Learn R

…as you learnt your mother tongue

Learn R

…as you learnt your mother tongue

Git hash: c9bb91a; Git date: 2016-11-22 09:02:12 +0200

Pedro J. Aphalo

Helsinki, 22 November 2016

Draft, 85% done
Available through Leanpub

Typeset with XeLaTeX in Lucida Bright and Lucida Sans using the KOMA-Script book class.
The manuscript was written using R with package knitr. The manuscript was edited in WinEdt and RStudio. The source files for the whole book are available at https://bitbucket.org/aphalo/using-r.

Contents

Preface

> *Do not struggle, just play! If going gets difficult and frustrating, take a break! If you get a new insight, take a break to enjoy the victory!*
>
> — Learning like a child

This book covers different aspects of the use of R. They are meant to be used possibly as a complement to a course or book, as explanations are rather short and terse. I do not discuss here statistics, just Ras a tool and language for data manipulation and display. The idea is for you to learn the Rlanguage like children learn a language: they work-out what the rules are, simply by listening to people speak and trying to utter what they want to tell their parents. I do give some explanations and comments, but the idea of these notes is mainly for you to use the numerous examples to find-out by yourself the overall patterns and coding philosophy behind the Rlanguage. Instead of parents being the sound board for your first utterances in R, the computer will play this role. You should look and try to repeat the examples, and then try your own hand and see how the computer responds, does it understand you or not?

When teaching I tend to lean towards challenging students rather than telling a simple story. I do the same here, because it is what I prefer as a student, and how I learn best myself. Not everybody learns best with the same approach, for me the most limiting factor is for what I listen to, or read, to be in a way or another challenging or entertaining enough to keep my thoughts focused. This I achieve best when making an effort to understand the contents or to follow the thread of teh plot of a story. So, be warned, reading this book will be about exploring a new world, this book aims to be a travel guide, neither a traveler's account, nor a cookbook of R recipes.

Do not expect to ever know everything about R! Rin a broad sense is vast because its capabilities can be expanded with independently developed packages. Currently there are close to ten thousand packages available for free. You just need to learn what you need. Being Rvery popular there is nowadays lots of information available, plus a helpful and open minded on-line community willing to help with those difficult problems for which Goggle will not be of help.

How to read this book? My idea is that you will run all the code ex-

amples and try as many other variations as needed until you are sure to understand the basic 'rules' of the R language and how each function or command described works. In R for each function, data set, etc. there is a help page available. In addition, if you use a front-end like RStudio, auto-completion is available as well as balloon help on the arguments accepted by functions. For scripts, there is syntax checking of the source code before its execution: *possible* mistakes and even formatting style problems are highlighted in the editor window. Error messages tend to be terse in R, and may require some lateral thinking and/or 'experimentation' to understand the real cause behind problems. When you are not sure to understand how some command works, it is useful in many cases to try simple examples for which you know the correct answer and see if you can reproduce them in R.

I recommend you to use as an editor or IDE (integrated development environment) RStudio. RStudio is user friendly, actively maintained, and available both in desktop and server versions. The desktop version runs on Windows, Linux, and OS X and other Unixes. In addition it is available for free! R itself also runs under all these operating systems and a few more. Being R a command line application in its simplest incarnation, it can be made to work on what nowadays are frugal computing resources equivalent to a personal computer of a couple of decades ago. Nowadays R can be made to run even on the Raspberry Pi, a Linux micro-controller board with the processing power of a modest smartphone. At the other end of the spectrum on really powerful servers it can be used for the analysis of big data sets with millions of observations. How powerful a computer you will need will depend on the size of the data sets to analyze and on how patient you are.

When searching for answers, asking for advice or reading books you will be confronted with different ways of doing the same tasks. Do not this overwhelm you, in most cases it will not matter as many computations can be done in R, as in any language, in several different ways, still obtaining the same result. The different approaches may differ mainly in two aspects: 1) how readable to humans are the instructions given to the computer as part of a script or program, and 2) how fast the code will run. Unless performance is an important bottleneck in your work, just concentrate on writing code that is easy to understand to you and to others, and consequently easy to check and reuse. Of course do always check any code you write for mistakes, preferably using actual numerical test cases for any complex calculation or even relatively simple scripts. Testing and validation are extremely important steps in data analysis, so get into this habit while reading this book. Testing how every function works as I will challenge you to do in this book, is at the core of any robust data analysis

or computing programming. In addition, when writing computer code, as for any other text intended for humans to read, consistent writing style and formatting go a long way in making your intentions clear.

These notes are work-in-progress. I will appreciate suggestions for further examples, notification of errors and unclear sections and also any larger contributions. Many of the examples here have been collected from diverse sources over many years and because of this not all sources are acknowledged. If you recognize any example as yours or someone else's please let me know so that I can add a proper acknowledgement. I warmly thank the students that over the years have asked the questions and posed the problems that have helped me write this text and correct the mistakes and voids of previous versions. I have also received help on on-line forums and in person from numerous people, learnt from archived e-mail list messages, blog posts, books, articles, tutorials, webinars, and by struggling to solve some new problems on my own. In many ways this text owes much more to people who are not authors than to myself. However, as I am the one who has written this version and decided what to include and exclude, as author, I take full responsibility for any errors and inaccuracies.

I have been using Rsince around 1998 or 1999, but I am still constantly learning new things about Ritself and Rpackages. With time it has replaced in my work as a researcher and teacher several other pieces of software: SPSS, Systat, Origin, Excel, and it has become a central piece of the tool set I use for producing lecture slides, notes, books and even web pages. This is to say that it is the most useful piece of software and programming language I have ever learnt to use. Of course, in time it will be replaced by something better, but at the moment it is the "hot" thing to learn for anybody with a need to analyse and display data.

Status as of 2016-11-19. I have updated the manuscript to track package updates since the previous version uploaded nearly five months ago, and added some examples of the new functionality added to packages 'ggpmisc', 'ggrepel', and 'ggplot2'. I have written new sections on packages 'viridis', 'gganimate', 'ggstance', 'ggbiplot', and 'ggalt'.

With respect to the chapter *Storing and manipulating data with R* I have put it on hold, except for the introduction, until I can see a soon to be published book covering the same subject. Hadley Wickham has named the set of tools developed by him and his collaborators as *tidyverse* to be described in the book titled *R for Data Science* by Grolemund and Wickham (O'Reilly).

An important update to ‘ggplot2’was released last week, and it includes changes to the behavior of some existing functions, specially faceting has become extensible through other packages. I will soon write the sections on faceting based on this new version of ‘ggplot2’.

I expect to upload the update to this manuscript in one or two months time.

1 Introduction

> *Although I whined and tried to hide under the rug, my inexorable publisher demanded an introduction…*
>
> — Ursula K. Le Guin
> *Buffalo Gals and other Animal Presences*, 1985

1.1 R's built-in help

To access help pages through the command prompt we use function `help()` or a question mark. Every object exported by an R package (functions, methods, classes, data) is documented. Sometimes a single help page documents several R objects. Usually at the end of the help pages some us examples are given.

```
help("sum")
?sum
```

1.2 Obtaining help from on-line forums

Netiquette
  StackOverflow

1.3 Online webbinars and courses

1.4 R, editors and IDEs

1.5 Packages and repositories

1.6 Other tools

2 R as a powerful calculator

> *The desire to economize time and mental effort in arithmetical computations, and to eliminate human liability to error, is probably as old as the science of arithmetic itself.*
>
> — Howard Aiken, *Proposed automatic calculating machine*, presented to IBM in 1937

2.1 Aims of this chapter

In my experience, for those not familiar with computing programming or scripting languages, and who have mostly used computer programs through visual interfaces making heavy use of menus and icons, the best first step in learning R is to learn the basics of the language through its use at the R command prompt. This will teach not only the syntax and grammar rules, but also give a glimpse at the advantages and flexibility of this approach to data analysis.

Menu-driven programs are not necessarily bad, they are just unsuitable when there is a need to set very many options and chose from many different actions. They are also difficult to maintain when extensibility is desired, and when independently developed modules of very different characteristics need to be integrated. Textual languages also have the advantage, to be dealt with in the next chapter, that command sequences can be stored as a human- and computer readable text file that keeps a record of all the steps used and that in most cases makes it trivial to reproduce the same steps at a later time. The scripts are also a very simple and handy way of communicating to others how to do a given data analysis.

2.2 Working at the R console

I assume here that you have installed or have had installed by someone else R and RStudio and that you are already familiar enough with RStudio to find your way around its user interface. The examples in this chapter use only the console window, and results are printed to the console. The values stored in the different variables are visible in the Environment tab in RStudio.

In the console you can type commands at the > prompt. When you end a line by pressing the return key, if the line can be interpreted as an R command, the result will be printed in the console, followed by a new > prompt. If the command is incomplete a + continuation prompt will be shown, and you will be able to type-in the rest of the command. For example if the whole calculation that you would like to do is $1 + 2 + 4$, if you enter in the console `1 + 2 +` in one line, you will get a continuation prompt where you will be able to type 3. However, if you type `1 + 2`, the result will be calculated, and printed.

When working at the command prompt, results are printed by default, but in other cases you may need to use the function `print` explicitly. The examples here rely on the automatic printing.

The idea with these examples is that you learn by working out how different commands work based on the results of the example calculations listed. The examples are designed so that they allow the rules, and also a few quirks, to be found by 'detective work'. This should hopefully lead to better understanding than just studying rules.

2.3 Examples with numbers

When working with arithmetic expressions the normal mathematical precedence rules are respected, but parentheses can be used to alter this order. Parentheses can be nested and at all nesting levels the normal rounded parentheses are used. The number of opening (left side) and closing (right side) parentheses must be balanced, and they must be located so that each enclosed term is a valid mathematical expression. For example while `(1 + 2) * 3` is valid, `(1 +) 2 * 3` is a syntax error as `1 +` is incomplete and cannot be calculated.

```
1 + 1
```

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

```
2 * 2
```

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

```
2 + 10 / 5
```

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

```
(2 + 10) / 5
```

```
## [1] 2.4
```

```
10^2 + 1
```

```
## [1] 101
```

```
sqrt(9)
```

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

```
pi # whole precision not shown when printing
```

```
## [1] 3.141593
```

```
print(pi, digits=22)
```

```
## [1] 3.1415926535897931
```

```
sin(pi) # oops! Read on for explanation.
```

```
## [1] 1.224606e-16
```

```
log(100)
```

```
## [1] 4.60517
```

```
log10(100)
```

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

```
log2(8)
```

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

```
exp(1)
```

```
## [1] 2.718282
```

One can use variables to store values. The 'usual' assignment operator is `<-`. Variable names and all other names in R are case sensitive. Variables `a` and `A` are two different variables. Variable names can be quite long, but usually it is not a good idea to use very long names. Here I am using very short names, that is usually a very bad idea. However, in cases like these examples where the stored values have no real connection to the real world and are used just once or twice, these names emphasize the abstract nature.

```
a <- 1
a + 1
```

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

```
a

## [1] 1

b <- 10
b <- a + b
b

## [1] 11

3e-2 * 2.0

## [1] 0.06
```

There are some syntactically legal statements that are not very frequently used, but you should be aware that they are valid, as they will not trigger error messages, and may surprise you. The important thing is that you write commands consistently. The assignment 'backwards' assignment operator `->` resulting code like `1 -> a` is valid but rarely used. The use of the equals sign (`=`) for assignment although valid is generally discouraged as it is seldom used as this meaning has not earlier been part of the Rlanguage. Chaining assignments as in the first line below is sometimes used, and signals to the human reader that `a`, `b` and `c` are being assigned the same value.

```
a <- b <- c <- 0.0
a

## [1] 0

b

## [1] 0

c

## [1] 0

1 -> a
a

## [1] 1

a = 3
a

## [1] 3
```

Numeric variables can contain more than one value. Even single numbers are `vector` s of length one. We will later see why this is important. As you have seen above the results of calculations were printed preceded with `[1]`. This is the index or position in the vector of the first number (or other value) displayed at the head of the current line.

One can use `c` 'concatenate' to create a vector of numbers from individual numbers.

```
a <- c(3,1,2)
a

## [1] 3 1 2

b <- c(4,5,0)
b

## [1] 4 5 0

c <- c(a, b)
c

## [1] 3 1 2 4 5 0

d <- c(b, a)
d

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

One can also create sequences using `seq` , or repeat values. In this case I leave to the reader to work out the rules by running these and his/her own examples.

```
a <- -1:5
a

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

b <- 5:-1
b

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

c <- seq(from = -1, to = 1, by = 0.1)
c

##  [1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2
## [10] -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7
## [19]  0.8  0.9  1.0

d <- rep(-5, 4)
d

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

Now something that makes Rdifferent from most other programming languages: vectorized arithmetic.

```
a + 1 # we add one to vector a defined above

## [1] 0 1 2 3 4 5 6

(a + 1) * 2

## [1]  0  2  4  6  8 10 12

a + b

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

a - a

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

It can be seen in first line above, another peculiarity of R, that is frequently called "recycling": as vector a is of length 6, but the constant 1 is a vector of length 1, this 1 is extended by recycling into a vector of the same length as the longest vector in the statement.

Make sure you understand what calculations are taking place in the chunk above, and also the one below.

```
a <- rep(1, 6)
a

## [1] 1 1 1 1 1 1

a + 1:2

## [1] 2 3 2 3 2 3

a + 1:3

## [1] 2 3 4 2 3 4

a + 1:4

## Warning in a + 1:4: longer object length is not a multiple of shorter
object length

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

A useful thing to know: a vector can have length zero. Vectors of length zero may seem at first sight quite useless, but in fact they are very useful. They allow the handling of "no input" or "nothing to do" cases as normal

cases, which in the absence of vectors of length zero would require to be treated as special cases. We also introduce here two useful functions, `length()` which returns the length of a vector, and `is.numeric()` that can be used to test if an Robject is `numeric`.

```
z <- numeric(0)
z

## numeric(0)

length(z)

## [1] 0

is.numeric(z)

## [1] TRUE
```

It is possible to *remove* variables from the workspace with `rm`. Function `ls()` returns a list all objects in the current environment, or by supplying a `pattern` argument, only the objects with names matching the `pattern`. The pattern is given as a regular expression, with `[]` enclosing alternative matching characters, `^` and `$` indicating the extremes of the name (start and end, respectively). For example `"^z$"` matches only the single character 'z' while `"^z"` matches any name starting with 'z'. In contrast `"^[zy]$"` matches both 'z' and 'y' but neither 'zy' nor 'yz', and `"^[a-z]"` matches any name starting with a lower case ASCII letter. If you are using RStudio, all objects are listed in the Environment pane, and the search box of the panel can be used to find a given object.

```
ls(pattern="^z$")

## [1] "z"

rm(z)
try(z)
ls(pattern="^z$")

## character(0)
```

There are some special values available for numbers. `NA` meaning 'not available' is used for missing values. Calculations can yield also the following values `NaN` 'not a number', `Inf` and `-Inf` for ∞ and $-\infty$. As you will see below, calculations yielding these values do **not** trigger errors or warnings, as they are arithmetically valid. `Inf` and `-Inf` are also valid numerical values for input and constants.

```
a <- NA
a

## [1] NA

-1 / 0

## [1] -Inf

1 / 0

## [1] Inf

Inf / Inf

## [1] NaN

Inf + 4

## [1] Inf

b <- -Inf
b * -1

## [1] Inf
```

Not available (`NA`) values are very important in the analysis of experimental data, as frequently some observations are missing from an otherwise complete data set due to "accidents" during the course of an experiment. It is important to understand how to interpret `NA` 's. They are simple place holders for something that is unavailable, in other words *unknown*. Any operation, even tests of equality, involving one or more `NA` 's return an `NA` . In other words when one input to a calculation is unknown, the result of the calculation is unknown.

```
A <- NA
A

## [1] NA

A + 1

## [1] NA

A + Inf

## [1] NA
```

One thing to be aware of, and which we will discuss again later, is that

numbers in computers are almost always stored with finite precision. This means that they not always behave as Real numbers as defined in mathematics. In Rthe usual numbers are stored as `double-precision floats`, which means that there are limits to the largest and smallest numbers that can be represented (approx. $-1 \cdot 10^{308}$ and $1 \cdot 10^{308}$), and the number of significant digits that can be stored (usually described as ϵ (epsilon, abbreviated `eps`, defined as the largest number for which $1 + \epsilon = 1$)). This can be sometimes important, and can generate unexpected results in some cases, especially when testing for equality. In the example below, the result of the subtraction is still exactly 1.

```
1 - 1e-20

## [1] 1
```

It is usually safer not to test for equality to zero when working with numeric values. One alternative is comparing against a suitably small number, which will depend on the situation, although `eps` is usually a safe bet, unless the expected range of values is known to be small. This type of precautions are specially important in what is usually called "production" code: a script or program that will be used many times and with little further intervention by the researcher or programmer. Such code must work correctly, or not work at all, and it should not under any imaginable circumstance possibly give a wrong answer.

```
eps <- .Machine$double.eps
abs(-1)

## [1] 1

abs(1)

## [1] 1

x <- 1e-40
abs(x) < eps * 2

## [1] TRUE

abs(x) < 1e-100

## [1] FALSE
```

The same precautions apply to tests for equality, so whenever possible according to the logic of the calculations, it is best to test for inequalities, for example using `x <= 1.0` instead of `x == 1.0`. If this is not possible,

then the tests should be treated as above, for example replacing `x == 1.0` with `abs(x - 1.0) < eps`. Function `abs()` returns the absolute value, in simple words, makes all values positive or zero, by changing the sign of negative values.

When comparing integer values these problems do not exist, as integer arithmetic is not affected by loss of precision in calculations restricted to integers (the `L` comes from 'long' a name sometimes used for a machine representation of integers. Because of the way integers are stored in the memory of computers, within the acceptable range, they are stored exactly. One can think of computer integers as a subset of whole numbers restricted to a certain range of values.

```
1L + 3L
```

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

```
1L * 3L
```

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

```
1L %/% 3L
```

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

```
1L %% 3L
```

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

```
1L / 3L
```

```
## [1] 0.3333333
```

The last statement in example immediately above, using the 'usual' division operator yields a floating-point `double` result, while the integer division operator %/% yields an `integer` result, and %% returns the remainder from the integer division.

Both doubles and integers are considered numeric. In most situations conversion is automatic and we do not need to worry about the differences between these two types of numeric values. This last chunk shows values returned that are either `TRUE` or `FALSE`. These are `logical` values that will be discussed in the next section.

```
is.numeric(1L)
```

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

```
is.double(1L)

## [1] FALSE

is.double(1L / 3L)

## [1] TRUE

is.numeric(1L / 3L)

## [1] TRUE
```

2.4 Examples with logical values

What in maths are usually called Boolean values, are called `logical` values in R. They can have only two values `TRUE` and `FALSE`, in addition to `NA` (not available). They are vectors as all other simple types in R. There are also logical operators that allow Boolean algebra (and support for set operations that we will only describe very briefly). In the chunk below we work with logical vectors of length one.

```
a <- TRUE
b <- FALSE
a

## [1] TRUE

!a # negation

## [1] FALSE

a && b # logical AND

## [1] FALSE

a || b # logical OR

## [1] TRUE
```

Again vectorization is possible. I present this here, and will come back to this later, because this is one of the most troublesome aspects of the Rlanguage for beginners. There are two types of 'equivalent' logical operators that behave differently, but use similar syntax! The vectorized operators have single-character names & and |, while the non vectorized ones have double-character names && and ||. There is only one version

of the negation operator ! that is vectorized. In some, but not all cases, a warning will indicate that there is a possible problem.

```
a <- c(TRUE,FALSE)
b <- c(TRUE,TRUE)
a

## [1]  TRUE FALSE

b

## [1] TRUE TRUE

a & b # vectorized AND

## [1]  TRUE FALSE

a | b # vectorized OR

## [1] TRUE TRUE

a && b # not vectorized

## [1] TRUE

a || b # not vectorized

## [1] TRUE
```

Functions `any` and `all` take a logical vector as argument, and return a single logical value 'summarizing' the logical values in the vector. `all` returns `TRUE` only if every value in the argument is `TRUE`, and `any` returns `TRUE` unless every value in the argument is `FALSE`.

```
any(a)

## [1] TRUE

all(a)

## [1] FALSE

any(a & b)

## [1] TRUE

all(a & b)

## [1] FALSE
```

Another important thing to know about logical operators is that they 'short-cut' evaluation. If the result is known from the first part of the statement, the rest of the statement is not evaluated. Try to understand what happens when you enter the following commands. Short-cut evaluation is useful, as the first condition can be used as a guard preventing a later condition to be evaluated when its computation would result in an error (and possibly abort of the whole computation).

```
TRUE || NA
```

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

```
FALSE || NA
```

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

```
TRUE && NA
```

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

```
FALSE && NA
```

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

```
TRUE && FALSE && NA
```

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

```
TRUE && TRUE && NA
```

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

When using the vectorized operators on vectors of length greater than one, 'short-cut' evaluation still applies for the result obtained.

```
a & b & NA
```

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

```
a & b & c(NA, NA)
```

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

```
a | b | c(NA, NA)
```

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

2.5 Comparison operators

Comparison operators yield as a result logical values.

```
1.2 > 1.0

## [1] TRUE

1.2 >= 1.0

## [1] TRUE

1.2 == 1.0 # be aware that here we use two = symbols

## [1] FALSE

1.2 != 1.0

## [1] TRUE

1.2 <= 1.0

## [1] FALSE

1.2 < 1.0

## [1] FALSE

a <- 20
a < 100 && a > 10

## [1] TRUE
```

Again these operators can be used on vectors of any length, returning as result a logical vector.

```
a <- 1:10
a > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

a < 5

##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

a == 5

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

```
all(a > 5)

## [1] FALSE

any(a > 5)

## [1] TRUE

b <- a > 5
b

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

any(b)

## [1] TRUE

all(b)

## [1] FALSE
```

Be once more aware of 'short-cut evaluation'. If the result would not be affected by the missing value then the result is returned. If the presence of the NA makes the end result unknown, then NA is returned.

```
c <- c(a, NA)
c > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE    NA

all(c > 5)

## [1] FALSE

any(c > 5)

## [1] TRUE

all(c < 20)

## [1] NA

any(c > 20)

## [1] NA

is.na(a)
```

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

is.na(c)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE  TRUE

any(is.na(c))

## [1] TRUE

all(is.na(c))

## [1] FALSE
```

This behaviour can be changed by using the optional argument `na.rm` which removes NA values **before** the function is applied. (Many functions in Rhave this optional parameter.)

```
all(c < 20)

## [1] NA

any(c > 20)

## [1] NA

all(c < 20, na.rm=TRUE)

## [1] TRUE

any(c > 20, na.rm=TRUE)

## [1] FALSE
```

You may skip until the end of the section on first read, also see page 11. Here are some examples for which the finite resolution of computer machine floats as compared to Real numbers as defined in mathematics makes a difference.

```
1e20 == 1 + 1e20
## [1] TRUE
1 == 1 + 1e-20
## [1] TRUE
0 == 1e-20
## [1] FALSE
```

As Rcan run on different types of computer hardware, the actual machine limits may vary. It is possible to obtain this values from variable `.Machine`.

```
.Machine$double.eps

## [1] 2.220446e-16

.Machine$integer.max

## [1] 2147483647
```

In many situations, when writing programs one should avoid testing for equality of floating point numbers ('floats'). Here we show how to handle gracefully rounding errors. As the example shows, some rounding errors may accumulate, and in practice `.Machine$double.eps` may be too large a value to safely use in tests for zero.

```
a == 0.0 # may not always work

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

abs(a) < 1e-15 # is safer

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

sin(pi) == 0.0 # angle in radians, not degrees!

## [1] FALSE

sin(2 * pi) == 0.0

## [1] FALSE

abs(sin(pi)) < 1e-15

## [1] TRUE

abs(sin(2 * pi)) < 1e-15

## [1] TRUE

sin(pi)

## [1] 1.224606e-16

sin(2 * pi)

## [1] -2.449213e-16

.Machine$double.eps # see help for .Machine for explanation

## [1] 2.220446e-16

.Machine$double.neg.eps

## [1] 1.110223e-16
```

2.6 Character values

Character variables can be used to store any character. Character constants are written by enclosing characters in quotes. There are three types of quotes in the ASCII character set, double quotes ", single quotes ', and back ticks `. The first two types of quotes can be used for delimiting characters. There are in Rtwo predefined vectors with characters for letters stored in alphabetical order.

```
a <- "A"
b <- letters[2]
c <- letters[1]
a

## [1] "A"

b

## [1] "b"

c

## [1] "a"

d <- c(a, b, c)
d

## [1] "A" "b" "a"

e <- c(a, b, "c")
e

## [1] "A" "b" "c"

h <- "1"
try(h + 2)
```

Vectors of characters are not the same as character strings. In character vectors each position in the vector is occupied by a single character, while in character strings, each string of characters, like a word enclosed in double or single quotes occupies a single position or slot in the vector.

```
f <- c("1", "2", "3")
g <- "123"
f == g

## [1] FALSE FALSE FALSE

f
```

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

g

## [1] "123"
```

One can use the 'other' type of quotes as delimiter when one wants to include quotes within a string. Pretty-printing is changing what I typed into how the string that is stored in R: I typed `b <- 'He said "hello" when he came in'` in the second statement below, try it.

```
a <- "He said 'hello' when he came in"
a

## [1] "He said 'hello' when he came in"

b <- 'He said "hello" when he came in'
b

## [1] "He said \"hello\" when he came in"
```

The outer quotes are not part of the string, they are 'delimiters' used to mark the boundaries. As you can see when `b` is printed special characters can be represented using 'escape sequences'. There are several of them, and here we will show just two, newline and tab. We also show here the different behaviour of `print()` and `cat()`, with `cat()` *interpreting* the escape sequences and `print()` not.

```
c <- "abc\ndef\txyz"
print(c)

## [1] "abc\ndef\txyz"

cat(c)

## abc
## def xyz
```

Above, you will not see any effect of these escapes when using `print`: `\n` represents 'new line' and `\t` means 'tab' (tabulator). The *scape codes* work only in some contexts, as when using `cat` to generate the output. They also are very useful when one wants to split an axis-label, title or label in a plot into two or more lines as they can be embedded in any string.

2.7 Finding the 'mode' of objects

Variables have *mode* that depends on what can be stored in them. But differently to other languages, assignment of to variable of a different mode is allowed and in most cases its mode changes together with its contents. However, there is a restriction that all elements in a vector, array or matrix, must be of the same mode, while this is not required for lists. Functions with names starting with `is.` are tests returning a logical value, `TRUE`, `FALSE` or `NA`.

```
my_var <- 1:5
mode(my_var)

## [1] "numeric"

is.numeric(my_var)

## [1] TRUE

is.logical(my_var)

## [1] FALSE

is.character(my_var)

## [1] FALSE

my_var <- "abc"
mode(my_var)

## [1] "character"
```

2.8 Type conversions

The least intuitive ones are those related to logical values. All others are as one would expect. By convention, functions used to convert objects from one mode to a different one have names starting with `as.`.

```
as.character(1)

## [1] "1"

as.character(3.0e10)

## [1] "3e+10"
```

```r
as.numeric("1")
## [1] 1
as.numeric("5E+5")
## [1] 5e+05
as.numeric("A")
## Warning: NAs introduced by coercion
## [1] NA
as.numeric(TRUE)
## [1] 1
as.numeric(FALSE)
## [1] 0
TRUE + TRUE
## [1] 2
TRUE + FALSE
## [1] 1
TRUE * 2
## [1] 2
FALSE * 2
## [1] 0
as.logical("T")
## [1] TRUE
as.logical("t")
## [1] NA
as.logical("TRUE")
## [1] TRUE
as.logical("true")
## [1] TRUE
as.logical(100)
## [1] TRUE
as.logical(0)
## [1] FALSE
as.logical(-1)
## [1] TRUE
```

```
f <- c("1", "2", "3")
g <- "123"
as.numeric(f)

## [1] 1 2 3

as.numeric(g)

## [1] 123
```

Some tricks useful when dealing with results. Be aware that the printing is being done by default, these functions return numerical values that are different from their input. Look at the help pages for further details. Very briefly `round` is used to round numbers to a certain number of decimal places after or before the decimal point, while `signif()` keeps the requested number of significant digits.

```
round(0.0124567, 3)

## [1] 0.012

round(0.0124567, 1)

## [1] 0

round(0.0124567, 5)

## [1] 0.01246

signif(0.0124567, 3)

## [1] 0.0125

round(1789.1234, 3)

## [1] 1789.123

signif(1789.1234, 3)

## [1] 1790

a <- 0.12345
b <- round(a, 2)
a == b

## [1] FALSE

a - b

## [1] 0.00345

b

## [1] 0.12
```

When applied to vectors, `signif` behaves slightly differently.

```
signif(c(123, 0.123), 3)

## [1] 123.000   0.123
```

Other functions relevant to the formatting of numbers and other output are `format()`, and `sprintf()`.

2.9 Vectors

You already know how to create a vector. Now we are going to see how to extract individual elements (e.g. numbers or characters) out of a vector. Elements are accessed using an index. The index indicates the position in the vector, starting from one, following the usual mathematical tradition. What in maths would be x_i for a vector x, in R is represented as `x[i]`. (In R indexes (or subscripts) always start from one, while in some other programming languages indexes start from zero.)

```
a <- letters[1:10]
a

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[2]

## [1] "b"

a[c(3,2)]

## [1] "c" "b"

a[10:1]

##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

The examples below demonstrate what is the result of using a longer vector of indexes than the indexed vector. The length of the indexing vector has no restriction, but the acceptable range of values for the indexes is given by the length of the indexed vector.

```
a[c(3,3,3,3)]

## [1] "c" "c" "c" "c"

a[c(10:1, 1:10)]
```

```
##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a" "a"
## [12] "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Negative indexes have a special meaning, they indicate the positions at which values should be excluded.

```
a[-2]

## [1] "a" "c" "d" "e" "f" "g" "h" "i" "j"

a[-c(3,2)]

## [1] "a" "d" "e" "f" "g" "h" "i" "j"
```

Results from indexing with out-of-range values may be surprising.

```
a[11]

## [1] NA

a[1:11]

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" NA
```

Results from indexing with special values may be surprising.

```
a[ ]

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[numeric(0)]

## character(0)

a[NA]

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

a[c(1, NA)]

## [1] "a" NA

a[NULL]

## character(0)

a[c(1, NULL)]

## [1] "a"
```

Another way of indexing, which is very handy, but not available in most other programming languages, is indexing with a vector of logical values. In practice, the vector of logical values used for 'indexing' is in most cases of the same length as the vector from which elements are going to be selected. However, this is not a requirement, and if the logical vector is shorter it is 'recycled' as discussed above in relation to operators.

```
a[TRUE]

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[FALSE]

## character(0)

a[c(TRUE, FALSE)]

## [1] "a" "c" "e" "g" "i"

a[c(FALSE, TRUE)]

## [1] "b" "d" "f" "h" "j"

a > "c"

##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

a[a > "c"]

## [1] "d" "e" "f" "g" "h" "i" "j"

selector <- a > "c"
a[selector]

## [1] "d" "e" "f" "g" "h" "i" "j"

which(a > "c")

## [1]  4  5  6  7  8  9 10

indexes <- which(a > "c")
a[indexes]

## [1] "d" "e" "f" "g" "h" "i" "j"

b <- 1:10
b[selector]

## [1]  4  5  6  7  8  9 10

b[indexes]

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

Make sure to understand the examples above. These type of constructs are very widely used in Rscripts because they allow for concise code that is easy to understand once you are familiar with the indexing rules.

Indexing can be used on both sides of an assignment. This may look rather esoteric at first sight, but it is just a simple extension of the logic of indexing described above.

```
a <- 1:10
a

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

a[1] <- 99
a

##  [1] 99  2  3  4  5  6  7  8  9 10

a[c(2,4)] <- -99
a

##  [1]  99 -99   3 -99   5   6   7   8   9  10

a[TRUE] <- 1
a

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

a <- 1
```

We can also have subscripting on both sides.

```
a <- letters[1:10]
a

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[1] <- a[10]
a

##  [1] "j" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a <- a[10:1]
a

##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "j"

a[10:1] <- a
a

##  [1] "j" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[5:1] <- a[c(TRUE,FALSE)]
a

##  [1] "i" "g" "e" "c" "j" "f" "g" "h" "i" "j"
```

Do play with subscripts to your heart's content, really grasping how they work and can how they can be used will be very useful in anything you do in the future with R.

2.10 Factors

Factors are used for indicating categories, most frequently the factors describing the treatments in an experiment, or categories in a survey. They can be created either from numerical or character vectors. The different possible values are called *levels*. Normal factors created with `factor` are unordered or categorical. Ralso defines ordered factors that can be created with function `ordered`.

```
my.vector <- c("treated", "treated", "control", "control", "control", "treated")
my.factor <- factor(my.vector)
my.factor <- factor(my.vector, levels=c("treatment", "control"))
```

It is always preferable to use meaningful names for levels, although it is possible to use numbers. The order of levels becomes important when plotting data, as it affects the order of the levels along the axes, or in legends. Converting factors to numbers is not intuitive, because even if the levels look like numbers when displayed, they are just character strings.

```
my.vector2 <- rep(3:5, 4)
my.factor2 <- factor(my.vector2)
as.numeric(my.factor2)

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

as.numeric(as.character(my.factor2))

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

Internally factor levels are stored as running numbers starting from one, and those are the numbers returned by `as.numeric()` when applied to a factor.

Factors are very important in R. In contrast to other statistical software in which the role of a variable is set when defining a model to be fitted or setting up a test, in Rmodels are specified exactly in the same way for ANOVA and regression analysis, as linear models. What 'decides' what type of model is fitted is whether the explanatory variable is a factor (giving ANOVA) or a numerical variable (giving regression). This makes a lot of

sense, as in most cases, considering an explanatory variable as categorical or not, depends on the design of the experiment or survey, in other words, is a property of the data rather than of the analysis.

2.11 Lists

Elements of a `list` are not ordered, and can be of different type. Lists can be also nested. Elements in list are named, and normally are accessed by name. Lists are defined using function `list`.

```
a.list <- list(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.list

## $x
## [1] 1 2 3 4 5 6
##
## $y
## [1] "a"
##
## $z
## [1]  TRUE FALSE

str(a.list)

## List of 3
##  $ x: int [1:6] 1 2 3 4 5 6
##  $ y: chr "a"
##  $ z: logi [1:2] TRUE FALSE

a.list$x

## [1] 1 2 3 4 5 6

a.list[["x"]]

## [1] 1 2 3 4 5 6

a.list[[1]]

## [1] 1 2 3 4 5 6

a.list["x"]

## $x
## [1] 1 2 3 4 5 6

a.list[1]

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

```
a.list[c(1,3)]

## $x
## [1] 1 2 3 4 5 6
##
## $z
## [1]  TRUE FALSE

try(a.list[[c(1,3)]])

## [1] 3
```

Using double square brackets for indexing gives the element stored in the list, in its original mode, in the example above, `a.list[["x"]]` returns a numeric vector, while `a.list[1]` returns a list containing the numeric vector `x`. `a.list$x` returns the same value as `a.list[["x"]]`, a numeric vector. While `a.list[c(1,3)]` returns a list of length two, `a.list[[c(1,3)]]`.

2.12 Data frames

Data frames are a special type of list, in which each element is a vector or a factor of the same length. The are created with function `data.frame` with a syntax similar to that used for lists. When a shorter vector is supplied as argument, it is recycled, until the full length of the variable is filled. This is very different to what we obtained in the previous section when we created a list.

```
a.df <- data.frame(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.df

##   x y     z
## 1 1 a  TRUE
## 2 2 a FALSE
## 3 3 a  TRUE
## 4 4 a FALSE
## 5 5 a  TRUE
## 6 6 a FALSE

str(a.df)

## 'data.frame': 6 obs. of  3 variables:
##  $ x: int  1 2 3 4 5 6
##  $ y: Factor w/ 1 level "a": 1 1 1 1 1 1
##  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE
```

```
a.df$x

## [1] 1 2 3 4 5 6

a.df[["x"]]

## [1] 1 2 3 4 5 6

a.df[[1]]

## [1] 1 2 3 4 5 6

class(a.df)

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

R is an object oriented language, and objects belong to classes. With function `class` we can query the class of an object. As we saw in the two previous chunks lists and data frames objects belong to two different classes.

We can add also to lists and data frames.

```
a.df$x2 <- 6:1
a.df$x3 <- "b"
a.df

##   x y     z x2 x3
## 1 1 a  TRUE  6  b
## 2 2 a FALSE  5  b
## 3 3 a  TRUE  4  b
## 4 4 a FALSE  3  b
## 5 5 a  TRUE  2  b
## 6 6 a FALSE  1  b
```

We have added two columns to the data frame, and in the case of column `x3` recycling took place. Data frames are extremely important to anyone analysing or plotting data in R. One can think of data frames as tightly structured work-sheets, or as lists. As you may have guessed from the examples earlier in this section, there are several different ways of accessing columns, rows, and individual observations stored in a data frame. The columns can to some extent be treated as elements in a list, and can be accessed both by name or index (position). When accessed by name, using `$` or double square brackets a single column is returned as a vector or factor. In contrast to lists, data frames are 'rectangular' and for this reason the values stored can be also accessed in a way similar to how elements in a matrix are accessed, using two indexes. As we saw for vectors indexes can be vectors of integer numbers or vectors of logical

values. For columns they can in addition be vectors of character strings matching the names of the columns. When using indexes it is extremely important to remember that the indexes are always given **row first**.

```
a.df[ , 1]   # first column

## [1] 1 2 3 4 5 6

a.df[ , "x"] # first column

## [1] 1 2 3 4 5 6

a.df[1, ]    # first row

##   x y    z x2 x3
## 1 1 a TRUE  6  b

a.df[1:2, c(FALSE, FALSE, TRUE, FALSE, FALSE)]

## [1]  TRUE FALSE

             # first two rows of the third column
a.df[a.df$z , ] # the rows for which z is true

##   x y    z x2 x3
## 1 1 a TRUE  6  b
## 3 3 a TRUE  4  b
## 5 5 a TRUE  2  b

a.df[a.df$x > 3, -3] # the rows for which x > 3 for

##   x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b

                 # all columns except the third one
```

When the names of data frames are long, complex conditions become awkward to write. In such cases `subset` is handy because evaluation is done in the 'environment' of the data frame, i.e. the names of the columns are recognized if entered directly.

```
subset(a.df, x > 3)

##   x y     z x2 x3
## 4 4 a FALSE  3  b
## 5 5 a  TRUE  2  b
## 6 6 a FALSE  1  b
```

When calling functions that return a vector, data frame, or other structure, the square brackets can be appended to the rightmost parenthesis of the function call, in the same way as to the name of a variable holding the same data.

```
subset(a.df, x > 3)[ , -3]

##   x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b

subset(a.df, x > 3)$x

## [1] 4 5 6
```

None of the examples in the last three code chunks alter the original data frame `a.df`. We can store the returned value using a new name, if we want to preserve `a.df` unchanged, or we can assign the result to `a.df` deleting in the process the original `a.df`. The next to examples do assignment to `a.df`, but either to only one columns, or by indexing the individual values in both the 'right side' and 'left side' of the assignment. Another way to delete a column from a data frame is to assign `NULL` to it.

```
a.df[["x2"]] <- NULL
a.df$x3 <- NULL
a.df

##   x y     z
## 1 1 a  TRUE
## 2 2 a FALSE
## 3 3 a  TRUE
## 4 4 a FALSE
## 5 5 a  TRUE
## 6 6 a FALSE
```

In the previous code chuck we deleted the last two columns of the data frame `a.df`. Finally an esoteric trick for you think about.

```
a.df[1:6, c(1,3)] <- a.df[6:1, c(3,1)]
a.df

##   x y z
## 1 0 a 6
## 2 1 a 5
## 3 0 a 4
```

```
## 4 1 a 3
## 5 0 a 2
## 6 1 a 1
```

Although in this last example we used numeric indexes to make in more interesting, in practice, especially in scripts or other code that will be reused, do use column names instead of positional indexes. This makes your code much more reliable, as changes elsewhere in the script are much less likely to lead to undetected errors.

2.13 Simple built-in statistical functions

Being R's main focus in statistics, it provides functions for both simple and complex calculations, going from means and variances to fitting very complex models. we will start with the simple ones.

```
x <- 1:20
mean(x)

## [1] 10.5

var(x)

## [1] 35

median(x)

## [1] 10.5

mad(x)

## [1] 7.413

sd(x)

## [1] 5.91608

range(x)

## [1]  1 20

max(x)

## [1] 20

min(x)

## [1] 1

length(x)

## [1] 20
```

2.14 Functions and execution flow control

Although functions can be defined and used at the command prompt, we will discuss them when looking at scripts in the next chapter. We will do the same in the case of flow-control statements (e.g. repetition and conditional execution).

Significance tests and model fitting will be the subject of later chapters, not yet written.

3 R Scripts and Programming

> *An R script is simply a text file containing (almost) the same commands that you would enter on the command line of R.*
>
> — Kickstarting R

3.1 What is a script?

We call *script* to a text file that contains the same commands that you would type at the console prompt. A true script is not for example an MS-Word file where you have pasted or typed some R commands. A script file has the following characteristics.

- The script is a text file (ASCII or some other encoding e.g. UTF-8 that R uses in your set-up).
- The file contains valid R statements (including comments) and nothing else.
- Comments start at a # and end at the end of the line. (True end-of line as coded in file, the editor may wrap it or not at the edge of the screen).
- The R statements are in the file in the order that they must be executed.
- R scripts have file names ending in `.r` or `.R`.

It is good practice to write scripts so that they are self-contained. Such a scripts will run in a new R session by including library commands to load all the required packages.

3.2 How do we use a scrip?

A script can be sourced.

If we have a text file called `my.first.script.r` containing the following text:

```
# this is my first R script
print(3+4)
```

And then source this file:

```
source("my.first.script.r")

## [1] 7
```

The results of executing the statements contained in the file will appear in the console. The commands themselves are not shown (the sourced file is not echoed) and the results will not be printed unless you include an explicit `print` command in the script. This applies in many cases also to plots—e.g. A figure created with `ggplot` needs to be printed if we want to see it when the script is run. Adding a redundant `print` is harmless.

From within RStudio, if you have an R script open in the editor, there will a "source" drop box (≠ DropBox) visible from where you can choose "source" as described above, or "source with echo" for the currently open file.

When a script is sourced, the output can be saved to a text file instead of being shown in the console. It is also easy to call R with the script file as argument directly at the command prompt of the operating system.

```
RScript my.first.script.r
```

You can open a 'shell' from the Tools menu in RStudio, to run this command. The output will be printed to the shell console. If you would like to save the output to a file, use redirection.

```
RScript my.first.script.r > my.output.txt
```

Sourcing is very useful when the script is ready, however, while developing a script, or sometimes when testing things, one usually wants to run (= execute) one or a few statements at a time. This can be done using the "run" button after either locating the cursor in the line to be executed, or selecting the text that one would like to run (the selected text can be part of a line, a whole line, or a group of lines, as long as it is syntactically valid).

3.3 How to write a script?

The approach used, or mix of approaches will depend on your preferences, and on how confident you are that the statements will work as expected.

If one is very familiar with similar problems One would just create a new text file and write the whole thing in the editor, and then test it. This is rather unusual.

If one if moderately familiar with the problem One would write the script as above, but testing it, part by part as one is writing it. This is usually what I do.

If ones mostly playing around Then if one is using RStudio, one type statements at the console prompt. As you should know by now, everything you run at the console is saved to the "History". In RStudio the History is displayed in its own pane, and in this pane one can select any previous statement and by pressing a single having copy and pasted to either the console prompt, or the cursor position in the file visible in the editor. In this way one can build a script by copying and pasting from the history to your script file the bits that have worked as you wanted.

3.3.1 Exercise

By now you should be familiar enough with R to be able to write your own script.

1. Create a new R script (in RStudio, from 'File' menu, "+" button, or by typing "Ctrl + Shift + N").
2. Save the file as "my.second.script.r".
3. Use the editor pane in RStudio to type some R commands and comments.
4. **Run** individual commands.
5. **Source** the whole file.

3.4 The need to be understandable to people

When you write a script, it is either because you want to document what you have done or you want re-use it at a later time. In either case, the script itself although still meaningful for the computer could become very obscure to you, and even more to someone seeing it for the first time.

How does one achieve an understandable script or program?

- Avoid the unusual. People using a certain programming language tend to use some implicit or explicit rules of style[1]. As a minimum try to be consistent with yourself.
- Use meaningful names for variables, and any other object. What is meaningful depends on the context. Depending on common use a single letter may be more meaningful than a long word. However self explaining names are better: e.g. using `n.rows` and `n.cols` is much clearer than using `n1` and `n2` when dealing with a matrix of data. Probably `number.of.rows` and `number.of.columns` would just increase the length of the lines in the script, and one would spend more time typing without getting much in return.
- How to make the words visible in names: traditionally in R one would use dots to separate the words and use only lower case. Some years ago, it became possible to use underscores. The use of underscores is quite common nowadays because in some contexts is "safer" as in some situations a dot may have a special meaning. What we call "camel case" is only infrequently used in R programming but is common in other languages like Pascal. An example of camel case is `NumCols`. In some cases it can become a bit confusing as in `UVMean` or `UvMean`.

3.4.1 Exercise

Here is an example of bad style in a script, edit so that it becomes easier to read.

```
a <- 2 # height
b <- 4 # length
C <-
    a *
b
C -> variable
      print(
"area: ", variable
)
```

3.5 Functions

When writing scripts, or any program, one should avoid repeating blocks of code (groups of statements). The reasons for this are: 1) if the code

[1] Style includes *indentation* of statements, *capitalization* of variable and function names.

needs to be changed, you have to make changes in more than one place in the file, or in more than one file. Sooner or later, some copies will remain unchanged by mistake. 2) it makes the script file longer, and this makes debugging, commenting, etc. more tedious, and error prone.

How do we avoid repeating bits of code? We write a function containing the statements that we would need to repeat, and then `call` (use) the function in their place.

Functions are defined by means of **function**, and saved like any other object in R by assignment to a variable. In the example below `x` and `y` are both formal parameters, or names used within the function for objects that will be supplied as "arguments" when the function is called. One can think of parameter names as place-holders.

```
my.prod <- function(x, y){x * y}
my.prod(4, 3)

## [1] 12
```

First some basic knowledge. In R, arguments are passed by copy. This is something very important to remember. Whatever you do within a function to modify an argument, its value outside the function will remain (almost) always unchanged.

```
my.change <- function(x){x <- NA}
a <- 1
my.change(a)
a

## [1] 1
```

Any result that needs to be made available outside the function must be returned by the function. If the function `return` is not explicitly used, the value returned by the last statement *executed* within the body of the function will be returned.

```
print.x.1 <- function(x){print(x)}
print.x.1("test")

## [1] "test"

print.x.2 <- function(x){print(x); return(x)}
print.x.2("test")

## [1] "test"
## [1] "test"
```

```
print.x.3 <- function(x){return(x); print(x)}
print.x.3("test")

## [1] "test"

print.x.4 <- function(x){return(); print(x)}
print.x.4("test")

## NULL

print.x.5 <- function(x){x}
print.x.4("test")

## NULL
```

Now we will define a useful function: a function for calculating the standard error of the mean from a numeric vector.

```
SEM <- function(x){sqrt(var(x)/length(x))}
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
SEM(x=a)

## [1] 1.796988

SEM(a)

## [1] 1.796988

SEM(a.na)

## [1] NA
```

For example in `SEM(a)` we are calling function `SEM` with `a` as argument.

The function we defined above may sometimes give a wrong answer because NAs will be counted by `length`, so we need to remove NAs before calling `length`.

```
simple_SEM <- function(x) {
 sqrt(var(x, na.rm=TRUE)/length(na.omit(x)))
}
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
simple_SEM(x=a)

## [1] 1.796988

simple_SEM(a)

## [1] 1.796988
```

```
simple_SEM(a.na)

## [1] 1.796988
```

R does not have a function for standard error, so the function above would be generally useful. If we would like to make this function both safe, and consistent with other R functions, one could define it as follows, allowing the user to provide a second argument which is passed as an argument to `var`:

```
SEM <- function(x, na.rm=FALSE){
  sqrt(var(x, na.rm=na.rm)/length(na.omit(x)))
}
SEM(a)

## [1] 1.796988

SEM(a.na)

## [1] NA

SEM(a.na, TRUE)

## [1] 1.796988

SEM(x=a.na, na.rm=TRUE)

## [1] 1.796988

SEM(TRUE, a.na)

## Warning in if (na.rm) "na.or.complete" else "everything": the condition
has length > 1 and only the first element will be used

## [1] NA

SEM(na.rm=TRUE, x=a.na)

## [1] 1.796988
```

In this example you can see that functions can have more than one parameter, and that parameters can have default values to be used if no argument is supplied. In addition if the name of the parameter is indicated, then arguments can be supplied in any order, but if parameter names are not supplied, then arguments are assigned to parameters based on their position. Once one parameter name is given, all later arguments need also to be explicitly matched to parameters. Obviously if given by position, then arguments should be supplied explicitly for all parameters at

'intermediate' positions.

3.5.1 Exercise

1) Test the behaviour of `print.x.1` and `print.x.5` at the command prompt, and in a script, by writing a script. The behaviour of one of these functions will be different when the script is source than at the command prompt. Explain why.

2) Define your own function to calculate the mean in a similar way as `SEM()` was defined above. Hint: function `sum()` could be of help.

3) Create some additional vectors containing `NA`s or not. Use them to test functions `simple_SEM()` and `SEM()` defined above, and then explain why `SEM()` returns always the correct value, even though "`na.omit(x)`" is non-conditionally (always) applied to `x` before calculating its length.

3.6 Control of execution flow

We call control of execution statements those that allow the execution of sections of code when a certain dynamically computed condition is TRUE. Some of the control of execution flow statements, function like 'ON-OFF switches' for program statements. Others, allow statements to executed repeatedly while or until a condition is met, or until all members of a list or a vector are processed.

3.6.1 Conditional execution

Non-vectorized

R has two types of "if" statements, non-vectorized and vectorized. We will start with the non-vectorized one, which is similar to what is available in most other computer programming languages.

Before this we need to explain compound statements. Individual statements can be grouped into compound statements by enclosed them in curly braces.

```
print("A")

## [1] "A"

{
  print("B")
  print("C")
}
```

```
## [1] "B"
## [1] "C"
```

The example above is pretty useless, but becomes useful when used together with 'control' constructs. The `if` construct controls the execution of one statement, however, this statement can be a compound statement of almost any length or complexity. Play with the code below by changing the value assigned to `printing`, including NA, and logical(0).

```
printing <- TRUE
if (printing) {
  print("A")
  print("B")
}

## [1] "A"
## [1] "B"
```

The condition '( )' can be anything yielding a logical vector, however, as this is not vectorized, only the first element will be used. Play with this example by changing the value assigned to `a`.

```
a <- 10.0
if (a < 0.0) print("'a' is negative") else print("'a' is not negative")

## [1] "'a' is not negative"

print("This is always printed")

## [1] "This is always printed"
```

As you can see above the statement immediately following `else` is executed if the condition is false. Later statements are executed independently of the condition.

Do you still remember the rules about continuation lines?

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

```
# 1
a <- 1
if (a < 0.0)
  print("'a' is negative") else
    print("'a' is not negative")

## [1] "'a' is not negative"
```

Why does the statement below (not here) trigger an error?

```
# 2 (not evaluated here)
if (a < 0.0) print("'a' is negative")
else print("'a' is not negative")
```

Play with the use conditional execution, with both simple and compound statements, and also think how to combine `if` and `else` to select among more than two options.

There is in R a `switch` statement, that we describe here, which can be used to select among "cases", or several alternative statements, based on an expression evaluating to a number or a character string. The switch statement returns a value, the value returned by the code corresponding to the matching switch value, or the default if there is no match, and a default has been included in the code. Both character values or numeric values can used.

```
my.object <- "two"
b <- switch(my.object,
            one = 1,
            two = 1 / 2,
            three = 1/ 4,
            0
)
b

## [1] 0.5
```

Do play with the use of the switch statement.

Vectorized

Vectorized conditional execution is coded by means of a **function** called `ifelse` (one word). This function takes three arguments: a logical vector, a result vector for TRUE, a result vector for FALSE. All three can be any construct giving the necessary argument as their return value. In the case of result vectors, recycling will apply if they are not of the correct length. The length of the result is determined by the length of the logical vector in the first argument!.

```
a <- 1:10
ifelse(a > 5, 1, -1)

##  [1] -1 -1 -1 -1 -1  1  1  1  1  1

ifelse(a > 5, a + 1, a - 1)
```

```
##  [1]  0  1  2  3  4  7  8  9 10 11

ifelse(any(a>5), a + 1, a - 1) # tricky

## [1] 2

ifelse(logical(0), a + 1, a - 1) # even more tricky

## logical(0)

ifelse(NA, a + 1, a - 1) # as expected

## [1] NA
```

Try to understand what is going on in the previous example. Create your own examples to test how `ifelse` works.

Exercise: write using `ifelse` a single statement to combine numbers from a and b into a result vector d, based on whether the corresponding value in c is the character ”a” or ”b”.

```
a <- rep(-1, 10)
b <- rep(+1, 10)
c <- c(rep("a", 5), rep("b", 5))
# your code
```

If you do not understand how the three vectors are built, or you cannot guess the values they contain by reading the code, print them, and play with the arguments, until you have clear what each parameter does.

3.6.2 Why using vectorized functions and operators is important

If you have written programs in other languages, it would feel to you natural to use loops (for, repeat while, repeat until) for many of the things for which we have been using vectorization. When using the R language it is best to use vectorization whenever possible, because it keeps the listing of scripts and programs shorter and easier to understand (at least for those with experience in R). However, there is another very important reason: execution speed. The reason behind this is that R is an interpreted language. In current versions of R it is possible to byte-compile functions, but this is rarely used for scripts, and even byte-compiled loops are much slower and vectorized functions.

However, there are cases were we need to repeatedly execute statements in a way that cannot be vectorized, or when we do not need to maximize execution speed. The R language does have loop constructs, and we will describe them next.

3.6.3 Repetition

The most frequently used type of loop is a `for` loop. These loops work in R are based on lists or vectors of values to act upon.

```
b <- 0
for (a in 1:5) b <- b + a
b

## [1] 15

b <- sum(1:5) # built-in function
b

## [1] 15
```

Here the statement `b <- b + a` is executed five times, with a sequentially taking each of the values in `1:5`. Instead of a simple statement used here, also a compound statement could have been used.

Here are a few examples that show some of the properties of `for` loops and functions, combined with the use of a function.

```
test.for <- function(x) {
  for (i in x) {print(i)}
}
test.for(numeric(0))
test.for(1:3)

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

test.for(NA)

## [1] NA

test.for(c("A", "B"))

## [1] "A"
## [1] "B"

test.for(c("A", NA))

## [1] "A"
## [1] NA

test.for(list("A", 1))

## [1] "A"
## [1] 1
```

```
test.for(c("z", letters[1:4]))

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

In contrast to other languages, in R function arguments are not checked for 'type' when the function is called. The only requirement is that the function code can handle the argument provided. In this example you can see that the same function works with numeric and character vectors, and with lists. We haven't seen lists before. As earlier discussed all elements in a vector should have the same type. This is not the case for lists. It is also interesting to note that a list or vector of length zero is a valid argument, that triggers no error, but that as one would expect, causes the statements in the loop body to be skipped.

Some examples of use of `for` loops — and of how to avoid there use.

```
a <- c(1, 4, 3, 6, 8)
for(x in a) x*2 # result is lost
for(x in a) print(x*2) # print is needed!

## [1] 2
## [1] 8
## [1] 6
## [1] 12
## [1] 16

b <- for(x in a) x*2 # doesn't work as expected, but triggers no error
b

## NULL

for(x in a) b <- x*2 # a bit of a surprise, as b is not a vector!
b

## [1] 16

for(i in seq(along.with = a)) {
  b[i] <- a[i]^2
  print(b)
}

## [1] 1
## [1]  1 16
## [1]  1 16  9
## [1]  1 16  9 36
## [1]  1 16  9 36 64
```

```
b # is a vector!

## [1]  1 16  9 36 64

# a bit faster if we first allocate a vector of the required length
b <- numeric(length(a))
for(i in seq(along.with = a)) {
  b[i] <- a[i]^2
  print(b)
}

## [1] 1 0 0 0 0
## [1]  1 16  0  0  0
## [1]  1 16  9  0  0
## [1]  1 16  9 36  0
## [1]  1 16  9 36 64

b # is a vector!

## [1]  1 16  9 36 64

# vectorization is simplest and fastest
b <- a^2
b

## [1]  1 16  9 36 64
```

Look at the results from the above examples, and try to understand where does the returned value come from in each case.

We sometimes may not be able to use vectorization, or may be easiest to not use it. However, whenever working with large data sets, or many similar datasets, we will need to take performance into account. As vectorization usually also makes code simpler, it is good style to use it whenever possible.

```
b <- numeric(length(a)-1)
for(i in seq(along.with = b)) {
  b[i] <- a[i+1] - a[i]
  print(b)
}

## [1] 3 0 0 0
## [1]  3 -1  0  0
## [1]  3 -1  3  0
## [1]  3 -1  3  2

# although in this case there were alternatives, there
# are other cases when we need to use indexes explicitly
b <- a[2:length(a)] - a[1:length(a)-1]
b
```

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

# or even better
b <- diff(a)
b

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

`seq(along.with = b)` builds a new numeric vector with a sequence of the same length as the length as the vector given as argument for parameter 'along'.

`while` loops are quite frequently also useful. Instead of a list or vector, they take a logical argument, which is usually an expression, but which can also be a variable. For example the previous calculation could be also done as follows.

```
a <- c(1, 4, 3, 6, 8)
i <- 1
while (i < length(a)) {
  b[i] <- a[i]^2
  print(b)
  i <- i + 1
}

## [1]  1 -1  3  2
## [1]  1 16  3  2
## [1]  1 16  9  2
## [1]  1 16  9 36

b

## [1]  1 16  9 36
```

Here is another example. In this case we use the result of the previous iteration in the current one. In this example you can also see, that it is allowed to put more than one statement in a single line, in which case the statements should be separated by a semicolon (;).

```
a <- 2
while (a < 50) {print(a); a <- a^2}

## [1] 2
## [1] 4
## [1] 16

print(a)

## [1] 256
```

Make sure that you understand why the final value of `a` is larger than 50.

`repeat` is seldom used, but adds flexibility as `break` can be in the middle of the compound statement.

```
a <- 2
repeat{
  print(a)
  a <- a^2
  if (a > 50) {print(a); break()}
}

## [1] 2
## [1] 4
## [1] 16
## [1] 256

# or more elegantly
a <- 2
repeat{
  print(a)
  if (a > 50) break()
  a <- a^2
}

## [1] 2
## [1] 4
## [1] 16
## [1] 256
```

Please, make sure you understand what is happening in the previous examples.

3.6.4 Nesting

All the execution-flow control statements seen above can be nested. We will show an example with two `for` loops. We first need a matrix of data to work with:

```
A <- matrix(1:50, 10)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
```

```
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

A <- matrix(1:50, 10, 5)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

# argument names used for clarity
A <- matrix(1:50, nrow = 10)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

A <- matrix(1:50, ncol = 5)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

A <- matrix(1:50, nrow = 10, ncol = 5)
A
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50
```

All the statements above are equivalent, but some are easier to read than others.

```
row.sum <- numeric() # slower as size needs to be expanded
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

```
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

Look at the output of these two examples to understand what is happening differently with `row.sum`.

The code above is very general, it will work with any size of two dimensional matrix, which is good programming practice. However, sometimes we need more specific calculations. `A[1, 2]` selects one cell in the matrix, the one on the first row of the second column. `A[1, ]` selects row one, and `A[ , 2]` selects column two. In the example above the value of `i` changes for each iteration of the outer loop. The value of `j` changes for each iteration of the inner loop, and the inner loop is run in full for each iteration of the outer loop. The inner loop index `j` changes fastest.

Exercises: 1) modify the example above to add up only the first three columns of A, 2) modify the example above to add the last three columns of A.

Will the code you wrote continue working as expected if the number of rows in A changed? and what if the number of columns in A changed, and the required results still needed to be calculated for relative positions? What would happen if A had fewer than three columns? Try to think first what to expect based on the code you wrote. Then create matrices of different sizes and test your code. After that think how to improve the code, at least so that wrong results are not produced.

Vectorization can be achieved in this case easily for the inner loop.

```
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- sum(A[i, ])
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

`A[i, ]` selects row `i` and all columns. In R, the row index always comes first, which is not the case in all programming languages.

Full vectorization can be achieved with `apply` functions.

