ PClass + Sex + Survived, data=titanic)
```

```
## , , Survived = 0
##
##       Sex
## PClass female male
##    1st      9  120
##    2nd     13  148
##    3rd    132  441
##
## , , Survived = 1
##
##       Sex
## PClass female male
##    1st    134   59
##    2nd     94   25
##    3rd     80   58
```

```
# The previous output is hard to read, consider using ftable on the result:
ftable(xtabs( ~ PClass + Sex + Survived, data=titanic))
```

```
##               Survived   0   1
## PClass Sex
## 1st    female            9 134
##        male            120  59
## 2nd    female           13  94
##        male            148  25
## 3rd    female          132  80
##        male            441  58
```

```
# Using dplyr, the result is a dataframe (actually, a tibble)
library(dplyr)
titanic %>% count(PClass, Sex, Survived)
```

```
## # A tibble: 12 x 4
##    PClass Sex    Survived     n
##    <fct>  <fct>     <int> <int>
##  1 1st    female        0     9
##  2 1st    female        1   134
##  3 1st    male          0   120
##  4 1st    male          1    59
##  5 2nd    female        0    13
##  6 2nd    female        1    94
##  7 2nd    male          0   148
##  8 2nd    male          1    25
##  9 3rd    female        0   132
## 10 3rd    female        1    80
## 11 3rd    male          0   441
## 12 3rd    male          1    58
```

3.1.3 Adding summary variables to dataframes

We saw how `tapply` can make simple tables of averages (or totals, or other functions) of some variable by the levels of one or more factor variables. The result of `tapply` is typically a vector with a length equal to the number of levels of the factor you summarized by (see examples in Section 3.1.1.1).

Consider the `allometry` dataset, which includes tree height for three species. Suppose you want to add a new variable 'MaxHeight', that is the maximum tree height observed per species. We can use `ave` to achieve this:

```
# Read data
allom <- allometry %>%
  mutate(MaxHeight = ave(height, species, FUN=max))

# Look at first few rows (or just type allom to see whole dataset)
head(allom)
```

```
##   species diameter height   leafarea branchmass MaxHeight
## 1    PSME    54.61  27.04 338.485622  410.24638      33.3
## 2    PSME    34.80  27.42 122.157864   83.65030      33.3
## 3    PSME    24.89  21.23   3.958274    3.51270      33.3
## 4    PSME    28.70  24.96  86.350653   73.13027      33.3
## 5    PSME    34.80  29.99  63.350906   62.39044      33.3
## 6    PSME    37.85  28.07  61.372765   53.86594      33.3
```

Note that you can use any function in place of `max`, as long as that function can take a vector as an argument, and returns a single number.

Try this yourself If you want results similar to `ave`, you can use `summaryBy` with the argument `full.dimension=TRUE`. Try `summaryBy` on the `pupae` dataset with that argument set, and compare the result to `full.dimension=FALSE`, which is the default.

3.1.4 Reordering factor levels based on a summary variable

It is often useful to tabulate your data in a meaningful order. We saw that, when using `summaryBy`, `tapply` or similar functions, that the results are always in the order of your factor levels. Recall that the default order is alphabetical. This is rarely what you want.

You can reorder the factor levels by some summary variable. For example,

```
# Reorder factor levels for 'Manufacturer' in the cereal data
# by the mean amount of sodium.

# Read data, show default (alphabetical) levels:
data(cereals)
levels(cereals$Manufacturer)
```

```
## [1] "A" "G" "K" "N" "P" "Q" "R"
```

```
# Now reorder:
cereals <- mutate(cereals,
                  Manufacturer = reorder(Manufacturer, sodium,
                                         median, na.rm=TRUE))

# And inspect the new levels
levels(cereals$Manufacturer)
```

```
## [1] "A" "N" "Q" "P" "K" "G" "R"
```

```
# Tables are now printed in order:
with(cereals, tapply(sodium, Manufacturer, median))
```

```
##     A     N     Q     P     K     G     R
##   0.0   7.5  75.0 160.0 170.0 200.0 200.0
```

This trick comes in handy when making barplots; it is customary to plot them in ascending order if there is no specific order to the factor levels, as in this example.

The following code produces Fig. 3.3.

```
# Here we read the data, add a reordered factor variable,
# and continue with making the summary table used in the plot.
data(coweeta)

coweeta_table <- coweeta %>%
  mutate(species = reorder(species, height, mean, na.rm=TRUE)) %>%
```

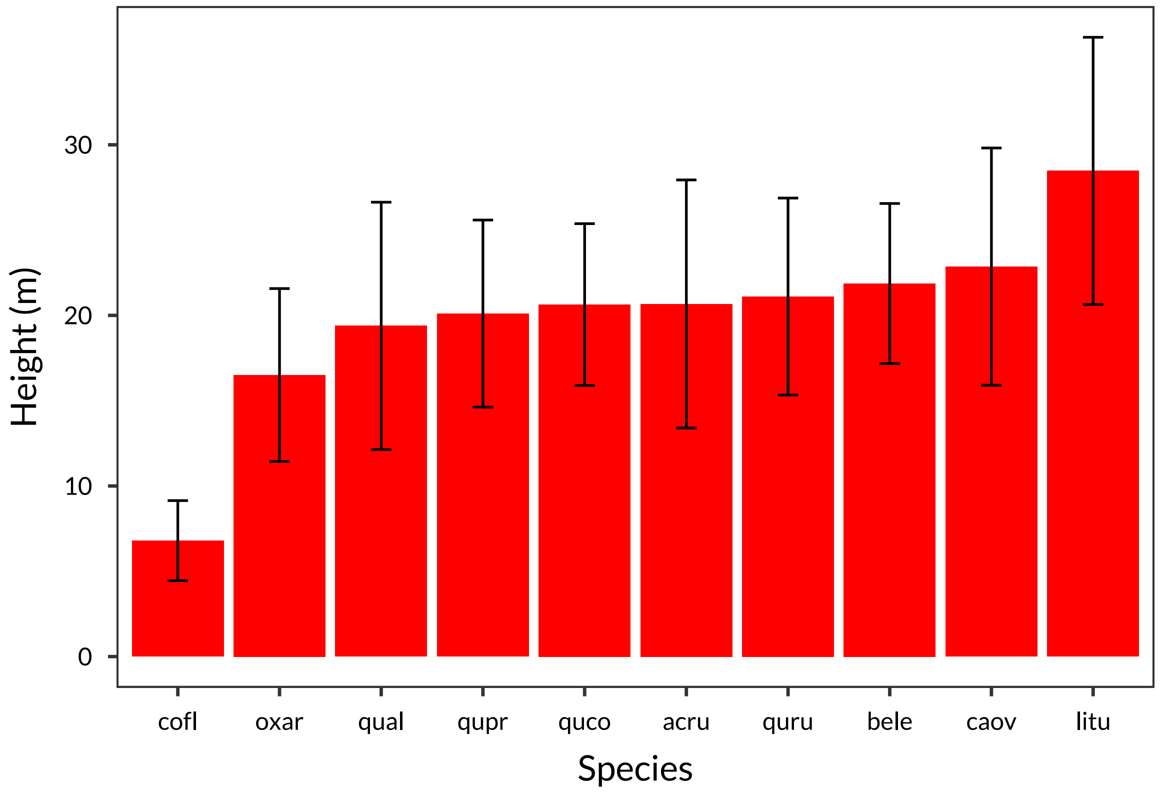


Figure 3.3: An ordered barplot for the coweeta tree data (error bars are 1 SD).

```
  group_by(species) %>%
  dplyr::summarize(height_mean = mean(height, na.rm=TRUE),
            height_sd = sd(height, na.rm=TRUE))

ggplot(coweeta_table, aes(x = species, y = height_mean)) +
  geom_bar(stat="identity", fill="red") +
  geom_errorbar(aes(ymin = height_mean - height_sd,
                  ymax = height_mean + height_sd), width=0.2) +
  labs(y = "Height (m)", x = "Species")
```

The above example uses the more modern approach with `ggplot2` and `dplyr`, but we can get practically the same result with `doBy` and `gplots` - below is the code for comparison (output not shown).

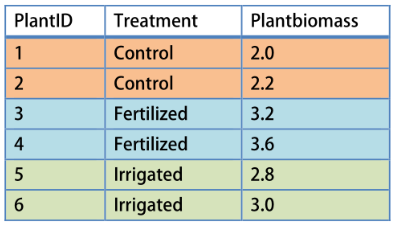
```
library(doBy)
coweeta$species <- with(coweeta, reorder(species, height, mean, na.rm=TRUE))
coweeta_agg <- summaryBy(height ~ species, data=coweeta, FUN=c(mean,sd))

# For barplot2, which adds options for error bars
library(gplots)

# This par setting makes the x-axis labels vertical, so they don't overlap.
par(las=2)
with(coweeta_agg, barplot2(height.mean, names.arg=species,
                           space=0.3, col="red",plot.grid=TRUE,
                           ylab="Height (m)",
                           plot.ci=TRUE,
                           ci.l=height.mean - height.sd,
                           ci.u=height.mean + height.sd))
```

Try this yourself The above example orders the factor levels by increasing median sodium levels. Try reversing the factor levels, using the following code after `reorder`. `coweeta$species <- factor(coweeta$species, levels=rev(levels(coweeta$species)))` Here we used `rev` to reverse the levels.

3.2 Combining dataframes

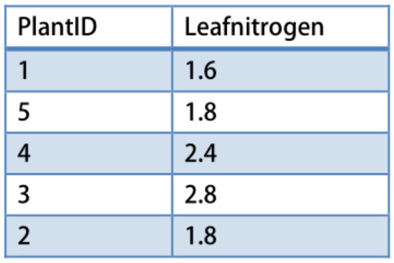
3.2.1 Merging dataframes

If we have the following dataset called `plantdat`,



PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and we have another dataset, that includes the same `PlantID` variable (but is not necessarily ordered, nor does it have to include values for every plant):

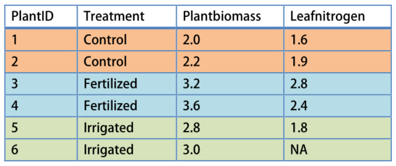


PlantID	Leafnitrogen
1	1.6
5	1.8
4	2.4
3	2.8
2	1.8

and execute the command

```
merge(plantdat, leafnitrogendata, by="PlantID")
```

we get the result



PlantID	Treatment	Plantbiomass	Leafnitrogen
1	Control	2.0	1.6
2	Control	2.2	1.9
3	Fertilized	3.2	2.8
4	Fertilized	3.6	2.4
5	Irrigated	2.8	1.8
6	Irrigated	3.0	NA

Note the missing value (`NA`) for the plant for which no leaf nitrogen data was available.

In many problems, you do not have a single dataset that contains all the measurements you are interested in – unlike most of the example datasets in this tutorial. Suppose you have two datasets that you would like to combine, or `merge`. This is straightforward in R, but there are some pitfalls.

Let's start with a common situation when you need to combine two datasets that have a different number of rows.

```
# Two dataframes
data1 <- data.frame(unit=c("x","x","x","y","z","z"),Time=c(1,2,3,1,1,2))
data2 <- data.frame(unit=c("y","z","x"), height=c(3.4,5.6,1.2))

# Look at the dataframes
data1
```

```
##   unit Time
## 1    x    1
## 2    x    2
## 3    x    3
## 4    y    1
## 5    z    1
## 6    z    2
```

```
data2
```

```
##   unit height
## 1    y    3.4
## 2    z    5.6
## 3    x    1.2
```

```
# Merge dataframes:
combdata <- merge(data1, data2, by="unit")

# Combined data
combdata
```

```
##   unit Time height
## 1    x    1    1.2
## 2    x    2    1.2
## 3    x    3    1.2
## 4    y    1    3.4
## 5    z    1    5.6
## 6    z    2    5.6
```

Sometimes, the variable you are merging with has a different name in either dataframe. In that case, you can either rename the variable before merging, or use the following option:

```
merge(data1, data2, by.x="unit", by.y="item")
```

Where `data1` has a variable called 'unit', and `data2` has a variable called 'item'.

Other times you need to merge two dataframes with multiple key variables. Consider this example, where two dataframes have measurements on the same units at some of the the same times, but on different variables:

```
# Two dataframes
data1 <- data.frame(unit=c("x","x","x","y","y","y","z","z","z"),
Time=c(1,2,3,1,2,3,1,2,3),
Weight=c(3.1,5.2,6.9,2.2,5.1,7.5,3.5,6.1,8.0))
data2 <- data.frame(unit=c("x","x","y","y","z","z"),
Time=c(1,2,2,3,1,3),
Height=c(12.1,24.4,18.0,30.8,10.4,32.9))

# Look at the dataframes
data1
```

```
##   unit Time Weight
## 1    x    1    3.1
## 2    x    2    5.2
## 3    x    3    6.9
## 4    y    1    2.2
## 5    y    2    5.1
## 6    y    3    7.5
## 7    z    1    3.5
## 8    z    2    6.1
## 9    z    3    8.0
```

```
data2
```

```
##   unit Time Height
## 1    x    1   12.1
## 2    x    2   24.4
## 3    y    2   18.0
## 4    y    3   30.8
## 5    z    1   10.4
## 6    z    3   32.9
```

```
# Merge dataframes:
combdata <- merge(data1, data2, by=c("unit","Time"))

# By default, only those times appear in the dataset that have measurements
# for both Weight (data1) and Height (data2)
combdata
```

```
##   unit Time Weight Height
## 1    x    1    3.1   12.1
## 2    x    2    5.2   24.4
## 3    y    2    5.1   18.0
## 4    y    3    7.5   30.8
## 5    z    1    3.5   10.4
## 6    z    3    8.0   32.9
```

```
# To include all data, use this command. This produces missing values for some times:
merge(data1, data2, by=c("unit","Time"), all=TRUE)
```

```
##   unit Time Weight Height
## 1    x    1    3.1   12.1
## 2    x    2    5.2   24.4
## 3    x    3    6.9     NA
## 4    y    1    2.2     NA
## 5    y    2    5.1   18.0
## 6    y    3    7.5   30.8
## 7    z    1    3.5   10.4
## 8    z    2    6.1     NA
## 9    z    3    8.0   32.9
```

```
# Compare this result with 'combdata' above!
```

3.2.2 Using join from `dplyr`

We showed how to use the `merge` function above, which is provided by base R. For larger datasets, it is advisable to use the `join*` functions from `dplyr`.

Instead of specifying which rows to keep with arguments `all.x`, `all`, etc., `dplyr` provides several functions that should make some intuitive sense. The table below compares `merge` and `join*`.

`merge()`	`dplyr::join*`
`merge(dat1, dat2, all = FALSE)`	`inner_join(dat1, dat2)`
`merge(dat1, dat2, all.x = TRUE)`	`left_join(dat1, dat2)`
`merge(dat1, dat2, all.y = TRUE)`	`right_join(dat1, dat2)`
`merge(dat1, dat2, all = TRUE)`	`full_join(dat1, dat2)`

One other function is provided that has no simple equivalent in base R, `anti_join`, which can be used to find all observations that have *no match* between the two datasets. This can be handy for error-checking.

Consider the cereal dataset, which gives measurements of all sorts of contents of cereals. Suppose the measurements for ‘protein’, ‘vitamins’ and ‘sugars’ were all produced by different laboratories, and each lab sent you a separate dataset. To make things worse, some measurements for sugars and vitamins are missing, because samples were lost in those labs.

```
# Read the three datasets given to you from the three different labs:
data(cereal1)
data(cereal2)
data(cereal3)

# As always, look at the first few rows of each dataset.
head(cereal1, 3)
```

```
##      Cereal.name protein
## 1 Frosted_Flakes       1
## 2     Product_19       3
## 3  Count_Chocula       1
```

```
head(cereal2, 3)
```

```
##     cerealbrand vitamins
## 1    Product_19      100
## 2 Count_Chocula       25
## 3    Wheat_Chex       25
```

```
head(cereal3, 3)
```

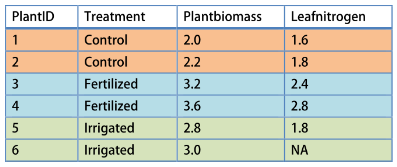
```
##             cerealname sugars
## 1       Frosted_Flakes     11
## 2           Product_19      3
## 3 Mueslix_Crispy_Blend     13
```

```r
# The name of the variable that ties the datasets together,
# the 'cereal name' differs between the datasets, as do the number of rows.
# We can use merge() three times, but the data are easiest to merge with dplyr:
library(dplyr)

cereal_combined <- full_join(cereal1, cereal2, by=c("Cereal.name" = "cerealbrand")) %>%
                   full_join(cereal3, by=c("Cereal.name" = "cerealname"))

cereal_combined
```

```
##                  Cereal.name protein vitamins sugars
## 1             Frosted_Flakes       1       NA     11
## 2                 Product_19       3      100      3
## 3              Count_Chocula       1       25     NA
## 4                 Wheat_Chex       3       25     NA
## 5                 Honey-comb       1       25     NA
## 6   Shredded_Wheat_spoon_size       3        0     NA
## 7        Mueslix_Crispy_Blend       3       25     13
## 8           Grape_Nuts_Flakes       3       25      5
## 9    Strawberry_Fruit_Wheats       2       NA      5
## 10                  Cheerios       6       25      1
# Note that missing values (NA) have been inserted where some data were not available.
```

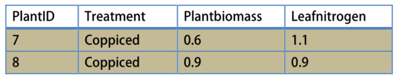
3.2.3 Row-binding dataframes

If we have the following dataset called `plantdat`,



PlantID	Treatment	Plantbiomass	Leafnitrogen
1	Control	2.0	1.6
2	Control	2.2	1.8
3	Fertilized	3.2	2.4
4	Fertilized	3.6	2.8
5	Irrigated	2.8	1.8
6	Irrigated	3.0	NA

and we have another dataset (`plantdatmore`), *with exactly the same columns* (including the names and order of the columns),

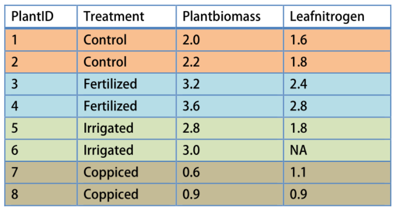


PlantID	Treatment	Plantbiomass	Leafnitrogen
7	Coppiced	0.6	1.1
8	Coppiced	0.9	0.9

and execute the command

```r
rbind(plantdat, plantdatmore)
```

we get the result



PlantID	Treatment	Plantbiomass	Leafnitrogen
1	Control	2.0	1.6
2	Control	2.2	1.8
3	Fertilized	3.2	2.4
4	Fertilized	3.6	2.8
5	Irrigated	2.8	1.8
6	Irrigated	3.0	NA
7	Coppiced	0.6	1.1
8	Coppiced	0.9	0.9

Using `merge`, we were able to glue dataframes together side-by-side based on one or more 'index' variables. Sometimes you have multiple datasets that can be glued together top-to-bottom, for example when you have multiple very similar dataframes. We can use the `rbind` function, like so:

```
# Some fake data
mydata1 <- data.frame(var1=1:3, var2=5:7)
mydata2 <- data.frame(var1=4:6, var2=8:10)

# The dataframes have the same column names, in the same order:
mydata1
```

```
##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
```

```
mydata2
```

```
##   var1 var2
## 1    4    8
## 2    5    9
## 3    6   10
```

```
# So we can use rbind to row-bind them together:
rbind(mydata1, mydata2)
```

```
##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    4    8
## 5    5    9
## 6    6   10
```

Let's look at the above `rbind` example again but with a modification where some observations are duplicated between dataframes. This might happen, for example, when working with files containing time-series data and where there is some overlap between the two datasets. The `union` function from the `dplyr` package only returns unique observations:

```
# Some fake data
mydata1 <- data.frame(var1=1:3, var2=5:7)
mydata2 <- data.frame(var1=2:4, var2=6:8)

# The dataframes have the same column names, in the same order:
mydata1
```

```
##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
```

```r
mydata2
```

```
##   var1 var2
## 1    2    6
## 2    3    7
## 3    4    8
```

```r
# 'rbind' leads to duplicate observations, 'union' removes these:
dplyr::union(mydata1, mydata2)
```

```
##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    4    8
```

```r
rbind(mydata1, mydata2)
```

```
##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    2    6
## 5    3    7
## 6    4    8
```

Sometimes, you want to `rbind` dataframes together but the column names do not exactly match. One option is to first process the dataframes so that they do match (using subscripting). Or, just use the `bind_rows` function from `dplyr`. Look at this example where we have two dataframes that have only one column in common, but we want to keep all the columns (and fill with `NA` where necessary),

```r
# Some fake data
mydata1 <- data.frame(index=c("A","B","C"), var1=5:7)
mydata2 <- data.frame(var1=8:10, species=c("one","two","three"))

# smartbind the dataframes together
dplyr::bind_rows(mydata1, mydata2)
```

```
##   index var1 species
## 1     A    5    <NA>
## 2     B    6    <NA>
## 3     C    7    <NA>
## 4  <NA>    8     one
## 5  <NA>    9     two
## 6  <NA>   10   three
```

Note: an equivalent function to bind dataframes side-by-side is `cbind`, which can be used instead of `merge` when no index variables are present. However, in this book, the use of `cbind` is discouraged for dataframes as it can lead to problems that are difficult to fix, and in all practical applications a merge is preferable.

3.3 Reshaping data

3.3.1 From wide to long

In the majority of analyses in R, we like to have our data in 'long' format (nowadays sometimes called the 'tidy' format), where the data for some kind of measurement are in one column, and we have one or more factor variables distinguishing groups like individual, date, treatment, and so on. It is however common encounter data in 'wide format', which can be converted to long format by reshaping appropriately.

The first example uses the dutch election data.

