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

Remko Duursma

2019-06-10

Contents

Chapter 1

Preface

This book is not an introduction to statistical theory, although sometimes we repeat a little theory to be able to make sense of R output. This book should not be the first statistics book you read. Instead, this book shows you how to implement many useful data analysis routines in R. We really assume that you have already learned statistics elsewhere and are here to learn how to actually do the analyses with R.

We have learned from experience that data practictioners, that means scientists and everyone else who has an interest in learning from data, do not learn statistical analyses by studying the underlying theory. They learn from doing, and from using examples written by others who have learned by doing. For this reason the book in front of you is largely a compendium of examples. We have chosen the examples and the exercises in the hope that they resemble real-world problems.

We have placed many *Try it yourself* boxes throughout the text to give you some ideas on what to try, but try to be creative and modify the examples to see what happens.

Chapter 2

Beginner skills

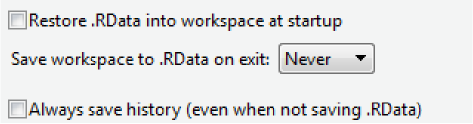
2.1 Installing R and Rstudio

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.

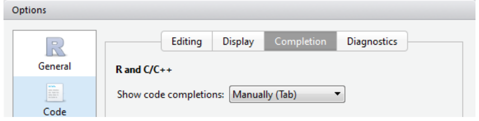


□ Restore .RData into workspace at startup

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.

2.1.1 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 2.8 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.

2.2 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 2.10.2.

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
```

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

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

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

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

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

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

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

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

```
# Exponents
15^2
```

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

```
# 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).

2.3 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:

```
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
```

Table 2.1: A selection of useful built-in functions in R.

Function	What it does	Example
‘length‘	Length of the vector	‘length(nums1)‘
‘rev‘	Reverses the elements of a vector	‘rev(nums1)‘
‘sort‘	Sorts the elements of a vector	‘sort(nums1, decreasing = TRUE)‘
‘order‘	The order of elements in a vector	‘order(nums1)‘
‘head‘	The first few elements of a vector	‘head(nums1, 5)‘
‘max‘	The maximum value	‘max(nums1)‘
‘min‘	The minimum value	‘min(nums1)‘
‘which.max‘	Which element of the vector is the max?	‘which.max(nums1)‘
‘which.min‘	Which element of the vector is the min?	‘which.min(nums1)‘
‘mean‘	Computes the average of the vector	‘mean(nums1)‘
‘median‘	Computers the median	‘median(nums1)‘
‘var‘	Variance	‘var(nums1)‘
‘sd‘	Standard deviation	‘sd(nums1)‘
‘cumsum‘	Cumulative sum (running total)	‘cumsum(nums1)‘
‘diff‘	Successive difference of a vector	‘diff(nums1)‘
‘unique‘	Unique values used in the vector	‘unique(nums1)‘
‘round‘	Rounds numbers to a specified number of decimal points	‘round(nums1, 2)‘

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

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

```r
sd(nums1)
```

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

```r
length(nums1)
```

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

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

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

```r
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.

2.4 Vectorized operations

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. In the file that opens, try typing the examples below (or copy-pasting for the lazy). 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.

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).

2.4.1 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
```

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

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

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

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

```
# 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 2.8 on how to install and load packages).

```
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).

2.5 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.

2.6 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)
```

2.7 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.

2.8 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.

2.8.1 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 don't 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 2.8.5). To use the package in your current R session, type:

```
library(gplots)
```

This loads the `gplots` package from your library 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)
```

2.8.2 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:

```
options(repos = c(CRAN = "https://cran.rstudio.com/"))
```

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

2.8.3 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:

```r
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.

2.8.4 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:

```r
.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.

2.8.5 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:

```r
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`.

2.8.6 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()
```

2.8.7 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:

```
# 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.

2.8.8 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:

```
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.

2.9 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:

```
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?".

2.10 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 4, 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.

2.10.1 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 4.3.

2.10.2 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.

2.10.2.1 Running R 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)"
```

2.10.2.2 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.

Chapter 3

Data skills

3.1 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 2.1.1)
- `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)

3.2 Generating data

3.2.1 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"
```

3.2.2 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] 0.9155525 0.9808714 0.6954055 0.7959569 0.8144647 0.8374154 0.7861935
##  [8] 0.5242173 0.5390675 0.7666385
```

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

```
## [1] 332.6357 648.0441 642.2567 985.0922 177.7964
```

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]  4 10 11  2 10  3 12  2 11 10  1  1  7  1 12  5  3 12 11  8
```

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

3.3 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 7.

3.3.1 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 2.6) 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)
```

3.3.2 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.

3.3.3 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=";")
```

3.3.4 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
```

3.3.5 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  500 observations and 5 variables    Maximum # NAs:0
##
##
##           Storage
## postId    integer
## id        integer
## name    character
## email   character
## body    character
```

3.3.6 (No)SQL databases

We show examples of how to obtain data from SQL and NoSQL databases in Chapter 7.

3.3.7 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'`).

3.3.8 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.**

3.3.9 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 7.

3.4 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 3.4.1), 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.

3.4.1 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.

3.4.2 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 3.7.3).

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
```

3.4.3 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 3.5.1).

```
# 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)
```

3.5 Extracting data

3.5.1 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 3.2.1.

```
# 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]  8.1 21.2  2.5 13.8  9.8
```

```
# 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.

3.5.1.1 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.

3.5.2 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 3.5.1), 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] 376.308256 169.163842  33.713580  54.267994 253.308265   6.551044
##  [7] 349.057010  15.476022 234.368784  91.538428
```

```
# 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
```

```
# 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
```

```
# (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.

3.5.3 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.

```
# 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
```

```
# (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
```

3.5.4 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 3.6). 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)
```

3.6 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 2.1, we suggest you switch that behaviour off.

3.7 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:

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

Also, R has a very useful built-in data type to represent **missing values**. This is represented by `NA` (Not Available) (see Section 3.7.3).

We will show how to convert between data types at the end of this chapter (Section 3.7.6).

3.7.1 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.

```r
# 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 ...
```

```r
# 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 3.8.4.

3.7.1.1 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
```

3.7.1.2 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.

3.7.1.3 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()`.

3.7.2 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

# 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:

```
# 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
```

```
# 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.

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

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

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

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

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

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

3.7.3 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:

```
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:

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

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

```
# 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,

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

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

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

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

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

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

3.7.3.1 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,

```
# 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,

```
# 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).

3.7.3.2 Not A Number

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

```r
# 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:

```r
1000/0
```

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

It produces 'Infinity' instead.

3.7.3.3 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`.

```r
# (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
```

```r
# 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(.))
```

3.7.3.4 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
```

3.7.4 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
```

3.7.4.1 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.

3.7.4.2 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/.

3.7.4.3 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)`).

```
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"
```

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

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

```
# 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"
```

3.7.4.4 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 3.7.4.1, we can use `paste` to achieve this, but there's a better way:

```
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`.

3.7.5 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.

3.7.5.1 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"
```

```
#-- 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:

```
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.

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

```
## [1] "2019-06-10"
```

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

```
## Time difference of 15724 days
```

3.7.5.2 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.

```
# 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:

```
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:

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

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

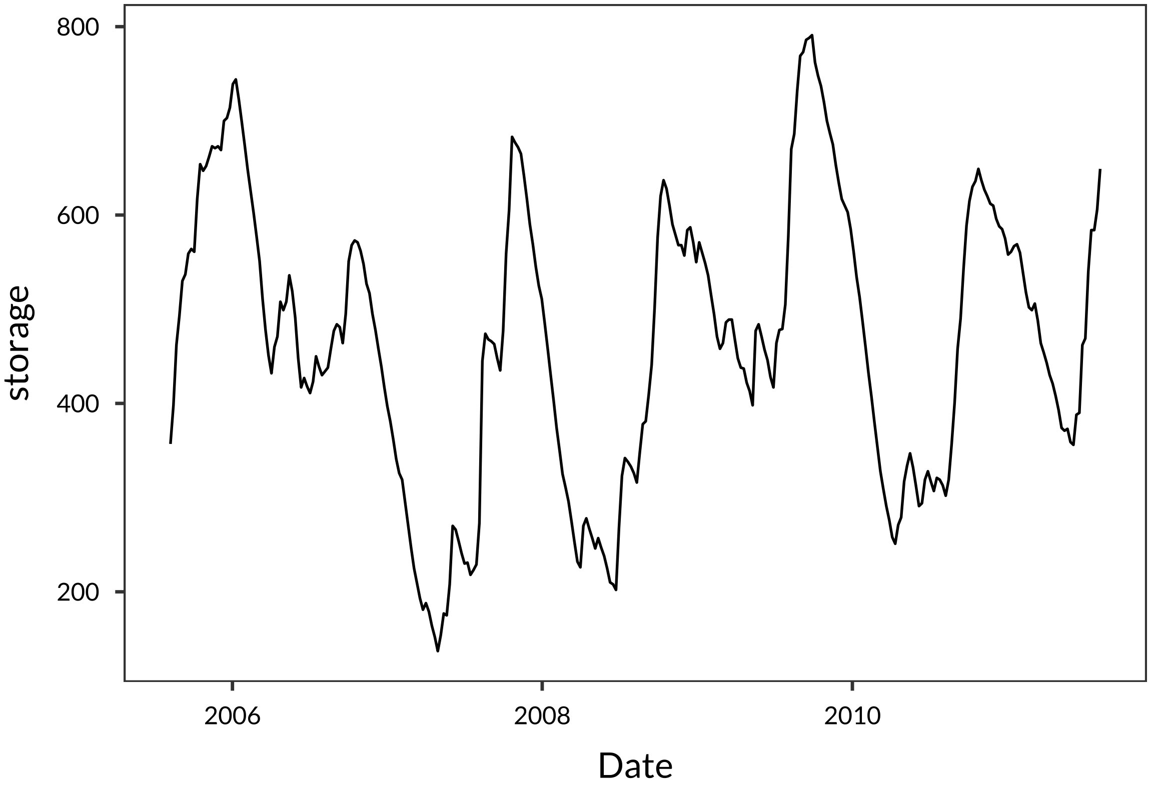


Figure 3.1: A simple plot of the hydro data.

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

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

```
#... 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. 3.1. Note how the `X` axis is automatically formatted to display the date in a (fairly) pretty way.

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

3.7.5.3 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] "2019-06-10 13:04:38 CEST"
```

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?

3.7.5.4 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. 3.2.

```
# 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
```

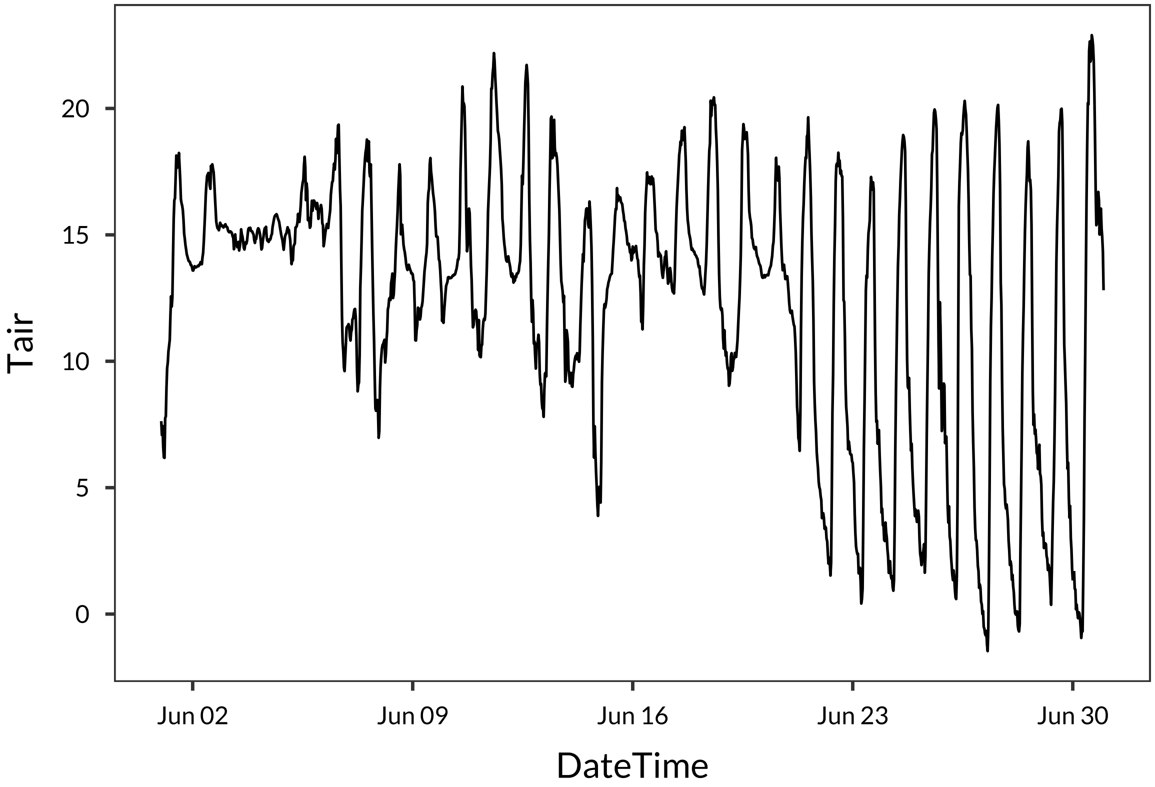


Figure 3.2: Air temperature for June at the HFE

```
# 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
```

```
# 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()
```

```
# 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"))
```

3.7.5.5 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 3.2.1).

```
# 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"
```

3.7.6 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
```

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

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

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

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

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

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

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

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

```
# 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:

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

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

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

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

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

```
# 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"
```

```
as.character(myfac)

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

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

## Warning: NAs introduced by coercion

## [1] 1001   NA

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

## [1] 280 400 650

# Convert to Date
as.Date(mydatetime)

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

# 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.

3.8 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 3.4.

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)
```

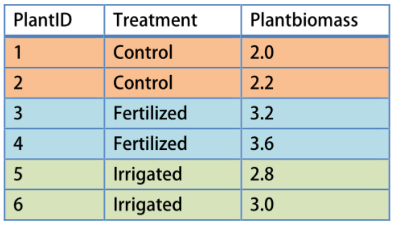
```
##
## Data frame:pupae 84 observations and 5 variables    Maximum # NAs:6
##
##
##               Levels Storage NAs
## T_treatment        2 integer   0
## CO2_treatment      2 integer   0
## Gender               integer   6
## PupalWeight           double   0
## Frass                 double   1
##
## +-------------+----------------+
## |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).

3.8.1 Making summary tables

3.8.1.1 Summarizing vectors with `tapply`

If we have the following dataset called `plantdat`,

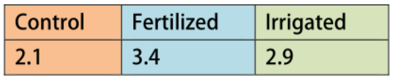


PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and execute the command

```
with(plantdat, tapply(Plantbiomass, Treatment, mean))
```

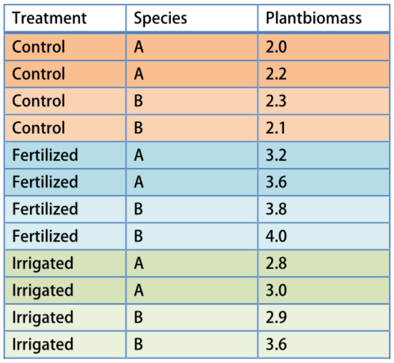
we get the result



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`,

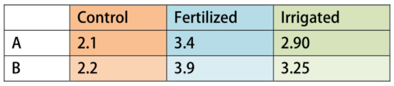


Treatment	Species	Plantbiomass
Control	A	2.0
Control	A	2.2
Control	B	2.3
Control	B	2.1
Fertilized	A	3.2
Fertilized	A	3.6
Fertilized	B	3.8
Fertilized	B	4.0
Irrigated	A	2.8
Irrigated	A	3.0
Irrigated	B	2.9
Irrigated	B	3.6

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).

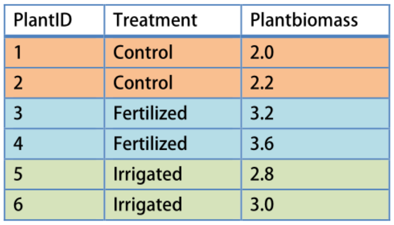
The limitations of tapply are that you can only summarize one variable at a time, and that the result is not a dataframe.

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.
# 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")
```

3.8.1.2 Quick summary tables with summaryBy

If we have the following dataset called plantdat,

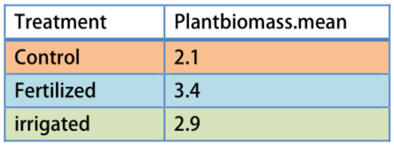


PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and execute the command

```
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,

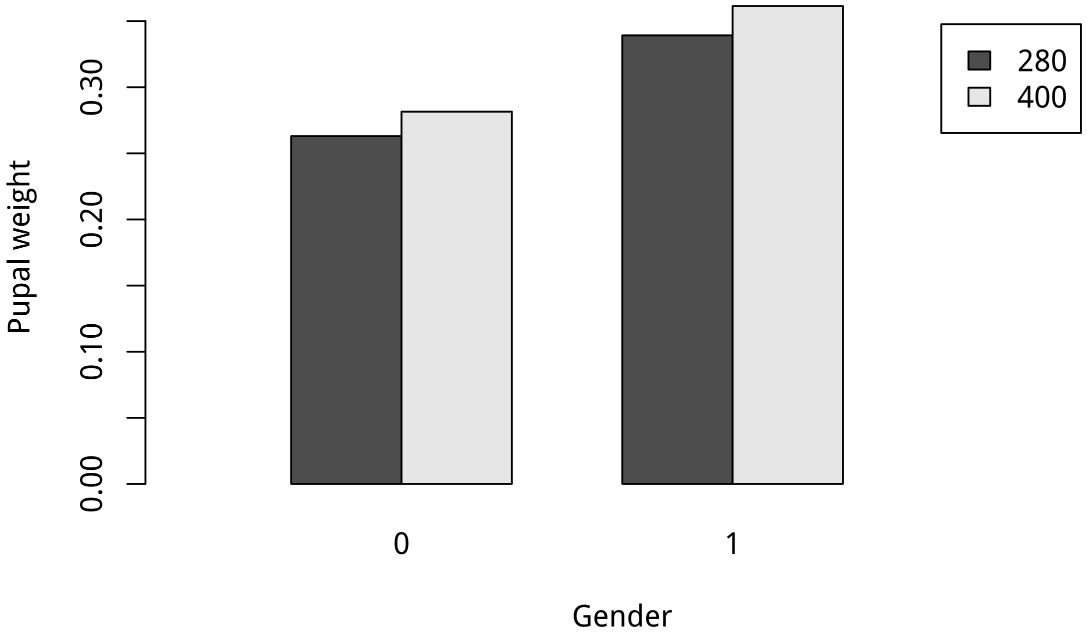
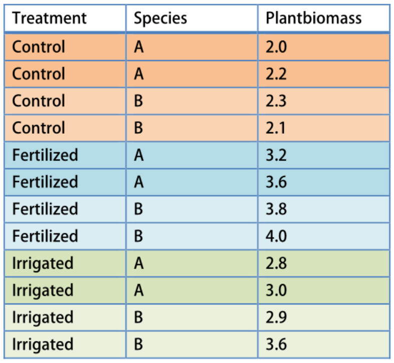


Figure 3.3: A grouped barplot of average pupal weight by CO2 and Gender for the pupae dataset. This is easily achieved via the use of tapply.

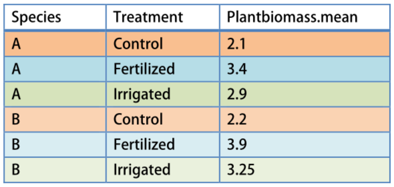


Treatment	Species	Plantbiomass
Control	A	2.0
Control	A	2.2
Control	B	2.3
Control	B	2.1
Fertilized	A	3.2
Fertilized	A	3.6
Fertilized	B	3.8
Fertilized	B	4.0
Irrigated	A	2.8
Irrigated	A	3.0
Irrigated	B	2.9
Irrigated	B	3.6

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).
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
```

3.8.1.3 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

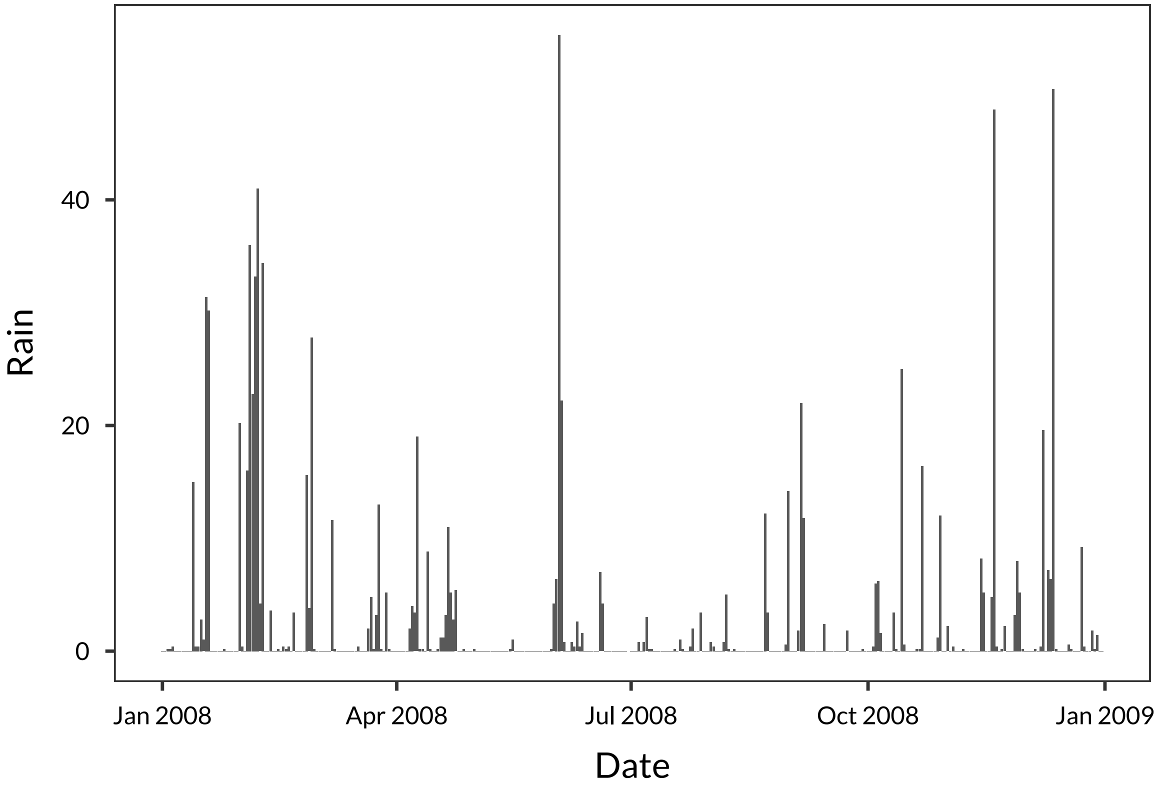


Figure 3.4: Daily rainfall at the HFE in 2008

data as daily averages (for temperature) and daily sums (for precipitation).