```
row.sum <- apply(A, MARGIN = 1, sum) # MARGIN=1 inidcates rows
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

How would you change this last example, so that only the last three columns are added up? (Think about use of subscripts to select a part of the matrix.)

There are many variants of `apply` functions, both in base R and in contributed packages.

3.7 Packages

In R speak 'library' is the location where 'packages' are installed. Packages are sets of functions, and data, specific for some particular purpose, that can be loaded into an R session to make them available so that they can be used in the same way as built-in R functions and data. The function `library` is used to load packages, already installed in the local R library, into the current session, while the function `install.packages` is used to install packages, either from a file, or directly from the internet into the library. When using RStudio it is easiest to use RStudio commands (which call `install.packages` and `update.packages`) to install and update packages.

```
library(graphics)
```

Currently there are thousands of packages available. The most reliable source of packages is CRAN, as only packages that pass strict tests and are actively maintained are included. In some cases you may need or want to install less stable code, and this is also possible. With package `devtools` it is even possible to install packages directly from Github, Bitbucket and a few other repos. These later installations are always installations from source (see below).

R packages can be installed either from source, or from already built 'binaries'. Installing from sources, depending on the package, may require quite a lot of additional software to be available. Under MS-Windows, very rarely the needed shell, commands and compilers are already available. Installing then is not too difficult (you will need RTools, and MiKTeX). For this reason it is the norm to install packages from binary .zip files. Under Linux most tools will be available, or very easy to install, so it is not unusual to install from sources. For OS X (Mac) the situation is somewhere in-between. If the tools are available, packages can be very easily installed from source from within RStudio.

The development of packages is beyond the scope of the current course, but it is still interesting to know a few things about packages. Using RStudio it is relatively easy to develop your own packages. Packages can be of very different sizes. Packages use a relatively rigid structure of folder for storing the different types of files, and there is a built-in help system, that one needs to use, so that the package documentation gets linked to the R help system when the package is loaded. In addition to R code, packages can call C, C++, FORTRAN, Java, etc. functions and routines, but some kind of 'glue' is needed, as data is stored differently. At least for C++, the recently developed Rcpp R package makes the gluing extremely easy.

One good way of learning how R works, is by experimenting with it, and whenever using a certain function looking at the help, to check what are all the available options.

4 R built-in functions

4.1 Loading data

To start with we need some data. Here we use `cars`, a data set included in base R.

```
data(cars)
```

4.2 Looking at the data

```
names(cars)

## [1] "speed" "dist"

head(cars)

##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10

tail(cars)

##    speed dist
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85

str(cars)

## 'data.frame': 50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

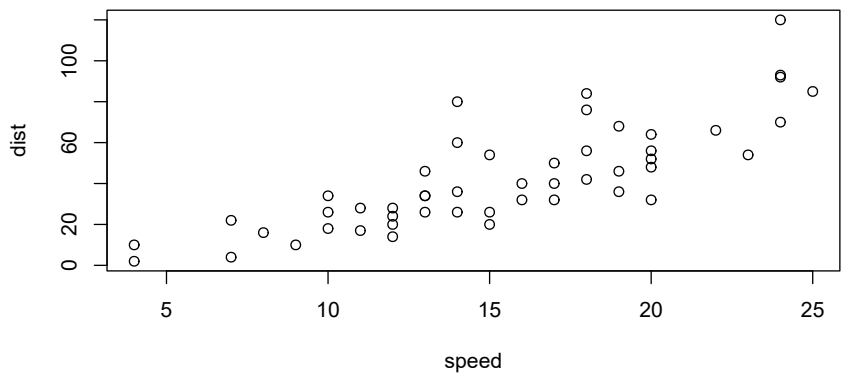
The `cars` data are stored as a data frame. Data frames consist in columns of equal length. The different columns can be different types (e.g. numeric and character). With `data()` we load it; with `names()` we obtain the names of the variables or columns. With head with can see the top several lines, and with tail the lines at the end. In general `data()` is used load R objects saved in a file format used by R. Text files con be read with functions `scan()`, `read.table()`, `read.csv()` and their variants.

It is also possible to 'import' data saved in files of *foreign* formats, defined by other programs. Packages such as 'foreign', 'readr', 'readxl', 'RNetCDF', 'jsonlite', etc. allow importing data from other statistic and data analysis applications and from standard data exchange formats. It is also good to keep in mind that in R urls are accepted as arguments to the `file` argument.

4.3 Plotting

The built-in generic function `plot` can be used to plot data. It is a generic function, that has suitable methods for different kinds of objects. In this section we only very briefly demonstrate the use of the most common base R's graphics functions.

```
plot(dist ~ speed, data=cars)
```



4.4 Fitting linear models

One important thing to remember is that model 'formulas' are used in different contexts: plotting, fitting of models, and tests like t-test. The basic syntax is rather consistently followed, although there are some exceptions.

4.4.1 Regression

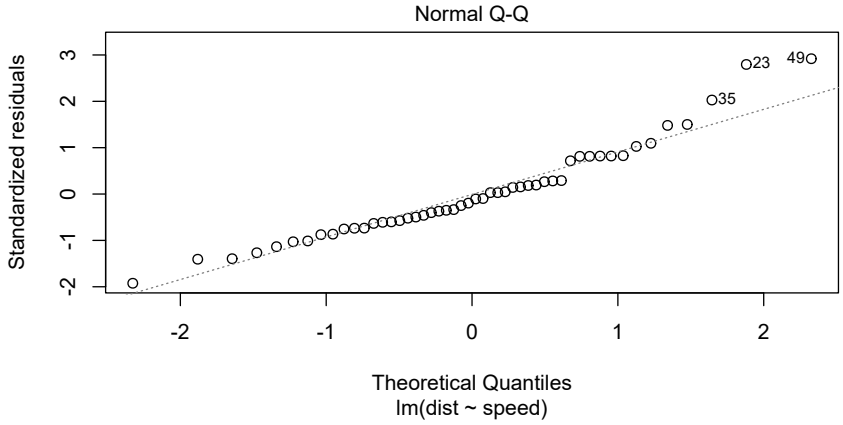
The R function `lm` is used next to fit linear models. If the explanatory variable is continuous, the fit is a regression. In the example below, `speed` is a numeric variable (floating point in this case). In the ANOVA table calculated for the model fit, in this case a linear regression, we can see that the term for `speed` has only one degree of freedom (df) for the denominator.

We first fit the model and save the output as `fm1` (A name I invented to remind myself that this is the first fitted-model in this chapter.

```
fm1 <- lm(dist ~ speed, data=cars)
```

The next step is diagnosis of the fit. Are assumptions of the linear model procedure used reasonably fulfilled? In R it is most common to use plots to this end. We show here only one of the four plots normally produced. This quantile vs. quantile plot allows to assess how much the residuals deviate from being normally distributed.

```
plot(fm1, which = 2)
```

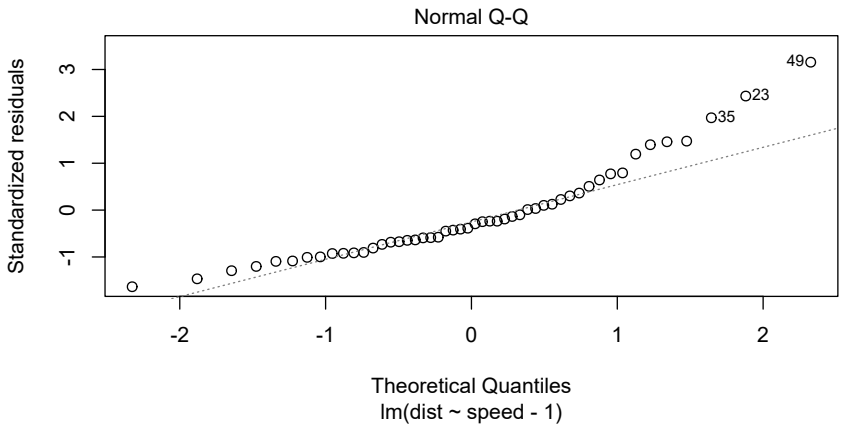


In the case of a regression, calling `summary()` with the fitted model object as argument is most useful as it provides a table of coefficient estimates and their errors. `anova()` applied to the same fitted object, returns the ANOVA table.

```
summary(fm1) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123
## speed         3.9324     0.4155   9.464 1.49e-12
##
## (Intercept) *
## speed       ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511,Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12

anova(fm1) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value   Pr(>F)
## speed      1  21186 21185.5  89.567 1.49e-12 ***
## Residuals 48  11354   236.5
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Let's look at each argument separately: `dist ~ speed` is the specification of the model to be fitted. The intercept is always implicitly included. To 'remove' this implicit intercept from the earlier model we can use `dist ~ speed - 1`. In what follows we fit a straight line through the origin ($x = 0$, $y = 0$).

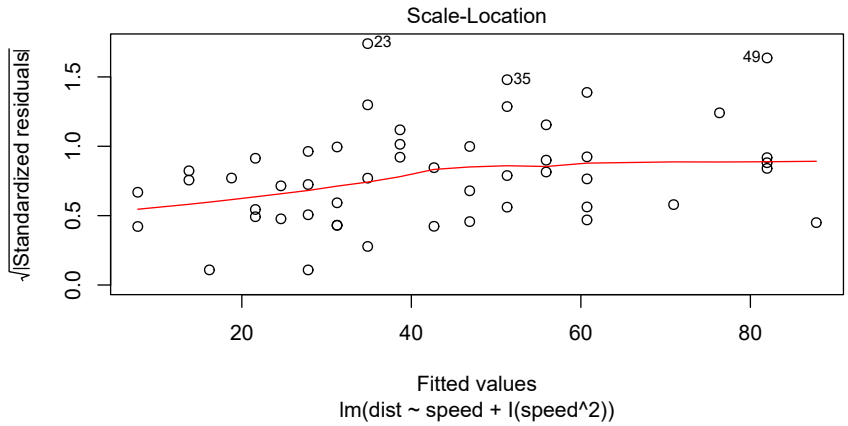
```r
fm2 <- lm(dist ~ speed - 1, data=cars)
plot(fm2, which = 2)
summary(fm2)

##
## Call:
## lm(formula = dist ~ speed - 1, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -26.183 -12.637  -5.455   4.590  50.181
##
## Coefficients:
##       Estimate Std. Error t value Pr(>|t|)
## speed   2.9091     0.1414   20.58   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.26 on 49 degrees of freedom
## Multiple R-squared:  0.8963,Adjusted R-squared:  0.8942
## F-statistic: 423.5 on 1 and 49 DF,  p-value: < 2.2e-16

anova(fm2)

## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value    Pr(>F)
## speed      1 111949  111949  423.47 < 2.2e-16 ***
## Residuals 49  12954     264
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



We now we fit a second degree polynomial.

```
fm3 <- lm(dist ~ speed + I(speed^2), data=cars) # we fit a model, and then save the res-
ult
plot(fm3, which = 3) # we produce diagnosis plots
summary(fm3) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed + I(speed^2), data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -28.720  -9.184  -3.188   4.628  45.152
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.47014   14.81716   0.167    0.868
## speed        0.91329    2.03422   0.449    0.656
## I(speed^2)   0.09996    0.06597   1.515    0.136
##
## Residual standard error: 15.18 on 47 degrees of freedom
## Multiple R-squared:  0.6673,Adjusted R-squared:  0.6532
## F-statistic: 47.14 on 2 and 47 DF,  p-value: 5.852e-12

anova(fm3) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##            Df  Sum Sq Mean Sq F value    Pr(>F)
## speed       1 21185.5 21185.5  91.986 1.211e-12
```

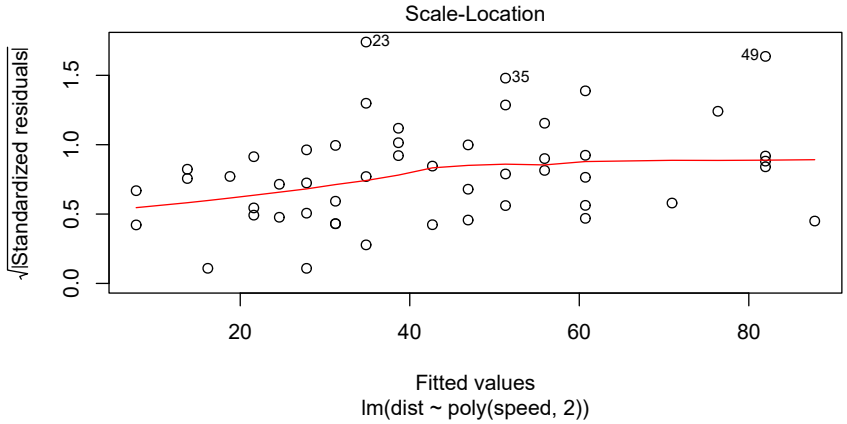
```
## I(speed^2)  1   528.8   528.8   2.296    0.1364
## Residuals  47 10824.7   230.3
##
## speed      ***
## I(speed^2)
## Residuals
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



The "same" fit using an orthogonal polynomial. Higher degrees can be obtained by supplying as second argument to `poly()` a different positive integer value.

```
fm3a <- lm(dist ~ poly(speed, 2), data=cars) # we fit a model, and then save the res-
ult
plot(fm3a, which = 3) # we produce diagnosis plots
summary(fm3a) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ poly(speed, 2), data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -28.720  -9.184  -3.188   4.628  45.152
##
## Coefficients:
##                 Estimate Std. Error t value
## (Intercept)       42.980      2.146  20.026
## poly(speed, 2)1  145.552     15.176   9.591
```

```
## poly(speed, 2)2   22.996     15.176   1.515
##                  Pr(>|t|)
## (Intercept)       < 2e-16 ***
## poly(speed, 2)1 1.21e-12 ***
## poly(speed, 2)2    0.136
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.18 on 47 degrees of freedom
## Multiple R-squared:  0.6673,Adjusted R-squared:  0.6532
## F-statistic: 47.14 on 2 and 47 DF,  p-value: 5.852e-12

anova(fm3a) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##                Df Sum Sq Mean Sq F value
## poly(speed, 2)  2  21714 10857.1  47.141
## Residuals      47  10825   230.3
##                   Pr(>F)
## poly(speed, 2) 5.852e-12 ***
## Residuals
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



We can also compare two models, to test whether one of models describes the data better than the other.

```
anova(fm2, fm1)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
##   Res.Df   RSS Df Sum of Sq      F  Pr(>F)
## 1     49 12954
## 2     48 11354  1    1600.3 6.7655 0.01232 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Or three or more models. But be careful, as the order of the arguments matters.

```
anova(fm2, fm1, fm3, fm3a)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
## Model 3: dist ~ speed + I(speed^2)
## Model 4: dist ~ poly(speed, 2)
##   Res.Df   RSS Df Sum of Sq      F  Pr(>F)
## 1     49 12954
## 2     48 11354  1   1600.26 6.9482 0.01133 *
## 3     47 10825  1    528.81 2.2960 0.13640
## 4     47 10825  0      0.00
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can use different criteria to choose the best model: significance based on P-values or information criteria (AIC, BIC). AIC and BIC penalize the resulting 'goodness' based on the number of parameters in the fitted model. In the case of AIC and BIC, a smaller value is better, and values returned can be either positive or negative, in which case more negative is better.

```
BIC(fm2, fm1, fm3, fm3a)

##      df      BIC
## fm2   2 427.5739
## fm1   3 424.8929
## fm3   4 426.4202
## fm3a  4 426.4202

AIC(fm2, fm1, fm3, fm3a)
```

```
##      df      AIC
## fm2   2 423.7498
## fm1   3 419.1569
## fm3   4 418.7721
## fm3a  4 418.7721
```

One can see above that these three criteria not necessarily agree on which is the model to be chosen.

anova `fm1`

BIC `fm1`

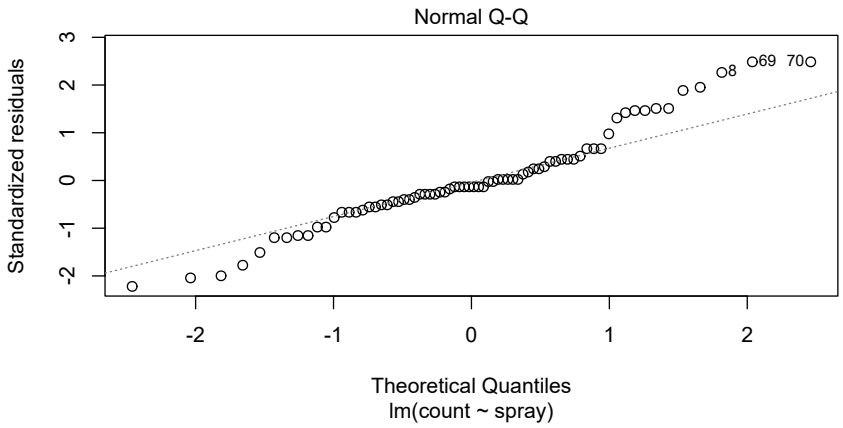
AIC `fm3`

4.4.2 Analysis of variance, ANOVA

We use as the `InsectSpray` data set, giving insect counts in plots sprayed with different insecticides. In these data `spray` is a factor with six levels.

```
fm4 <- lm(count ~ spray, data = InsectSprays)
```

```
plot(fm4, which = 2)
```




```
anova(fm4)
```

```
## Analysis of Variance Table
##
## Response: count
##           Df Sum Sq Mean Sq F value    Pr(>F)
## spray      5 2668.8  533.77  34.702 < 2.2e-16 ***
## Residuals 66 1015.2   15.38
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4.4.3 Analysis of covariance, ANCOVA

When a linear model includes both explanatory factors and continuous explanatory variables, we say that analysis of covariance (ANCOVA) is used. The formula syntax is the same for all linear models, what determines the type of analysis is the nature of the explanatory variable(s). Conceptually a factor (an unordered categorical variable) is very different from a continuous variable.

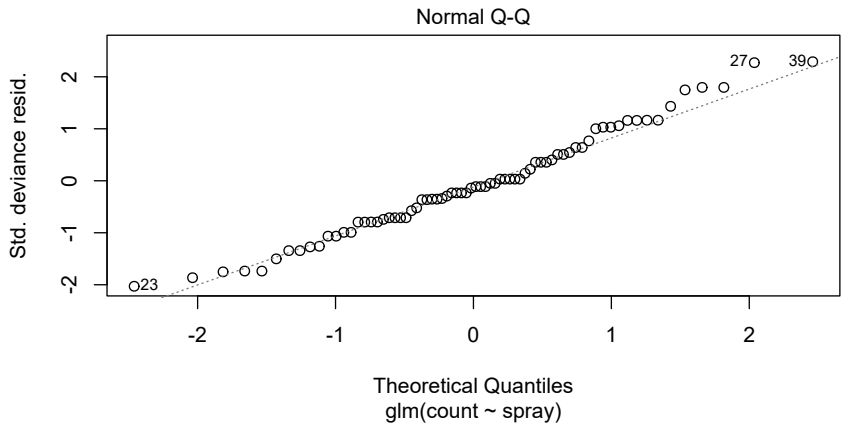
4.5 Generalized linear models

Linear models make the assumption of normally distributed residuals. Generalized linear models are more flexible, and allow the assumed distribution to be selected as well as the link function. For the analysis of the `InsectSpray` data set, above (section 4.4.2 on page 66) the Normal distribution is not a good approximation as count data deviates from it. This was visible in the quantile–quantile plot above.

For count data GLMs provide a better alternative. In the example below we fit the same model as above, but we assume a quasi-Poisson distribution instead of the Normal.

```
fm10 <- glm(count ~ spray, data = InsectSprays, family = quasipoisson)
plot(fm10, which = 2)
anova(fm10, test = "F")

## Analysis of Deviance Table
##
## Model: quasipoisson, link: log
##
## Response: count
##
## Terms added sequentially (first to last)
##
##
```

```
##       Df Deviance Resid. Df Resid. Dev      F
## NULL                     71     409.04
## spray  5   310.71        66      98.33 41.216
##          Pr(>F)
## NULL
## spray < 2.2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



5 Storing and manipulating data with R

5.1 Introduction

By reading previous chapters, you have already become familiar with base R's classes, methods, functions and operators for storing and manipulating data. Several recently developed packages provide somehow different, and in my view easier, ways of working with data in R without compromising performance to a level that would matter outside the realm of 'big data'. Some other recent packages emphasize computation speed, at some cost with respect to simplicity of use, and in particular intuitiveness. Of course, as with any user interface, much depends on one's own preferences and attitudes to data analysis. However, a package designed for maximum efficiency like 'data.table' requires of the user to have a good understanding of computers to be able to understand the compromises and the unusual behavior compared to the rest of R. I will base this chapters on what I mostly use myself for everyday data analysis and scripting, and exclude the complexities of R programming and package development.

The chapter is divided in three sections, the first one deals with reading data from files produced by other programs or instruments, or typed by users outside of R, and querying databases and very briefly on reading data from the internet. The second section will deal with transformations of the data that do not combine different observations, although they may combine different variables from a single observation event, or select certain variables or observations from a larger set. The third section will deal with operations that produce summaries or involve other operations on groups of observations.

5.2 Data input

5.2.1 Text files

5.2.2 Worksheets

5.2.3 Statistical software

5.2.4 Databases

5.2.5 Data acquisition from web

5.2.6 Data acquisition from physical devices

5.3 Pipes and tees

5.3.1 Processing data step by step

5.3.2 Pipes and tees in the Unix shell

5.3.3 Pipes and tees in R scripts

5.4 Row-wise data manipulations

5.4.1 Computations

5.4.2 Subsetting

5.4.3 Merging and joints

5.5 Column-wise data manipulations

5.5.1 Grouping

5.5.2 Summaries

5.5.3 Variable selection

5.6 Data output

5.6.1 Text files

5.6.2 Worksheets

5.6.3 Statistical software

5.6.4 Databases

5.6.5 Publication to the web

5.6.6 Control of physical devices

6 Plots with 'ggplot2'

6.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(ggplot2)
library(tikzDevice)
```

We set a font of larger size than the default

```
theme_set(theme_grey(14))
```

6.2 Introduction

Being R extensible, in addition to the built-in plotting functions, there are several alternatives provided by packages. Of the general purpose ones, the most extensively used are `Lattice` and 'ggplot2'. There are additional packages that add extra functionality to these packages.

In the examples in this chapter we describe the of use package 'ggplot2'. We start with an introduction to the 'grammar of graphics' and 'ggplot2'. There is ample literature on the use of 'ggplot2', including the very good reference documentation at `http://ggplot2.org/`. The 'ggplot2'book (**Wickham2016**) is the authoritative reference, as it is authored by the developer of 'ggplot2'. The book 'R Graphics Cookbook' (Chang 2013) is very useful as a reference as it contains many worked out examples. Much of the literature available at this time is for older versions of 'ggplot2'but we here describe version 2.2.0, and highlight the most important incompatibilities that need to be taken into account when using versions of 'ggplot2'earlier than 2.2.0. There is no comprehensive text on packages extending 'ggplot2'so I will describe some of them in later chapters. In the present chapter we describe the functions and methods defined in package 'ggplot2', in chapter 7 on page 183 we describe extensions to 'ggplot2'defined in other packages, except for those related to plotting data onto maps and other images, described in chapter 8 on page 245.

6.3 Grammar of graphics

What separates 'ggplot2'from base-R and trellis/lattice plotting functions is the use of a grammar of graphics (the reason behind 'gg' in the name of the package). What is meant by grammar in this case is that plots are assembled piece by piece from different 'nouns' and 'verbs'. Instead of using a single function with many arguments, plots are assembled by combining different elements with operators `+` and %+%. Furthermore, the constructions is mostly semantic-based and to a large extent how the plot looks when is printed, displayed or exported to a bitmap or vector graphics file is controlled by themes.

6.3.1 Mapping

When we design a plot, we need to map data variables to aesthetics (or graphic 'properties'). Most plots will have an x dimension, which is considered an aesthetic, and a variable containing numbers mapped to it. The position on a 2D plot of say a point will be determined by x and y aesthetics, while in a 3D plot, three aesthetics need to be mapped x, y and z. Many aesthetics are not related to coordinates, they are properties, like color, size, shape, line type or even rotation angle, which add an additional dimension on which to plot variables.

6.3.2 Geometries

Geometries describe the graphics representation of the data: for example, `geom_point`, plots a 'point' or symbol for each observation, while `geom_line`, draws line segments between successive observations. Some geometries rely on statistics, but most 'geoms' default to the identity statistics.

6.3.3 Statistics

Statistics are 'words' that represent calculation of summaries or some other values from the data, and these values can be plotted with a geometry. For example `stat_smooth` fits a smoother, and `stat_summary` applies a summary function. Statistics are applied automatically by group when data has been grouped by mapping additional aesthetics such as color.

But as all this is easier to show by example than to explain, after this short introduction we will focus on examples showing how to produce graphs of increasing complexity.

6.3.4 Scales

Scales give the relationship between data values and the aesthetic values to be actually plotted. Mapping a variable to the 'color' aesthetic only tells that different values stored in the mapped variable will be represented by different colors. A scale, such as `scale_color_continuous` will determine which color in the plot corresponds to which value in the variable. Scales are used both for continuous variables, such as numbers, and categorical ones such as factors.

6.3.5 Themes

How the plots look when displayed or printed can be altered by means of themes. A plot can be saved without adding a theme and then printed or displayed using different themes. Also individual theme elements can be changed, and whole new themes defined. This adds a lot of flexibility and helps in the separation of the data representation aspects from those related to the graphical design.

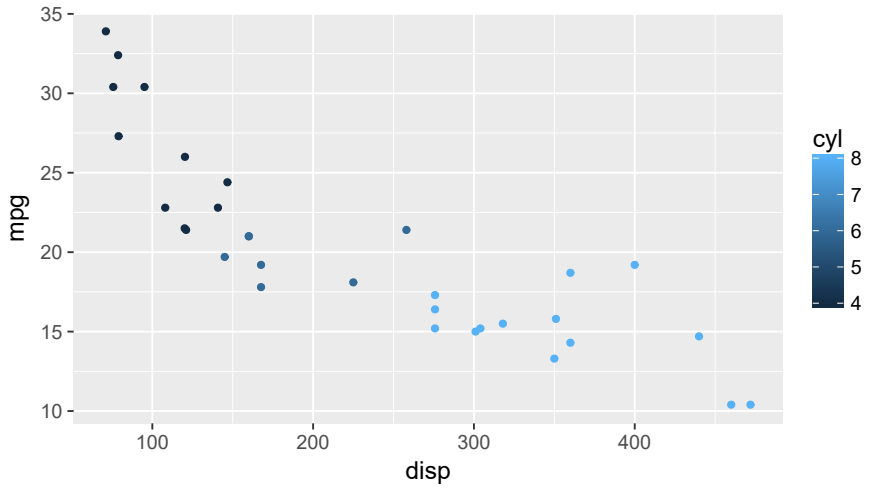
As discussed above the grammar of graphics is based on aesthetics (`aes`) as for example color, geometric elements `geom_…` such as lines, and points, statistics `stat_…` , scales `scale_…` , labels `labs` , and themes `theme_…` . Plots are assembled from these elements, we start with a plot with two aesthetics, and one geometry.

6.4 Scatter plots

In the examples that follow we will use the `mtcars` data set included in R. To learn more about this data set, `help("mtcars")` at the R command prompt.

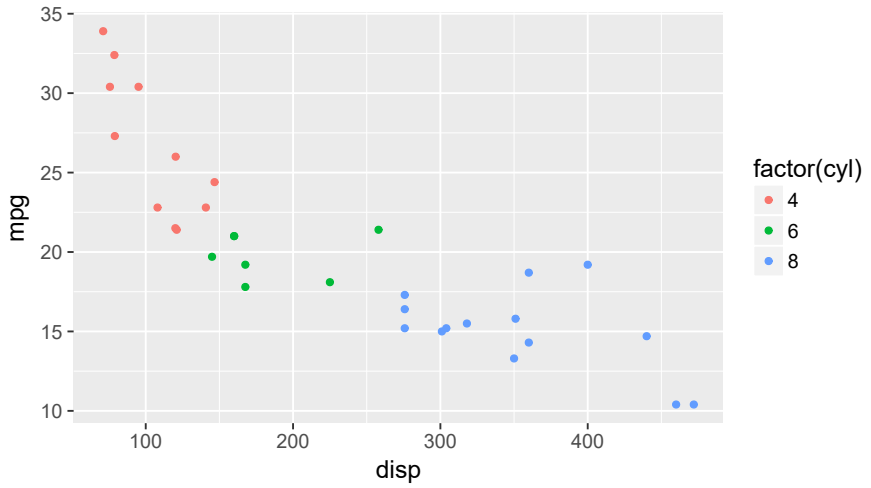
Data variables can be 'mapped' to *aesthetics*. Variables to be represented in a plot can be either continuous (numeric) or discrete (categorical, factor). Variable `cyl` is encoded in the `mtcars` data frame as numeric values. Even though only three values are present, a continuous color scale is used by default.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, colour = cyl)) +
  geom_point()
```



Some scales exist in two 'flavours', one suitable for continuous variables and another for discrete variables. We can convert `cyl` into a factor 'on-the-fly' to force the use of a discrete color scale.

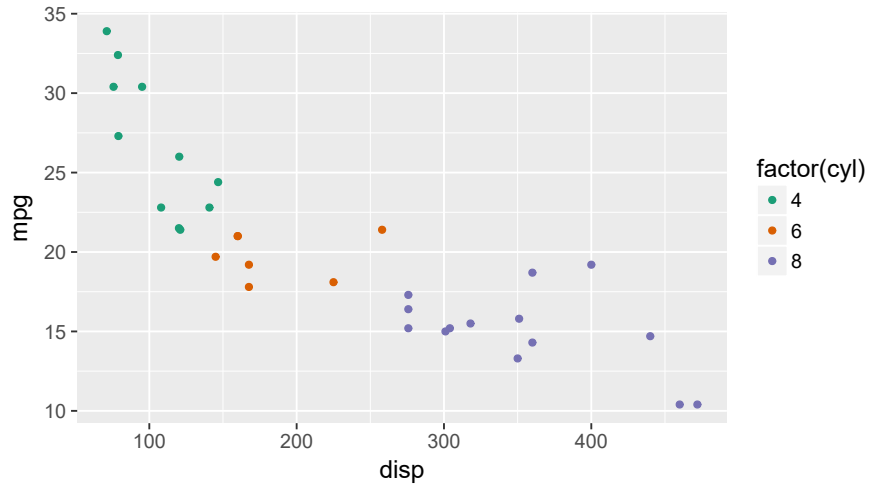
```
ggplot(data = mtcars, aes(x = disp, y = mpg, colour = factor(cyl))) +
  geom_point()
```



Using an aesthetic, involves the mapping of values in the data to aesthetic values such as colours. The mapping is defined by means of scales. If we now consider the `colour` aesthetic in the previous statement, a default discrete colour scale was used. In this case if we would like different

colours used for the three values, but still have them selected automatically, we can select a different colour palette:

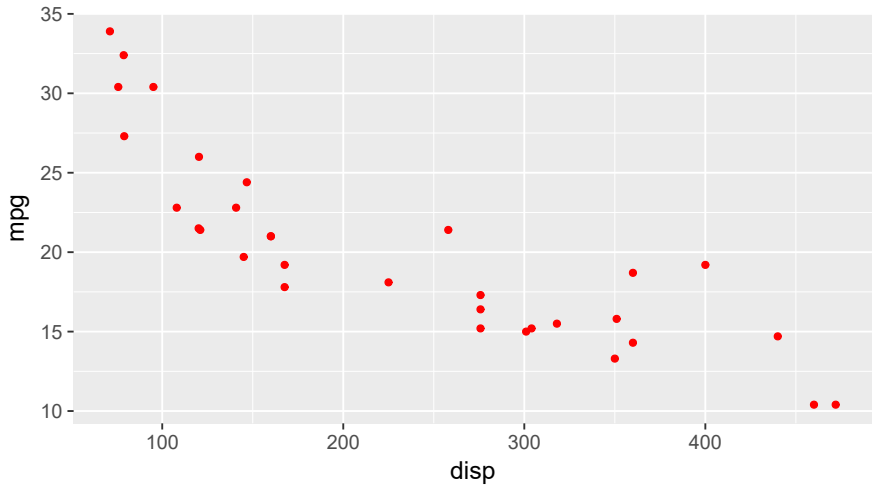
```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point() +
  scale_color_brewer(type = "qual", palette = 2)
```



Within `aes()` the aesthetics are interpreted as being a function of the values in the data—i.e. to be a mapped. If given outside `aes()` they are interpreted as constants values, which apply to one geom if given within the call to `geom_xxx` but outside `aes()`. The aesthetics and data given as `ggplot()`'s arguments become the defaults for all the geoms, but geoms also accept aesthetics and data as arguments, which when supplied locally override the whole-plot defaults. In the example below, we override the default colour of the points.

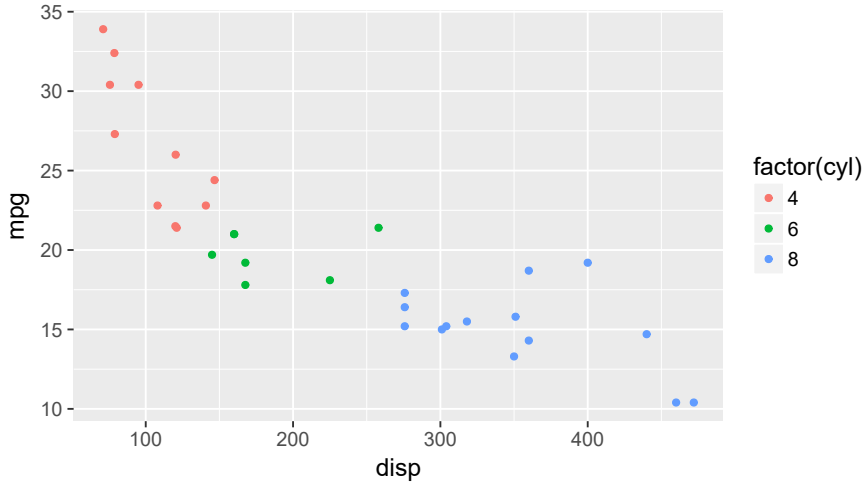
If we *set* the `color` aesthetic to a constant value, `"red"`, all points are plotted in red.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point(color = "red")
```



If we *map* the `color` aesthetic to a variable, `factor(cyl)`, points get colors according to the levels of the factor, and by default a `guide` or *key* for the mapping is also added.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point()
```

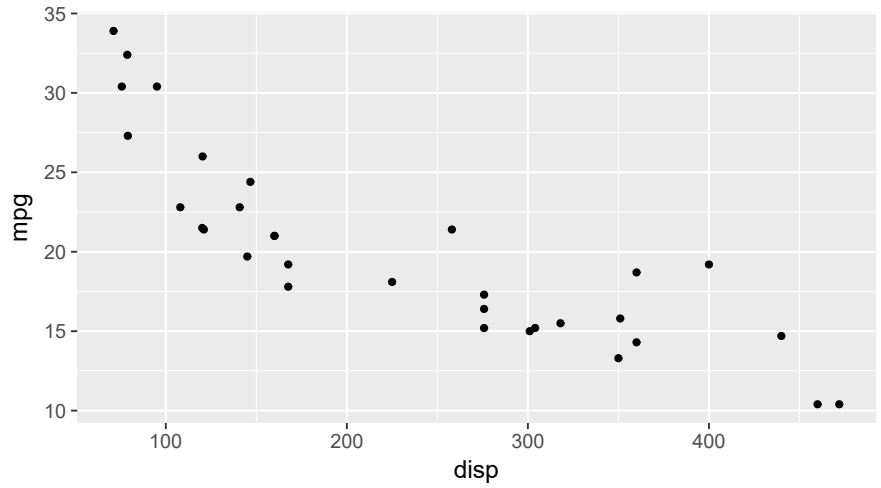


As with any R function it is possible to pass arguments by position to `aes` when mapping variables to *aesthetics* but this makes the code more difficult to read and less tolerant to possible changes to the definitions

of functions. It is not recommended to use this terse style in scripts or package coding. However, it can be used by experienced users at the command prompt usually without problems.

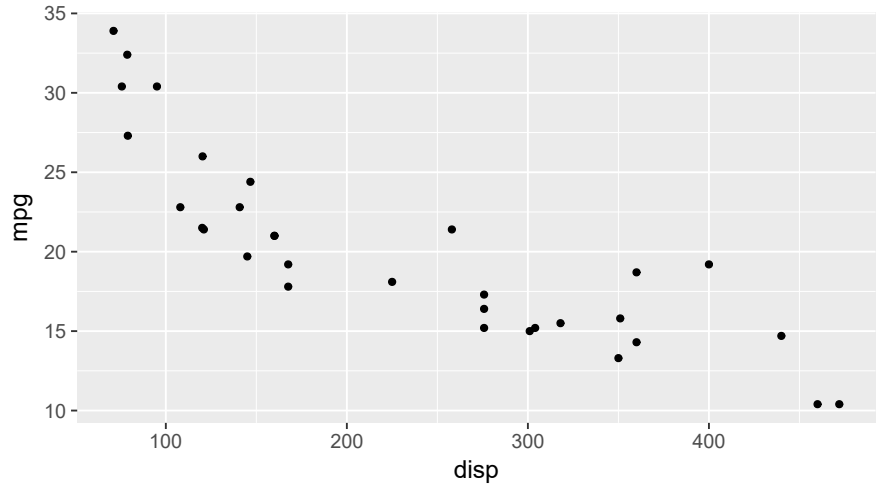
Mapping passing arguments by *name* to `aes`.

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point()
```



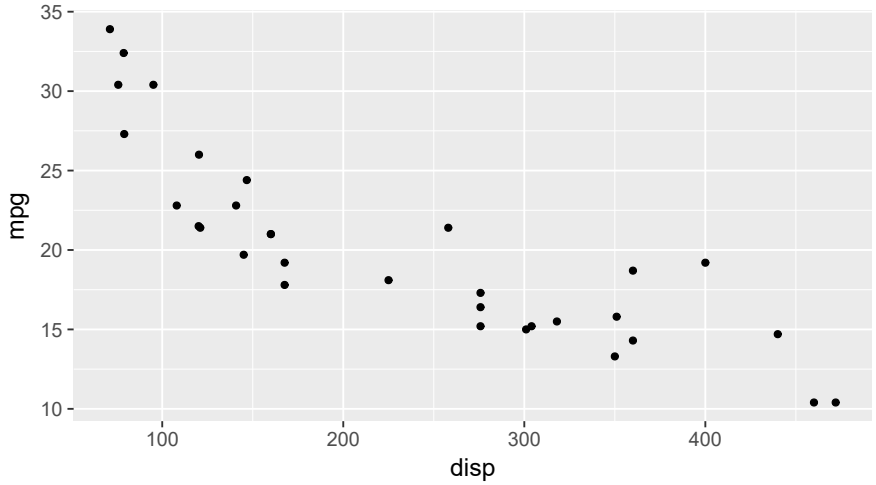
If we swap the order of the arguments we still obtain the same plot.

```
ggplot(data = mtcars, aes(y = mpg, x = disp)) +
  geom_point()
```



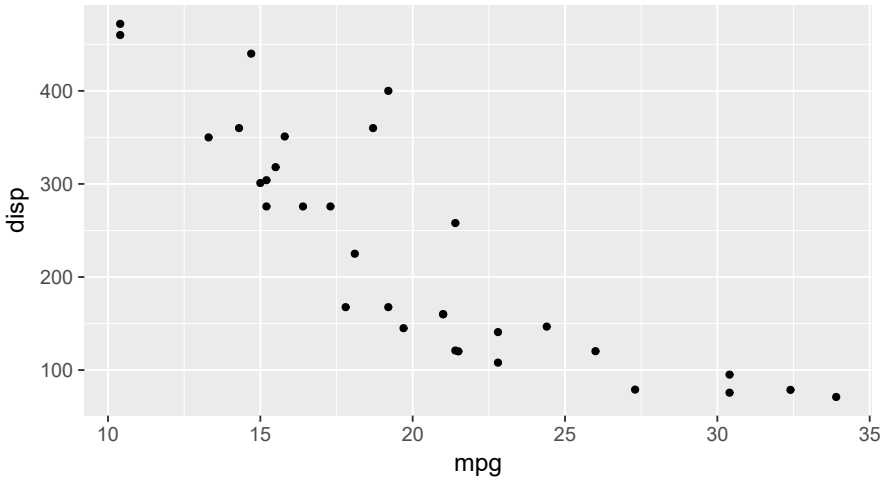
Mapping passing arguments by *position* to `aes`.

```
ggplot(mtcars, aes(disp, mpg)) +
  geom_point()
```



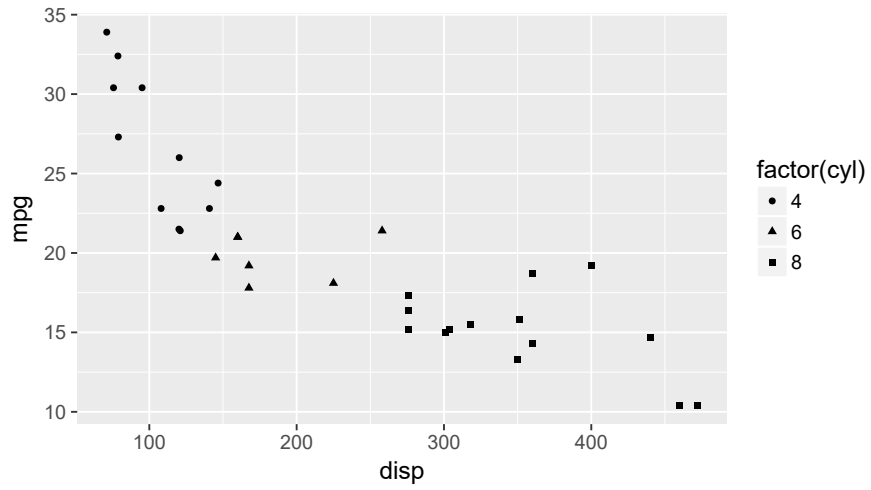
If we swap the order of the arguments we obtain a different plot as the mappings are swapped.

```
ggplot(mtcars, aes(mpg, disp)) +
  geom_point()
```



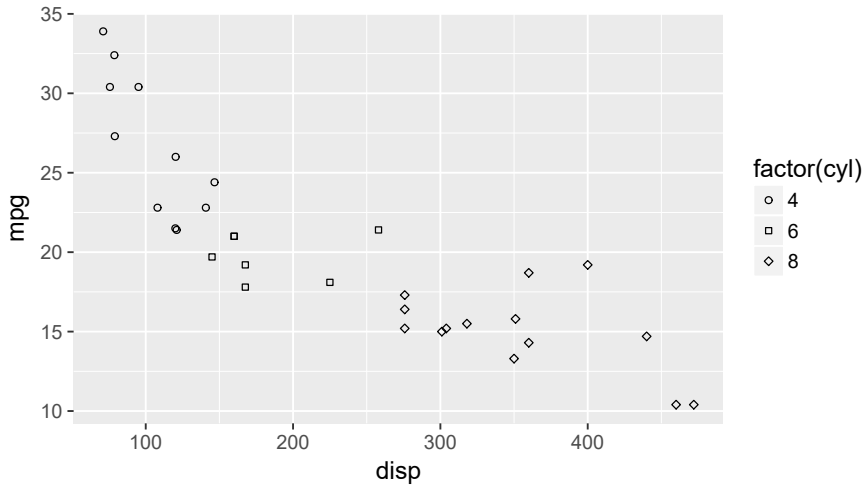
When not relying on colors, the most common way of distinguishing groups of observations in scatter plots is to use the `shape` of the points as an *aesthetic*.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point()
```



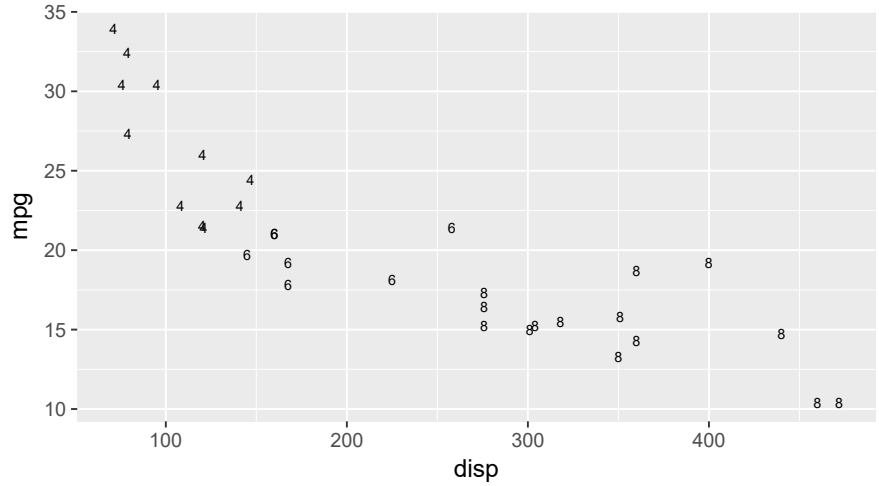
We can use `scale_shape_manual` to choose each shape to be used. We set three "open" shapes that we will see later are very useful as they obey both `color` and `fill` *aesthetics*.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point() +
  scale_shape_manual(values = c(21, 22, 23))
```



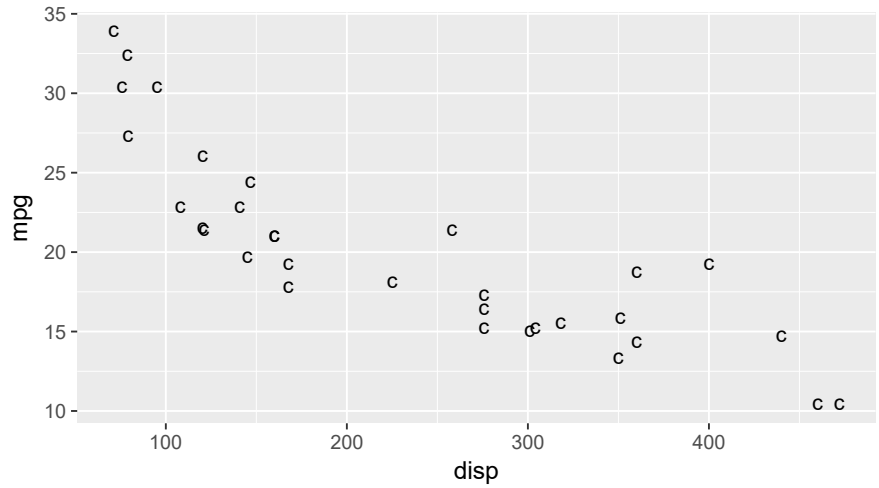
It is also possible to use character strings as shapes. The character is centred on the position of the observation. Conceptually using `character` values for `shape` is different to using `geom_text` as in the later case there is much more flexibility as character strings and expressions are allowed in addition to single characters. Also positioning with respect to the coordinates of the observations can be asjusted through justification. While `geom_text` is usually used for annotations, the present example treats the character string as a symbol. (This also opens the door to the use as shapes of symbols defined in special fonts.)

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point(size = 2.5) +
  scale_shape_manual(values = c("4", "6", "8"), guide = FALSE)
```



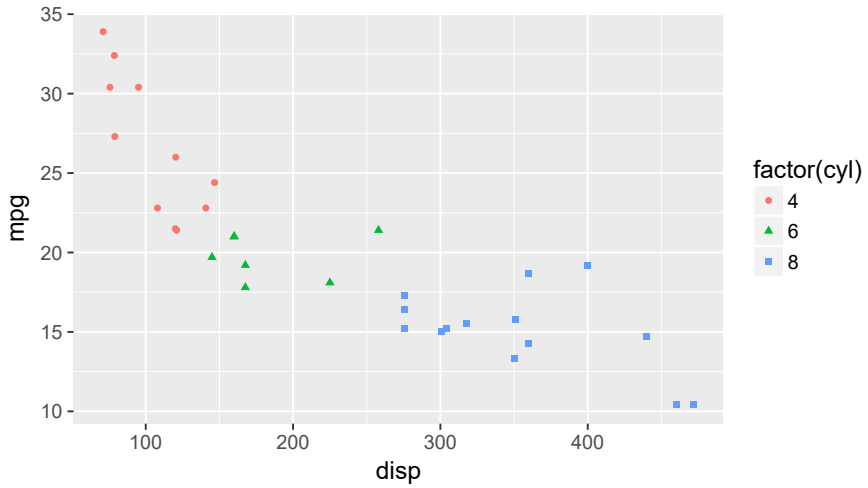
Only the first character of the string is used, and this is sometimes limiting, as with two digit numbers or alphanumeric codes. For example the second character is ignored in the code below.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point(size = 4) +
  scale_shape_manual(values = c("c4", "c6", "c8"), guide = FALSE)
```



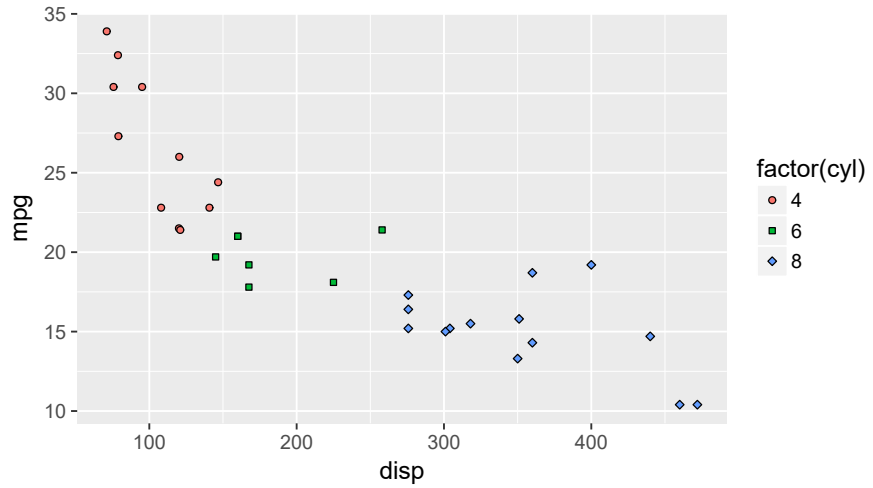
As seen earlier one variable can be mapped to more than one aesthetic allowing redundant aesthetics. This may seem wasteful, but it is extremely useful as it allows one to produce figures that even when produced in color, can still be read if reproduced as monochrome images.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          shape = factor(cyl),
                          color = factor(cyl))) +
  geom_point()
```



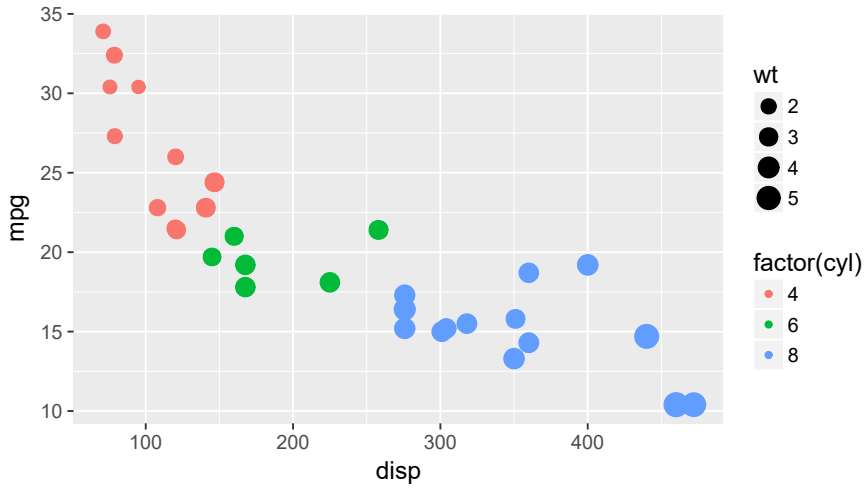
Here we will map `fill` to `cyl`, and we could in addition map `color` to a different grouping as color controls color of the border of the shapes.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          shape = factor(cyl),
                          fill = factor(cyl))) +
  geom_point() +
  scale_shape_manual(values = c(21, 22, 23))
```



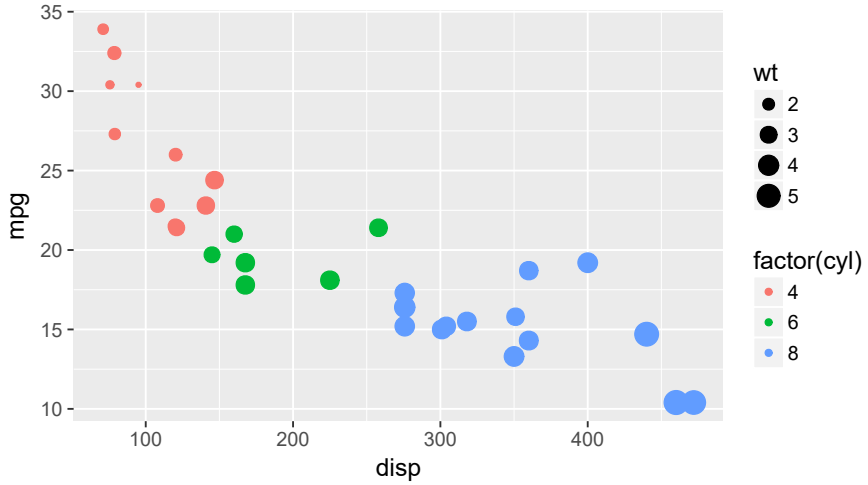
We can create a "bubble" plot by mapping the `size` *aesthetic* to a continuous variable. In this case, one has to think what is visually more meaningful. Although the radius of the shape is frequently mapped, in general due to how human perception works, mapping a variable to the area of the shape is in general more useful by being perceptually closer to a linear mapping. For this example we add a new variable to the plot. the weight of the car in tons and map it to the area of the points.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl),
                          size = wt)) +
  scale_size_area() +
  geom_point()
```



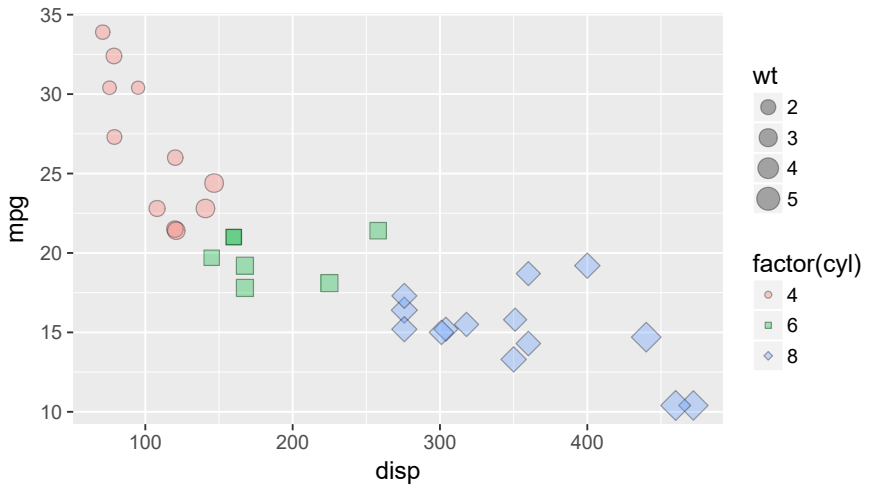
If we use a radius based scale the "impression" we get is that some cars had very light weight, which is not the case.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl),
                          size = wt)) +
  scale_size() +
  geom_point()
```



As a final example of how to combine different aesthetics, we use in a single plot several of the different mappings described in earlier examples.

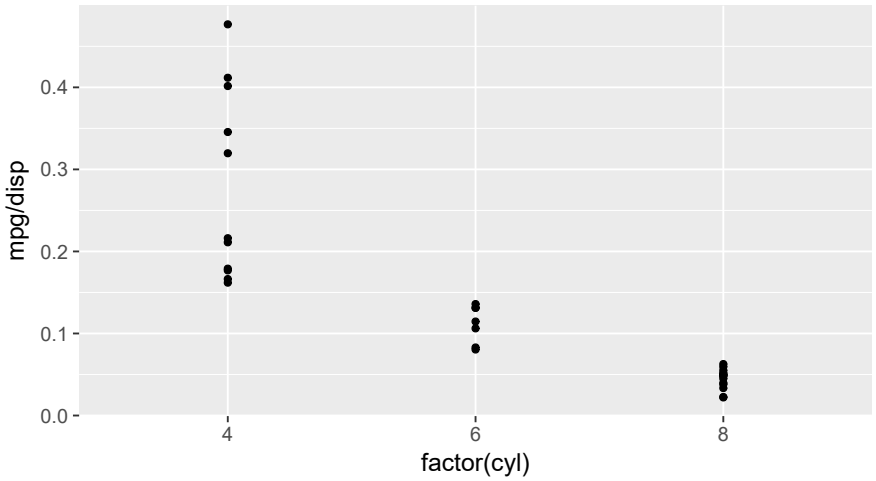
```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          shape = factor(cyl),
                          fill = factor(cyl),
                          size = wt)) +
  geom_point(alpha = 0.33, color = "black") +
  scale_size_area() +
  scale_shape_manual(values = c(21, 22, 23))
```



Data assigned to an *aesthetic* can be the 'result of a computation'. In other words, the values to be plotted do not need to be stored in the data frame passed as argument to `data`, the first formal parameter of `ggplot()`

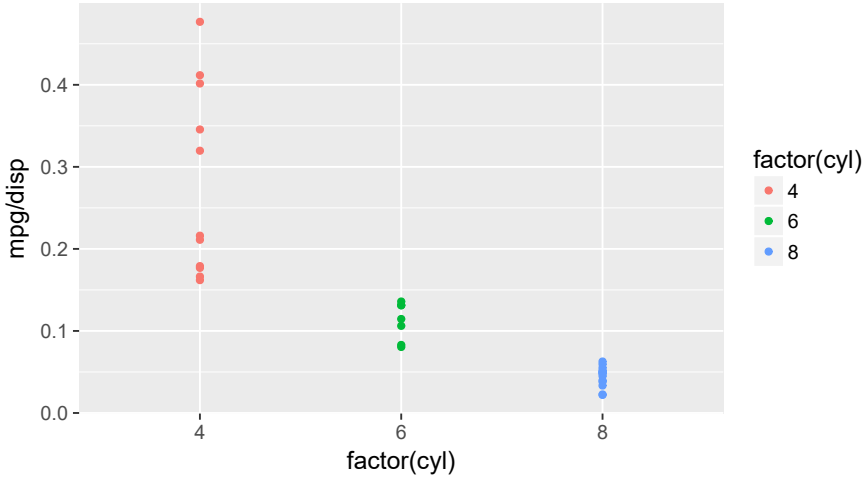
Here we plot the ratio of miles-per-gallon, `mpg`, and the engine displacement (volume), `disp`. Instead of mapping as above `disp` to the x *aesthetic*, we map `factor(cyl)` to x. In contrast to the continuous variable `disp` we earlier used, now we use a factor, so a discrete (categorical) scale is used by default for x.

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg / disp)) +
  geom_point()
```



Although `factor(cyl)` is mapped to x, we can map it in addition to `color`. This may be useful when desiring to keep the design consistent across plots, for example this one and those above.

```
ggplot() +
  aes(x = factor(cyl), y = mpg / disp,
      colour = factor(cyl)) +
  geom_point(data = mtcars)
```



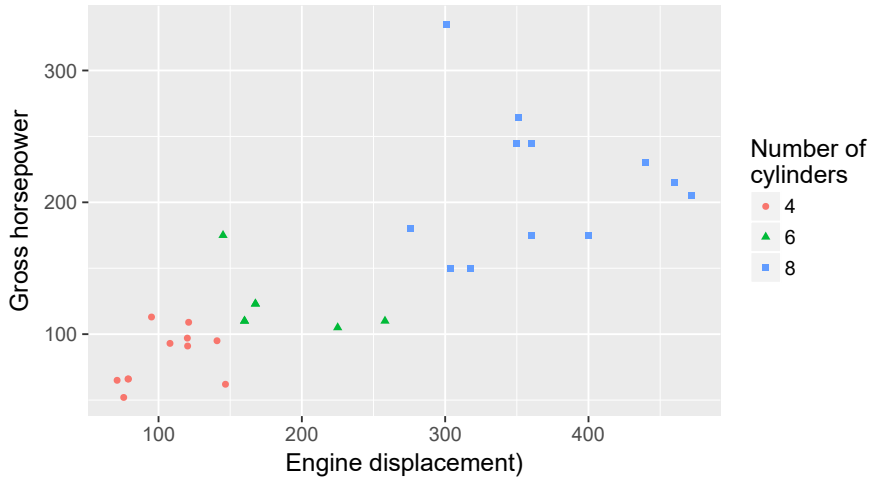
The code in the next chunk is also valid, it returns a blank plot. This apparently useless plot, can be very useful when writing functions that

return `ggplot` objects or build them piece by piece in a loop.