```
data(dutchelection)
head(dutchelection,3)
```

```
##         Date  VVD PvdA  PVV CDA   SP D66  GL  CU SGP PvdD FiftyPlus
## 1 2012-03-22 22.1 16.8 13.9 9.4 16.8 7.7 4.5 3.3 1.5  2.4       1.1
## 2 2012-04-05 23.6 17.1 13.3 8.8 16.3 8.7 4.1 3.2 1.4  2.0       0.8
## 3 2012-04-19 24.0 17.3 12.0 8.2 17.0 8.8 3.5 3.3 1.6  3.1       0.8
```

The data include percentage votes for 11 Dutch political parties in a 2012 election, and somewhat intuitively the parties have been ordered as columns (so that each row adds up to ca. 100%, ignoring some tiny parties). For purposes of analysis, we would like to reshape this dataset to long format, so that we have just columns 'Date', 'Party', and 'Poll'. Right now 'Party' is in the column names, and 'Poll' represents the polling numbers; the actual data in the cells.

As is often the case, there are many ways to solve this. For most basic reshaping tasks we recommend the `tidyr` package, however some other methods may be needed for more complex tasks, as we find in further examples below.

```
library(tidyr)
# The new dataframe will have a column 'Party' with current columns,
# *except* Date (hence the -Date),
# and values will be stored in 'Poll'
elect_long <- gather(dutchelection, Party, Poll, -Date)
head(elect_long,3)
```

```
##         Date Party Poll
## 1 2012-03-22   VVD 22.1
## 2 2012-04-05   VVD 23.6
## 3 2012-04-19   VVD 24.0
```

```
# Identical results can be had with melt() from reshape2,
# a function which otherwise includes more options.
# See `?melt.data.frame` for more options (not `?melt`,
# which is the generic function).
library(reshape2)
elect_long <- melt(dutchelection, variable.name="Party", value.name="Poll", id.vars="Date")
```

The advantage of `melt` is that we can include more than one ID variables, like in the following example. This example also shows the common need for some text processing after reshaping.

In this simple dataset, length of feet and hands was measured on two persons on three consecutive days. We want to reshape it to end up with columns 'Day', 'Person', 'Length' and 'BodyPart' (feet or hands).

```
dat <- data.frame(Day=rep(1:3, each=2), Person=rep(letters[1:2], 3),
                  Length.feet = rep(c(2,3), 3),
                  Length.hands=rep(c(3,4),3))
dat
```

```
##   Day Person Length.feet Length.hands
## 1   1      a           2            3
## 2   1      b           3            4
## 3   2      a           2            3
## 4   2      b           3            4
## 5   3      a           2            3
## 6   3      b           3            4
```

```
# Now reshape to long format, using melt.data.frame
dat_long <- melt(dat, id.vars=c("Day","Person"), value.name="Length",
    variable.name="BodyPart") %>%
  mutate(BodyPart = gsub("Length.", "", BodyPart))

dat_long
```

```
##    Day Person BodyPart Length
## 1    1      a     feet      2
## 2    1      b     feet      3
## 3    2      a     feet      2
## 4    2      b     feet      3
## 5    3      a     feet      2
## 6    3      b     feet      3
## 7    1      a    hands      3
## 8    1      b    hands      4
## 9    2      a    hands      3
## 10   2      b    hands      4
## 11   3      a    hands      3
## 12   3      b    hands      4
```

3.3.2 From long to wide

Occasionally we like to use wide format, where groups of data are placed next to each other instead of on top of each other. For the first example consider this simple dataset with a set of questions asked to two persons,

```
survey <- read.table(text="user question    answer
a   question_1  hi
a   question_2  hey
a   question_3  oh
a   question_4  no
a   question_5  yes
a   question_6  obv
b   question_1  cool
b   question_2  good
b   question_3  yes
b   question_4  sweet
b   question_5  wow
b   question_6  no", header=TRUE)

survey
```

```
##    user   question answer
## 1     a question_1     hi
## 2     a question_2    hey
## 3     a question_3     oh
## 4     a question_4     no
## 5     a question_5    yes
## 6     a question_6    obv
## 7     b question_1   cool
## 8     b question_2   good
## 9     b question_3    yes
## 10    b question_4  sweet
## 11    b question_5    wow
## 12    b question_6     no
```

In this case we actually want to have all data for each 'user' in a single row, thus creating columns 'question_1', 'question_2' and so on. The first solution uses `spread` from `tidyr`.

```
library(tidyr)
spread(survey, question, answer)
```

```
##   user question_1 question_2 question_3 question_4 question_5 question_6
## 1    a         hi        hey         oh         no        yes        obv
## 2    b       cool       good        yes      sweet        wow         no
```

The second solution uses `dcast` from `reshape2`. I show this solution because unlike `spread`, `dcast` can be used for more complicated reshaping problems.

```
library(reshape2)
dcast(survey, user ~ question, value.var="answer")
```

```
##   user question_1 question_2 question_3 question_4 question_5 question_6
## 1    a         hi        hey         oh         no        yes        obv
## 2    b       cool       good        yes      sweet        wow         no
```

Here, the formula indicates 'user goes in rows, question in columns', and `value.var` is the name of the variable used to populate the cells.

For the second, more complex, example we use a small version of the `eucface_gasexchange` dataset, which includes measurements of photosynthesis (`Photo`) on two experimental plots (`Plot`) on three Dates (`Date`), under two experimental treatments (`treatment`). Clearly we have multiple levels of nesting of our data, and each of these nesting levels is represented by a different variable.

```
gas <- read.table(text="Date    Plot    treatment   Photo
A   1   Amb 14.4
A   1   Ele 16.3
A   2   Amb 17.8
A   2   Ele 13.3
B   1   Amb 16.4
B   1   Ele 19
B   2   Amb 15
B   2   Ele 10.8
C   1   Amb 17.5
C   1   Ele 19.8
C   2   Amb 13.5
C   2   Ele 12.1
", header=TRUE)
```

Now we have two variables that describe which measurements 'belong together' (experimental plot, and treatment). We can no longer use `spread` (which uses just one of those variables), but `dcast` does work nicely with multiple groupings:

```
dcast(gas, Date + Plot ~ treatment, value.var="Photo")
```

```
##   Date Plot  Amb  Ele
## 1    A    1 14.4 16.3
## 2    A    2 17.8 13.3
## 3    B    1 16.4 19.0
## 4    B    2 15.0 10.8
## 5    C    1 17.5 19.8
## 6    C    2 13.5 12.1
```

We have another possibility here, to instead place the values for the individual plots and treatments next to each other. Note we now move `Plot` to the right-hand side of the formula in `dcast`.

```
dcast(gas, Date ~ Plot + treatment, value.var="Photo")
```

```
##   Date 1_Amb 1_Ele 2_Amb 2_Ele
## 1    A  14.4  16.3  17.8  13.3
## 2    B  16.4  19.0  15.0  10.8
## 3    C  17.5  19.8  13.5  12.1
```

3.4 More complex `dplyr` examples

In this chapter, we have learned many new tools to work with data - filtering, summarizing, reshaping, and so on. One advantage of the `dplyr` package over functions in base R is that the various operations can be efficiently combined with the `%>%` operator, combining all your data converting and shaping steps into one.

This will become clear with some examples.

3.4.1 Tree growth data: filter and multiple groupings

The first example uses the `hfeifbytree` data from the `lgrdata` package. This dataset contains measurements of height and stem diameter on nearly 1200 Eucalyptus trees in Sydney, repeated 17 times. The trees grow in plots subjected to different treatments (control, fertilization, irrigation, or both).

On some of the dates, all trees were measured, while on other dates a small subsample was taken (ca. 10% of all trees). We want to make a plot of average tree height by treatment over time, but use only the dates were all trees were measured. The following code makes Fig. 3.4.

```
data(hfeifbytree)
```

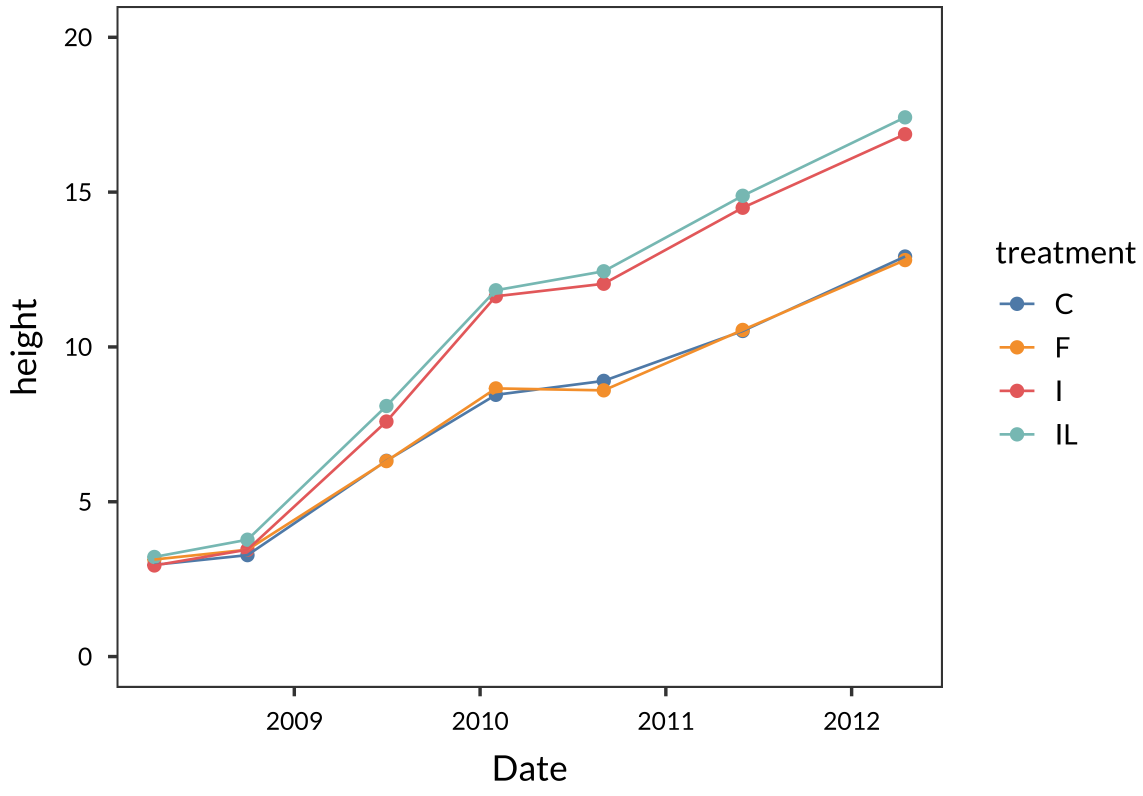


Figure 3.4: Average tree height by treatment over time, for the hfeifbytree data

```r
library(dplyr)
library(ggplot2)
library(ggthemes)

hfeif_meanh <-
  mutate(hfeifbytree, Date = as.Date(Date)) %>%  # Convert to proper Date
  group_by(Date) %>%                # Set up the grouping variable
  filter(n() > 500) %>%             # Keep groups with more than 500 observations
  group_by(Date, treat) %>%         # New grouping; to get average by Date and treatment
  summarize(height = mean(height, na.rm=TRUE)) %>% # Average height, discard NA.
  rename(treatment = treat)         # rename a variable

# It is possible to make the plot inside the data pipeline, but
# we recommend separating them!
ggplot(hfeif_meanh, aes(x = Date, y = height, col = treatment)) +
  geom_point(size = 2) +
  geom_line() +
  scale_colour_tableau() +
  ylim(c(0,20))
```

3.4.2 Crude oil production: find top exporters

In the second example we will use the `oil` data, a dataset with annual crude oil production for the top 8 oil-producing countries since 1971. We want to find the top three countries for the period 1980-1985, sort them, and replace the country abbreviations with country names.

```r
data(oil)

# First we make a dataframe with the abbreviations.
# Very handy here is tribble from tibble; making it easier to write
# small dataframes directly into your script.
library(tibble)
abb_key <- tribble(~country, ~country_full,
                      "MEX",    "Mexico",
                      "USA",    "USA",
                      "CHN",    "China",
                      "IRN",    "Iran",
                      "SAU",    "Saudi-Arabia",
                      "IRQ",    "Iraq",
                      "KWT",    "Kuwait",
                      "VEN",    "Venezuela")

# To avoid a warning, we first convert country to character
# (not strictly necessary)
oil_top <-
  mutate(oil,
         country = as.character(country),
         production_M = production / 1000) %>%  # Convert to million tonnes.
  full_join(abb_key, by="country") %>%          # Add full country name
  filter(year %in% 1980:1985) %>%               # Subset for years between 1980-1985
  group_by(country_full) %>%
  summarize(production_M = mean(production_M)) %>%
  arrange(desc(production_M)) %>%               # Sort by production; in descending order
  as.data.frame %>%                             # To be consistent, output a dataframe
  head(., 3)                                    # Only show the top 32

# Finally make a table that looks nice in an rmarkdown document
library(pander)
oil_top %>% pander(.,
                   caption = "Top three oil producing countries, 1980-1985.",
                   col.names = c("Country", "Annual production (MTOE)"))
```

Table 3.2: Top three oil producing countries, 1980-1985.

Country	Annual production (MTOE)
USA	443.5
Saudi-Arabia	335.1
Mexico	138.8

Try this yourself Run the example above, and modify the code so that we find the *maximum* oil production for each country, and the table should show the 2 countries with the lowest maximum.

3.4.3 Weight loss data: make an irregular timeseries regular

In this example we will use the `weightloss` data, again from the `lgrdata` package. A person recorded his weight on irregular intervals (1-3 days), to check if his diet was having the desired effect. Because the dataset is irregular, it is not easy to calculate daily weight loss for the entire dataset. We can use a combination of `dplyr`, `zoo` and `padr` to clean up this dataset.

The `pad` function is quite powerful: it takes a dataframe, figures the time-indexing variable (in our case, Date), and stretches the dataset to include all Dates along the dataset, filling NA where no data were available.

The `zoo` package contains many useful functions for timeseries data. Here we use `na.approx` to linearly interpolate missing values. We also make a plot of the weight loss data, clarifying which data were interpolated. The following code make Fig. 3.5.

```
library(lubridate)
library(dplyr)
library(zoo)
library(padr)

data(weightloss)

weightloss2 <- mutate(weightloss,
                      Date = dmy(Date),                # proper Date class
                      Weight = Weight * 0.4536) %>%    # convert to kg
               pad(interval ="day") %>%                # timeseries is now regular
               mutate(measured = !is.na(Weight),   # FALSE when the value was interpolated
                      Weight = na.approx(Weight))      # linear interpolation (zoo)

ggplot(weightloss2, aes(x = Date, y = Weight)) +
  geom_point(colour = ifelse(weightloss2$measured, "darkgrey", "red2"))
```

3.5 Exercises

3.5.1 Summarizing the cereal data

1. Read the cereal data, and produce quick summaries using `str`, `summary`, `contents` and `describe` (recall that the last two are in the `Hmisc` package). Interpret the results.
2. Find the average sodium, fiber and carbohydrate contents by `Manufacturer`. Use either `summaryBy` or `dplyr`.
3. Add a new variable 'SodiumClass', which is 'high' when sodium > 150 and 'low' otherwise. Make sure the new variable is a factor. Look at the examples in Section 2.7.1 to recall how to do this.

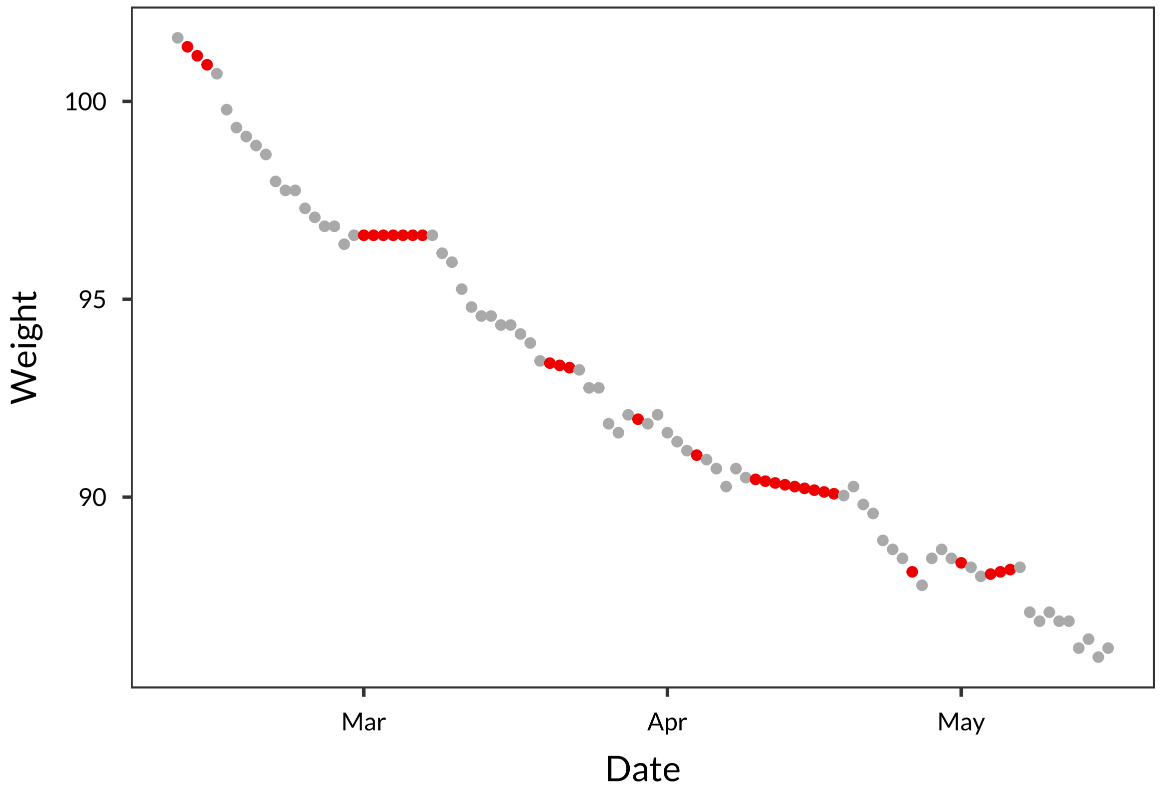


Figure 3.5: The weightloss data, with linearly interpolated missing values (red).

Now, find the average, minimum and maximum sugar content for 'low' and 'high' sodium. *Hint:* make sure to use `na.rm=TRUE`, because the dataset contains missing values.

4. Find the maximum sugar content by Manufacturer and sodiumClass, using `tapply`. Inspect the result and notice there are missing values. Try to use `na.rm=TRUE` as an additional argument to `tapply`, only to find out that the values are still missing. Finally, use `xtabs` (see Section 3.1.2, p. 3.1.2) to count the number of observations by the two factors to find out if we have missing values in the `tapply` result.
5. Repeat the previous question with `summaryBy` or `dplyr`. Compare the results.
6. Count the number of observations by Manufacturer and whether the cereal is 'hot' or 'cold', using `xtabs` (see Section 3.1.2).

}

3.5.2 Words and the weather

1. Using the 'Age and memory' dataset (`memory` from the `lgrdata` package), find the mean and maximum number of words recalled by 'Older' and 'Younger' age classes.
2. The `hfemet2008` dataset contains meteorological measurements at a station near Sydney, Australia. Find the mean air temperature by month. To do this, first add the month variable as shown in Section 2.7.5.3.

3.5.3 Merging data

1. Load the `pupae` dataset. The data contain measurements of larva ('pupae') weight and 'frass' (excrement) production while allowed to feed on leaves, grown under different concentrations of carbon dioxide (CO2). Also read this short dataset, which gives a label 'roomnumber' for each CO_2 treatment.