The following code produces a daily weather dataset, and Fig. 3.4.

```r
# 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")
```

3.8.1.4 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
  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
```

3.8.2 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
```

3.8.3 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 3.8.1.1).

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 treeSizeClass MaxHeight
## 1    PSME    54.61  27.04 338.485622  410.24638         large      33.3
## 2    PSME    34.80  27.42 122.157864   83.65030        medium      33.3
## 3    PSME    24.89  21.23   3.958274    3.51270         small      33.3
## 4    PSME    28.70  24.96  86.350653   73.13027        medium      33.3
## 5    PSME    34.80  29.99  63.350906   62.39044        medium      33.3
## 6    PSME    37.85  28.07  61.372765   53.86594        medium      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.

3.8.4 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. 3.5.

```
# 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)) %>%
  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`

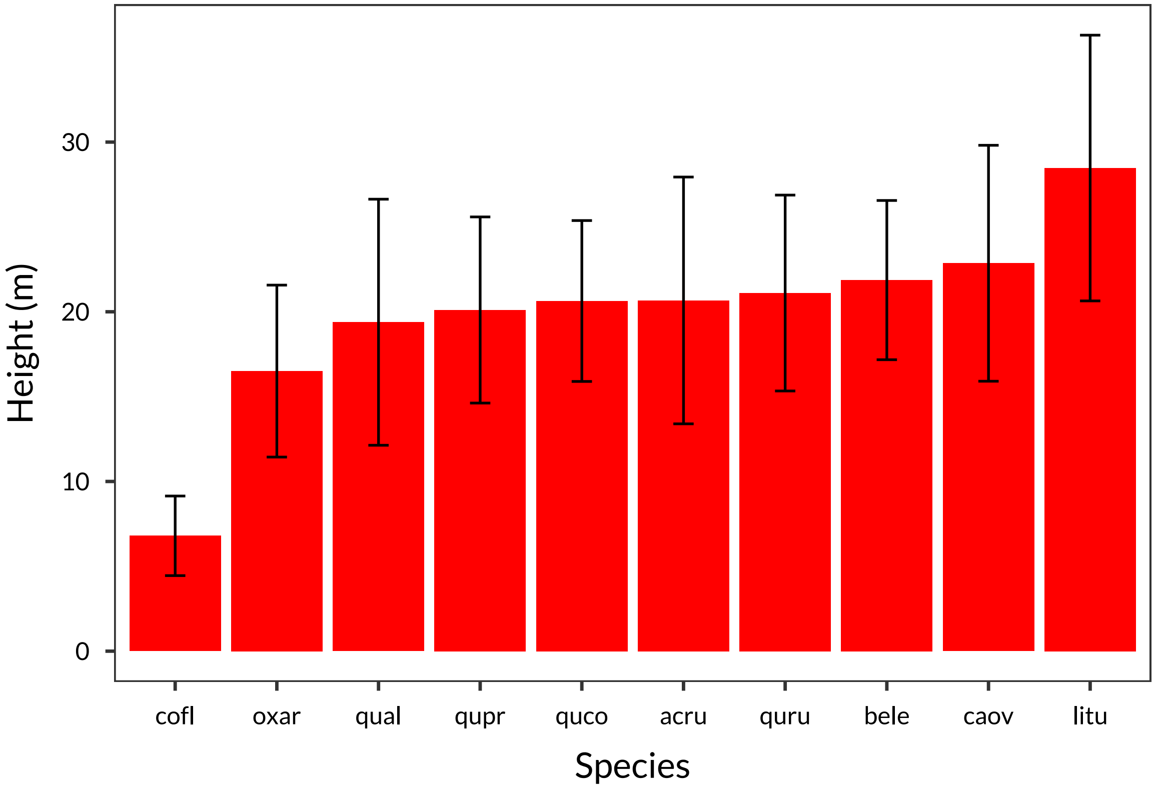


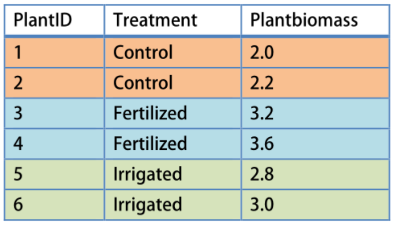
Figure 3.5: An ordered barplot for the coweeta tree data (error bars are 1 SD).

<- factor(coweeta$species, levels=rev(levels(coweeta$species))) Here we used rev to reverse the levels.

3.9 Combining dataframes

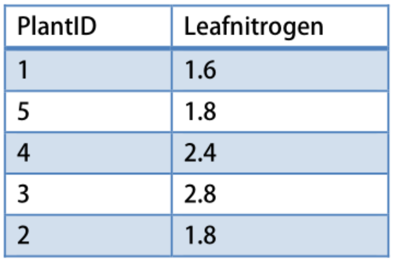
3.9.1 Merging dataframes

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):

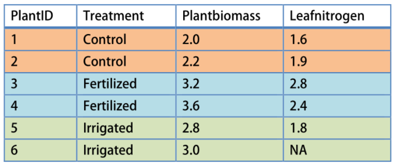


PlantID	Leafnitrogen
1	1.6
5	1.8
4	2.4
3	2.8
2	1.8

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!
```

3.9.2 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.

```r
# 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
```

```r
head(cereal2, 3)
```

```
##     cerealbrand vitamins
## 1    Product_19      100
## 2 Count_Chocula       25
## 3    Wheat_Chex       25
```

```r
head(cereal3, 3)
```

```
##              cerealname sugars
## 1       Frosted_Flakes     11
## 2           Product_19      3
## 3 Mueslix_Crispy_Blend     13
```

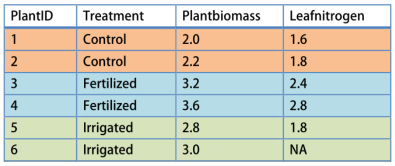
```r
# The name of the variable that ties the datasets together,
# the 'cereal name' differs between the datasets, as do the number of rows.
# We can use merge() three times, but the data are easiest to merge with dplyr:
library(dplyr)
```

```
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.
```

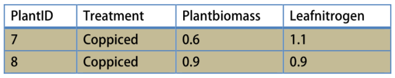
3.9.3 Row-binding dataframes

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),

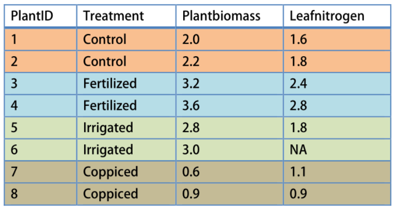


PlantID	Treatment	Plantbiomass	Leafnitrogen
7	Coppiced	0.6	1.1
8	Coppiced	0.9	0.9

and execute the command

```
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
```

```
mydata2
```

```
##   var1 var2
## 1    2    6
## 2    3    7
## 3    4    8
```

```
# '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
```

```
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),

```
# 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.

3.10 Reshaping data

3.10.1 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
```

3.10.2 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
```

3.11 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.

3.11.1 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. 3.6.

```
data(hfeifbytree)

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
```

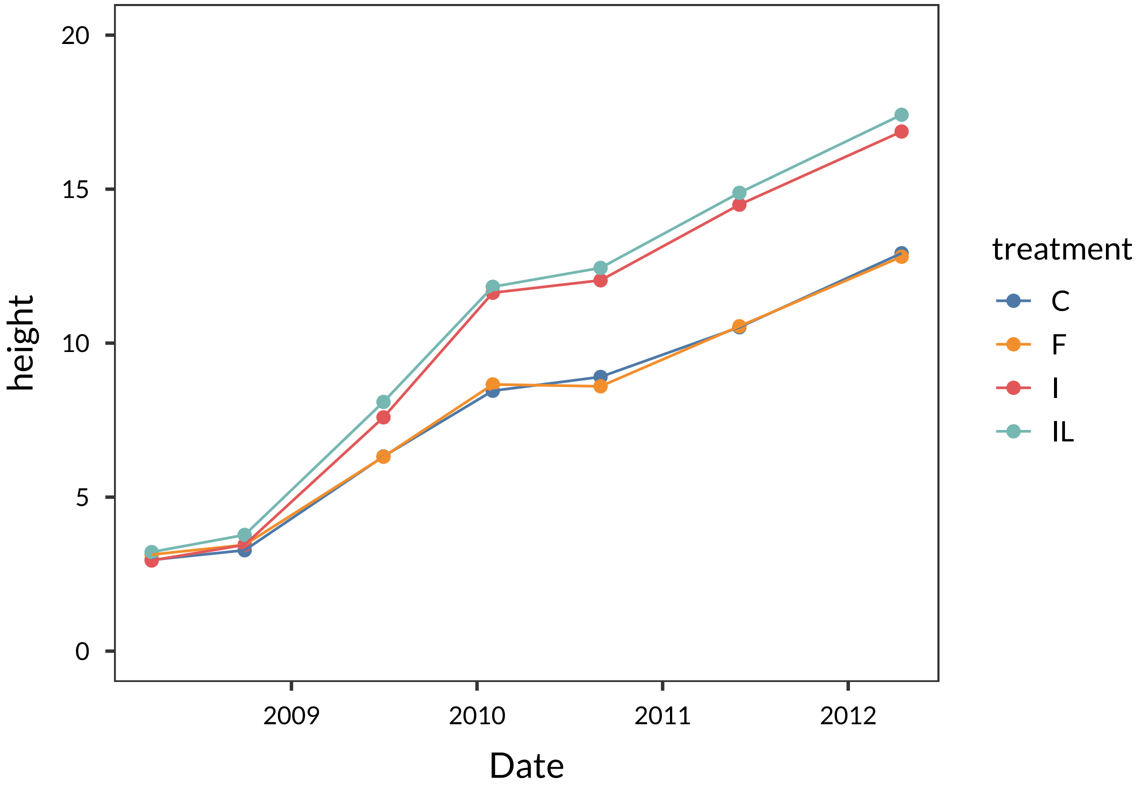


Figure 3.6: Average tree height by treatment over time, for the hfeifbytree data

```r
# 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))
```

3.11.2 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

3.11.3 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. **??**.

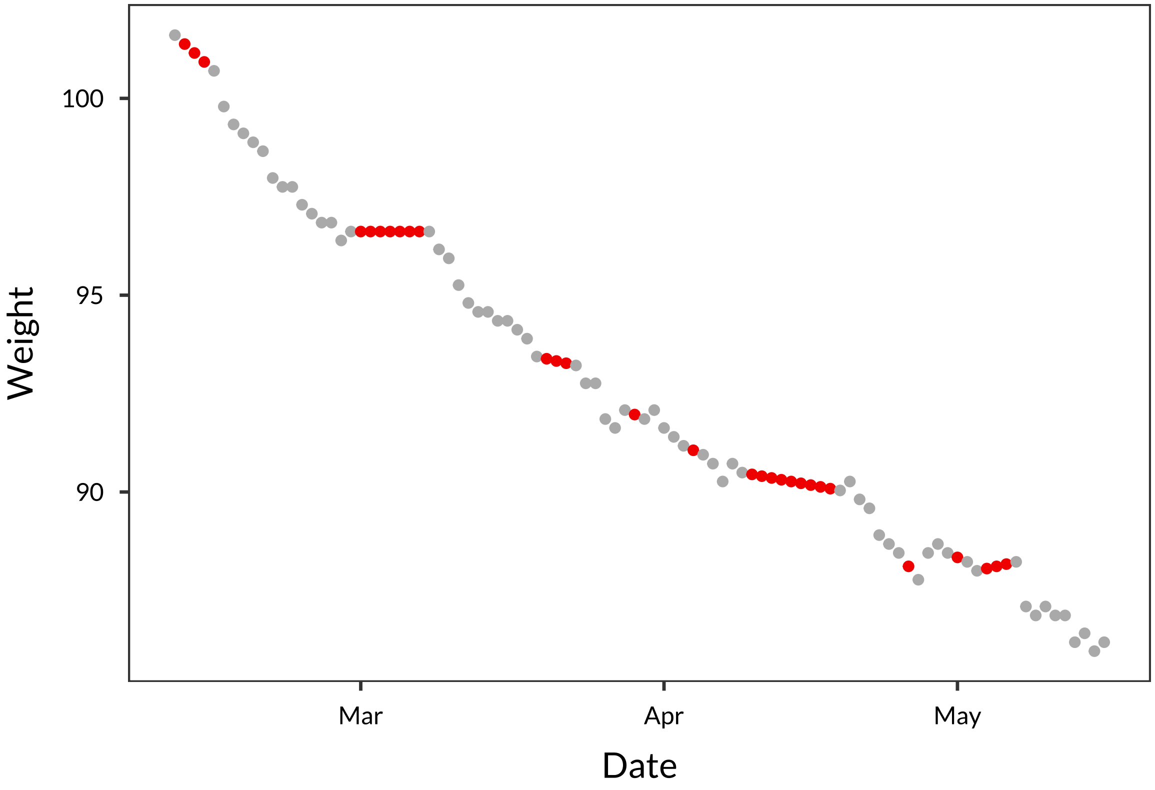


Figure 3.7: The weightloss data, with linearly interpolated missing values (red).

```r
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"))
```

Chapter 4

Visualizing data and making reports

4.1 Introduction

Data visualization is an integral part of any data analytical project. We use data viz to spot patterns in data, check model assumptions, communicate findings, and to convince the audience that our analysis and conclusions make sense. Data viz is not something you only do at the end, but we switch cycle between analysis and visualization. It is therefore not only an important skill to make beautiful visualizations in the final communication of your results, but also to be able to make quick & dirty graphs at any step in the analysis.

We do not only communicate with visualizations, but like to combine text, tables, figures, images, and well-formatted text in reports. We now have a powerful tool available in R, the `rmarkdown` document - a flexible toolkit to generate documents containing all results from R (with or without the code). These reports ensure that the results you see are made from the code in the document, we never have to ask: which version of my code produced those results? We already used `rmarkdown` in Chapter 2, but go a few steps further here and provide useful tips & tricks.

This chapter is different from the rest: we do not provide a complete beginners guide to either data visualization with `ggplot2`, or reporting with `rmarkdown`. We point to online resources for getting started, including comprehensive manuals, and galleries of examples. Instead, we focus on pointing out typical difficulties when getting started with these tools.

For data visualization, also be aware that the entire book in front of you includes many figures produced with `ggplot2` or base graphics. As always, learning from example is the best way.

4.2 Visualizing data

4.2.1 Base graphics or `ggplot2`?

R comes with two graphics systems, that cannot be used together. The `base` graphics system has been part of R from the start, and continues to be used today. The second system is the `grid` system, and the most popular package that uses this system is `ggplot2`, a package in development for over 10 years that now sees very wide application, and numerous add-ons and extensions.

Personally, I have used base graphics since 2001, and resisted switching to `ggplot2` for a very long time. I now use `ggplot2` for visualizing data, but continue to use base graphics as well, for a few reasons - especially because many add-on packaged include some quick plotting function that returns a base plot.

You can read around the web for many opinionated accounts on why we should use `ggplot2` or the other (for example here).

A few points to consider:

- Use `ggplot2` if you are a newcomer, because it is easier to make more complex graphs. For example, it is much easier to make multi-panel, multi-line, multi-everything figures than with base graphics. Notable examples include multi-panel figures with grouped bar plots, adding regression and smoothed lines *with* confidence bands to plots, overlapping area plots, histograms and ribbons. All of these plots are possible with base, but usually require more code and more programming.
- It is generally easier to achieve consistent figure formatting with `ggplot2`, through the use of themes.
- Many packages include some built-in plotting function, more often than not these use base graphics. It is usually much easier to use those functions directly than to implement the figure from scratch in `ggplot2`. A basic understanding of base graphics is therefore useful in practice.
- Sometimes, base graphics are easier, especially when you are experimenting and want to make quick and dirty plots. To plot a mathematical function, we can do `curve(x^3, from=-1, to=1)` - this is not so easy in `ggplot2`.
- There is no support for 3D graphics in `ggplot2`, base has a number of possibilities. That said, you probably do not want to make many 3D plots.
- Use base graphics when you are making a simple plot, and wish to avoid *dependencies*. Keeping dependencies to a minimum will make it more likely your code works in the future, and on someone else's machine. The `ggplot2` package is rather stable these days, but has changed dramatically in the past - often breaking old code.

For the reasons mentioned above, in this book we mostly use ggplot2, except when base graphics are a lot easier (that is, use less code).

4.2.2 Base graphics

In base graphics, we can choose from the following main plotting functions (this list is not complete, especially if you consider add-on packages):

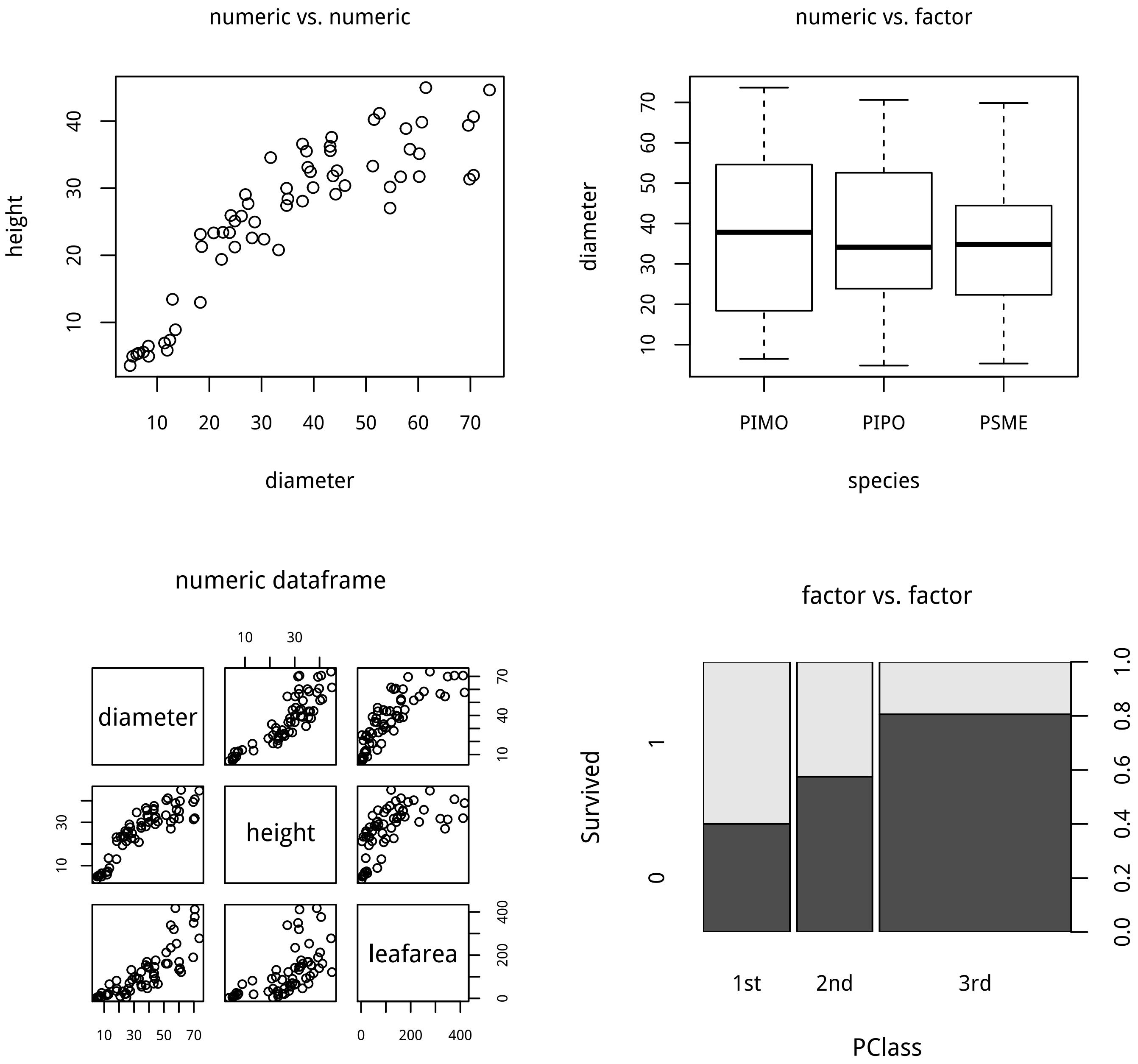
Function	Key functions
`plot`	Scatter plots, line plots, various others.
`barplot`	Bar plot (including stacked and grouped bar plots)
`hist`	Histograms and (relative) frequency diagrams
`curve`	Curves of mathematical expressions
`pie`	Pie charts (for less scientific uses)
`boxplot`	Box-and-whisker plots
`symbols`	Like scatter plot, but symbols are sized by another variable

Here it is useful to know that `plot` is a so-called *generic function*, which has *methods* for many special objects in R. For example, if we fit a linear regression:

```
lmfit <- lm(disp ~ hp, data = mtcars)
```

We can use the very brief command `plot(lmfit)` to produce four diagnostic plots. This works because a function called `plot.lm` is actually used when you send an `lm` object to the `plot` function (thus we can look at useful options in this case in the help page `?plot.lm`).