```
ggplot()
```


We can set the labels for the different aesthetics, and give a title (`\n` means 'new line' and can be used to continue a label in the next line). In this case, if two aesthetics are linked to the same variable, the labels supplied should be identical, otherwise two separate *keys* will be produced.

```
ggplot(data = mtcars,
       aes(x=disp, y=hp, colour=factor(cyl),
           shape=factor(cyl))) +
  geom_point() +
  labs(x="Engine displacement)",
       y="Gross horsepower",
       colour="Number of\ncylinders",
       shape="Number of\ncylinders")
```



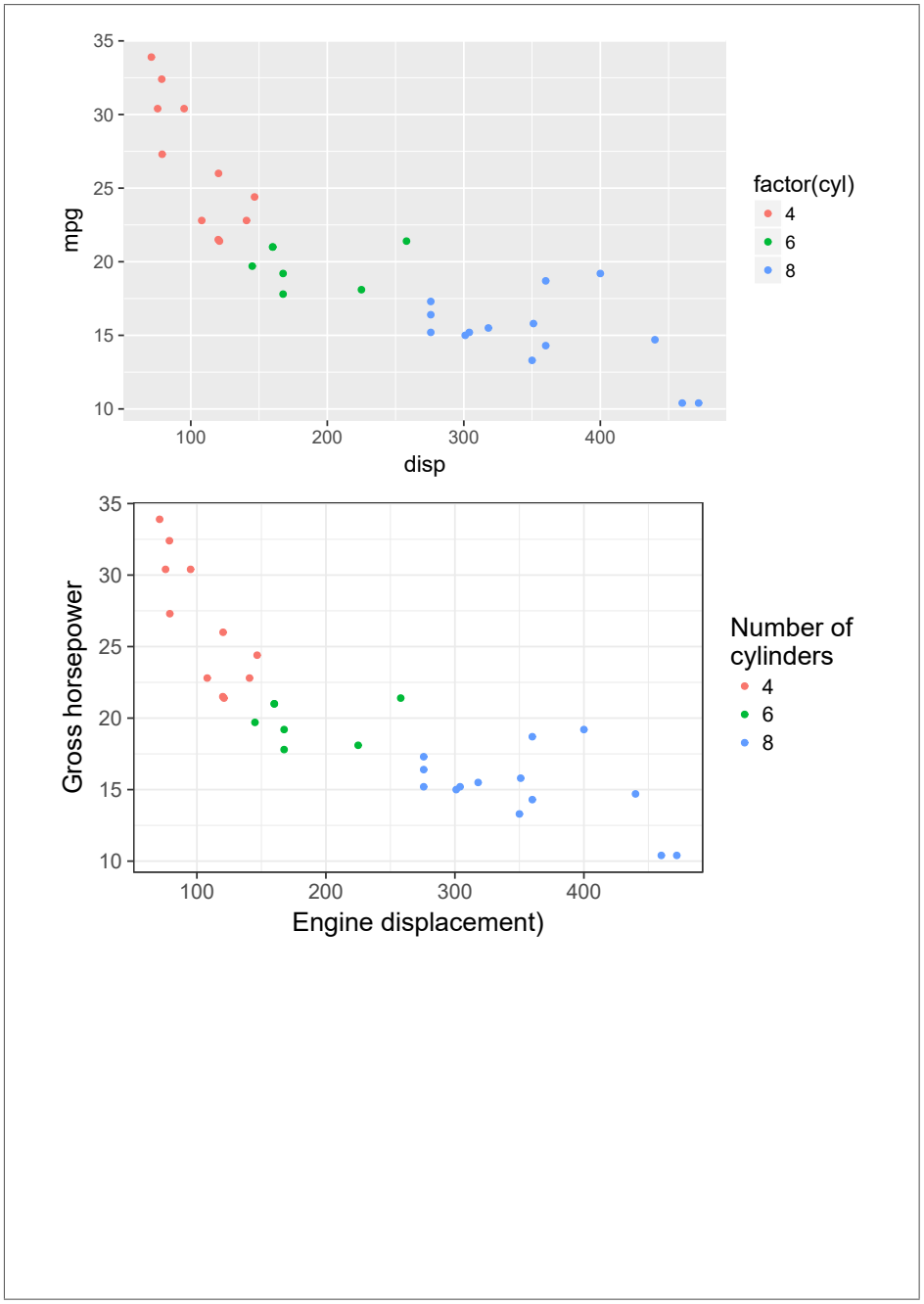
Please, see section ?? on page ?? for more an extended description of the use of `labs`.

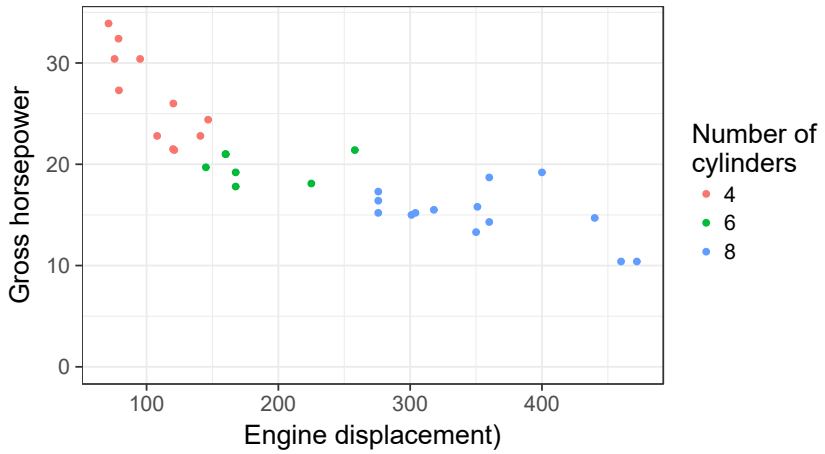
We can assign a ggplot object or a part of it to a variable, and then assemble a new plot from the different pieces.

```
myplot <- ggplot(data = mtcars,
                 aes(x=disp, y=mpg,
                 colour=factor(cyl))) +
          geom_point()

mylabs <- labs(x="Engine displacement)",
               y="Gross horsepower",
               colour="Number of\ncylinders",
               shape="Number of\ncylinders")
```

And now we can assemble them into plots.

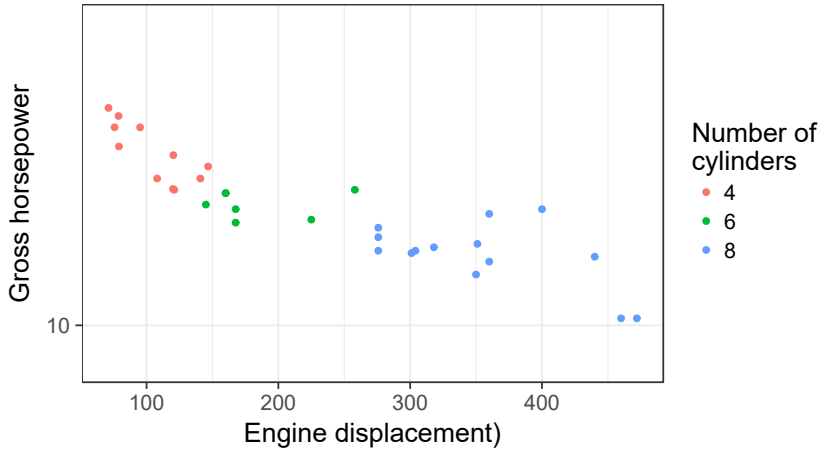
```
myplot
myplot + mylabs + theme_bw(16)
myplot + mylabs + theme_bw(16) + ylim(0, NA)
```


35
30
25
20
15
10
mpg
100
200
300
400
disp
factor(cyl)
4
6
8
Gross horsepower
Engine displacement)
Number of cylinders



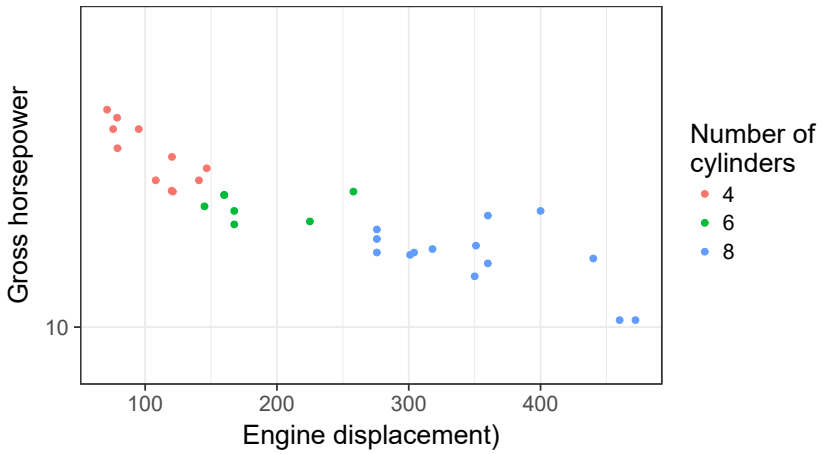
We can also save intermediate results.

```
mylogplot <- myplot + scale_y_log10(limits=c(8,55))
mylogplot + mylabs + theme_bw(16)
```



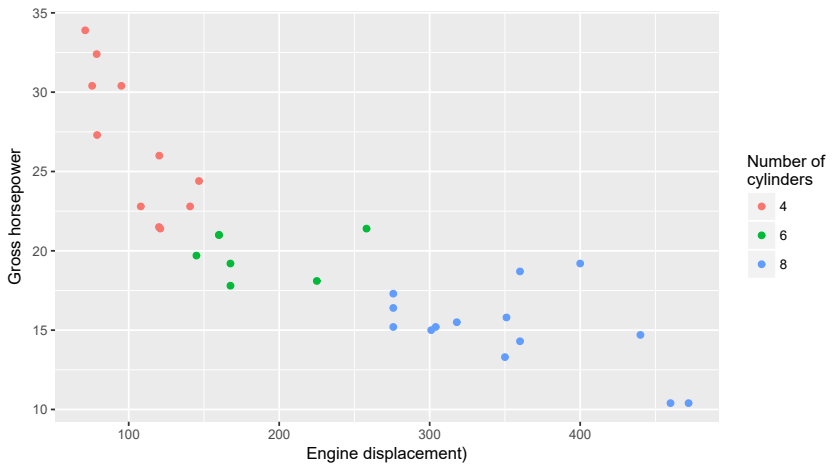
If the pieces to put together do not include a "ggplot" object, we can put them into a "list" object.

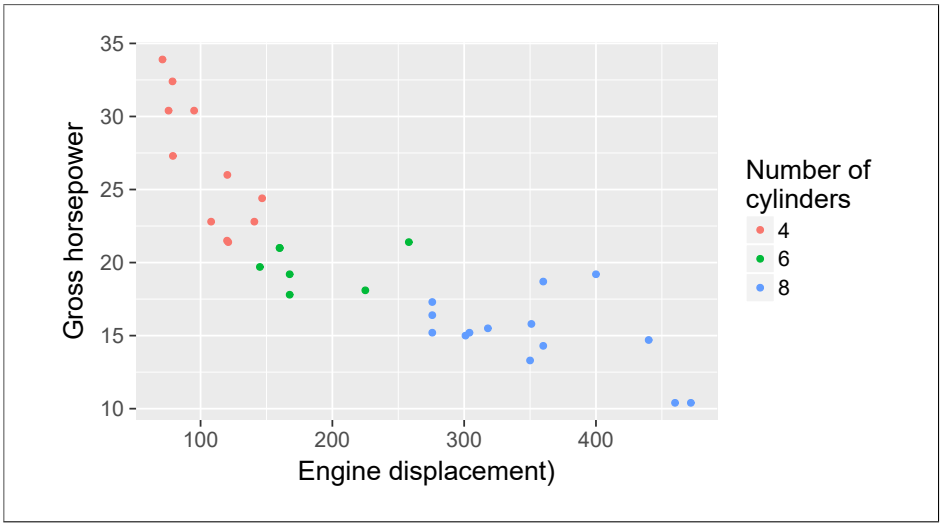
```
myparts <- list(mylabs, theme_bw(16))
mylogplot + myparts
```



The are a few predefined themes in package 'ggplot2'and additional ones in other packages such as 'cowplot', even the default `theme_grey` can come in handy because the first parameter to themes is the point size used as reference to calculate all other font sizes. You can see in the two examples bellow, that the size of all text elements changes proportionally.

```
myplot + mylabs + theme_grey(10)
myplot + mylabs + theme_grey(16)
```

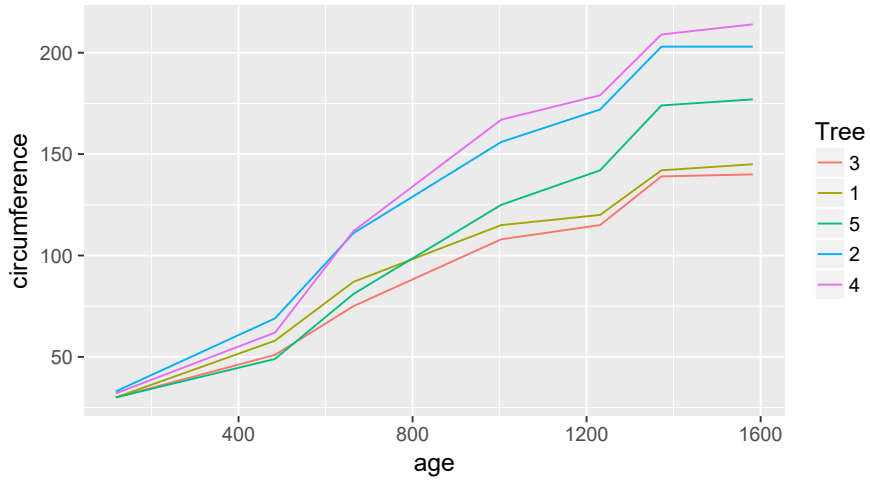




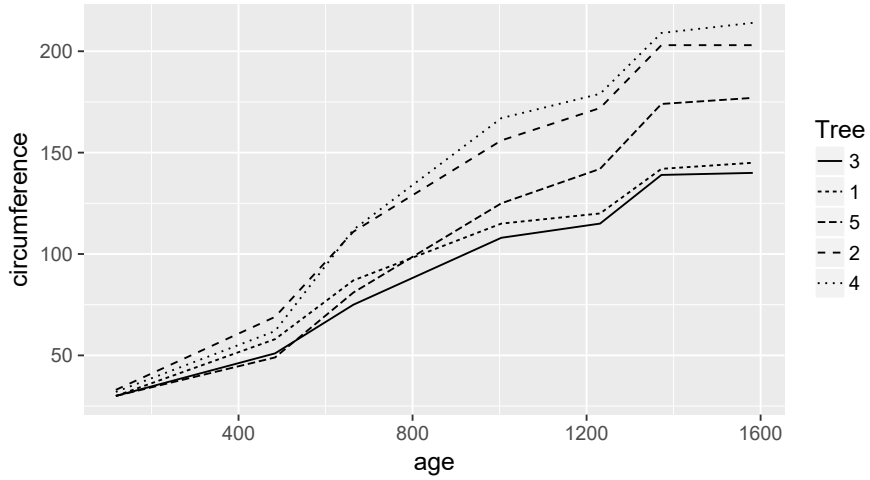
6.5 Line plots

For line plots we use `geom_line`. The `size` of a line is its thickness, and as we had `shape` for points, we have `linetype` for lines. In a line plot observations in successive rows of the data frame, or the subset corresponding to a group, are joined by straight lines. We use a different data set included in R, Orange, with data on the growth of five orange trees. See the help page for `Orange` for details.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line()
```




```
ggplot(data = Orange,
       aes(x = age, y = circumference, linetype = Tree)) +
  geom_line()
```

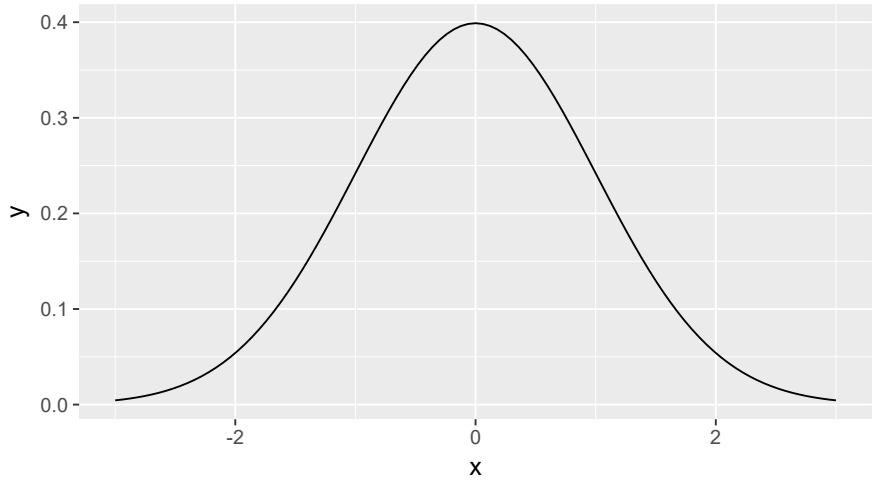


Much of what was described above for scatter plots can be adapted to line plots.

6.6 Plotting functions

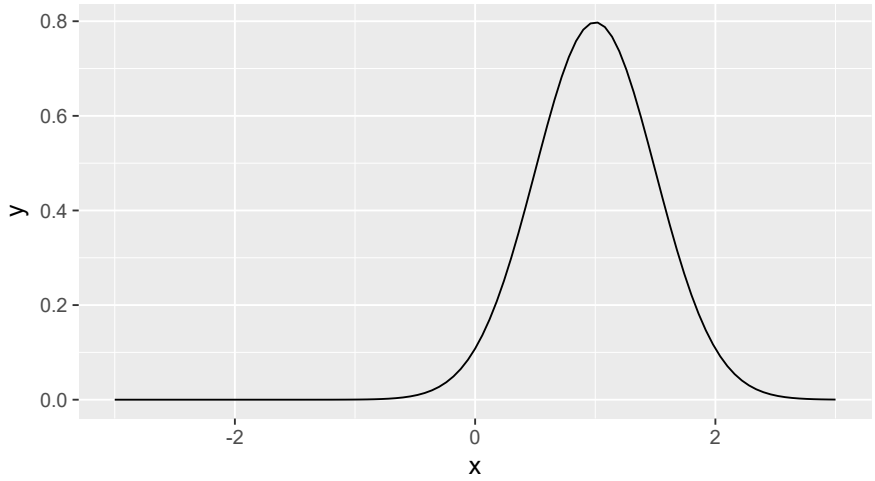
In addition to plotting data from a data frame with variables to map to x and y *aesthetics*, it is possible to have only a variable mapped to x and use `stat_function` to generate the values to be mapped to y using a function. This avoids the need to generate data beforehand (the number of data points to be generated can be also set).

```
ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun=dnorm)
```



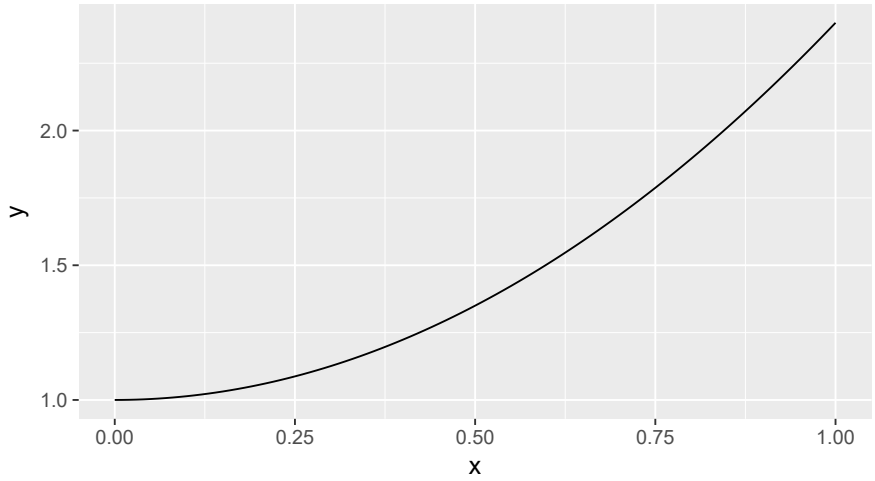
Using a list we can even pass by name additional arguments to a function.

```
ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun = dnorm, args = list(mean = 1, sd = .5))
```



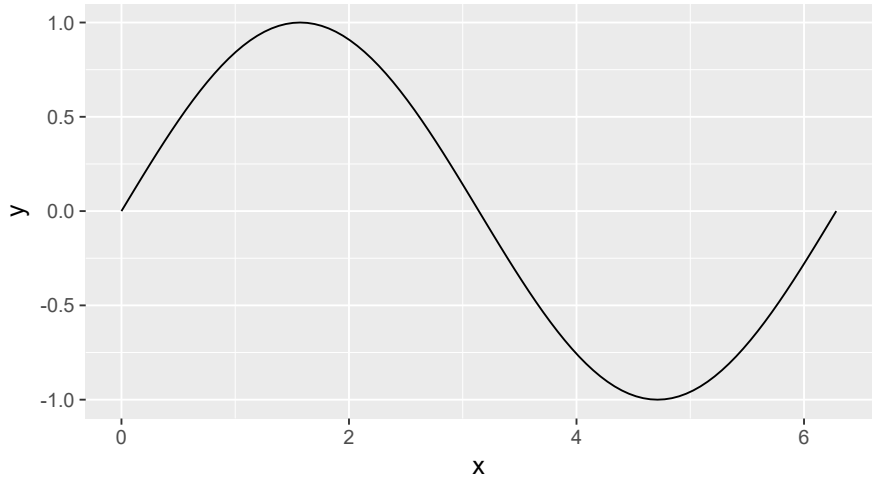
Of course, user-defined functions (not shown), and anonymous functions can also be used.

```
ggplot(data.frame(x=0:1), aes(x=x)) +
  stat_function(fun = function(x, a, b){a + b * x^2},
                args = list(a = 1, b = 1.4))
```



Here is an example of a predefined function, but in this case the default breaks (tick positions) are not the best.

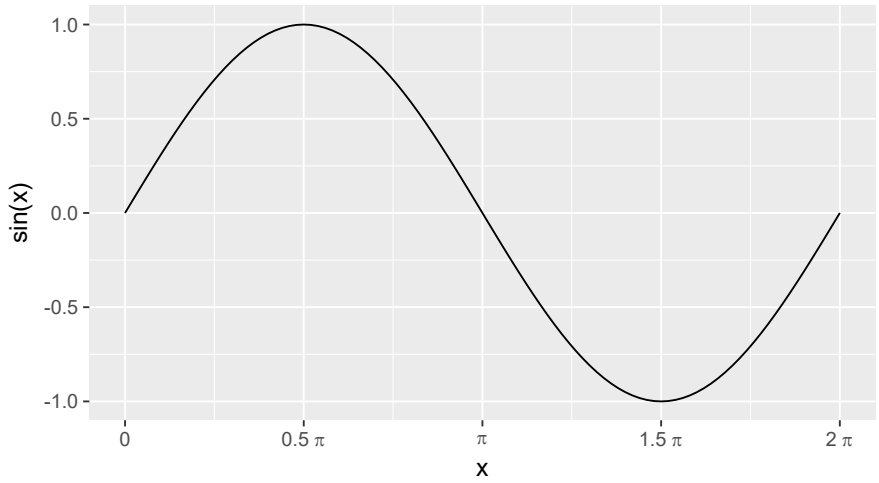
```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin)
```



We need to change the x-axis scale to better suit the sin function and the use of radians as angular units[1].

```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
  labs(y="sin(x)")
```

[1] The use of `expression` is explained in detail in section **??**, an the use of `scales` in section **??**.

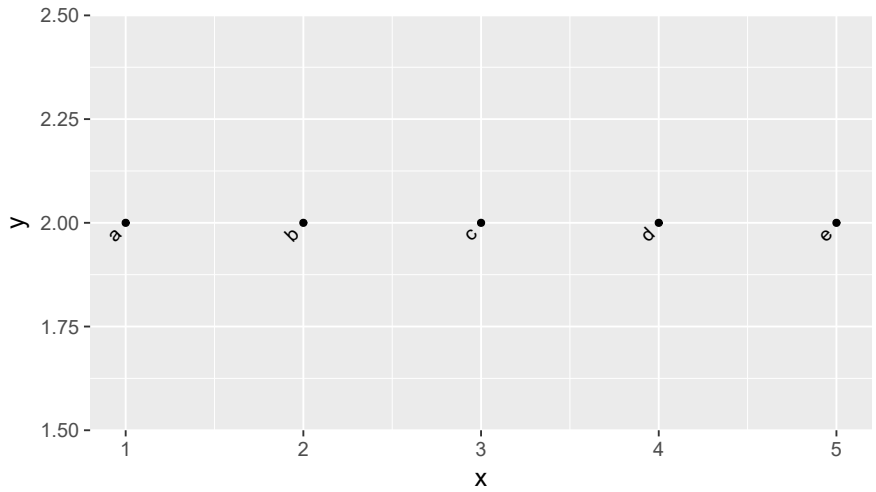


6.7 Plotting text and expressions

We can use `geom_text` or `geom_label` to add text labels to observations. For `geom_text`, the aesthetic `label` gives text and the usual aesthetics `x` and `y` the location of the labels. As one would expect the `color` aesthetic can be also used for the text. In addition `angle` and `vjust` and `hjust` can be used to rotate the label, and adjust its position. The default value of 0.5 for both `hjust` and `vjust` centres the label. The centre of the text is at the supplied `x` and `y` coordinates. 'Vertical' and 'horizontal' for justification refer to the text, not the plot. This is important when `angle` is different from zero. Negative justification values, shift the label left or down, and positive values right or up. A value of 1 or 0 sets the text so that its edge is at the supplied coordinate. Values outside the range $0 \ldots 1$ shift the text even further away, however, based on the length of the string. In the case of `geom_label` the text is enclosed in a rectangle, which obeys the `fill` *aesthetic*. However, it does not support rotation with `angle`.

```
my.data <-
  data.frame(x=1:5, y=rep(2, 5),
             label=paste(letters[1:5], " "))

ggplot(my.data, aes(x,y,label=label)) +
```

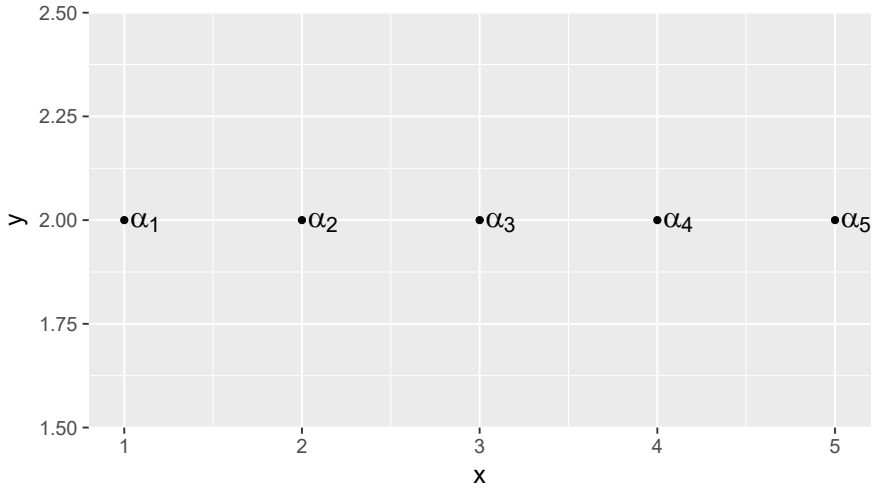
```
  geom_text(angle=45, hjust=1) + geom_point()
```



In this example we use `paste()` (which uses recycling here) to add a space at the end of each label. Justification values outside the range $0 \ldots 1$ are allowed, but are relative to the width of the label. As the default font used in this case has variable width characters, the justification would be inconsistent (e.g. try the code above but using `hjust` set to 3 instead of to 1 without pasting a space character to the labels.)

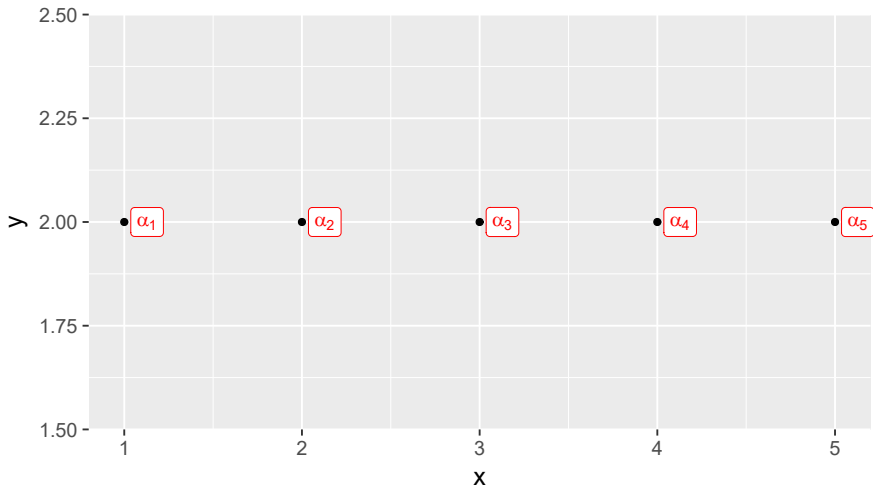
Plotting expressions (mathematical expressions) involves passing as `label` data character strings that can be parsed as expressions, and setting `parse = TRUE`.

```
my.data <-
  data.frame(x=1:5, y=rep(2, 5),
             label=paste("alpha[", 1:5, "]", sep = ""))

ggplot(my.data, aes(x,y,label=label)) +
  geom_text(hjust=-0.2, parse=TRUE, size = 6) +
  geom_point()
```



A similar example using `geom_label`.

```
ggplot(my.data, aes(x, y, label = label)) +
  geom_label(hjust = -0.2, parse = TRUE, size = 4,
             fill = "white", colour = "red") +
  geom_point()
```



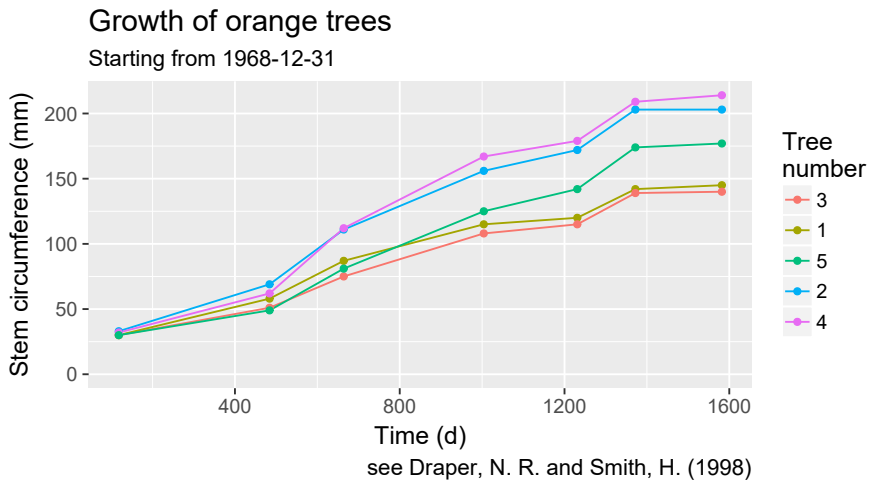
See R's 'plotmath' demo for more information on the syntax of expressions.

6.8 Axis- and key labels, titles, subtitles and captions

I describe this in the same section, and immediately after the section on plotting text labels, as they are added to plots using similar approaches. Be aware that the default jsutification of plot titles has changed in 'ggplot2'version 2.2.0 from centered to left justified. At the same time, support for subtitles and captions was added.

The most flexible approach is to use `labs` as it allows the user to set the text or expressions to be used for these different elements.

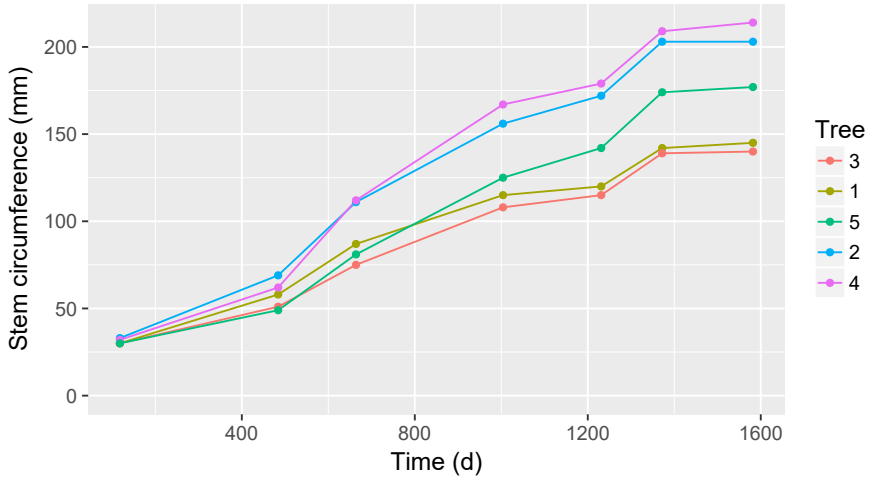
```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
  expand_limits(y = 0) +
  labs(title = "Growth of orange trees",
       subtitle = "Starting from 1968-12-31",
       caption = "see Draper, N. R. and Smith, H. (1998)",
       x = "Time (d)",
       y = "Stem circumference (mm)",
       color = "Tree\nnumber")
```



There are in addition `labs` some convenience functions for setting the axis labels, `xlab` and `ylab`.

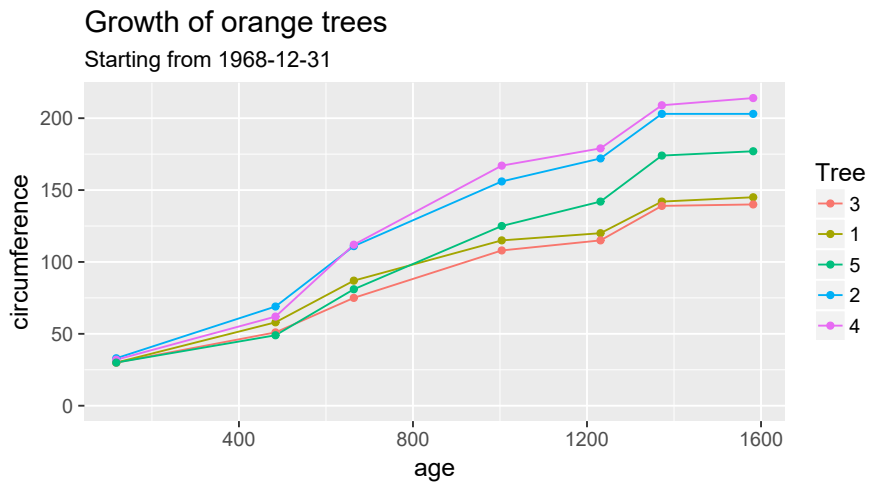
```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
```

```
  expand_limits(y = 0) +
  xlab("Time (d)") +
  ylab("Stem circumference (mm)")
```



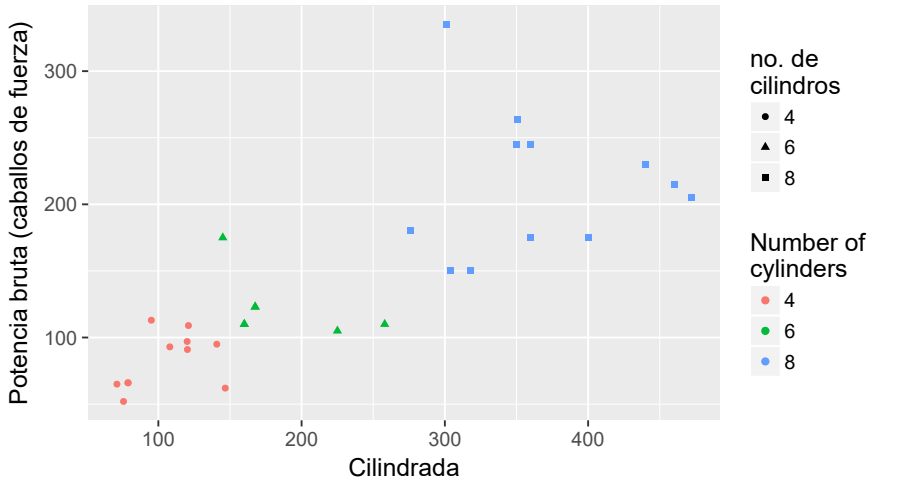
An additional convenience function, `ggtitle` can be used to add a title and optionally a subtitle.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
  expand_limits(y = 0) +
  ggtitle("Growth of orange trees",
          subtitle = "Starting from 1968-12-31")
```



Function `update_labels` allows the replacement of labels in an existing plot.

```
p <-
    ggplot(data = mtcars,
           aes(x=disp, y=hp, colour=factor(cyl),
               shape=factor(cyl))) +
      geom_point() +
      labs(x="Engine displacement)",
           y="Gross horsepower",
           color="Number of\ncylinders",
           shape="Number of\ncylinders")

update_labels(p, list(x = "Cilindrada",
                      y = "Potencia bruta (caballos de fuerza)",
                      color = "no. de\ncilindros",
                      shape = "no. de\ncilindros"))
```



The labels used in keys and axis tick-labels for factor levels can be changed through the different discrete and manual *scales* as described in section 6.16 on page 148.

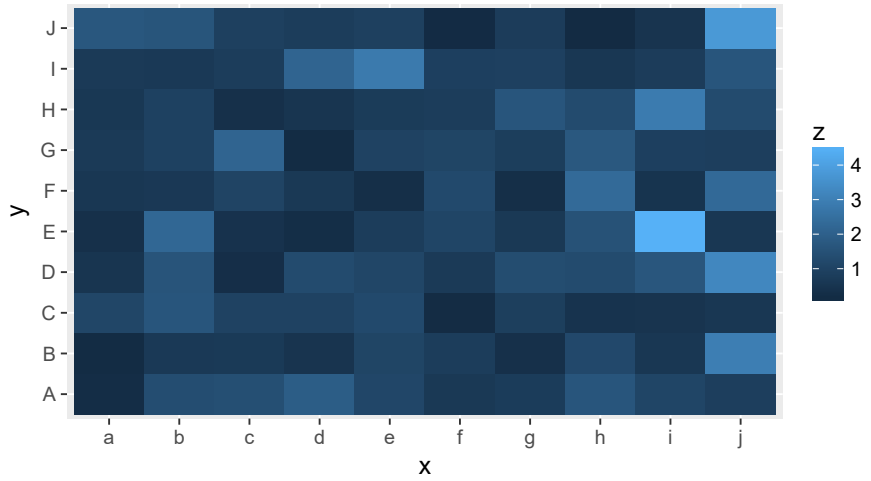
6.9 Tile plots

For the special case of heat maps see section 6.22.1 on page 171.

We here generate 100 random draws from the F distribution with degrees of freedom $\nu_1 = 5, \nu_2 = 20$.

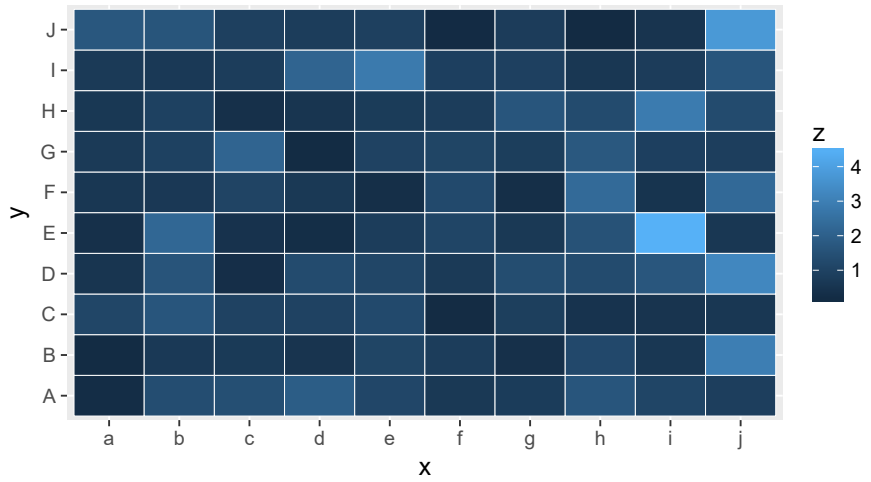
```
set.seed(1234)
randomf.df <- data.frame(z = rf(100, df1 = 5, df2 = 20),
                         x = rep(letters[1:10], 10),
                         y = LETTERS[rep(1:10, rep(10, 10))])
```

```
ggplot(randomf.df, aes(x, y, fill = z)) +
  geom_tile()
```



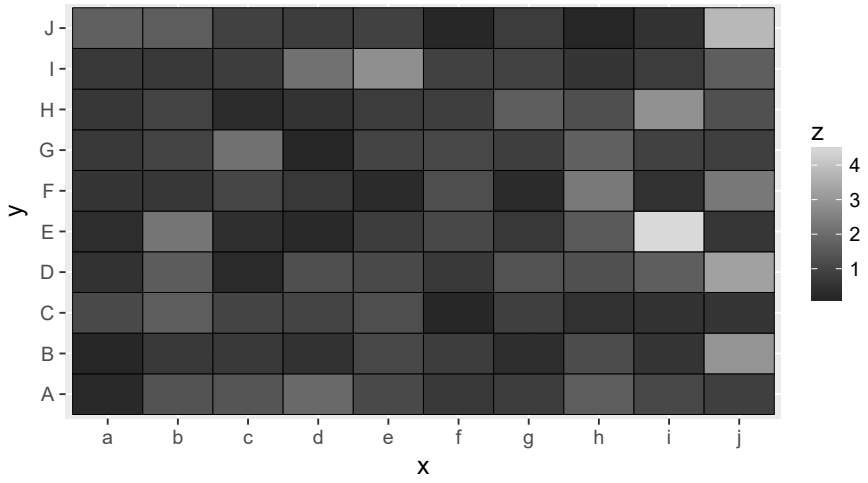
We can use `"white"` or some other contrasting color to better delineate the borders of the tiles.

```
ggplot(randomf.df, aes(x, y, fill = z)) +
  geom_tile(color = "white")
```



Any continuous fill scale can be used to control the appearance. Here we show a tile plot using a grey gradient.

```
ggplot(randomf.df, aes(x, y, fill = z)) +
  geom_tile(color = "black") +
  scale_fill_gradient(low = "grey15", high = "grey85", na.value = "red")
```

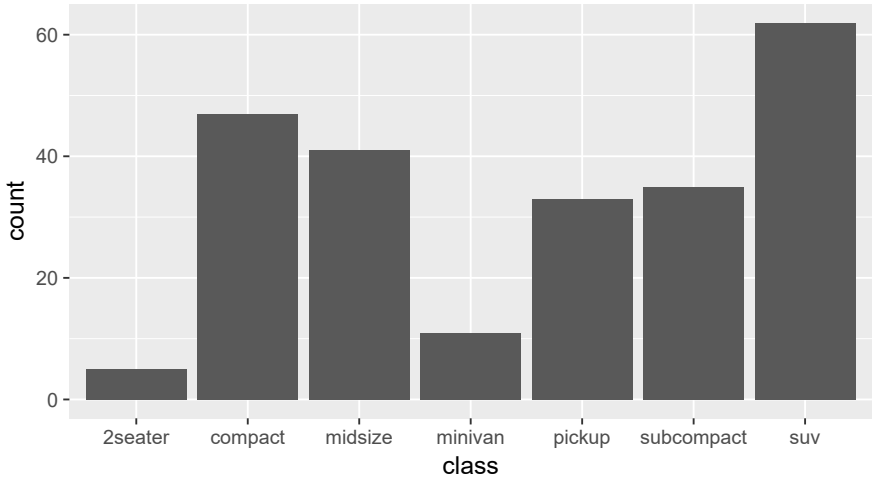


6.10 Bar plots

R users not familiar yet with 'ggplot2'are frequently surprised by the default behaviour of `geom_bar` as it uses `stat_count` to compute the value plotted, rather than plotting values as is (see section 6.12 on page 111). The default can be changed, but `geom_col` is equivalent to `geom_bar` used with `"identity"` as argument to parameter `stat`. The *statistic* `stat_identity` just echoes its input. In previous sections, as when plotting points and lines, this statistic was used by default.

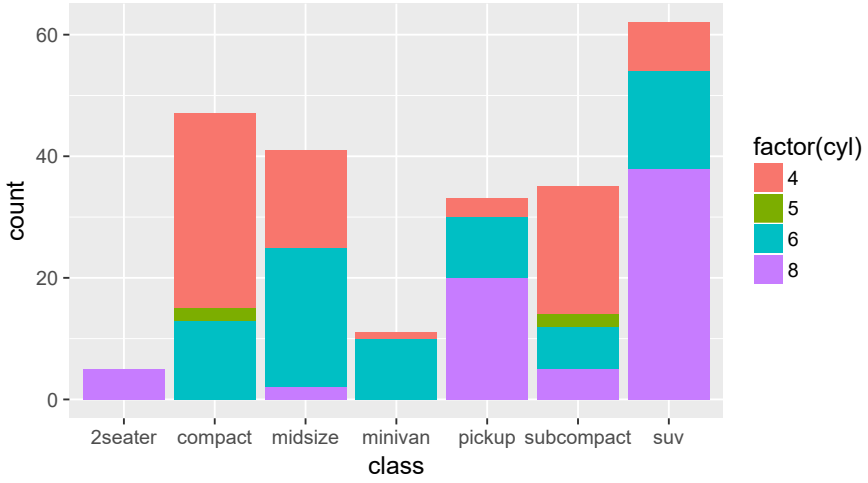
In this bar plot, each bar shows the number of observations in each `class` of car in the data set. We use a data set included in 'ggplot2'for this example based on the documentation.

```
ggplot(mpg, aes(class)) + geom_bar()
```



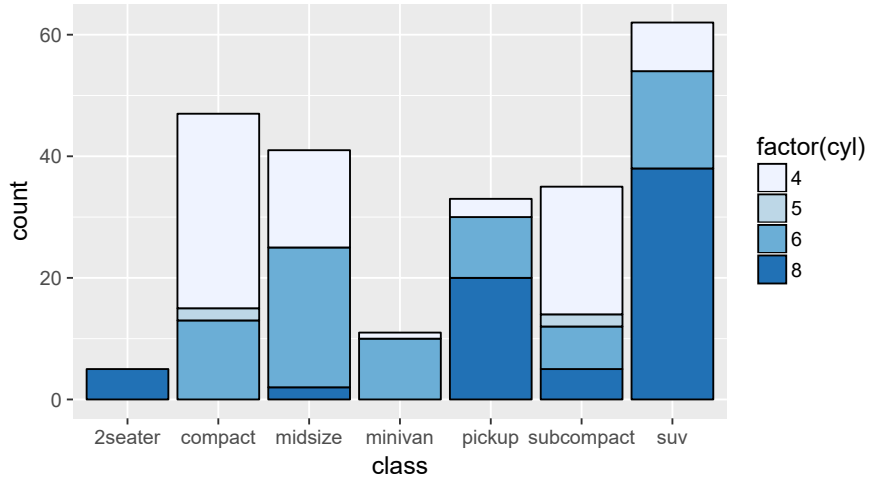
We can easily get stacked bars grouped by the number of cylinders of the engine.

```
ggplot(mpg, aes(class, fill = factor(cyl))) + geom_bar()
```



The default palette used for `fill` is rather ugly, so we also show the same plot with another scale for fill.

```
ggplot(mpg, aes(class, fill = factor(cyl))) +
  geom_bar(color = "black") +
  scale_fill_brewer()
```



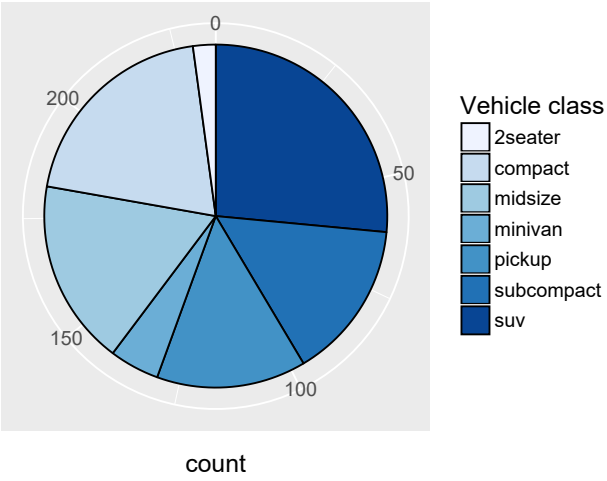
6.11 Circular plots

Under circular plots I include pie charts. Here we add a new "word" to the grammar of graphics, *coordinates*, such as `coord_polar()` in the next examples. The default coordinate system for x and y *aesthetics* is cartesian.

Pie charts are more difficult to read: our brain is more comfortable at comparing lengths than angles. If used, they should only be used to show composition, or fractional components that add up to a total. In this case only if the number of "pie slices" is small (rule of thumb: less than seven).

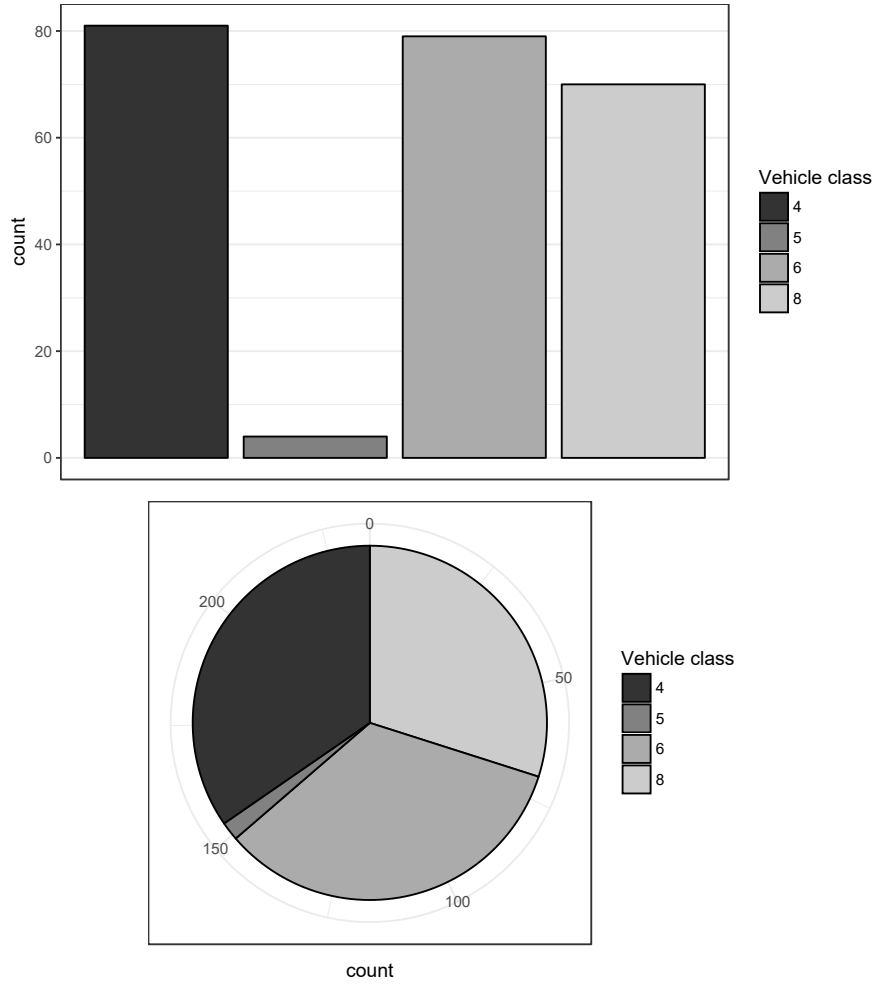
We make the equivalent of the first bar plot above. As we are still using `geom_bar` the default is `stat_count`. As earlier we use the brewer scale for nicer colors.

```
ggplot(data = mpg, aes(x = factor(1), fill = factor(class))) +
  geom_bar(width = 1, color = "black") +
  coord_polar(theta = "y") +
  scale_fill_brewer() +
  scale_x_discrete(breaks = NULL) +
  labs(x = NULL, fill = "Vehicle class")
```



Even with four slices pie charts can be difficult to read. Compare the following bar plot and pie chart.

```
ggplot(data = mpg, aes(x = factor(cyl), fill = factor(cyl))) +
  geom_bar(color = "black") +
  scale_fill_grey() +
  scale_x_discrete(breaks = NULL) +
  labs(x = NULL, fill = "Vehicle class") +
  theme_bw()

ggplot(data = mpg, aes(x = factor(1), fill = factor(cyl))) +
  geom_bar(width = 1, color = "black") +
  coord_polar(theta = "y") +
  scale_fill_grey() +
  scale_x_discrete(breaks = NULL) +
  labs(x = NULL, fill = "Vehicle class") +
  theme_bw()
```



6.12 Plotting summaries

The summaries discussed in this section can be superimposed on raw data plots, or plotted on their own. Beware, that if scale limits are manually set, the summaries will be calculated from the subset of observations within these limits. Scale limits can be altered when explicitly defining a scale or by means of functions `xlim()` and `ylim`. See section **??** for a way of constraining the viewport (the region visible in the plot) while keeping

the scale limits on a wider range of x and y values.

6.12.1 Statistical "summaries"

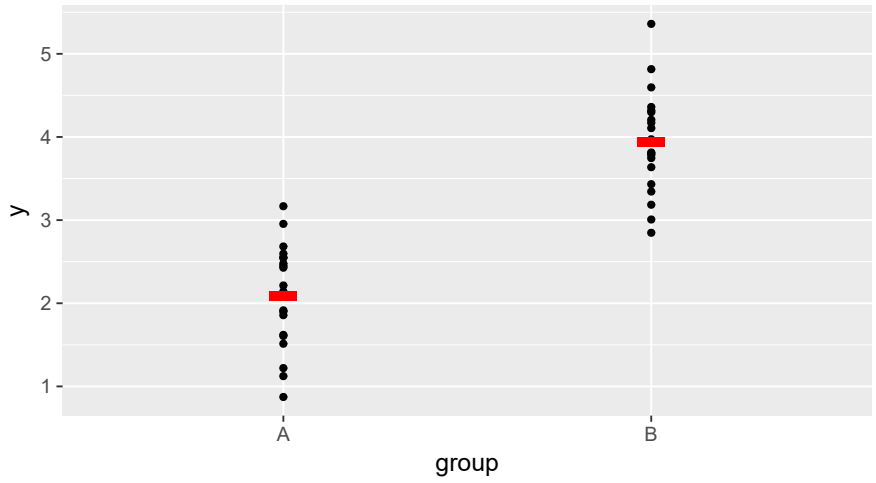
It is possible to summarize data on-the-fly when plotting. We describe in the same section the calculation of measures of central position and of variation, as `stat_summary` allows them to be calculated in the same function call.

For the examples we will generate some normally distributed artificial data.