```
|CO2_treatment |Roomnumber  |
|-------------:|-----------:|
|280           |1           |
|400           |2           |
```

To read this dataset, consider the `data.frame` function described in Section 2.4.1.

1. Merge the short dataset onto the pupae data. Check the result.

3.5.4 Merging multiple datasets

Read Section 3.2.2, and learn how to merge more than two datasets together.

First, run the following code to construct three dataframes that we will attempt to merge together.

```
dataset1 <- data.frame(unit=letters[1:9], treatment=rep(LETTERS[1:3],each=3),
                       Damage=runif(9,50,100))
unitweight <- data.frame(unit=letters[c(1,2,4,6,8,9)], Weight = rnorm(6,100,0.3))
treatlocation <- data.frame(treatment=LETTERS[1:3], Glasshouse=c("G1","G2","G3"))
```

1. Merge the three datasets together, to end up with one dataframe that has the columns 'unit', 'treatment', 'Glasshouse', 'Damage' and 'Weight'. Some units do not have measurements of `Weight`. Merge the datasets in two ways to either include or exclude the units without `Weight` measurements. To do this, either use `merge` or `dplyr::join` twice - you cannot merge more than two datasets in one step.

}

3.5.5 Ordered boxplot

1. First read the `cereals` data and learn how to make a boxplot, using R base graphics:

Notice the *ordering* of the boxes from left to right, and compare it to `levels` of the factor variable `Manufacturer`.

2. Now, redraw the plot with Manufacturer in order of increasing mean sodium content (use `reorder`, see Section 3.1.4).
3. Inspect the help page (`?boxplot`), and change the boxplots so that the width varies with the number of observations per manufacturer (*Hint:* find the `varwidth` argument).

Chapter 4

Functions, lists and loops

4.1 Introduction

This chapter demonstrates some building blocks of programming with R. The objective is to learn skills that help you with batch analyses - repeating similar tasks for many subsets of data, and to write functions to help you organize your code.

For batch analyses, we first have to know how to write our own functions (Section 4.2), so that we can apply the custom function to any new subset of data. This approach results in far less, and much more readable code than copy-pasting similar code for different datasets.

We also delve into "lists" in R (Section 4.3), a versatile data object that you have worked with already - even if you didn't know it. Understanding lists well in R is the key to more complex analyses, and more readable workflows.

We also take a brief look at 'for loops' (Section 4.4), a basic programming utility that we rarely need in R, since almost all functions are vectorized. Sometimes it is however more convenient to use loops, or makes the code just a little more easy to work with.

Finally we introduce many more advanced concepts for writing functions (Section 4.5 and 4.6), often to make functions more versatile and less error-prone.

Packages used in this chapter

We use no new packages in this chapter except the `wrapr` package. All other functionality is included in base R, or commonly used packages `dplyr`, `ggplot2`, `lubridate`, and `lgrdata` for the example data.

4.2 Writing functions

We have already used many built-in functions throughout this tutorial, but you can become very efficient at complex data tasks when you write your own simple functions. Writing your own functions can help with tasks that are carried out many times, which would otherwise result in a lot of code.

For example, suppose you frequently convert units from pounds to kilograms. It would be useful to have a function that does this, so you don't have to type the conversion factor every time. This is also good practice, as it reduces the probability of making typos.

```
# This function takes a 'weight' argument and multiplies it with some number
# to return kilograms.
poundsToKg <- function(weight){
  weight * 0.453592
}
```

We can use this `function` just like any other in R, for example, let's convert 'weight' to kilograms in the weightloss data.

```
# Read data
library(lgrdata)
data(weightloss)

# Convert weight to kg.
weightloss$Weight <- poundsToKg(weightloss$Weight)
```

Let's write a function for the standard error of the mean, a function that is not built-in in R.

```
# Compute the standard error of the mean for a vector
SEmean <- function(x){
  se <- sd(x) / sqrt(length(x))
  return(se)
}
```

Here, the `function` SEmean takes one 'argument' called `x` (i.e., input), which is a numeric vector. The standard error for the mean is calculated in the first line, and stored in an object called `se`, which is then returned as output. We can now use the function on a numeric vector like this:

```
# A numeric vector
unifvec <- runif(10, 1,2)

# The sample mean
mean(unifvec)
```

```
## [1] 1.541051
```

```
# Standard error for the mean
SEmean(unifvec)
```

```
## [1] 0.0996315
```

Info In general, avoid using names for your functions that are already in use by base R, or commonly used add-on packages. If two or more functions with the same name have been defined, R will first use the one defined by the user (and available in the 'global environmment'), see Section 8.4.

4.2.1 Returning results

What if a function should return not just one result, as in the examples above, but many results?

For example, this function computes the standard deviation and standard error of a vector, and returns both stored in a vector. Note that we also use the `SEmean` function, which we defined above.

```r
# An function that computes the SE and SD of a vector
seandsd <- function(x){

  seresult <- SEmean(x)
  sdresult <- sd(x)

  # Store results in a vector with names
  vec <- c(SE = seresult, SD = sdresult)

return(vec)
}

# Test it:
x <- rnorm(100, mean=20, sd=4)
seandsd(x)
```

```
##       SE       SD
## 0.412936 4.129360
```

4.2.2 Functions without arguments

Sometimes, a function takes no arguments (input) at all.

```r
sayhello <- function()message("Hello!")

sayhello()
```

```
## Hello!
```

We now know enough about writing functions to start using them in many real-world applications, but return to writing more advanced functions in Section 4.5.

First, we take a close look at the most versatile data structure in R : the list.

4.3 Working with lists

Sofar, we have worked a lot with vectors, with are basically strings of numbers or bits of text. In a vector, each element has to be of the same data type. Lists are a more general and powerful type of vector, where each element of the `list` can be anything at all. This way, lists are a very flexible type of object to store a lot of information that may be in different formats.

Lists can be somewhat daunting for the beginning `R` user, which is why most introductory texts and tutorials skip them altogether. However, with some practice, lists can be mastered from the start. Mastering a few basic skills with lists can really help increase your efficiency in dealing with more complex data analysis tasks.

To make a list from scratch, you simply use the `list` function. Here is a list that contains a numeric vector, a character vector, and a dataframe:

```r
mylist <- list(a=1:10, txt=c("hello","world"), dfr=data.frame(x=c(2,3,4),y=c(5,6,7)))
```

4.3.1 Indexing lists

To extract an element from this list, you may do this by its name ('a','txt' or 'dfr' in this case), or by the element number (1,2,3). For lists, we use a double square bracket for indexing. Consider these examples,

```
# Extract the dataframe:
mylist[["dfr"]]
```

```
##   x y
## 1 2 5
## 2 3 6
## 3 4 7
```

```
# Is the same as:
mylist$dfr
```

```
##   x y
## 1 2 5
## 2 3 6
## 3 4 7
```

```
# Extract the first element:
mylist[[1]]
```

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

Note that in these examples, the contents of the elements of the list are returned (for 'dfr', a dataframe), but the result itself is not a list anymore. If we select multiple elements, the result should still be a list. To do this, use the single square bracket.

Look at these examples:

```
# Extract the 'a' vector, result is a vector:
mylist[['a']]
```

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

```
# Extract the 'a' vector, result is a list:
mylist['a']
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# Extract multiple elements (result is still a list):
mylist[2:3]
```

```
## $txt
## [1] "hello" "world"
##
## $dfr
##   x y
## 1 2 5
## 2 3 6
## 3 4 7
```

4.3.2 Converting lists to dataframes or vectors

Although lists are the most flexible way to store data and other objects in larger, more complex, analyses, ultimately you would prefer to output as a dataframe or vector.

Let's look at some examples using `do.call(rbind,...)` and `unlist`.

```
# A list of dataframes:
dfrlis <- list(data1=data.frame(a=1:3,b=2:4), data2=data.frame(a=9:11,b=15:17))
dfrlis
```

```
## $data1
##   a b
## 1 1 2
## 2 2 3
## 3 3 4
##
## $data2
##    a  b
## 1  9 15
## 2 10 16
## 3 11 17
```

```
# Since both dataframes in the list have the same number of columns and names,
# we can 'successively row-bind' the list like this:
do.call(rbind, dfrlis)
```

```
##          a  b
## data1.1  1  2
## data1.2  2  3
## data1.3  3  4
## data2.1  9 15
## data2.2 10 16
## data2.3 11 17
```

```
# A list of vectors:
veclis <- list(a=1:3, b=2:4, f=9:11)

# In this case, we can use the 'unlist' function, which will
# successively combine the three vectors into one:
unlist(veclis)
```

```
## a1 a2 a3 b1 b2 b3 f1 f2 f3
##  1  2  3  2  3  4  9 10 11
```

In real-world applications, some trial-and-error will be necessary to convert lists to more pretty formats.

4.3.3 Combining lists

Combining two lists can be achieved using `c()`, like this:

```
veclis <- list(a=1:3, b=2:4, f=9:11)
qlis <- list(q=17:15)
c(veclis,qlis)
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] 2 3 4
##
## $f
## [1]  9 10 11
##
## $q
## [1] 17 16 15
```

```
# But be careful when you like to quickly add a vector
# the 'veclis'. You must specify list() like this
veclis <- c(veclis, list(r=3:1))
```

4.3.4 Extracting output from built-in functions

One reason to gain a better understanding of lists is that many built-in functions return not just single numbers, but a diverse collection of outputs, organized in lists. Think of the linear model function (`lm`), it returns a lot of things at the same time (not just the p-value).

Let's take a closer look at the `lm` output to see if we can extract the adjusted R^2.

```
# Read data
data(allometry)

# Fit a linear model
lmfit <- lm(height ~ diameter, data=allometry)

# And save the summary statement of the model:
lmfit_summary <- summary(lmfit)

# We already know that simply typing 'summary(lmfit)' will give
# lots of text output. How to extract numbers from there?
# Let's look at the structure of lmfit:
str(lmfit_summary)
```

```
## List of 11
##  $ call         : language lm(formula = height ~ diameter, data = allometry)
##  $ terms        :Classes 'terms', 'formula'  language height ~ diameter
##   .. ..- attr(*, "variables")= language list(height, diameter)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "height" "diameter"
##   .. .. .. ..$ : chr "diameter"
##   .. ..- attr(*, "term.labels")= chr "diameter"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(height, diameter)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "height" "diameter"
##  $ residuals    : Named num [1:63] -8.84 1.8 0.743 2.499 4.37 ...
##   ..- attr(*, "names")= chr [1:63] "1" "2" "3" "4" ...
##  $ coefficients : num [1:2, 1:4] 7.5967 0.5179 1.4731 0.0365 5.157 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "(Intercept)" "diameter"
##   .. ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
##  $ aliased      : Named logi [1:2] FALSE FALSE
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "diameter"
##  $ sigma        : num 5.55
##  $ df           : int [1:3] 2 61 2
##  $ r.squared    : num 0.768
##  $ adj.r.squared: num 0.764
##  $ fstatistic   : Named num [1:3] 202 1 61
##   ..- attr(*, "names")= chr [1:3] "value" "numdf" "dendf"
##  $ cov.unscaled : num [1:2, 1:2] 7.05e-02 -1.54e-03 -1.54e-03 4.32e-05
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "(Intercept)" "diameter"
##   .. ..$ : chr [1:2] "(Intercept)" "diameter"
##  - attr(*, "class")= chr "summary.lm"
```

```
# The output of lm is a list, so we can look at the names of # that list as well:
names(lmfit_summary)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

So, now we can extract results from the summary of the fitted regression. Also look at the help file `?summary.lm`, in the section 'Values' for a description of the fields contained here.

To extract the adjusted R^2, for example:

```
lmfit_summary[["adj.r.squared"]]
```

```
## [1] 0.7639735
```

```
# Is the same as:
lmfit_summary$adj.r.squared
```

```
## [1] 0.7639735
```

This sort of analysis will be very useful when we do many regressions, and want to summarize the results in a table.

Try this yourself Run the code in the above examples, and practice extracting some other elements from the linear regression. Compare the output to the summary of the `lm` fit (that is, compare it to what `summary(lmfit)` shows on screen).

4.3.5 Creating lists from dataframes

For more advanced analyses, it is often necessary to repeat a particular analysis many times, for example for sections of a dataframe.

Using the `allom` data for example, we might want to split the dataframe into three dataframes (one for each species), and repeat some analysis for each of the species. One option is to make three subsets (using `subset`), and repeating the analysis for each of them. But what if we have hundreds of species?

A more efficient approach is to `split` the dataframe into a list, so that the first element of the list is the dataframe for species 1, the 2nd element species 2, and so on. In case of the allom dataset, the resulting list will have three components.

Let's look at an example on how to construct a list of dataframes from the allom dataset, one per species:

```r
# Read allom data and make sure 'species' is a factor:
data(allometry)
is.factor(allometry$species)
```

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

```r
# The levels of the factor variable 'species'
levels(allometry$species)
```

```
## [1] "PIMO" "PIPO" "PSME"
```

```r
# Now use 'split' to construct a list:
allomsp <- split(allometry, allometry$species)

# The length of the list should be 3, with the names equal to the
# original factor levels:
length(allomsp)
```

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

```r
names(allomsp)
```

```
## [1] "PIMO" "PIPO" "PSME"
```

Try this yourself Run the code in the above example, and confirm that `allomsp[[2]]` is identical to taking a subset of `allom` of the second species in the dataset (where 'second' refers to the second level of the factor variable `species`, which you can find out with `levels`).

Let's look at an example using the `hydro` data. The data contains water levels of a hydrodam in Tasmania, from 2005 to 2011.

```r
# Read hydro data, and convert Date to a proper date class.
data(hydro)

library(lubridate)
library(dplyr)

hydro <- mutate(hydro,
                Date = dmy(Date),
                year = year(Date))
```

```
# Look at the Date range:
range(hydro$Date)
```

```
## [1] "2005-08-08" "2011-08-08"
```

```
# Let's get rid of the first and last years (2005 and 2011) since they are incomplete
hydro <- filter(hydro, !year %in% c(2005,2011))

# Now split the dataframe by year. This results in a list, where every
# element contains the data for one year:
hydrosp <- split(hydro, hydro$year)

# Properties of this list:
length(hydrosp)
```

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

```
names(hydrosp)
```

```
## [1] "2006" "2007" "2008" "2009" "2010"
```

To extract one element of the two lists that we created (`allomsp` and `hydrosp`), recall the section on indexing lists.

4.3.6 Applying functions to lists

We will introduce two basic tools that we use to apply functions to each element of a list: `sapply` and `lapply`. The `lapply` function always returns a list, whereas `sapply` will attempt to simplify the result. When the function returns a single value, or a vector, `sapply` can often be used. In practice, try both and see what happens!

4.3.6.1 Using `sapply`

First let's look at some simple examples:

```
# Let's make a simple list with only numeric vectors (of varying length)
numlis <- list(x=1000, y=c(2.1,0.1,-19), z=c(100,200,100,100))

# For the numeric list, let's get the mean for every element, and count
# the length of the three vectors.
# Here, sapply takes a list and a function as its two arguments,
# and applies that function to each element of the list.
sapply(numlis, mean)
```

```
##      x      y      z
## 1000.0   -5.6  125.0
```

```
sapply(numlis, length)
```

```
## x y z
## 1 3 4
```

You can of course also define your own functions, and use them here. Let's look at another simple example using the `numlis` object defined above.

For example,

```
# Let's find out if any diameters are duplicated in the allom dataset.
# A function that does this would be the combination of 'any' and 'duplicated',
anydup <- function(vec)any(duplicated(vec))
# This function returns TRUE or FALSE

# Apply this function to numlis (see above):
sapply(numlis, anydup)
```

```
##     x     y     z
## FALSE FALSE  TRUE
```

```
# You can also define the function on the fly like this:
sapply(numlis, function(x)any(duplicated(x)))
```

```
##     x     y     z
## FALSE FALSE  TRUE
```

Now, you can use any function in `sapply` as long as it returns a single number based on the element of the list that you used it on. Consider this example with `strsplit`.

```
# Recall that the 'strsplit' (string split) function usually returns a list of values.
# Consider the following example, where the data provider has included the units in
# the measurements of fish lengths. How do we extract the number bits?
fishlength <- c("120 mm", "240 mm", "159 mm", "201 mm")

# Here is one solution, using strsplit
strsplit(fishlength," ")
```

```
## [[1]]
## [1] "120" "mm"
##
## [[2]]
## [1] "240" "mm"
##
## [[3]]
## [1] "159" "mm"
##
## [[4]]
## [1] "201" "mm"
```

```
# We see that strsplit returns a list, let's use sapply to extract only
# the first element (the number)
splitlen <- strsplit(fishlength," ")
sapply(splitlen, function(x)x[1])
```

```
## [1] "120" "240" "159" "201"
```

```
# Now all you need to do is use 'as.numeric' to convert these bits of text to numbers.
```

The main purpose of splitting dataframes into lists, as we have done above, is so that we can save time with analyses that have to be repeated many times. In the following examples, you must have already produced the objects `hydrosp` and `allomsp` (from examples in the previous section).Both those objects are *lists of dataframes*, that is, each element of the list is a dataframe in itself. Let's look at a few examples with `sapply` first.

```r
# How many observations per species in the allom dataset?
sapply(allomsp, nrow)
```

```
## PIMO PIPO PSME
##   19   22   22
```

```r
# Here, we applied the 'nrow' function to each separate dataframe.
# (note that there are easier ways to find the number of observations per species!,
# this is just illustrating sapply.)

# Things get more interesting when you define your own functions on the fly:
sapply(allomsp, function(x)range(x$diameter))
```

```
##       PIMO  PIPO  PSME
## [1,]  6.48  4.83  5.33
## [2,] 73.66 70.61 69.85
```

```r
# Here, we define a function that takes 'x' as an argument:
# sapply will apply this function to each element of the list,
# one at a time. In this case, we get a matrix with ranges of the diameter per species.

# How about the correlation of two variables, separate by species:
sapply(allomsp, function(x)cor(x$diameter, x$height))
```

```
##      PIMO      PIPO      PSME
## 0.9140428 0.8594689 0.8782781
```

```r
# For hydro, find the number of days that storage was below 235, for each year.
sapply(hydrosp, function(x)sum(x$storage < 235))
```

```
## 2006 2007 2008 2009 2010
##    0   18    6    0    0
```

4.3.6.2 Using `lapply`

The `lapply` function is much like `sapply`, except it always returns a list.