Similarly, `plot` will make different plots based on what type of data you use. In Fig. **??** four examples are shown; each of these is the result from a call to `plot()`, but with different data types.



The way that plot gives different results depending on data types is both a good thing (for basic plots, you need very little code), and a bad thing because reading the code often does not tell you what sort of plot will be produced, and where to read the documentation (?plot, ?plot.default, ?plot.factor?).

4.2.3 ggplot2

The ggplot2 package is an incredibly rich and powerful toolkit to construct plots. It is far beyond the scope of this book to provide a complete overview of the package, but luckily lots of free online resources are available (see Section 4.2.3.5).

We will show the functionality of ggplot2 by providing two examples, one scatterplot, and one barplot. Since we use ggplot2 throughout this book, you can find many more examples. The following three examples were chosen because they highlight a few key options.

4.2.3.1 Preparing the data for plotting

To use ggplot2, our data have to always be in *long format*, where each row represents an observation, and each column in the dataframe represents a type of measurement. Most of the datasets included in the lgrdata package are already in long format. One exception is the dutchelection dataset:

```r
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
```

We cannot really use ggplot2 when the data are formatted like this. Instead we want to have a dataframe with columns 'Date', 'party', and 'vote_percent', so that we can label the graph by political party. As we will see in Section 3.10, quite magically we can use melt from reshape2 :

```r
library(reshape2)
dutchlong <- melt(dutchelection, variable.name = "party", value.name="vote_percent")
head(dutchlong, 3)
```

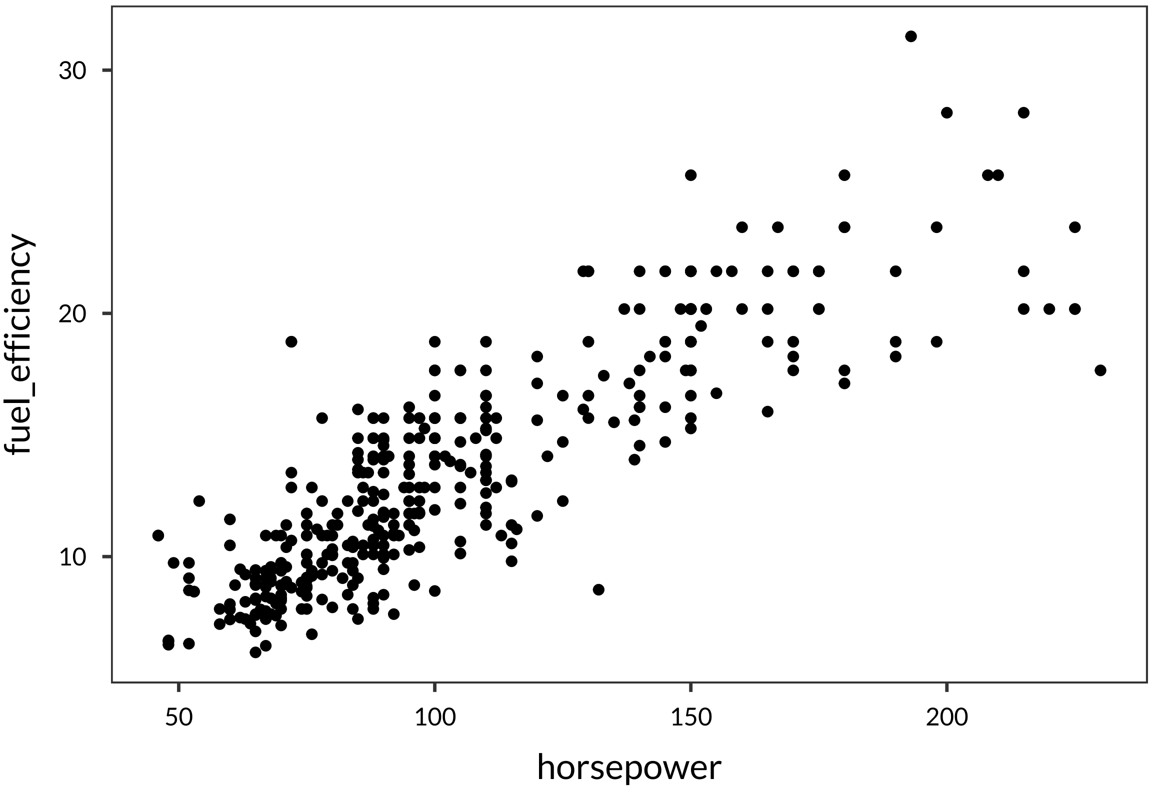
```
##         Date party vote_percent
## 1 2012-03-22   VVD         22.1
## 2 2012-04-05   VVD         23.6
## 3 2012-04-19   VVD         24.0
```

4.2.3.2 Scatterplot example

Here we show two examples of a scatter plot. In practice you start with the simplest plot you can think of, then gradually add 'layers' and other elements. Here, the quickest way to make a scatter plot is like this,

```r
# Minimal
data(automobiles)

ggplot(automobiles, aes(x = horsepower, y = fuel_efficiency)) +
  geom_point()
```

```
## Warning: Removed 6 rows containing missing values (geom_point).
```

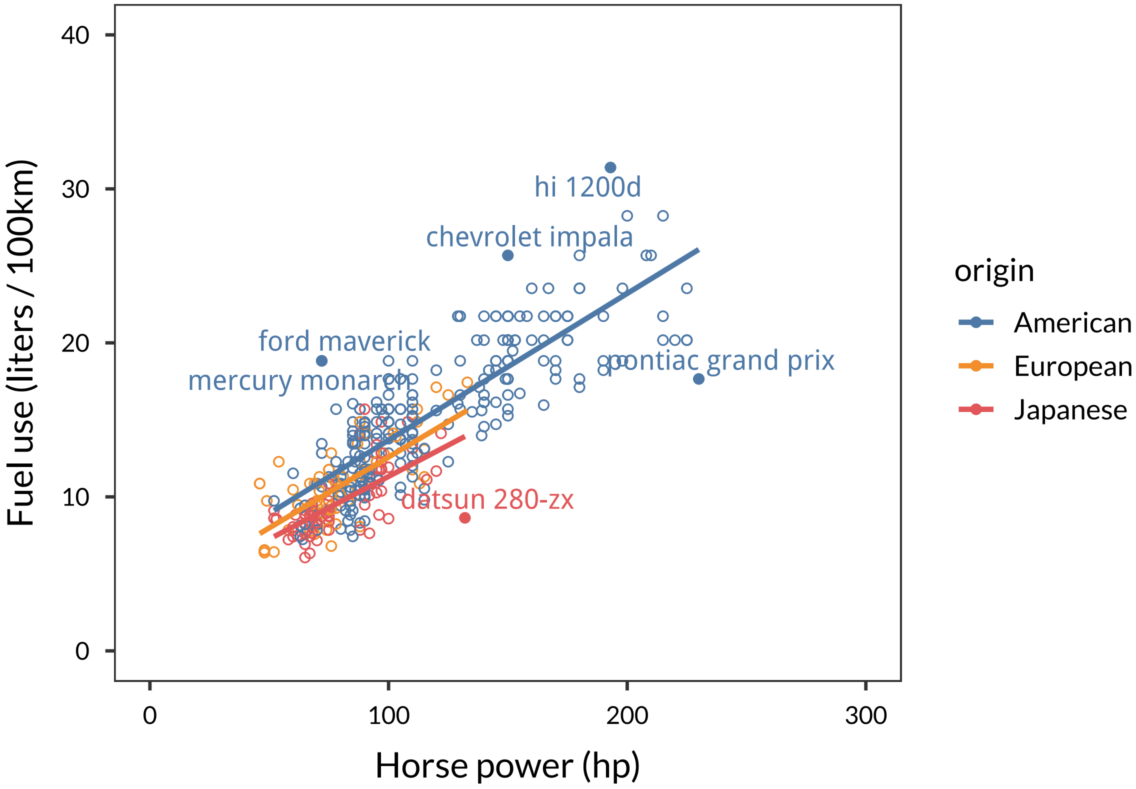


Next, we want to extend this example by including various other elements:

- Plotting symbols colored by Region
- A linear regression line, one for each of the three Regions
- Text labels identifying outliers
- A legend
- Better axis labels

Study the example below how each of these was included:

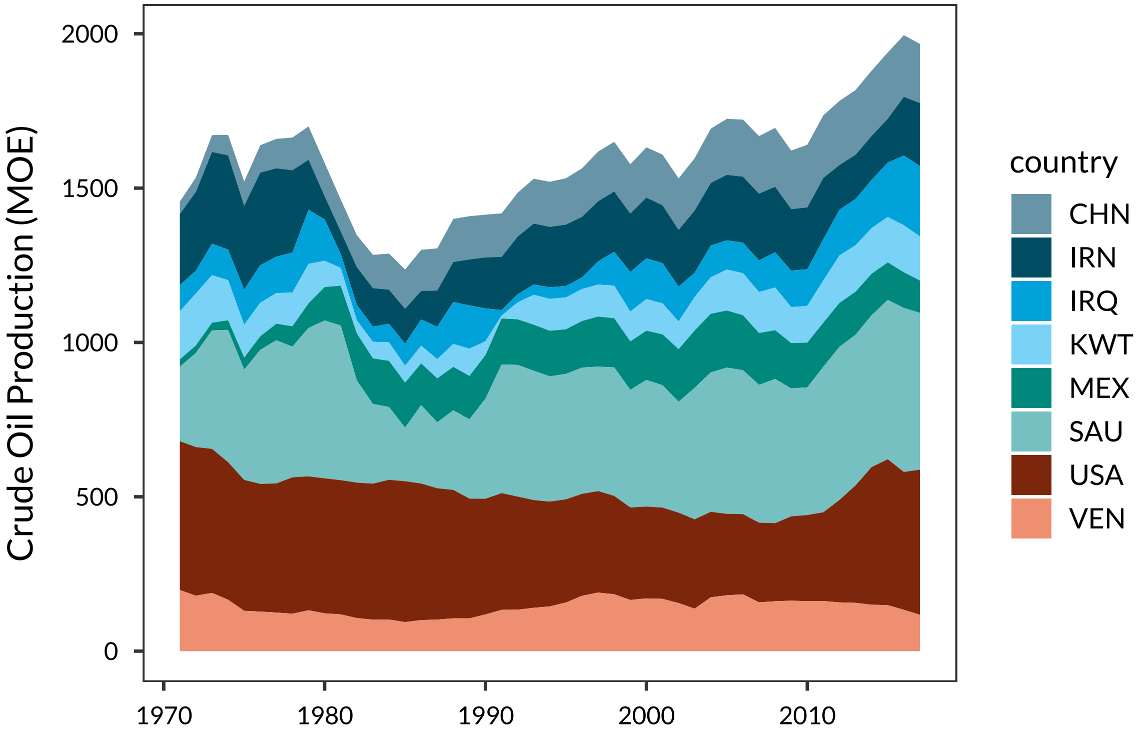
```
# Extended
data(automobiles)

library(ggrepel)
library(dplyr)

automobiles2 <- mutate(automobiles,
       resid = residuals(lm(fuel_efficiency ~ horsepower,
                            na.action=na.exclude)))

ggplot(automobiles2, aes(x = horsepower, y = fuel_efficiency,
                        col = origin,
                        label = car_name)) +
  geom_point(pch = ifelse(abs(automobiles2$resid) > 6, 19, 1)) +
  geom_text_repel(data = subset(automobiles2, abs(resid) > 6),
                show.legend=FALSE) +
  lims(x = c(0,300), y = c(0,40)) +
  stat_smooth(method = "lm", se = FALSE) +
  scale_colour_tableau() +
  labs(x = "Horse power (hp)", y = "Fuel use (liters / 100km)")
```



4.2.3.3 Area plot example

A 'stacked area chart' can be quickly implemented with the `geom_area` function, like so:

```
data(oil)
ggplot(oil, aes(x=year, y=production/1000, fill=country)) +
  geom_area() +
  labs(x = "", y = "Crude Oil Production (MOE)") +
  scale_fill_economist()
```



4.2.3.4 Barplot example

More complex barplots is one area where ggplot2 is much, much easier to use than base graphics. If you want to represent count data as bars - a common application of barplots - ggplot2 does the counting for you automatically, saving a step where such tables of counts have to be constructed.

If instead you want to make a barplot of data *as they are* (that is, don't make tables first), use the setting `geom_bar(stat = "identity")`, instead of `stat = "count"`, as in the example below.

In the following example we use the `titanic` data, available in the `lgrdata` package.

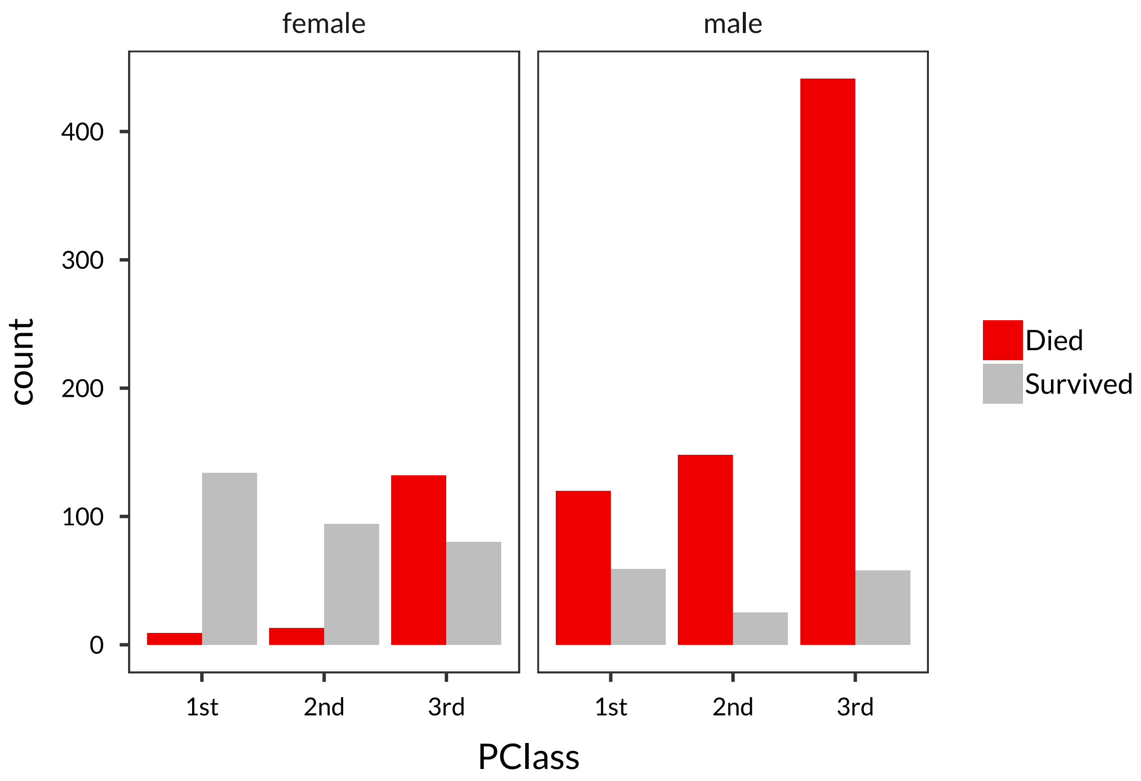
```
library(lgrdata)
data(titanic)

# Important to convert factor variables to factors!
# 'Survived' is coded as 0,1 - but this is really a factor.
titanic$Survived <- as.factor(titanic$Survived)

library(ggplot2)
library(ggthemes)
```

The following code is the minimal code needed to make a barplot with two panels (one for female passengers, one for male), counting passengers that have survived or died, for each of the three passenger classes.

```
# Result not shown - see next example
ggplot(titanic, aes(x=PClass, fill=Survived)) +
  geom_bar(stat="count", position=position_dodge()) +
  facet_wrap(~Sex)
```

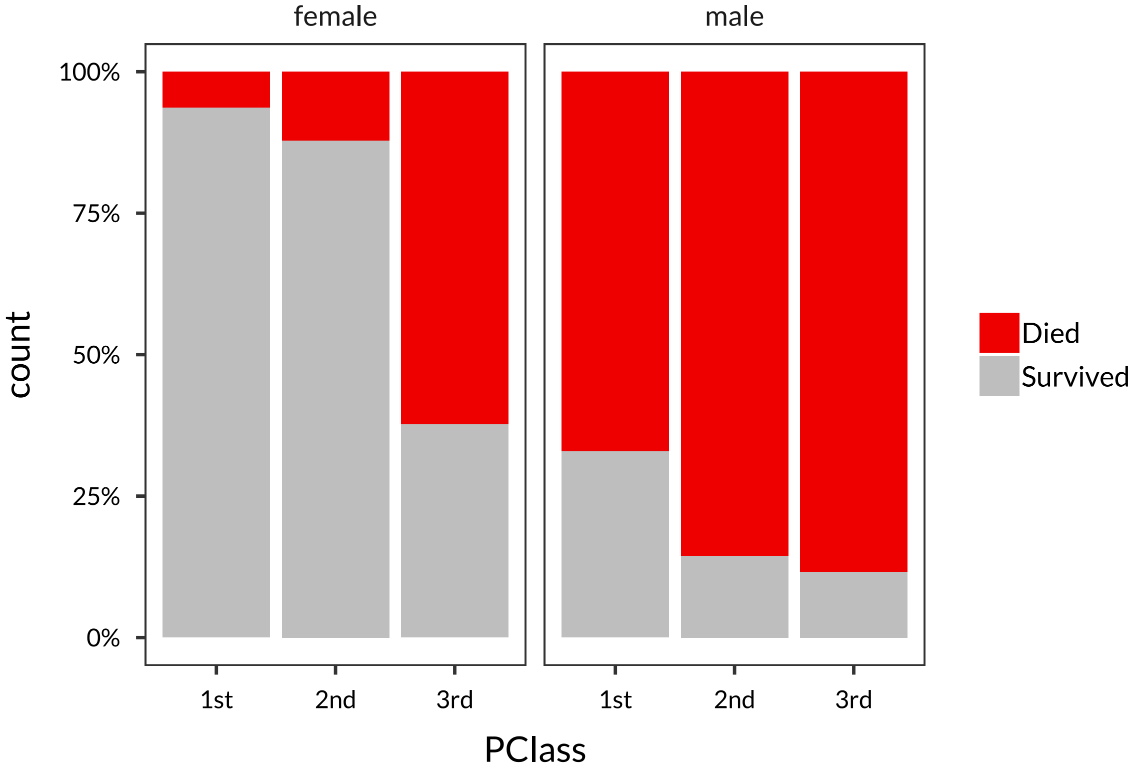
You can run the code above yourself to see what is produced. It does not look great, so instead we re-format the legend, the colors, etc., and end up with Fig. **??**. For both of these figures, *make sure that `Survived` is a factor variable*, otherwise the plot won't look right.

```
ggplot(titanic, aes(x=PClass, fill=Survived)) +        # bars by passenger class,
                                                       # split by 'Survived'
  scale_fill_manual(labels=c("Died", "Survived"),      # labels for legend
                    values=c("red2","grey")) +         # colors for bars (and legend)
  geom_bar(stat="count", position=position_dodge()) +  # count data, bars side by side
  facet_wrap(~Sex) +                                   # panels by Sex
  theme(legend.title=element_blank())                  # no legend title
```



Same figure but stacked bars, expressed as percent of total (Fig. **??**).

```
ggplot(titanic, aes(x=PClass, fill=as.factor(Survived))) +
  scale_fill_manual(labels=c("Died", "Survived"),      # labels for legend
                    values=c("red2","grey")) +         # colors for bars (and legend)
  geom_bar(stat="count", position=position_fill()) +   # count the data, stack the bars
  scale_y_continuous(labels=scales::percent) +         # express y-axis as %
  facet_wrap(~Sex) +                                   # two panels for each sex
  theme(legend.title=element_blank())                  # no legend title
```



4.2.3.5 Resources

The web is bursting with great `ggplot2` tutorials and collections of examples. If you are stuck on some aspect, simply type your question into Google, and you will usually end up at a useful Stackoverflow post.