```
fake.data <- data.frame(
  y = c(rnorm(20, mean=2, sd=0.5),
        rnorm(20, mean=4, sd=0.7)),
  group = factor(c(rep("A", 20), rep("B", 20)))
  )
```

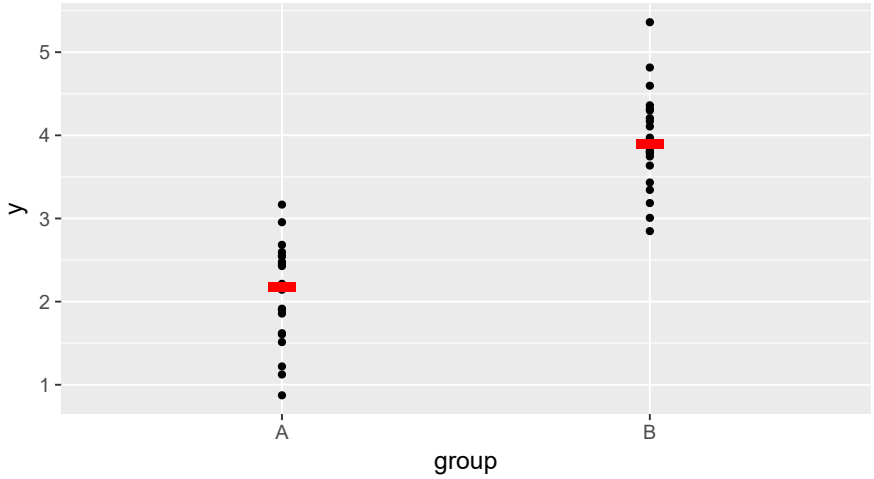
We first use scatter plots for the examples, later we give some additional examples for bar plots. We will reuse a "base" plot in a series of examples, so that the differences are easier to appreciate. We first add just the mean. In this case we need to pass as argument to `stat_summary` the `geom` to use, as the default one, `geom_pointrange`, expects data for plotting error bars in addition to the mean.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
 stat_summary(fun.y = "mean", geom="point", color="red", shape="-", size=20)
```



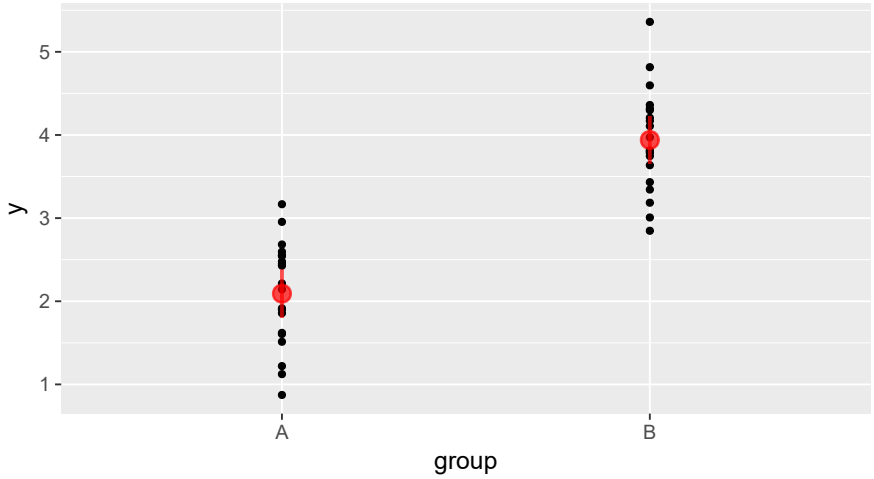
Then the median, by changing the argument passed to `fun.y`.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.y = "median", geom="point", colour="red", shape="-", size=20)
```



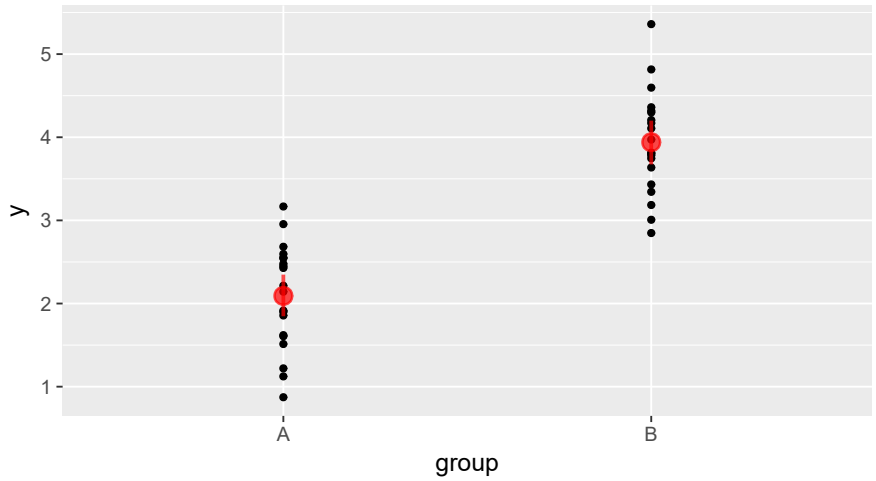
We can add the mean and $p = 0.95$ confidence intervals assuming normality (using the t distribution):

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_cl_normal", colour="red", size=1, alpha=0.7)
```



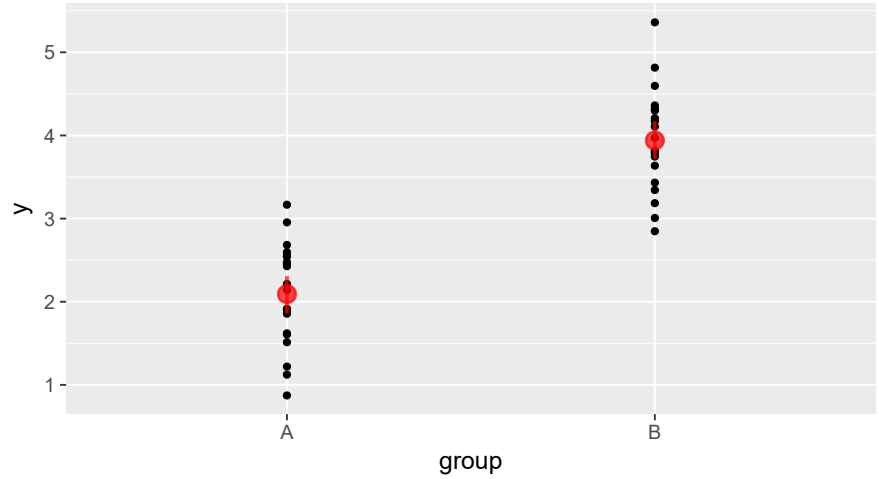
We can add the means and $p = 0.95$ confidence intervals not assuming normality (using the actual distribution of the data by bootstrapping):

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_cl_boot", colour="red", size=1, alpha=0.7)
```



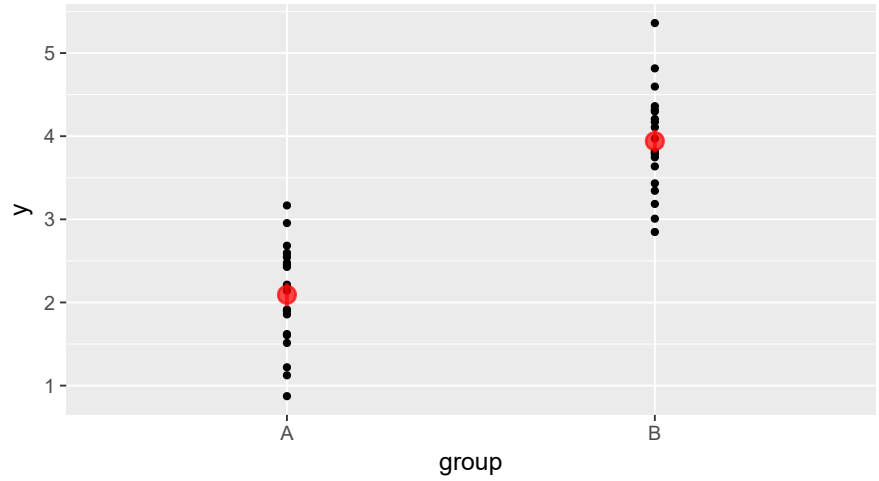
If needed, we can display less restrictive confidence intervals, at $p = 0.90$ in this example, by means of `conf.int = 0.90` passed as a list to the underlying function being called.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_cl_boot",
               fun.args = list(conf.int = 0.90),
               colour = "red", size = 1, alpha = 0.7)
```



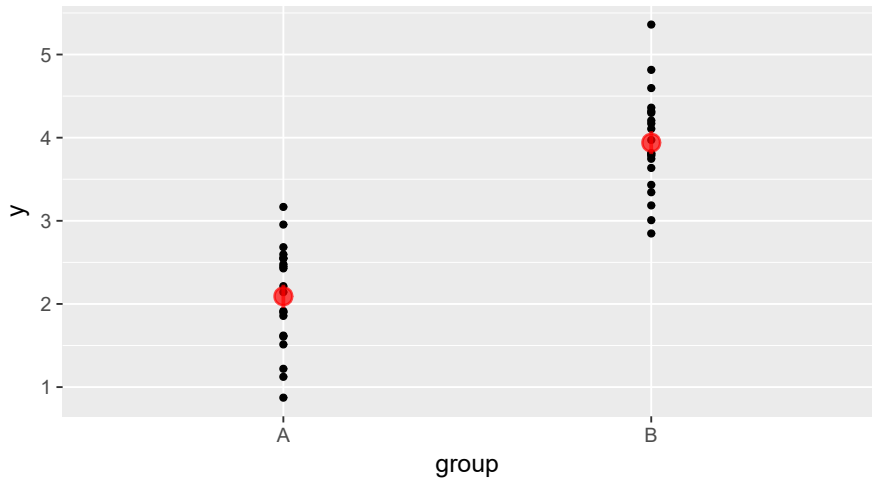
We can plot error bars corresponding to ±s.e. (standard errors) with the function `"mean_se"`, added in 'ggplot2'2.0.0.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_se",
               colour="red", size=1, alpha=0.7)
```



As `mult` is the multiplier based on the probability distribution used, by default student's t, by setting it to one, we get also standard errors of the mean.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_cl_normal",
               fun.args = list(mult = 1),
               colour="red", size=1, alpha=0.7)
```

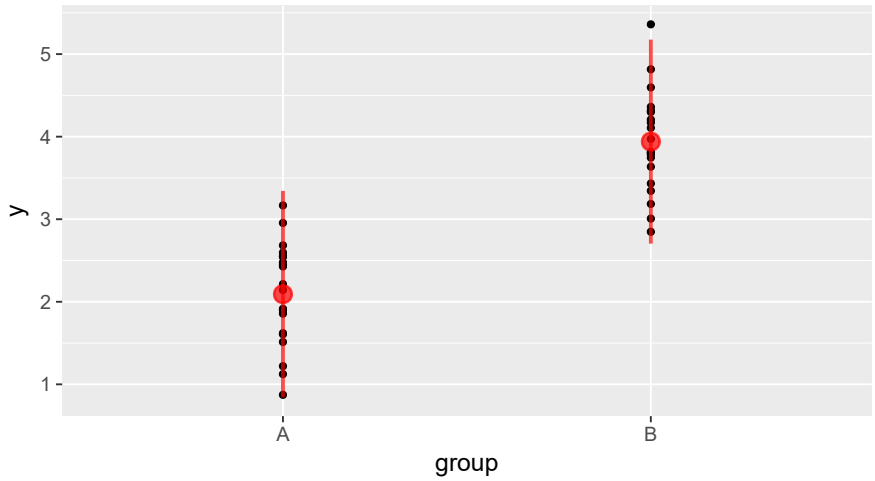


However, be aware that the code such as below (NOT EVALUATED HERE), as used in earlier versions of 'ggplot2', needs to be rewritten as above.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_cl_normal", mult = 1,
               colour="red", size=1, alpha=0.7)
```

Finally we can plot error bars showing $\pm$s.d. (standard deviation).

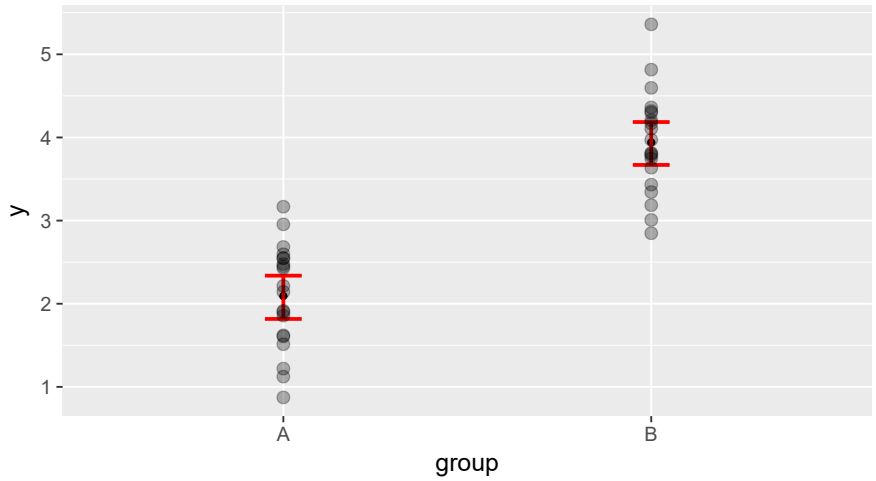
```
ggplot(data=fake.data, aes(y=y, x=group)) +
  geom_point() +
  stat_summary(fun.data = "mean_sdl", colour="red", size=1, alpha=0.7)
```



We do not give an example here, but instead of using these functions (from package 'Hmisc') it is possible to define one's own functions. In addition as arguments to any function used, except for the first one containing the actual data, are supplied as a list through formal argument `fun.args`, there is a lot of flexibility with respect to what functions can be used.

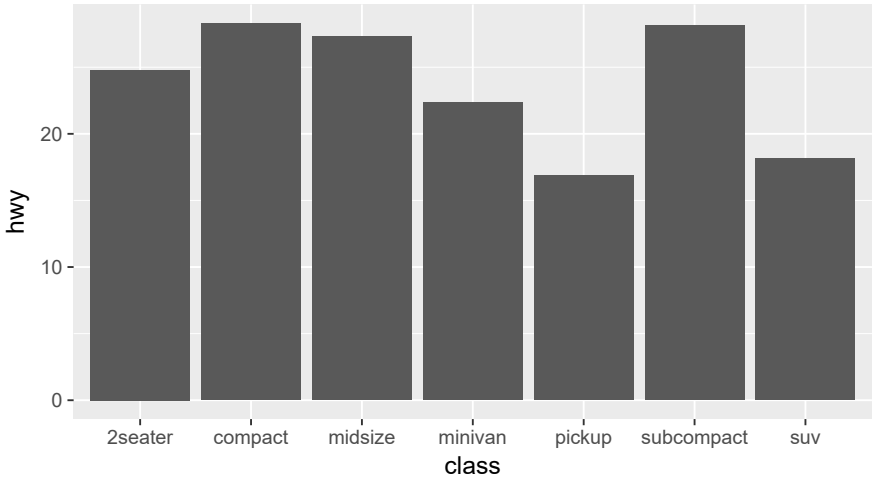
Finally we plot the means in a scatter plot, with the observations superimposed and $p = 0.95$ confidence interval (the order in which the geoms are added is important: by having `geom_point` last it is plotted on top of the bars. In this case we set fill, colour and alpha (transparency) to constants, but in more complex data sets mapping them to factors in the data set can be used to distinguish them. Adding `stat_summary` twice allows us to plot the mean and the error bars using different colors.

```
ggplot(data=fake.data, aes(y=y, x=group)) +
  stat_summary(fun.y = "mean", geom = "point",
               fill="white", colour="black") +
  stat_summary(fun.data = "mean_cl_boot",
               geom = "errorbar",
               width=0.1, size=1, colour="red") +
  geom_point(size=3, alpha=0.3)
```

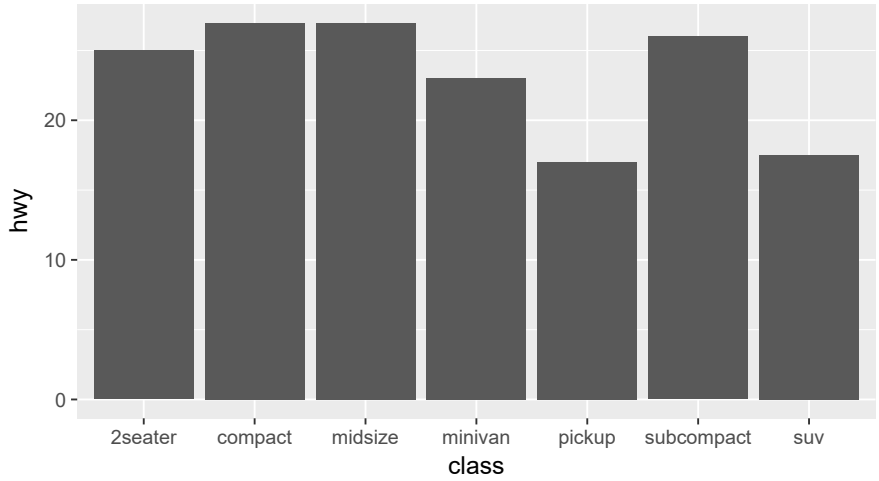


Similarly as with scatter plots, we can plot summaries as bars plots and add error bars. If we supply a different argument to `stat` we can for example plot the means or medians for a variable, for each `class` of car.

```
ggplot(mpg, aes(class, hwy)) + geom_bar(stat = "summary", fun.y = mean)
```

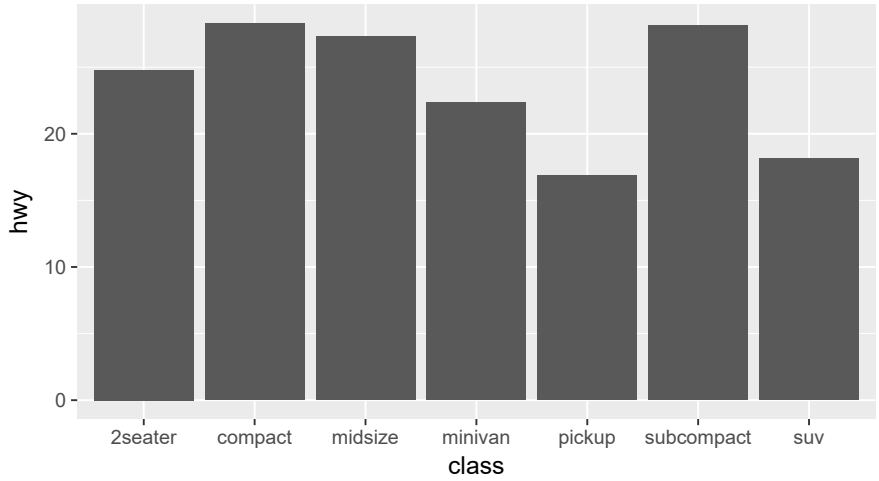



```
ggplot(mpg, aes(class, hwy)) + geom_bar(stat = "summary", fun.y = median)
```



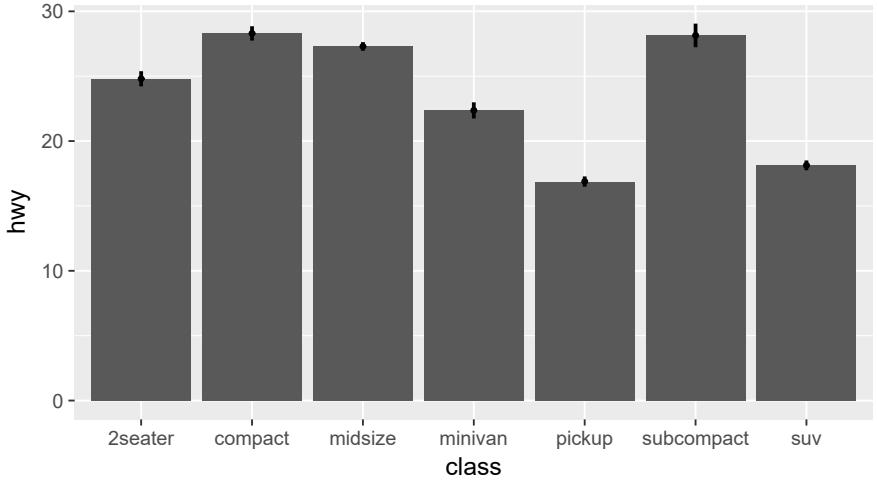
The "reverse" syntax is also possible, we can add the *statistics* to the plot object and pass the *geometry* as an argument to it.

```
ggplot(mpg, aes(class, hwy)) + stat_summary(geom = "col", fun.y = mean)
```



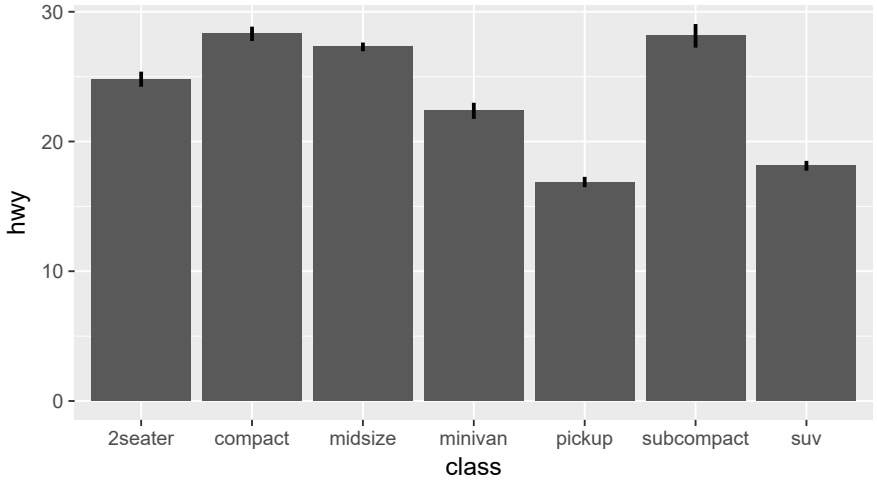
And we can easily add error bars to the bar plot. We use `size` to make the lines of the error bar thicker, and a value smaller than zero for `fatten` to make the point smaller. The default `geom` for `stat_summary` is `"pointrange"`.

```
ggplot(mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun.y = mean) +
  stat_summary(fun.data = "mean_se", size = 1, fatten = 0.5)
```



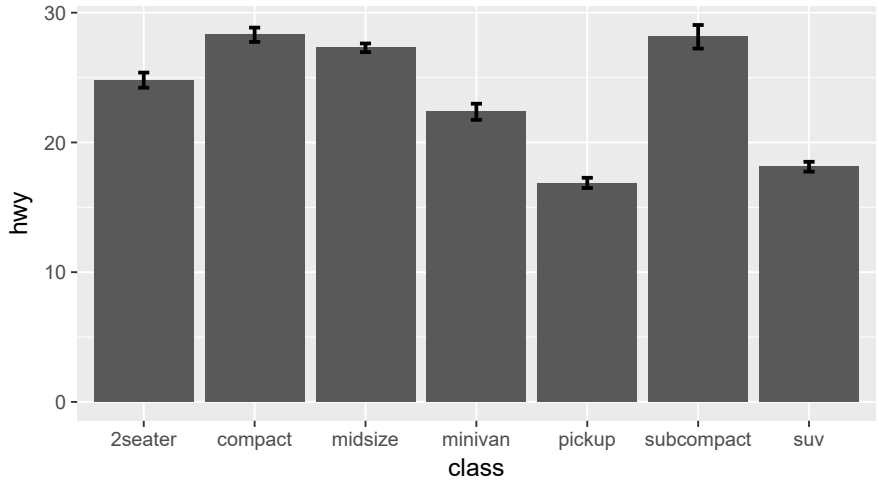
Instead of making the point smaller, we can pass `"linerange"` as argument to eliminate the point completely.

```
ggplot(mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun.y = mean) +
  stat_summary(geom = "linerange", fun.data = "mean_se", size = 1)
```



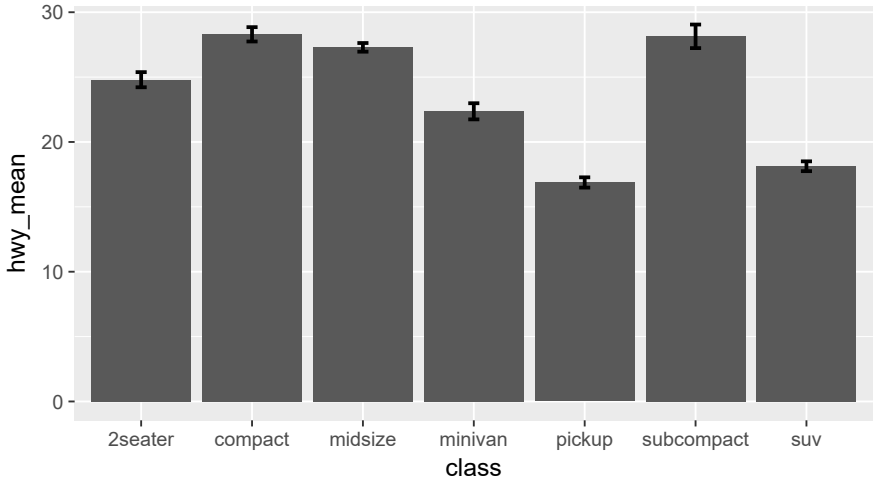
Passing `"errorbar"` to `geom` results in more traditional error bars, however, this type of error bars has been criticized as adding unnecesary clutter to plots (**TufteXXXX**). We use `width` to reduce the width of the cross lines at the ends of the bars.

```
ggplot(mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun.y = mean) +
 stat_summary(geom = "errorbar", fun.data = "mean_se", width = 0.1, size = 1)
```



If we have ready calculated values for the summaries, we can still obtain the same plots. Here we calculate the summaries before plotting, and then redraw the plot immediately above.

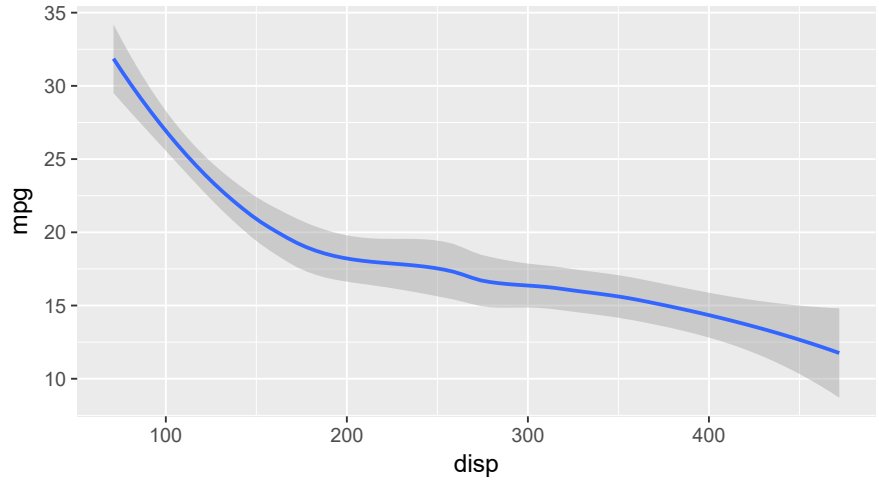
```
mpg_g <- dplyr::group_by(mpg, class)
mpg_summ <- dplyr::summarise(mpg_g, hwy_mean = mean(hwy),
                                    hwy_se = sd(hwy) / sqrt(n()))

ggplot(mpg_summ, aes(x = class,
                     y = hwy_mean,
                     ymax = hwy_mean + hwy_se,
                     ymin = hwy_mean - hwy_se)) +
  geom_col() +
  geom_errorbar(width = 0.1, size = 1)
```



6.13 Fitted smooth curves

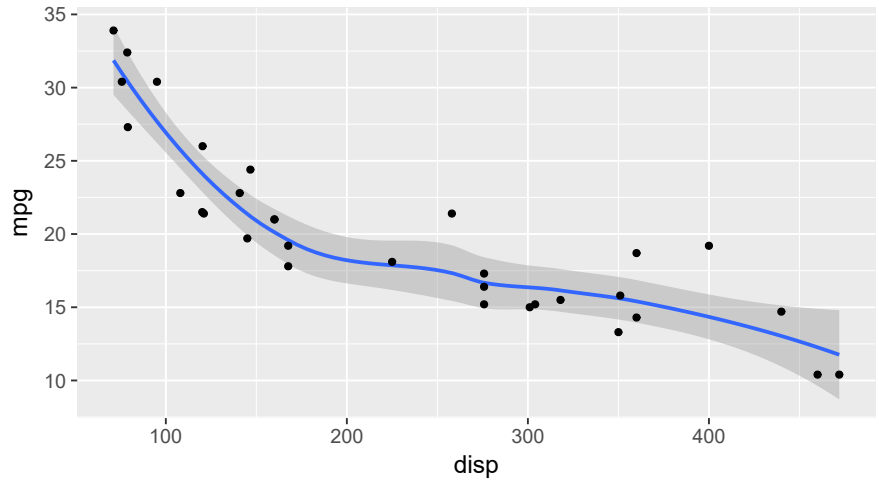
The *statistic* `stat_smooth` fits a smooth curve to observations in the case when the scales for x and y are continuous. For the first example, we use the default smoother, a spline. The type of spline is automatically chosen based on the number of observations.

```
ggplot(data = mtcars, aes(x=disp, y=mpg)) +
       stat_smooth()

## `geom_smooth()` using method = 'loess'
```



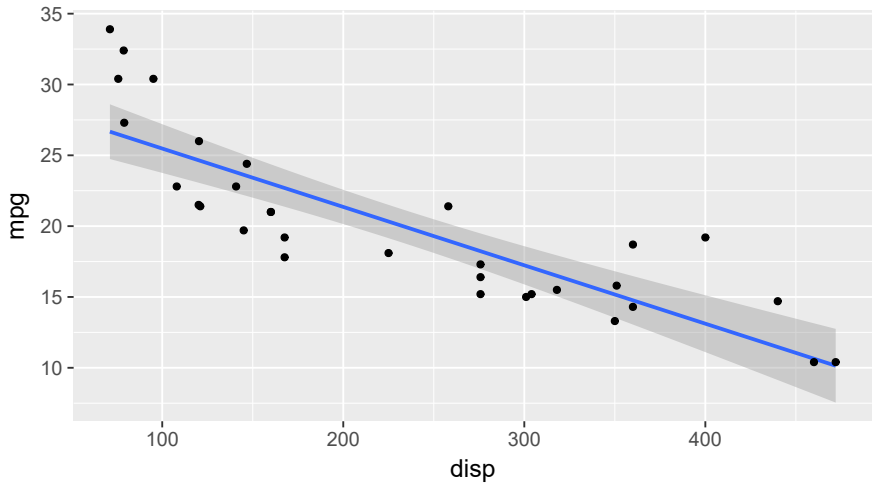
In most cases we will want to plot the observations as points together with the smoother. Can can plot the observation on top of the smoother, as done here, or the smoother on top of the observations.

```
ggplot(data = mtcars, aes(x=disp, y=mpg)) +
  stat_smooth() +
  geom_point()

## `geom_smooth()` using method = 'loess'
```



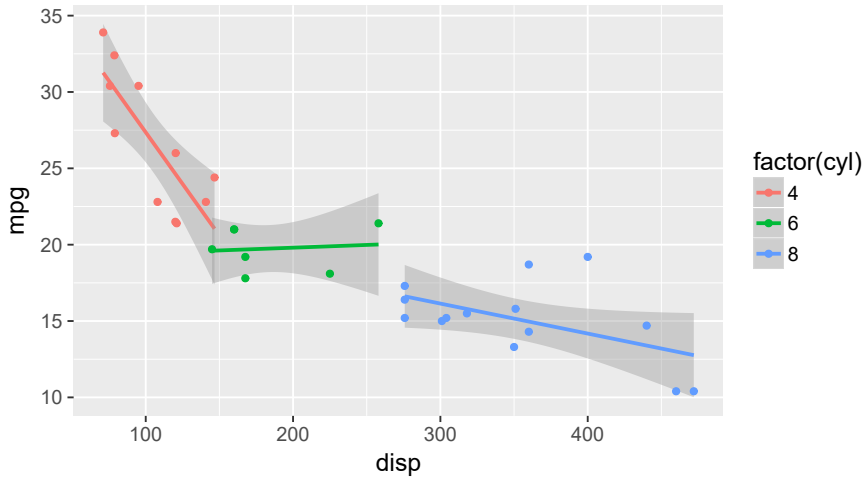
Instead of using the default spline, we can fit a different model. In this example we use a linear model as smoother, fitted by `lm`.

```
ggplot(data = mtcars, aes(x=disp, y=mpg)) +
  stat_smooth(method="lm") +
  geom_point()
```



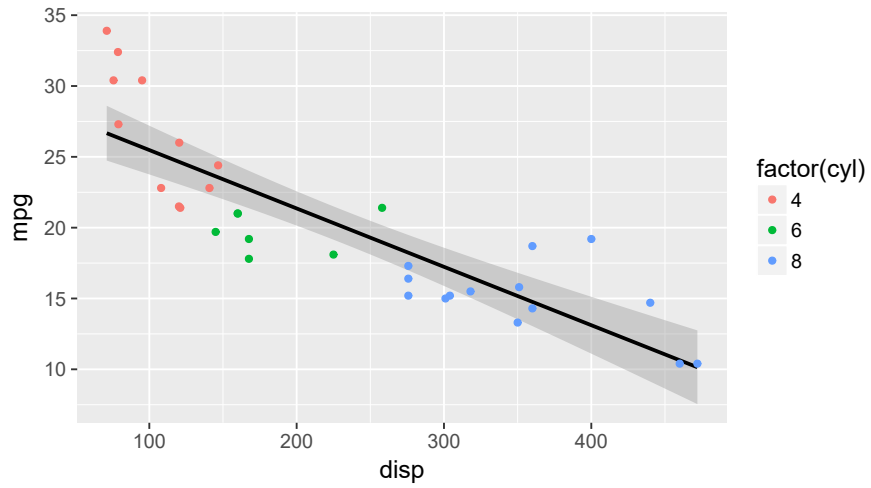
These data are really grouped, so we map the grouping to the `color` *aesthetic*. Now we get three groups of points with different colours but also three separate smooth lines.

```
ggplot(data = mtcars, aes(x=disp, y=mpg, color=factor(cyl))) +
  stat_smooth(method="lm") +
  geom_point()
```



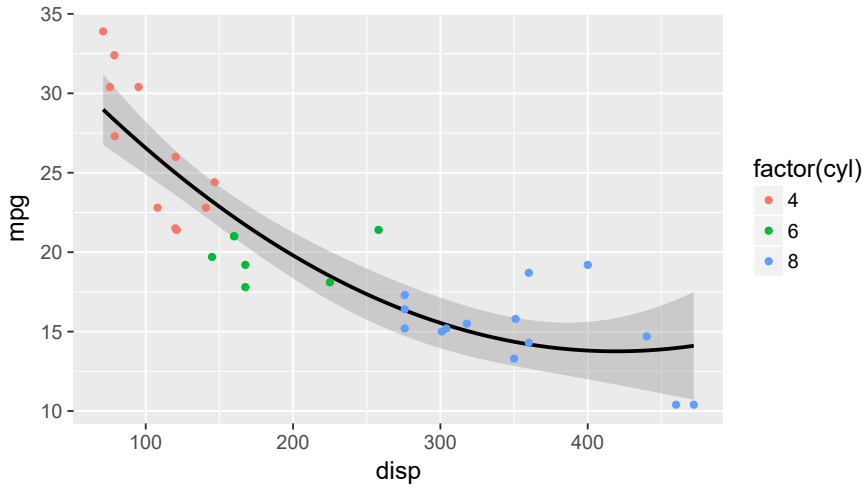
To obtain a single smoother for the three groups, we need to set the mapping of the `color` *aesthetic* to a constant within `stat_smooth`. This local value overrides the default for the whole plot set with `aes` just for this single *statistic*. We use `"black"` but this could be replaced by any other color definition known to R.

```
ggplot(data = mtcars, aes(x=disp, y=mpg, color=factor(cyl))) +
  stat_smooth(method="lm", colour="black") +
  geom_point()
```



Instead of using the default `formula` for a linear regression as smoother, we pass a different `formula` as argument. In this example we use a polynomial of order 2 fitted by `lm`.

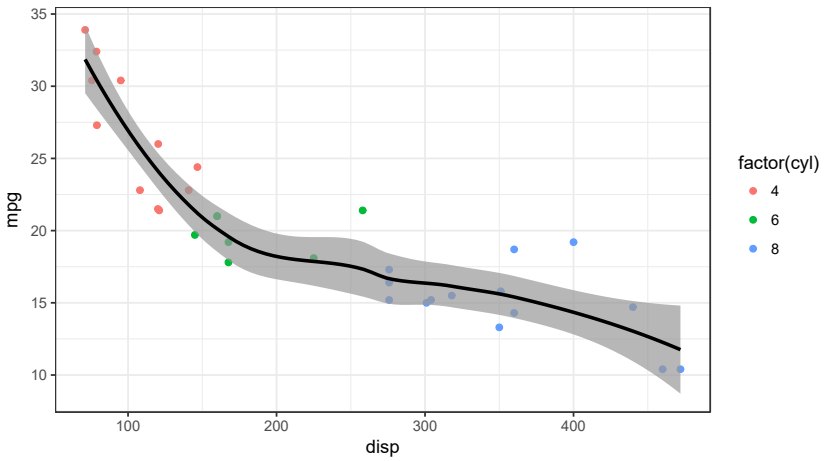
```
ggplot(data = mtcars, aes(x=disp, y=mpg, color=factor(cyl))) +
  stat_smooth(method="lm", formula=y~poly(x,2), colour="black") +
  geom_point()
```



It is possible to use other types of models, including GAM and GLM, as smoothers, but we will not give examples of the use of these more advanced models in this section.

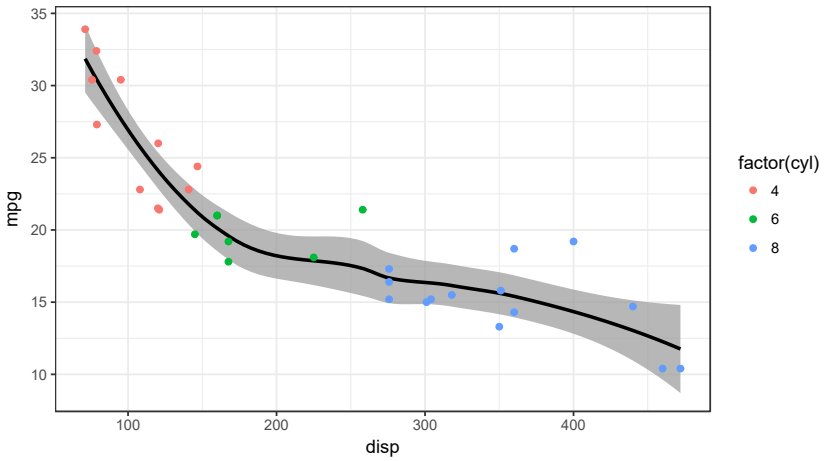
The different geoms and elements can be added in almost any order to a ggplot object, but they will be plotted in the order that they are added. The `alpha` (transparency) aesthetic can be mapped to a constant to make underlying layers visible, or `alpha` can be mapped to a data variable for example making the transparency of points in a plot depend on the number of observations used in its calculation.

```
ggplot(data = mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_point() +
  geom_smooth(colour="black", alpha=0.7) +
  theme_bw()

## `geom_smooth()` using method = 'loess'
```



The plot looks different if the order of the geoms is swapped. The data points overlapping the confidence band are more clearly visible in this second example because they are above the shaded area instead of bellow it.

```
ggplot(data = mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_smooth(colour="black", alpha=0.7) +
  geom_point() +
  theme_bw()

## `geom_smooth()` using method = 'loess'
```



6.14 Frequencies and densities

A different type of summaries are frequencies and empirical density functions. These can be calculated in one or more dimensions. Sometimes instead of being calculated, we rely on the density of graphical elements to convey the density. Sometimes, scatter plots using a well chosen value for `alpha` give a satisfactory impression of the density. Rug plots, described below work in a similar way.

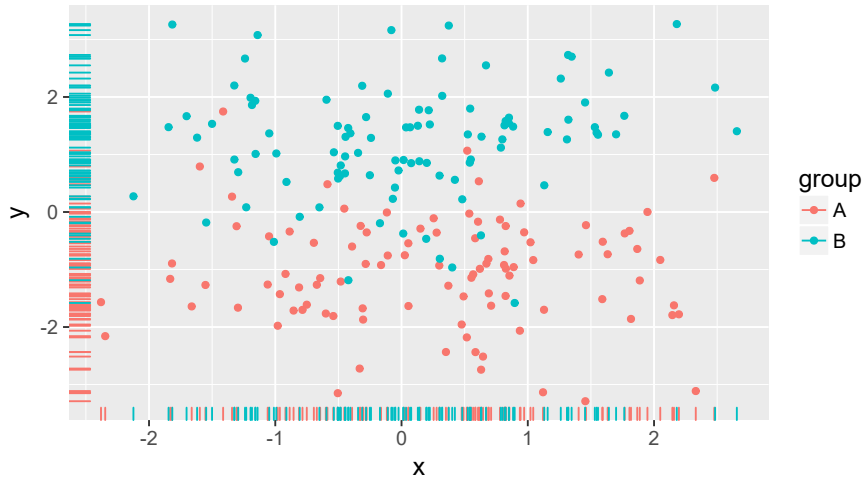
6.14.1 Marginal rug plots

Rarely rug-plots are used by themselves. Instead they are usually an addition to scatter plots. An example follows. They make it easier to see the distribution along the x- and y-axes.

We generate new fake data by random sampling from the normal distribution. We use `set.seed(1234)` to initialize the pseudo-random number generator so that the same data are generated each time the code is run.

```
set.seed(12345)
my.data <-
  data.frame(x = rnorm(200),
             y = c(rnorm(100, -1, 1), rnorm(100, 1, 1)),
             group = factor(rep(c("A", "B"), c(100, 100))) )
```

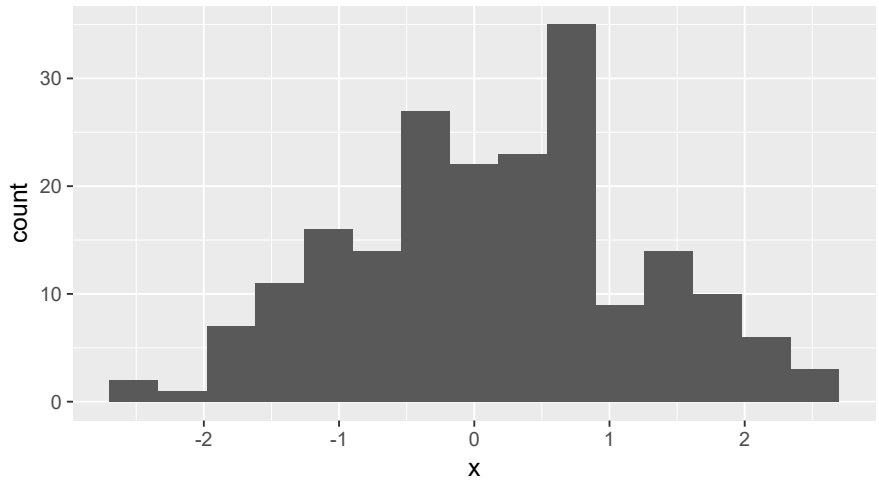
```
ggplot(my.data, aes(x, y, colour = group)) +
  geom_point() +
  geom_rug()
```



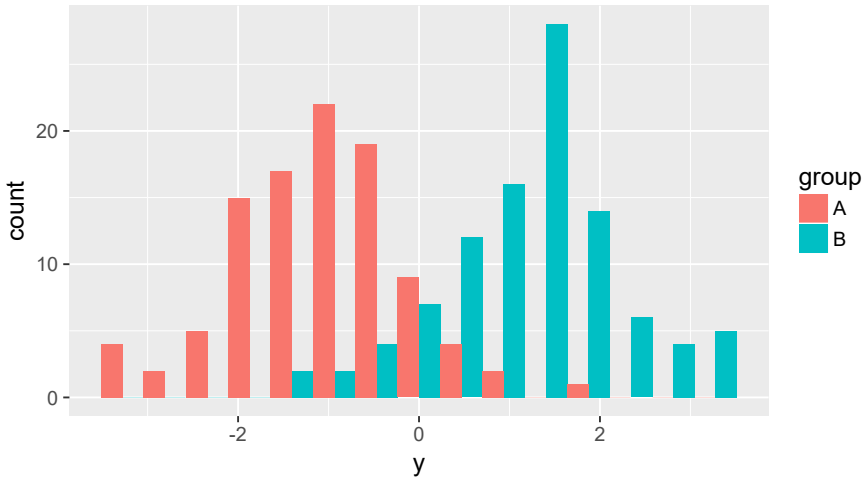
6.14.2 Histograms

Histograms are defined by how the plotted values are calculated. Although they are most frequently plotted as bar plots, many bar plots are not histograms. Although rarely done in practice, a histogram could be plotted using a different *geometry* and `stat_bin` the *statistic* used by default by `geom_histogram`. This statistics does binning of observations before computing frequencies, as is suitable for continuous x scales. For categorical data `stat_count` should be used, which as seen in section 6.10 on page 107 is the default `stat` for `geom_bar`.

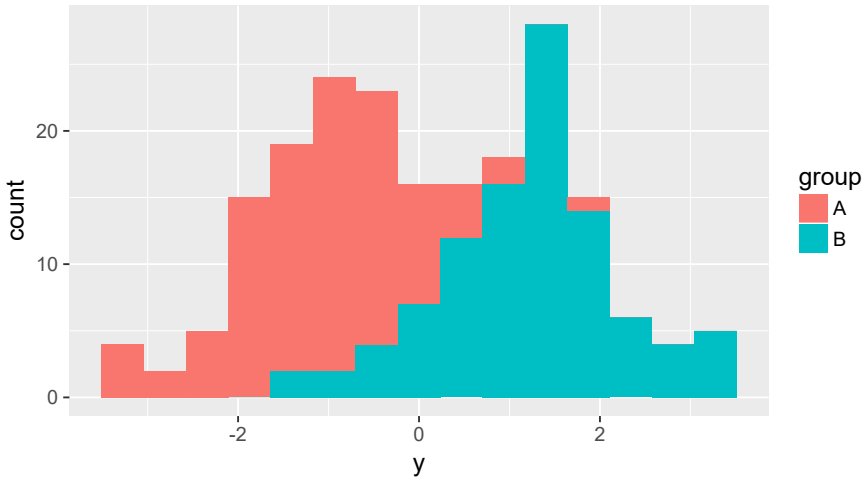
```
ggplot(my.data, aes(x)) +
  geom_histogram(bins = 15)
```



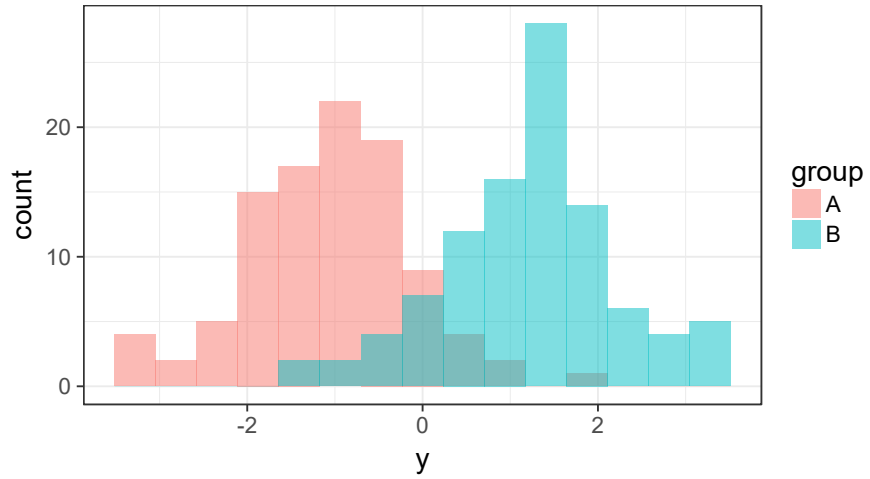

```
ggplot(my.data, aes(y, fill = group)) +
  geom_histogram(bins = 15, position = "dodge")
```




```
ggplot(my.data, aes(y, fill = group)) +
  geom_histogram(bins = 15, position = "stack")
```

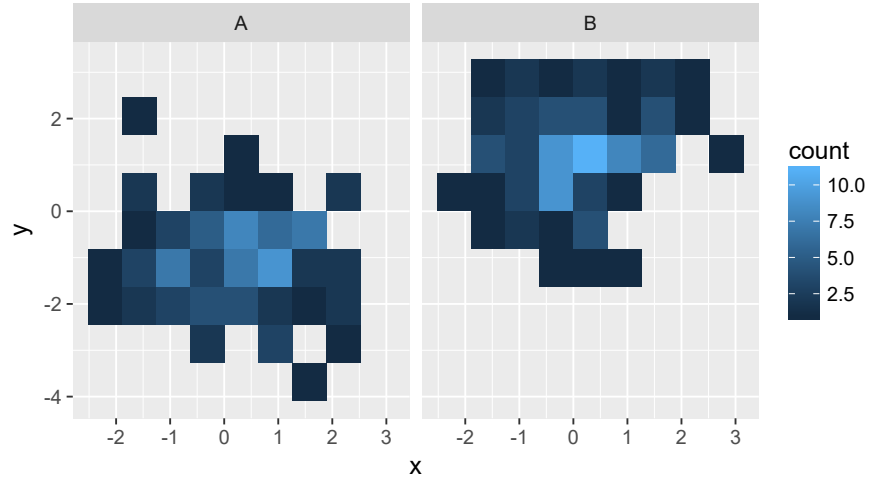



```
ggplot(my.data, aes(y, fill = group)) +
  geom_histogram(bins = 15, position = "identity", alpha = 0.5) +
  theme_bw(16)
```



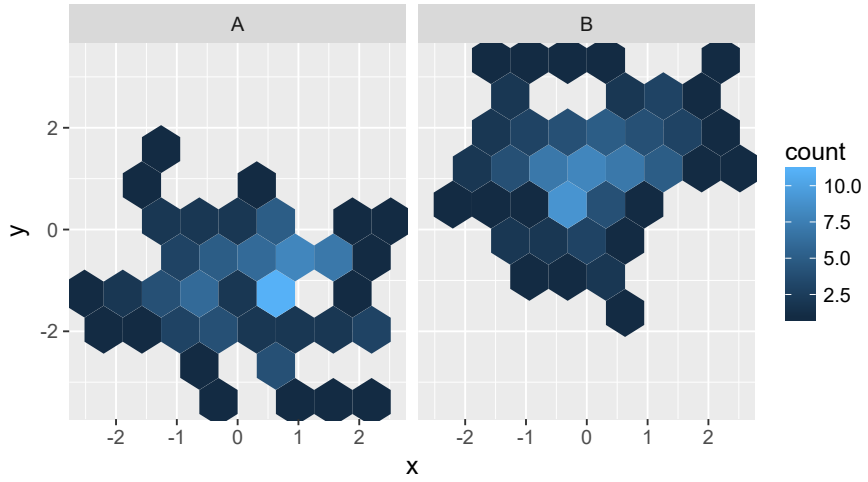
The *geometry* `geom_bin2d` by default uses the *statistic* `stat_bin2d` which can be thought as a histogram in two dimensions. The frequency for each rectangle is mapped onto a `fill` scale.

```
ggplot(my.data, aes(x, y)) +
  geom_bin2d(bins = 8) +
  facet_wrap(~group)
```



The *geometry* `geom_hex` by default uses the *statistic* `stat_binhex` which can be thought as a histogram in two dimensions. The frequency for each hexagon is mapped onto a `fill` scale.

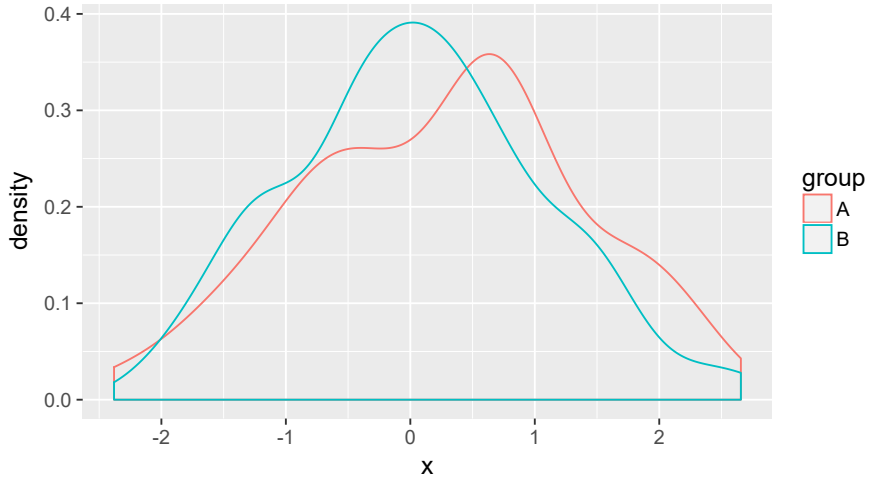
```
ggplot(my.data, aes(x, y)) +
  geom_hex(bins = 8) +
  facet_wrap(~group)

## Loading required package: methods
```



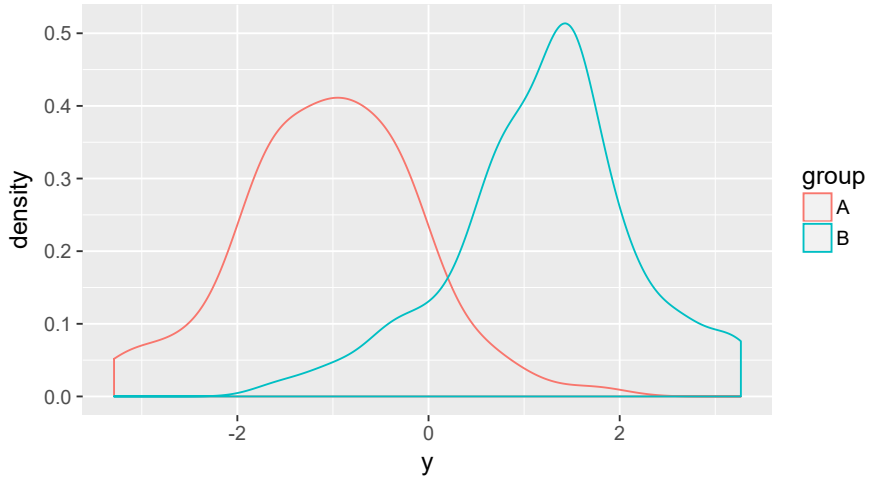
6.14.3 Density plots

Empirical density functions are the equivalent of a histogram, but are continuous and not calculated using bins. They can be calculated in 1 or 2 dimensions (2d), for x or x and y respectively. As with histograms it is possible to use different *geometries* to visualize them.

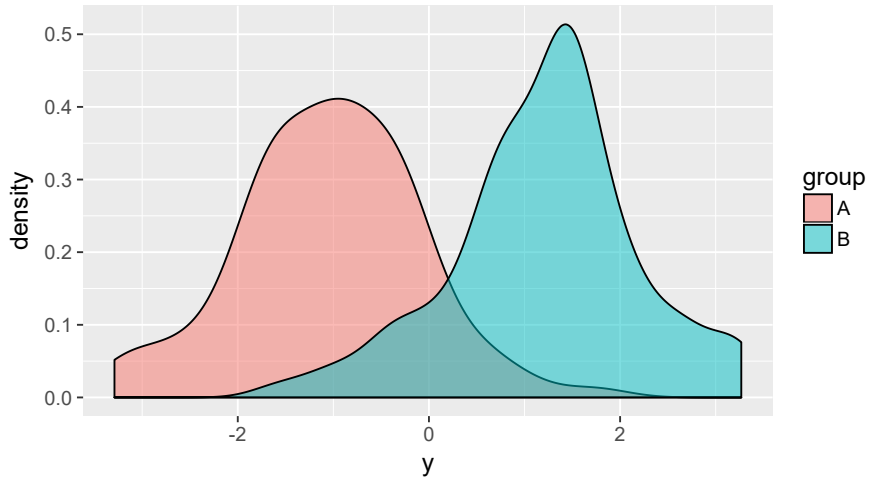
```
ggplot(my.data, aes(x, colour = group)) +
  geom_density()
```



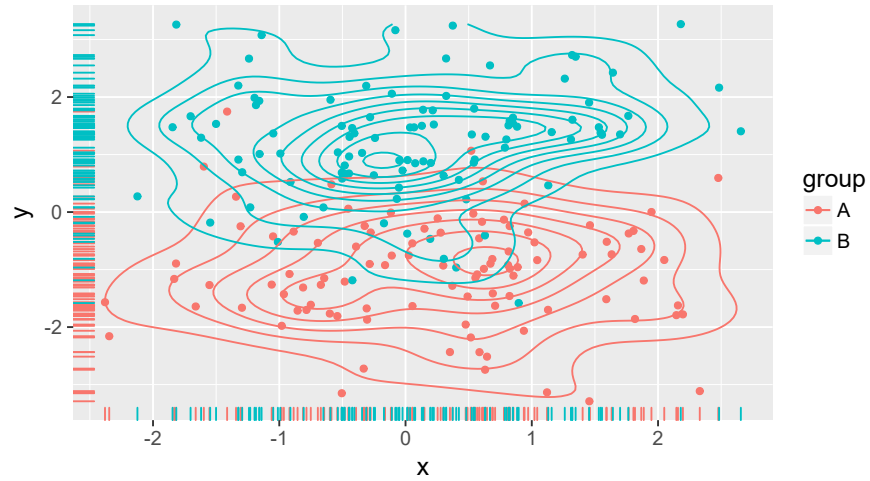

```
ggplot(my.data, aes(y, colour = group)) +
  geom_density()
```



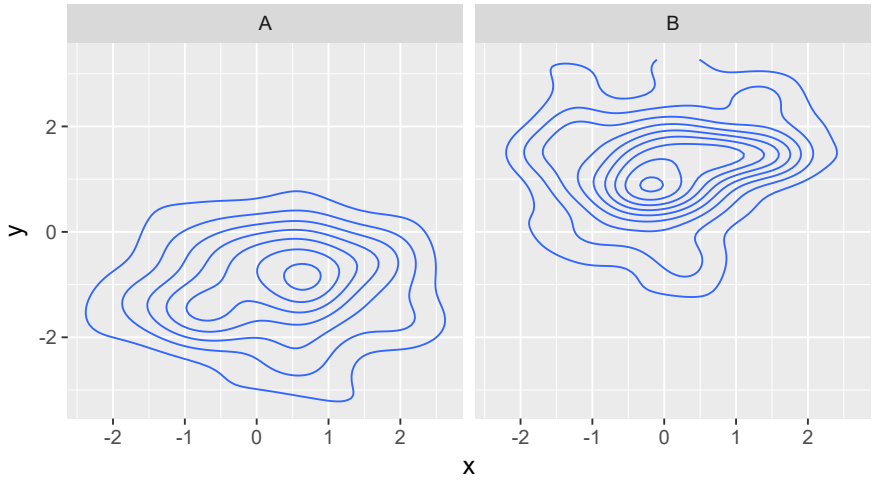

```
ggplot(my.data, aes(y, fill = group)) +
  geom_density(alpha = 0.5)
```



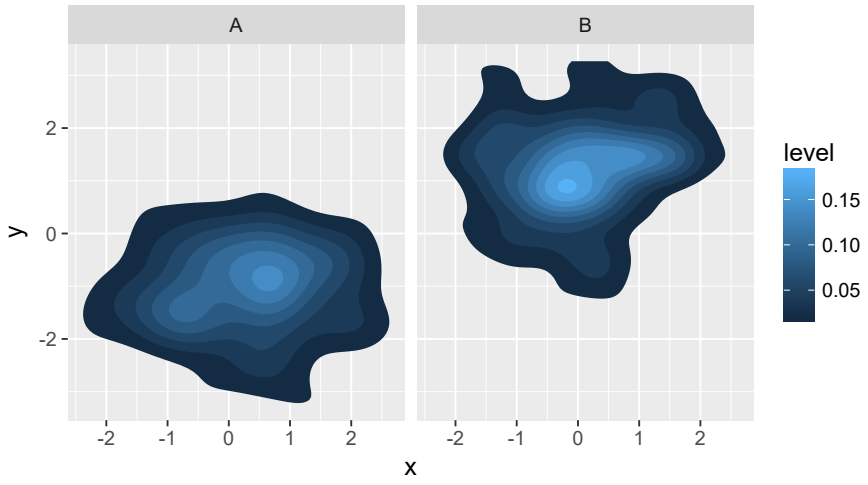

```
ggplot(my.data, aes(x, y, colour = group)) +
  geom_point() +
  geom_rug() +
  geom_density_2d()
```




```
ggplot(my.data, aes(x, y)) +
  geom_density_2d() +
  facet_wrap(~group)
```




```
ggplot(my.data, aes(x, y)) +
stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group)
```

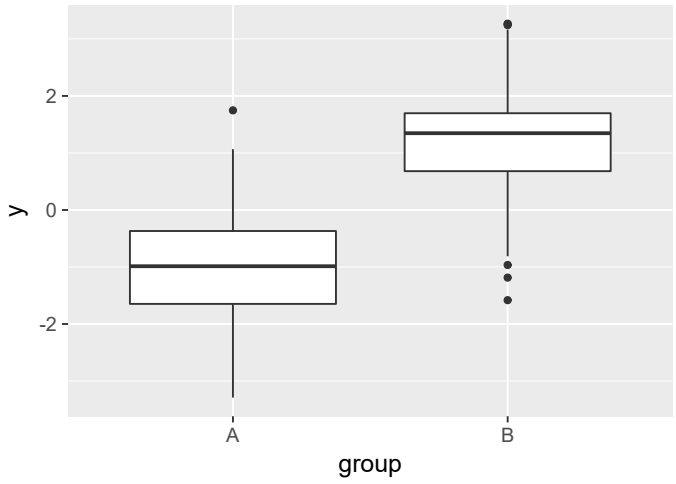


6.14.4 Box and whiskers plots

Box and whiskers plots, also very frequently called just boxplots, are also summaries that convey some of the characteristics of a distribution. Although they can be calculated and plotted based on just a few observa-

tions, they are not useful unless each box plot is based in more than 10 to 15 observations.

```
ggplot(my.data, aes(group, y)) +
  geom_boxplot()
```

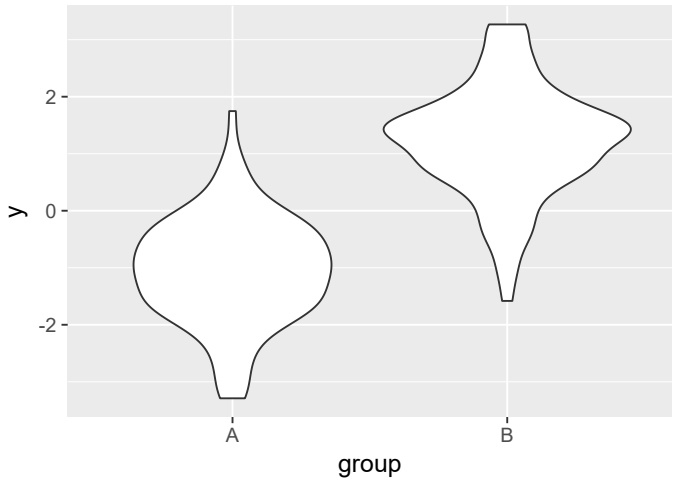


As with other *geometries* their appearance obeys both the usual *aesthetics* such as color, and others specific to these type of visual representation.

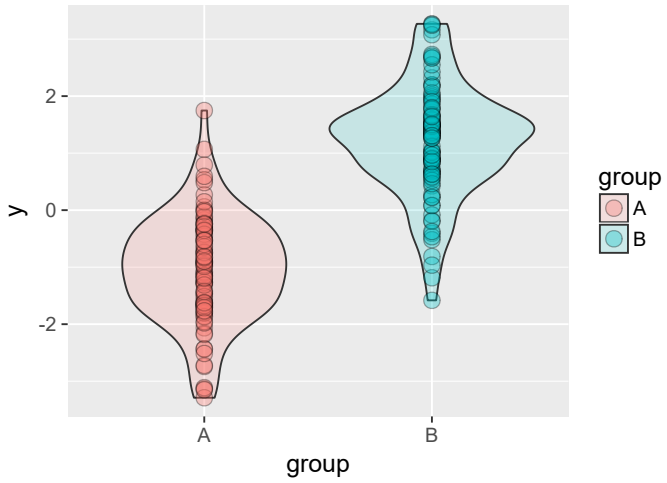
6.14.5 Violin plots

Violin plots are a more recent development than box plots, and usable with relatively large numbers of observations. They could be thought as being a sort of hybrid between an empirical density function and a box plot. As is the case with box plots, they are particularly useful when comparing distributions of related data, side by side.