For example,

```r
# Get a summary of the hydro dataset by year:
lapply(hydrosp, summary)
```

```
## $`2006`
##       Date                storage           year
##  Min.   :2006-01-02   Min.   :411.0   Min.   :2006
##  1st Qu.:2006-04-01   1st Qu.:449.5   1st Qu.:2006
##  Median :2006-06-29   Median :493.0   Median :2006
##  Mean   :2006-06-29   Mean   :514.3   Mean   :2006
##  3rd Qu.:2006-09-26   3rd Qu.:553.8   3rd Qu.:2006
##  Max.   :2006-12-25   Max.   :744.0   Max.   :2006
##
## $`2007`
##       Date               storage           year
##  Min.   :2007-01-01   Min.   :137.0   Min.   :2007
##  1st Qu.:2007-04-02   1st Qu.:223.0   1st Qu.:2007
##  Median :2007-07-02   Median :319.0   Median :2007
##  Mean   :2007-07-02   Mean   :363.2   Mean   :2007
##  3rd Qu.:2007-10-01   3rd Qu.:477.0   3rd Qu.:2007
##  Max.   :2007-12-31   Max.   :683.0   Max.   :2007
##
## $`2008`
##       Date               storage           year
##  Min.   :2008-01-07   Min.   :202.0   Min.   :2008
##  1st Qu.:2008-04-05   1st Qu.:267.8   1st Qu.:2008
##  Median :2008-07-03   Median :345.0   Median :2008
##  Mean   :2008-07-03   Mean   :389.5   Mean   :2008
##  3rd Qu.:2008-09-30   3rd Qu.:551.8   3rd Qu.:2008
##  Max.   :2008-12-29   Max.   :637.0   Max.   :2008
##
## $`2009`
##       Date               storage           year
##  Min.   :2009-01-05   Min.   :398.0   Min.   :2009
##  1st Qu.:2009-04-04   1st Qu.:467.0   1st Qu.:2009
##  Median :2009-07-02   Median :525.5   Median :2009
##  Mean   :2009-07-02   Mean   :567.6   Mean   :2009
##  3rd Qu.:2009-09-29   3rd Qu.:677.8   3rd Qu.:2009
##  Max.   :2009-12-28   Max.   :791.0   Max.   :2009
##
## $`2010`
##       Date               storage           year
##  Min.   :2010-01-04   Min.   :251.0   Min.   :2010
##  1st Qu.:2010-04-03   1st Qu.:316.0   1st Qu.:2010
##  Median :2010-07-01   Median :368.5   Median :2010
##  Mean   :2010-07-01   Mean   :428.5   Mean   :2010
##  3rd Qu.:2010-09-28   3rd Qu.:577.5   3rd Qu.:2010
##  Max.   :2010-12-27   Max.   :649.0   Max.   :2010
```

Suppose you have multiple similar datasets in your working directory, and you want to read all of these into one list, use `lapply` like this (run this example yourself and inspect the results).

```
# Names of your datasets:
filenames <- c("pupae.csv","pupae.csv","pupae.csv")
# (This toy example will read the same file three times).

# Read all files into one list,
alldata <- lapply(filenames, read.csv)

# Then, if you are sure the datasets have the same number of columns and names,
# use do.call to collapse the list:
dfrall <- do.call(rbind, alldata)
```

Try this yourself Recall the use of `dir` to list files, and even to find files that match a specific pattern (see Section 1.8. Read all CSV files in your working directory (or elsewhere) into a single list, and count the number of rows for each dataframe.

Finally, we can use `lapply` to do all sorts of complex analyses that return any kind of object. The use of `lapply` with lists ensures that we can organize even large amounts of data in this way.

Let's do a simple linear regression of log(leafarea) on log(diameter) for the Allometry dataset, by species:

```
# Run the linear regression on each element of the list, store in a new object:
lmresults <- lapply(allomsp, function(x)lm(log10(leafarea) ~ log10(diameter), data=x))

# Now, lmresults is itself a list (where each element is an object as returned by lm)
# We can extract the coefficients like this:
sapply(lmresults, coef)
```

```
##                       PIMO       PIPO       PSME
## (Intercept)     -0.3570268 -0.7368336 -0.3135996
## log10(diameter)  1.5408859  1.6427773  1.4841361
```

```
# This shows the intercept and slope by species.
# Also look at (results not shown):
# lapply(lmresults, summary)

# Get R2 for each model. First write a function that extracts it.
getR2 <- function(x)summary(x)$adj.r.squared
sapply(lmresults, getR2)
```

```
##      PIMO      PIPO      PSME
## 0.8738252 0.8441844 0.6983126
```

Try this yourself Try to fully understand the difference between `sapply` and `lapply` by using `lapply` in some of the examples where we used `sapply` (and vice versa).

4.4 Loops

Loops can be useful when we need to repeat certain analyses many times, and it is difficult to achieve this with `lapply` or `sapply`. To understand how a `for` loop works, look at this example:

```
for(i in 1:5){
  message(i)
}
```

```
## 1

## 2

## 3

## 4

## 5
```

Here, the bit of code between {} is executed five times, and the object `i` has the values 1,2,3,4 and 5, in that order. Instead of just printing `i` as we have done above, we can also index a vector with this object:

```
# make a vector
myvec <- round(runif(5),1)

for(i in 1:length(myvec)){
  message("Element ", i, " of the vector is: ", myvec[i])
}
```

```
## Element 1 of the vector is: 0.3

## Element 2 of the vector is: 0.6

## Element 3 of the vector is: 1

## Element 4 of the vector is: 0.1

## Element 5 of the vector is: 0.1
```

Note that this is only a toy example: the same result can be achieved by simply typing `myvec`.

One common application of `for` loops in R arises when producing multiple plots in a `pdf`. Consider this template (make your own working example based on any dataset).

We assume here you have a dataframe called 'dataset' with a *factor* 'species', for which you want to create separate plots of Y vs. X.

```
pdf("somefilename.pdf", onefile=TRUE)
for(lev in levels(dataset$species)){

  with(subset(dataset, species==lev),
       plot(X,Y,
            main=as.character(lev)))
}
dev.off()
```

Here, we have to call `dev.off()` to close the pdf.

4.4.1 Better indexing

In the first example with `for` loops, we used `1:length(myvec)` to make *indices* for the for loop. This is fine when you know for sure `myvec` exists and isn't `NULL` or `NA` (in which case the indices will not be what you expect - try it out!).

A safer alternative in `for` loops is to use `seq_along` like in the following example. Here we substitute different parts of a string using a `for` loop.

```
mystring <- "A fish called Wanda"

replacements <- c("fish","called","Wanda")

for(i in seq_along(replacements)){
  print(gsub(replacements[i], "dog", mystring))
}
```

```
## [1] "A dog called Wanda"
## [1] "A fish dog Wanda"
## [1] "A fish called dog"
```

Now, when you place code similar to the above in a more generic function, you don't have to worry when `replacements` is empty for some reason, the `for` loop will simply be skipped.

4.5 Functions : advanced concepts

4.5.1 Why should I write more functions?

Organizing your complex work flow into many custom functions, instead of long scripts with just the raw code, has some key advantanges:

- Using functions makes it clear what the **inputs** are, and what are the **outputs**. For example, if you run `result <- myfunction(mydata)`, you know that `mydata` was used to create `result`. Instead, if you place all your code *plain* in a script and use `source("myscript.R")`, what objects are created? Are there any output? If so, where? Using functions this way allows a logical flow of your scripts.
- Using functions is much **safer** because objects created in the body of the function when it is executed only *live* for as long as the function is running. Once the function *returns* a result, all objects inside the function cease to exist. This means the objects do not use memory, and cannot cause conflicts with other objects in the session.
- Using functions allows you to **avoid any duplicated code**. Duplicated code is terrible because when you want to modify your code, you have to repeat the modification (and remember where!). As a rule of thumb, if you do something 3 times, you should write a function. This function can also take some arguments, allowing it to do slightly different operations from different calls.
- For different projects, you often find yourself using the same bits of code, for example to read certain data, format a certain table, or make some plot. It quickly makes sense to collect this sort of code into a collection of your own functions. When these functions become **generic** enough (applicable in many projects), you should consider bundling them in an R package.

4.5.2 Wrapper functions

We often need to write simple functions that adjust one or two arguments to other functions. For example, suppose we often make plots with filled circles, with our favorite color ("dimgrey") :

```r
library(lgrdata)
data(howell)
plot(age, height, data=howell, pch=19, col="dimgrey")
```

We could of course *always* specify these arguments, or we can write a function that sets those defaults. It would look like the following, except this function is incomplete, since we have *hardcoded* the other plotting arguments (the dataset, and the x and y variables):

```r
# This function is not how we want it yet!
plot_filled_grey <- function(){
  plot(age, height, data=howell, pch=19, col="dimgrey")
}
```

We would like to be able to call the function via `plot_filled_grey(age, height, data=howell)`, in other words all arguments to our *wrapper function* should be passed to the underlying function. We have a very handy utility in R to do this, the three dots (`...`) :

```r
plot_filled_grey <- function(...){
  plot(..., pch=19, col="dimgrey")
}
```

The function now works as intended. It can be further improved if we realize that the plotting color cannot be changed - it is always "dimgrey". We want the *default* value to be "dimgrey", but with an option to change it. This can be achieved like so,

```r
plot_filled_grey <- function(..., col="dimgrey"){
  plot(..., pch=19, col=col)
}
```

Here, `col=col` sets the color in the call to `plot` with the default value specified in the wrapper function ("dimgrey"), but the user can also change it as usual.

Try this yourself Take the `plot_filled_grey` function above, test it on some data, and modify it so that the plotting symbol can also be changed, but has a default value of 19.

4.5.3 Wrapper functions to `ggplot2` or `dplyr`

In the previous section we saw how to write wrapper functions, functions that change a few arguments to some other function. These sort of functions are very helpful because we can save a lot of space by reusing a certain template. Suppose we want to make a plot with `ggplot2`, a simple scatter plot. We can achieve this via (result not shown),

```r
data(howell)
library(ggplot2)

ggplot(howell, aes(x = weight, y = height)) +
  geom_point(size = 0.8, col = "dimgrey") +
  stat_smooth(method = "loess", span = 0.7, col="black")
```

We already used quite a bit of code for this simple plot, but imagine you have set various other options, formatting changes, axis limits and so on - you end up with a lot of code for one plot. If we want to reuse the code for another plot, for two other variables from the same dataframe, copy-pasting the code and modifying leads to even more code.

Writing wrapper functions for `ggplot2` (or `dplyr`, see below, or many other cases) is more difficult because the arguments in the `ggplot2` code we want to change (height and weight) *are not quoted* - they are variables inside a dataframe (`howell` in our case).

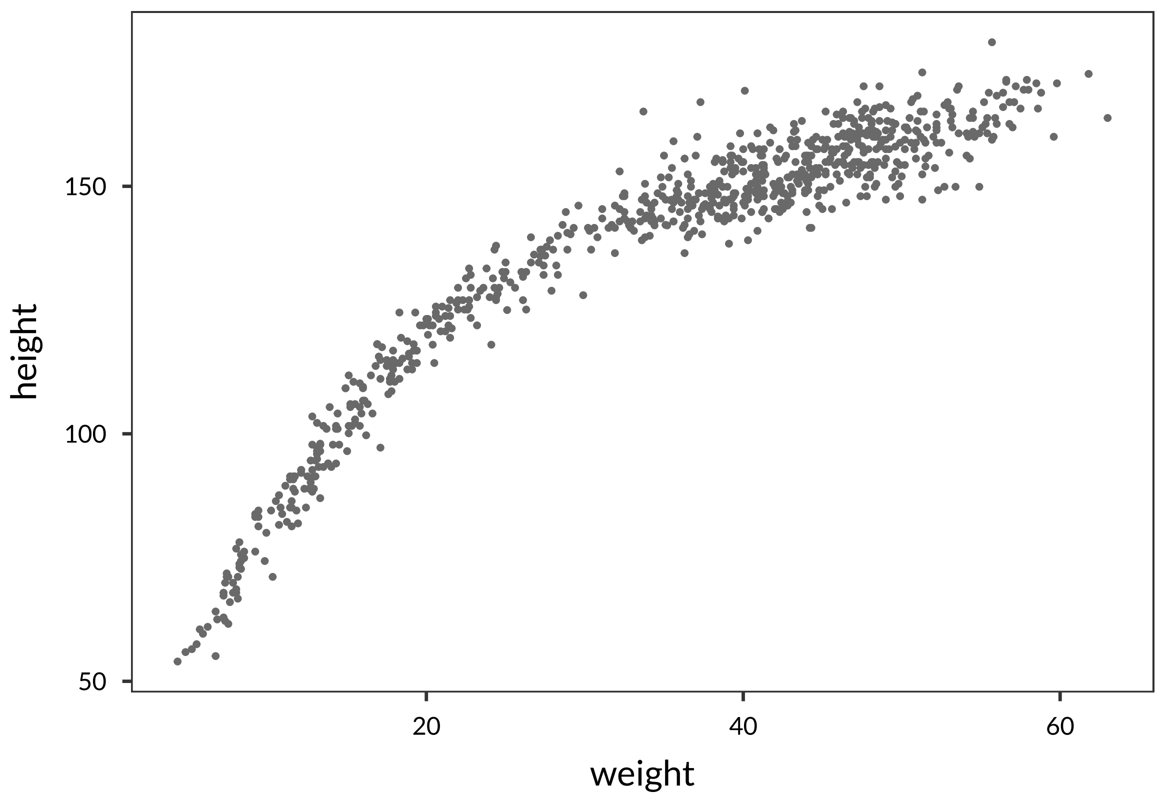
Suppose we have defined somewhere (perhaps as arguments in a function) that the following two variables should be used for the plot:

```r
xvar <- "weight"
yvar <- "height"
```

If we use the same code as in the previous example, it does not work because the variable names need to be *not quoted*. The following (rather obscure) syntax converts the quoted variable name, to a variable to be found within the provided data.

```r
library(lgrdata)
data(howell)
```

```r
ggplot(howell, aes(x = !!sym(xvar), y = !!sym(yvar))) +
geom_point(size = 0.8, col = "dimgrey")
```



We can now write a wrapper function for a `ggplot2` plot. The same approach works for many other similar wrapper functions, for example around `dplyr` operations.

```r
# Make a scatter plot with the howell data.
plot_scatter_howell <- function(xcol, ycol){
    ggplot(howell, aes(x = !!sym(xcol), y = !!sym(ycol))) +
    geom_point(size = 0.8, col = "dimgrey")
}

# The function can be used with quoted names:
plot_scatter_howell("height", "weight")
```

To also pass the dataset as an argument (making the function more general), we do not have to use any special syntax, but can immediately set it as an argument. The special syntax only applies when we are looking for variables in a dataframe.

```r
# Our function now also takes a dataset as an argument
plot_scatter_howell2 <- function(xcol, ycol, dataset){

  ggplot(dataset, aes(x = !!sym(xvar), y = !!sym(yvar))) +
  geom_point(size = 0.8, col = "dimgrey")

}
```

```r
# We can use it as,
plot_scatter_howell2("height", "weight", dataset = howell)
```

4.5.4 Functions that take vectors as input

Many functions in R are *vectorized*, which means you can send a *vector* as an argument and the function will be automatically evaluated for each element of the vector. For example:

```r
# Count nr of characters in one string:
nchar("A sentence")
```

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

```r
# the first argument to nchar() is vectorized:
nchar(c("A sentence", "is", "useful"))
```

```
## [1] 10  2  6
```

Not all functions are vectorized, and when you write your own more complex functions you will often run into situations where you would like an argument to be vectorized but run into difficulty.

Take the following example:

```r
# A complex calculation.
# If x is less than "change_val" (default 5),
# return the product, otherwise the sum.
multiplex <- function(x, change_val = 5){

  if(x < change_val){
    prod(1:x)
  } else {
    sum(1:x)
  }

}

# This should give 4*3*2*1
multiplex(4)
```

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

A function like this is really only useful if it is vectorized over `x`, but it does not work as intended:

```r
# The function is not vectorized over x:
multiplex(x = c(4,8))
```

```
## Warning in if (x < change_val) {: the condition has length > 1 and only the
## first element will be used
```

```
## Warning in 1:x: numerical expression has 2 elements: only the first used
```

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

```r
# ... or change_val
multiplex(x = 5, change_val = c(4,6))
```

```
## Warning in if (x < change_val) {: the condition has length > 1 and only the
## first element will be used
```

```
## [1] 15
```

You have at least 3 ways to vectorize any function. If you are only interested in vectorizing one argument at a time, you can use `sapply` as we have seen before:

```
sapply(c(4,8), multiplex)
```

```
## [1] 24 36
```

For more than one argument at a time you can use the flexible `mapply`, which allows us to specify the arguments to the function as vectors or lists:

```
# First call multiplex(x=4, change_val = 3), then
# multiplex(x=8, change_val = 7)
mapply(multiplex, x = c(4,8), change_val = c(3,7))
```

```
## [1] 10 36
```

Using `mapply` you can write your own wrapper functions (4.5.2) quite easily. If this is too much work for you, consider using the rather magical `Vectorize` (which really is itself a wrapper around `mapply`):

```
multiplex2 <- Vectorize(multiplex)
multiplex2(c(4,8))
```

```
## [1] 24 36
```

Inspect `?Vectorize` to find that you can specify which arguments will be vectorized, and whether to *simplify* the result (which works akin to `sapply`, making a vector or a matrix of a result where possible).

A final option is to write a *recursive* function that wraps around itself, executing the function once when the argument is not vectorized, and otherwise using `sapply` or `mapply` to return a vector of results:

```
multiplex3 <- function(x, change_val = 5){

  # Make the calculation when a single argument given:
  if(length(x) == 1){

    if(x < change_val){
      prod(1:x)
    } else {
      sum(1:x)
    }

  } else {

    # Otherwise use mapply, each time calling *this function*,
    # recursively.
    mapply(multiplex3, x = x, change_val = change_val)
  }

}

multiplex3(c(4,8))
```

```
## [1] 24 36
```

This final solution can be found in many base R functions, and is a flexible interface to allow vectorized arguments.

4.5.5 If, then, else ... switch

We have already seen the basic `if` statement in many examples, which can be extended with multiple `else if` statements:

```
# NOT an elegant use of multiple if statements.
animal_weight <- function(animal){

  if(animal == "cat"){
    weight <- 10
  } else if(animal == "cow"){
    weight <- 1000
  } else if(animal == "mouse"){
    weight <- 1
  } else if(animal == "dog"){
    weight <- 20
  }

return(weight)
}


animal_weight("dog")
```

```
## [1] 20
```

The use of many `if` statements like this cause a lot of repetitive code. Usually if you have many `if then else` statements, you can either think of a simpler design for your function, or you can use `switch` where appropriate:

```
animal_weight <- function(animal){


  weight <- switch(animal,
                   mouse = 1,
                   cat = 10,
                   dog = 20,
                   cow = 1000,
                   NA)  # Return NA if the animal is not one of those above

return(weight)
}
```

Here `switch` improves the readability of the code a lot. If you want to *switch* a numeric input, `switch` is perhaps not the best solution.

If you find yourself writing code like this:

```
# An input value
num <- 3
```

```
# DO NOT do this.
switch(as.character(num),
       "1" = "Door number one.",
       "2" = "Door number two.",
       "3" = "Door number three."
       )
```

```
## [1] "Door number three."
```

Then you are overthinking this! The above can be accomplished with basic indexing:

```
num <- 3

door_options <- paste0("Door number ", c("one","two","three"), ".")

door_options[num]
```

```
## [1] "Door number three."
```

Try this yourself Write the previous example as a custom R function that takes `num` as an argument, and the noun (here, "Door") as an optional second argument with a default value.

4.6 Robust functions: defensive programming

Once you write more functions that perhaps become more generic and widely applicable within your projects, you want these functions to be more robust, that is, to fail less often and more elegantly. Functions fail if the wrong inputs were provided to them (by a careless programmer or data faults), or if some (rare) exception is reached. In this section we will look at a few tools to help you write more robust functions.

4.6.1 Argument matching

Returning to our `animal_weight` function from the previous section,

```
# A slightly shorter version of animal_weight().
animal_weight <- function(animal){
  switch(animal,
           mouse = 1,
           cat = 10,
           dog = 20,
           cow = 1000,
           NA)

}
```

Here the last argument to `switch` indicates the value to return if the animal does not match any of the above. What if instead we want the function to stop and return a message if the value of the argument is not supported by the function? We can use `match.arg`, as follows:

```r
# Define animal_weight2 as a function which only allows 4 types of animal:
animal_weight2 <- function(animal = c("mouse","cat","dog","cow")){

  animal <- match.arg(animal)

  switch(animal,
             mouse = 1,
             cat = 10,
             dog = 20,
             cow = 1000)

}

# Now see what happens when we give a wrong animal:
animal_weight2("giraffe")
```

The output of the above will be:

```
Error in match.arg(animal) :
  'arg' should be one of "mouse", "cat", "dog", "cow"
```

Using `match.arg` resulted in a *very clear error message*. This sort of message makes it easy to find the source of the error.