A selection of useful resources is:

- My favorite resources with lots of small examples: http://www.cookbook-r.com/Graphs/
- A very practical gallery guide at http://www.sthda.com/english/wiki/ggplot2-essentials
- Official page with (extensive) documentation: https://ggplot2.tidyverse.org/
- A more gentle introduction to `ggplot2` at https://r4ds.had.co.nz/data-visualisation.html
- An attractive gallery of useful plots, with all code at http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Co html

The `ggplot2` package has many extensions, packages that offer additional options, themes, plotting layouts, etc. In this book, we only use a couple of add-on packages for reasons of brevity. These include:

- `ggthemes` : an excellent collection of themes and color palettes
- `scales` : useful functions for formatting axes, adjusting colors
- `gridExtra` : to arrange multiple, separate plots, on one page

4.3 Making reports

Script files are good for simple analyses, and they are great for storing small amounts of code that you would like to use in lots of different projects. They are also the right solution if you want to produce R scripts that are executed as part of a larger project, for example from the command line, or executed on a schedule (e.g. daily data quality checking). In Section 2.10.2 we give a few more tips.

But, scripts are not the best way to share your results with others. Instead, we strongly suggest you use R markdown - a document format that includes both text with markup, and *code chunks*, bits of R code that is executed, and the results placed in the document.

Markdown is a simple set of rules for formatting text files so that they are both human-readable and processable by software, and is used not just by `rmarkdown` but in many applications (like Github and Stackoverflow). R Markdown simply adds R, by allowing code chunks with various options.

4.3.1 Getting started

RStudio offers a handy editor for markdown files, and starts with an example file that shows many of the features. Start a new markdown file by choosing File > New File > R Markdown.... For now, leave the default settings. The new R markdown document (which is just a text file with the extension `.Rmd`) already contains some example code. Run this example by clicking the button just above the markdown document (Knit HTML). The example code will generate a new document (an HTML page), which opens in your browser You can see that the output contains text, R code, and even a plot.

Try this yourself Try ‘knitting’ the example rmarkdown file (the one you see when you open your first `rmarkdown` file). First install the `knitr` and `rmarkdown` packages. Next, click the ‘Knit’ button in the upper-right hand corner just above the file. An HTML file was created. Find it and open it.

4.3.2 Tips

Start with an example document

When you open a new rmarkdown file (see previous section), the first step is to delete everything in the example document, *except the top section and the first code chunk*. Your rmarkdown document will now look like this:

```
---
title: "Untitled"
output: html_document
---

{r setup, include=FALSE} knitr::opts_chunk$set(echo = TRUE)
```

Modify the output style

You can add lots of options to the header (the bit between --), that affect how the conversion to HTML works (including style, settings for the table of contents, author, date, etc.). See the Rstudio help page for the HTML version of `rmarkdown` by following this link.

We list a few useful options; to see how these can be used search for them on the link above.

- `code_folding: hide`, hides the code in the document, but adds a button to reveal the code.
- `toc_float: true`, a very nice floating table of contents (the contents are always visible in a menu to the left)
- `theme: cosmo`, or lots of other choices to be picked from this gallery, changes the colors, fonts, and general appearance of the document.

Learn markdown syntax for text formatting

As mentioned, the rest of the document contains two types of input: text with markup, and code chunks. Open a new rmarkdown document to see what the text with input looks like, and note the use of sections, formatting (bold, italic, etc.), weblinks, and so on. Follow the Rstudio `rmarkdown` lessons to learn more.

Set options for individual code chunks

The code chunks contain R code, which will be executed, and the results will be visible in the eventual document (in our case, an HTML page). An R code chunk may look like this,

```
{r titanicglm, echo = TRUE, eval = TRUE} fit <- glm(Survived ~ Age, data = titanic)
```

Here, `titanicglm` is a label for the chunk (you don't need to set it), and `echo = TRUE` make the code visible in the document, and `eval=TRUE` actually runs the code in the chunk. Both these options are used a lot. You use `echo=FALSE` to not show code, for example when there is too much messy code to share for this particular block. You use `eval = FALSE` to disable execution of a single code chunk. This is very useful when you have some code that simply does not work yet, but you hate using comments (#) to disable it, instead you can place the code in its own chunk and set `echo=FALSE, eval=FALSE`.

Set options for figure output

An important step is to control the output of figures. The documentation (see @Section(rmarkdownresources)) is a bit overwhelming, but here is an example with some useful settings:

```
{r titanicplot1, fig.cap="Number of passengers by Sex and PClass", fig.width=4,
fig.height=4, dev="svg", echo = -1}`r''` par(bg="azure") with(titanic, spineplot(Sex,
PClass))
```

Here, `fig.cap` makes a figure caption (but figures will not be automatically numbered, see below for an alternative), `fig.width` and `fig.height` set the figure size in inches, `dev="svg"` sets the graphic device to SVG (an outstanding format for the web) - you need to install `svglite` for this to work. Finally `echo=-1` hides the first row of the code in the output (in this case, we don't want the reader to be bothered by graphical settings in `par()`).

Set options for all code chunks

The section shown above that contains `knitr::opts_chunk$set(echo = TRUE)` sets *chunk options for the entire document*. An often used feature here is to hide all code from the eventual document by setting `echo=FALSE`, or set the figure size for all chunks with `fig.width = 5, fig.height = 5`, or use a particular graphics device (`dev = 'svg'`).

Auto-numbering of figures and tables

In the example above, we set a caption for a figure, but the figures are not automatically numbered in `rmarkdown`. To enable this, all you have to do is install the `bookdown` package, and replace the header with this example:

```
---
title: "Untitled"
output: bookdown::html_document2
---
```

Everything else should work as expected, but now figures that contain a `fig.cap` setting will be automatically numbered!

Make R output pretty

One thing you notice quickly is that R output that is usually printed to the screen, will look exactly the same in an rmarkdown document - ugly. For example, try this code chunk:

```
summary(titanic)
```

The output in the HTML is not attractive, to say the least.

A number of packages will convert raw R output to markdown or HTML output, which can be used effectively in rmarkdown documents. Here we recommend using the `pander` package to convert many R objects to readable output for use in rmarkdown documents, and the `kable` function from the `knitr` package to convert things that look like dataframes. Both packages are very easy to use.

```
library(pander)    # for pander()
library(knitr)     # for kable()
library(magrittr)  # for %>%

summary(titanic) %>% pander
summary(titanic) %>% kable
```

Try the `pander` function on many outputs from statistical analyses as well, for example:

```
test_output <- t.test(Age ~ Sex, data = titanic)
pander(test_output)
```

4.3.3 Resources

The documentation for `rmarkdown` based documents is a little scattered across various sources. The reason for this is that multiple packages are involved in making an HTML page from an `.Rmd` file.

- The `knitr` package handles the *execution* of the R code in the document, and the *integration* of the R output into the document. It provides all options that are set at the top of each code chunk, for example to show or hide the code, to control figure size and resolution, captions for tables. All chunk options are listed on the `knitr` website.
- The `rmarkdown` package (from Rstudio) handles the *conversion* to the desired output format, for example HTML or PDF. To do so, it uses Pandoc (see below), which is installed automatically with Rstudio. The `rmarkdown` package provides all options at the top of the `Rmd` document (the so-called YAML header), including the desired output format, and to control many options like the table of contents, format style, custom CSS, and so on. All options for `rmarkdown` are described

on Rstudio's website. (Make sure to scroll down and select an output format to read about available options).

- Finally, Pandoc is the actual engine to convert our raw text file (including R output formatted by `knitr`) to an HTML page, or some other format. Normally you don't have to learn much about Pandoc's options, since many settings are already provided by `rmarkdown`, but the various formatting rules and conversion options are described here.

Further reading A comprehensive list of useful and more advanced features in rmarkdown can be read here: https://holtzy.github.io/Pimp-my-rmd/

Chapter 5

Basic statistics and linear modelling

5.1 Introduction

This book is not an *Introduction to statistics*. There are many books focused on *statistics*, with or without example R code. The focus throughout this book is on the R code itself, as we try to present clear and short solutions to common analysis tasks. In the following, we assume you have a basic understanding of linear regression, Student's t-tests, ANOVA, and confidence intervals for the mean.

Linear modelling is a general technique that includes linear regression, but also basic means testing, comparing samples, and more complex combinations of class and numeric predictors.

In linear modelling we are interested in understanding variation in a single *numeric response variable* (or sometimes binary). We can include all sorts of *predictors* ('explanatory variables', 'independent variables'), depending on what we chose we end up with various techniques:

- factor variable(s), test difference in mean in our predictor across the factor ("Analysis of variance", "t-test")
- a single numeric predictor (simple linear regression)
- multiple numeric predictors, and interactions (multiple linear regression)
- mixture of factors and numeric predictors (in the simplest case, analysis of covariance - ANCOVA)

There are many more complicated possibilities, but the above is what this chapter is all about.

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`, `ggthemes`, `scales` and `ggpubr` (these are frequently omitted from the examples):

Otherwise:

- `lgrdata` (for the example datasets)
- `pastecs` (for descriptive statistics)
- `dplyr` (for various data skills and the pipe operator %>%)
- `pander` (to format R output for use in rmarkdown documents)
- `broom` (for tidy printing of fitted models)
- `multcomp` (for multiple comparisons)
- `visreg` (for visualizing regression models)

- car (for various functions used in regression)
- emmeans (for computing marginal effects from regression models)

5.2 Probability distributions

You will have encountered a number of probability distributions before. For example, the *Binomial* distribution is a model for the distribution of the number of *successes* in a sequence of independent trials, for example, the number of heads in a coin tossing experiment. Another commonly used discrete distribution is the *Poisson*, which is a useful model for many kinds of count data. Of course, the most important distribution of all is the *Normal* or Gaussian distribution.

R provides sets of functions to find densities, cumulative probabilities, quantiles, and to draw random numbers from many important distributions. The names of the functions all consist of a one letter prefix that specifies the type of function and a stem which specifies the distribution. Look at the examples in the table below.

Prefix	Meaning
‘d‘	density
‘p‘	cumulative probability
‘q‘	quantile
‘r‘	simulate

Suffix	Meaning
‘binom‘	Binomial
‘pois‘	Poisson
‘norm‘	Normal
‘t‘	Student’s t
‘chisq‘	Chi-squared
‘f‘	F

Using the prefix and the suffix, we can construct each desired function. For example,

```
# Calculate the probability of 3 heads out of 10 tosses of a fair coin.
# This is a (d)ensity of a (binom)ial distribution.
dbinom(3, 10, 0.5)
```

```
## [1] 0.1171875
```

```
# Calculate the probability that a normal random variable (with
# mean of 3 and standard deviation of 2) is less than or equal to 4.
# This is a cumulative (p)robability of a (norm)al variable.
pnorm(4, 3, 2)
```

```
## [1] 0.6914625
```

```
# Find the t-value that corresponds to a 2.5% right-hand tail probability
# with 5 degrees of freedom.
# This is a (q)uantile of a (t)distribution.
qt(0.975, 5)
```

```
## [1] 2.570582
```

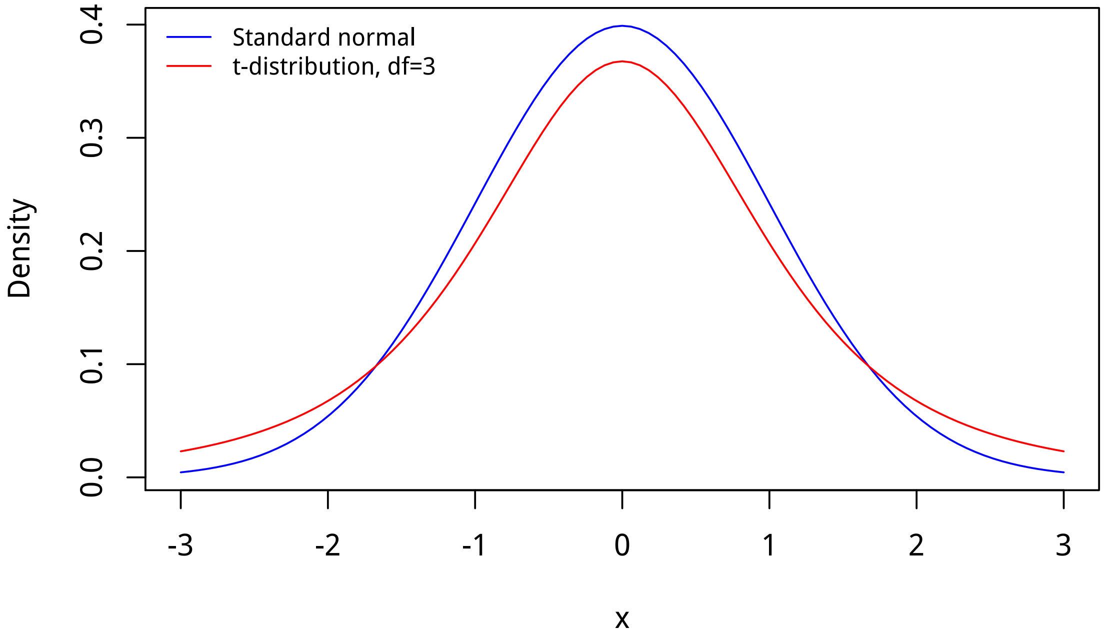


Figure 5.1: Two univariate distributions plotted with curve()

```
# Simulate 5 Poisson random variables with a mean of 3.
# This is a set of (r)andom numbers from a (pois)son distribution.
rpois(5, 3)
```

```
## [1] 1 3 3 3 5
```

See the help page `?Distributions` for more details.

To make a quick plot of a distribution, we can use the density function in combination with `curve`. The following code makes Fig. 5.1.

```
# A standard normal distribution
curve(dnorm(x, sd=1, mean=0), from=-3, to=3,
      ylab="Density", col="blue")

# Add a t-distribution with 3 degrees of freedom.
curve(dt(x, df=3), from =-3, to=3, add=TRUE, col="red")

# Add a legend (with a few options, see ?legend)
legend("topleft", c("Standard normal","t-distribution, df=3"), lty=1, col=c("blue","red"),
       bty='n', cex=0.8)
```

Try this yourself Make a histogram of a sample of random numbers from a distribution of your choice. use `hist` and one of the functions starting with `r` from above.

5.3 Descriptive Statistics

Descriptive statistics summarise some of the properties of a given data set. Generally, we are interested in measures of location (central tendency, such as mean and median) and scale (variance or standard deviation). Other descriptions can include the sample size, the range, and so on. We already encountered a number of functions that can be used to summarize a vector.

Let's look at some examples for the Pupae dataset.

```
# Read data
data(pupae)

# Extract the weights (for convenience)
weight <- pupae$PupalWeight

# Find the number of observations
length(weight)
```

```
## [1] 84
```

```
# Find the average (mean) weight
mean(weight)
```

```
## [1] 0.3110238
```

```
# Find the Variance
var(weight)
```

```
## [1] 0.004113951
```

Note that R will compute the sample variance (not the population variance). The standard deviation can be calculated as the square root of the variance, or use the `sd` function directly.

```
# Standard Deviation
sqrt(var(weight))
```

```
## [1] 0.06414009
```

```
# Standard Deviation
sd(weight)
```

```
## [1] 0.06414009
```

Robust measures of the location and scale are the median and inter-quartile range; R has functions for these.

```
# median and inter-quartile range
median(weight)
```

```
## [1] 0.2975
```

```
IQR(weight)
```

```
## [1] 0.09975
```

The median is the 50th percentile or the second quartile. The `quantile` function can compute quartiles as well as arbitrary percentiles/quantiles.

```
# Default: computes quartiles.
quantile(weight)
```

```
##      0%     25%     50%     75%    100%
## 0.17200 0.25625 0.29750 0.35600 0.47300
```

```
# Or set any quantiles
quantile(weight, probs=seq(0,1,0.1))
```

```
##     0%    10%    20%    30%    40%    50%    60%    70%    80%    90%
## 0.1720 0.2398 0.2490 0.2674 0.2892 0.2975 0.3230 0.3493 0.3710 0.3910
##   100%
## 0.4730
```

Missing Values: All of the above functions will return `NA` if the data contains *any* missing values. However, they also provide an option to remove missing values (`NA`s) before their computations (see also Section 3.7.3).

```
mean(pupae$Frass)
```

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

```
mean(pupae$Frass, na.rm=TRUE)
```

```
## [1] 1.846446
```

The `summary` function provides a lot of the above information in a single command:

```
summary(weight)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1720  0.2562  0.2975  0.3110  0.3560  0.4730
```

The `moments` package provides 'higher moments' if required, for example, the `skewness` and `kurtosis`.

```
# load the moments package
library(moments)
skewness(weight)
```

```
## [1] 0.3851656
```

```
kurtosis(weight)
```

```
## [1] 2.579144
```

The `pastecs` package includes a useful function that calculates many descriptive statistics for numeric vectors, including the standard error for the mean (for which R has no built-in function).

```
library(pastecs)
```

```
##
## Attaching package: 'pastecs'
## The following object is masked from 'package:magrittr':
## 
##     extract
```

```
## The following objects are masked from 'package:dplyr':
## 
##     first, last
```

```r
# see ?stat.desc for description of the abbreviations
stat.desc(weight)
```

```
##      nbr.val     nbr.null       nbr.na          min          max
## 84.000000000  0.000000000  0.000000000  0.172000000  0.473000000
##        range          sum       median         mean      SE.mean
##  0.301000000 26.126000000  0.297500000  0.311023810  0.006998258
## CI.mean.0.95          var      std.dev     coef.var
##  0.013919253  0.004113951  0.064140091  0.206222446
```

```r
# conveniently, the output is a character vector which we can index by name,
# for example extracting the standard error for the mean
stat.desc(weight)["SE.mean"]
```

```
##     SE.mean
## 0.006998258
```

Sometimes you may wish to calculate descriptive statistics for subgroups in the data. We will come back to this extensively in Section 3.8.1 and later sections.

5.4 Testing differences between groups

5.4.1 Testing a single sample

In some applications we simply want to know whether the mean of our data (the sample) is equal to some hypothesized value, or whether it is clearly higher or lower. The simplest approach is to compute a 95% *confidence interval for the mean*, and then check whether the hypothesized value falls inside this interval (in which case you cannot conclude the value is different).

One way to get confidence intervals in R is to use the quantile functions for the relevant distribution. A $100(1-\alpha)$% confidence interval for the mean on normal population is given by,

$$\bar{x} \pm t_{\alpha/2,n-1}\frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the sample mean, s the sample standard deviation and n is the sample size. $t_{\alpha/2,n-1}$ is the $\alpha/2$ tail point of a t-distribution on $n-1$ degrees of freedom. That is, if T has a t-distribution on $n-1$ degrees of freedom.

$$P(T \leq t_{\alpha/2,n-1}) = 1 - \alpha/2$$

The R code for this confidence interval can be written as,

```
# Sample data - the pupae data.
data(pupae)
weight <- pupae$PupalWeight

# 95% confidence interval - set to 0.1 for a 90% interval
alpha <- 0.05
xbar <- mean(weight)
s <- sd(weight)
n <- length(weight)
half.width <- qt(1-alpha/2, n-1)*s/sqrt(n)

# Confidence Interval
c(xbar - half.width, xbar + half.width)
```

```
## [1] 0.2971046 0.3249431
```

Here, we assumed a normal distribution for the population. You may have been taught that if `n` is *large*, say n>30, then you can use a normal approximation. That is, replace `qt(1-alpha/2, n-1)` with `qnorm(1-alpha/2)`, but there is no need, R can use the t-distribution for any *n* (and the results will be the same, as the t-distribution converges to a normal distribution when the df is large).

Try this yourself Confirm that the t-distribution converges to a normal distribution when *n* is large (using `qt` and `qnorm`).

5.4.2 Hypothesis testing

There may be a reason to ask whether a dataset is consistent with a certain mean. For example, are the pupae weights consistent with a population mean of 0.29? For normal populations, we can use Student's t-test, available in R as the `t.test` function. In all following results, we use the `pander` function (from the `pander`) package to make the output short and readable (and in markdown format). For best results, use it in an rmarkdown document (though it works fine otherwise as well). We assume you have loaded these two packages:

```
library(pander)
library(dplyr)
```

Let's test the null hypothesis that the population mean is 0.29:

```
t.test(weight, mu=0.29) %>% pander
```

Table 5.1: One Sample t-test: `weight`

Test statistic	df	P value	Alternative hypothesis	mean of x
3.004	83	0.00352 * *	two.sided	0.311

Note that we get the t-statistic, degrees of freedom (n-1) and a P value for the test, with the specified alternative hypothesis (not equal, i.e. two-sided). In addition, `t.test` gives us the estimated mean of the sample.

We can use `t.test` to do one-sided tests,

```
t.test(weight, mu=0.29, alternative="greater") %>% pander
```

Table 5.2: One Sample t-test: weight

Test statistic	df	P value	Alternative hypothesis	mean of x
3.004	83	0.00176 * *	greater	0.311

The t.test is appropriate for data that is approximately normally distributed. You can check this using a histogram or a QQ-plot (see Sections sec:hist and sec:diagplots). If the data is not very close to a normal distribution then the t.test is often still appropriate, as long as the sample is large.

If the data is not normal and the sample size is small, there are a couple of alternatives: transform the data (often a log transform is enough) or use a *nonparametric* test, in this case the Wilcoxon signed rank test. We can use the wilcox.test function for the latter, its interface is similar to t.test and it tests the hypothesis that the data is symmetric about the hypothesized population mean. For example,

```
wilcox.test(weight, mu=0.29) %>% pander
```

Table 5.3: Wilcoxon signed rank test with continuity correction: weight

Test statistic	P value	Alternative hypothesis
2316	0.009279 * *	two.sided

5.4.2.1 Test for proportions

Sometimes you want to test whether observed proportions are consistent with a hypothesized population proportion. For example, consider a coin tossing experiment where you want to test the hypothesis that you have a fair coin (one with an equal probability of landing heads or tails). In your experiment, you get 60 heads out of 100 coin tosses. Do you have a fair coin? We can use the prop.test function:

```
# 60 'successes' out of a 100 trials, the hypothesized probability is 0.5.
prop.test(x=60, n=100, p=0.5) %>% pander
```

Table 5.4: 1-sample proportions test with continuity correction: 60 out of 100, null probability 0.5

Test statistic	df	P value	Alternative hypothesis	p
3.61	1	0.05743	two.sided	0.6

Likewise, we can perform one-sided tests with the argument alternative="greater" (not shown).