```
ggplot(my.data, aes(group, y)) +
  geom_violin()
```




```
ggplot(my.data, aes(group, y, fill = group)) +
  geom_violin(alpha = 0.16) +
  geom_point(alpha = 0.33, size = rel(4),
             colour = "black", shape = 21)
```



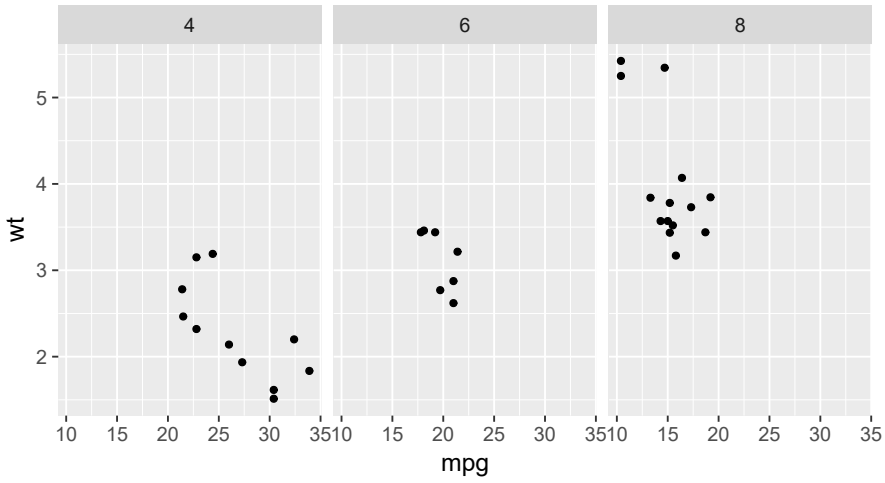
As with other *geometries* their appearance obeys both the usual *aesthetics* such as color, and others specific to these type of visual representation.

6.15 Using facets

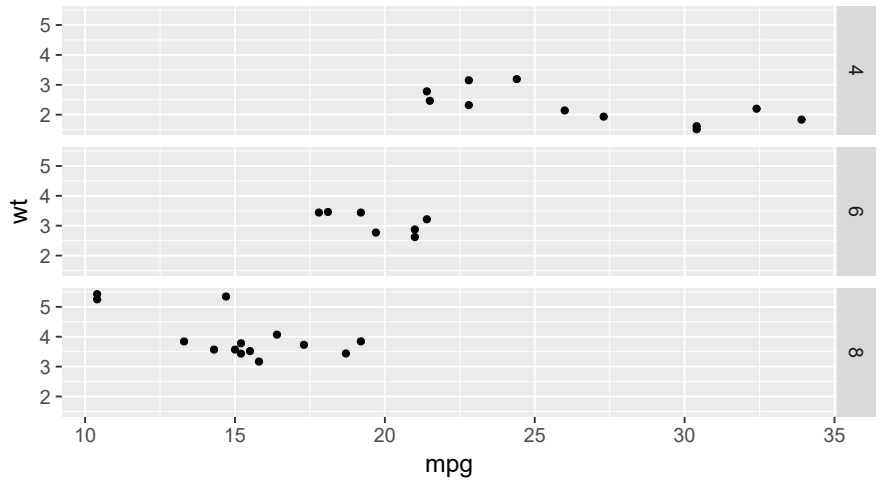
Sets of coordinated plots are a very useful tool for visualizing data. These became popular through the `trellis` graphs in S, and the `lattice` package in R. The basic idea is to have row and/or columns of plots with common scales, all plots showing values for the same response variable. This is useful when there are multiple classification factors in a data set. Similarly looking plots but with free scales or with the same scale but a 'floating' intercept are sometimes also useful. In 'ggplot2'there are two possible types of facets: facets organized in a grid, and facets along a single 'axis' but wrapped into several rows. In the examples below we use `geom_point` but faceting can be used with any `ggplot` object (even with maps, spectra and ternary plots produced by functions in packages 'ggmap', 'ggspectra'and 'ggtern'.

> The code underlying faceting has been rewritten in 'ggplot2'version 2.2.0. All the examples given here are backwards compatible with versions 2.1.0 and possibly 2.0.0. The new functionality is related to the writing of extensions or controlled through themes, and will be discussed in other sections.

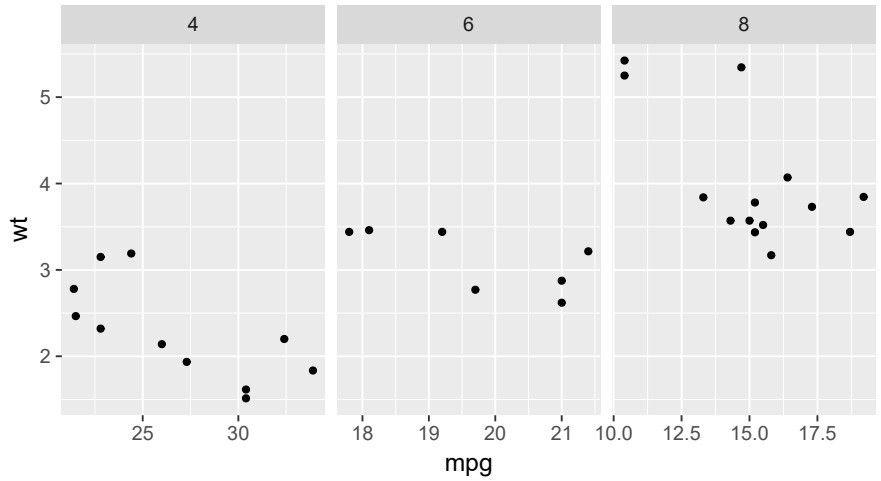
```
p <- ggplot(data = mtcars, aes(mpg, wt)) + geom_point()
# With one variable
p + facet_grid(. ~ cyl)
```



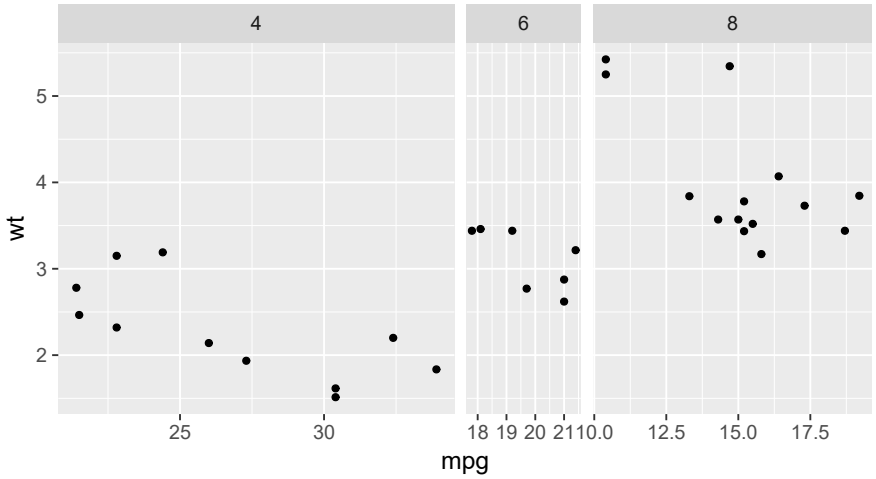
```
p + facet_grid(cyl ~ .)
```



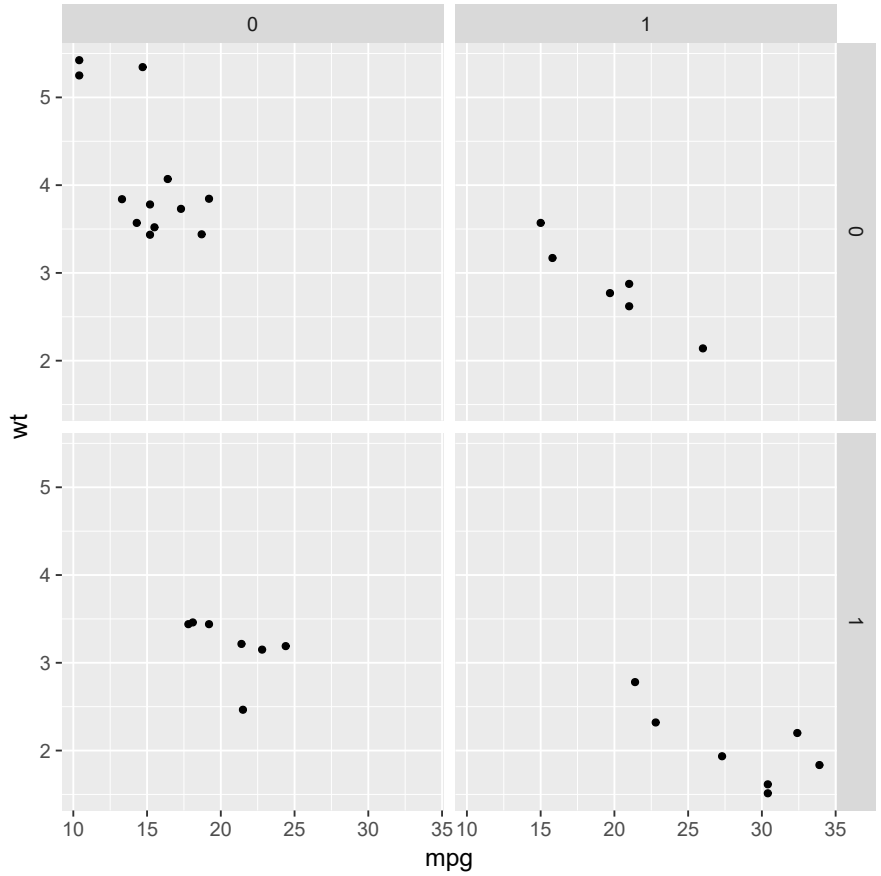

```
p + facet_grid(. ~ cyl, scales = "free")
```



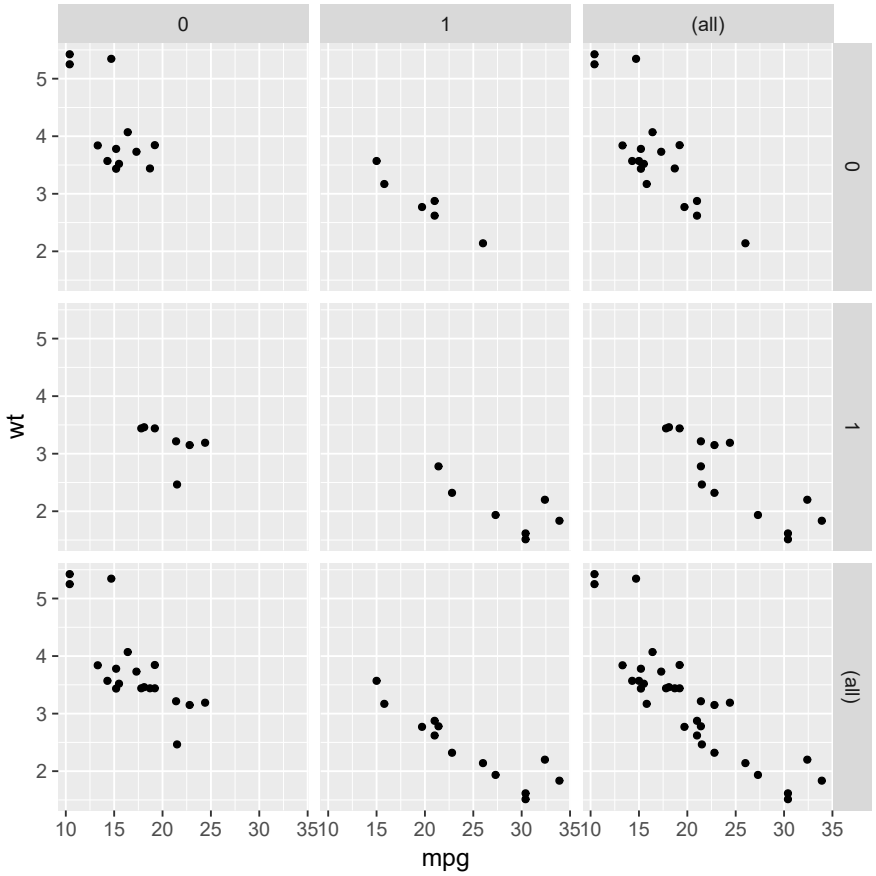

```
p + facet_grid(. ~ cyl, scales = "free", space = "free")
```



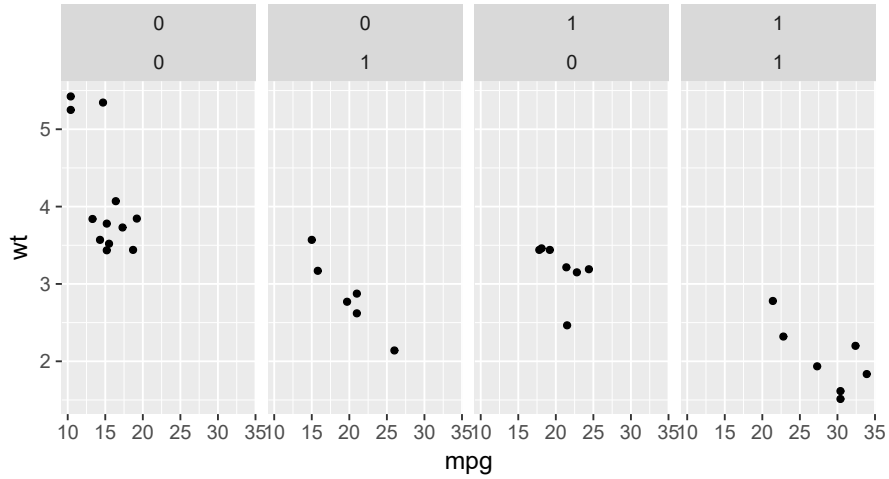

```
p + facet_grid(vs ~ am)
```



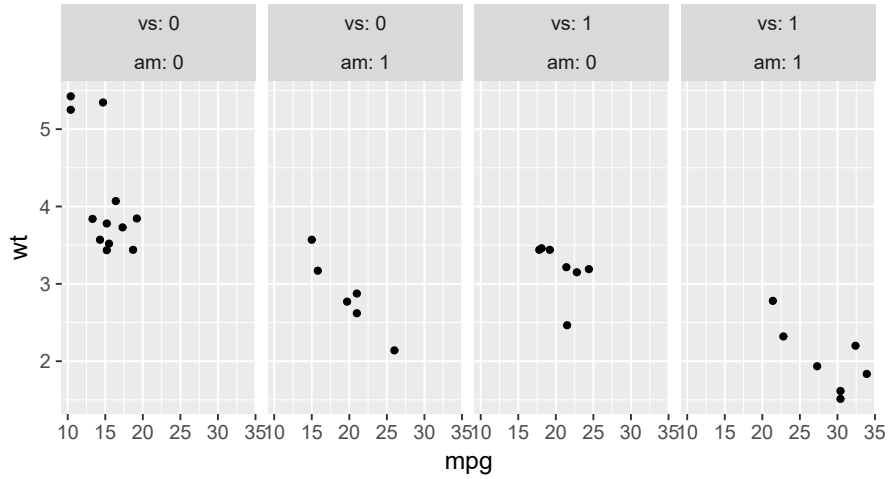

```
p + facet_grid(vs ~ am, margins=TRUE)
```



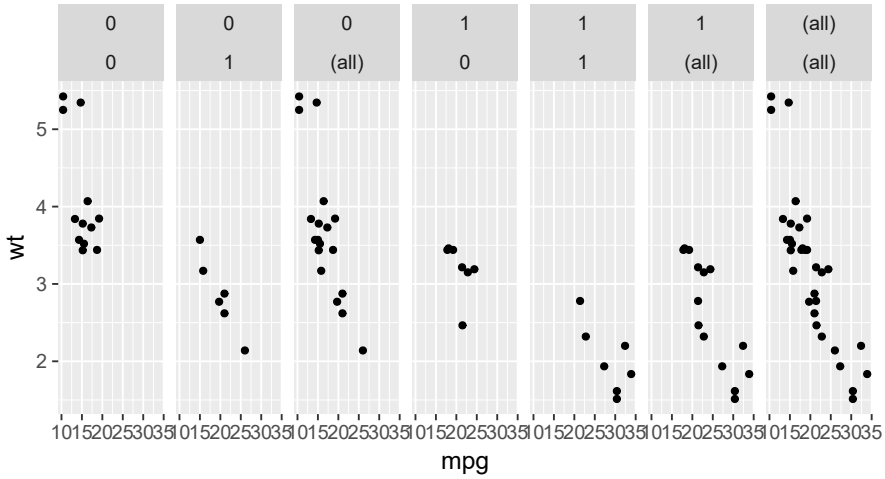

```
p + facet_grid(. ~ vs + am)
```



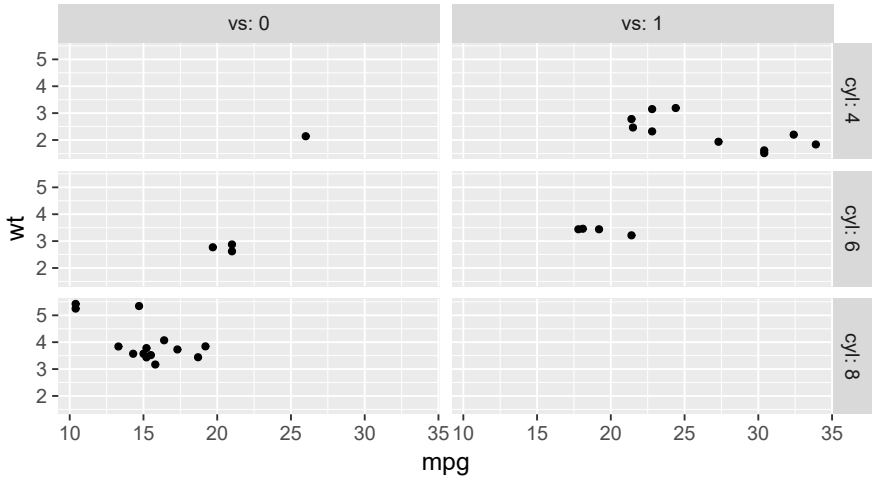

```
p + facet_grid(. ~ vs + am, labeller = label_both)
```




```
p + facet_grid(. ~ vs + am, margins=TRUE)
```



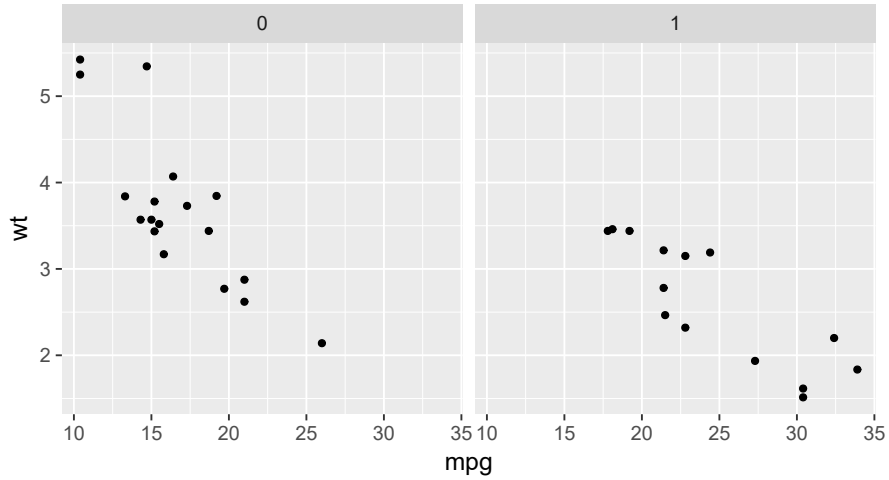

```
p + facet_grid(cyl ~ vs, labeller = label_both)
```




```
mtcars$cyl12 <- factor(mtcars$cyl,
                       labels = c("alpha", "beta", "sqrt(x, y)"))
p1 <- ggplot(data = mtcars, aes(mpg, wt)) +
      geom_point() +
      facet_grid(. ~ cyl12, labeller = label_parsed)
```

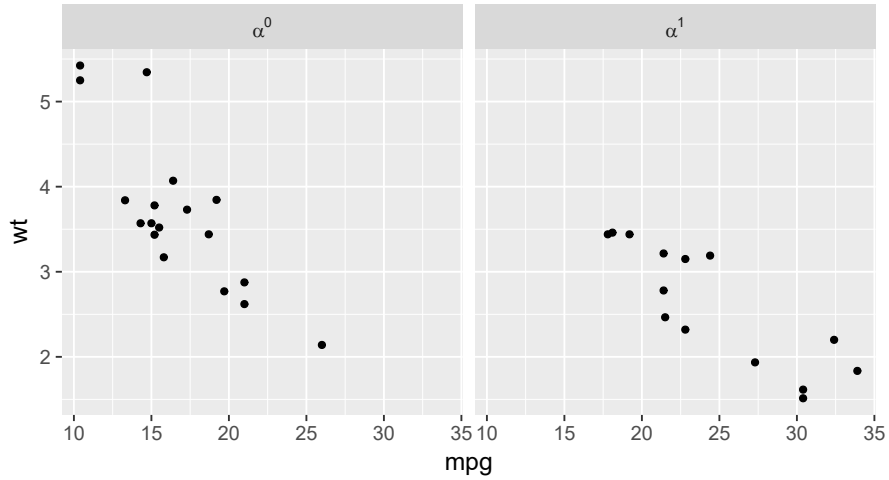
Here we use as `labeller` function `label_bquote()` with a special syntax that allows us to use an expression where replacement based on the facet (panel) data takes place.

```
p + facet_grid(. ~ vs, labeller = label_bquote(alpha ^ .(vs)))
```



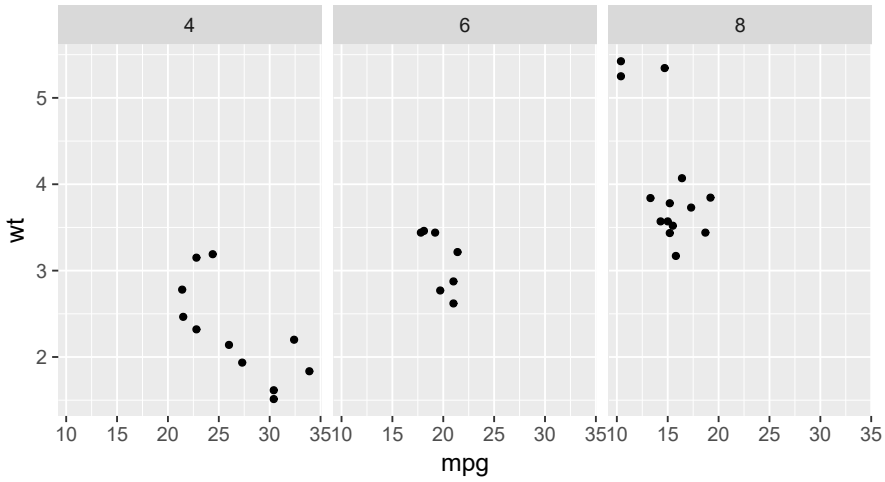
In versions of 'ggplot2'2 before 2.0.0, `labeller` was not implemented for `facet_wrap()`, it was only available for `facet_grid()`.

```
p + facet_wrap(~ vs, labeller = label_bquote(alpha ^ .(vs)))
```



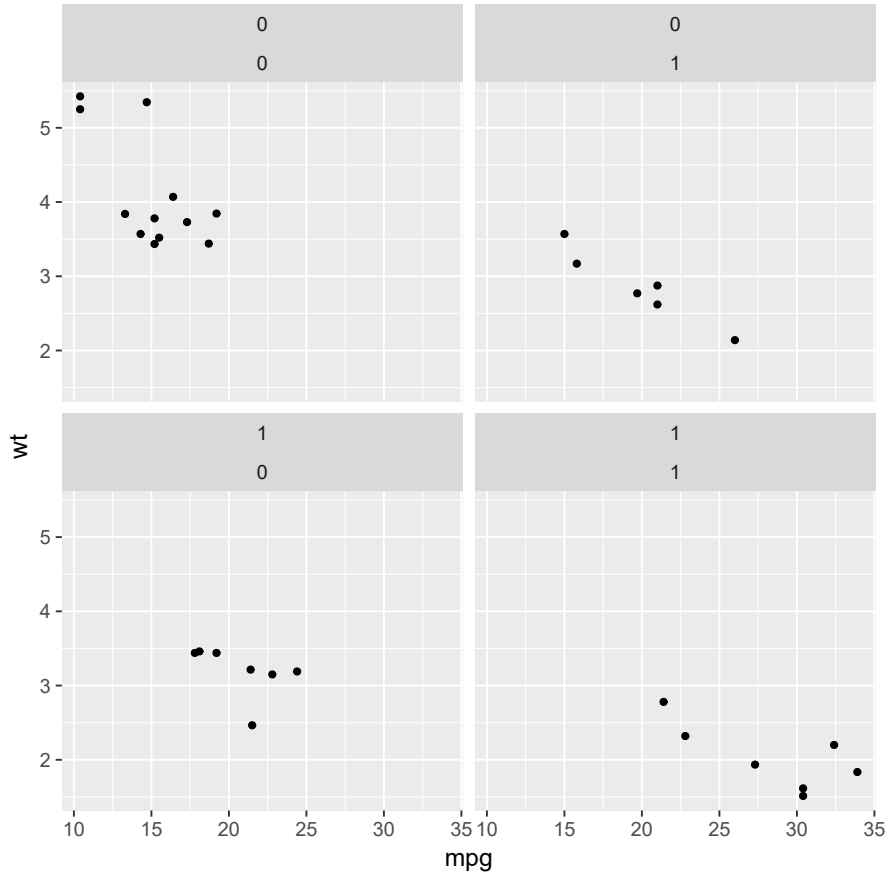
A minimal example of a wrapped facet. In this case the number of levels is small, when they are more the row of plots will be wrapped into two or more continuation rows. When using `facet_wrap()` there is only one dimension, so no '.' is needed before or after the tilde.

```
p + facet_wrap(~ cyl)
```



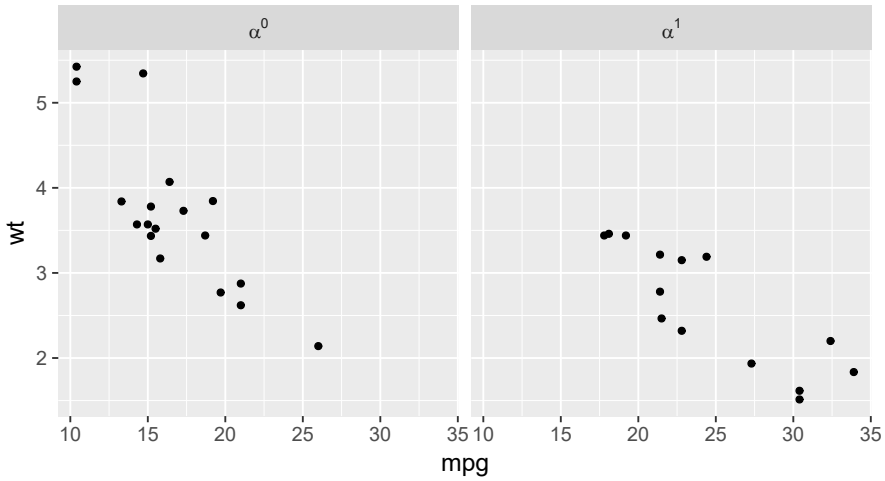
An example showing that even though faceting with `facet_wrap()` is along a single, possibly wrapped, row, it is possible to produce facets based on more than one variable.

```
p + facet_wrap(~ vs + am, ncol=2)
```



In versions of 'ggplot2'before 2.0.0, `labeller` was not implemented for `facet_wrap()`, it was only available for `facet_grid()`. In the current version it is implemented for both.

```
p + facet_wrap(~ vs, labeller = label_bquote(alpha ^ .(vs)))
```



6.16 Scales

Scales map data onto aesthetics. There are different types of scales depending on the characteristics of the data being mapped: scales can be continuous or discrete. And of course, there are scales for different attributes of the plotted geometrical object, such as `color`, `size`, position (`x, y, z`), `alpha` or transparency, `angle`, justification, etc. This means that many properties of, for example, the symbols used in a plot can be either set by a constant, or mapped to data. The most elemental mapping is `identity`, which means that the data is taken at its face value. In a numerical scale, say `scale_x_continuous`, this means that for example a '5' in the data is plotted at a position in the plot corresponding to the value '5' along the x-axis. A simple mapping could be a log10 transformation, that we can easily achieve with the pre-defined `scale_x_log10` in which case the position on the x-axis will be based on the logarithm of the original data. A continuous data variable can, if we think it useful for describing our data, be mapped to continuous scale either using an identity mapping or transformation, which for example could be useful if we want to map the value of a variable to the area of the symbol rather than its diameter.

Discrete scales work in a similar way. We can use `scale_colour_identity` and have in our data a variable with values that are valid colour names like "red" or "blue". However we can also map the `colour` aesthetic to

a factor with levels like "control", and "treatment", an these levels will be mapped to colours from the default palette, unless we chose a different palette, or even use `scale_colour_manual` to assign whatever colour we want to each level to be mapped. The same is true for other discrete scales like symbol `shape` and `linetype`. Remeber that for example for colour, and 'numbers' there are both discrete and continuous scales available. Mapping colour or fill to `NA` makes such observation invisible.

Advanced scale manipulation requires package `scales` to be loaded, although 'ggplot2'2.0.0 and later re-exports many functions from package `scales`. Some simple examples follow.

We generate new fake data.

```
fake2.data <-
  data.frame(y = c(rnorm(20, mean=20, sd=5),
                   rnorm(20, mean=40, sd=10)),
             group = factor(c(rep("A", 20), rep("B", 20))),
             z = rnorm(40, mean=12, sd=6))
```

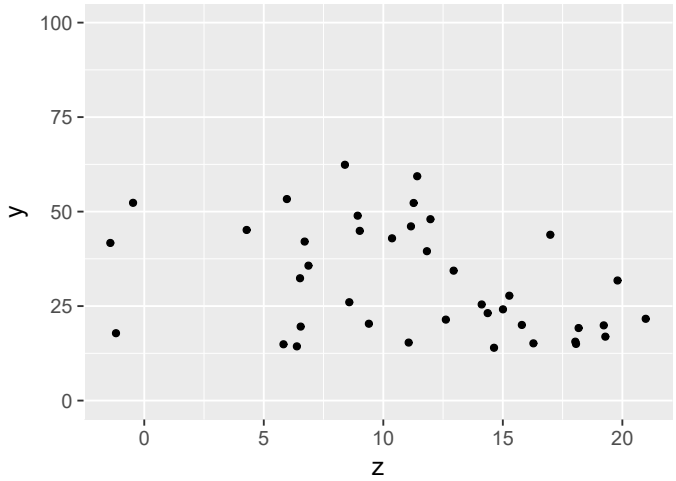
6.16.1 Continuous scales for x and y

Limits

To change the limits of the y-scale, `ylim()` is a convenience function used for modification of the `lims` (limits) of the scale used by the y aesthetic. We here exemplify the use of `ylim` only, but `xlim` can be used equivalently for the x scale.

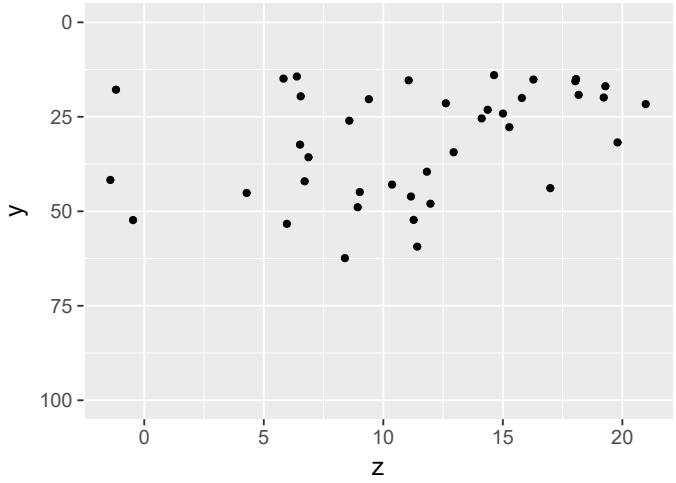
We can set both limits, minimum and maximum.

```
ggplot(fake2.data, aes(z, y)) + geom_point() + ylim(0, 100)
```



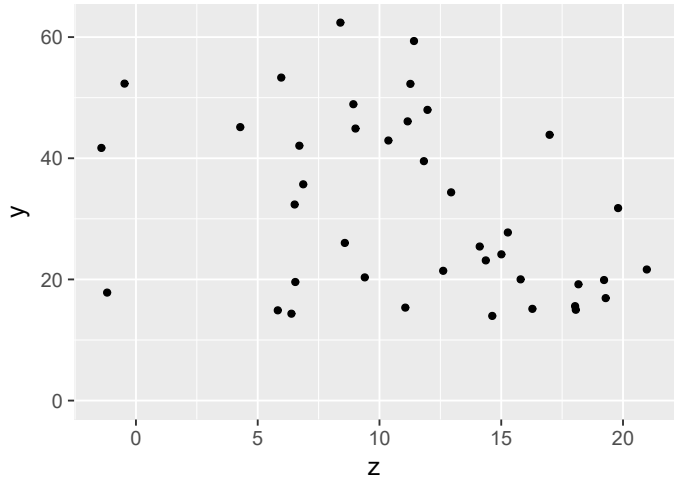
We can set both limits, minimum and maximum, reversing the direction of the axis scale.

```
ggplot(fake2.data, aes(z, y)) + geom_point() + ylim(100, 0)
```



We can set one limit and leave the other one free.

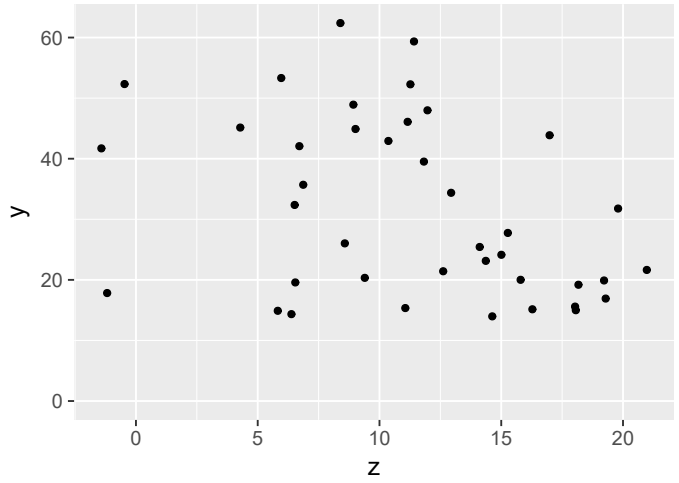
```
ggplot(fake2.data, aes(z, y)) + geom_point() + ylim(0, NA)
```



We can use `lims` with discrete scales, listing all the levels that are to be included in the scale, even if they are missing from a given data set, such as after subsetting.

And we can expand the limits, to set a default minimum range, that will grow when needed to accommodate all observations in the data set. Of course here `x` and `y` refer to the *aesthetics* and not to names of variables in data frame `fake2.data`.

```
ggplot(fake2.data, aes(z, y)) + geom_point() + expand_limits(y = 0, x = 0)
```



Transformed scales

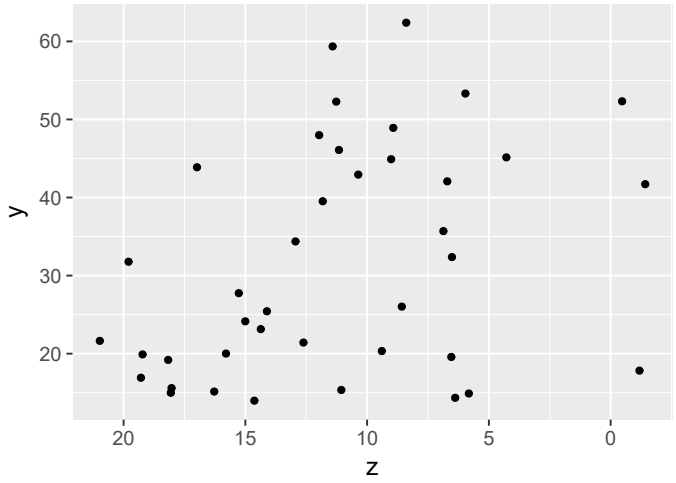
The default scale used by the `y` aesthetic uses `position = "identity"`, but there are predefined for transformed scales.

Although transformations can be passed as argument to `scale_x_continuous` and `scale_y_continuous`, there are predefined convenience scale functions for `log10`, `sqrt` and `reverse`.

> Similarly to the maths functions of R, the name of the scales are `scale_x_log10` and `scale_y_log10` rather than `scale_y_log` because in R the function `log` returns the natural or Neperian logarithm.

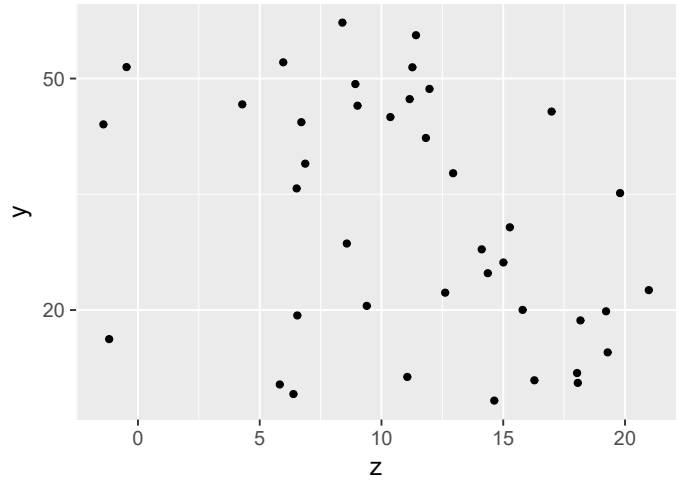
We can use `scale_x_reverse` to reverse the direction of a continuous scale,

```
ggplot(fake2.data, aes(z, y)) + geom_point() + scale_x_reverse()
```



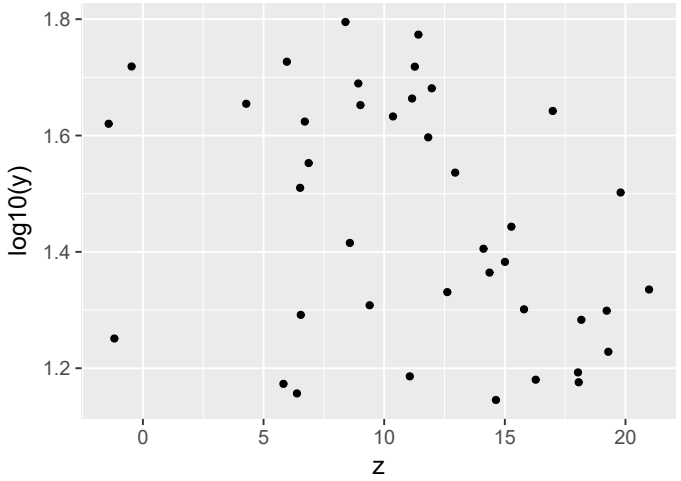
Axis tick-labels display the original values before applying the transformation. The `"breaks"` need to be given in the original scale as well. We use `scale_y_log10` to apply a $\log_{10}$ transformation to the x values.

```
ggplot(fake2.data, aes(z, y)) + geom_point() + scale_y_log10(breaks=c(10,20,50,100))
```



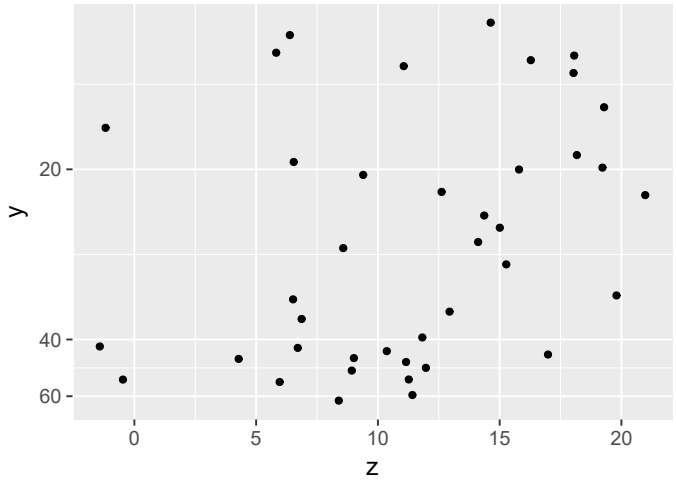
In contrast, transforming the data on-the-fly when mapping it to the x *aesthetic*, results in tick-labels expressed in the logarithm of the original data.

```
ggplot(fake2.data, aes(z, log10(y))) + geom_point()
```



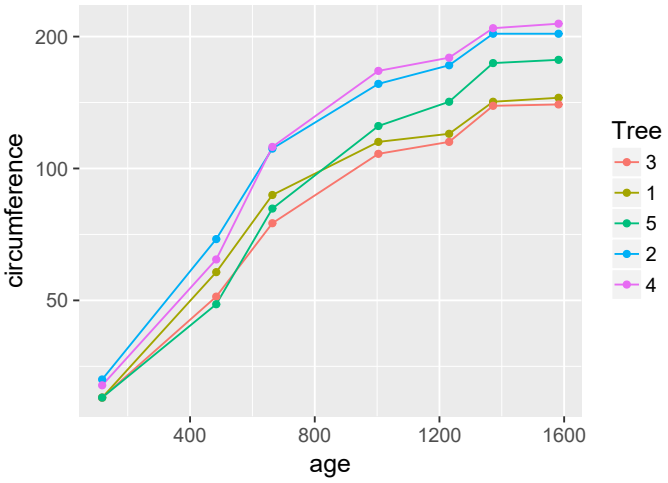
We show here how to specify a transformation to a continuous scale, using a predefined "transformation" object.

```
ggplot(fake2.data, aes(z, y)) + geom_point() +
  scale_y_continuous(trans = "reciprocal")
```



Natural logarithms are important in growth analysis as the slope against time gives the relative growth rate. We show this with the `Orange` data set.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
  scale_y_continuous(trans = "log", breaks = c(20, 50, 100, 200))
```



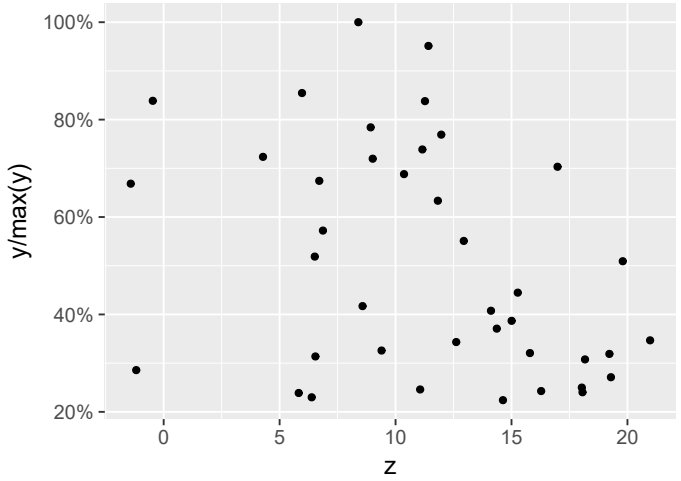
In section 6.22.3 on page 175 we define and use a transformation object.

When combining scale transformations and summaries, one should be aware of which data are used, transformed or not.

Tick labels

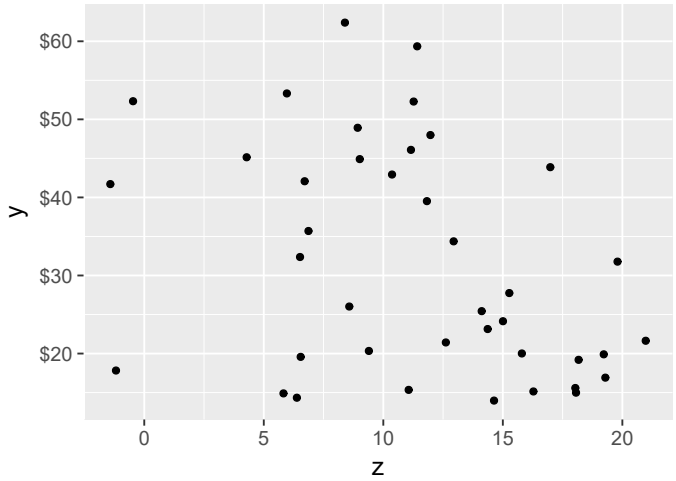
Finally, when wanting to display tick labels for data available as fractions as percentages, we can use `labels = scales::percent`.

```
ggplot(fake2.data, aes(z, y / max(y))) +
  geom_point() +
  scale_y_continuous(labels = scales::percent)
```



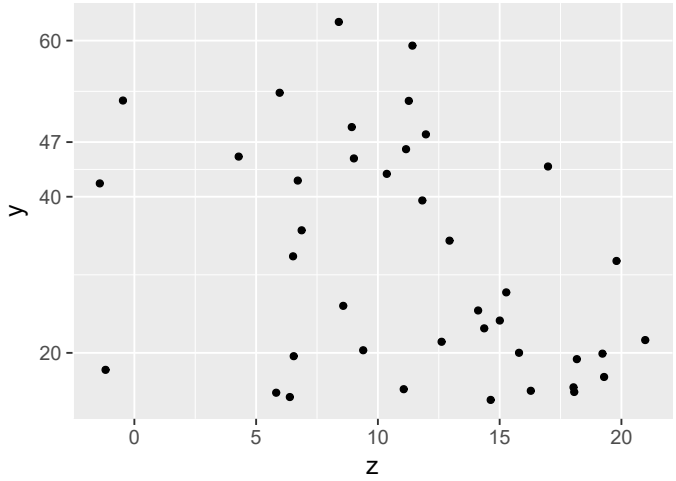
In the case of currency we can use `labels = scales::dollar`, and if we want to use commas for decimal point we can use `labels = scales::comma`.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(labels = scales::dollar)
```



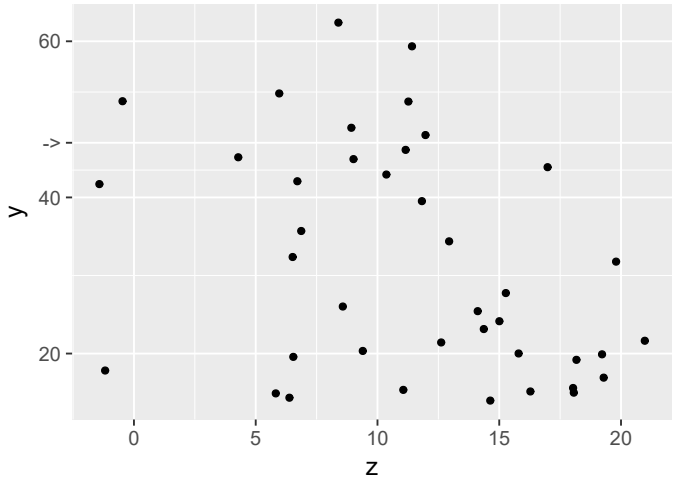
When using breaks, we can just accept the default labels for the `breaks`.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(breaks = c(20, 40, 47, 60))
```



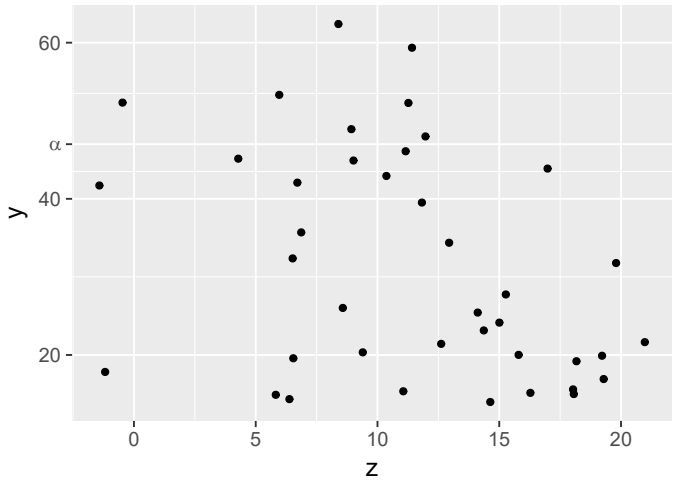
We can also set tick labels manually, in parallel to the setting of `breaks`.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(breaks = c(20, 40, 47, 60), labels = c("20", "40", "->", "60"))
```



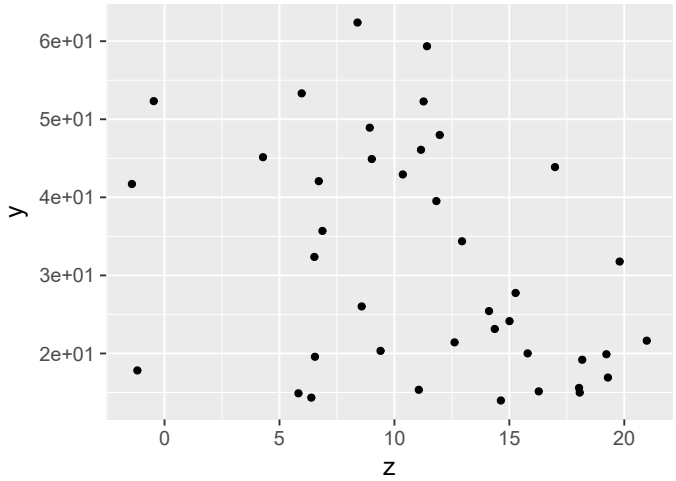
Using an expression we obtain a Greek letter.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(breaks = c(20, 40, 47, 60), labels = c("20", "40", ex-
pression(alpha), "60"))
```



We can pass a function that accepts the breaks and returns labels to `labels`. Package 'scales'defines several formatters, or we can define our own. For `log10` scales

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(labels = scales::scientific_format())
```



Please, see section 6.22.3 on page 175 for an example of the use of `scales::math_format` together with a logarithmic transformation of the data.

6.16.2 Time and date scales for x and y

Limits

Time and date scales are conceptually similar to continuous numeric scales, but use special data types and formatting for labels. We can set limits and breaks using constants as time or dates. These are most easily input with the functions in packages 'lubridate' or 'anytime'.

Please, see section **??** on page **??** for examples.

Axis labels

By default the tick labels produced and their formatting is automatically selected based on the extent of the time data. For example, if we have all data collected within a single day, then the tick labels will show hours and minutes. If we plot data for several years, the labels will show the date portion of the time instant. The default is frequently good enough, but it is possible, as for numbers to use different formatter functions to generate the tick labels.

6.16.3 Discrete scales for x and y

In the case of ordered or unordered factors, the tick labels are by default the names of the factor levels. Consequently one roundabout way to obtaining the desired tick labels is to use them as factor levels. This approach is not recommended as in most cases the text of the desired tick labels may not be recognized as a valid name making the code using them difficult to type in scripts or at the command prompt. It is best to use simple mnemonic short names for factor levels and variables, and to set suitable labels when plotting, as we will show here.

When using factors, the ordering used for plotting levels is the one they have in the factor. When a factor is created, the default is for levels to be stored in alphabetical order. This default can be easily overridden at the time of creation, as well as the order modified at a later time.

```
default.fct <- factor(c("a", "c", "f", "f", "a", "d"))
levels(default.fct)

## [1] "a" "c" "d" "f"
```

```
levels.fct <- factor(c("a", "c", "f", "f", "a", "d"),
                     levels = c("f", "a", "d", "c"))
levels(levels.fct)

## [1] "f" "a" "d" "c"
```

Reorder can be used to change the order of the levels based on the values of a numeric variable. We will visit once again the `Orange` data set.

```
my1.Tree <- with(Orange,
                 reorder(Tree, -circumference))
levels(Orange$Tree)

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

levels(my1.Tree)

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

Which is equivalent to reversing the order in this particular case.

```
my2.Tree <- with(Orange,
                 factor(Tree,
                 levels = rev(levels(Tree))))
levels(Orange$Tree)

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

levels(my2.Tree)

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

We restore the default ordering.

```
my3.Tree <- with(Orange,
                 factor(Tree,
                 levels = sort(levels(Tree))))
levels(Orange$Tree)

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

levels(my3.Tree)

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

We can set the levels in any arbitrary order by explicitly listing the level names, not only at the time of creation but also later. Here we show that it is possible to not only reorder exisitng levels, but even to add a level for which there are no observations.

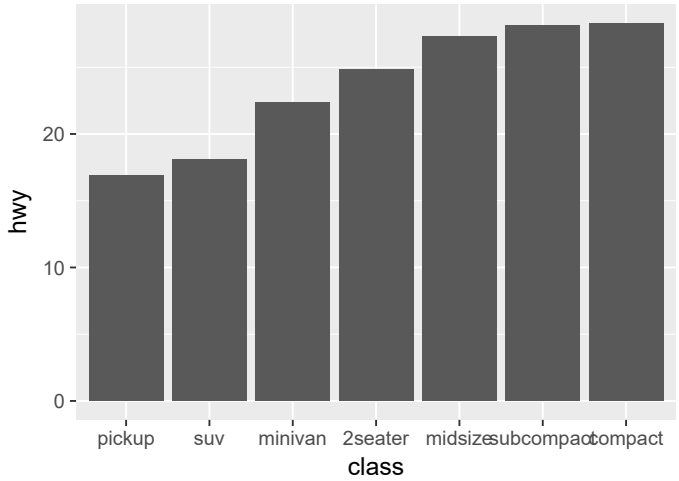
```
my3.Tree <- with(Orange,
                 factor(Tree,
                 levels = c("1", "2", "3", "4", "5", "9")))
levels(Orange$Tree)

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

levels(my3.Tree)

## [1] "1" "2" "3" "4" "5" "9"
```

We use here once again the `mpg` data set.

We order the columns in the plot based on `mpg$hwy` by reordering `mpg$class`. This approach makes sense if this ordering is needed for all plots. It is always bad to keep several versions of a single data set as

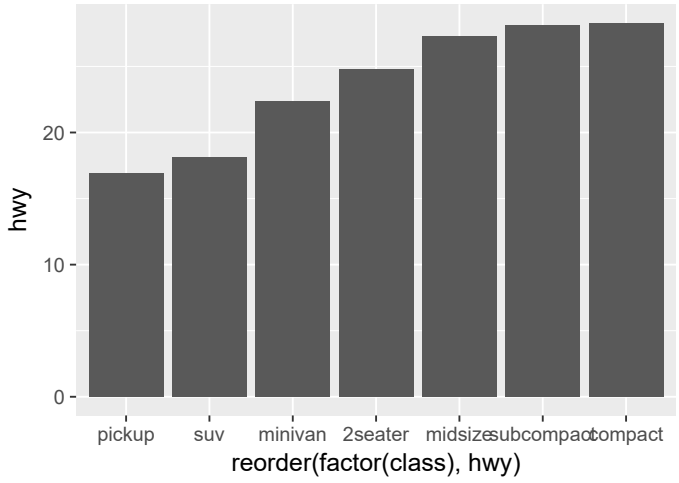
it easily leads to mistakes and confusion.

```
my.mpg <- mpg
my.mpg$class <- with(my.mpg, reorder(factor(class), hwy))
ggplot(my.mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun.y = mean)
```



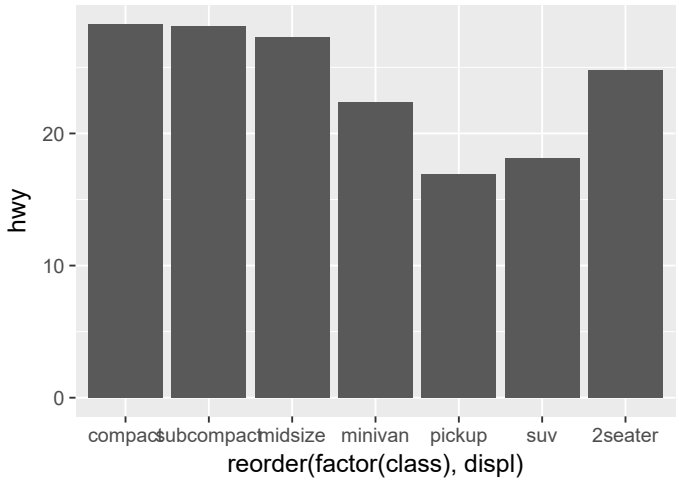
Or the same on-the-fly, which is much better as the data remains unmodified..

```
ggplot(mpg, aes(reorder(factor(class), hwy), hwy)) +
  stat_summary(geom = "col", fun.y = mean)
```



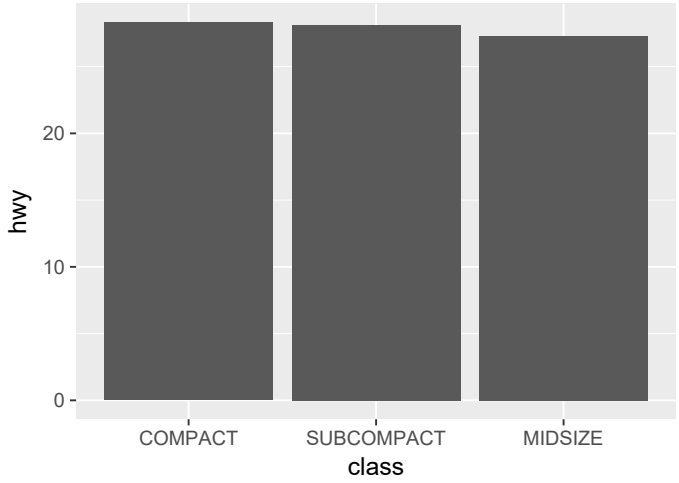
Or ordering based on a different variable, `displ`.

```
ggplot(mpg, aes(reorder(factor(class), displ), hwy)) +
  stat_summary(geom = "col", fun.y = mean)
```



Alternatively we can use `scale_x_discrete` to reorder and select the columns without altering the data. If we use this approach to subset the data, then to avoid warnings we need to add `na.rm = TRUE`. We use the scale in this example to convert level names to uppercase. The complementary function of `toupper` is `tolower`.

```
ggplot(mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun.y = mean, na.rm = TRUE) +
  scale_x_discrete(limits = c("compact", "subcompact", "midsize"),
  labels = toupper)
```



6.16.4 Size

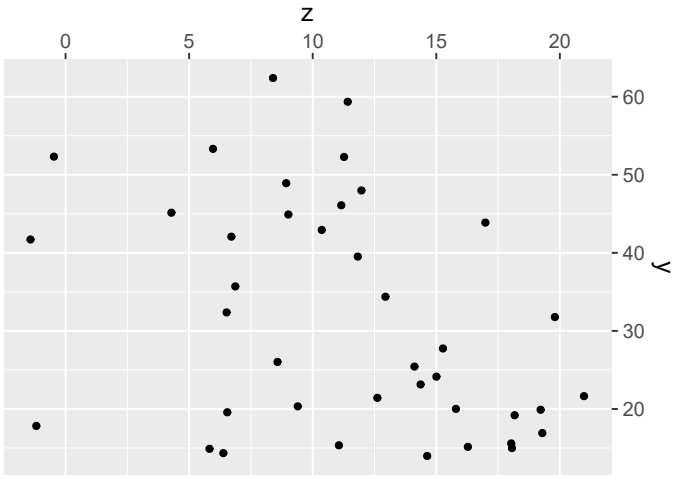
For the `size` *aesthetic* several scales are available, both discrete and continuous. They do not differ much from those already described above. *Geometries* `geom_point`, `geom_line`, `geom_hline`, `geom_vline`, `geom_text`, `geom_label` obey `size` as expected. In the case of `geom_bar`, `geom_col`, `geom_area` and all other geometric elements bordered by lines, `size` is obeyed by these border lines. In fact, other aesthetics natural for lines such as `linetype` also apply to these borders.

When using `size` scales, `breaks` and `labels` affect the key or `guide`. In scales that produce a key passing `guide = FALSE` removes the key corresponding to the scale.