A side-effect of using `match.arg` is that it performs *partial matching*, meaning you can also call the function with only part of the argument value, for example try `animal_weight("mo")`. Personally I dislike partial matching as it rarely benefits us, but can cause problems that are hard to find. Probably best to not rely on it in any real code, though it can help save a few key strokes.

4.6.2 Argument checking

Many functions expect a certain argument type, and make no sense or will fail when other argument types are provided. The function below make a sentence into sentence case, with the first letter of each word capitalized.

```r
sentence_case <- function(x){

  stopifnot(is.character(x))

  w <- sapply(strsplit(x, " "), function(ch){
    substr(ch,1,1) <- toupper(substr(ch,1,1))
    return(ch)
  })

  paste(w, collapse = " ")

}
```

Now this function call works fine,

```r
sentence_case("breaking news: man arrested for everything")
```

```
## [1] "Breaking News: Man Arrested For Everything"
```

```
sentence_case(100)
```

But this one does not (try it out yourself!).

4.6.3 Sending warnings, errors, messages

In more complex projects, it can be difficult to find which part of your code caused an error or unexpected result. Leaving useful messages, warnings, or exiting when you know things will go wrong later can be very useful in developing more debuggable code.

```
example_function_warnings <- function(data){

  stopifnot(is.vector(data))

  if(length(data) > 1000){
    warning("This might take a while!")
  }

  if(length(data) == 0){
    stop("Data is empty")
  }

  if(all(is.na(data))){
    message("Only missing data sent to example_function_warnings()")
  }

  # actually do something
  # ...
}
```

The use of `message` is nice to provide some indication to you where code was executed. Scattering a messages around your code (with an indication where we are in the code) can be invaluable during development of more complex projects, with lots of functions.

Sometimes you want *fewer* warnings or messages, for example arising from code that works fine but always produces an annoying warning. You can switch them off with these two functions:

```
suppressMessages({
  # code here that produces messages
})
```

```
## NULL
```

```
suppressWarnings({
  # Code here with warnings you want to suppress.
  # Don't use this unless you know what you are doing!
})
```

```
## NULL
```

4.6.4 Executing code when the function ends (or fails)

A useful tool to make functions more robust (less likely to fail) is to safely run some code when the function exits unexpectedly. One example arises with sending figures to an open PDF document: after we are done, the PDF needs to be closed with `dev.off()` (see Example in Section 4.4).

In this example, we have written a function that makes a simple plot in a pdf:

```
# Plot first 2 columns of a dataset against each other, make a PDF.
plot_first_2_columns <- function(data, output = "plot.pdf"){

  pdf(output)
  plot(data[[1]], data[[2]])
  dev.off()

}
```

This is OK, but what if we send a dataset with just one column, or some other corrupted data or wrong argument? The pdf will be opened but not closed, because `plot(data...)` will cause an error, causing the function to exit (and not run the rest of the code).

A better implementation uses `on.exit`, like so:

```
# Plot first 2 columns of a dataset against each other, make a PDF.
plot_first_2_columns <- function(data, output = "plot.pdf"){

  pdf(output)
  on.exit(dev.off())

  plot(data[[1]], data[[2]])

}
```

4.6.5 Catch errors with `try`

The ultimate tool to make functions more robust is `try`, a mechanism to catch errors and avoid termination of your code.

Although you can use `tryCatch` as an interface, I personally prefer the flexible use of `try`, like so:

```
# A wrapper around log(), avoiding an error when the input is bad,
# and instead returning a missing value.
safe_logarithm <- function(x){

  out <- try(log(x))

  # If log(x) caused an error, out will be a special
  # object with class 'try-error'
  if(inherits(out, "try-error")){

    return(NA)

  }
out
}
```

Now `safe_logarithm("a")` returns `NA`, but you do see the error message printed (unless you look around `?try` for a way to suppress the error message).

4.6.6 Writing database wrapper functions

This example combines various concepts from this section, and is a useful set of functions when you commonly work with databases. First review 'working with databases' if you need to (Section XXX). The functions below will be developed step-by-step. Please carefully review this section if you want to adapt this code to your situation.

After we are done reading and writing to a (usually remote) database, the connection to it should be *closed* with `dbDisconnect` (in the DBI package). Sometimes we forget, or code fails, or whatever - leaving extra connections open that can cause various problems (especially in production-ready code).

The following example can be modified to suite your needs (see Section @ref(???)) :

```
# A function to make a connection to your database
# Now, instead of copying this bit of code to all your scripts, you
# define the function once and run con <- db_connection()
db_connection <- function(){
  DBI::dbConnect(drv = PostgreSQL(),
                 user = "mjvgsdzg",
                 password = "my_password",
                 host = "balarama.db.elephantsql.com",
                 port = 5432,
                 dbname = "mjvgsdzg")
}


# A function to open a database connection, read and return data:
read_my_data <- function(tablename, ...){

  con <- db_connection()
  on.exit(dbDisconnect(con))

  data <- dbReadTable(con, tablename, ...)

}
```

With the above `read_my_data` you can read your favorite table from a database, simply with `mydata <- read_my_data("cooldatatable")`. However, this is a poor function if you are planning to read many tables from this database in a short time, then it makes sense to keep the connection open the entire time (especially for remote databases, as is common).

The following improvement allows passing a connection object, or creating it if it was not passed as an argument:

```
read_my_data2 <- function(tablename, con = NULL, ...){

  if(missing(con)){
    con <- db_connection()
    on.exit(dbDisconnect(con))
  }

  data <- dbReadTable(con, tablename, ...)

}
```

Now we can do,

```
my_con <- db_connection()

flowers <- read_my_data2("flower", con = my_con)
eggs <- read_my_data2("eggs", con = my_con)
butter <- read_my_data2("butter", con = my_con)
milk <- read_my_data2("milk", con = my_con)

dbDisconnect(my_con)
```

If we leave out the `con` argument, the connection will be opened and closed for you.

4.7 Exercises

4.7.1 Writing functions

1. Write a function that adds two numbers, and divides the result by 2.
2. You learned in Section 2.7.4 that you can take subset of a string using the `substr` function. First, using that function to extract the first 2 characters of a bit of text. Then, write a function called `firstTwoChars` that extracts the first two characters of any bit of text.
3. Write a function that checks if there are any missing values in a vector (using `is.na` and `any`). The function should return `TRUE` if there are missing values, and `FALSE` if not.
4. Improve the function so that it tells you which of the values are missing, if any (*Hint:*use the `which` function). You can use `message` to write messages to the console.
5. The function `readline` can be used to ask for data to be typed in. First, figure out how to use `readline` by reading the corresponding help file. Then, construct a function called `getAge` that asks the user to type his/her age. (*Hint:* check the examples in the `readline` help page).
6. **hard** Recall the functions `head` and `tail`. Write a function called `middle` that shows a few rows around (approx.) the 'middle' of the dataset. *Hint:* use `nrow`, `print`, and possibly `floor`.

4.7.2 Working with lists

First read the following list:

```
veclist <- list(x=1:5, y=2:6, z=3:7)
```

1. Using `sapply`, check that all elements of the list are vectors of the same length. Also calculate the sum of each element.

2. Add an element to the list called 'norms' that is a vector of 10 numbers drawn from the standard normal distribution.
3. Using the `pupae` data, use a t-test to find if PupalWeight varies with temperature treatment, separate for the two CO_2 treatments (so, do two t-tests). You **must** use `split` and `lapply`.
4. For this exercise use the `coweeta` data - a dataset with measurements of tree size. Split the data by `species`, to produce a list called `coweeta_sp`. Keep only those species that have at least 10 observations. (*Hint:* first count the number of observations per species, save that as a vector, find which are at least 10, and use that to subscript the list.) If you don't know how to do this last step, skip it and continue to the next item.
5. Using the split Coweeta data, perform a linear regression of `log10(biomass)` on `log10(height)`, separately by species. Use `lapply`.

4.7.3 Functions for histograms

First run this code to produce two vectors.

```
x <- rnorm(100)
y <- x + rnorm(100)
```

1. Run a linear regression y = f(x), save the resulting object. Look at the structure of this object, and note the names of the elements. Extract the residuals and make a histogram.
2. **Hard**. From the previous question, write a function that takes an `lm` object as an argument, and plots a histogram of the residuals.

4.7.4 Using functions to make many plots

1. Read the cereals data. Create a subset of data where the `Manufacturer` has at least two observations (use `table` to find out which you want to keep first). Don't forget to drop the empty factor level you may have created!
2. Make a single PDF with six plots, with a scatter plot between potassium and fiber for each of the six (or seven?) Manufacturers. (*Hint:* ook at the template for producing a PDF with multiple pages at the bottom of Section 4.4.

4.7.5 Monthly weather plots

1. For the HFE weather dataset (`hfemet2008`) that makes a scatter plot between PAR (a measure of light intensity) and VPD (a measure of the dryness of air).
2. Then, split the dataset by month (recall Section 4.3.5), and make twelve such scatter plots. Save the result in a single PDF, or on one page with 12 small figures.

Chapter 5

R on the web

5.1 Introduction

This short chapter will show how to interact with, and deploy web services from R. An API ('Application Programming Interface') is a way to access data or services on a server, for example to download or upload data. The most common architecture for APIs is known as RESTful Web Services (or 'REST API' for short). We will use the `httr` package to download data from REST APIs.

We start with downloading data from, and querying remotely stored databases, and include two working examples: a database (updated every 15min) in a MongoDB database, and a single table stored in a PostgreSQL database.

Packages used in this chapter

- By now standard packages, `lgrdata`, `dplyr`, `lubridate`
- `plumber` for launching web services.
- `DBI` to interface with databases.
- `dbplyr` to make this interface even easier.
- `purrr` (for `flatten_dfr`, though the package offers much more).

5.2 Reading data from remote databases

5.2.1 NoSQL

In this example we show how to read data from MongoDB - a popular NoSQL database. With the `mongolite` package it is very straightforward to read data that can be transformed to a dataframe. Here, every item in the database is a piece of JSON, but of equal length and with the same names, so that each item can become a row in a dataframe.

For a remotely stored MongoDB database, you first have to construct the URL, which includes a username and password. The URL looks like,

`"mongodb://username:password@hostingsite.com:portnr/databasename"`

The next example is a working example of a database stored on mlab.com, and grabs data using the `mongolite` package.

The database contains data on number of cars parked on 13 car parks in Almere, the Netherlands. The data are updated every 15 minutes.

```r
# A working URL (5/6/2019)
parkingdata_url <- "mongodb://guest:123password123@ds229186.mlab.com:29186/almereparking"

#
library(mongolite)

# Make a database connection. Here you don't download data yet, just
# set up a connection.
db <- mongo(collection = "almereparkingjson",
            url = parkingdata_url)

# We can now download the last 1000 rows of the data.
parkingdata <- db$find(limit = 1000, sort = '{"updated": -1}')
```

If you simply want to download all data, just do `db$find()`, it takes less than a minute.

Further reading The above example shows how to make a query with the `mongolite` package, you can read more options in the documentation.

5.2.2 SQL

Here, we briefly show how to interact with SQL databases. There are many R packages available, partly because there are many implementations of SQL databases (including postgres, which we show here). A unified interface is provided by the `DBI` package (link to documentation), which allows us to use nearly the same code for very different databases.

Here, we only show how to read rectangular data with the `DBI` and `dbplyr` packages.

With `dbConnect`, we setup a connection to the remotely available database (which does not download data yet). Using the connection we can list the tables available, download a table, and many other things.

```r
# General database interface
library(DBI)

# Drivers for postgres.
library(RPostgreSQL)

# For generating SQL queries.
library(dplyr)
library(dbplyr)

# Set up connection.
# !! This is a working connection (6/5/2019)
con <- dbConnect(drv = PostgreSQL(),
                 user = "mjvgsdzg",
                 password = "RsIxTmV9a95Gbb6Vo7IgG9wZNiv4Hq5L",
                 host = "balarama.db.elephantsql.com",
                 port = 5432,
                 dbname = "mjvgsdzg")

# List available tables (only one in this case)
dbListTables(con)

# Makes a direct connection to the table, but does
# not download anything yet:
cars <- tbl(con, "automobiles")

# We can download the entire table with collect,
# but note that in practice we rarely do this as databases may be extremely large.
cars_data <- collect(cars)

# We can now make queries using dplyr syntax, to collect only subsets of data.
# for example, find all cars with weight over 2000kg.
heavycars_query <- cars %>%
  filter(weight > 2000)

# What does the query look like in SQL?
show_query(heavycars_query)

# Now send the query to the database, collect the data.
heavycars <- collect(heavycars_query)


# Many other functions from dplyr or other can be used on databases,
# but not everything! Some trial and error may be necessary.
# Find all cars that have 'buick' in the name, using grepl:
buick_query <- filter(cars,
                grepl("buick", car_name)
               )
```

Try this yourself Run both examples above - getting data from a (continuously updated_ MongoDB database, and a simple static PostreSQL database. Try applying some `dplyr` filters and summaries on the data.

5.3 Accessing an API

A REST API is simply presented as a URL, to which a specific service can be accessed, and even parameters passed (when necessary). A URL may simply look like:

`https://someserver.com/getdata`

or may be more complicated, with queries, authentication, and so on. Let's look a simple example, using the `httr` package:

```
library(httr)

# The first step is to reach out to the server, and get a response.
# Here, g is a complicated object that we do not inspect directly.
```

```
g <- GET("https://opendata.cbs.nl/ODataApi/OData/37789ksz/TypedDataSet")

# Important: if the server could be reached, you have permission,
# and the service is available, you should see 200 as the response.
# Other HTML codes include 404 (service not found).
status_code(g)
```

```
## [1] 200
```

```
# We can read the actual data with the `content` function:
dat <- content(g)

# In this case, dat has two components - the interesting one is 'value'
uitkering_data_list <- dat$value
```

Unfortunately the result is a nested list, with NULL elements in some. To make this into a dataframe, we want to flatten each list, each of these will become a row in our dataframe, and bind all the rows together. This is such a common problem that I will show a robust solution:

```
library(dplyr) # bind_rows
library(purrr) # flatten_dfr

uitkering_data <- lapply(uitkering_data_list, flatten_dfr) %>%
  bind_rows
```

We now have data from the Dutch government on social benefits, a dataset that gets updated frequently. The use of the API directly rather than downloading the data manually is clearly a much better option.

As we saw in this example, the challenge is often not so much getting *something* from the web service, but often we have substantial work to shape it into a useful object in R (usually: a dataframe).

Try this yourself Try downloading data from this URL (containing Dutch demography data since 1898).

https://opendata.cbs.nl/ODataApi/odata/37556/TypedDataSet

5.4 Deploying an API with `plumber`

With `plumber` we can quickly set up an API, that we can use from within any other programming language, web site, or program. Normally you would deploy the API on a server running R, so that the R service can be used from anywhere. Here, to keep things simple, we deploy the API on your own machine ('locally'), but it will work similarly to when it runs on a remote server.

The first step is to create a standalone script (with extension .R) that defines as many *functions* as you like. Each function can take arguments. Take the following example, paste it in a script, call it `plumber.R`.

```
# plumber.R

#* Echo back the input, in uppercase
#* @param msg The message to echo
#* @get /echo
```

```
function(msg=""){
  list(msg = paste0("The message is: '", toupper(msg), "'"))
}

#* Plot a histogram
#* @png
#* @get /plot
function(){
  rand <- rnorm(100)
  hist(rand)
}

#* Return the sum of two numbers
#* @param a The first number to add
#* @param b The second number to add
#* @get /sum
function(a, b){
  as.numeric(a) + as.numeric(b)
}
```

The first function sends back an uppercase version of a message, the second function plots a histogram, and the third function adds two numbers. Obviously, you can write much, much more complicated R functions that can be subsequently called using the API. These functions are chosen because they return different things: text, a plot, and a number. Let's see how they work in practice.

With our API 'endpoints' (the three functions) defined, we can start the API via:

```
library(plumber)

# There are no other arguments besides a directory to find the file.
p <- plumb("plumber.R")

# Deploy the API.
p$run(port=8001)
```

Your R process will now be busy (until you close it), with a message like:

```
Starting server to listen on port 8001
```

This will also open up a `swagger` window, which is a simple interface to our own Web service. Here you can 'Try out' the various end points.

Try this yourself Try out the web service you just created as follows: open up a browser, and try each of the three endpoints:

```
127.0.0.1:8001/echo?msg="make me uppercase!"            127.0.0.1:8001/plot"
127.0.0.1:8001/sum?a=2&b=3"
```

Now go back to the script above, and notice how each endpoint is defined (`echo`, `plot`, and `sum`)

Further reading The above example comes straight from the `plumber` documentation, which you can read here: https://www.rplumber.io/

More complicated examples are available as well.

Chapter 6

Version control with git

6.1 Should I learn version control?

When you use version control, you can do the following:

- Keep a history of all the changes you have made to your code.
- Be able to revert to an old version, or briefly look around in old code, or find any code you have written at some time in the past.
- Undo fatal coding mistakes.
- Avoid having multiple versions of the same script (`myscrip_v1.R`, `myscript_v1_mod2.R` etc.), and commented-out bits of code that you no longer need.
- Collaborate with others on the same code, and easily merge the changes you and your collaborators make.
- Maintain an online backup of your code base.

These are all basic capabilities of version control, and as you can see an absolute requirement for even the somewhat serious coder. Here we use version control with *git*, the most popular system. In this chapter, I will give an informal introduction to version control, and focus on its use in Rstudio. The command line will also be introduced, particularly for somewhat more advanced options.

6.2 Basics of git

6.2.1 Basic principles

With git, every folder that you work in can have its own version control repository. This allows you to keep your projects neatly organized in separate folders, each with its own history and set of files that belong together. Organizing code in smaller projects also makes for a natural connection to *projects* in Rstudio, as we will see shortly.

Not all files in the folder need to be kept track of in the git repository. You can specify yourself which files you want to have under version control. Generally speaking it's up to you which files to track, but a few guidelines are helpful. Mind you that exceptions will arise for each of these ideas; every project is different.

- It is useful to follow the idea that *input files* (code, data) are the most important files in your project, whereas *output files* (figures, processed data, markdown reports, etc.) should be treated as temporary. If you set up your workflow well, you should always be able to generate all outputs from the inputs. As a result, generally speaking **it is a good idea to version control all input files, but none of your output files}.
- Avoid tracking binary files (non-text files). The two reasons are that a) git cannot see incremental changes to these files (only that the entire file has changed), and b) continuously updating larger binary files will increase the size of the git repository (a collection of hidden files storing your history) - eventually to unmanageable bloatedness.
- There is no need to track the Rstudio project file (*.Rproj), and probably better if you don't especially when collaborating.
- Text-based datasets can be stored very well in git repositories, but that git is not a data management tool per se. It is therefore a bad idea to include *large* datasets, only smallish datasets in the repository.

You can decide manually which files to track, and you can also "ignore" files by listing them in a special file, `.gitignore`, as we will see later.

With git, you can 'store' a snapshot of all the tracked files in your repository as often as you like. Normally, this snapshot represents a significant change you made to your code, some new feature added, new figures coded, or whatever. It can be any point during your work where you think it may be useful to return to. With every snapshot, you can decide which changes to the tracked files should be included ('staged'). It is thus **important to realize that a version control repository acts on multiple files in the folder**, not separately for each file. This way we can also add and delete files, and keep track of their histories as well (e.g. return to a version of the repository when a particular file was not deleted).

When we store a snapshot of the repository, this is called a **commit**. When we commit changes to the files in the repository, we do this because we may wish to return to this point, or clearly document the changes we have made. When a commit is made, a message has to be written to document what we have done (as we will see further below). Every time we commit, we can manually select which files should be added to the next snapshot. This process is called **staging**, and it allows you to decide to keep some changes, and undo all other changes, for example.

Further reading It needs to be stressed that this chapter is a brief non-technical introduction to the topic. It only covers the basic capabilities of git, and stresses the use of Rstudio as an interface to git. You can find many *git* tutorials online if you want a bit more depth.

6.2.1.1 Local and remote repositories

An important concept when starting with git is that there is a distinction between **local** and **remote** repositories. With git, your version control **is fundamentally done on your computer, that is, it is a local repository**. Optionally, this local repository may be synced with a remote repository (on some online web service). It is thus important to understand that **a)** you do not need to have a remote repository for git to function (you do not need an internet connection) and **b)** you must have a local repository even if you also have a remote one.