5.4.3 Inference for two populations

Commonly, we wish to compare two (or more) populations. For example, the pupae dataset has pupal weights for female and male pupae. We may wish to compare the weights of males (gender=0) and females (gender=1).

To compare the pupal weights of males and females, we use the *formula* interface for `t.test`. The formula interface is important because we will use it in many other functions, like linear regression and linear modelling.

```
# (output not shown)
# We assume equal variance between the groups, giving slightly more power,
# but see section 'unequal variances' further below.
t.test(PupalWeight ~ Gender, data=pupae, var.equal=TRUE)
```

5.4.3.1 Paired data

The `t.test` can also be used when the data are paired, for example, measurements taken before and after some treatment on the same subjects. For this example, we will use the pulse data - data on pulse rates of individuals before and after exercise. We will test the simple idea that pulse rate is different after exercise. To do this, we first extract only those subjects that exercised (`Ran=1`),

```
# (output not shown)
pulse <- read.table("ms212.txt", header=TRUE)
pulse.before <- with(pulse, Pulse1[Ran==1])
pulse.after <- with(pulse, Pulse2[Ran==1])
t.test(pulse.after, pulse.before, paired=TRUE)
```

5.4.3.2 Unequal variances

The default for the two-sample `t.test` is actually to *not* assume equal variances. The theory for this kind of test is quite complex, and the resulting t-test is now only approximate, with an adjustment called the 'Satterthwaite' or 'Welch' approximation made to the degrees of freedom.

```
t.test(PupalWeight ~ Gender,  data=pupae) %>% pander
```

Table 5.5: Welch Two Sample t-test: `PupalWeight` by `Gender` (continued below)

Test statistic	df	P value	Alternative hypothesis
-7.413	74.63	1.587e-10 * * *	two.sided

mean in group 0	mean in group 1
0.2725	0.3513

Since this modified t-test makes fewer assumptions, you could ask why we ever use the equal variances form. If the assumption is reasonable, then this (equal variances) form will have more power, i.e. will reject the null hypothesis more often when it is actually false.

5.4.4 Testing many groups at once (ANOVA)

One-way ANOVA (ANalysis Of VAriance) can be used to compare means across two or more populations. We will not go into the theory here, but the foundation of ANOVA is comparing the variation *between* the group means to the variation *within* the groups (using an F-statistic).

We can use either the aov function or lm to perform ANOVAs. We will focus exclusively on the latter as it can be generalized more easily to other models. The use of aov is only appropriate when you have a balanced design (i.e., the same sample sizes in each of your groups).

To use lm for an ANOVA, we need a dataframe containing a (continuous) response variable and a factor variable that defines the groups. For example, in the Coweeta dataset, the species variable is a factor that defines groups by species. We can compute (for example) the mean height by species. Let's look at an example using the Coweeta data, but with only four species to simplify the output.

```
library(dplyr)

# Take a subset and drop empty levels with droplevels.
data(coweeta)
cowsub <- filter(coweeta, species %in% c("bele","cofl","oxar","quru")) %>%
  droplevels

# # Quick summary table (uses dplyr)
group_by(cowsub, species) %>%
  summarize(height = mean(height))
```

```
## # A tibble: 4 x 2
##   species height
##   <fct>    <dbl>
## 1 bele     21.9
## 2 cofl      6.80
## 3 oxar     16.5
## 4 quru     21.1
```

We might want to ask, does the mean height vary by species? Before you do any test for significance, a graphical summary of the data is always useful. For this type of data, box plots are preferred since they visualize not just the means but also the spread of the data (Fig. 5.2).

```
boxplot(height~species, data=cowsub)
```

It seems like some of the species differences are quite large. We can fit a one-way ANOVA with lm, like so:

```
fit1 <- lm(height ~ species, data=cowsub)
```

The fitted coefficients can be seen in summary(fit1), or more concisely with coef(fit1), but we prefer the use of the broom package, especially the functions tidy and glance :

```
library(broom)
tidy(fit1)
```

```
## # A tibble: 4 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   21.9        1.56     14.0   7.02e-14
## 2 speciescofl  -15.1        2.93     -5.15  2.04e- 5
```

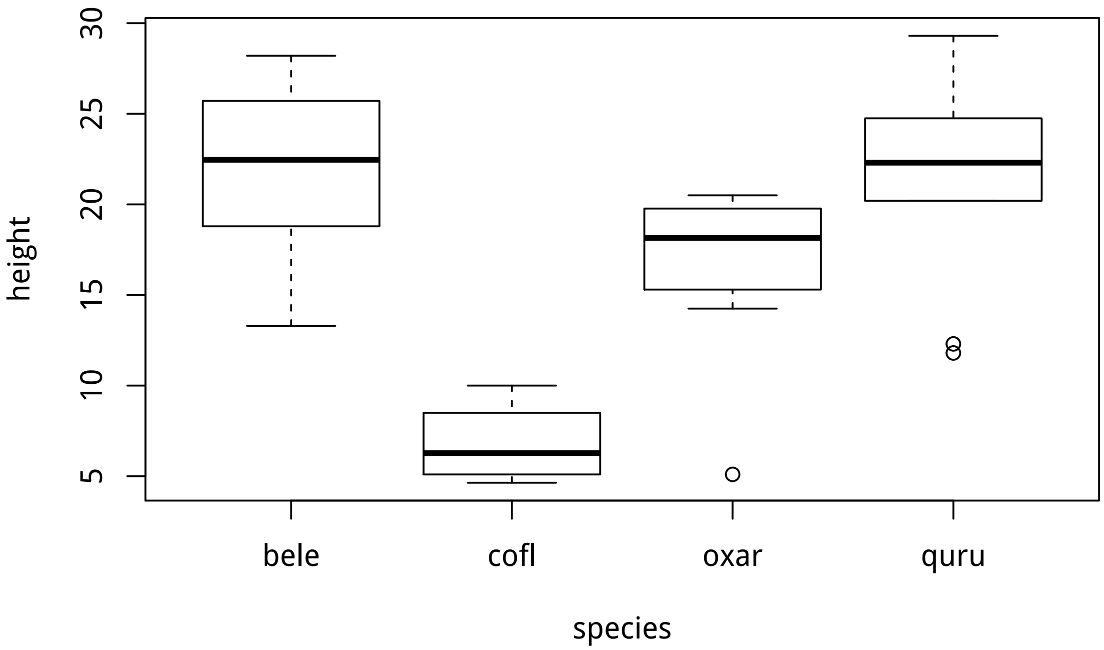


Figure 5.2: Simple box plot for the Coweeta data.

```
## 3 speciesoxar   -5.36       2.35    -2.28  3.04e- 2
## 4 speciesquru   -0.761      2.27    -0.335 7.40e- 1
```

Here, `tidy` gives the table of coefficients. Notice the four estimated `Coefficients`, these represent the so-called *contrasts*. In this case, `Intercept` represents the mean of the *first* species, `bele`. The next three coefficients are the differences between each species and the first (e.g., species `cofl` has a mean that is -15.07 lower than `bele`). Also shown are the *t*-statistic (and p-value) for each coefficient, for a test where the value is compared to zero. Not surprisingly the `Intercept` (i.e., the mean for the first species) is significantly different from zero (as indicated by the very small p-value). Two of the next three coefficients are also significantly different from zero.

We can get more details of the fit using `glance`.

```
glance(fit1)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl> <dbl>
## 1     0.533         0.481  4.95      10.3 1.11e-4     4  -91.4  193.  200.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

Here we see various useful statistics including the R squared (goodness of fit), and the p-value of the overall model. This p-value tells us whether the whole model is significant. In this case, it is comparing a model with four coefficients (one for each species) to a model that just has the same mean for all groups. In this case, the model is highly significant – i.e. there is evidence of different means for each group. In other words, a model where the mean varies between the four species performs much better than a model with a single grand mean.

5.4.5 Multiple comparisons

The ANOVA, as used in the previous section, gives us a single p-value for the overall 'species effect'. The summary statement further shows whether individual species are different from the first level in the model, which is not always useful. If we want to know whether the four species were all different from each other, we can use a multiple comparison test.

> **Caution** Multiple comparisons on linear models where you have more than one factor variable are tricky, and probably best left alone. Read the help page ?`glht` for more information.

We will use the `glht` function from the `multcomp` package (as a side note, base R includes the `TukeyHSD` function, but that does not work with `lm`, only with `aov`).

```
# First fit the linear model again (a one-way ANOVA,
# because species is a factor)
lmSpec <- lm(height ~ species, data=cowsub)

# Load package
library(multcomp)

# Make a 'general linear hypothesis' object, Tukey style.
# (Note many other options in ?glht)
tukey_Spec <- glht(lmSpec, linfct=mcp(species="Tukey"))

# Print a summary. This shows p-values for the null hypotheses
# that species A is no different from species B, and so on.
summary(tukey_Spec)
```

```
##
##   Simultaneous Tests for General Linear Hypotheses
##
## Multiple Comparisons of Means: Tukey Contrasts
##
##
## Fit: lm(formula = height ~ species, data = cowsub)
##
## Linear Hypotheses:
##                  Estimate Std. Error t value Pr(>|t|)
## cofl - bele == 0 -15.0695     2.9268  -5.149   <0.001 ***
## oxar - bele == 0  -5.3620     2.3467  -2.285   0.1247
## quru - bele == 0  -0.7614     2.2731  -0.335   0.9866
## oxar - cofl == 0   9.7075     3.0295   3.204   0.0169 *
## quru - cofl == 0  14.3081     2.9729   4.813   <0.001 ***
## quru - oxar == 0   4.6006     2.4039   1.914   0.2434
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## (Adjusted p values reported -- single-step method)
```

```
# Some of the species are different from each other, but not all.
```

Try this yourself In the above summary of the multiple comparison (`summary(tukey_Spec)`), the p-values are adjusted for multiple comparisons with the so-called 'single-step method'. To use a different method for the correction (there are many), try the following example: `summary(tukey_Spec, test=adjusted("Shaffer"))` Also look at the other options in the help pages for `?adjusted` and `?p.adjust`.

We can also produce a quick plot of the multiple comparison, which shows the pair-wise differences between the species with confidence intervals.

This code produces Fig. 5.3.

```
# A plot of a fitted 'glht' object (multiple comparison)
plot(tukey_Spec)
```

5.4.6 Comparing many groups by two predictors (two-way ANOVA)

Sometimes there are two (or more) *treatment* factors. The 'age and memory' dataset (see `?memory`) includes the number of words remembered from a list for two age groups and five memory techniques.

This dataset is balanced, as shown below. in a table of counts for each of the combinations. First we fit a linear model of the *main effects*.

```
data(memory)

# To make the later results easier to interpret, reorder the Process
# factor by the average number of words remembered.
memory <- mutate(memory,
```

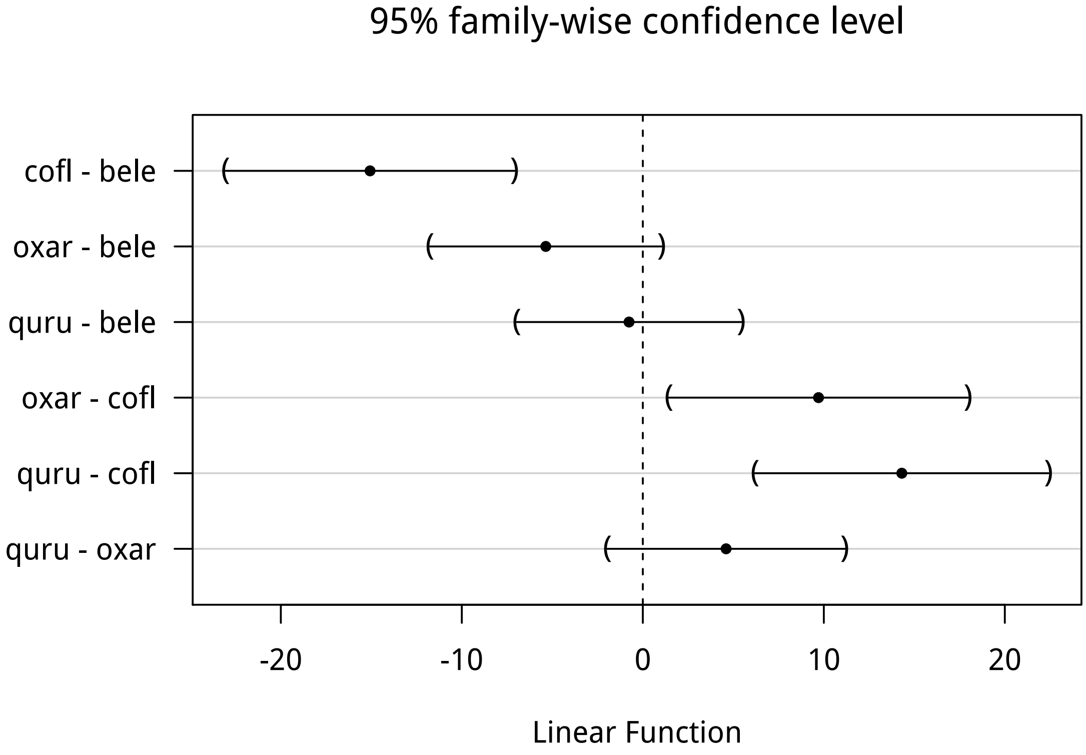


Figure 5.3: A standard plot of a multiple comparison.

```
                  Process = reorder(Process, Words, mean))

# Count nr of observations
xtabs( ~ Age + Process, data=memory)
```

```
##            Process
## Age         Counting Rhyming Adjective Imagery Intentional
##   Older           10      10        10      10          10
##   Younger         10      10        10      10          10
```

```
# Fit linear model
fit2 <- lm(Words ~ Age + Process, data=memory)

# Full output (not shown)
# summary(fit2)

# Instead, show just coefficient summary table in nice format
broom::tidy(fit2)
```

```
## # A tibble: 6 x 5
##   term                estimate std.error statistic  p.value
##   <chr>                  <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)             5.20     0.763     6.81  8.99e-10
## 2 AgeYounger              3.10     0.623     4.97  2.94e- 6
## 3 ProcessRhyming          0.5      0.985     0.507 6.13e- 1
## 4 ProcessAdjective        6.15     0.985     6.24  1.24e- 8
## 5 ProcessImagery          8.75     0.985     8.88  4.41e-14
## 6 ProcessIntentional      8.90     0.985     9.03  2.10e-14
```

The summary of the fitted model displays the individual t-statistics for each estimated coefficient. As with the one-way ANOVA, the significance tests for each coefficient are performed relative to the base level (by default, the first level of the factor). In this case, for the Age factor, the Older is the first level, and for the Process factor, Adjective is the first level. Thus all other coefficients are tested relative to the "Older/Adjective" group. The F-statistic at the end is for the overall model, it tests whether the model is significantly better than a model that includes only a mean count.

If we want to see whether Age and/or Process have an effect, we need F-statistics for these terms. Throughout this book, to compute p-values for terms in linear models, we use the Anova function from the car package.

```
# Perform an ANOVA on a fitted model, giving F-statistics
library(car)
Anova(fit2)
```

```
## Anova Table (Type II tests)
##
## Response: Words
##            Sum Sq Df F value    Pr(>F)
## Age        240.25  1  24.746 2.943e-06 ***
## Process   1514.94  4  39.011 < 2.2e-16 ***
## Residuals  912.60 94
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

In this form, the F-statistic is formed by comparing models that do not include the term, but include all others. For example, Age is tested by comparing the full model against a model that includes all other terms (in this case, just Process).

5.4.7 Interactions

An important question when we have more than one factor in an experiment is whether there are any interactions. For example, do Process effects differ for the two Age groups, or are they simply additive? We can add interactions to a model by modifying the formula. An interaction is indicated using a ":". We can also include all *main effects and interactions* using the * operator.

```
# Two equivalent ways of specifying a linear model that includes all main effects
# and interactions:
fit3 <- lm(Words ~ Age + Process + Age:Process, data=memory)

# Is the same as:
fit3.2 <- lm(Words ~ Age * Process, data=memory)
Anova(fit3.2)
```

```
## Anova Table (Type II tests)
##
## Response: Words
##              Sum Sq Df F value    Pr(>F)
## Age          240.25  1 29.9356 3.981e-07 ***
## Process     1514.94  4 47.1911 < 2.2e-16 ***
## Age:Process  190.30  4  5.9279 0.0002793 ***
## Residuals    722.30 90
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The Anova table shows that the interaction is significant. When an interaction is significant, this tells you nothing about the direction or magnitude of the interaction term. You can inspect the estimated coefficients in the summary output, but we recommend to first visualize the interaction with simple plots, as the coefficients can be easily misinterpreted. One way to visualize the interaction is to use the interaction.plot function, as in the following example.

This code produces Fig. 5.4. If there were no interaction between the two factor variables, you would expect to see a series of parallel lines (because the effects of Process and Age would simply be additive).

```
# Plot the number of words rememberd by Age and Process
# This standard plot can be customized in various ways, see ?interaction.plot
with(memory, interaction.plot(Age, Process, Words))
```

Try this yourself When there is a signficant interaction term, we might want to know under what levels of one factor is the effect of another factor signficant. This can be done easily using functions in the emmeans package. Load the emmeans package, and run the code fit3.emm <- emmeans(fit3, ~ Age | Process), followed by pairs(fit3.emm). You can now inspect in great detail differences between levels of your predictors.

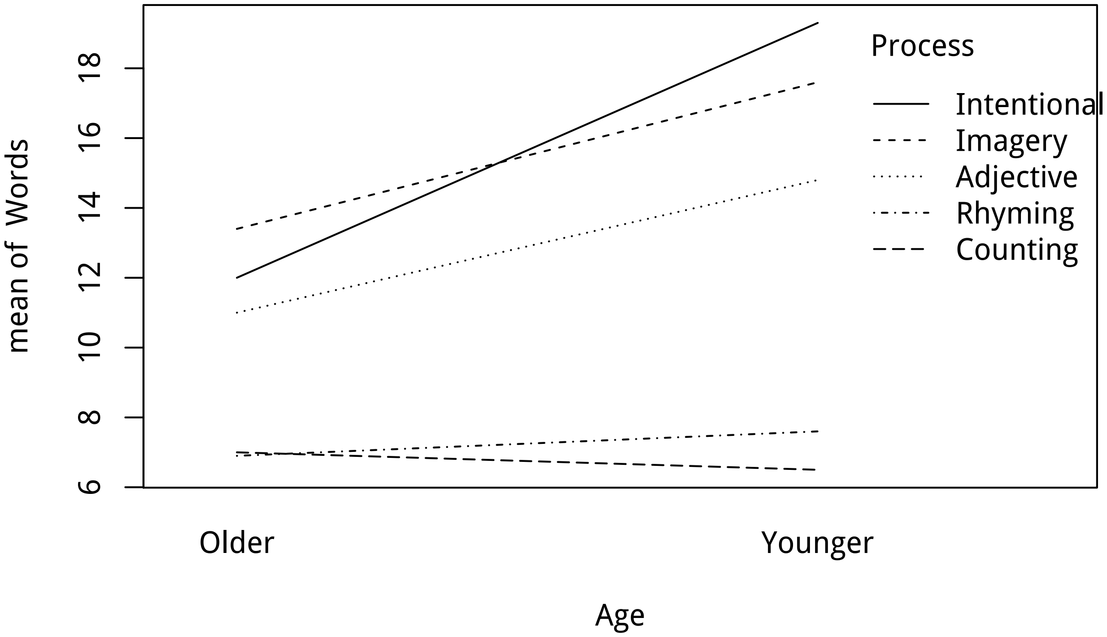


Figure 5.4: An interaction plot for the memory data, indicating a strong interaction (because the lines are not parallel).

5.4.8 Comparing models

In the above example, we fitted two models for the Memory dataset: one without, and one with the interaction between Process and Age. We assessed the significance of the interaction by inspecting the p-value for the Age:Process term in the Anova statement. Another possibility is to perform a likelihood ratio test on two 'nested' models, the model that includes the term, and a model that excludes it. We can perform a likelihood ratio test with the anova function, not to be confused with Anova from the car package![1]

```
# We can perform an anova on two models to compare the fit.
# Note that one model must be a subset of the other model.
# In this case, the second model has the same predictors as the first plus
# the interaction, and the likelihood ratio test thus tests the interaction.
anova(fit2,fit3)
```

```
## Analysis of Variance Table
##
## Model 1: Words ~ Age + Process
## Model 2: Words ~ Age + Process + Age:Process
##   Res.Df   RSS Df Sum of Sq      F    Pr(>F)
## 1     94 912.6
## 2     90 722.3  4     190.3 5.9279 0.0002793 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

A second common way to compare different models is by the AIC (*Akaike's Information Criterion*), which is calculated based on the likelihood (a sort of goodness of fit), and penalized for the number of parameters (i.e. number of predictors in the model). The model with the lowest AIC is the preferred model. Calculating the AIC is simple,

```
AIC(fit2, fit3)
```

```
##      df      AIC
## fit2  7 518.9005
## fit3 11 503.5147
```

We once again conclude that the interaction improved the model fit substantially.