6.16.5 Color and fill

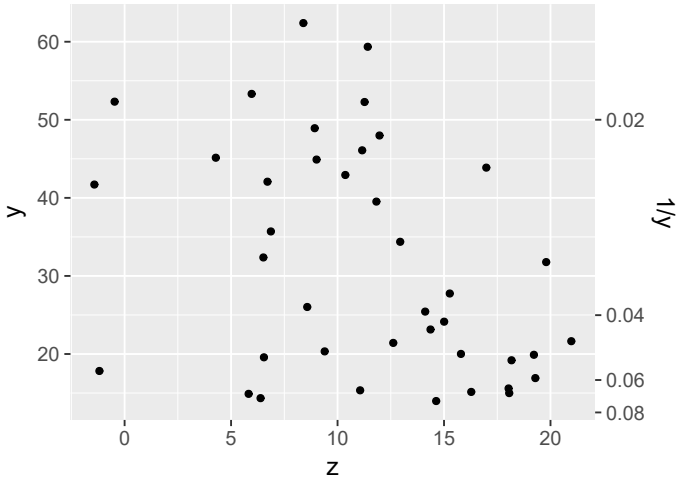
6.16.6 Position of axes

```
ggplot(fake2.data, aes(z, y)) + geom_point() +
  scale_x_continuous(position = "top") +
  scale_y_continuous(position = "right")
```

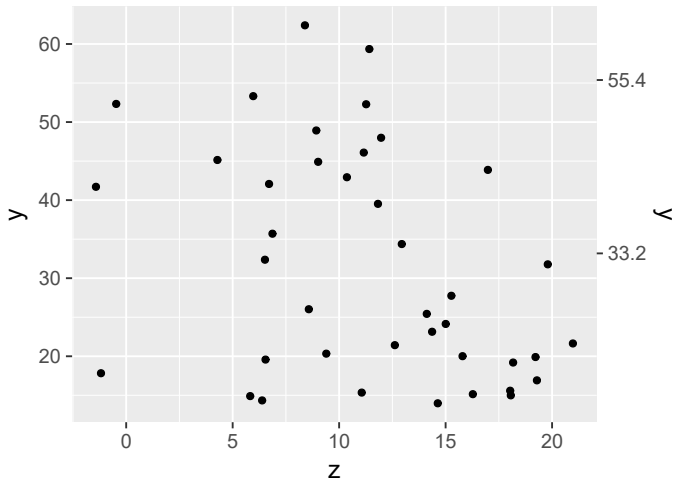


6.16.7 Secondary axes

```
ggplot(fake2.data, aes(z, y)) + geom_point() +
  scale_y_continuous(
    "y",
    sec.axis = sec_axis(~ . ^-1, name = "1/y")
  )
```



```
ggplot(fake2.data, aes(z, y)) + geom_point() +
  scale_y_continuous(
    "y",
    sec.axis = sec_axis(~ ., name = "y", breaks = c(33.2, 55.4))
  )
```

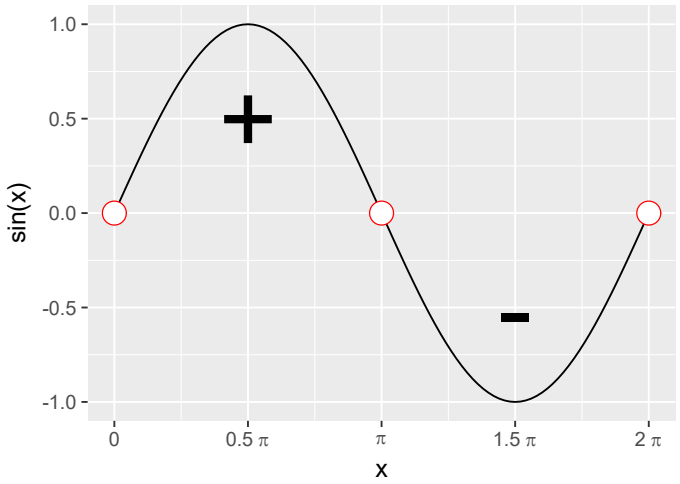


6.17 Adding annotations

Annotations use the data coordinates of the plot, but do not 'inherit' data or aesthetics from the `ggplot` object. In this example we pass directly expressions as tick labels through the scale. Do notice that we use recycling for setting the breaks, as `c(0, 0.5, 1, 1.5, 2) * pi` is equivalent to `c(0, 0.5 * pi, pi, 1.5 * pi, 2 * pi)`. Annotations are plotted at their own position, unrelated to any observation in the data.

```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
  labs(y="sin(x)") +
  annotate(geom="text",
           label=c("+", "-"),
           x=c(0.5, 1.5) * pi, y=c(0.5, -0.5),
           size=20) +
  annotate(geom="point",
```

```
            colour="red",
            shape=21,
            fill="white",
            x=c(0, 1, 2) * pi, y=0,
            size=6)
```



6.18 Themes

6.18.1 Predefined themes

6.18.2 Tweaking a theme

6.18.3 Defining a new theme

6.19 Advanced topics

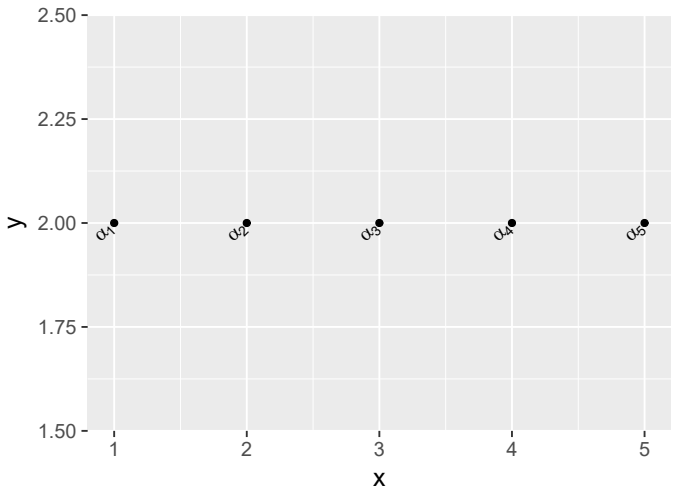
6.20 Using `plotmath` expressions

Expressions are very useful but rather tricky to use because the syntax is unusual. In `ggplot` one can either use expressions explicitly, or supply them as character string labels, and tell `ggplot` to parse them. For titles, axis-labels, etc. (anything that is defined with `labs`) the expressions have to be entered explicitly, or saved as such into a variable, and the variable supplied as argument. When plotting expressions using `geom_text` expression arguments should be supplied as character strings and the

optional argument `parse = TRUE` used to tell the geom to interpret the labels as expressions. We will go through a few useful examples.

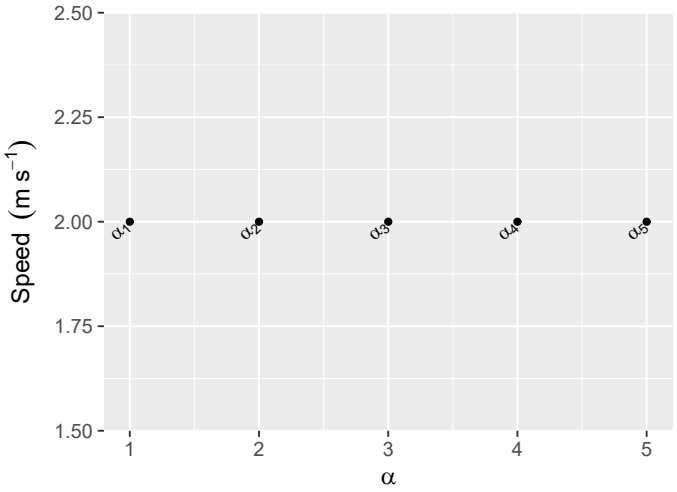
We will revisit the example from the previous section, but now using subscripted Greek α for labels. In this example we use as subscripts numeric values from another variable in the same dataframe.

```
my.data <-
  data.frame(x = 1:5, y = rep(2, 5),
             label = paste("alpha[", 1:5, "]", sep = ""))
my.data$greek.label <- paste("alpha[", my.data$x, "]", sep="")
(fig <- ggplot(my.data, aes(x,y,label=greek.label)) +
   geom_text(angle=45, hjust=1.2, parse=TRUE) + geom_point())
```



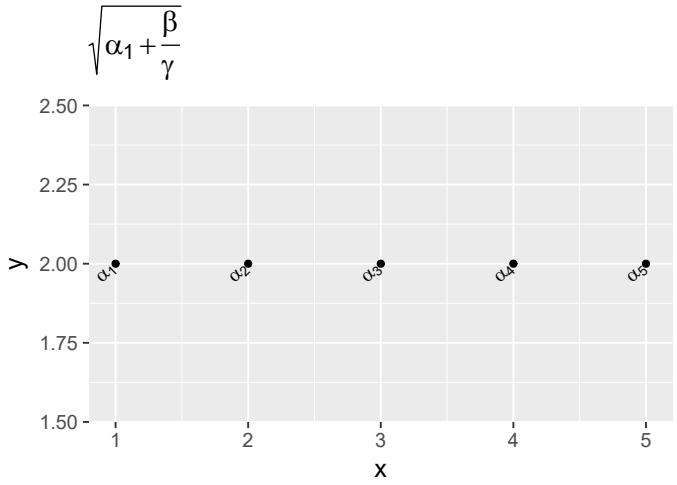
Setting an axis label with superscripts. The easiest way to deal with spaces is to use ‘ ’ or ‘ ’. One can connect pieces that would otherwise cause errors using ‘*’. If we

```
fig + labs(x=expression(alpha), y=expression(Speed~~(m~s^{-1})))
```



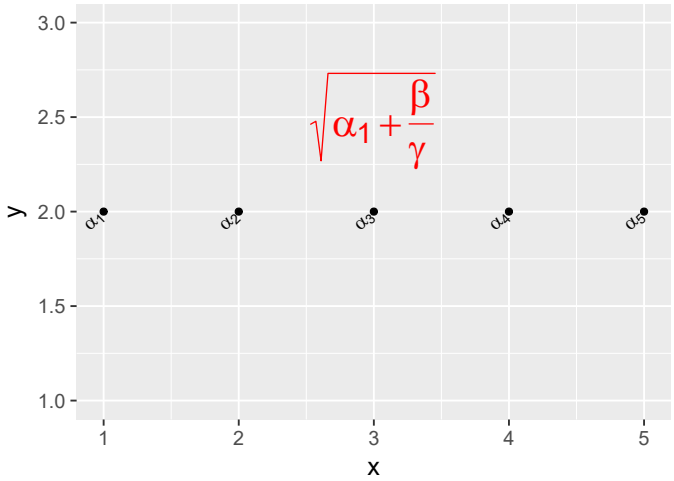
It is possible to store expressions in variables.

```
my.title <- expression(sqrt(alpha[1] + frac(beta, gamma)))
fig + labs(title=my.title)
```



Annotations are plotted ignoring the default aesthetics, but still make use of geoms, so labels for annotations also have to be supplied as character strings and parsed.

```
fig + ylim(1,3) +
  annotate("text", label="sqrt(alpha[1] + frac(beta, gamma))",
           y=2.5, x=3, size=8, colour="red", parse=TRUE)
```



hWe discuss how to use expressions as facet labels in section **??**.

6.21 Generating output files

It is possible, when using RStudio, to directly export the displayed plot to a file. However, if the file will have to be generated again at a later time, or a series of plots need to be produced with consistent format, it is best to include the commands to export the plot in the script.

In R, files are created by printing to different devices. Printing is directed to a currently open device. Some devices produce screen output, others files. Devices depend on drivers. There are both devices that or part of R, and devices that can be added through packages.

A very simple example of PDF output (width and height in inches):

```
fig1 <- ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun=dnorm)
pdf(file="fig1.pdf", width=8, height=6)
print(fig1)
dev.off()
```

Encapsulated Postscript output (width and height in inches):

```
postscript(file="fig1.eps", width=8, height=6)
print(fig1)
dev.off()
```

There are Graphics devices for BMP, JPEG, PNG and TIFF format bitmap files. In this case the default units for width and height is pixels. For example we can generate TIFF output:

```
tiff(file="fig1.tiff", width=1000, height=800)
print(fig1)
dev.off()
```

6.21.1 Using LaTeX instead of plotmath

To use LaTeX syntax in plots we need to use a different *software device* for output. It is called `Tikz` and defined in package 'tikzDevice'. This device generates output that can be interpreted by LaTeX either as a self-contained file or as a file to be input into another LaTeX source file. As the bulk of this handbook does not use this device, we will use it explicitly and input the files into this section. A TeX distribution should be installed, with LaTeX and several (LaTeX) packages including 'tikz'.

6.21.2 Fonts

Font face selection, weight, size, maths, etc. are set with LaTeX syntax. The main advantage of using LaTeX is the consistency between the typesetting of the text body and figure labels and legends. For those familiar with LaTeX not having to remember/learn the syntax of plotmath will a bonus.

We will revisit the example from the previous sections, but now using LaTeX for the subscripted Greek α for labels instead of `plotmath`. In this example we use as subscripts numeric values from another variable in the same dataframe.

6.22 Examples

In this section we first produce some publication-ready plots requiring the use of different combinations of what has been presented earlier in this chapter and then we recreate some well known plots, using versions from Wikipedia articles as models. Our objective here is to show, how by combining different words and modifiers from the grammar of graphics we can build step by step very complex plots and/or annotate them with sophisticated labels. Here we do not any packages extending 'ggplot2'2. Even more elaborate versions are presented in later chapters using 'ggplot2'with other packages.

6.22.1 Heat maps

Heat maps are 3D plots, with two axes with cartesian coordinates giving origin to rectangular tiles, with a third dimension represented by the `fill` of the tiles. They are used to describe deviations from a reference or controls condition, with for example, blue representing values below the reference and red above. A color gradient represents the size of the deviation. Simple heat maps can be produced directly with 'ggplot2'functions and methods. Heat maps with similitude trees obtained through clustering require additional tools.

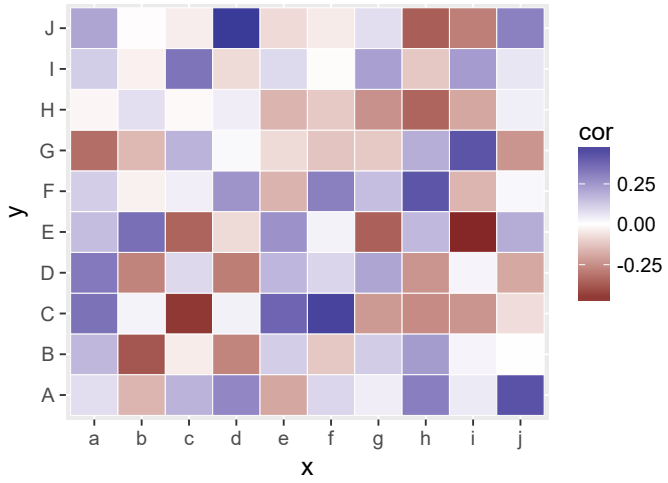
The main difference with a generic tile plot (See section 6.9 on page 105) is that the fill scale is centred on zero and the red to blue colours used for fill represent a "temperature". Nowadays, the name *heatmap* is also used for tile plots using other color for fill, as long as they represent deviations from a central value.

To obtain a heat map, then we need to use as fill scale `scale_fill_gradient2`. In the first plot we use the default colors for the fill, and in second example we use different ones.

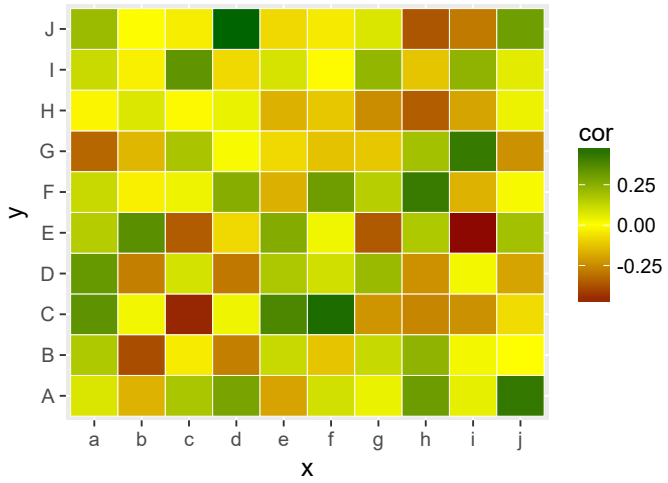
For the examples in this section we use artificial data to build a correlation matrix, which we convert into a data frame before plotting.

```
set.seed(123)
x <- matrix(rnorm(200), nrow=20, ncol=10)
y <- matrix(rnorm(200), nrow=20, ncol=10)
cor.mat <- cor(x,y)
cor.df <- data.frame(cor = as.vector(cor.mat),
           x = rep(letters[1:10], 10),
           y = LETTERS[rep(1:10, rep(10, 10))])
```

```
ggplot(cor.df, aes(x, y, fill = cor)) +
  geom_tile(color = "white") +
  scale_fill_gradient2()
```




```
ggplot(cor.df, aes(x, y, fill = cor)) +
  geom_tile(color = "white") +
  scale_fill_gradient2(low = "darkred", mid = "yellow",
  high = "darkgreen")
```



6.22.2 Quadrat plots

A quadrat plot is usually a scatter plot, although sometimes lines are also used. The scales are symmetrical both for x and y and negative and

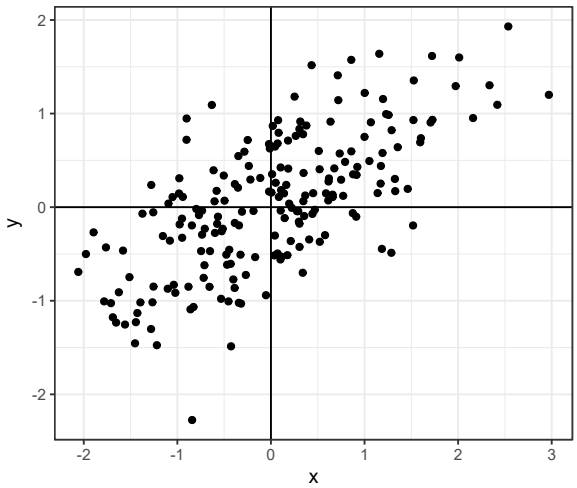
positive ranges: the origin $x = 0, y = 0$ is at the geometrical center of the plot.

We generate an artificial data set with `y` values correlated to `x` values.

```
set.seed(4567)
x <- rnorm(200, sd = 1)
quadrat_data.df <- data.frame(x = x,
                              y = rnorm(200, sd = 0.5) + 0.5 * x)
```

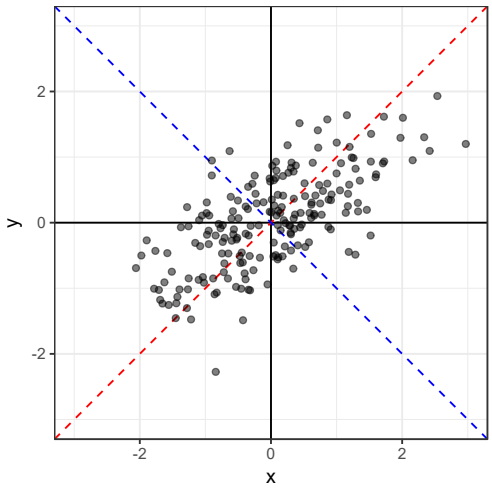
Here we draw a simple quadrat plot, by adding two lines and using fixed coordinates with a 1:1 ratio between x and y scales.

```
ggplot(data = quadrat_data.df, aes(x, y)) +
    geom_vline(xintercept = 0) +
    geom_hline(yintercept = 0) +
    geom_point() +
    coord_fixed(ratio = 1) +
    theme_bw()
```



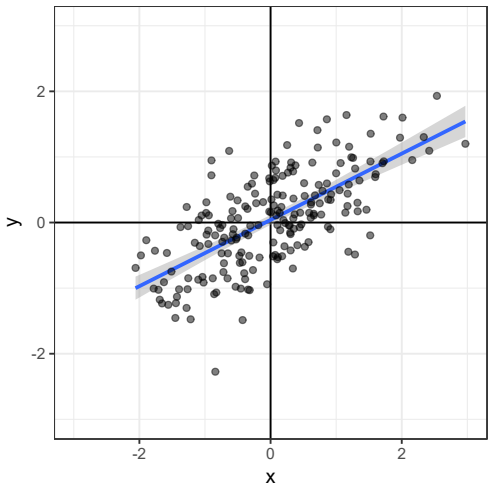
We may want to add lines showing 1:1 slopes, make the axes limits symmetric, and make points semi-transparent to allow overlapping points to be visualized. We expand the limits with `expand_limits` rather that set them with `limits` or `xlim` and `ylim`, so that if there are observations in the data set outside our target limits, the limits will still include them. In other words, we set a minimum expanse for the limits of the axes, but allow them to *grow* further if needed.

```
ggplot(data = quadrat_data.df, aes(x, y)) +
    geom_vline(xintercept = 0) +
    geom_hline(yintercept = 0) +
   geom_abline(slope = 1, intercept = 0, color = "red", linetype = "dashed") +
   geom_abline(slope = -1, intercept = 0, color = "blue", linetype = "dashed") +
    geom_point(alpha = 0.5) +
    scale_color_identity(guide = FALSE) +
    scale_fill_identity(guide = FALSE) +
    coord_fixed(ratio = 1) +
    expand_limits(x = -3, y = -3) +
    expand_limits(x = +3, y = +3) +
    theme_bw()
```



It is also easy to add a linear regression line with its confidence band.

```
ggplot(data = quadrat_data.df, aes(x, y)) +
    geom_vline(xintercept = 0) +
    geom_hline(yintercept = 0) +
    stat_smooth(method = "lm") +
    geom_point(alpha = 0.5) +
    coord_fixed(ratio = 1) +
    expand_limits(x = -3, y = -3) +
    expand_limits(x = +3, y = +3) +
    theme_bw()
```



6.22.3 Volcano plots

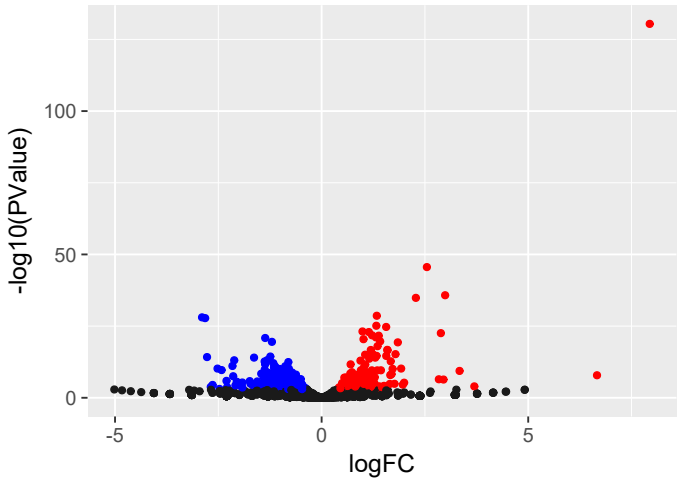
A volcano plot is just an elaborate version of a scatter plot, and can be created with 'ggplot2'functions. We here demonstrate how to create a volcano plot with tick labels in untransformed units, off-scale values drawn at the edge of the plotting region and highlighted with a different shape, and points color coded according to whether expression is significantly enhanced or depressed, or the evidence for the direction of the effect is inconclusive. We use a random sample of size 5000 from real data from an RNAseq experiment.

```
load(file = "data/volcano-example.rda")
head(clean5000.df, 4)

##            logFC     logCPM        LR     PValue
## 3949  -0.6598151 -1.1420105 1.3056330 0.25318685
## 3799  -0.2532147  4.3228981 2.5362574 0.11125823
## 23191  0.9622687  3.9734193 6.1996314 0.01277769
## 15665  0.2185446 -0.3219897 0.3089822 0.57830543
##       outcome
## 3949        0
## 3799        0
## 23191       0
## 15665       0
```

First we create a no-frills volcano plot. This is just an ordinary scatter plot, with a certain way of transforming the P-values. We

do this transformation on the fly when mapping the y aesthetic with `y = -log10(PValue)`.

```
ggplot(data = clean5000.df,
         aes(x = logFC,
             y = -log10(PValue),
             color = factor(outcome))) +
    geom_point() +
    scale_color_manual(values = c("blue", "grey10", "red"), guide = FALSE)
```

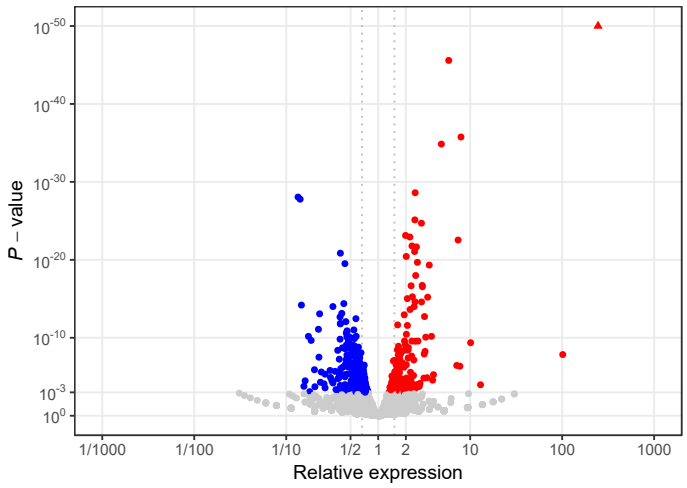


Now we add quite many tweaks to the x and y scales. 1) we show tick labels in back-transformed units, at *nice* round numbers. 2) We add publication-ready axis labels. 3) We restrict the limits of the x and y scales, but use `oob = scales::squish` so that instead of being dropped observations outside the range limits are plotted at the limit and highlighted with a a different `shape`. We also use the black and white *theme* instead of the default one.

As we assume the reverse log transformation to be generally useful we define a function `reverselog_trans` for it. In the plot we use this function to set the transformation as part of the y-scale definition, so that we can directly map P-values to the y *aesthetic*.

```
reverselog_trans <- function(base = exp(1)) {
  trans <- function(x) -log(x, base)
  inv <- function(x) base^(-x)
  scales::trans_new(paste0("reverselog-", format(base)), trans, inv,
                    scales::log_breaks(base = base),
                    domain = c(1e-100, Inf))
```

```
}

ggplot(data = clean5000.df,
         aes(x = logFC,
             y = PValue,
             color = factor(outcome),
             shape = factor(ifelse(PValue <= 1e-50, "out", "in")))) +
    geom_vline(xintercept = c(log2(2/3), log2(3/2)), linetype = "dotted",
               color = "grey75") +
    geom_point() +
   scale_color_manual(values = c("blue", "grey80", "red"), guide = FALSE) +
   scale_x_continuous(breaks = c(log2(1e-3), log2(1e-2), log2(1e-1), log2(1/2),
                                 0, log2(2), log2(1e1), log2(1e2), log2(1e3)),
                      labels = c("1/1000", "1/100", "1/10", "1/2", "1",
                                 "2", "10", "100", "1000"),
                      limits = c(log2(1e-3), log2(1e3)),
                      name = "Relative expression",
                      minor_breaks = NULL) +
    scale_y_continuous(trans = reverselog_trans(10),
                       breaks = c(1, 1e-3, 1e-10, 1e-20, 1e-30, 1e-40, 1e-50),
                      labels = scales::trans_format("log10",
                                                    scales::math_format(10^.x)),
                      limits = c(1, 1e-50), # axis is reversed!
                      name = expression(italic(P)-{value}),
                      oob = scales::squish,
                      minor_breaks = NULL) +
    scale_shape(guide = FALSE) +
    theme_bw()
```



6.22.4 Anscombe's regression examples

This is another figure from Wikipedia `http://commons.wikimedia.org/wiki/File:Anscombe.svg?uselang=en-gb`.

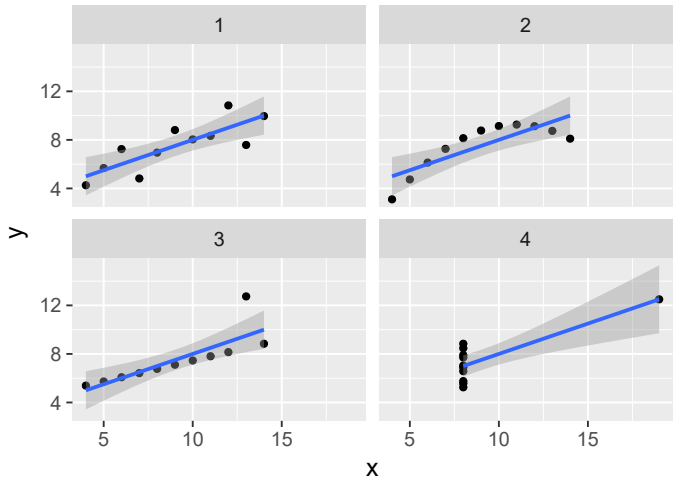
This classical example form Anscombe (1973) demonstrates four very different data sets that yield exactly the same results when a linear regression model is fit to them, including $R^2 = 0.666$. It is usually presented as a warning about the need to check model fits beyond looking at R^2 and other parameter's estimates.

I will redraw the Wikipedia figure using 'ggplot2', but first I rearrange the original data.

```
# we rearrange the data
my.mat <- matrix(as.matrix(anscombe), ncol=2)
my.anscombe <- data.frame(x = my.mat[ , 1],
                          y = my.mat[ , 2],
                          case=factor(rep(1:4, rep(11,4))))
```

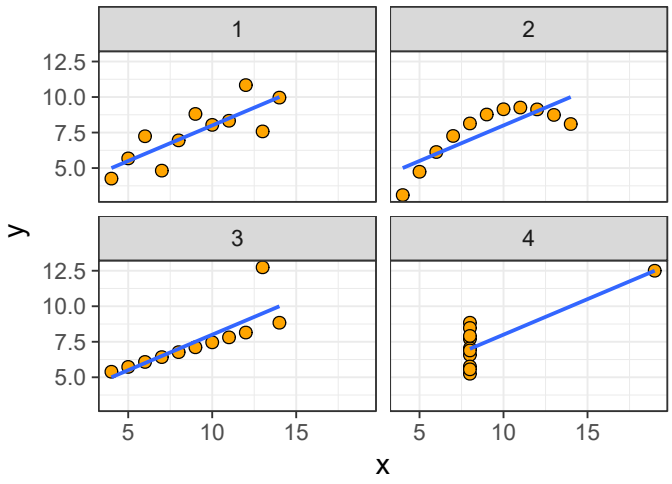
Once the data is in a data frame, plotting the observations plus the regression lines is easy.

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point() +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2)
```



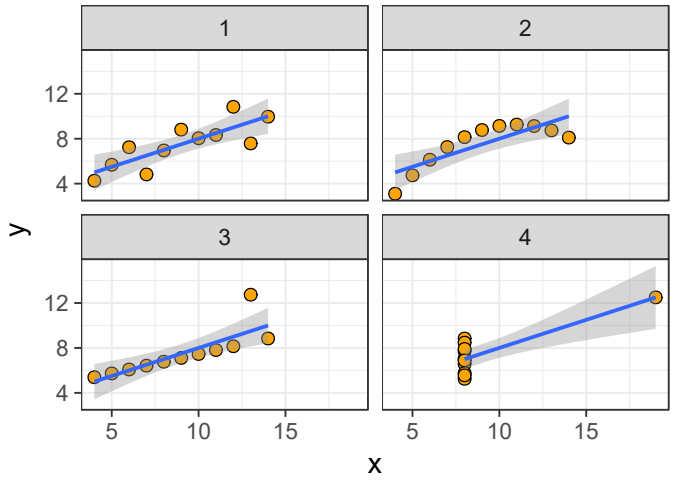
It is not much more difficult to make it look similar to the Wikipedia original.

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm", se=FALSE) +
  facet_wrap(~case, ncol=2) +
  theme_bw(16)
```



Although I think that the confidence bands make the point of the example much clearer.

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2) +
  theme_bw(16)
```

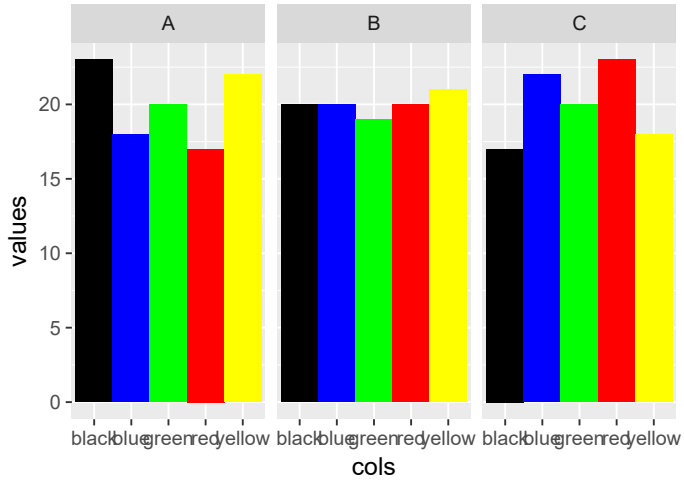


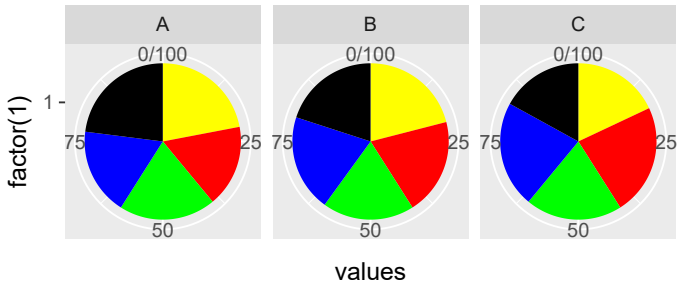
6.23 Pie charts vs. bar plots example

There is an example figure widely used in Wikipedia to show how much easier it is to 'read' bar plots than pie charts (`http://commons.wikimedia.org/wiki/File:Piecharts.svg?uselang=en-gb`).

Here is my 'ggplot2'version of the same figure, using much simpler code and obtaining almost the same result.

```
example.data <-
  data.frame(values = c(17, 18, 20, 22, 23,
                        20, 20, 19, 21, 20,
                        23, 22, 20, 18, 17),
             examples= rep(c("A", "B", "C"), c(5,5,5)),
             cols = rep(c("red", "blue", "green", "yellow", "black"), 3)
  )

ggplot(example.data, aes(x=cols, y=values, fill=cols)) +
  geom_col(width = 1) +
  facet_grid(.~examples) +
  scale_fill_identity()
ggplot(example.data, aes(x=factor(1), y=values, fill=cols)) +
  geom_col(width = 1) +
  facet_grid(.~examples) +
  scale_fill_identity() +
  coord_polar(theta="y")
```






```
try(detach(package:tikz))
try(detach(package:ggplot2))
```

7 Extensions to 'ggplot2'

7.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(tibble)
library(ggplot2)
library(viridis)
library(ggrepel)
library(ggpmisc)
library(ggseas)
library(gganimate)
library(ggstance)
library(ggbiplot)
library(ggalt)
library(xts)
library(MASS)
```

We set a font larger size than the default

```
theme_set(theme_grey(14))
```

7.2 Introduction

In this chapter we describe some of the packages that add additional functionality or *graphical designs* of plots to package 'ggplot2'. Several new packages were written after 'ggplot2' version 2.0.0 was released, because this version for the first time made it straightforward to write these extensions. To keep up-to-date with the release of new extensions I recommend to regularly check the site 'ggplot2 Extensions' (maintained by Daniel Emaasit) at `https://www.ggplot2-exts.org/`.

Some of the packages described in this chapter were not yet in CRAN at the time of writing. However, it is possible to install packages directly from Github or Bitbucket using functions from package 'devtools'. We show here as examples the code needed for packages 'gganimate' and 'ggstance'.

```
devtools::install_github("dgrtwo/gganimate")
devtools::install_github("lionel-/ggstance")
```

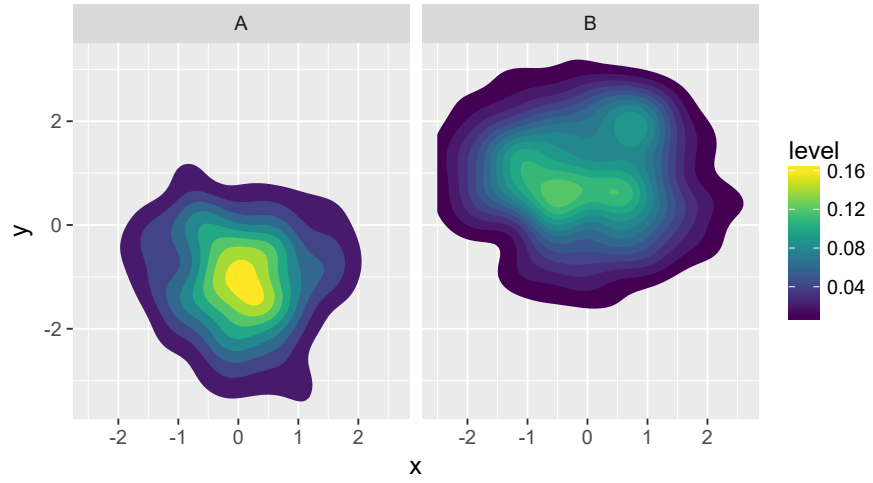
In this chapter we use mostly the modernized data frames of package 'tibble'. The main reason is that the `tibble` constructor does not by default convert character variables into factors as the `data.frame` constructor does. The format used for printing is also improved. It is possible to use `data.frame` instead of `tibble` in the examples below, but in some cases you will need to add `stringsAsFactors = FALSE` to the call.

7.3 'viridis'

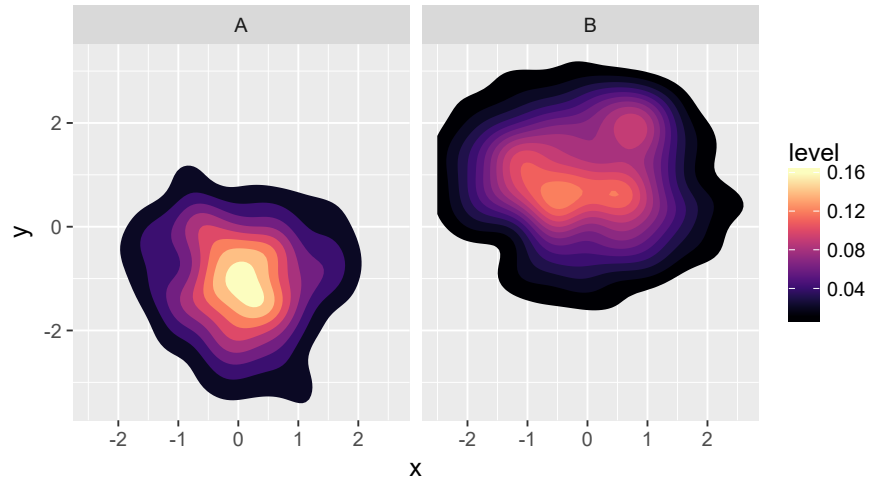
Package 'viridis'defines color palettes and fill and color scales with colour selected based on human perception, with special consideration of visibility for those with different kinds of color blindness and well as in grey-scale reproduction.

```
set.seed(56231)
my.data <- tibble(x = rnorm(500),
                  y = c(rnorm(250, -1, 1), rnorm(250, 1, 1)),
                  group = factor(rep(c("A", "B"), c(250, 250))) )
```

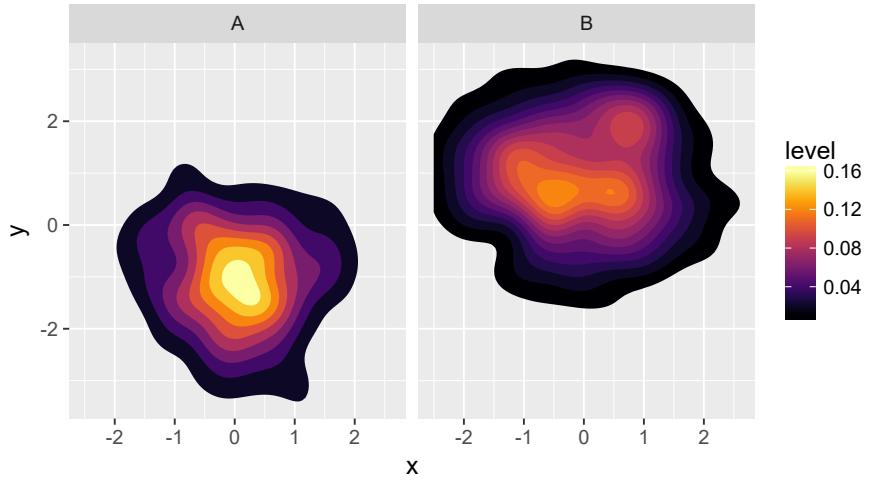
```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_viridis()
```



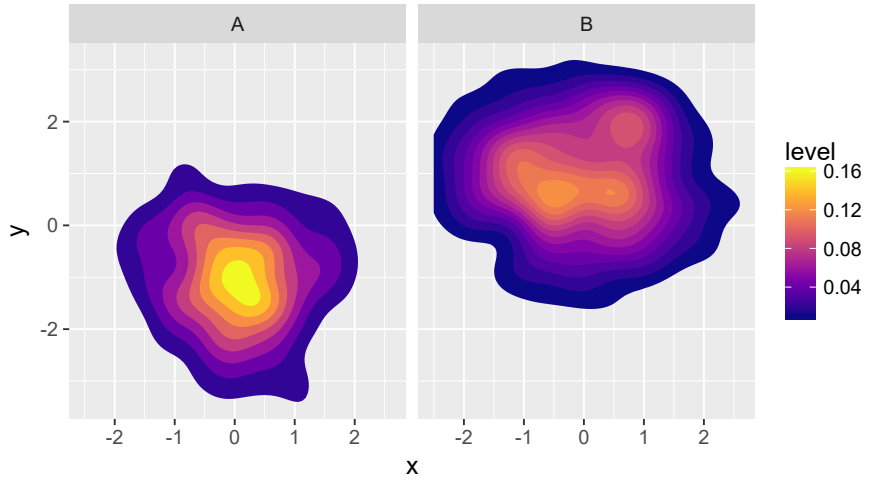

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_viridis(option = "magma")
```



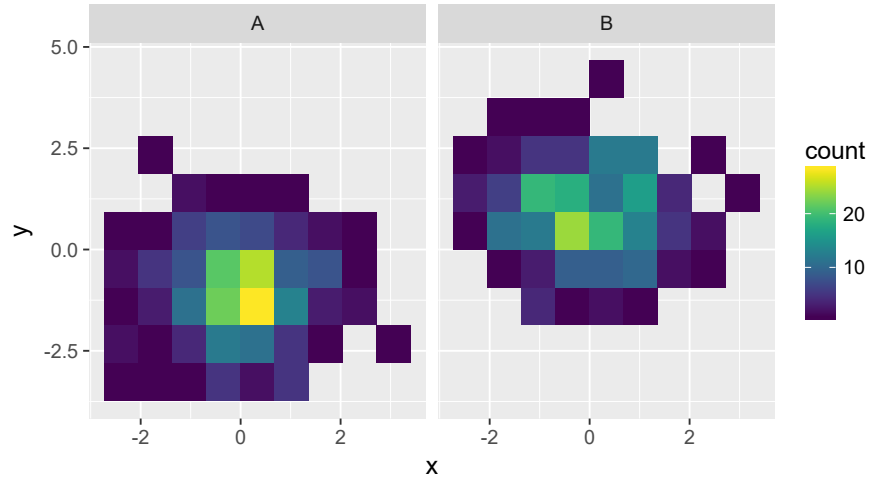

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_viridis(option = "inferno")
```



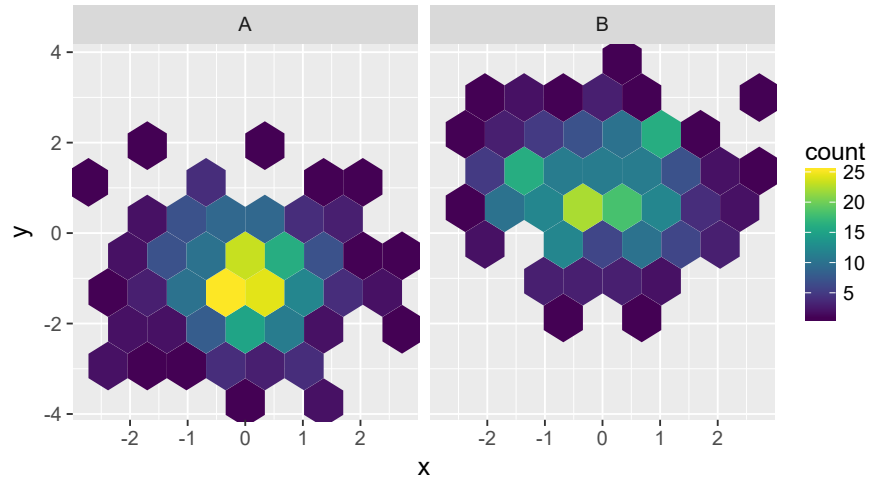

```
ggplot(my.data, aes(x, y)) +
  stat_density_2d(aes(fill = ..level..), geom = "polygon") +
  facet_wrap(~group) +
  scale_fill_viridis(option = "plasma")
```




```
ggplot(my.data, aes(x, y)) +
  geom_bin2d(bins = 8) +
  facet_wrap(~group) +
  scale_fill_viridis()
```




```
ggplot(my.data, aes(x, y)) +
  geom_hex(bins = 8) +
  facet_wrap(~group) +
  scale_fill_viridis()
```



7.4 'gganimate'

Package 'gganimate' allows the use of package 'animation' in ggplots with a syntax consistent with the grammar of graphics. It adds a new aesthetic

`frame`, which can be used to map *groups* of data to *frames* in the animation.

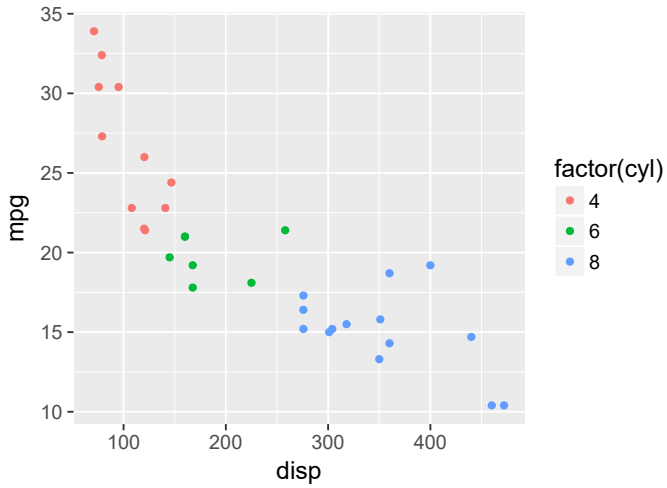
Use of the package is extremely easy, but installation can be somehow tricky because of system requirements. Just, make sure to have ImageMagic installed and included in the search `PATH`.

We modify an example from section 6.4 on page 6.4. We add the `frame` aesthetic to the earlier figure.

```
p <- ggplot(data = mtcars,
       aes(x = disp, y = mpg, colour = factor(cyl), frame  = cyl)) +
  geom_point()
```

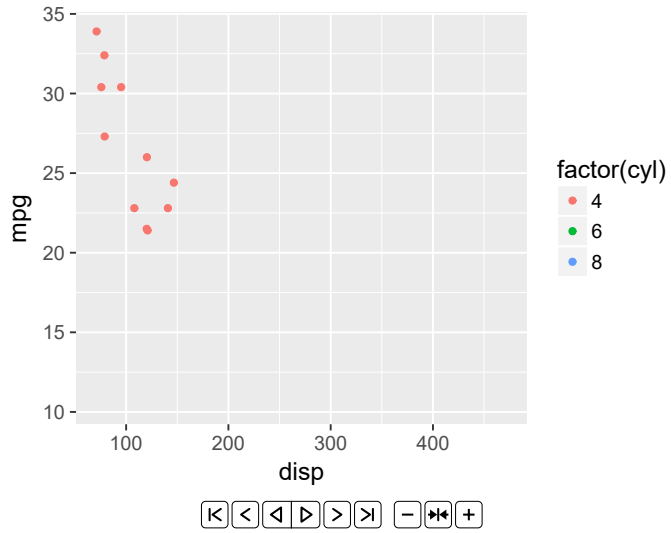
Now we can print `p` as a normal plot,

```
p
```



Or display an animation. The animation will look differently depending on the output format, and the program used for viewing it. For example, in this PDF files, the animation will work when viewed with Adobe Viewer or Adobe Acrobat but not in Sumatra PDF viewer. We add `title_frame = FALSE` as a title does not seem useful in this simple animation.

```
gg_animate(p, title_frame = FALSE)
```



Or save it to a file.

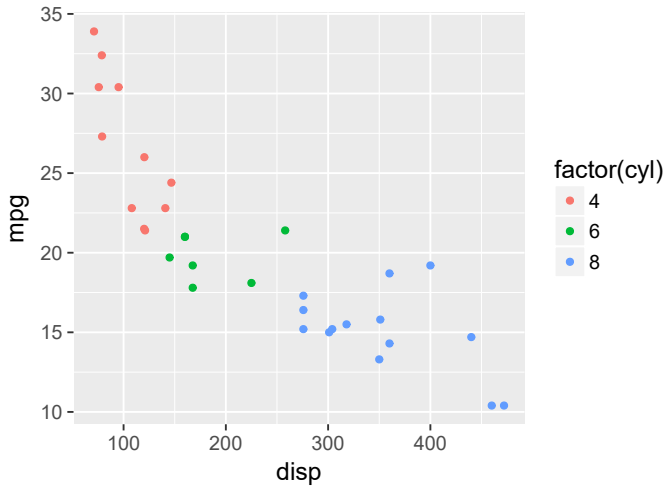
```
gg_animate(p, "p-animation.gif")
```

Cumulative animations are also supported. We here use the same example with three frames, but this type of animation is particularly effective for time series data. To achieve this we only need to add `cumulative = TRUE` to the aesthetics mappings.

```
p <- ggplot(data = mtcars,
       aes(x = disp, y = mpg, colour = factor(cyl),
           frame  = cyl, cumulative = TRUE)) +
  geom_point()
```

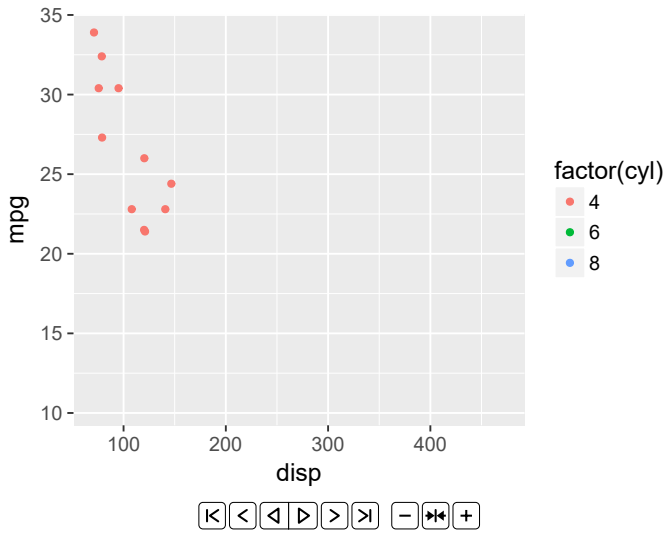
Now we can print `p` as a normal plot,

```
p
```



Or display an animation. The animation will look differently depending on the output format, and the program used for viewing it. For example, in this PDF files, the animation will work when viewed with Adobe Viewer or Adobe Acrobat but in Sumatra PDF viewer.

```
gg_animate(p, title_frame = FALSE)
```



7.5 'ggstance'

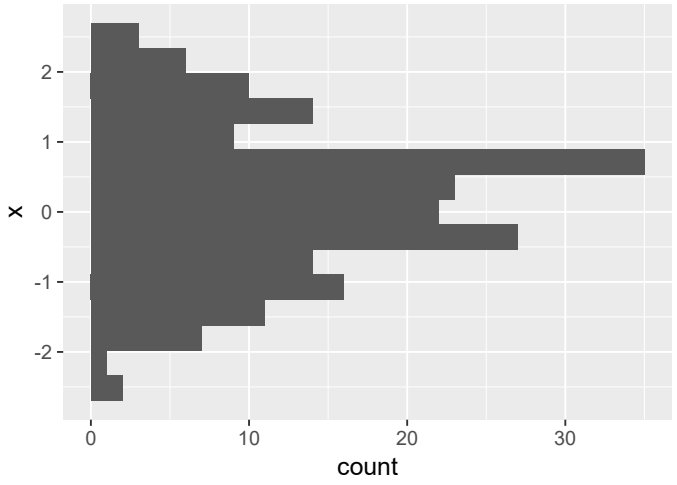
Package 'ggstance' defines horizontal versions of common ggplot *geoms*, *stats* and *positions*. Although 'ggplot2'defines `coord_flip`, 'ggstance' provides a more intuitive user interface and more consistent plot formatting.

Currently the package defines **horizontal** *geoms* `geom_barh()`, `geom_histogramh()`, `geom_linerangeh()`, `geom_pointrangeh()`, `geom_errorbarh()`, `geom_crossbarh()`, `geom_boxploth()`, and `geom_violinh()`. It also defines **horizontal** *stats* `stat_binh()`, `stat_boxploth()`, `stat_counth()`, and `stat_xdensity()` and **vertical** *positions* `position_dodgev`, `position_nudgev`, `position_fillv`, `position_stackv`, and `position_jitterdodgev`.

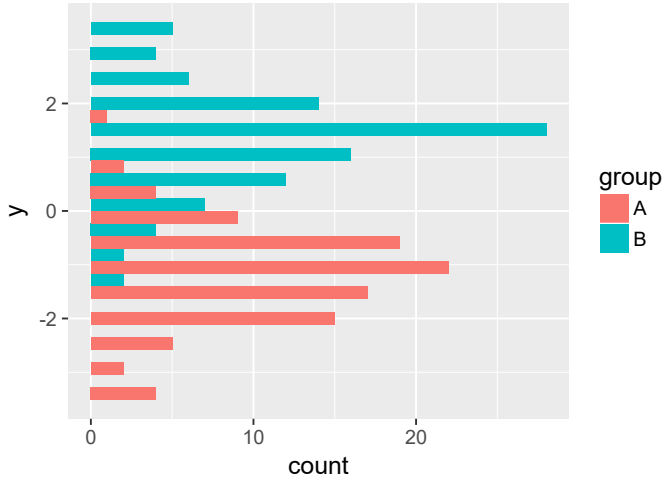
We will give give only a couple of examples, as their use has no surprises. First we make horizontal versions of the histogram plots shown in section 6.14.2 on page 129.

```
set.seed(12345)
my.data <- tibble(x = rnorm(200),
                  y = c(rnorm(100, -1, 1), rnorm(100, 1, 1)),
                  group = factor(rep(c("A", "B"), c(100, 100))) )
```

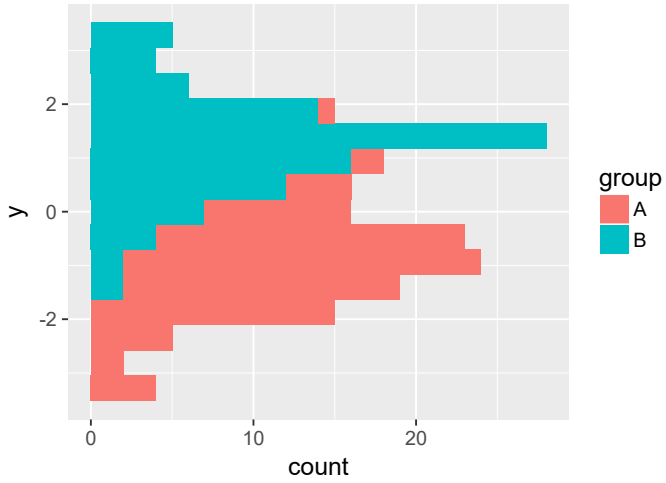
```
ggplot(my.data, aes(y = x)) +
  geom_histogramh(bins = 15)
```



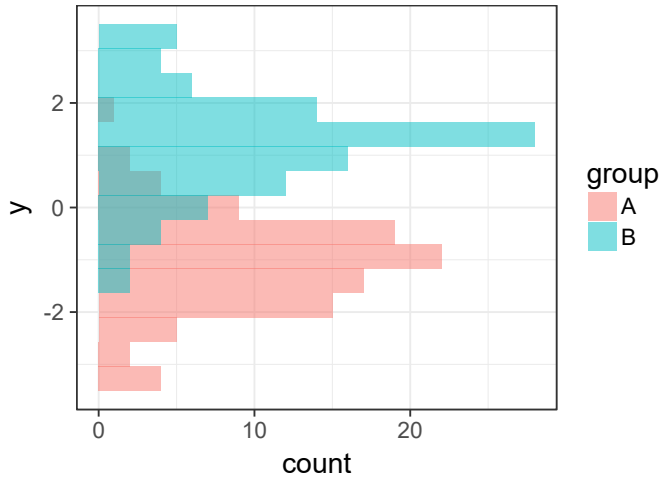
```
ggplot(my.data, aes(y = y, fill = group)) +
  geom_histogramh(bins = 15, position = "dodgev")
```




```
ggplot(my.data, aes(y = y, fill = group)) +
  geom_histogramh(bins = 15, position = "stackv")
```

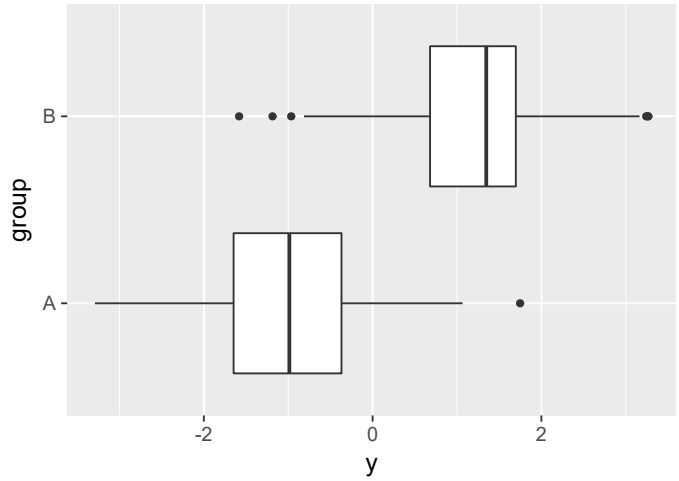



```
ggplot(my.data, aes(y = y, fill = group)) +
  geom_histogramh(bins = 15, position = "identity", alpha = 0.5) +
  theme_bw(16)
```



Now we make an horizontal version of the boxplot shown in section 6.14.4 on page 135.

```
ggplot(my.data, aes(y, group)) +
  geom_boxploth()
```

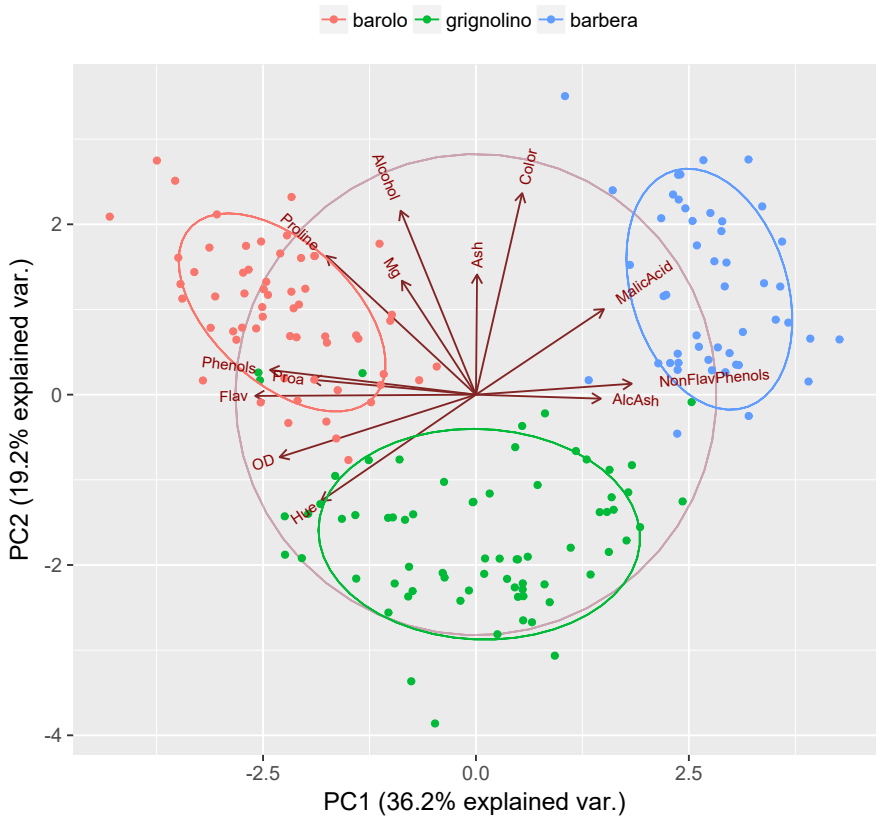


7.6 'ggbiplot'

Package 'ggbiplot' defines two functions, `ggscreeplot()` and `ggbiplot()`. These functions make it easy to nicely print the results from principal components analysis done with `prcomp`.

For the time being we reproduce an example from the package README.

```
data(wine)
wine.pca <- prcomp(wine, scale. = TRUE)
ggbiplot(wine.pca, obs.scale = 1, var.scale = 1,
  groups = wine.class, ellipse = TRUE, circle = TRUE) +
  scale_color_discrete(name = '') +
  theme(legend.direction = 'horizontal', legend.position = 'top')
```



7.7 'ggalt'

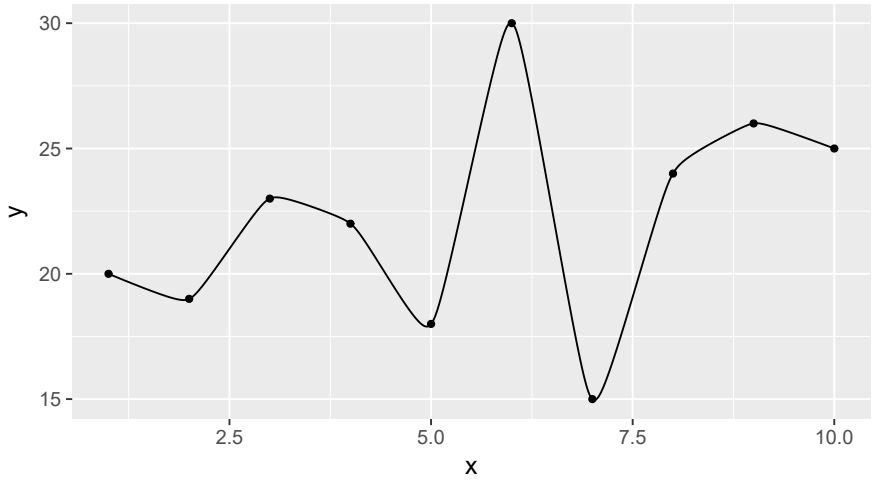
Package 'ggalt' defines *geoms* `geom_xspline()`, `geom_bkde()`, `geom_bkde2d()`, `geom_stateface()`, `geom_encircle()`, `geom_lollipop()`, `geom_dumbbell()`, and `geom_stepribbon()`; *stats* `stat_xspline()`, `stat_bkde()`, `stat_bkde2d()`, and `stat_ash()`; *scale* `scale_fill_pokemon`; *formatter* `byte_format`.