Finally, do not confuse 'github' with 'git', the former is a website (github.com) used for hosting remote git repositories, the latter is the actual version control system. There are other websites of that kind (for example bitbucket.org). The inventor of git has nothing to do with these websites.

6.2.1.2 Rstudio projects

If you end up working primarily from Rstudio, we very strongly recommend the use of **Projects** in Rstudio to keep your work organized, and to help set the working directory.

Rstudio projects are really just small files added to a certain folder, which "tags" that folder as containing a particular project. You can switch easily between different projects, or have multiple instances of Rstudio open, all with different projects (something I do a lot). When you switch to a project, *it also sets the working directory* to the folder where the .Rproj file is located.

We make an Rstudio project in Section 6.3.1 and also return to projects in Section 7.2.

6.2.2 Installing git

To get started, first install the actual git program from https://git-scm.com/. During the installation process, do not change any of the defaults, just press OK when prompted.

The first time you use git on a computer, you have to set your name and email. These will be appended to each time you commit (very useful when collaborating!).

Normally you can open a 'git bash shell' (a command line window) from Rstudio, as we show in Section 6.6. This first time, however, you *might* have to find `Git Bash` in the list of installed programs.

With the shell open, type the following commands, using the name and email address you want appended to each commit. You can change this setting any time, the current setting will be applied to the next commit.

```
git config --global user.name "My Name"
git config --global user.email "myemail@somewhere.com"
```

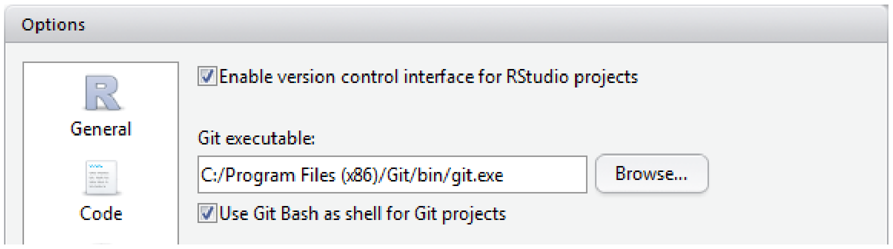
To check the settings configuration file (which is stored in your home directory), use :

```
git config --list
```

You can now close the shell. We return to the shell in a later section. The built-in menu's in Rstudio are fine for 95% of our work with git.

6.2.3 Using git in Rstudio

After installation, Rstudio may have trouble finding the git executable (a reboot may help, but this does not always fix the problem). We can let Rstudio know where git was installed by clicking `Browse` in the following screenshot (on Windows it will most likely be installed in this exact directory).



☑ Enable version control interface for RStudio projects
Git executable: C:/Program Files (x86)/Git/bin/git.exe
☑ Use Git Bash as shell for Git projects

Click `Browse` to find the directory where git.exe is installed, on Windows it will be this exact directory. You may not have to do this if you rebooted after installing git.

6.3 Local git repositories

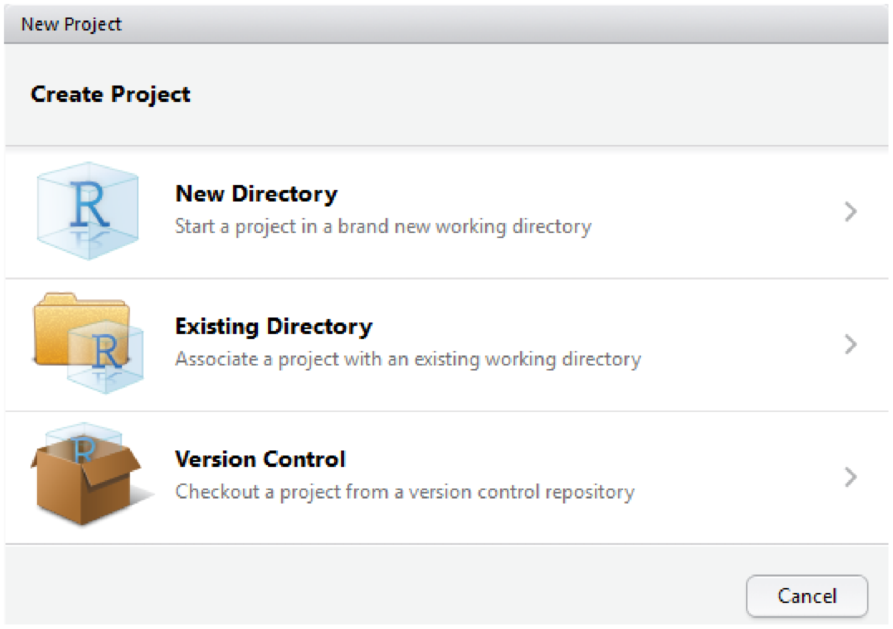
Before we turn to remote repositories, we first need a basic understanding of local repositories. Unlike some version control systems, git can work with only a local repository; this can be very useful when you want to commit changes when you are offline.

The workflow with a local git repository is as follows:

- Make changes to your code
- Stage changes, this will mark the changes to the files to be added to the next 'commit'.
- Commit your changes to the local git repository.
- Repeat.

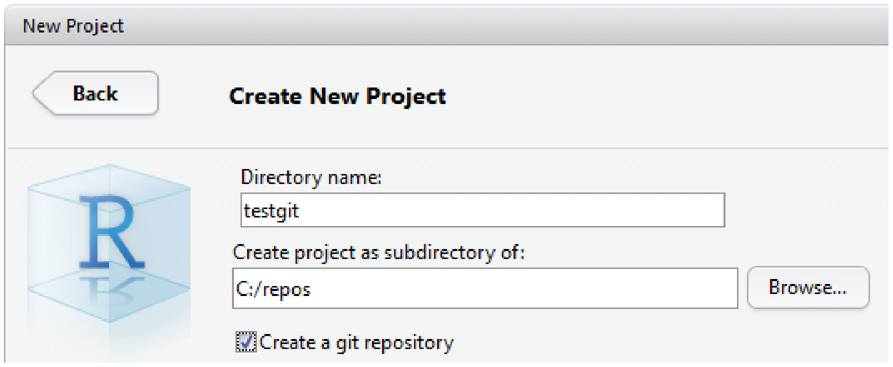
6.3.1 A first session with version control in Rstudio

Let's start a new Rstudio project with version control. In this example we will make a simple local project, with no remote repository defined. We do that in Section 6.4.


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

To start a new, local, project with git enabled click 'New Directory' (not Version Control!). Alternatively you can use an 'Existing Directory' and create a project there.

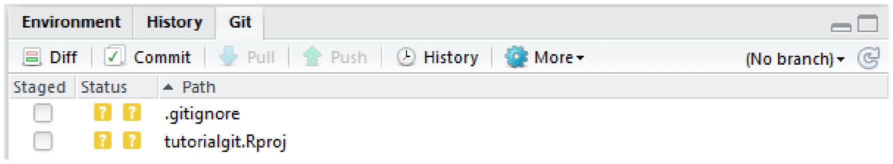
On the next tab, select `Empty Project`, and the following window appears. Here, give the project a reasonable name- it will be used to create a new folder if you selected 'New Directory' in the above window. Now, also check the box that says 'Create git repository'.



Directory name: testgit
Create project as subdirectory of: C:/repos
Browse...
[✓] Create a git repository

Enter a name for the new project, the directory where the project will be created (as a subdirectory with the project's name). Check 'Create a git repository' to enable git.

Click OK, and if successful, you wil switch to a new instance of Rstudio. The git tab should look like the following figure.

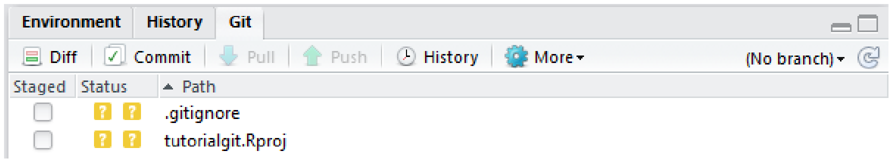


Staged	Status	Path
☐	? ?	.gitignore
☐	? ?	tutorialgit.Rproj

What your git tab should look like if you started an empty project in a new directory.

The two files in the folder show up with two question marks. This means that git is not sure if you want to track these files. By default, it won't track them. The `.gitignore` file is a special file that lists all the files you always will want to ignore. It is very useful and we will use it later.

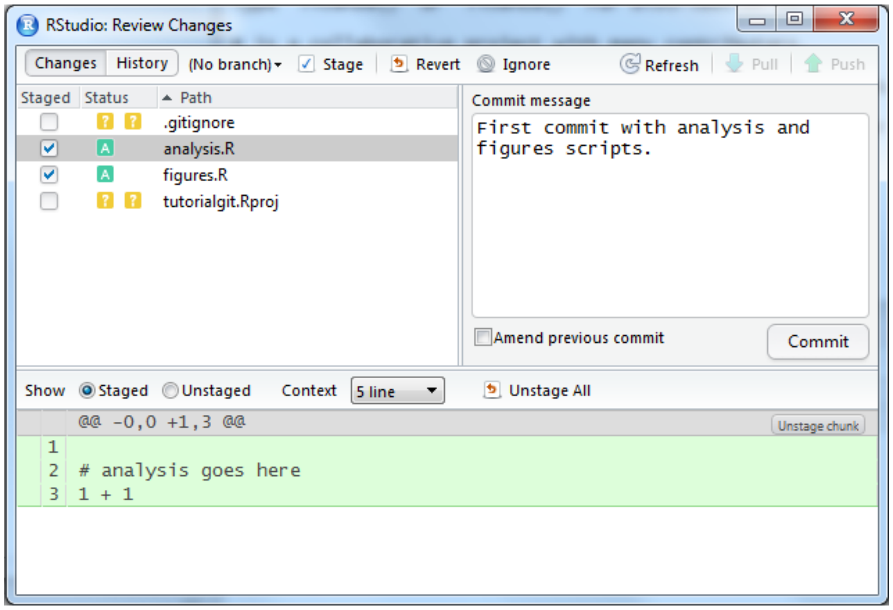
Now, let's open two new scripts and write some code. I have called my new scripts `analysis.R` and `figures.R`. As you expect, these two files will show up in the git tab with two question marks. Notice the `Staged` column in the above figure? Let's stage both files by checking them, as in the below figure.



Staged	Status	Path
☐	? ?	.gitignore
☐	? ?	tutorialgit.Rproj

Two new files are staged for the next commit. The green 'A' means 'Add', because these files were not already being tracked by the git repository.

We will now make our **first commit**. With the files staged as in the screenshot, click the 'Commit' button.

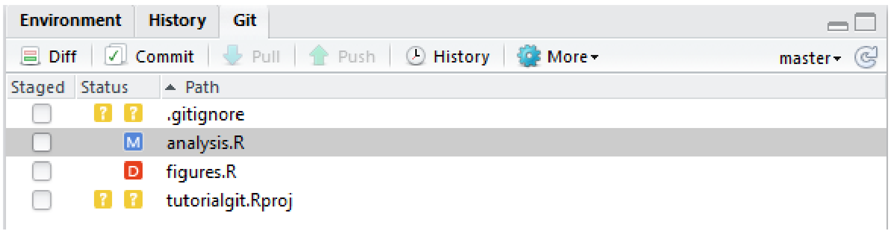


```
First commit with analysis and
figures scripts.
```

```
@@ -0,0 +1,3 @@

# analysis goes here
1 + 1
```

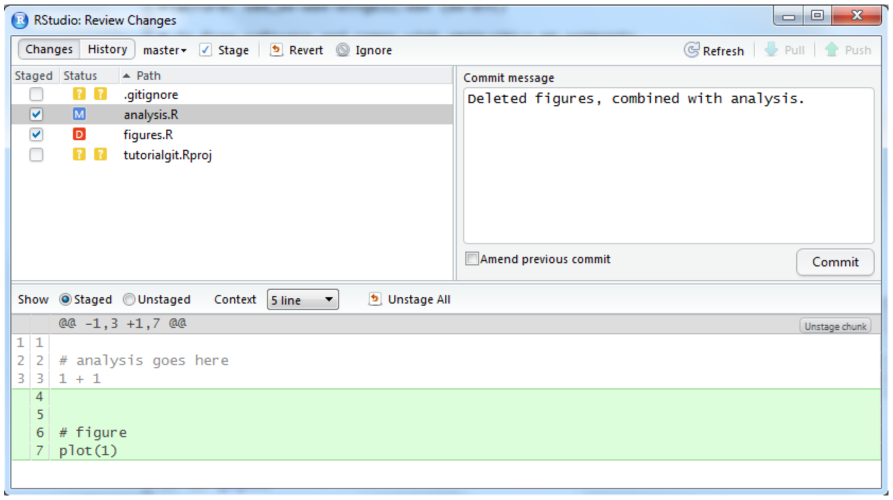
With the two new files staged, you can commit their changes to the local git repository. You must first write a meaningul message in the box on the right.

The files will now have disappeared from the git tab. That's good - their versions are the same as the last time you committed, that is, they have not changed in content. Now let's modify `analysis.R`, and delete `figures.R` (maybe you decided to combine code from these files into a single file), and see what happens.



Here we have modified (M) and deleted (D) files that are being tracked by git.

Now, stage both files, and press Commit. Write a message that briefly describes your changes and click 'Commit' in the next window as well:



Staged	Status	Path
☐	? ?	.gitignore
✓	M	analysis.R
✓	D	figures.R
☐	? ?	tutorialgit.Rproj

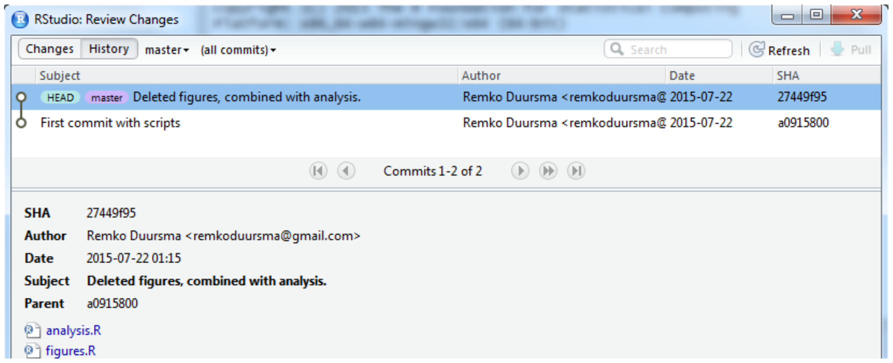
Commit message

```
Deleted figures, combined with analysis.
```

```
@@ -1,3 +1,7 @@

# analysis goes here
1 + 1


# figure
plot(1)
```

With the modified and deleted file selected, write a message and click Commit.

Let's now look at the history of our first project sofar. Find the 'History' button in the git tab. It will look like the following screenshot.



Subject	Author	Date	SHA
HEAD master Deleted figures, combined with analysis.	Remko Duursma <remkoduursma@	2015-07-22	27449f95
First commit with scripts	Remko Duursma <remkoduursma@	2015-07-22	a0915800

SHA	27449f95
Author	Remko Duursma <remkoduursma@gmail.com>
Date	2015-07-22 01:15
Subject	Deleted figures, combined with analysis.
Parent	a0915800

analysis.R
figures.R

History of the project sofar. The HEAD tag means this is the current location of the working directory. As we will see later, we can check out older versions, in which case HEAD will move to whatever commit we are looking at. Shown are the messages we wrote when committing, the author and date (taken from the git configuration), and the SHA.

The 'SHA' is the unique code given to each commit. You normally don't have to use it in Rstudio, but it is important in the command line commands given below. We never need all characters from the SHA, only the first few (normally 5 or so). You can find the (abbreviated) SHA in the right-hand column of the history.

The question marks for `.gitignore` and the `Rproj` indicate that these files have not been staged in any previous commit. If you do not want to ever track them, see Section 6.5.

6.3.2 Deleting files and sensitive information

Some new users are surprised by how git deals with deleting files in your repository. If you delete a file, git will mark the file as deleted (with a red D in the Rstudio interface), and you can stage the change to also remove the file from git (so it won't be tracked anymore). Even though the file is deleted, it is still contained in the *history* of your repository. That means you (or someone else) can return to a state of the repository that included the file, and read its contents.

If you accidentally committed a file with sensitive information, such as passwords, deleting the file does not remove it from the history. There is also a way to completely remove files, even from the history, but this is a bit cumbersome and best avoided. Normally we do not like to change history unless some of the files are very large or contain sensitive information.

6.3.3 Reverting changes

To undo changes to files in Rstudio, and reset to the last known version of that file (i.e. the the last commit), select the file and click the 'Revert' button (this button is only visible in the Commit window, otherwise it is grouped under the 'More' button). Sometimes this menu in Rstudio is a little *buggy*, reverting only one file at a time.

6.4 Remote git repositories

Version control with git becomes especially useful if you host your code in an online ('remote') repository. It is an essential step for collaboration (even if this is just with yourself, on multiple computers). Popular sites to host git repositories are <www.github.com> and <www.bitbucket.org>. I will not provide a guide to using either site, I will leave this to you to figure out. Either way, the first step is to create a user account, and set up the remote to be able to communicate with your computer.

A few tips to get started:

- A remote repository is simply a copy of your local project, think of it as a backup (with few extras).
- It is generally a good idea to organize your local repositories in a single directory, a location with a short path is useful since you will be accessing your repositories via the command line.
- You can add git version control to an existing project, but the easiest, most fail-safe way is to create a remote repository, then make a new project in Rstudio pulling from that repository, and adding your files into the project. We will follow this approach below.

6.4.0.1 HTTPS or SSH?

There are two ways to connect to a git repository hosted on a remote (like github). The easiest setup is connecting via HTTPS, but this will require you to enter your password every time you try to update the remote. I recommend HTTPS only when you are using a public computer.

A better approach is to use SSH, since you never have to type your password - a sort of password will be saved on your computer.

6.4.0.2 Setting up SSH

If you are using SSH for authentication, you must first make a sort of password (an 'SSH key'), which is stored on your computer as well as on the hosting site. This is a straightforward process thanks to Rstudio.

- Go to `Tools/Global Options...` then click on the `Git/SVN` button, and note the 'SSH/RSA key' field.
- This field is probably blank, now click on `Create RSA Key`. Leave the passphrase blank (unless you would like an extra layer of security), and click OK.
- Next, click 'View RSA Key' (a small link), and copy the key (Ctrl-C).

The next step is to paste the SSH key in the remote hosting site, so it knows who you are when you try to use the repository.

On bitbucket, when logged in, click 'Manage Account', and find 'SSH Keys' on the left. Then click 'Add Key', and you now have to paste the key that you copied in the previous step.

On github, go to Settings/SSH and GPG keys and make a "New SSH key", and paste it in as above.

You can now paste it into the field. Also add a nice label for the key (identifying the computer you are working on).

6.4.1 Making a remote repository

Now, let's create a remote repository. On bitbucket (when you are logged in), click the "+" in the left-hand side menu, on github click the plus ('+') button in the top right and `Create repository`. Decide for yourself on the settings for the repository, such as Private/Public etc.

The key thing for us is to find the address of the repository, we need this to use the remote in Rstudio. When using SSH, it will look like (on bitbucket):

```
git@bitbucket.org:remkoduursma/tutorialgit.git
```

When using HTTPS, it will look like this:

```
https://remkoduursma@bitbucket.org/remkoduursma/tutorialgit.git
```

6.4.1.1 Adding a remote repository to an existing project

To do this, you have to use the command line. If you want to use only Rstudio's menu's, skip to the next section.

With the command line (bash) open in the working directory, type the following two commands where you replace the *github* URL with the one matching your remote repository.