5.4.9 Diagnostics

The standard ANOVA assumes normality of the residuals, and we should always check this assumption with some diagnostic plots (Fig. 5.5). Although R has built-in diagnostic plots, we prefer the use of qqPlot and residualPlot, both from the car package.

```
par(mfrow=c(1,2))

library(car)

# Residuals vs. fitted
residualPlot(fit3)
```

[1] You may have seen the use of 'anova' on a single model, which also gives an ANOVA table but is confusing because a sequential (Type-I) test is performed, which we strongly advise against as it is never the most intuitive test of effects.

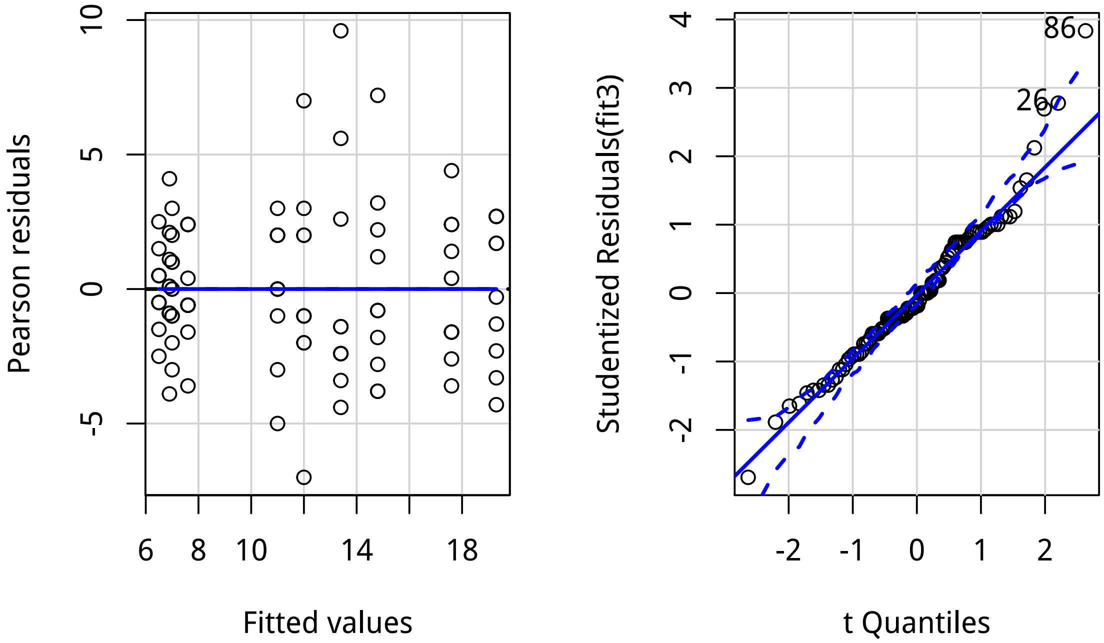


Figure 5.5: Simple diagnostic plots for the Memory ANOVA.

```
# QQ-plot of the residuals
qqPlot(fit3)
```

```
## [1] 26 86
```

The QQ-plot shows some slight non-normality in the upper right. The non-normality probably stems from the fact that the `Words` variable is a 'count' variable.

> **Try this yourself** Check whether a log-transformation of `Words` makes the residuals closer to normally distributed.

5.5 Linear regression

5.5.1 Icecream sales: a motivating example

It is well known that ice cream sales are higher during warm weather. As an ice cream salesman, you hardly have to apply regression analysis to know this - it is just obvious. Suppose that one year, you sell icecream at Oosterpark in Amsterdam. Sales are pretty good, the summer is nice and warm. The next year you try your luck at the Dappermarkt, a busy market in the east of Amsterdam. You tally your sales every week, and finally make a plot comparing average weekly sales:

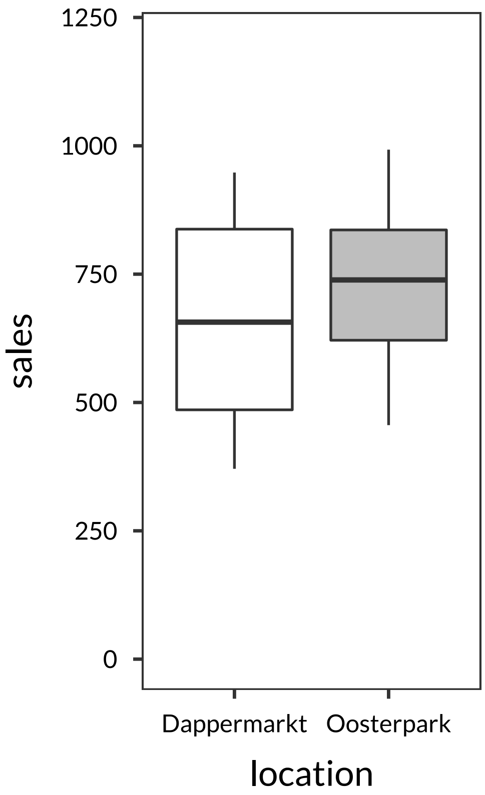


Figure 5.6: Average weekly ice cream sales at two locations, over two years

```
data(icecream)

library(ggplot2)
library(ggthemes)

ggplot(icecream, aes(x = location, y = sales, fill = location)) +
  scale_fill_manual(values=c("white","grey")) +
  geom_boxplot() +
  theme(aspect.ratio = 2, legend.position="none") +
  lims(y=c(0,1200))
```

Based on this simple boxplot, you would either conclude that there is no difference in sales, or that perhaps sales were slightly better in Oosterpark. We can also perform a *t*-test for two samples, which tests the hypothesis that the two means are not different (in other words, location has no effect on sales).

```
t.test(sales ~ location, data=icecream)
```

```
##
##  Welch Two Sample t-test
##
## data:  sales by location
## t = -1.3227, df = 36.442, p-value = 0.1942
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -183.16655   38.51966
## sample estimates:
## mean in group Dappermarkt  mean in group Oosterpark
##                  660.4856                  732.8090
```

Note The example above - as all other examples in this chapter - shows the raw output produced by R. To produce attractive output for use in an rmarkdown document, use the `pander` package: `pander::pander(t.test(sales ~ location, data=icecream))`. The `pander` function also works for many other R objects.

From this test we conclude no difference: the P value is very large (meaning the observed sample difference is not unusual, and could have easily arisen by chance).

Of course, there is something rather fishy about this experiment. We are testing the difference between two locations - but both locations were 'tested' in different years, when other conditions may have been different. In other words, we have not performed a proper experiment at all - changing just one variable at a time - location is *confounded* by the year.

We do not need to get into detail here to design a proper scientific ice cream sales experiment, instead we are going to make the point that we can account for some *confounding variables* via regression analysis. How about temperature? As an astute ice cream salesman, you downloaded weekly average air temperature for Amsterdam, and added these to the data. Now let's make a plot using *air temperature as a covariate*.

```r
# Assuming you have loaded ggplot2 and ggthemes
ggplot(icecream, aes(x = temperature, y = sales, fill = location)) +
  geom_point(shape = 21, size=3) +
  scale_fill_manual(values = c("white","dimgrey")) +
  lims(y=c(0,1200), x=c(0,35))
```

In this figure we arrive at a completely different conclusion - but only because we have taken into account the covariate weekly temperature. Now, *at a given temperature*, ice cream sales were higher at the new location - Dappermarkt. This is an example of how a confounding variable can mask (or even reverse) effects of some other variable. It seems obvious from the figure that temperature was overall lower when you sold ice cream at the Dappermarkt, explaining why average weekly sales were in the end no different.

Try this yourself Use a *t*-test to test that temperature was lower at the Dappermarkt location.

Using regression analysis, we can test the assertion that ice cream sales are higher at Dappermarkt, *at a given temperature*. But before we do that, we first introduce the tools to test difference between means (as we just did with a simple two-sample t-test), and introduce basic linear regression. We return to the icecream problem in Section 5.6.2.

5.5.2 Simple and multiple linear regression

To fit linear models of varying complexity, we can use the `lm` function. The simplest model is a straight-line relationship between an `x` and a `y` variable. In this situation, the assumption is that the `y`-variable

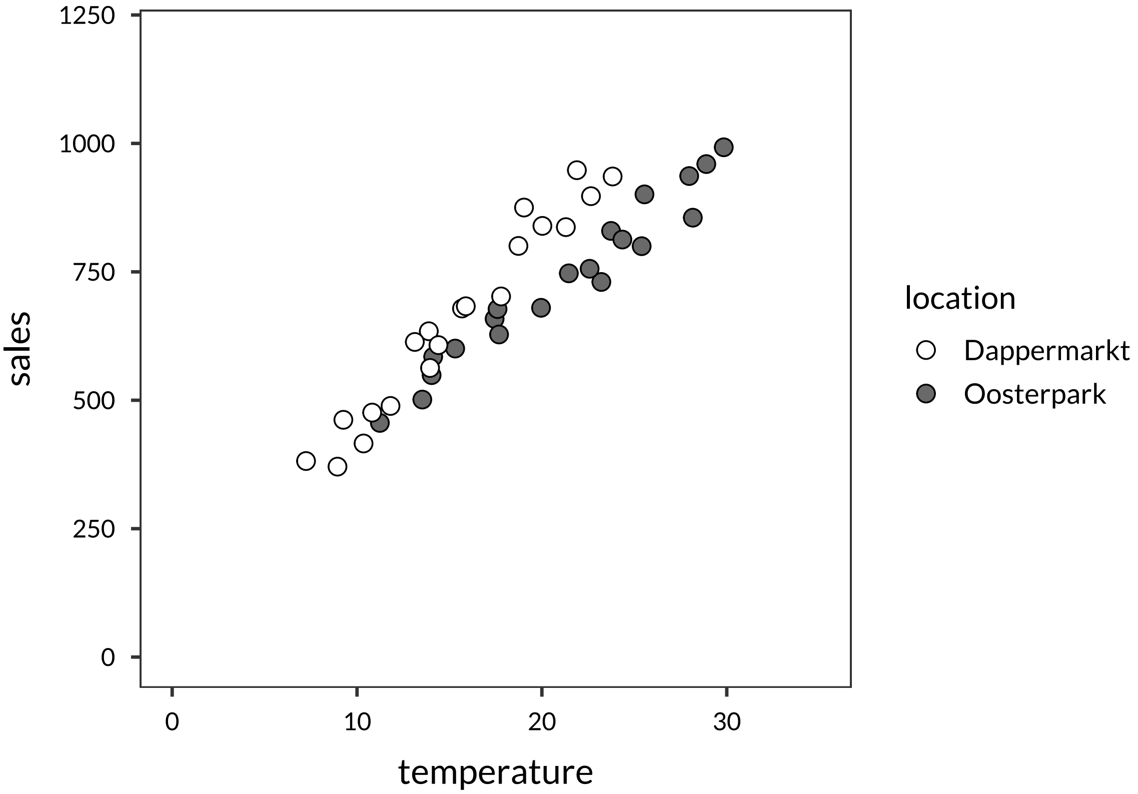


Figure 5.7: Ice cream sales by location, and varying with temperature.

(the response) is a linear function of the x-variable (the independent variable), plus some random noise or measurement error. For the simplest case, both x and y are assumed to be continuous variables.

We use this method to study the relationship between two continuous variables: a *response* ('y variable') and a *predictor* (or independent variable) ('x variable'). We can use multiple regression to study the relationship between one response and more than one predictor.

We are going to use the Cereals data to inspect the "health rating" (rating), and two predictors: fibre content (fiber) and sugar content (sugar). To start with, let's make two scatter plots, side by side.

```
# Read the data, if you haven't already
data(cereals)

library(scales)
library(gridExtra)

# Two simple scatter plots on a log-log scale
g1 <- ggplot(cereals, aes(y = rating, x = fiber)) +
  geom_point(pch=15) + lims(y=c(0,100)) +
  geom_smooth(method='lm')

g2 <- ggplot(cereals, aes(y = rating, x = sugars)) +
  geom_point(pch=19, col="dimgrey") + lims(y=c(0,100)) +
  geom_smooth(method='lm')

grid.arrange(g1, g2, ncol = 2)
```

Here we make use of ggplot2's built-in regression lines, which are easily added to the plot. The default behaviour is to add a 95% confidence band as well. In this case we immediately see that fiber is a worse predictor than sugars for the health rating, given the wider confidence interval for the location of the mean response (i.e. the regression line).

We are going to fit a model that looks like, using statistical notation,

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \varepsilon \tag{5.1}$$

where α is the intercept, x_1 and x_2 are the two predictors (fiber and sugars), and the *two* slopes are β_1 and β_2. We are particularly interested in testing for significant effect of both predictors, which is akin to saying that we are testing the values of the two slopes against zero.

```
# Fit a multiple regression without interactions, and inspect the summary
fit4 <- lm(rating ~ sugars + fiber, data=cereals)

# A detailed summary (not shown for brevity)
# summary(fit4)

# A clean summary, especially for in an rmarkdown file
fit4 %>% pander
```

Table 5.7: Fitting linear model: rating ~ sugars + fiber

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	52.17	1.556	33.54	4.266e-46
sugars	-2.244	0.1632	-13.75	5.975e-22

	Estimate	Std. Error	t value	Pr(>|t|)
fiber	2.867	0.2979	9.623	1.276e-14

As you can see, we again use the pander package to give super clean results, which can be used in an rmarkdown document. We see here that overall we have an R^2 of 81% (not bad), and that both predictors contribute significantly (the P values are very low for both).

Next, we perform some diagnostic plots (shown in Fig. 5.9). The standard errors (and confidence intervals, and hypothesis tests) will be unreliable if the assumptions of normality (right panel) or linearity (left panel) are seriously violated. If you are only interested in the mean response against your predictors, but not so much their uncertainties, you can usually ignore these plots.

```
# Basic diagnostic plots.
par(mfrow=c(1,2))
car::residualPlot(fit4)
car::qqPlot(fit4)
```

```
## [1] 27 31
```

Now that we have a fitted model, what do its predictions actually look like? What relationship have we learned between health rating, sugars, and fiber content? The visreg package provides a very concise method to visualize (many kinds of) fitted regression models.

In this case, the command visreg::visreg(fit4) would show two plots very similar to the ones above - which does not really add much information. What if we can show the effects of both predictors, in just one plot? The following code produces Fig. 5.10.

```
library(visreg)

visreg(fit4, "sugars", by = "fiber", overlay = TRUE,
       gg = TRUE) +
  lims(y = c(0,100))
```

Here we clearly, and in one panel, visualize the negative effect of adding more sugar in a cereal, and the positive effect of adding more fiber. The visreg function plots the *predictions* from our model at an arbitrary selection of predictor values (here, 0, 1.5 and 5).

Try this yourself In the call to visreg, use the argument breaks=c(1,5) to set the predictor values to those of your choice.

We can also make 3D plots, though these are not always useful, and I personally prefer a 'pseudo-3D' plot as in the example above to visualize the contribution of two variables.

Of course, we are not restricted to just two predictors, and we can include interactions as well. In the previous model (fit4) the effects of fiber and sugars were *additive*, in other words the effect of fiber was independent of the amount of sugars. We can test this assumption by adding the interaction to the model.

```
# A multiple regression that includes all main effects as wel as interactions
fit5 <- lm(rating ~ sugars + fiber + fiber:sugars, data=cereals)
summary(fit5) %>% pander::pander(.)
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	51.38	1.695	30.31	9.996e-43

	Estimate	Std. Error	t value	Pr(>|t|)
sugars	-2.097	0.2056	-10.2	1.277e-15
fiber	3.226	0.428	7.537	1.12e-10
sugars:fiber	-0.07363	0.06316	-1.166	0.2475

Table 5.9: Fitting linear model: rating ~ sugars + fiber + fiber:sugars

Observations	Residual Std. Error	R^2	Adjusted R^2
76	6.112	0.8198	0.8123

(*Note*: the formula can also be written as `fiber*sugars`)

In this case, the interaction is not significant: note the large P value for the *sugars:fiber* interaction. We can also inspect the `AIC`, and notice that the total model AIC has not decreased, which also indicates the model has not improved by adding an interaction term.

```
AIC(fit4, fit5)
```

```
##      df      AIC
## fit4  4 496.1572
## fit5  5 496.7359
```

5.6 Linear models with factors and continuous variables

So far we have looked at ANOVA, including two-way ANOVA, where a response variable is modelled as dependent on two treatments or factors, and *regression* including multiple linear regression where a response variable is modelled as dependent on two continuous variables. These are just special cases of the *linear model*, and we can extend these simple models by including a mix of factors and continuous predictors. The situation where we have one continuous variable and one factor variable was classically`known as ANCOVA (analysis of covariance), but using the`lm' function we can specify any model with a combination of predictors.

In the following example, we inspect the height of trees against the stem diameter (`DBH`) for the Coweeta example dataset. When plotting all data, we are led to believe that the overall response of height to DBH is highly non-linear (left panel). However, when we fit a simple linear model for each species separately (right panel), the relationship appears to consist of a combination of linear responses.

```
data(coweeta)

g1 <- ggplot(coweeta, aes(x = DBH, y = height)) +
  geom_point() + geom_smooth(method="loess", se=FALSE) +
  scale_colour_tableau() + lims(x=c(0,60), y=c(0,30))

g2 <- ggplot(coweeta, aes(x = DBH, y=height, col=species)) +
  geom_point() + geom_smooth(method="lm", se=FALSE) +
  scale_colour_tableau() + lims(x=c(0,60), y=c(0,30))