The highlights are use of functions from package 'KernSmooth' for density estimation, the provision of *X-splines* and for formating "bytes" in the usual way used when describing computer memory.

First example is the use of *x-splines* which are very flexible splines that are smooth (have a continuous first derivative). They can be tuned from interpolation (passing through every observation) to being rather "stiff" smoothers.

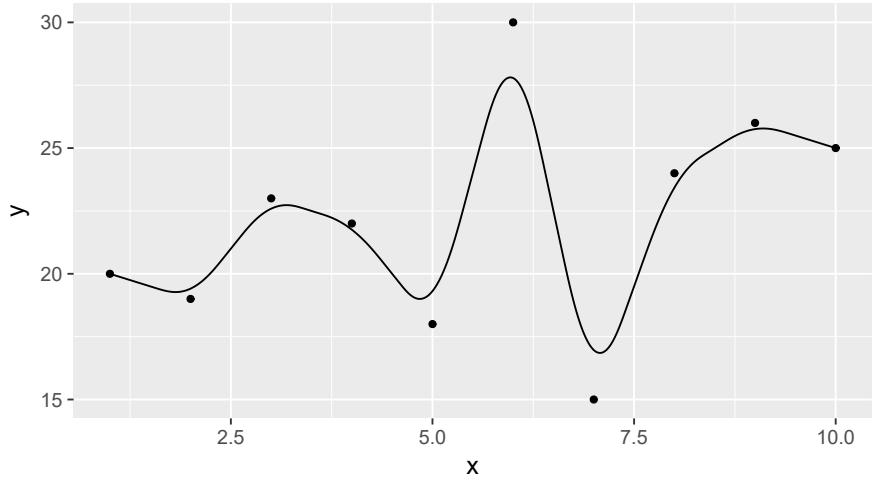
```
set.seed(1816)
dat <- tibble(x=1:10,
                  y=c(sample(15:30, 10)))
```

```
ggplot(dat, aes(x, y)) +
  geom_point() +
  geom_xspline()
```



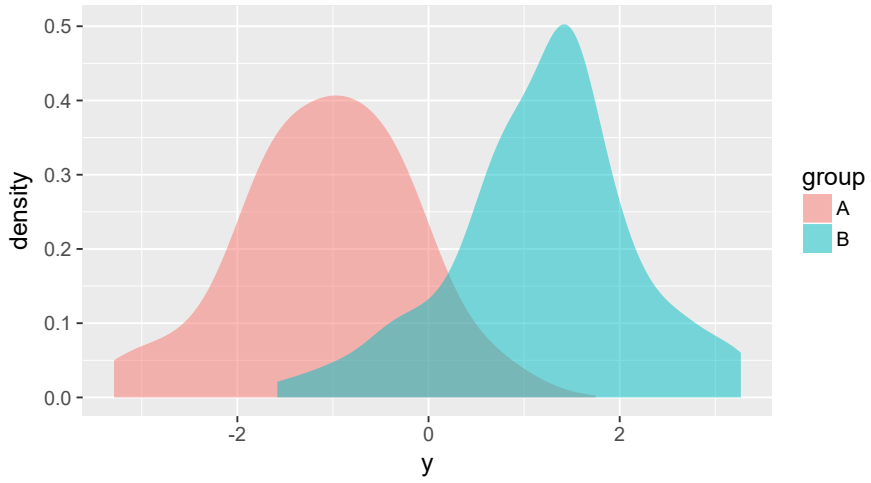
The "flexibility" of the spline can be adjusted by passing an `numeric` argument to parameter `spline_shape`.

```
ggplot(dat, aes(x, y)) +
  geom_point() +
  geom_xspline(spline_shape=0.4)
```

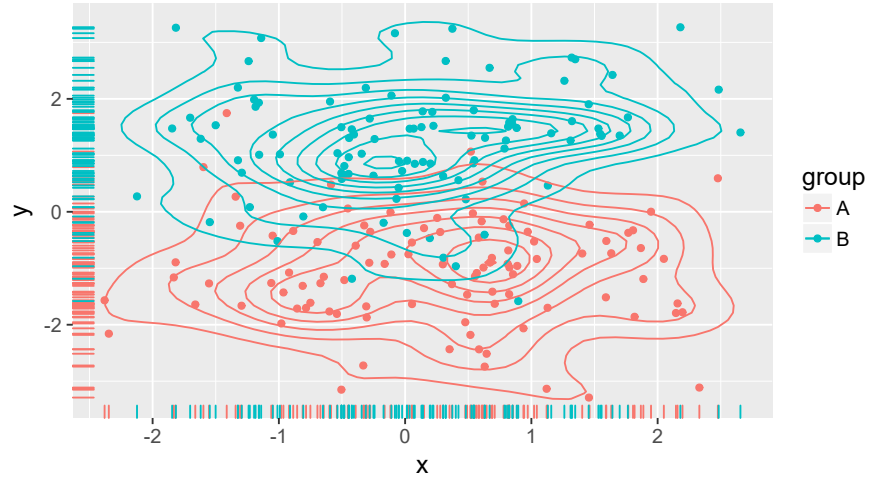


We also redo some of the densitiy plot examples from 6.14.3 on page 132.

```
ggplot(my.data, aes(y, fill = group)) +
  geom_bkde(alpha = 0.5)

## Bandwidth not specified.  Using '0.37', via KernSmooth::dpik.
## Bandwidth not specified.  Using '0.29', via KernSmooth::dpik.
```

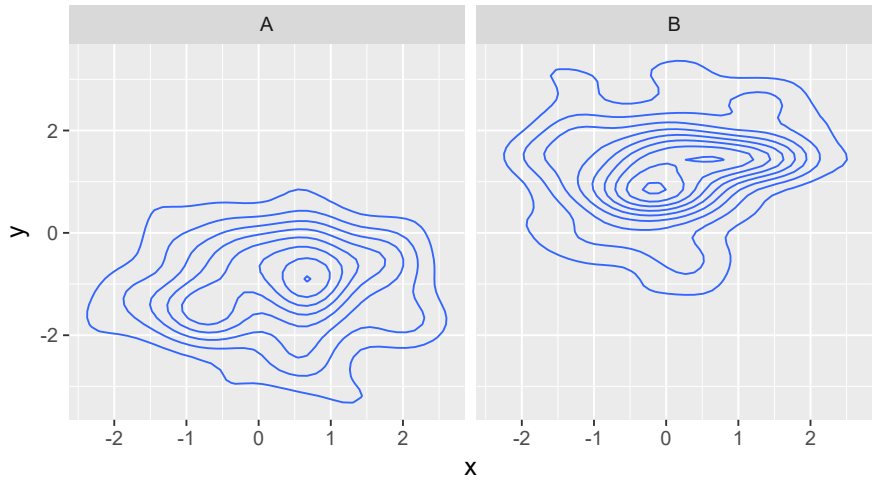



```
ggplot(my.data, aes(x, y, colour = group)) +
  geom_point() +
  geom_rug() +
  geom_bkde2d()

## Bandwidth not specified.  Using ['0.39', '0.37'], via KernSmooth::dpik.
## Bandwidth not specified.  Using ['0.42', '0.29'], via KernSmooth::dpik.
```



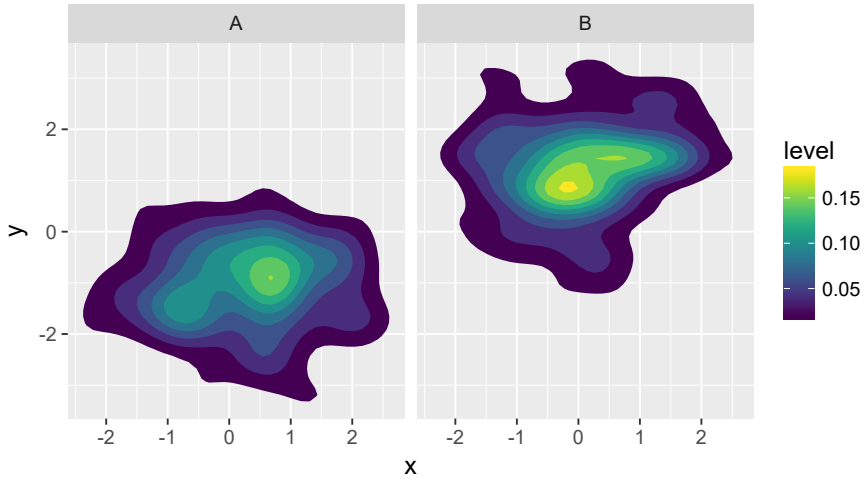

```
ggplot(my.data, aes(x, y)) +
    geom_bkde2d() +
    facet_wrap(~group)
```

```
## Bandwidth not specified. Using ['0.39', '0.37'], via KernSmooth::dpik.
## Bandwidth not specified. Using ['0.42', '0.29'], via KernSmooth::dpik.
```



We here use a scale from package ‘viridis’ described in section 7.3 on page 184.

```
ggplot(my.data, aes(x, y)) +
    stat_bkde2d(aes(fill = ..level..), geom = "polygon") +
    facet_wrap(~group) +
    scale_fill_viridis()

## Bandwidth not specified. Using ['0.39', '0.37'], via KernSmooth::dpik.
## Bandwidth not specified. Using ['0.42', '0.29'], via KernSmooth::dpik.
```



7.8 'ggExtra'

coming soon.

7.9 'ggfortify'

coming soon.

7.10 'ggradar'

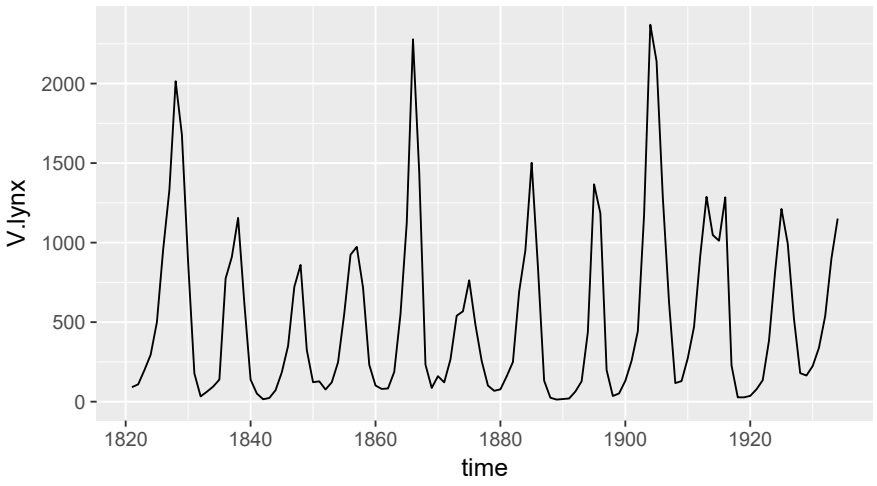
coming soon.

7.11 'ggseas'

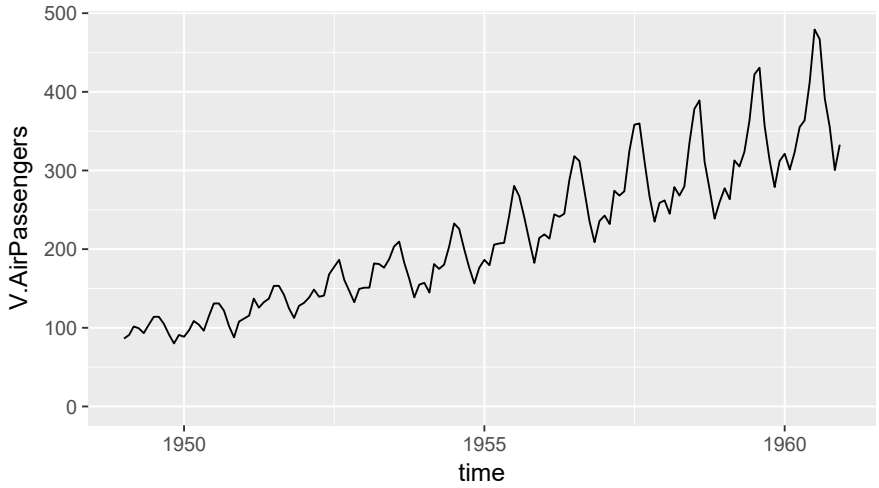
The focus of this extension to 'ggplot2' is the seasonal decomposition of time series done on the fly while creating a ggplot. Package 'ggseas' defines five *statistics*, `stat_index`, `stat_decomp`, `stat_rollapplyr`, `stat_stl`, and `stat_seas`. By default they all use `geom_line`. This package also defines function `tsdf` that needs to be used to convert time series to data frames to pass as `data` argument to `ggplot`.

Index referenced to the first two observations in the series. Here we use `try_tibble` from our package 'ggpmisc'. Function `tsdf` can be also used.

```
ggplot(try_tibble(lynx, "year", as.numeric = TRUE),
       aes(x = time, y = V.lynx)) +
       stat_index(index.ref = 1:2) +
       expand_limits(y = 0)
```



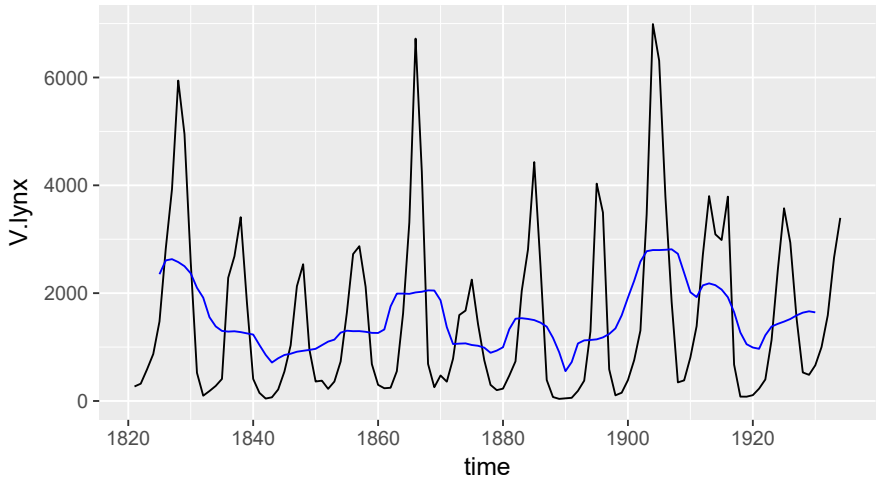

```
ggplot(try_tibble(AirPassengers, "month"),
       aes(time, V.AirPassengers)) +
       stat_index(index.ref = 1:10) +
       expand_limits(y = 0)
```



Rolling average.

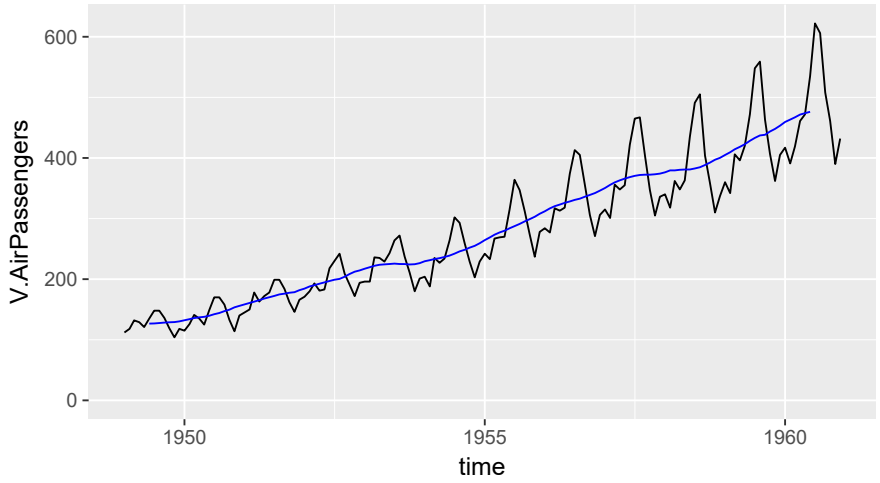
We use a with of 9, which seems to be approximately the length of the cycle.

```
ggplot(try_tibble(lynx, "year", as.numeric = TRUE),
       aes(x = time, y = V.lynx), na.rm = TRUE) +
       geom_line() +
       stat_rollapplyr(width = 9, align = "center", color = "blue") +
       expand_limits(y = 0)

## Warning: Removed 8 rows containing missing values (geom_path).
```



For monthly data on air travel, it is clear that a width of 12 observations (months) is best.

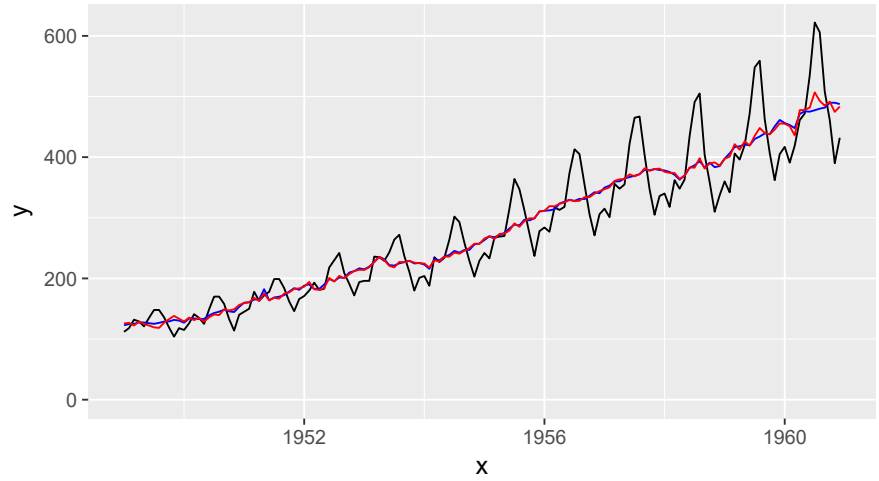
```
ggplot(try_tibble(AirPassengers, "month"),
       aes(time, V.AirPassengers)) +
       geom_line() +
       stat_rollapplyr(width = 12, align = "center", color = "blue") +
       expand_limits(y = 0)

## Warning: Removed 11 rows containing missing values (geom_path).
```



Seasonal decomposition.

Using function `tsdf` from package 'ggseas'.

```
ggplot(tsdf(AirPassengers),
       aes(x, y)) +
       geom_line() +
       stat_seas(colour = "blue") +
       stat_stl(s.window = 7, color = "red") +
       expand_limits(y = 0)

## Calculating starting date of 1949 from the data.
## Calculating frequency of 12 from the data.
## Calculating frequency of 12 from the data.
```



Currently, the `tibble` object returned by `try_tibble` gives an error, so the chunk below is not evaluated.

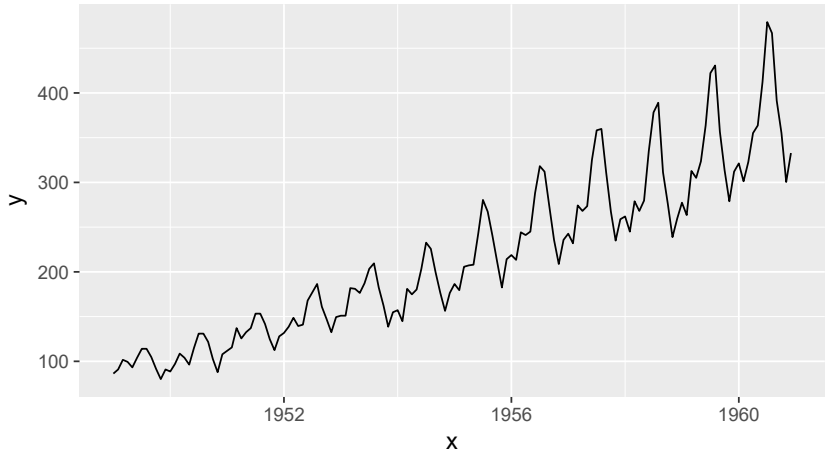
```
ggplot(try_tibble(AirPassengers, "month"),
       aes(time, V.AirPassengers)) +
       geom_line() +
       stat_seas(colour = "blue") +
       stat_stl(s.window = 7, color = "red") +
       expand_limits(y = 0)
```

As `ggplot` is a generic we can easily define a specialization for class `ts`.

```
ggplot.ts <- function(data,
                      mapping = NULL,
                      ...,
                      environment = parent.frame()) {
   if (is.null(mapping)) {
     mapping <- aes_(~x, ~y)
   }
   ggplot2::ggplot(data = ggseas::tsdf(data),
          mapping =  mapping,
          ... = ...,
          environment = environment)
}
```

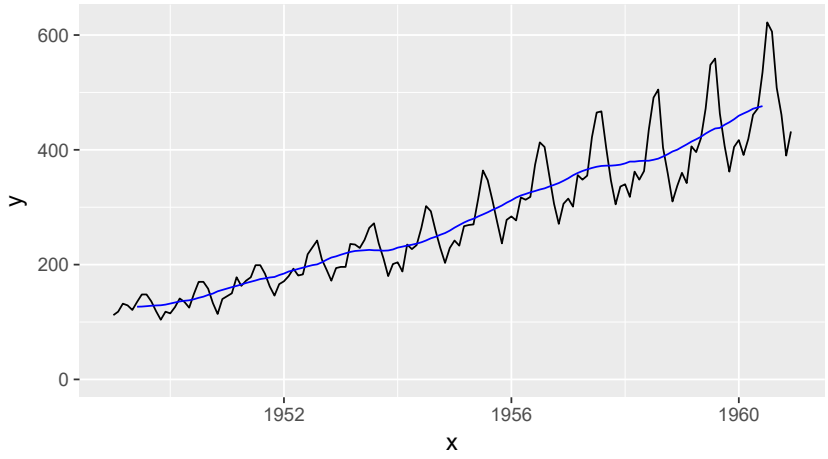
And subsequently use to plot time series objects of class `ts`, code like

```
ggplot(AirPassengers) +
    stat_index(index.ref = 1:10)
```



or

```
ggplot(AirPassengers)  +
    geom_line() +
    stat_rollapplyr(width = 12, align = "center", color = "blue") +
    expand_limits(y = 0)

## Warning: Removed 11 rows containing missing values (geom_path).
```



7.12 'ggnetwork'

coming soon.

7.13 'ggpmisc'

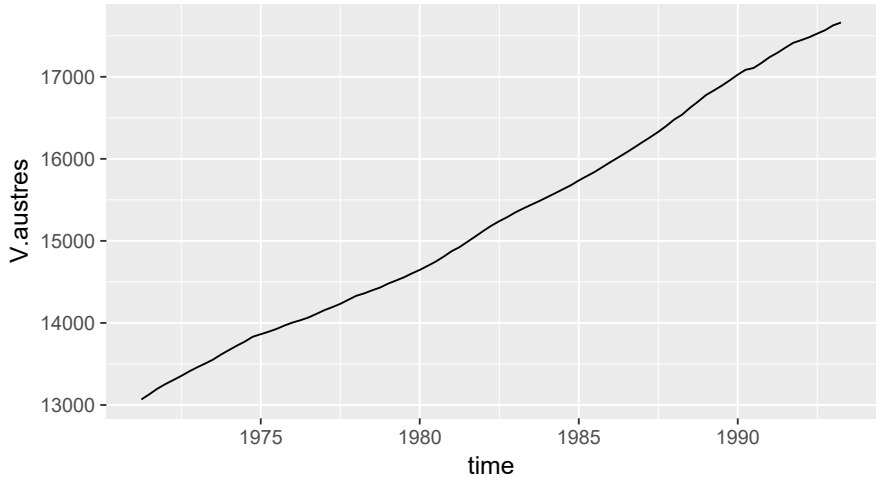
Package 'ggpmisc'is a package developed by myself as a result of questions from work mates and in Stackoverflow, or functionality that I have needed in my own research or for teaching. It provides new stats for everyday use: `stat_peaks()`, `stat_valleys()`, `stat_poly_eq()`, `stat_fit_glance()`, `stat_fit_deviations()`, and `stat_fit_augment()`. A function for converting time-series data to a data frame that can be easily plotted with 'ggplot2'. It also provides some debugging tools that echo the data received as input: `stat_debug_group()`, `stat_debug_panel()`, and codegeom_debug(), and `geom_null()` that does not plot its input.

7.13.1 Plotting time-series

Instead of creating a new statistics or geometry for plotting time series we provide a function that can be used to convert time series objects into data frames suitable for plotting with 'ggplot2'. A single function `try_tibble()` (also available as `try_data_frame()`) accepts time series objects saved with different packages as well as R's native `ts` objects. The *magic* is done mainly by package 'xts'to which we add a very simple wrapper to obtain a data frame.

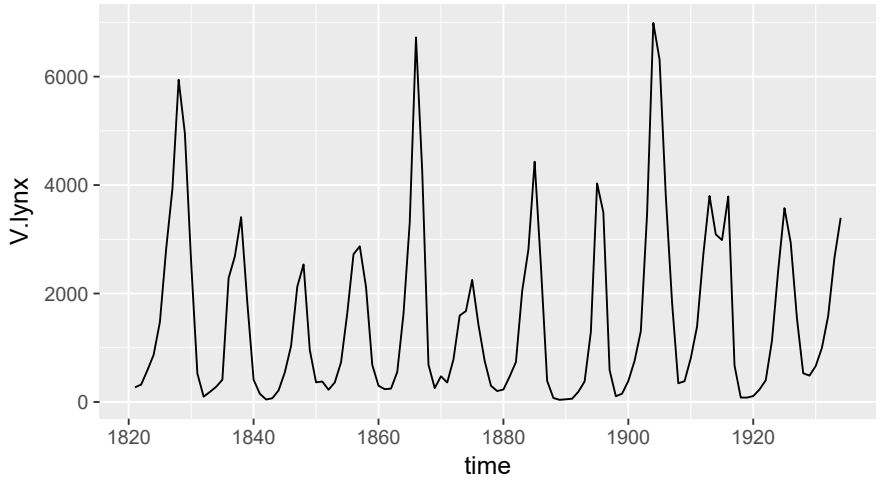
We exemplify this with some of the time series data included in R. In the first example we use the default format for time.

```
ggplot(try_tibble(austres),
       aes(time, V.austres)) +
  geom_line()
```

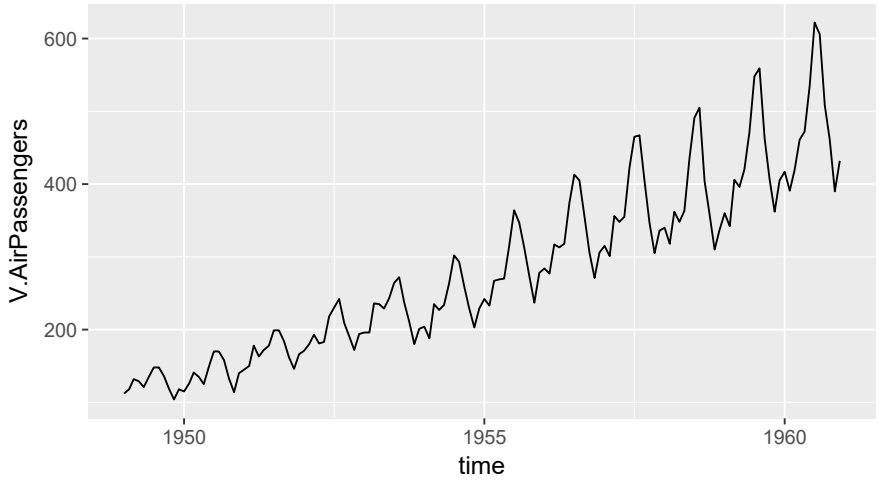


In the second example we use years in numeric format for expressing 'time'.

```
ggplot(try_tibble(lynx, "year", as.numeric = TRUE),
       aes(x = time, y = V.lynx)) +
  geom_line()
```




```
ggplot(try_tibble(AirPassengers, "month"),
       aes(time, V.AirPassengers)) +
  geom_line()
```



Multivariate time series are also supported.

7.13.2 Peaks and valleys

Peaks and valleys are local (or global) maxima and minima. These stats return the x and y values at the peaks or valleys plus suitable labels, and default aesthetics that make easy their use with several different geoms, including `geom_point`, `geom_text`, `geom_label`, `geom_vline`, `geom_hline` and `geom_rug`, and also with geoms defined by package 'ggrepel'. Some examples follow.

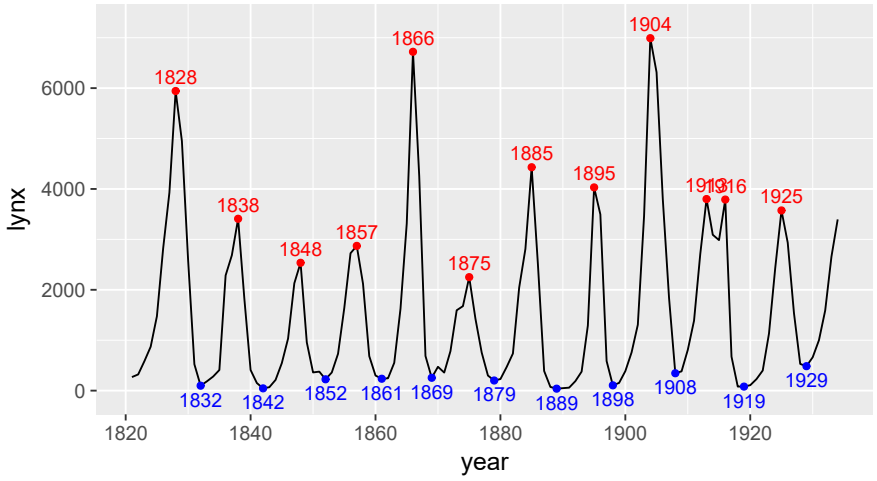
There are many cases, for example in physics and chemistry, but also when plotting time-series data when we need to automatically locate and label local maxima (peaks) or local minima (valleys) in curves. The statistics presented here are useful only for dense data as they do not fit a peak function but instead simply search for the local maxima or minima in the observed data. However, they allow flexible generation of labels on both x and y peak or valley coordinates.

We use as example the same time series as above. In the next several examples we demonstrate some of this flexibility.

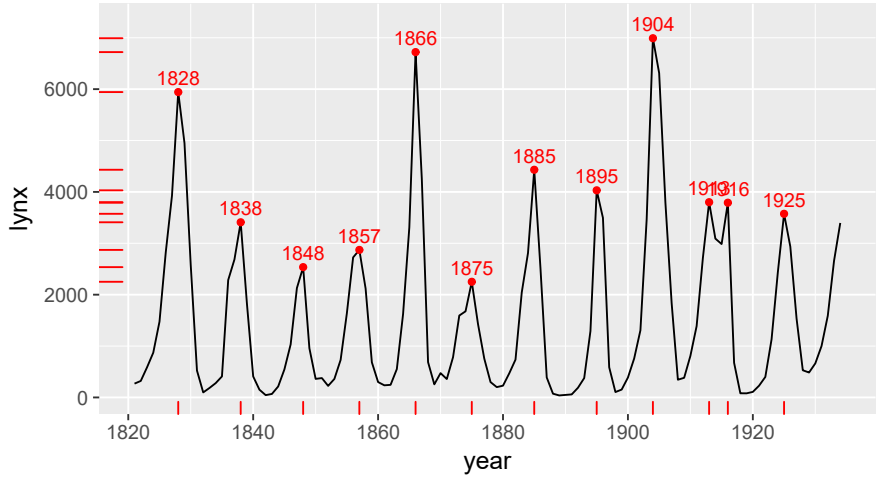
> As we are passing a `matrix` to the constructor, in this case, using the `tibble` constructor would fail.

```
lynx.df <- data.frame(year = as.numeric(time(lynx)), lynx = as.matrix(lynx))
```

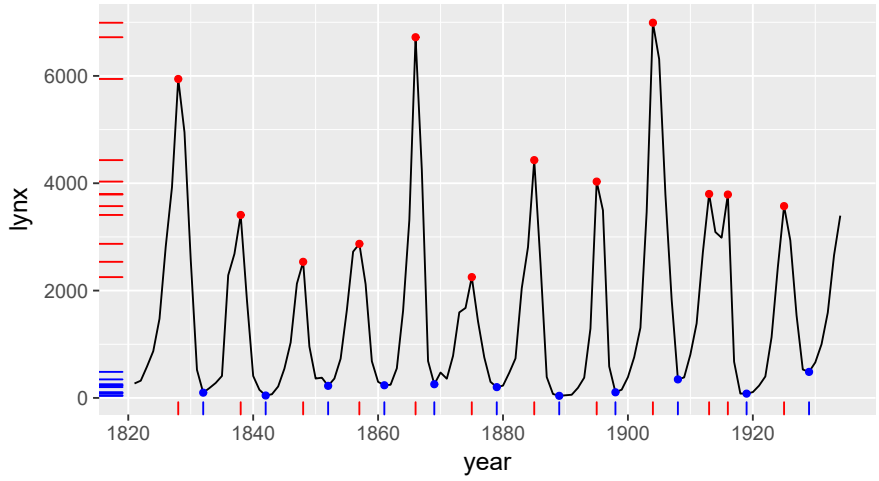
```
ggplot(lynx.df, aes(year, lynx)) + geom_line() +
  stat_peaks(colour = "red") +
  stat_peaks(geom = "text", colour = "red",
             vjust = -0.5, x.label.fmt = "%4.0f") +
  stat_valleys(colour = "blue") +
  stat_valleys(geom = "text", colour = "blue",
               vjust = 1.5, x.label.fmt = "%4.0f") +
  ylim(-100, 7300)
```




```
ggplot(lynx.df, aes(year, lynx)) + geom_line() +
  stat_peaks(colour = "red") +
  stat_peaks(geom = "rug", colour = "red") +
  stat_peaks(geom = "text", colour = "red",
             vjust = -0.5, x.label.fmt = "%4.0f") +
  ylim(NA, 7300)
```

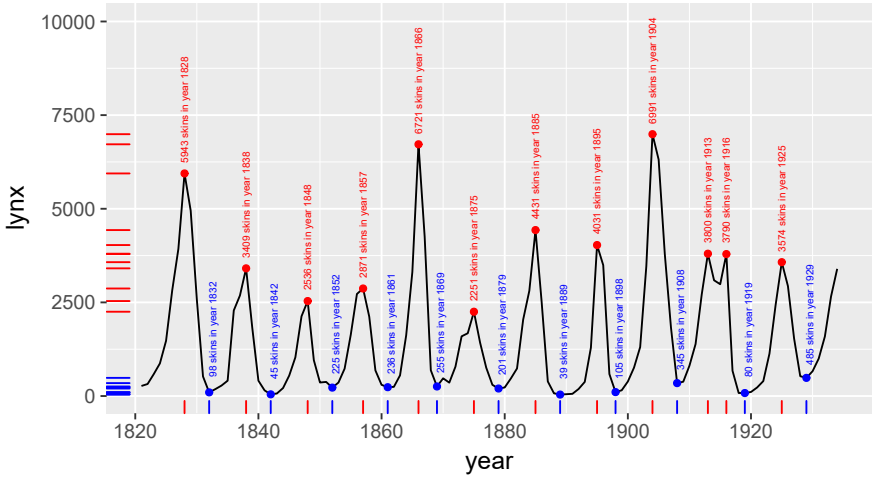



```
ggplot(lynx.df, aes(year, lynx)) + geom_line() +
  stat_peaks(colour = "red") +
  stat_peaks(geom = "rug", colour = "red") +
  stat_valleys(colour = "blue") +
  stat_valleys(geom = "rug", colour = "blue")
```




```
ggplot(lynx.df, aes(year, lynx)) + geom_line() +
  stat_peaks(colour = "red") +
  stat_peaks(geom = "rug", colour = "red") +
  stat_peaks(geom = "text", colour = "red",
             hjust = -0.1, label.fmt = "%4.0f",
```

```
            angle = 90, size = rel(2),
            aes(label = paste(..y.label..,
                              "skins in year", ..x.label..))) +
  stat_valleys(colour = "blue") +
  stat_valleys(geom = "rug", colour = "blue") +
  stat_valleys(geom = "text", colour = "blue",
            hjust = -0.1, vjust = 1, label.fmt = "%4.0f",
            angle = 90, size = rel(2),
            aes(label = paste(..y.label..,
                              "skins in year", ..x.label..))) +
  ylim(NA, 10000)
```



Of course, if one finds use for it, the peaks and/or valleys can be plotted on their own. Here we plot an "envelope" using `geom_line()`.

```
ggplot(lynx.df, aes(year, lynx)) +
  stat_peaks(geom = "line") + stat_valleys(geom = "line")
```

7.13.3 Equations as labels in plots

How to add a label with a polynomial equation including coefficient estimates from a model fit seems to be a frequently asked question in Stackoverflow. The parameter estimates are extracted automatically from a fit object corresponding to each *group* or panel in a plot and other aesthetics for the group respected. An aesthetic is provided for this, and only this. Such a statistics needs to be used together with another geom or stat like geom smooth to add the fitted line. A different approach, discussed in Stackoverflow, is to write a statistics that does both the plotting of the polynomial and adds the equation label. Package 'ggpmisc'defines `stat_poly_eq()` using the first approach which follows the 'rule' of using one function in the code for a single action. In this case there is a drawback that the users is responsible for ensuring that the model used for the label and the label are the same, and in addition that the same model is fitted twice to the data.

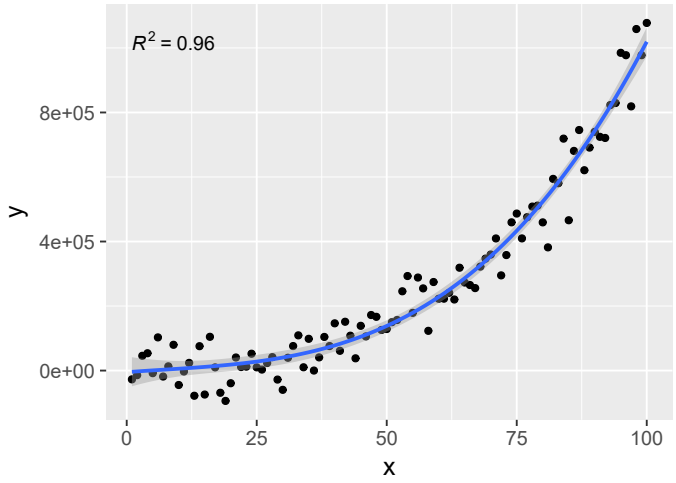
We first generate some artificial data.

```
set.seed(4321)
# generate artificial data
x <- 1:100
y <- (x + x^2 + x^3) +
  rnorm(length(x), mean = 0, sd = mean(x^3) / 4)
my.data <- tibble(x, y,
                  group = rep(c("A", "B"), 50),
                  y2 = y * c(0.5,2))
```

Linear models

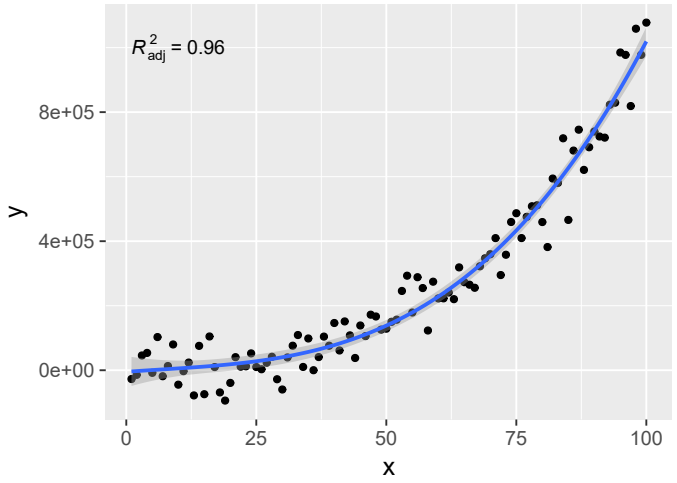
This section shows examples of linear models with one independent variables, including different polynomials. We first give an example using default arguments.

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(formula = formula, parse = TRUE)
```

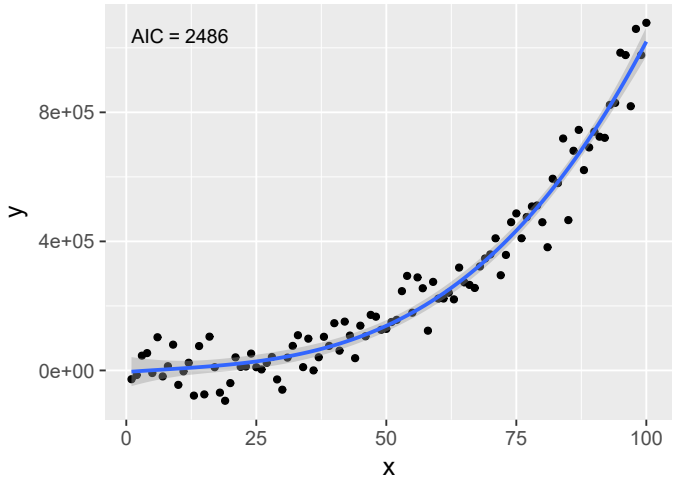


`stat_poly_eq()` makes available five different labels in the returned data frame. R^2, $R_mathrmadj^2$, AIC, BIC and the polynomial equation. R^2 is used by default, but `aes()` can be used to select a different one.

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..adj.rr.label..),
               formula = formula, parse = TRUE)
```

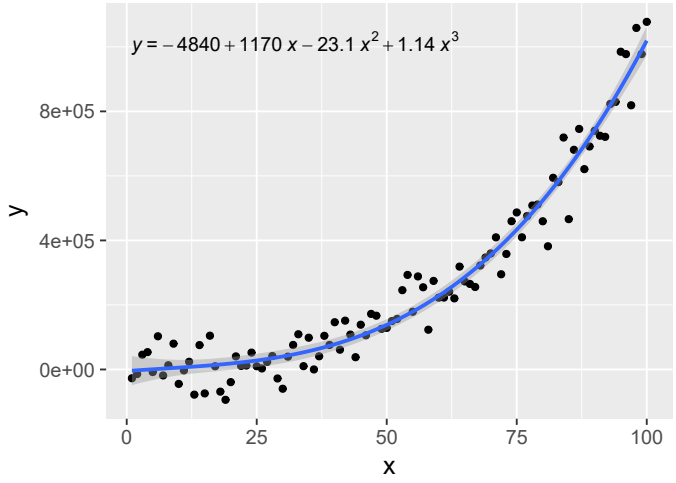



```
formula <- y ~ poly(x, 3, raw = TRUE)
  ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..AIC.label..),
               formula = formula, parse = TRUE)
```



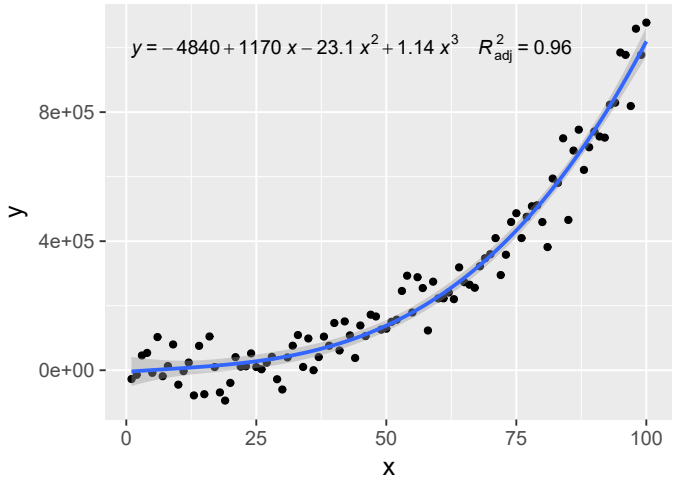

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
```

```
  stat_poly_eq(aes(label = ..eq.label..),
               formula = formula, parse = TRUE)
```

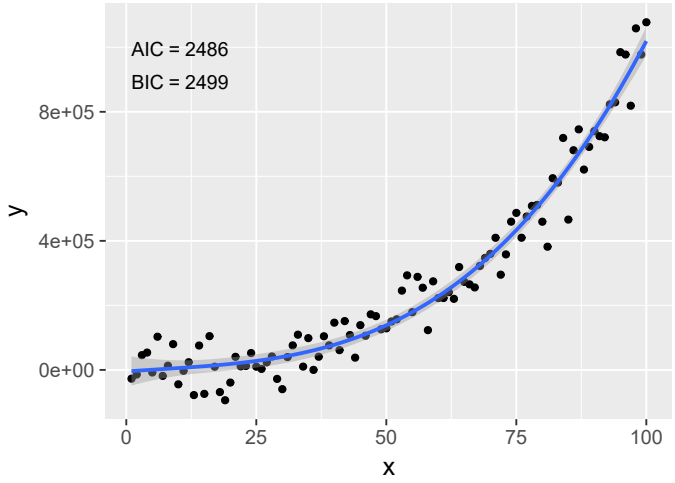


Within `aes()` it is possible to *compute* new labels based on those returned plus "arbitrary" text. The supplied labels are meant to be *parsed* into R expressions, so any text added should be valid for a string that will be parsed.

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = paste(..eq.label..,
                                 ..adj.rr.label..,
                                 sep = "~~~~")),
               formula = formula, parse = TRUE)
```



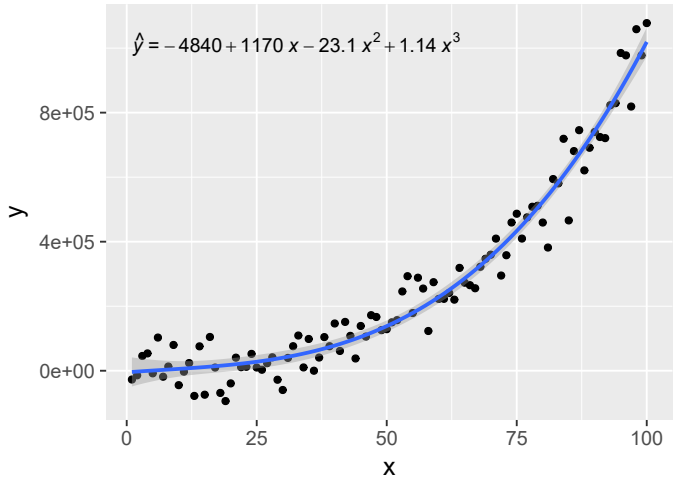

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = paste("atop(", ..AIC.label.., ",",
                                 ..BIC.label.., ")",

                                 sep = "")),
               formula = formula, parse = TRUE)
```



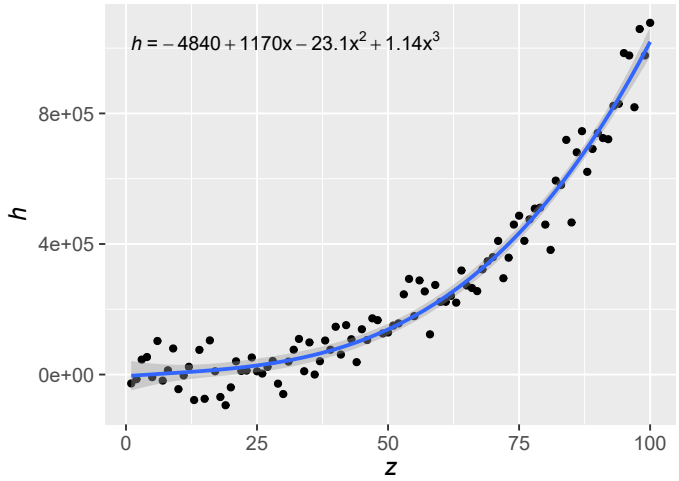
Two examples of removing or changing the *lhs* and/or the *rhs* of the equation. (Be aware that the equals sign must be always enclosed in back-

ticks in a string that will be parsed.)

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..eq.label..),
               eq.with.lhs = "italic(hat(y))~`=`~",
formula = formula, parse = TRUE)
```

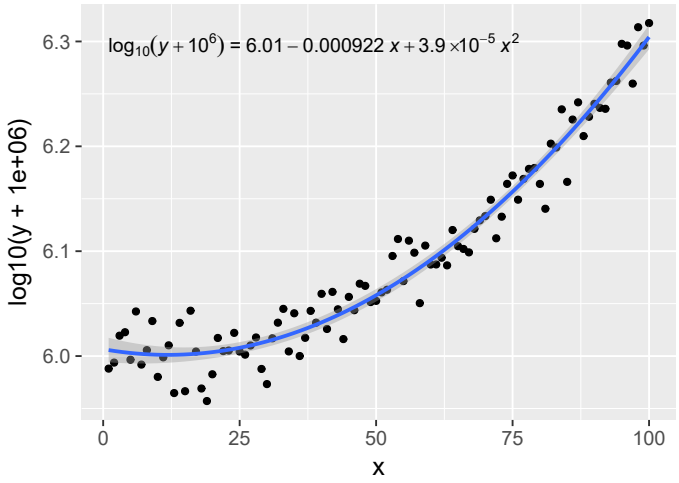



```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  labs(x = expression(italic(z)), y = expression(italic(h)) ) +
  stat_poly_eq(aes(label = ..eq.label..),
               eq.with.lhs = "italic(h)~`=`~",
               eq.x.rhs = "~italic(z)",
               formula = formula, parse = TRUE)
```



As any valid R expression can be used, Greek letters are also supported, as well as the inclusion in the label of variable transformations used in the model formula.

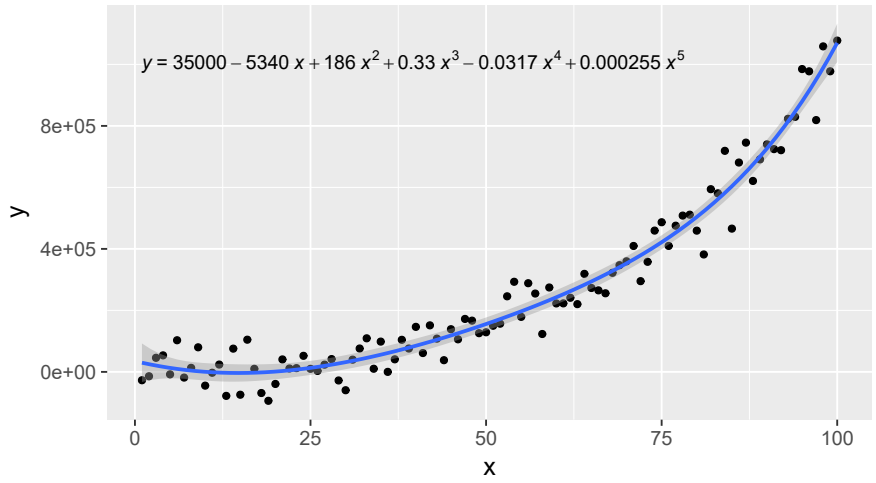
```
formula <- y ~ poly(x, 2, raw = TRUE)
ggplot(my.data, aes(x, log10(y + 1e6))) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..eq.label..),
               eq.with.lhs = "plain(log)[10](italic(y)+10^6)~`=`~",
               formula = formula, parse = TRUE)
```



Example of a polynomial of fifth order.

```
opts_chunk$set(opts_fig_wide)
```

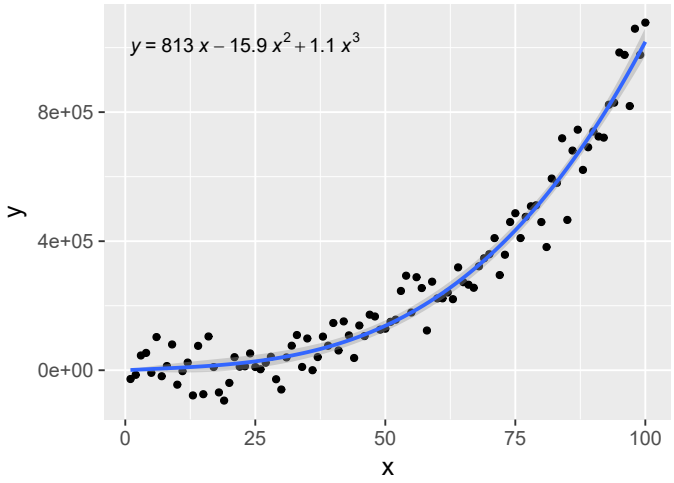
```
formula <- y ~ poly(x, 5, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..eq.label..),
               formula = formula, parse = TRUE)
```




```
opts_chunk$set(opts_fig_narrow)
```

Intercept forced to zero—line through the origin.

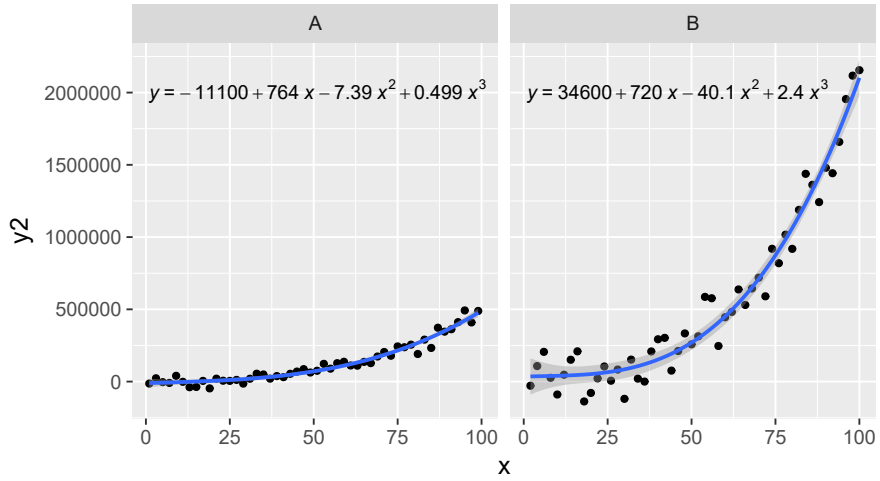
```
formula <- y ~ x + I(x^2) + I(x^3) - 1
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..eq.label..),
               formula = formula, parse = TRUE)
```



We give some additional examples to demonstrate how other components of the `ggplot` object affect the behaviour of this statistic.

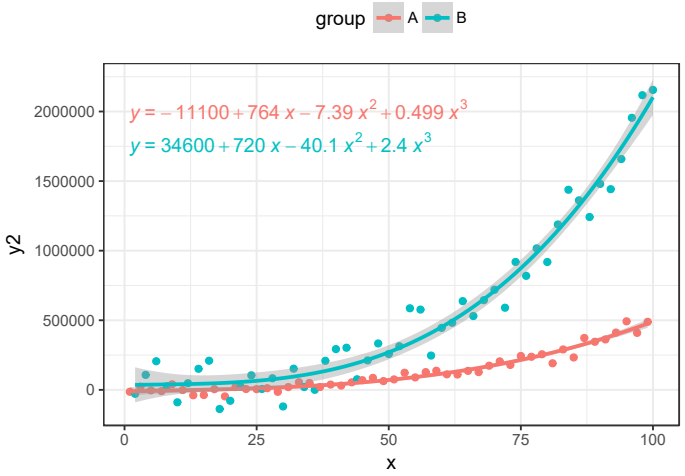
Facets work as expected either with fixed or free scales. Although bellow we had to adjust the size of the font used for the equation.

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y2)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..eq.label..), # size = 2.8,
               formula = formula, parse = TRUE) +
  facet_wrap(~group)
```



Grouping, in this example using colour aesthetic also works as expected.

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y2, colour = group)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_poly_eq(aes(label = ..eq.label..),
               formula = formula, parse = TRUE) +
  theme_bw() +
  theme(legend.position = "top")
```



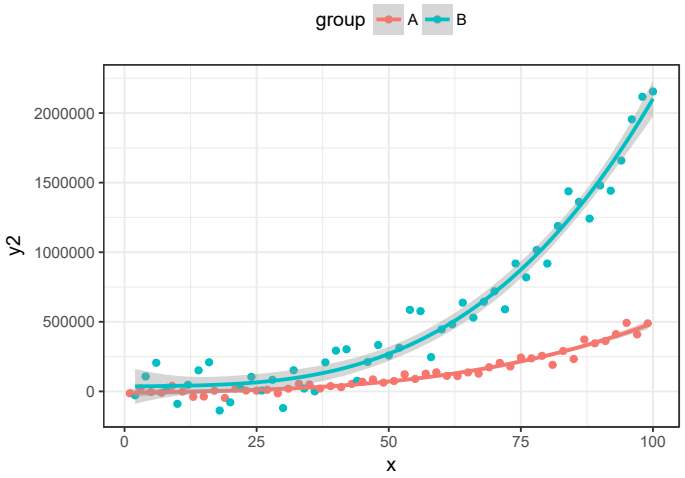
Other types of models

Another statistic, `stat_fit_glance()` allows lots of flexibility, but at the moment there is no equivalently flexible version of `stat_smooth()`.

We give an example with a linear model, showing a P-value (a frequent request for which I do not find much use).

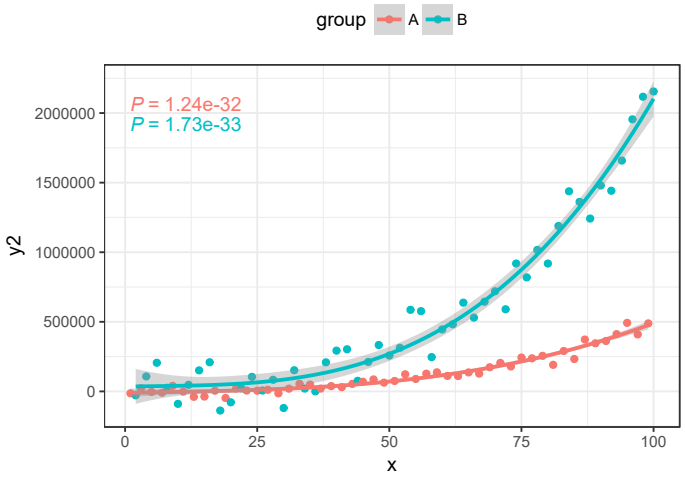
We use `geom_debug()` to find out what values `stat_glance()` returns for our linear model, and add labels with P-values for the fits.

```
formula <- y ~ x + I(x^2) + I(x^3)
ggplot(my.data, aes(x, y2, colour = group)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_fit_glance(method.args = list(formula = formula),
                  geom = "debug",
                  summary.fun = print,
                  summary.fun.args = list()) +
  theme_bw() +
  theme(legend.position = "top")

## Input 'data' to 'geom_debug()':

##    colour hjust vjust r.squared adj.r.squared
## 1 #F8766D     0   1.4 0.9619032     0.9594187
## 2 #00BFC4     0   2.8 0.9650270     0.9627461
##       sigma statistic      p.value df    logLik
## 1  29045.57  387.1505 1.237801e-32  4 -582.6934
## 2 118993.86  423.0996 1.732868e-33  4 -653.2037
##        AIC      BIC     deviance df.residual x
## 1 1175.387 1184.947  38807664340          46 1
## 2 1316.407 1325.968 651338752799          46 1
##         y PANEL group
## 1 2154937     1     1
## 2 2154937     1     2
##    colour hjust vjust r.squared adj.r.squared
## 1 #F8766D     0   1.4 0.9619032     0.9594187
## 2 #00BFC4     0   2.8 0.9650270     0.9627461
##       sigma statistic      p.value df    logLik
## 1  29045.57  387.1505 1.237801e-32  4 -582.6934
## 2 118993.86  423.0996 1.732868e-33  4 -653.2037
##        AIC      BIC     deviance df.residual x
## 1 1175.387 1184.947  38807664340          46 1
## 2 1316.407 1325.968 651338752799          46 1
##         y PANEL group
## 1 2154937     1     1
## 2 2154937     1     2
```



Using the information now at hand we create some labels.