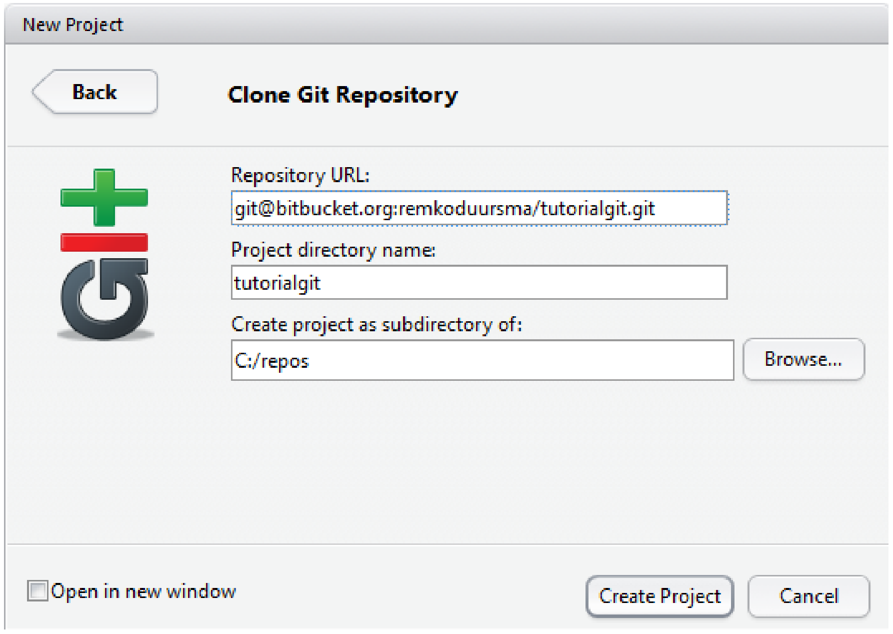
```
git remote add origin git@github.com:USERNAME/REPOSITORY.git
git push -u origin master
```

The remote repository will now be an exact copy of your local one (besides untracked files).

6.4.1.2 Cloning : make a local copy of a remote repository

The easiest approach is to start with an empty remote repository and use it as the basis for a new project in Rstudio. Then, you can add your files to this repository and start tracking (a selection of) them.

First, make a remote repository, find the address (SSH or HTTPS, see previous section), and copy it. In Rstudio, go `File/New Project.../Version Control/Git` and paste the address in the URL box (HTTPS or SSH, it will be recognized). Rstudio is going to create a (nearly empty) folder with a git repository. It will create this as a subfolder of the directory in the window you see (the third field). I recommend you keep your git repositories in a local folder in the root, **that contains no spaces in the name**. For example, `c:/repos/`.



Repository URL: git@bitbucket.org:remkoduursma/tutorialgit.git

Project directory name: tutorialgit

Create project as subdirectory of: C:/repos

☐ Open in new window

Paste the remote address of the repository in the first field, the second field will automatically get the name of the remote repository, and for the third field select a directory yourself (but see text).

When you click OK, Rstudio should open up with the new project.

6.4.2 Workflow

Whether or not you have a remote repository enabled, you will still work in a local repository as explained in Section 6.3, and occassionally synchronize the two. In other words, **committing changes is always local**. When you want to sync with a remote, you either do a **push** (send your commits to the remote repository), or a **pull** (get the latest commits from the remote that you don't have).

It is important to note that **you can only push if the local up to date with remote**. We will see in the working example how to deal with the situation where we are 'behind' the remote (that is, have not included some commits stored in the remote repository).

The standard workflow for code stored in a remote repository is:

1. Pull - this makes sure you start with the version contained in the remote repository.
2. Write code, commit changes.
3. Repeat step 2. until you are finished with this session.
4. Push - this sends all the commits you made to the remote repository.

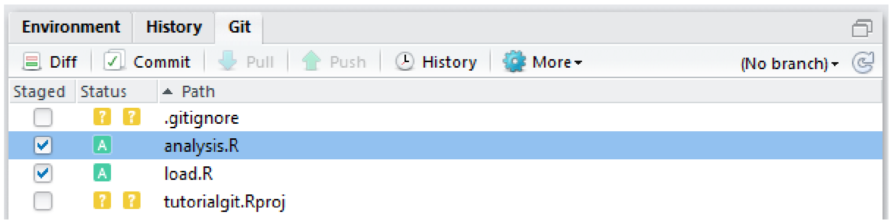
It is thus important to understand that **you don't have to push every time you commit changes**. In fact it is better if you don't, because it is easier to fix problems in the local repository, than when the commits have been pushed to the remote.

6.4.3 A session with a remote repository

In this section, I will show a typical workflow with a remote repository. In this example I assume you have made a remote repository, and we will clone the empty repository into a new Rstudio project (see

'Cloning' in Section 6.4.1.

If these first two steps - explained in the previous section - were successful, you now have a new, empty project open in Rstudio. I have copied two scripts that I wrote previously into the folder. In the git tab in Rstudio, you have to 'stage' the added files by clicking the check box. The two new files now show up with a green 'A' :

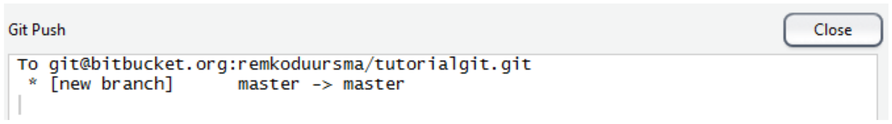


Staged	Status	Path
☐	? ?	.gitignore
✓	A	analysis.R
✓	A	load.R
☐	? ?	tutorialgit.Rproj

Adding two files to be included in the next commit. Note that 'Push' and 'Pull' are greyed out, this is normal because we have no commits in the repository.

Next, click commit (opens up a new window), write a reasonable message in the text box on the right, and click 'Commit'. You now have a single commit in the **local** repository. Again, it is crucial you understand that this commit is not yet on the remote.

Let's **push** this first commit to the remote repository. Just click 'Push' on the git tab, and a new window opens up in Rstudio with the response from the remote. It will say something like,

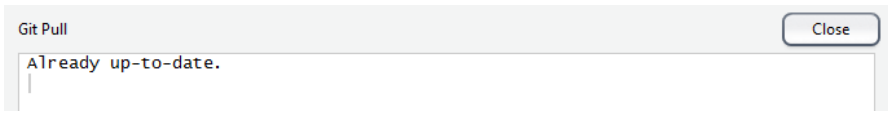


```
To git@bitbucket.org:remkoduursma/tutorialgit.git
 * [new branch]      master -> master
```

Response of the remote after the first push (opens in Rstudio when you click 'push'). Note that a new branch is created because this is the very first commit on the remote repository. All commits will always go to the 'master' branch (in this tutorial, at least).

Now go to the online version (on github.com or bitbucket.org), and check that your code is really there. Also look for the 'commits' (a list of all your commits) and the source code (you can, quite conveniently, browse the code online).

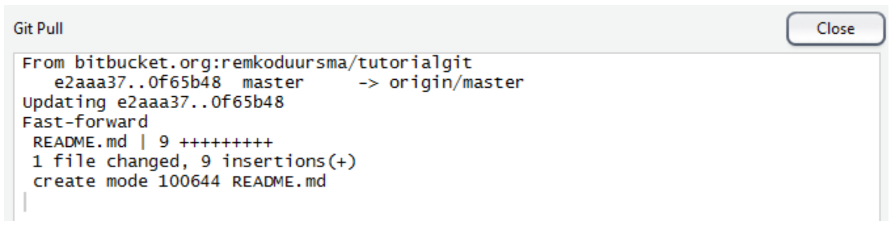
Next, suppose you start a new session with this project. As described in Section 6.4.2, you should always start with a **pull**, except when you are really 100% sure that your local is already up to date with the remote. If you are already up to date, the output will be:



```
Already up-to-date.
```

Output from a 'pull' when the local repository is already synced with the remote.

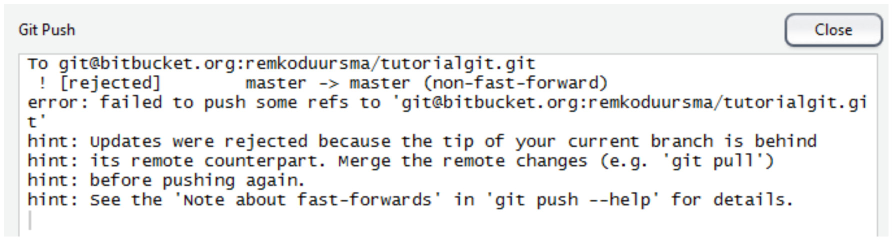
In the next example, I added the file `README.md` to the repository on another computer. This time, the output from 'pull' looks like:



```
From bitbucket.org:remkoduursma/tutorialgit
   e2aaa37..0f65b48  master     -> origin/master
Updating e2aaa37..0f65b48
Fast-forward
 README.md | 9 +++++++++
 1 file changed, 9 insertions(+)
 create mode 100644 README.md
```

Output from a 'pull' when the local repository is behind the remote. This is a successful message. It says that the two repositories have been successfully merged.

Finally, a common situation arises where you have not pulled from the remote, have done work locally and committed the changes. It is possible that the remote includes changes that you don't have, for example commits created by a collaborator and pushed to the same remote repository, or work done by you on another computer. If the remote is ahead of you, and you are trying to push local commits that the remote does not yet have, you will see the output as in the following screenshot.



```
To git@bitbucket.org:remkoduursma/tutorialgit.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@bitbucket.org:remkoduursma/tutorialgit.gi
t'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Output when attempting to push to a remote when the remote is ahead of the local repository.

The solution to this problem is to first do a pull, and then try to push again. The amazing thing about git is that even if your local commits and the commits on the remote that you don't yet have include changes on the same files, they will be automatically merged in the new commit.

There are cases, however, where **conflicts** arise - if the local and remote changes are on the same line of text. In that case, you will have to fix the conflicts manually. It is however much better to avoid these conflicts by always **beginning your session with a pull, and ending with a push**. That way you minimize the risk of changing the same line of code twice.

6.5 Ignoring files

As discussed, you don't want all of your local files to be copied to the remote repository. These include private data, large files, most binary files, sensitive information like passwords.

You can keep a list of files to ignore in the file `.gitignore`, which will most likely already be generated for you in the working directory.

Here is an example, with some useful settings

```
# No need to track the Rproj file
*.Rproj

# Private data: all files in data/ and config/
data/
config/

# Ignore all these types
*.xlsx
*.pdf

# ... except this file
!public_result.pdf
```

You can inspect which files are being tracked with the command line:

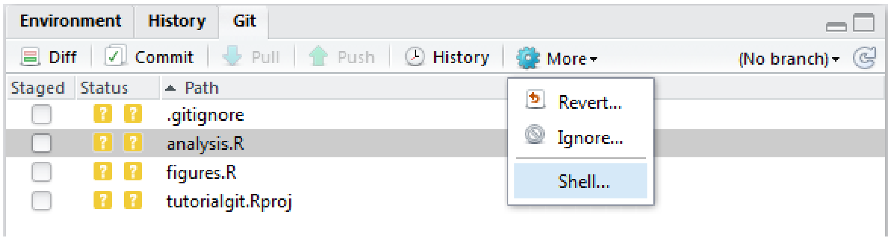
```
git ls-files
```

It is also always a good idea to inspect the remote repository to see if anything was uploaded you did not want to share. If you accidentally do push passwords to a remote repository, the best way is to **delete the remote repository and make a new one with the same name**.

6.6 Using the command line

Rstudio is very convenient for day-to-day version control, but for more advanced options, you must know how to use the command line. You may also find that when you gain more *git skills*, that using the command line is just as convenient while providing more control.

First, we show you how to open the 'shell', a command window where the full capabilities of git can be used.



From the 'Git' tab in Rstudio (same pane as Environment and History), open the Shell like this.

The shell is different from the Windows command line! It comes installed with git, and recognizes most unix commands.

Note that **if the output is longer than the screen, output will end with a ':'. In that case, press `q` to return to the command prompt}.

6.6.1 Basic commands

I will not cover git via the command line in detail here, there are many excellent resources available. Needless to say, everything you can do in Rstudio with a click of a button can be achieved in the command line, and much, much more. Quite useful is a sort of 'history',

```
git log
```

which lists each commit (and its SHA and commit message). If you only want to see the last, say 2 commits, do,

```
git log -2
```

and remember to press `q` when the output is longer than the window to return to the prompt.

6.6.2 Useful tasks

This section lists a number of useful things you can do with git from the command line. It is by no means exhaustive.

6.6.2.1 Adding a remote to a local repository

Suppose you have a local project as the example workflow in Section 6.3.1, and you want to start a remote repository. You now simply have to let git know that you have a remote as well. To do this, you have to open a Console (see Section 6.6) and use this command:

```
git remote add origin git@bitbucket.org:remkoduursma/tutorialgit.git
```

where the right-hand side is the address of your remote (found in the last section).

Then, make your changes as usual, and use this special `push` command the first time you push to the remote (after that, you can do the usual simple 'git push').

```
git push -u origin master
```

6.6.2.2 List all tracked files

If you are unsure which files are being tracked, you could browse the remote repository (if you have one), or use this command:

```
git ls-files
```

6.6.2.3 Looking around in an old version

Perhaps you don't actually want to undo changes, but instead inspect some older code, from which you can copy-paste bits you accidentally deleted. We can use the `checkout` command for this, which let's you temporarily look at an old commit.

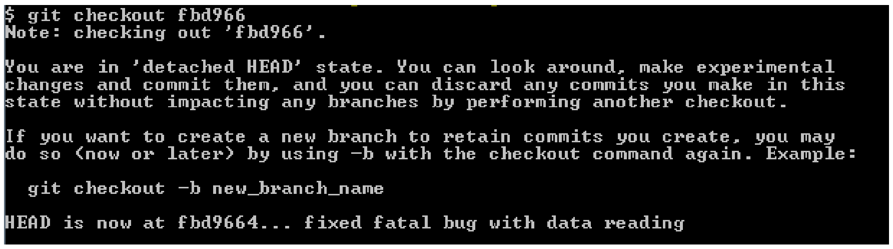
First we find the SHA of the commit, either in the history (in Rstudio), or via `git log` as shown above.

```
git log -2
```

Now suppose we want to 'check out' (look around and browse) the commit with SHA `fbd966` (recall, you only ever need the first 5 or 6 characters of the commit).

```
git checkout fbd966
```

It is crucial you understand that you can now only look around, but you have not reverted to this old version! This is very useful to take a look at old code, maybe copy some bits and pieces to a new file, and then return to the current version. To make sure you know this, git tells you as much:



```
$ git checkout fbd966
Note: checking out 'fbd966'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at fbd9664... fixed fatal bug with data reading
```

Output of the command line after doing a checkout. Basically git is trying to tell you your visit to this old version of the code is for looking around only.

To return back to the current, last version, do:

```
git checkout master
```

6.6.2.4 Undoing uncommited changes

Suppose you have committed your work, then made a bunch more changes that you are very unhappy with. But, luckily, you haven't committed them yet. You can use the 'Revert' button in Rstudio (as mentioned previously), but this really removes any chance that you *maybe* want to use the new edits.

Using the command line, you can use the `git stash` command as follows:

```
git stash
```

The files will not automatically appear as they were in the last commit. The advantage of using `stash` here - instead of the Revert button in Rstudio - is that if you change your mind you can do `git stash apply`. Basically your uncommitted changes are kept in a 'stash', not deleted.

6.6.2.5 Deleting the last commit

Suppose you have made changes, committed them, and then found out these are terrible and you want to get rid of them. We can use the `reset` command. **Use this at your own risk, it cannot be undone!**.

First let's look at the last two commits,

```
git log -2
```

So we want to get rid of commit `f76gb`, and revert back to `hu88e`.

```
git reset --hard hu88e
```

Instead of `--hard`, you can use the `--soft` option to revert back to the old commit, but keep the changes since then as unstaged changes to the files. This is useful if you want to keep some of the changes, but manually remove or fix some other changes.

6.6.2.6 Deleting files

We already saw that when you delete a tracked file from a git repository, you have to stage the change just like you stage a modification to a file. Here are a couple of useful commands related to deleting files locally.

Remove from local repository (i.e. stop tracking) all locally deleted files:

```
git rm $(git ls-files --deleted)
```

The next command is handy to clean your working directory. Use this at your own risk though!

Delete all untracked files:

```
git clean -f
```

To do the same, but include subdirectories (use this at even more own risk!):

```
git clean -f -d
```

6.6.2.7 Searching the history

There are a few ways to search the history - all of them require the command line. It may be useful to find code you deleted (on purpose), or to find all the occassions where some word occurs in changed lines of code, or list all the commits that acted on a particular file.

Find all commits with ‘someword’ in the commit message.

```
git log --grep=someword
```

Find all commits where ‘someword’ occurs in changed lines of code

```
git log -G someword
```

List all commits that include changes on ‘somefile.R’. This even works when the file was deleted at some point.

```
git log -- somefile.R
```

6.6.2.8 Accessing git help files

You can learn more about the many options of git via the help files, each of these can be accessed like this.

Show help file for ‘log’.