grid.arrange(g1, g2, ncol = 2)
```

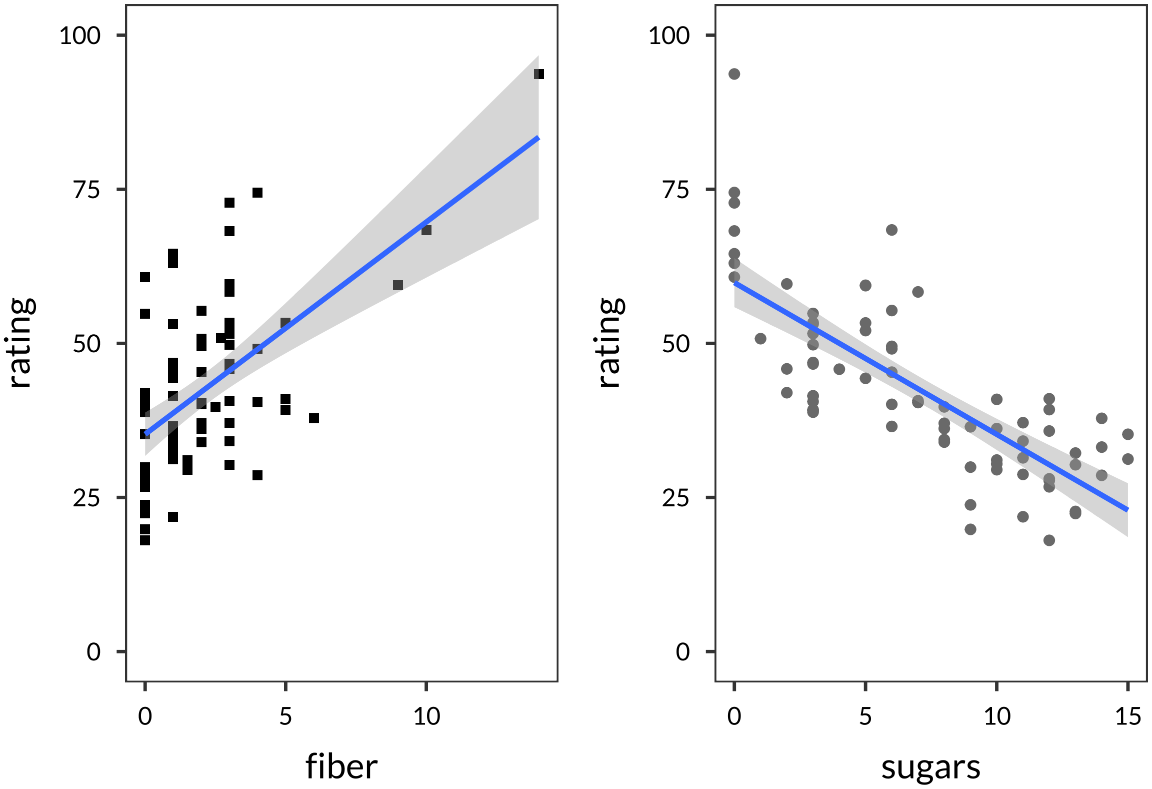


Figure 5.8: Leaf area as a function of height and diameter (note the log-log scale).

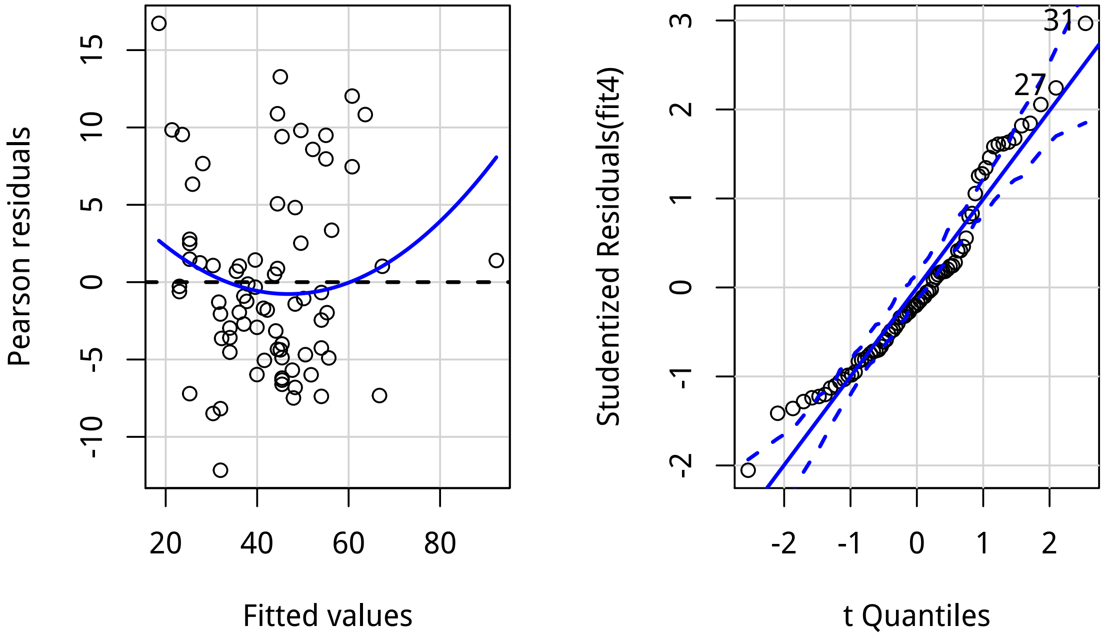


Figure 5.9: Diagnostic plots for linear regression of the Cereals data.

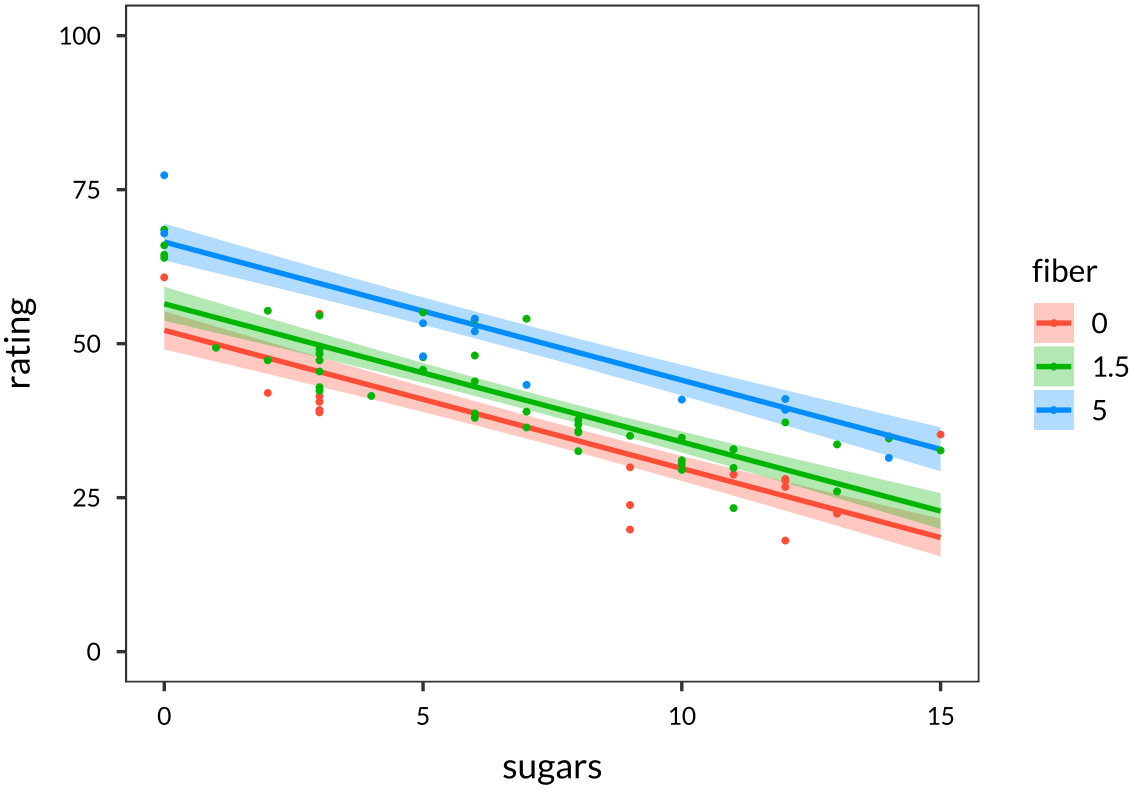
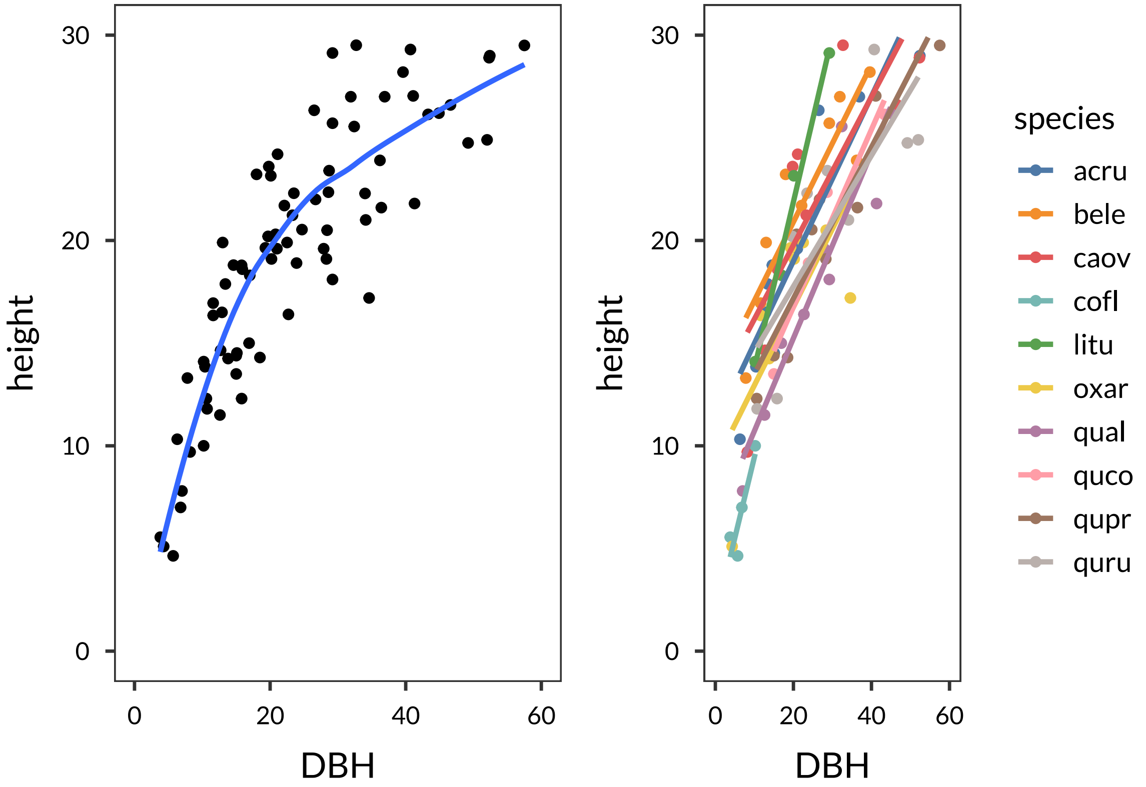


Figure 5.10: Visualizing linear regression of the Cereals data with visreg.



To fit a model that allows a different response for each species, we use the notation from before:

```
fit6 <- lm(height ~ species * DBH, data=coweeta)
```

5.6.1 Visualizing fitted regression models

The coefficients associated with a factor predictor in a linear model are given as contrasts (i.e. differences between factor levels).

While this is useful for comparisons of treatments, it is often more instructive to visualize the predictions at various combinations of factor levels.

A number of options exist to extract and visualize fitted regression models - to make sense of differences between groups, the effects of covariates, and the impact of interactions. We prefer the `visreg` package, which can be used to make attractive plots of the predictions of a linear model.

The following example makes Fig. 5.11.

```
library(visreg)

# Load data (lgrdata package needed)
data(memory)

# To make the later results easier to interpret, reorder the Process
# factor by the average number of words rememberer (we did this earlier in
# the chapter already for this dataset, it is repeated here).
```

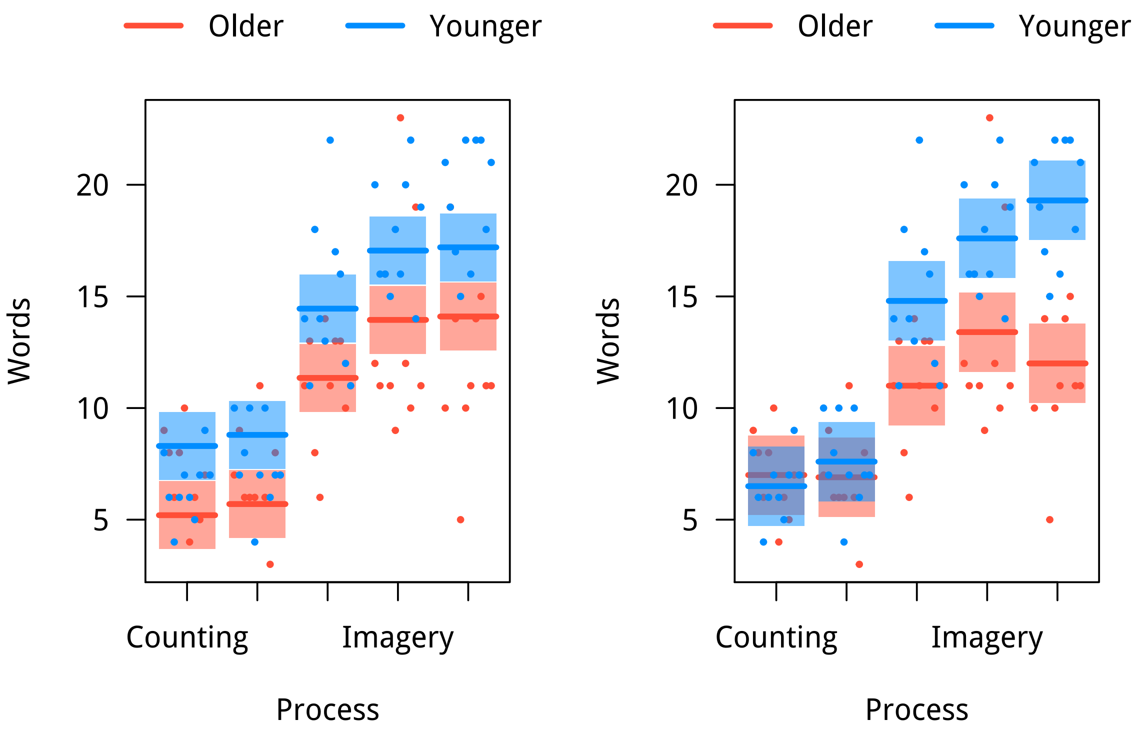


Figure 5.11: Visualization of two fitted linear model with the visreg package. The model on the right includes an interaction between Process and Age, the model on the left does not.

```r
memory <- mutate(memory,
                 Process = reorder(Process, Words, mean))

# Two linear models: one without, and one with an interaction
fit7 <- lm(Words ~ Age + Process, data=memory)
fit8 <- lm(Words ~ Age*Process, data=memory)

# Here we specify which variable should be on the X axis (Process),
# and which variable should be added with different colours (Age).
visreg(fit7, "Process", by="Age", overlay=TRUE)
visreg(fit8, "Process", by="Age", overlay=TRUE)
```

You can test for yourself that the interaction is a highly significant predictor in the model.

5.6.2 Group differences in regression models: back to ice cream sales

At the beginning of this chapter (Section 5.5.1), we looked at an example where including a covariate drastically changed our conclusions about an effect. Simply testing the difference between two locations where we sold ice creams showed no effect, but including temperature at a covariate, it appeared that *at a given temperature*, there was a difference between the locations.

Here, we continue this example as it shows that we need to closely inspect a fitted model before making any conclusions - and that estimated coefficients of a regression model can lead to wrong conclusions, if not carefully examined.

To continue, first read in the icecream data as shown in Section 5.5.1. We now fit a linear model with one numeric, and one factor variable. We immediately include all effects, including the interaction.

```
lm_ice1 <- lm(sales ~ location * temperature, data=icecream)
```

You can inspect all details in the summary of this model yourself (`summary(lm_ice1)`) - here we just look at the estimated coefficients from the model. Recall that in a linear model, the intercept is assigned to the first level of the factor variable in the model, in this case `levels(icecream$location)` shows that this is 'Dappermarkt'.

```
pander(lm_ice1)
```

Table 5.10: Fitting linear model: sales ~ location * temperature

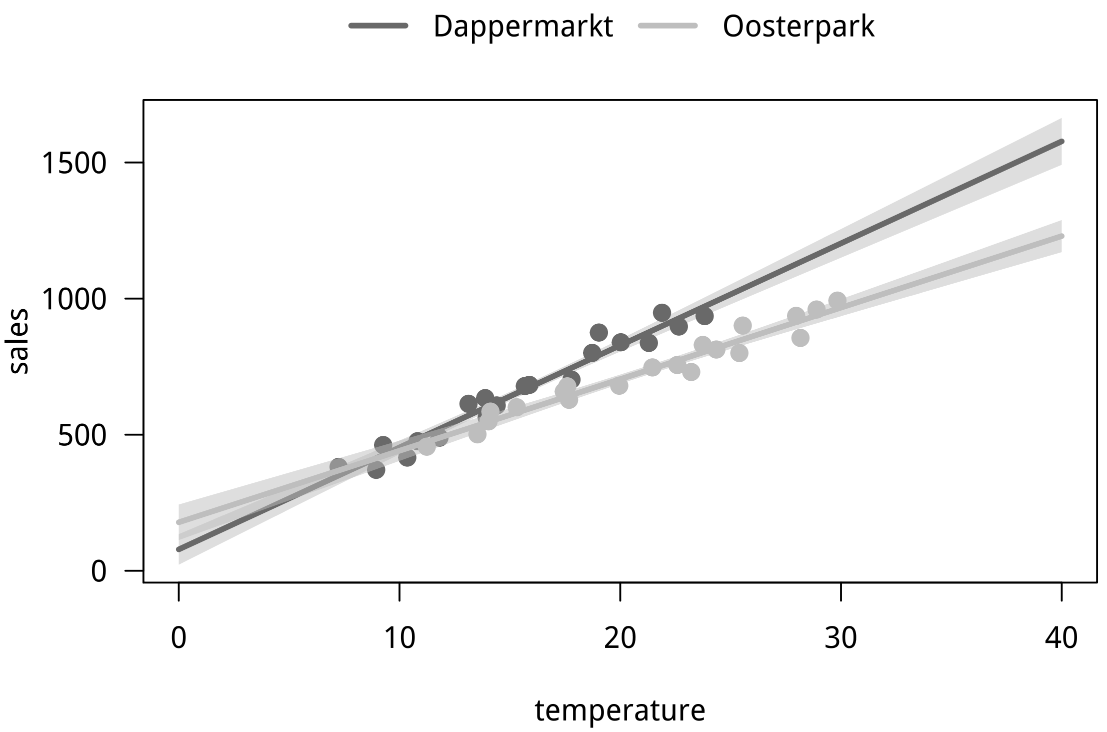
	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	78.41	27.72	2.829	0.00759
locationOosterpark	99.64	42.53	2.343	0.02477
temperature	37.48	1.704	22	1.837e-22
locationOosterpark:temperature	-11.19	2.255	-4.963	1.683e-05

Four coefficients are shown, these are:

- *Intercept* : ice cream sales for the first level of 'Location' (Dappermarkt), when temperature equals zero (i.e. the intersection of the regression line with the y-axis when x = 0).
- *locationOosterpark* : The difference in intercept for Oosterpark location, compared to the first level (Dappermarkt). A positive value here seems to indicate higher sales at the Oosterpark location (when temperature is zero - read further below!).
- *temperature* : The slope of sales per unit temperature (i.e. for every degree increase in temperature, sales go up this much), for Dappermarkt.
- *locationOosterpark:temperature* : the difference in *slope* for the Oosterpark, compared to Dappermarkt. This shows the slope is lower for Oosterpark.

Based on the above, it seems difficult to draw conclusions about the difference between locations. This is because the intercept is not at all informative: it quantifies ice cream sales when temperature equals zero, but we do not even try to sell ice cream when it is cold outside. To visualize what is going on, we modify the standard `visreg` plot to draw the regression lines all the way to zero. We also show how to adjust the appearance of the plot (see `?plot.visreg` for details).

```
visreg(lm_ice1, "temperature", by="location",
       overlay = TRUE, xlim=c(0,40),
       line.par=list(col=c("dimgrey","grey")),
       fill.par=list(col=c("#BEBEBE80","#BEBEBE80")),
       points.par=list(col=c("dimgrey","grey"), cex=1.2))
```



We now see that the regression lines 'cross over' around a temperature of 10, giving a higher intercept for Oosterpark.

A meaningful test in situations like this is to test for group differences *at a given value of our covariate(s)*. A number of approaches exist in various add-on packages, but we prefer to first show the basic approach using `predict` - a function that has methods for practically all models we use. It also offers great flexibility, and we don't have to wonder what exactly we are testing.

Somewhat arbitrarily, we want to test for location differences at a temperature of 25 degrees. To do this, I will construct confidence intervals for both locations, as follows. In the call to `predict`, it is important to list values for all variables in the model. This makes this approach less practical for models with many more regressors (see below for another approach).

```
# Predict from model, at temperature =25.
pred_ice <- predict(lm_ice1,
                    newdata = data.frame(temperature = 25,
                                         location = levels(icecream$location)),
                    se = TRUE)
```

If you inspect this object, you will find that sales at Dappermarkt are 1015.4 with a standard error of 18.1, and sales at Oosterpark are 835.2 (SE 10.1). The standard errors may be used to construct approximate 95% confidence intervals (by using twice the standard error).

A more convenient approach is to use the `emmeans` package to extract *marginal effects* of our fitted model. The default approach in this package is to test for group differences at the average level of other predictors. This approach is also called 'least-square means'. However, we can adjust what value we want to use.

```
library(emmeans)
emmeans(lm_ice1, specs = "location", at = list(temperature = 25))
```

```
## NOTE: Results may be misleading due to involvement in interactions

##  location    emmean   SE df lower.CL upper.CL
##  Dappermarkt   1015 18.1 36      979     1052
##  Oosterpark     835 10.1 36      815      856
##
## Confidence level used: 0.95
```

You can ignore the warning in this case, since we have used both the predictors in our model - there are no other 'hidden' predictors that may complicate matters. As you can see for yourself, the `emmeans` package gives the same results as the basic `predict` approach, but it is certainly much more practical.

5.6.3 Logarithmic transformation

For data that are very heteroscedasctic, that is, have quickly increasing variance with the value of the predictor, it is usually necessary to use a logarithm transformation of the data. In extreme cases, not applying the log-transform leads to very poor model fits, with very large influence of a few points, and possibly a fit that does not describe the data well at all.

See Fig. 5.12 for an example where a log-transform is used.

```
data(coweeta)
library(scales)

g1 <- ggplot(coweeta, aes(x = biomass, y = folmass)) +
  geom_point() +
  labs(x = "Mass of tree (kg)", y = "Mass of leaves (kg)")

g2 <- g1 +
  scale_x_log10() +
  scale_y_log10() +
  annotation_logticks() +
  expand_limits(x=c(1,10^4), y=100)

grid.arrange(g1, g2, ncol = 2)
```

Try this yourself Redo the above plot, using `origin` (i.e. American, Japanese or European) to color the symbols. This is another good example that you should always consider covariates in a linear model.

The following code fits two linear models to the data above, one to the raw, untransformed data, and one after log10-transforming both the predictor and the response variable. In previous examples we used `residualPlot` from the `car` package to quickly make plots of residuals versus fitted values, here we implement a simple version using `ggplot2` (and note that we jump ahead and define our own function, a topic we return to in Chapter @()).

```
lm1 <- lm(folmass ~ biomass, data = coweeta)
lm2 <- lm(log10(folmass) ~ log10(biomass), data = coweeta)
```

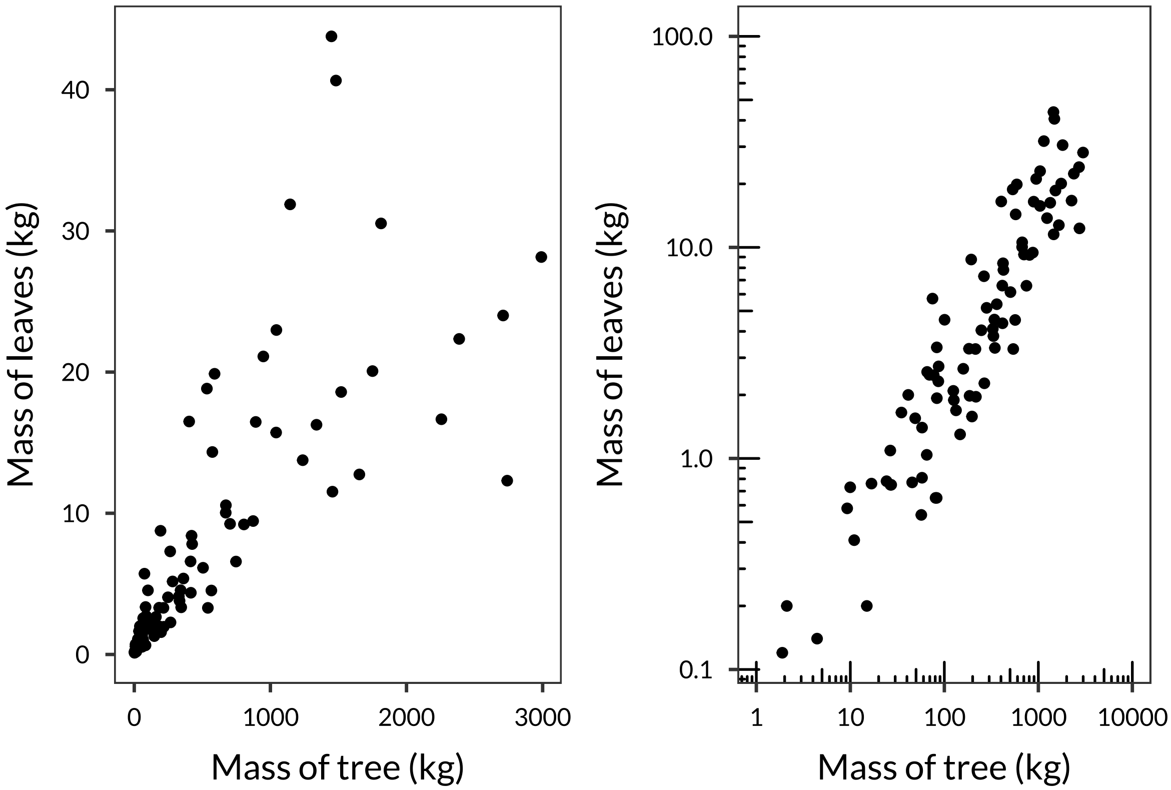


Figure 5.12: An example of a dataset that screams for a logarithmic transformation. Left panel: untransformed data, showing very non-constant variance, indicating larger variation in the response variable for larger individuals. Right panel: both variables have been log-transformed, giving a nice linear relationship with constant variance.

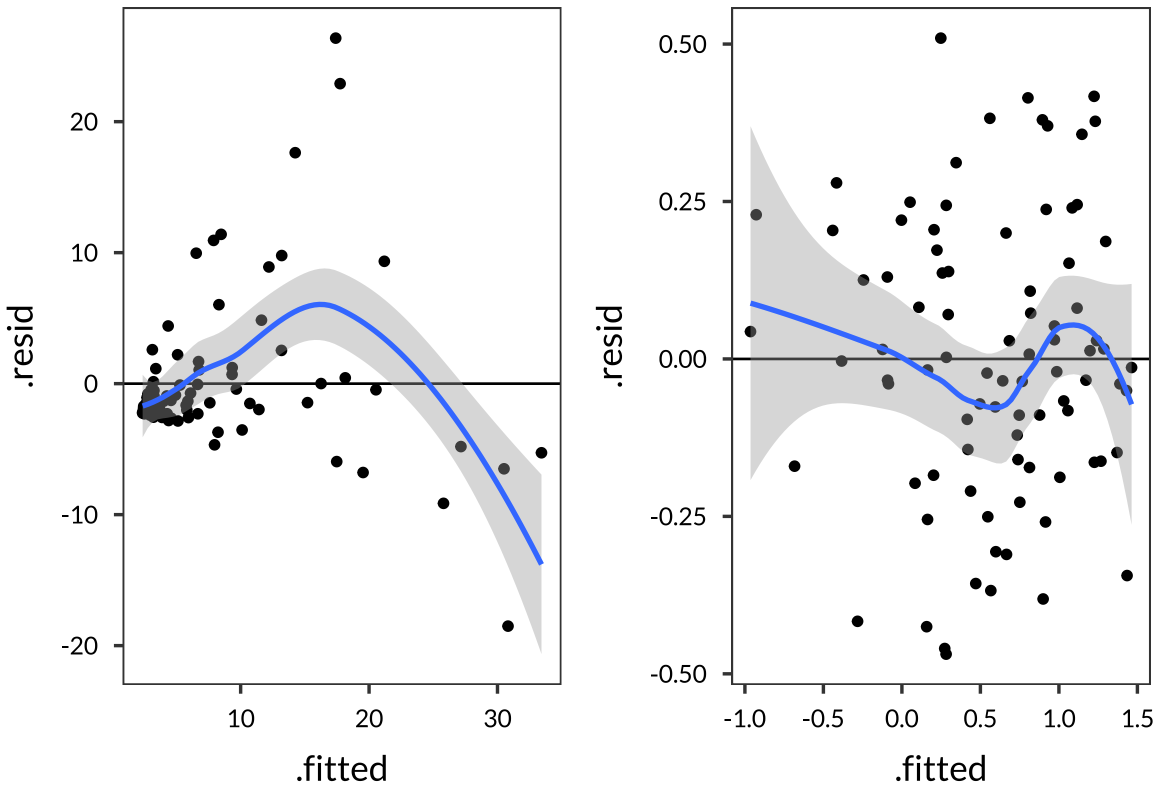


Figure 5.13: Residuals versus fitted for the Coweeta biomass regression model, either without (left panel) or with (right panel) log-transformation of x and y variables.

```r
residPlot <- function(model){
  ggplot(model, aes(.fitted, .resid)) +
    geom_hline(yintercept = 0) +
    geom_point() +
    stat_smooth(method="loess", span=0.8)
}

grid.arrange(residPlot(lm1), residPlot(lm2), ncol = 2)
```

As Fig. 5.13 shows, the log-transformation results in much better model diagnostics. We can expect that the model will give an overall better fit to the data, because when we did not transform, the location of the regression line will be overly influenced by values far away from the line. After transformation, this effect will be much reduced, and the model will be more robust.

Second, the standard errors of the coefficients in the untransformed fit are very much biased, leading to confidence intervals that are much too narrow. In this case, a log-transformation could be used to remedy this problem, but in other cases we want to avoid transformations, or they are simply not adequate. In that case, we might use the bootstrap.

This well known and often used transformation does have some consequences for interpreting the linear model fit that are often misunderstood. The key aspect to understand is that the model is no longer *additive* as the usual linear model (where effects of predicts are simply added), but rather *multiplicative* (effects of various predictors are *multiplied*). This has consequences for understanding interactions in the model as well.

5.6.4 Adding quadratic and polynomial terms

So far we have seen models with just linear terms, but it is straightforward and often necessary to add quadratic (x^2) or higher-order terms (e.g. x^3) when the response variable is far from linear in the predictors. You can add any transformation of the predictors in an `lm` model by nesting the transformation inside the `I()` function, like so:

```
data(allometry)
lmq1 <- lm(height ~ diameter + I(diameter^2), data=allometry)
```

This model fits both the linear (`diameter`) and squared terms (`I(diameter\^2)`) of the predictor, as well as the usual intercept term. If you want to fit all polynomial terms up to some order, you can use the `poly` function like so,

```
lmq1 <- lm(height ~ poly(diameter, 2), data=allometry)
```

This model specification is exactly equivalent to the above, and is more convenient when you have multiple quadratic / polynomial terms and interactions with factor variables.

The following example quickly tests whether a quadratic term improves the model fit of `height` vs. `diameter` for the species `PIPO` in the allometry dataset.

```
data(allometry)
pipo <- subset(allometry, species == "PIPO")

# Fit model with the quadratic term:
lmq2 <- lm(height ~ diameter + I(diameter^2), data=pipo)

# The very small p-value for the quadratic terms shows that the
# relationship is clearly not linear, but better described with a
# quadratic model.
Anova(lmq2)
```

```
## Anova Table (Type II tests)
##
## Response: height
##               Sum Sq Df F value    Pr(>F)
## diameter      821.25  1  53.838 5.905e-07 ***
## I(diameter^2) 365.78  1  23.980 0.0001001 ***
## Residuals     289.83 19
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

When fitting a quadratic model, it is again very useful to use `visreg` to inspect the model that is estimated, since it becomes even more difficult to make sense of the various linear, quadratic, and intercept terms, especially when interactions with factor variables are added. Consider this example for the allometry dataset, which makes Fig. 5.14.

```
# Fit a linear model with linear, quadratic terms, and all species interactions.
allomquadfit <- lm(height ~ poly(diameter, 2)*species, data=allometry)
```

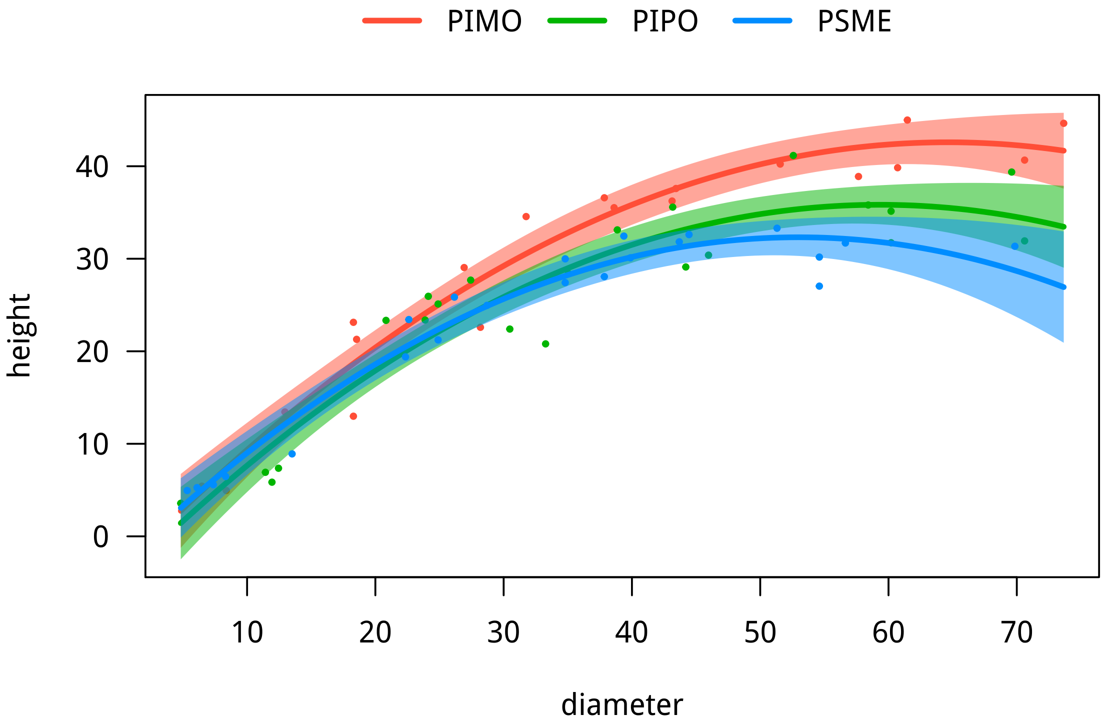


Figure 5.14: Height modelled as a quadratic function of diameter, by species.

```r
# Inspect summary(allomquadfit) to confirm you cannot possibly make sense of this
# many coefficients.

# But plotting the predictions is much easier to understand:
visreg(allomquadfit, "diameter", by="species", overlay=TRUE)
```

5.6.5 Which predictors are more important?

Frequently we are interested in knowing which of the predictor variables explain more variation in the response variable, in other words are more strongly correlated with the response.

One often-seen approach is to compute all correlations between the response and each of the predictors, and conclude that those with higher correlation coefficients will be most important. The flaw in this approach is that predictor variables are very often correlated with each other, making the approach potentially misleading. Cross-correlations between variables can easily hide important predictor variables. Nonetheless it can be a useful start, as we do here with the `automobiles` data. The following code makes Fig. 5.15.

```r
data(automobiles)

# A modern implementation of a correlation plot in the corrplot package.
```

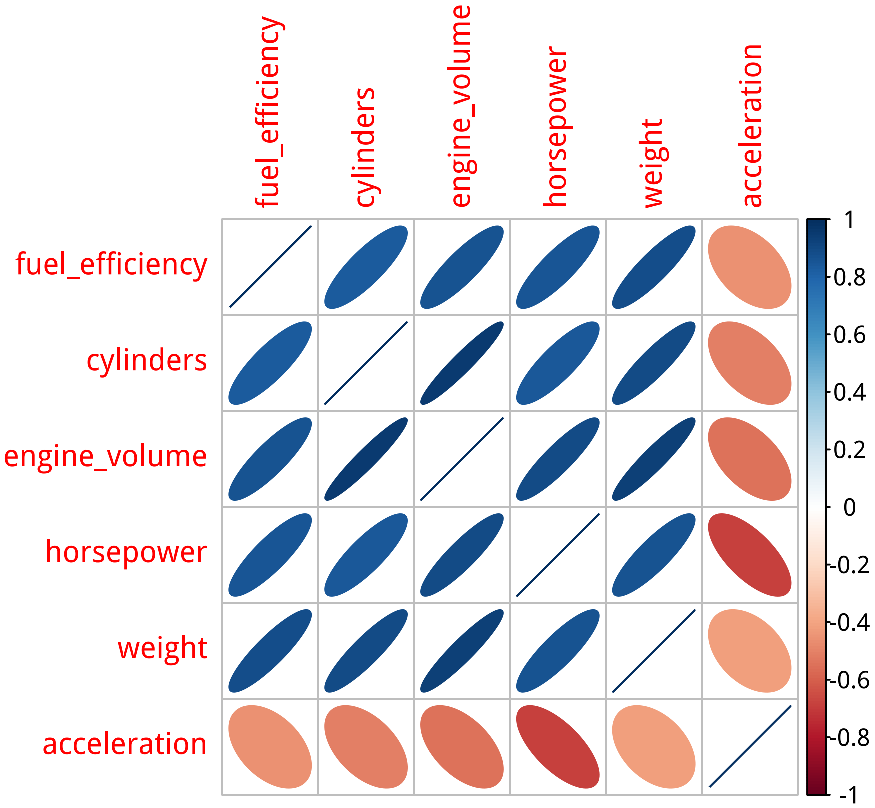


Figure 5.15: Correlation plot for the automobiles data.

```r
library(corrplot)
auto_cor <- cor(automobiles[,4:9], use = "complete.obs")
corrplot(auto_cor, method = "ellipse")
```

As we can see in Fig. 5.15, the response variable (fuel_efficiency) seems well correlated with four of the response variables, but weaker with `acceleration`.

Sometimes we see the use of the absolute value of the t-statistic to rank variables by 'importance' in the regression model. Within a single model, the t-statistic will be exactly related to the p-value (low p-values mean large t-statistic). Both t-statistics and p-values are very poor measures of importance. A large t-statistic may also arise when the sample size is large, the measurements are accurate, or there is low variability overall.

Instead what we are often interested in is how much variation is explained by each variable in the model. This calculation is complicated by the fact that predictor variables tend to be correlated with each other, so you cannot simply add up the R^2 due to each predictor separately.

The `relaimpo` package includes a number of importance measures - the recommended version can be interpreted as the % variation explained by each predictor variable:

```r
# A model with five predictors
fit_fuel1 <- lm(fuel_efficiency ~ cylinders + engine_volume +
                                  horsepower + weight + acceleration,
                data = automobiles)
```

```
# For importance measures
library(relaimpo)

# recommended method lmg in relaimpo
imp_fitfuel <- calc.relimp(fit_fuel1, type="lmg")

# The resulting object is a bit complicated.
# If you print the object (simply type imp_fitfuel), lots of information is given.
# Instead we just need this:
sort(imp_fitfuel$lmg, TRUE)
```

```
##        weight    horsepower engine_volume     cylinders  acceleration
##    0.21667697    0.19895781    0.18619931    0.17158534    0.04737539
```

The values mean that `weight` explains 21% of the variation, but acceleration only 4%. Note that the sum of these values adds up (nearly precisely) to the R^2 of the model (look at `summary(fit_fuel1)`).

Try this yourself A more common approach is to test how mucb the R^2 decreases if a predictor is dropped from the model. When important variables are removed, the model fits a lot worse overall. This can be achieved via:

```
drop1(fit_fuel1)
```

Where RSS is the 'residual sum of squares' after dropping the term; higher means the variable is 'more important'.

Chapter 6

Functions, lists and loops

6.1 Introduction

This chapter demonstrates some building blocks of programming with R. The main purpose here is to allow batch analyses - repeating similar tasks for many subsets of data. To do this, we first have to know how to write our own functions, 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, 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.

Finally we take a brief look at 'for loops', 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.

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.

6.2 Writing simple 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.

```r
# 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] 1.38083
```

```
# Standard error for the mean
SEmean(unifvec)
```

```
## [1] 0.0767329
```

> **Try this yourself** You can use functions that you defined yourself just like any other function, for example in `summaryBy`. First read in the `SEmean` function defined in the example above, and then use the cereal data to calculate the mean and SE of `rating` by `Manufacturer` (or use data of your choosing).

6.2.1 Functions with many arguments

Functions can also have multiple arguments. The following very simple function takes two numbers, and finds the absolute difference between them, using `abs`.

```
# Define function
absDiff <- function(num1,num2)abs(num1 - num2)

# Test it with two numbers:
absDiff(5,1)
```

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

```
# As in many functions in R, you can also give multiple values
# as an argument.
# The following returns the absolute difference between
# 1 and 3, then 5 and 6, and 9 and 0 (in that order).
absDiff(c(1,5,9), c(3,6,0))
```

```
## [1] 2 1 9
```

6.2.2 Functions can return many 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.

```
# 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(seresult, sdresult)
  names(vec) <- c("SE","SD")

return(vec)
}

# Test it:
x <- rnorm(100, mean=20, sd=4)
seandsd(x)
```

```
##        SE        SD
## 0.3553867 3.5538671
```

6.2.3 Functions without arguments

Sometimes, a function takes no arguments (input) at all. Consider this very helpful example.

```
sayhello <- function()message("Hello!")

sayhello()
```

```
## Hello!
```

We will return to defining our own functions when we look at applying functions many times to sections of a dataframe (Section 6.3.6).

6.2.4 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 favorit 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.

6.2.5 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 scatter plot with a loess smoother line. 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). We avoid a more technical explanation of this problem, but simply present a solution with the wrapr package.

Using let from wrapr, we can turn our unquoted variables into quoted ones, like so:

```r
library(wrapr)

let(c(xvar = "weight", yvar = "height"), {
  ggplot(howell, aes(x = xvar, y = yvar)) +
  geom_point(size = 0.8, col = "dimgrey")
})
```

The point of placing our plotting code inside let is that we can now write our simple wrapper function like before:

```r
# Make a scatter plot with the howell data.
plot_scatter_howell <- function(xcol, ycol){
  let(c(xvar = xcol, yvar = ycol), {
    ggplot(howell, aes(x = xvar, y = yvar)) +
    geom_point(size = 0.8, col = "dimgrey")
  })
}

# The function can be used with quoted names:
plot_scatter("height", "weight")
```

It is important to understand that, in this example, let is used *only to turn character arguments into unquoted names*, in this case "height" turns into height for use in ggplot, and so on. We cannot pass height and weight as unquoted names, because R would look for those objects directly (from the 'global environment'), rather than as variables in the howell dataset.

To also pass the dataset as an argument (making the function more general), we do not have to use let, but can immediately set it as an argument (because it is available in the global environment).

```r
# Our function now also takes a dataset as an argument
plot_scatter <- function(xcol, ycol, dataset){
  let(c(xvar = xcol, yvar = ycol), {
    ggplot(dataset, aes(x = xvar, y = yvar)) +
    geom_point(size = 0.8, col = "dimgrey")
  })
}

# We can use it as,
plot_scatter("height", "weight", howell)
```

6.3 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:

```
mylist <- list(a=1:10, txt=c("hello","world"), dfr=data.frame(x=c(2,3,4),y=c(5,6,7)))
```

6.3.1 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
```

6.3.2 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
```

```r
# 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.

6.3.3 Combining lists

Combining two lists can be achieved using `c()`, like this:

```r
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
```

```r
# 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))
```

6.3.4 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.

```r
# 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)
```

```r
# 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"
```

```r
# 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).

6.3.5 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:

```
# Read allom data and make sure 'species' is a factor:
data(allometry)
is.factor(allometry$species)
```

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

```
# The levels of the factor variable 'species'
levels(allometry$species)
```

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

```
# 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
```

```
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.

```
# 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.

6.3.6 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!

6.3.6.1 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.

```
# How many observations per species in the allom dataset?
sapply(allomsp, nrow)
```

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

```
# 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
```

```
# 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
```

```
# 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
```

6.3.6.2 Using `lapply`

The `lapply` function is much like `sapply`, except it always returns a list.

For example,

```
# 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 2.6. 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).

6.4 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: 0.1

## Element 3 of the vector is: 0.1

## Element 4 of the vector is: 0.7

## Element 5 of the vector is: 0.2
```

Note that this is only a toy example: the same result can be achieved by simply typing myvec.

Now let's look at a useful application of a for loop: producing multiple plots in a pdf, using the allomsp object we created earlier.

This bit of code produces a pdf in your current working directory. If you can't find it, recall that you can use getwd() to get the current working directory.

```
# Open a pdf to send the plots to:
pdf("Allom plot by species.pdf", onefile=TRUE)
for(i in 1:3){
  with(allomsp[[i]],
       plot(diameter, leafarea, pch=15, xlim=c(0,80), ylim=c(0,450),
            main=levels(allom$species)[i]))
}
# Close the pdf (important!)
dev.off()
```

Here, we create three plots (i goes from 1 to 3), every time using a different element of the list allomsp. First, i will have the value 1, so that we end up using the dataframe allomsp[[1]], the first element of the list. And so on. Take a look at the resulting PDF to understand how the code works.

Note: On windows (with Adobe reader) If the pdf (Allom plot by species.pdf) is open, the above will fail. If you try this anyway, close the pdf and try again. You may have to run the command dev.off() another time to make sure the device is ready.

Another way to achieve the same result is to avoid splitting the dataframe into a list first, and simply take subsets on the fly. 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()
```

Chapter 7

R on the web

7.1 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).

7.2 Reading data from remote databases

7.2.1 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.

```
# 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.

7.2.2 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.

```
# 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.

7.3 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.
```

```r
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
```

```r
# 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:

```r
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

7.4 Deploying an API with `plumber`

7.4.1 Deploying a simple API

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`.

```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.