```
git log --help
```

Perhaps more usefully, if you don’t know how to do something in git, simply type your question in Google (“git how to delete untracked files”).

Chapter 7

Project management and workflow

7.1 Tips on organizing your code

In this chapter, we present a few tips on how to improve your workflow and organization of scripts, functions, raw data, and outputs (figures, processed data, etc.). The structure that we present below is just an example, and will depend on the particular project, your requirements, how much time you have, and personal preference.

The main **challenge** in developing more complex workflows, where you have multiple data sources, scripts for various analyses, and outputs of various kinds (figures, markdown documents, prepared data etc.) is to keep things organized, avoid *clutter*, and make sure you know how the outputs were produced.

All projects are different, and we encourage you to experiment with different workflows and organization of your script(s) and outputs.

The following is a **rule of thumb** list for R project management:

- Use 'projects' in Rstudio to manage your files and workspace.
- Use *git* version control (see Chapter 6).
- Use a logical folder structure inside your projects, keeping similar files together (data, scripts, output, etc.).
- Avoid writing long scripts, instead break them into a logical collection of shorter scripts.
- Load all required packages in a separate script.
- Outputs (figures, processed datasets) are *disposable*, your scripts can always re-produce the output.
- Keep function declarations separate from other code.
- Write functions as much as possible.
- Add a 'README.md' file to your project, markdown-formatted file explaining what the project does, a list of any dependencies, how to run the code, where to find the output, etc.

In this chapter we show an example project structure, which uses most of the above rules to come up with a transparent project workflow. If you follow (something like) the structure we show here, you have the added benefit that your directory is fully portable. That is, you can zip it, email it to someone, they can unzip it and run the entire analysis.

For effective project management we find using custom functions to organize our work most useful. See Chapter 4 for general introduction to functions, and Section 7.6 on how to organize your code with functions.

7.2 Set up a project in Rstudio

The most important tip is to *use projects in Rstudio*. Projects are an efficient method to keep your files organized, and to keep all your different projects separated. There is a natural tendency for analyses to grow over time, to the point where they become too large to manage properly. The way we usually deal with this is to try to split projects into smaller ones, even if there is some overlap between them.

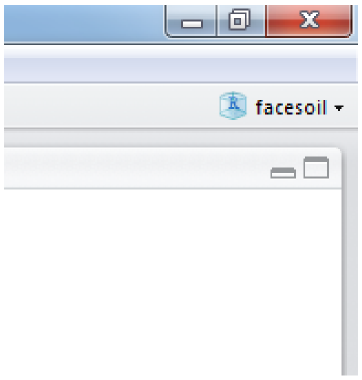
Caution Stop using `setwd()` in any of your scripts. This is never a good idea, for various reasons. Instead use Rstudio projects as a way to set the working directory automatically (and cleanly).

You can also stop using `rm(list=ls())` in any of your scripts. The problem with this command is that it does not clean *everything* : all packages are still loaded, and hidden objects also remain (ones starting with `.`), and certain `options` may have been set. Instead, test reproducing your project by selecting `Session/Restart R` and running the project.

In Rstudio, click on the menu item `File/New Project...`. If you already have a folder for the project, take the 2nd option (`Existing directory`), otherwise create a folder as well by choosing the 1st option (`New project`). We will discuss "version control" in the next chapter.

Browse for the directory you want to create a project in, and click `Choose`. This creates a file with extension `.Rproj`. Whenever you open this project, Rstudio will set the working directory to the location of the project file. If you use projects, you no longer need to set the working directory manually as we showed in Section 1.8.

Rstudio has now switched to your new project. Notice in the top-right corner there is a button that shows the current project. For the example project 'facesoil', it looks like this:

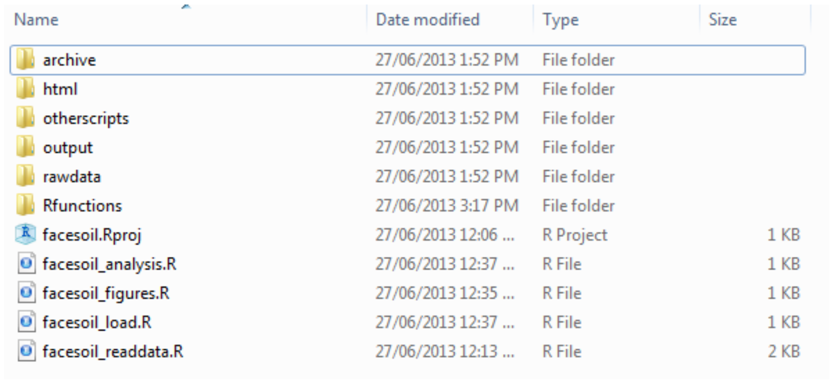


The Project button in Rstudio

By clicking on that button you can easily switch over to other projects. The working directory is automatically set to the right place, and all files you had open last time are remembered as well. As an additional bonus, the workspace is also cleared. This ensures that if you switch projects, you do not inadvertently load objects from another project.

7.3 Directory structure

For the 'facesoil' project, we came up with the following directory structure. Each item is described further below.



Name	Date modified	Type	Size
archive	27/06/2013 1:52 PM	File folder	
html	27/06/2013 1:52 PM	File folder	
otherscripts	27/06/2013 1:52 PM	File folder	
output	27/06/2013 1:52 PM	File folder	
rawdata	27/06/2013 1:52 PM	File folder	
Rfunctions	27/06/2013 3:17 PM	File folder	
facesoil.Rproj	27/06/2013 12:06 ...	R Project	1 KB
facesoil_analysis.R	27/06/2013 12:37 ...	R File	1 KB
facesoil_figures.R	27/06/2013 12:35 ...	R File	1 KB
facesoil_load.R	27/06/2013 12:37 ...	R File	1 KB
facesoil_readdata.R	27/06/2013 12:13 ...	R File	2 KB

Folder structure; just an example

7.3.1 `rawdata`

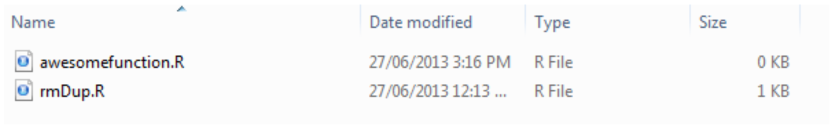
If your project contains any raw data files (within) *keep your raw data separate from everything else*. Here we have placed our raw CSV files in the `rawdata` directory.

In some projects it makes sense to further keep raw data files separate from each other, for example you might have subfolders in the rawdata folder that contain different types of datasets (e.g. 'rawdata/leafdata', 'rawdata/isotopes'). Again, the actual solution will depend on your situation, but it is at least very good practice to store your raw data files in a separate folder.

7.3.2 `Rfunctions`

If you do not frequently write functions already, you should force yourself to do so. Particularly for tasks that you do more than once, functions can greatly improve the clarity of your scripts, helps you avoid mistakes, and makes it easier to reuse code in another project.

It is good practice to keep functions in a separate folder, for example `Rfunctions`, with each function in a separate file (with the extension `.R`). It may look like this,



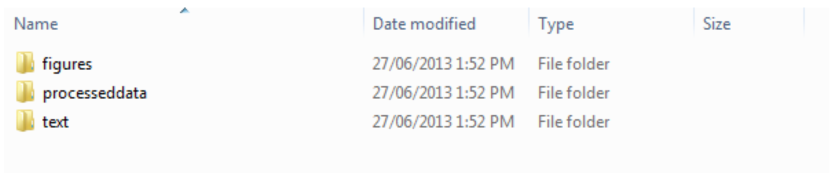
Name	Date modified	Type	Size
awesomefunction.R	27/06/2013 3:16 PM	R File	0 KB
rmDup.R	27/06/2013 12:13 ...	R File	1 KB

Contents of Rfunctions folder, example.

We will use `source()` to load these functions, see further below.

7.3.3 `output`

It is a good idea to send all output from your R scripts to a separate folder. This way, it is very clear what the *outputs* of the analysis are. It may also be useful to have subfolders specifying what type of output it is. Here we decided to split it into figures, processeddata, and text :



Name	Date modified	Type	Size
figures	27/06/2013 1:52 PM	File folder	
processeddata	27/06/2013 1:52 PM	File folder	
text	27/06/2013 1:52 PM	File folder	

Contents of Rfunctions folder, example.

7.4 The R scripts

A few example scripts are described in the following sections. Note that these are just examples, the actual setup will depend on your situation, and your personal preferences. The main point to make here is that it is tremendously useful to separate your code into a number of separate scripts. This makes it easier to maintain your code, and for an outsider to follow the logic of your workflow.

7.4.1 `facesoil_analysis.R`

This is our 'master' script of the project. It calls (i.e., executes) a couple of scripts using `source`. First, it 'sources' the `facesoil\_load.R` script, which loads packages and functions, and reads raw data. Next, we do some analyses (here is a simple example where we calculate daily averages), and call a script that makes the figures (`facesoil_figures.R`).

Note how we direct all output to the `output` folder, by specifying the *relative path*, that is, the path relative to the current working directory.

```r
# Calls the load script.
source("facesoil_load.R")

# Export processed data
write.csv(allTheta, "output/processeddata/facesoil_allTheta.csv",
          row.names=FALSE)

## Aggregate by day

# Make daily data
allTheta$Date <- as.Date(allTheta$DateTime)
allTheta_agg <- summaryBy(. ~ Date + Ringnr, data=allTheta,
                          FUN=mean, keep.names=TRUE)

# Export daily data
write.csv(allTheta_agg, "output/processeddata/facesoil_alltheta_daily.csv",
          row.names=FALSE)

## make figures
source("figures.R")
```

7.4.2 `facesoil_figures.R`

In this example we make the figures in a separate script. If your project is quite small, perhaps this makes little sense. When projects grow in size, though, I have found that collecting the code that makes the figures in a separate script really helps to avoid clutter.

Also, you could have a number of different 'figure' scripts, one for each 'sub-analysis' of your project. These can then be sourced in the master script (here `facesoil\_analysis.R`), for example, to maintain a transparent workflow.

Here is an example script that makes figures only. Note the use of `dev.copy2pdf`, which will produce a PDF and place it in the `output/figures` directory.

```r
# Make a plot of soil water content over time
pdf("./output/figures/facesoil_overtime.pdf")
with(allTheta, plot(DateTime, R30.mean, pch=19, cex=0.2,
                    col=Ringnr))
dev.off()

# More figures go here!
```

The above is OK, but we can do better by writing it into a function, and then calling it to make the PDF. Even better, the PDF making can be done Here we also use `on.exit` to safely close the PDF, see Section @ref(onexit.

Like so,

```r
# A function that defines our plot
# Write functions like these, and collect them in a separate script,
# for example "figure_definitions.R".
soilplot_1 <- function(data){
  with(data, plot(DateTime, R30.mean, pch=19, cex=0.2,
                  col=Ringnr))
}

# A generic function that makes a PDF of a provided function call
# Place this function in a script with a collection of functions.
to.pdf <- function(expr, filename, ...) {

  pdf(filename, ...)
  on.exit(dev.off())

  # A trick to run the provided function call in the global environment
  eval.parent(substitute(expr))
}


# Finally, after having sourced both our function definition,
# and the generic function to.pdf, we can make the PDFs.
to.pdf(
  soilplot_1(allTheta),
  filename = "output/figures/figure1.pdf")
)
```

7.4.3 `facesoil_load`

This script contains all the bits of code that are

- Cleaning the workspace
- Loading homemade functions
- Reading and pre-processing the raw data

It is useful to load all packages in one location in your, which makes it easy to fix problems should they arise (i.e., some packages are not installed, or not available).

```r
# Load packages
source("load_packages.R")

# Source functions (this loads functions but does no actual work)
source("Rfunctions/rmDup.R")

# Make the processed data (this runs a script)
source("facesoil_readdata.R")
```

7.4.4 `load_packages.R`

We find it very convenient to collect all `library` calls throughout your project in a single script. The advantage is that at the top of one of the main analysis scripts, we can simply call `source("load_packages.R")`. If any packages are missing, or something else failed, we know before try to we execute any other code.

It may also be convenient to suppress all messages we see when loading packages. An example script may look like:

```r
suppressPackageStartupMessages({
  library(dplyr)
  library(lubridate)
  library(glue)
})
```

The *disadvantage* of a loading script like this is that we assume that the user has installed all of the required packages. In Rstudio, however, if you open this script - a small message will appear at the top of the script, "Would you like to install missing packages?". If you click OK all packages mentioned in the script that you have not installed will be installed for you.

Another approach uses the `pacman` package, which automatically installs missing packages (see also Section **??**):

```r
if(!require(pacman))install.packages("pacman")
```

```
## Loading required package: pacman
```

```r
pacman::p_load(gplots, geometry, rgl, remotes, svglite)
```

Further reading To learn more about advanced management of R package dependencies, read Section 8

Caution Never include `install.packages` in any of your *scripts* in your project. You do not want to call it more than once, otherwise the execution of the project will be much slower (and require an internet connection).

7.4.5 `facesoil_readdata.R`

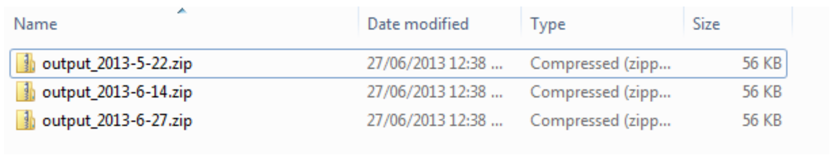
This script produces a dataframe based on the raw CSV files in the `rawdata` folder. The example below just reads a dataset and changes the DateTime variable to a POSIXct class. In this script, I normally also do all the tedious things like deleting missing data, converting dates and times, merging, adding new variables, and so on. The advantage of placing all of this in a separate script is that you keep the boring bits separate from the code that generates results, such as figures, tables, and analyses.

```r
# Read raw data from 'rawdata' subfolder
allTheta <- read.csv("rawdata/FACE_SOIL_theta_2013.csv")

# Convert DateTime
allTheta$DateTime <- ymd_hms(as.character(allTheta$DateTime))

# Add Date
allTheta$Date <- as.Date(allTheta$DateTime)

# Etc.
```

7.5 Archiving the output

In the example workflow we have set up in the previous sections, all items in the output folder will be automatically overwritten every time we run the master script `facesoil_analysis.R`. One simple way to back up your previous results is to create a zipfile of the entire output directory, place it in the `archive` folder, and rename it so it has the date as part of the filename.

After a while, that directory may look like this:



Name	Date modified	Type	Size
output_2013-5-22.zip	27/06/2013 12:38 ...	Compressed (zipp...	56 KB
output_2013-6-14.zip	27/06/2013 12:38 ...	Compressed (zipp...	56 KB
output_2013-6-27.zip	27/06/2013 12:38 ...	Compressed (zipp...	56 KB

Contents of Rfunctions folder, example.

If your `processData` folder is very large, this may not be the optimal solution. Perhaps the `processedData` can be in a separate output folder, for example.

7.5.1 Adding a Date stamp to output files

Another option is to use a slightly different output filename every time, most usefully with the current Date as part of the filename. The following example shows how you can achieve this with the `today` from the `lubridate` package, and the `glue` package:

```r
# For the following to work, load lubridate
# Recall that in your workflow, it is best to load all packages in one place.
library(lubridate)
library(glue)

# Make a filename with the current Date:
fn <- glue("output/figures/FACE_soilfigure1_{today()}.pdf")
fn
```

```
## output/figures/FACE_soilfigure1_2020-03-03.pdf
```

```r
# Also add the current time, make sure to reformat as ':' is not allowed!
fn <- glue("output/figures/FACE_soilfigure1_{format(now(),'%Y-%m-%d_%H-%M')}.pdf")
fn
```

```
## output/figures/FACE_soilfigure1_2020-03-03_22-33.pdf
```

7.6 A logical structure for your scripts

7.6.1 Write functions, not long scripts

If a script becomes too long, write more functions. Writing your own functions is the most important advise if you want to write and maintain robust, complex projects.

As pointed out in the Chapter on Project management (7), save these functions separately, for example "R/functions.R", and `source` them with:

```r
source("R/functions.R")
```

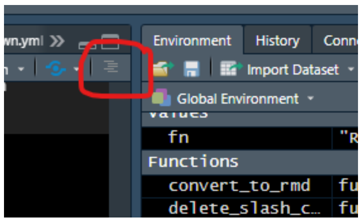
Unfortunately `source` is not vectorized, so to read all R scripts from a subdirectory you can simply do,

```r
for(fn in dir("R", pattern = "[.]R$", full.names = TRUE)){
  source(fn)
}
```

Try this yourself Write the above snippet into a function, which takes the directory to search as an argument.

7.6.2 Divide your script into functional blocks

Dividing your scripts into a few functional blocks can help readability and reliability. With special formatting, you can even improve the *table of contents* (TOC) menu in Rstudio for a script. Run the example below, and then find the TOC button in Rstudio:



Access the (nearly) automatic TOC in Rstudio

In the following example script, note that we load all packages at the beginning of the script, so that when something goes wrong at that stage, we know before executing any of the 'real' code.

Also note the use of `#-----`, this helps to make the TOC as mentioned above.

```r
# An example script
# 2020, Author


#----- Load packages -----
library(dplyr)
library(rvest)
library(stringr)
library(glue)

#----- Custom functions -----
source("R/functions.R")
source("R/database_functions.R")

#----- Configuration -----

# Load configuration (passwords etc., see next Section!)
.conf <- yaml::read_yaml(file = "config.yml")

#----- Database -----

# Make database connection
db_con <- make_database_connection_knmi(.conf)

# Download data
cloud_data <- download_cloud_data(con = db_con)

# Archive the data
fn <- glue("archive/out_{Sys.Date()}.rds")
try(saveRDS(cloud_data, fn), silent = TRUE)

#----- Visualization -----

# Make visuals
make_cloud_maps(data = cloud_data)

#----- Model -----

# Do some advanced modelling
model_run <- run_cloudy_model(data = cloud_data)

# Upload the model results to a remote database
upload_model_db16(model_run, config = .conf)
```

The *fictional* script above is just an example how you can divide a *master script* into logical statements, using functions that perform all the underlying tasks.

One major advantage of the above approach is because functions execute their “inner workings” in a **separate environment**, which means that objects inside a function are not visible either outside the function (like the main script) or in any other script.

That way, executing the script above does not produce any objects in the environment (the memory) other than the ones *returned* by the functions. All the intermediate objects that were executed inside each function have disappeared, freeing memory and avoiding conflicts.

Chapter 8

Mastering package dependencies

Besides having to write dependable code yourself, we also have to worry about dependencies of R packages, that is, code written by other R users.

Further reading First scan through the section on R Packages (1.10) to make sure you grasp all the fundamentals of R packages.

When developing more robust project workflows, we have to worry about several issues to do with the **reproducibility** of our code.

- What package version did we use?
- Does our code still work for newer package versions?
- Do loaded package have conflicts?
- Can we bundle the R packages along with our code?

8.1 What version?

When an external package has changed on CRAN since you last used it, the *version* will have changed. You can inspect the version of a package with,

```
packageVersion("dplyr")
```

```
## [1] '0.8.3'
```

In a script, you can make sure a package is loaded with *at least* some version:

```
packageVersion("dplyr") > "0.8.1"
```

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

Which you can use like,

```
if(packageVersion("dplyr") < "0.8.1"){
  stop("First update the dplyr package!")
}
```

8.2 All packages: loaded or installed

To list all loaded packages, do:

```r
pck <- .packages()
pck
```

```
##  [1] "knitr"      "magrittr"  "tibble"    "stats"     "graphics"
##  [6] "grDevices" "utils"     "datasets"  "methods"   "base"
```

And to list all *installed* packages,

```r
.packages(TRUE)
```

8.3 Avoiding conflicts

To make sure you use a function from the intended package, you can use the `::` operator.

```r
out <- dplyr::select(mtcars, cyl, disp)
```

In this example, we avoid a common problem with `select` and other functions in packages with very general names.

When writing packages and other production-level code, the use of `::` is strongly recommended. In practice in writing dependable scripts, I recommend using `::` where practical, and less for functions that have obviously unique names in the context.

Finally, one way to avoid conflicts is to use as few packages as possible, as described in a later section.

Further reading When loading the `tidyverse` package, a collection of R packages by the same group of developers, we are presented with a list of conflicts with other installed packages. Try it out.

8.4 The search path and conflicts

After you load a package with `library()`, it is added to the *search path*. This is an order of environments where R will look for the value of some object - often an R function.

Use `search` to inspect the current search path:

```r
search()
```

```
##  [1] ".GlobalEnv"        "package:knitr"     "package:magrittr"
##  [4] "package:tibble"    "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"
```

When you type a function in R, like `summary()`, where will R find the definition of that function? In the search path above, it **starts at the first entry**, the global environment (containing the objects you see in the Environment pane in Rstudio). The search continues until a function is found with the name `summary`, and then R stops searching.

If you have **conflicts**, where the name of a function is defined in more than one package, the order in which you loaded the packages will affect from which package the function will actually be used. However, do not use this fact as a way to manage package dependencies, instead read the next section carefully.

To find the actual **location of the package files** on your system, use the following command (this might be helpful to archive all actual package files used in a project).

```
searchpaths()
```

```
##  [1] ".GlobalEnv"
##  [2] "c:/RLIBRARY/knitr"
##  [3] "c:/RLIBRARY/magrittr"
##  [4] "c:/RLIBRARY/tibble"
##  [5] "C:/Program Files/R/R-3.6.2/library/stats"
##  [6] "C:/Program Files/R/R-3.6.2/library/graphics"
##  [7] "C:/Program Files/R/R-3.6.2/library/grDevices"
##  [8] "C:/Program Files/R/R-3.6.2/library/utils"
##  [9] "C:/Program Files/R/R-3.6.2/library/datasets"
## [10] "C:/Program Files/R/R-3.6.2/library/methods"
## [11] "Autoloads"
## [12] "C:/PROGRA~1/R/R-36~1.2/library/base"
```

8.5 Bundling package dependencies with `renv`

If you want to be able to run your project a long time after you created it, one of the packages may have changed - causing an error or unexpected result. Although this is comparatively rare, it is still a possibility you want to avoid.

If we transport the script to another location, how can we be sure that the code will be working? Sometimes, newer versions of packages break old code, and in some cases packages are no longer hosted on CRAN after some time (though this is rare).

Another problem arises because all your R projects normally share the same library of R packages on your machine. That means that if you install a newer version in one project, it might cause code to stop working *in another project*.

The new `renv` package by Rstudio is at the time of writing the best solution to creating a project that is truly portable, since we can transport the packages alongside with the code.

Try this yourself Please read the documentation and installation instructions here: https://rstudio.github.io/renv/.

In practice, we basically just run this command once in a new (or existing) project:

```
renv::init()
```

Caution The above command can take >10min to execute.

All package dependencies are downloaded and kept in a special `renv` subfolder (the original source code for all the packages).

After initializing `renv` in a directory, you can keep using `install.packages` and `library` as usual, the packages are now installed in your local `renv` folder, and in a global **cache** which `renv` manages for you.

If you are using, say package version 1.2, and version 1.3 is installed on your machine from a different project, the `renv` project will still use 1.2, unless we update this package locally ourselves (simply with `install.packages`).

8.6 Minimizing package dependencies

Although a solution like `renv` can help us to make scripts reproducable, in practice this rule of thumb is a valuable lesson:

Use as few external packages in your code as possible

Complicated projects tend to break easily, and often external R packages are to blame.

- Don't use packages when you only use one simple function from it. In that case, copy the function to your own code base, and `source` it in a script.
- Try to use packages that are still in active development, have a significant user- (or fan-) base, such as the `tidyverse` packages, and are popular.

These warnings should not shy you away from exploring the great universe of >15000 R packages that are now publicly available!