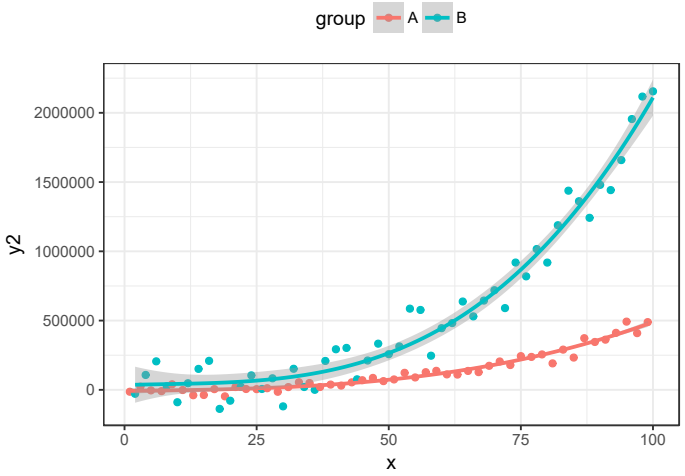
```
formula <- y ~ x + I(x^2) + I(x^3)
ggplot(my.data, aes(x, y2, colour = group)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
        stat_fit_glance(aes(label    =    paste('italic(P)~`=`~',    sig-
nif(..p.value.., 3)), sep = ""),
                  parse = TRUE,
                  method.args = list(formula = formula),
                  geom = "text") +
  theme_bw() +
  theme(legend.position = "top")

## Warning: Ignoring unknown aesthetics: sep
```



We use `geom_debug()` to find out what values `stat_glance()` returns for our resistant linear model fitted with `MASS:rlm()`.

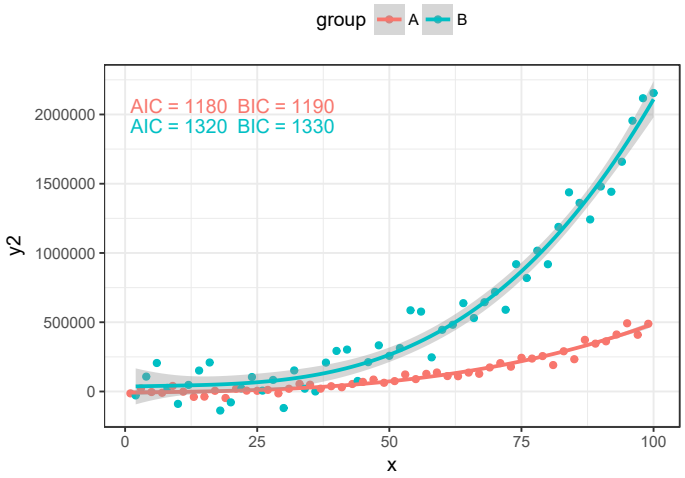
```
formula <- y ~ x + I(x^2) + I(x^3)
ggplot(my.data, aes(x, y2, colour = group)) +
  geom_point() +
  geom_smooth(method = "rlm", formula = formula) +
  stat_fit_glance(method.args = list(formula = formula),
                  geom = "debug",
                  method = "rlm",
                  summary.fun = print,
                  summary.fun.args = list()) +
  theme_bw() +
  theme(legend.position = "top")

## Input 'data' to 'geom_debug()':

##    colour hjust vjust     sigma converged
## 1 #F8766D     0   1.4  20078.62      TRUE
## 2 #00BFC4     0   2.8 126111.74      TRUE
##       logLik      AIC      BIC     deviance x
## 1 -582.8362 1175.672 1185.232  39029842201 1
## 2 -653.2392 1316.478 1326.039 652263183741 1
##         y PANEL group
## 1 2154937     1     1
## 2 2154937     1     2
##    colour hjust vjust     sigma converged
## 1 #F8766D     0   1.4  20078.62      TRUE
## 2 #00BFC4     0   2.8 126111.74      TRUE
##       logLik      AIC      BIC     deviance x
## 1 -582.8362 1175.672 1185.232  39029842201 1
## 2 -653.2392 1316.478 1326.039 652263183741 1
```

```
##         y PANEL group
## 1 2154937     1     1
## 2 2154937     1     2
```



Using the information now at hand we create some labels.

```
formula <- y ~ x + I(x^2) + I(x^3)
ggplot(my.data, aes(x, y2, colour = group)) +
  geom_point() +
  geom_smooth(method = "rlm", formula = formula) +
  stat_fit_glance(aes(label = paste('AIC~`=`~', signif(..AIC.., 3),
                      "~~", 'BIC~`=`~', signif(..BIC.., 3), sep = "")),
                  parse = TRUE,
                  method = "rlm",
                  method.args = list(formula = formula),
                  geom = "text") +
  theme_bw() +
  theme(legend.position = "top")
```

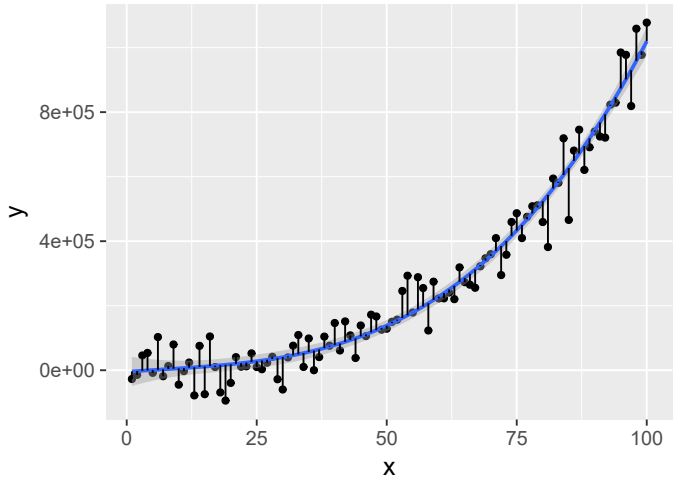


In a similar way one can generate labels for any fit supported by package 'broom'.

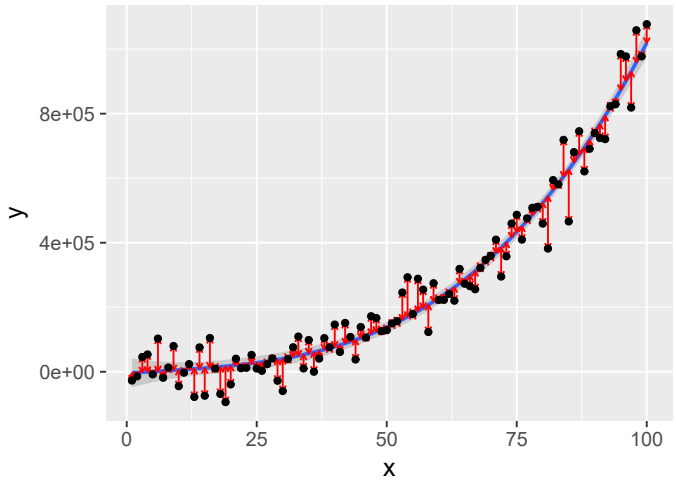
7.13.4 Highlighting deviations from fitted line

First an example using default arguments.

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = formula) +
  stat_fit_deviations(formula = formula)
```

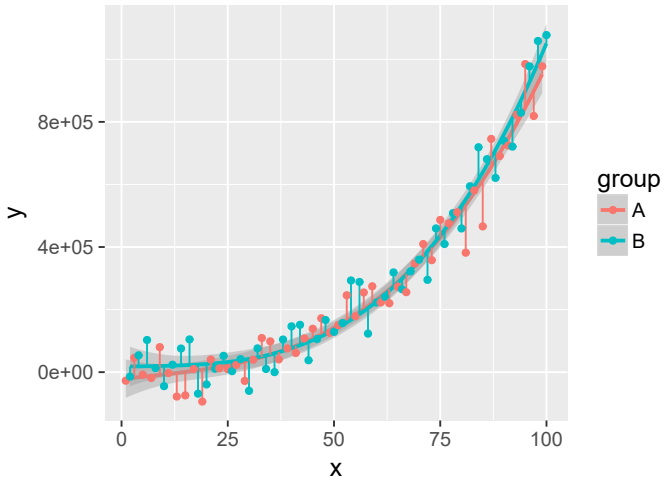



```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y)) +
  geom_smooth(method = "lm", formula = formula) +
  stat_fit_deviations(formula = formula, color = "red",
                      arrow = arrow(length = unit(0.015, "npc"),
                                    ends = "both")) +
  geom_point()
```



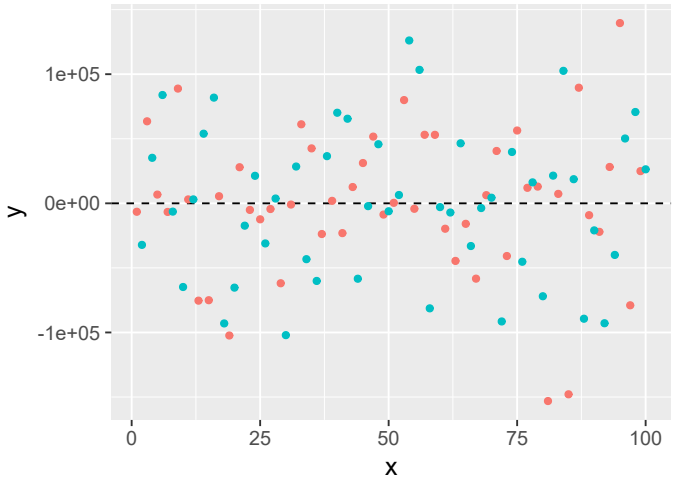

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y, colour = group)) +
  geom_smooth(method = "lm", formula = formula) +
```

```
  stat_fit_deviations(formula = formula) +
  geom_point()
```



7.13.5 Plotting residuals from linear fit

```
formula <- y ~ poly(x, 3, raw = TRUE)
ggplot(my.data, aes(x, y, colour = group)) +
  geom_hline(yintercept = 0, linetype = "dashed") +
  stat_fit_residuals(formula = formula)
```



7.13.6 Filtering observations based on local density

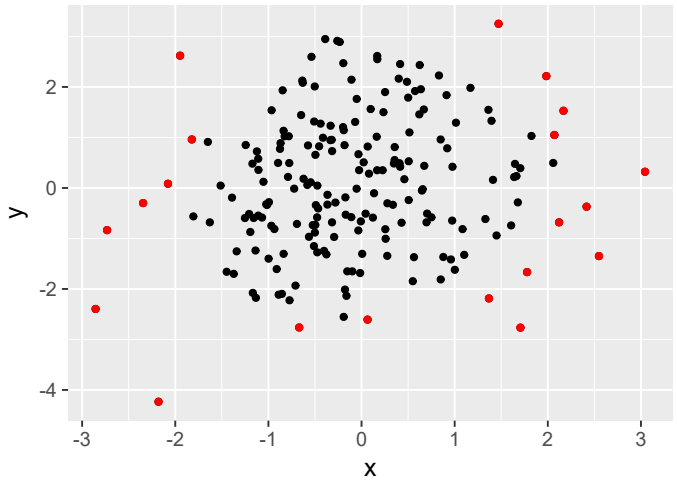
Statistics `stat_dens2d_filter` works best with clouds of observations, so we generate some random data.

```
set.seed(1234)
nrow <- 200
my.2d.data <- tibble(
    x = rnorm(nrow),
    y = rnorm(nrow) + rep(c(-1, +1), rep(nrow / 2, 2)),
    group = rep(c("A", "B"), rep(nrow / 2, 2))
   )
```

In most recipes in the section we use `stat_dens2d_filter` to highlight observations with the `color` aesthetic. Other aesthetics can also be used.

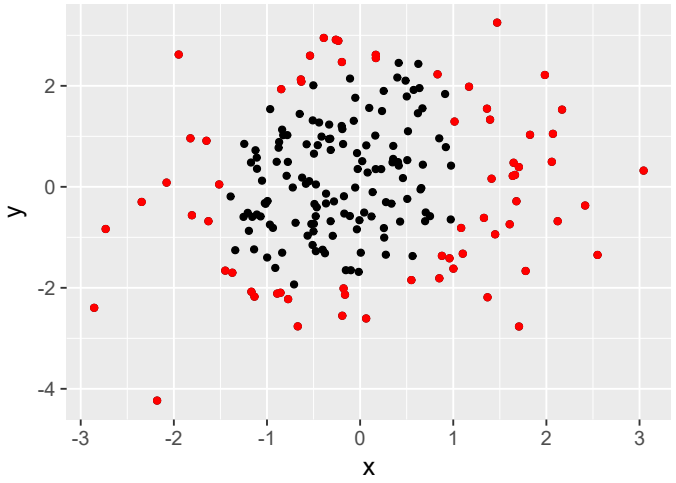
By default 1/10 of the observations are kept from regions of lowest density.

```
ggplot(my.2d.data, aes(x, y)) +
  geom_point() +
  stat_dens2d_filter(color = "red")
```



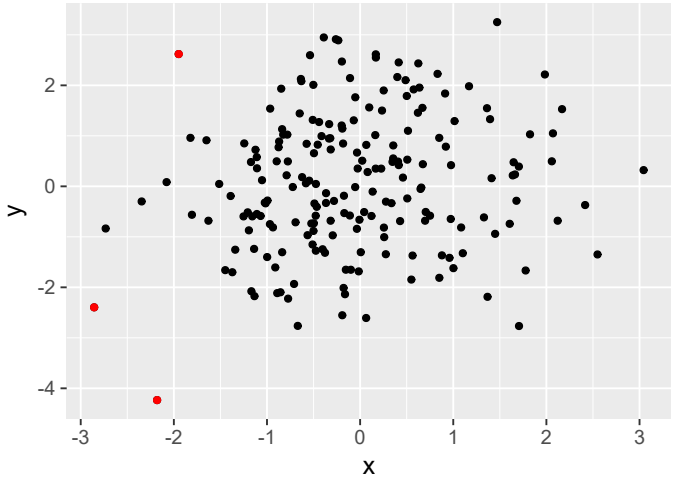
Here we change the fraction to 1/3.

```
ggplot(my.2d.data, aes(x, y)) +
  geom_point() +
  stat_dens2d_filter(color = "red",
                     keep.fraction = 1/3)
```



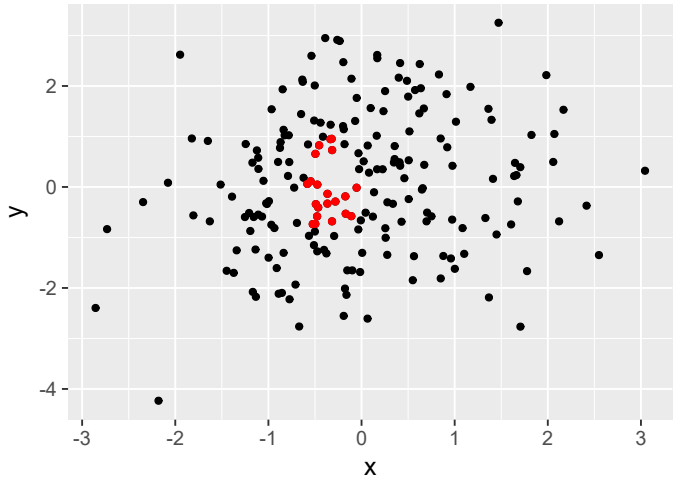
We can also set a maximum number of observations to keep.

```
ggplot(my.2d.data, aes(x, y)) +
  geom_point() +
  stat_dens2d_filter(color = "red",
                     keep.number = 3)
```

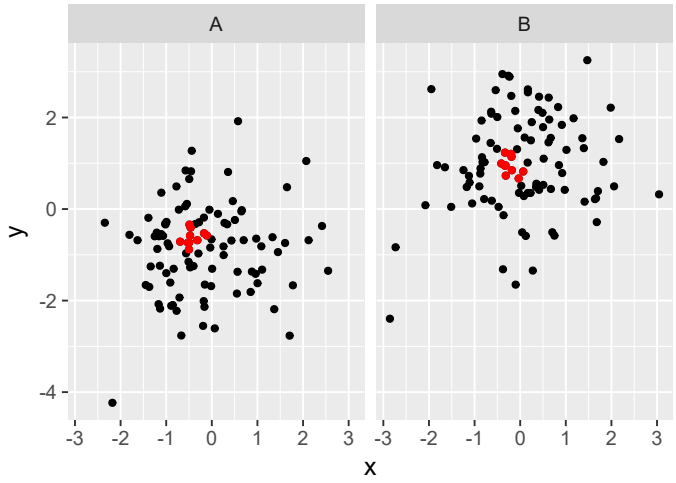


We can also keep the observations from the densest areas instead of the from the sparsest.

```
ggplot(my.2d.data, aes(x, y)) +
  geom_point() +
  stat_dens2d_filter(color = "red",
                     keep.sparse = FALSE)
```

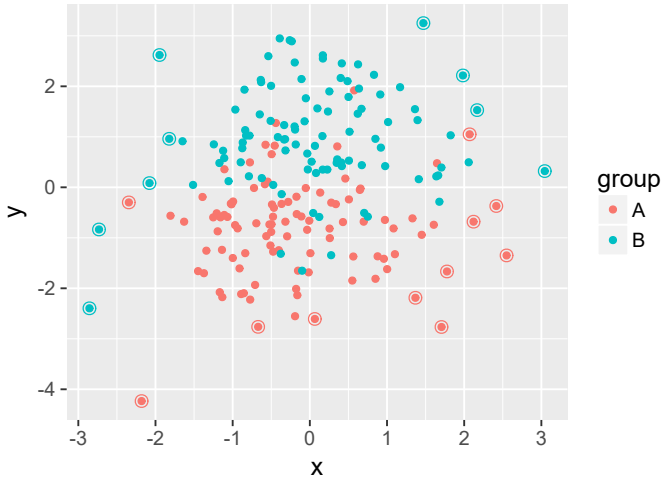



```
ggplot(my.2d.data, aes(x, y)) +
  geom_point() +
  stat_dens2d_filter(color = "red",
                     keep.sparse = FALSE) +
  facet_grid(~group)
```

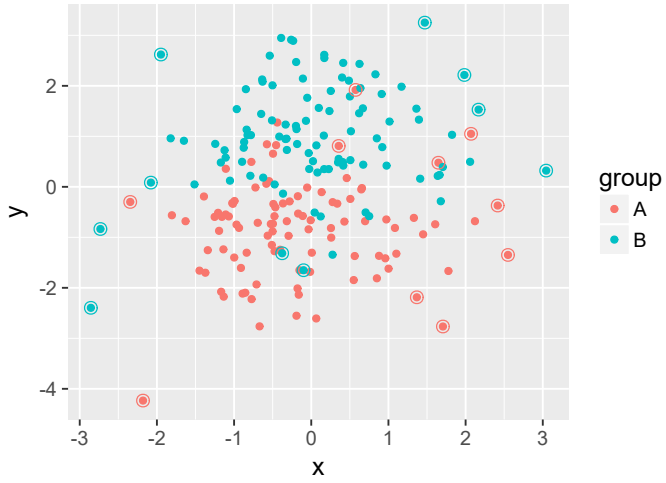


In addition to `stat_dens2d_filter` there is `stat_dens2d_filter_g`. The difference is in that the first one computes the density on a plot-panel basis while the second one does it on a group basis. This makes a difference only when observations are grouped based on another aesthetic within each panel.

```
ggplot(my.2d.data, aes(x, y, color = group)) +
  geom_point() +
  stat_dens2d_filter(shape = 1, size = 3)
```




```
ggplot(my.2d.data, aes(x, y, color = group)) +
  geom_point() +
  stat_dens2d_filter_g(shape = 1, size = 3)
```



A related stat `stat_dens2d_label`, also defined in package 'ggpmisc'is described in section 7.14.2 on page 239.

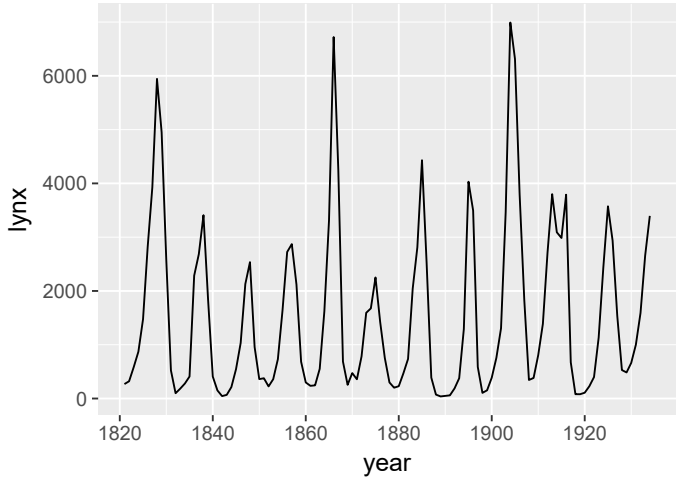
7.13.7 Learning and/or debugging

A very simple stat named `stat_debug()` can save the work of adding print statements to the code of stats to get information about what data is being passed to the `compute_group()` function. Because the code of this function is stored in a `ggproto` object, at the moment it is impossible to directly set breakpoints in it. This `stat_debug()` may also help users diagnose problems with the mapping of aesthetics in their code or just get a better idea of how the internals of 'ggplot2'work.

```
ggplot(lynx.df, aes(year, lynx)) + geom_line() +
  stat_debug_group()
```

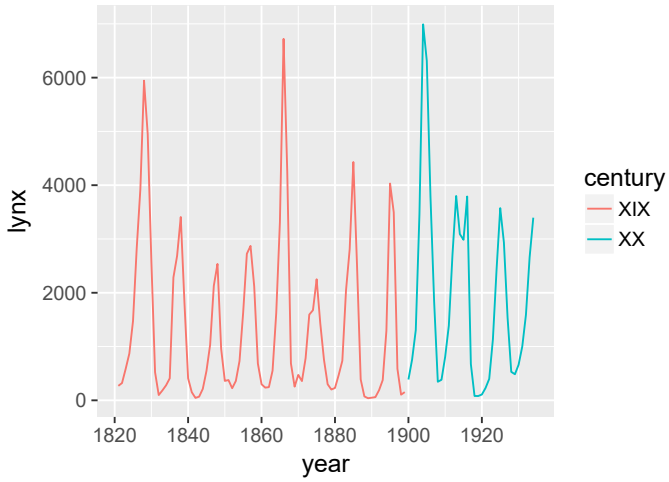
```
## [1] "Input 'data' to 'compute_group()':"
## # A tibble: 114 □ 4
##        x     y PANEL group
## *  <dbl> <dbl> <int> <int>
## 1   1821   269     1    -1
## 2   1822   321     1    -1
## 3   1823   585     1    -1
## 4   1824   871     1    -1
## 5   1825  1475     1    -1
## 6   1826  2821     1    -1
## 7   1827  3928     1    -1
## 8   1828  5943     1    -1
```

```
## 9   1829  4950     1    -1
## 10  1830  2577     1    -1
## # ... with 104 more rows
```



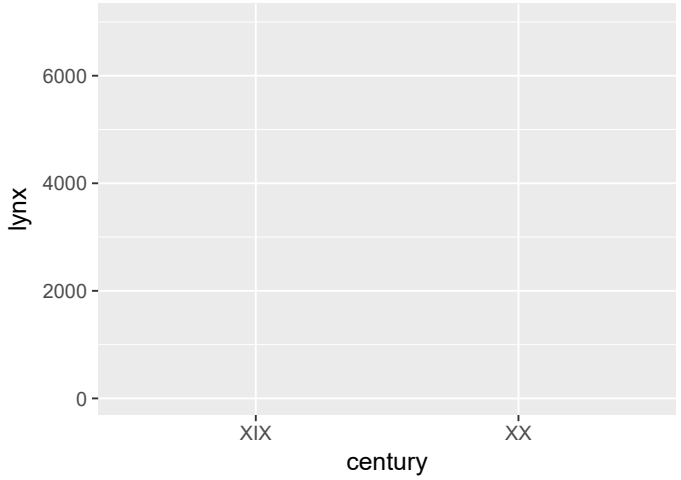

```
lynx.df$century <- ifelse(lynx.df$year >= 1900, "XX", "XIX")
ggplot(lynx.df, aes(year, lynx, color = century)) +
  geom_line() +
  stat_debug_group()

## [1] "Input 'data' to 'compute_group()':"
## # A tibble: 79 □ 5
##        x     y colour PANEL group
## *  <dbl> <dbl>  <chr> <int> <int>
## 1   1821   269    XIX     1     1
## 2   1822   321    XIX     1     1
## 3   1823   585    XIX     1     1
## 4   1824   871    XIX     1     1
## 5   1825  1475    XIX     1     1
## 6   1826  2821    XIX     1     1
## 7   1827  3928    XIX     1     1
## 8   1828  5943    XIX     1     1
## 9   1829  4950    XIX     1     1
## 10  1830  2577    XIX     1     1
## # ... with 69 more rows
## [1] "Input 'data' to 'compute_group()':"
## # A tibble: 35 □ 5
##        x     y colour PANEL group
## *  <dbl> <dbl>  <chr> <int> <int>
## 1   1900   387     XX     1     2
## 2   1901   758     XX     1     2
## 3   1902  1307     XX     1     2
```

```
## 4   1903  3465     XX     1     2
## 5   1904  6991     XX     1     2
## 6   1905  6313     XX     1     2
## 7   1906  3794     XX     1     2
## 8   1907  1836     XX     1     2
## 9   1908   345     XX     1     2
## 10  1909   382     XX     1     2
## # ... with 25 more rows
```



By means of `geom_debug` it is possible to ”print” to the console the data returned by a ggplot statistic.

```
ggplot(lynx.df, aes(century, lynx)) +
  geom_blank() +
  stat_summary(fun.y = median,
               geom = "debug", summary.fun = head, summary.fun.args = list())

## Input 'data' to 'geom_debug()':

##   x group ymin    y ymax PANEL
## 1 1     1   NA  684   NA     1
## 2 2     2   NA 1388   NA     1
```



7.14 'ggrepel'

Package 'ggrepel'is under development by Kamil Slowikowski. It does a single thing, relocates text labels so that they do not overlap. This is achieved through two geometries that work similarly to those provided by 'ggplot2'except for the relocation. This is incredibly useful both when labeling peaks and valleys and when labeling points in scatter-plots. This is a significant problem in bioinformatics plots and in maps.

7.14.1 New geoms

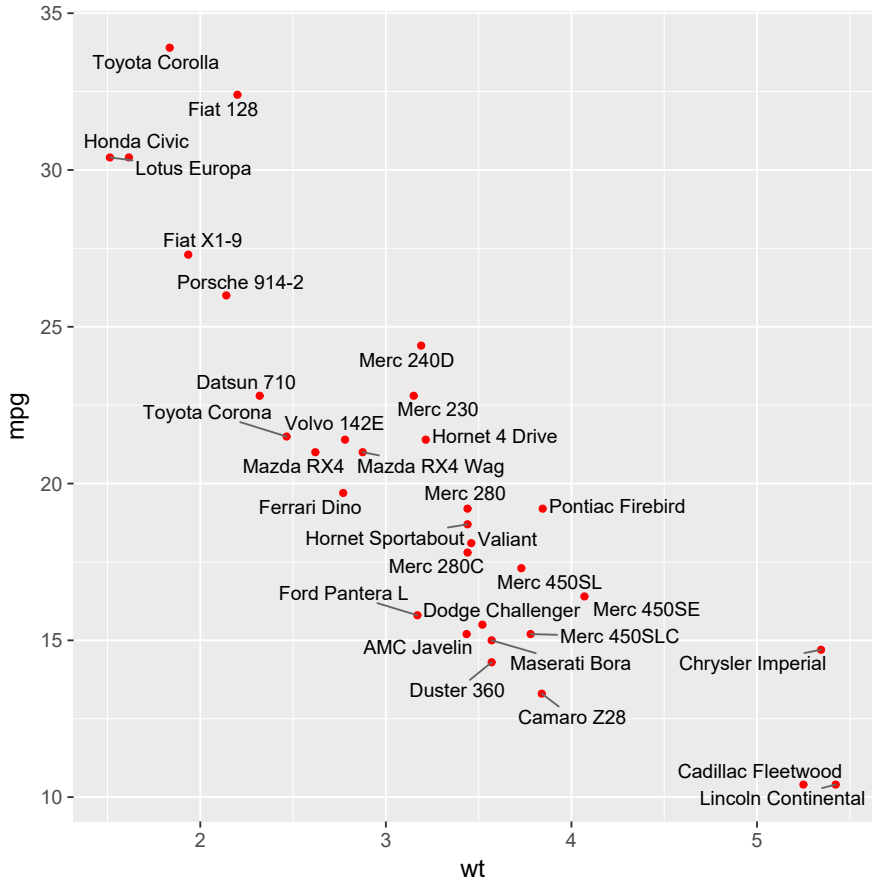
Package 'ggrepel'provides two new geoms: `geom_text_repel` and `geom_label_repel`. They are used similarly to `geom_text` and `geom_label` but the text or labels "repel" each other so that they rarely overlap unless the plot is very crowded. The vignette *ggrepel Usage Examples* provides very nice examples of the power and flexibility of these geoms. The algorithm used for avoiding overlaps through repulsion is iterative, and can be slow when the number of labels or observations are in the thousands.

I reproduce here some simple examples from the 'ggrepel'vignette.

```
opts_chunk$set(opts_fig_wide_square)
```

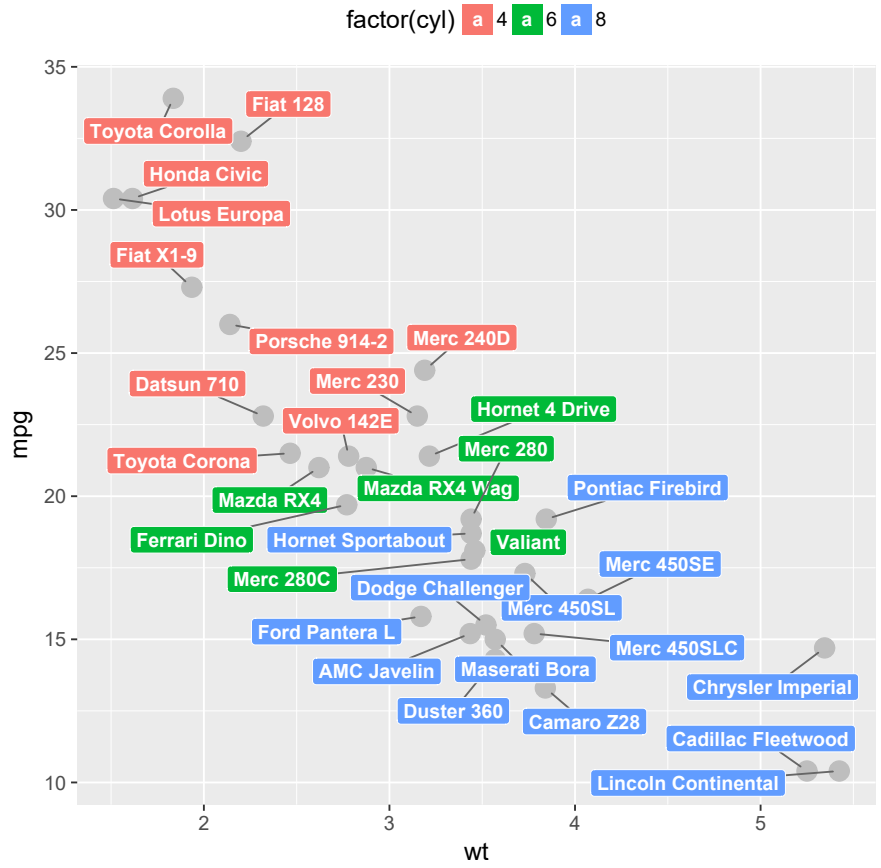
Just using defaults, we avoid overlaps among text items on the plot. `geom_text_repel` has some parameters matching those in `geom_text`, but those related to manual positioning are missing except for `angle`. Several new parameters control both the appearance of text and the function of the repulsion algorithm.

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(color = 'red') +
  geom_text_repel(aes(label = rownames(mtcars)))
```



The chunk below shows how to change the appearance of labels. `geom_label_repel` is comparable to `geom_label`, but with repulsion.

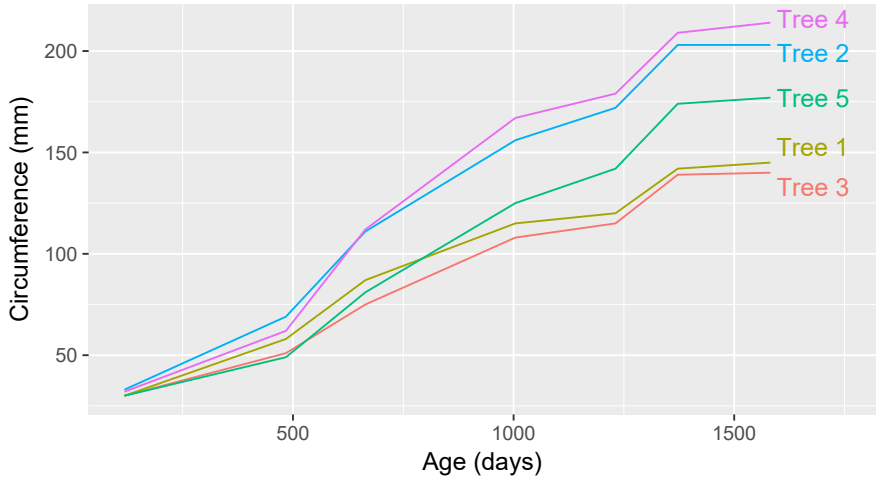
```
set.seed(42)
ggplot(mtcars) +
  geom_point(aes(wt, mpg), size = 5, color = 'grey') +
  geom_label_repel(
  aes(wt, mpg, fill = factor(cyl), label = rownames(mtcars)),
      fontface = 'bold', color = 'white',
      box.padding = unit(0.25, "lines"),
      point.padding = unit(0.5, "lines")) +
  theme(legend.position = "top")
```



The parameters `nudge_x` and `nudge_y` allow strengthening or weakening the repulsion force, or favouring a certain direction. We also need to expand the x-axis high limit to make space for the labels.

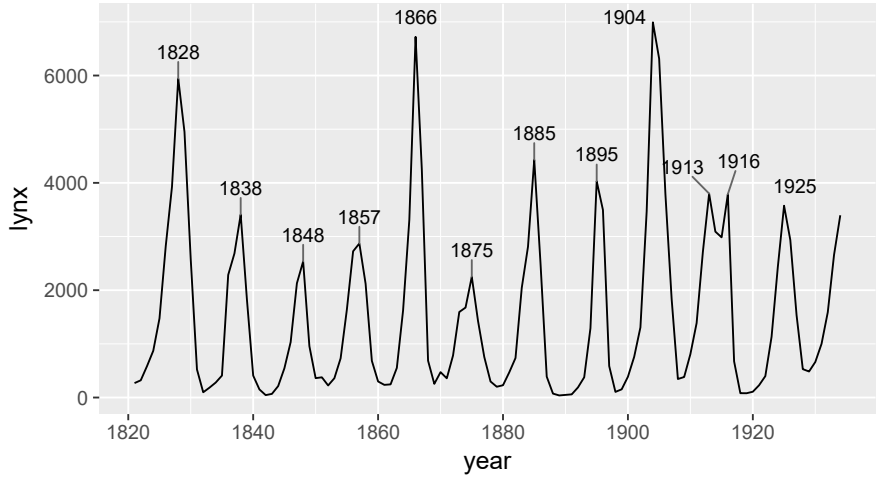
```
opts_chunk$set(opts_fig_wide)
```

```
set.seed(42)
ggplot(Orange, aes(age, circumference, color = Tree)) +
  geom_line() +
  expand_limits(x = max(Orange$age) * 1.1) +
  geom_text_repel(data = subset(Orange, age == max(age)),
                  aes(label = paste("Tree", Tree)),
                  size = 5,
                  nudge_x = 65,
                  segment.color = NA) +
  theme(legend.position = "none") +
  labs(x = "Age (days)", y = "Circumference (mm)")
```



We can combine `stat_peaks` from package 'ggpmisc'with the use of repulsive text to avoid overlaps between text items. We use `nudge_y = 500` to push the text upwards.

```
ggplot(lynx.df, aes(year, lynx)) +
  geom_line() +
  stat_peaks(geom = "text_repel", nudge_y = 500)
```



7.14.2 Selectively plotting repulsive labels

To repel text or labels so that they do not overlap unlabelled observations, one can set the labels to an empty character string `""`. Setting labels to `NA` skips the observation completely, as is the usual behavior in 'ggplot2'2 geoms, and can result in text or labels overlapping those observations. Labels can be set manually to `""`, but in those cases where all observations have labels in the data, but we would like to plot only those in low density regions, this can be automated. Geoms `geom_text_repel` and `geom_label_repel` from package 'ggrepel'can be used together with `stat_dens2d_label` from package 'ggpmisc'.

To demonstrate this we first generate suitable data and labels.

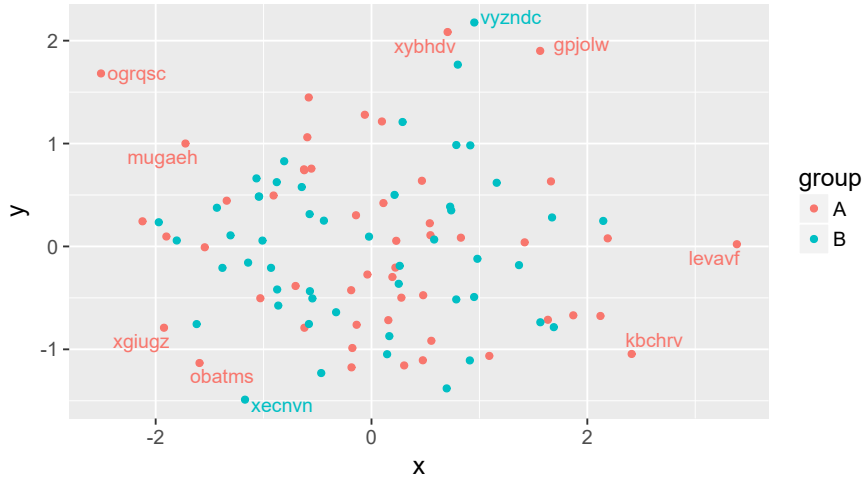
```
# Make random labels
random_string <- function(len = 6) {
paste(sample(letters, len, replace = TRUE), collapse = "")
}
```

```
# Make random data.
set.seed(1001)
myl.data <- tibble(
  x = rnorm(100),
  y = rnorm(100),
  group = rep(c("A", "B"), c(50, 50)),
  lab = replicate(100, { random_string() })
)
```

```
head(myl.data)

## # A tibble: 6 □ 4
##            x           y group    lab
##        <dbl>       <dbl> <chr>  <chr>
## 1  2.1886481  0.07862339     A emhufi
## 2 -0.1775473 -0.98708727     A yrvrlo
## 3 -0.1852753 -1.17523226     A wrpfpp
## 4 -2.5065362  1.68140888     A ogrqsc
## 5 -0.5573113  0.75623228     A wfxezk
## 6 -0.1435595  0.30309733     A zjccnn
```

The first example uses defaults.

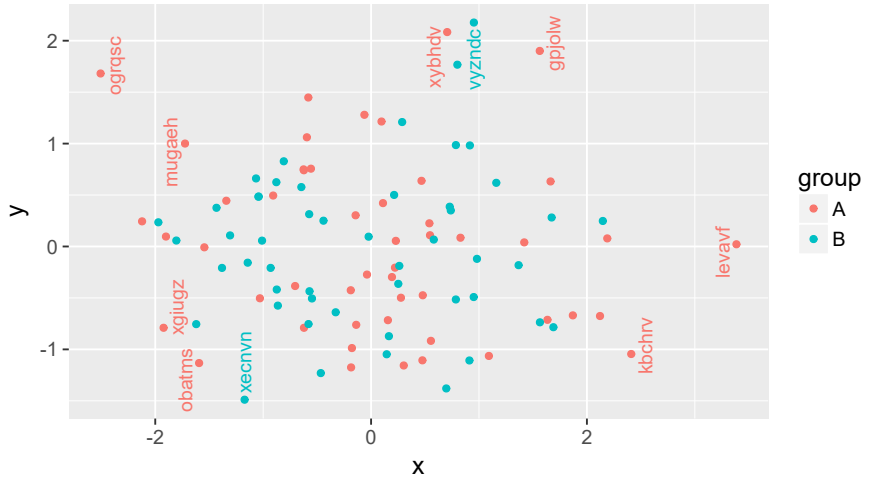
```
ggplot(data = myl.data, aes(x, y, label = lab, color = group)) +
  geom_point() +
  stat_dens2d_labels(geom = "text_repel")
```



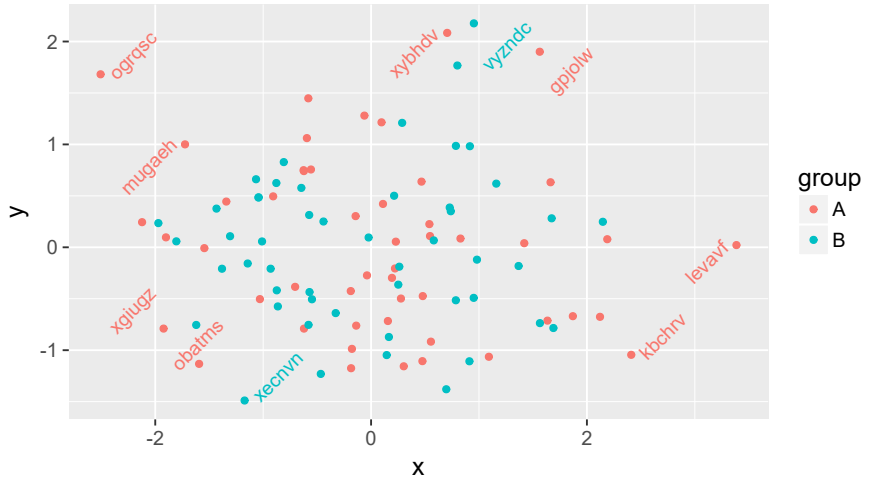
The fraction of observations can be plotted, as well as the maximum number can be both set through parameters, as shown in section 7.13.6 on page 228.

Something to be aware of when rotating labels is that repulsion is always based on bounding box that does not rotate, which for long labels and angles that are not multiples of 90 degrees, reserves too much space and leaves gaps between segments and text. Compare the next two figures.

```
ggplot(data = myl.data, aes(x, y, label = lab, color = group)) +
  geom_point() +
  stat_dens2d_labels(geom = "text_repel", angle = 90)
```

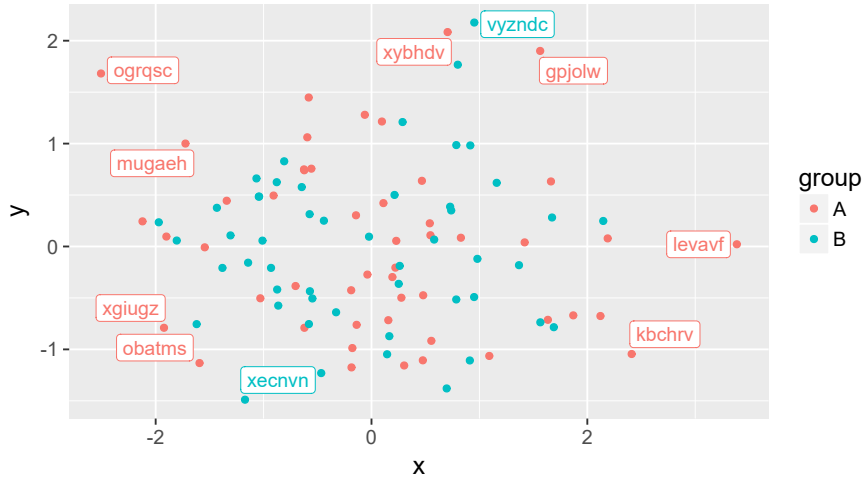



```
ggplot(data = myl.data, aes(x, y, label = lab, color = group)) +
  geom_point() +
  stat_dens2d_labels(geom = "text_repel", angle = 45)
```



Labels cannot be rotated.

```
ggplot(data = myl.data, aes(x, y, label = lab, color = group)) +
  geom_point() +
  stat_dens2d_labels(geom = "label_repel")
```



7.15 'ggsci'

coming soon.

7.16 'ggthemes'

coming soon.

7.17 Examples

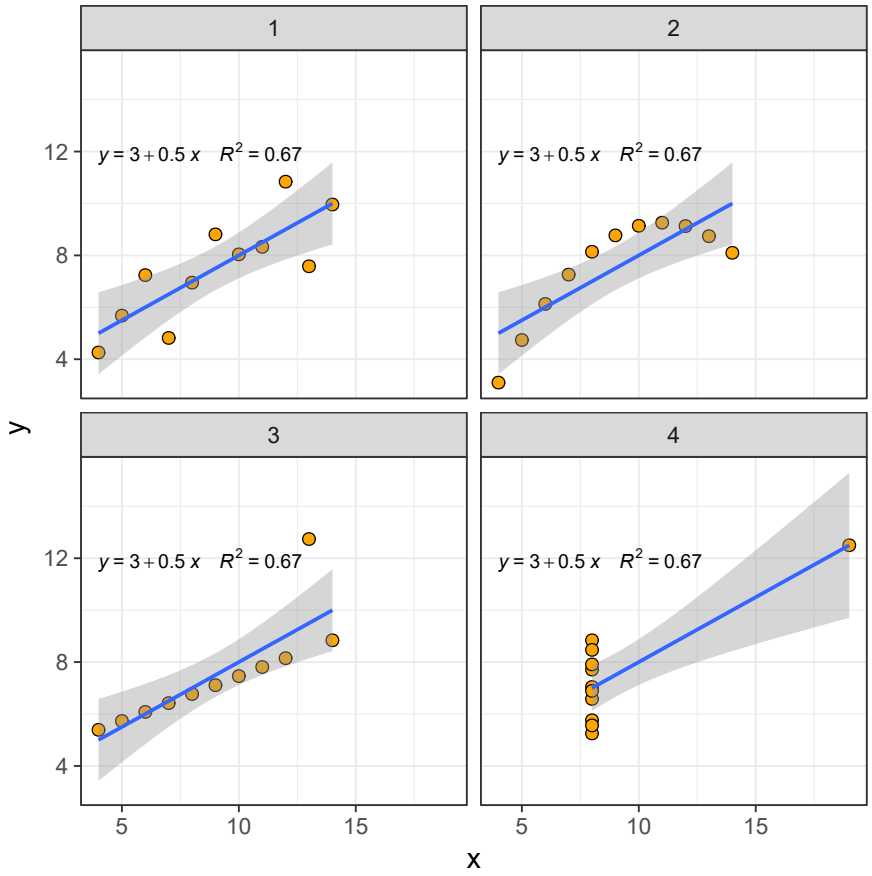
7.17.1 Anscombe's example revisited

To make the example self contained we repeat the code from chapter ??.

```
# we rearrange the data
my.mat <- matrix(as.matrix(anscombe), ncol=2)
my.anscombe <- tibble(x = my.mat[ , 1],
                      y = my.mat[ , 2],
                      case=factor(rep(1:4, rep(11,4))))
```

```
ggplot(my.anscombe, aes(x = x, y = y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm") +
  stat_poly_eq(formula = y ~ x, parse = TRUE,
```

```
          aes(label = paste(..eq.label.., ..rr.label.., sep = "~~~~"))) +
  facet_wrap(~case, ncol=2) +
  theme_bw(16)
```



7.17.2 Heatmaps

7.17.3 Volcano plots

7.17.4 Quadrat plots

```
try(detach(package:MASS))
try(detach(package:xts))
try(detach(package:ggalt))
```

```
try(detach(package:ggbiplot))
try(detach(package:ggstance))
try(detach(package:gganimate))
try(detach(package:ggpmisc))
try(detach(package:ggrepel))
try(detach(package:viridis))
try(detach(package:tibble))
```

8 Plotting maps with ggmap

8.1 Plotting data onto maps

```
library(ggmap)
library(rgdal)

## Loading required package: sp
## rgdal: version: 1.2-4, (SVN revision 643)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.0.1, released 2015/09/15
##    Path to GDAL shared files:    C:/Users/aphalo/Documents/R/win-
library/3.3/rgdal/gdal
##  Loaded PROJ.4 runtime: Rel.  4.9.2, 08 September 2015, [PJ_VERSION:
492]
##    Path to PROJ.4 shared files:    C:/Users/aphalo/Documents/R/win-
library/3.3/rgdal/proj
##  Linking to sp version: 1.2-3
```

Another extension to package `ggplot2` is package `ggmap`. Package `ggmap` makes it possible to plot data using normal `ggplot2` syntax on top of a map. Maps can be easily retrieved from the internet through different services. Some of these services require the user to register and obtain a key for access. As Google Maps do not require such a key for normal resolution maps, we use this service in the examples.

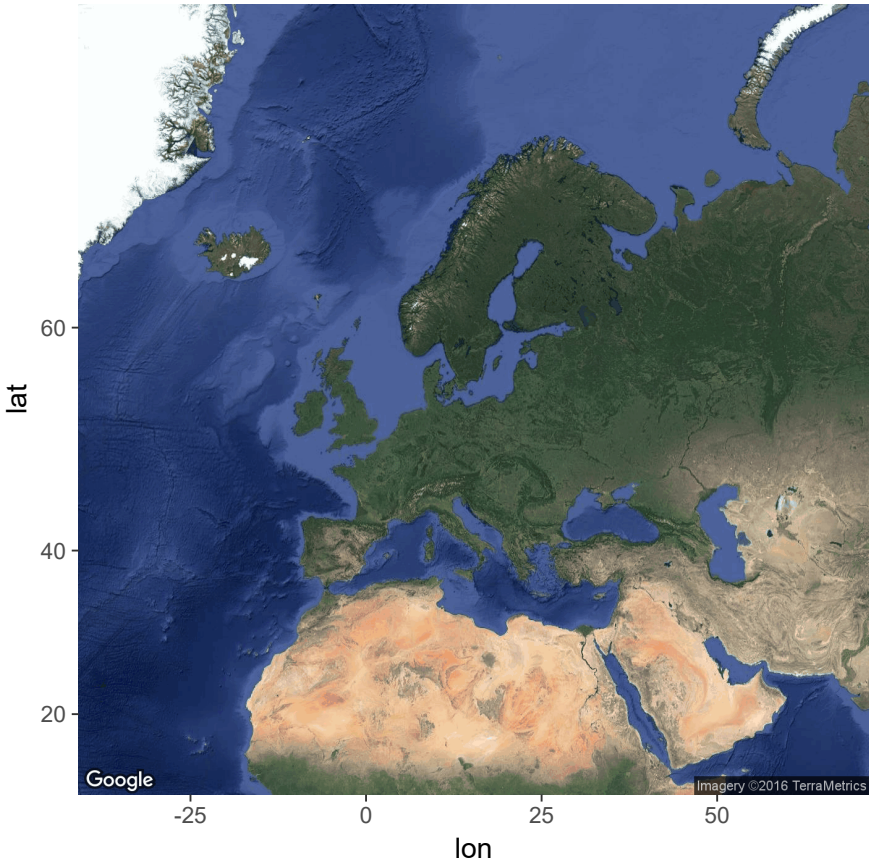
The first step is to fetch the desired map. One can fetch the maps base on any valid Google Maps search term, or by giving the coordinates at the center of the map. Although `zoom` defaults to "auto", frequently the best result is obtained by providing this argument. Valid values for `zoom` are integers in the range 1 to 20.

> We will fetch maps from Google Maps. We have disabled the messages, to avoid repeated messages about Google's terms of use.
>
> **Google Maps API Terms of Service:** `http://developers.google.com/maps/terms`
>
> **Information from URL:** `http://maps.googleapis.com/maps/api/geocode/json?address=Europe&sensor=false`

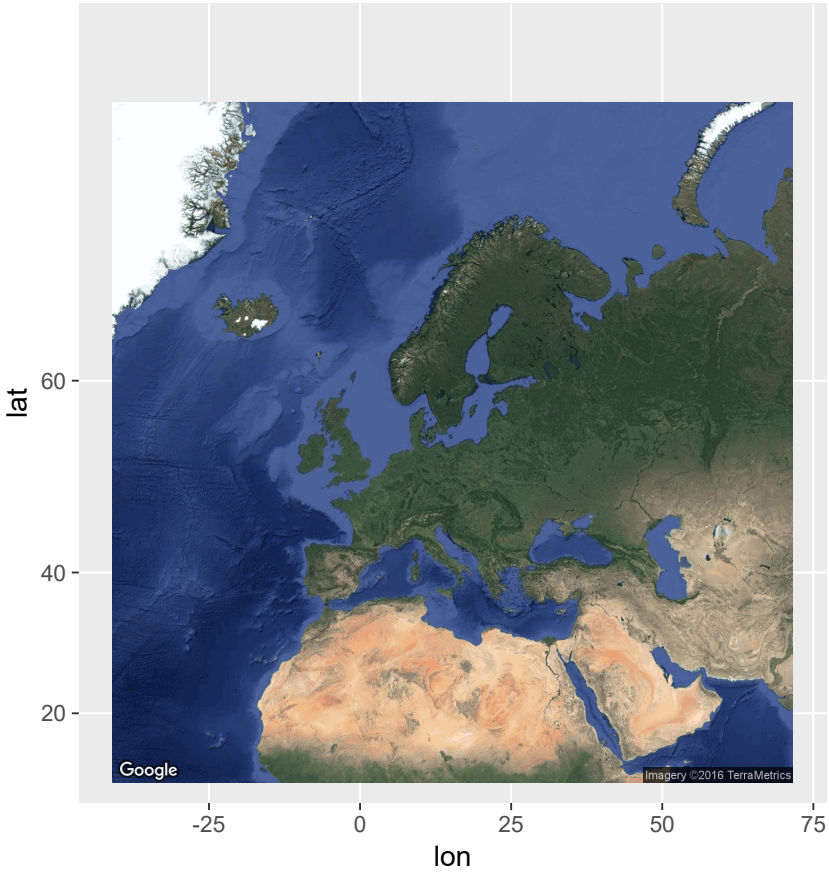
Map from URL: http://maps.googleapis.com/maps/api/staticmap?center=Europe&zoom=3&size=%20640x640&scale=%202&maptype=terrain&sensor=false

We start by fetching and plotting a map of Europe of type `satellite`. We use the default extent `panel`, and also the extent `device` and `normal`. The `normal` plot includes axes showing the coordinates, while `device` does not show them, while `panel` shows axes but the map fits tightly into the drawing area:

```
Europe1 <- get_map("Europe", zoom=3, maptype="satellite")
ggmap(Europe1)

ggmap(Europe1, extent = "device")

## Warning: `panel.margin` is deprecated. Please use `panel.spacing`
property instead

ggmap(Europe1, extent = "normal")
```


lat
60
40
20
Google
Imagery ©2016 TerraMetrics
-25
0
25
50
lon

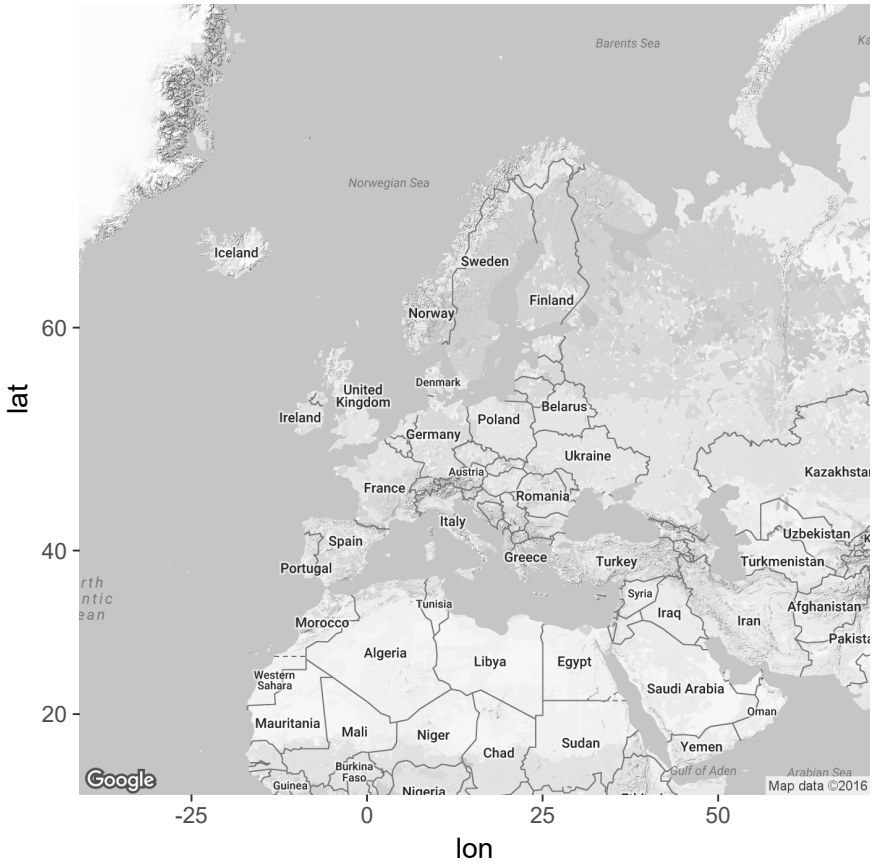

Google
Imagery ©2016 TerraMetrics



To demonstrate the option to fetch a map in black and white instead of the default colour version, we use a map of Europe of type `terrain`.

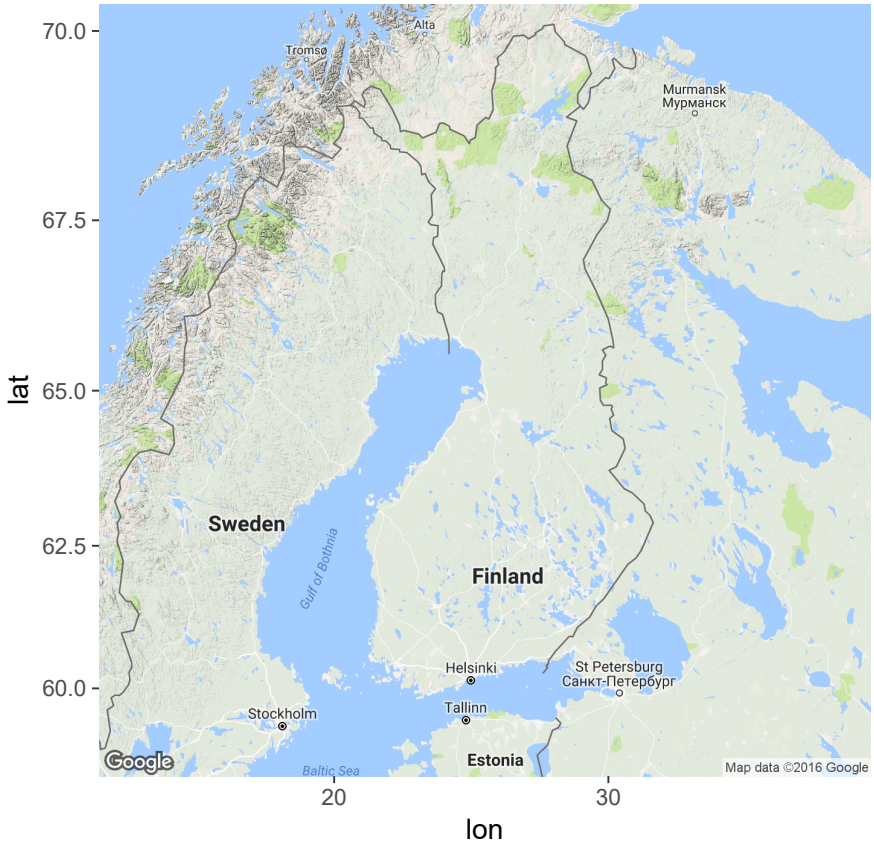
```
Europe2 <- get_map("Europe", zoom=3,
                   maptype="terrain")
ggmap(Europe2)

Europe3 <- get_map("Europe", zoom=3,
                   maptype="terrain",
                   color="bw")
ggmap(Europe3)
```


Barents Sea
Norwegian Sea
Iceland
Sweden
Finland
Norway
Denmark
United Kingdom
Ireland
Belarus
Poland
Germany
Ukraine
Kazakhstan
Austria
France
Romania
Italy
Uzbekistan
Spain
Greece
Turkey
Turkmenistan
Portugal
Syria
Iraq
Afghanistan
Tunisia
Iran
Morocco
Pakistan
Algeria
Libya
Egypt
Western Sahara
Saudi Arabia
Oman
Mauritania
Mali
Niger
Chad
Sudan
Yemen
Burkina Faso
Guinea
Nigeria
Gulf of Aden
Arabian Sea
Google
Map data ©2016
60
40
20
lat
-25
0
25
50
lon



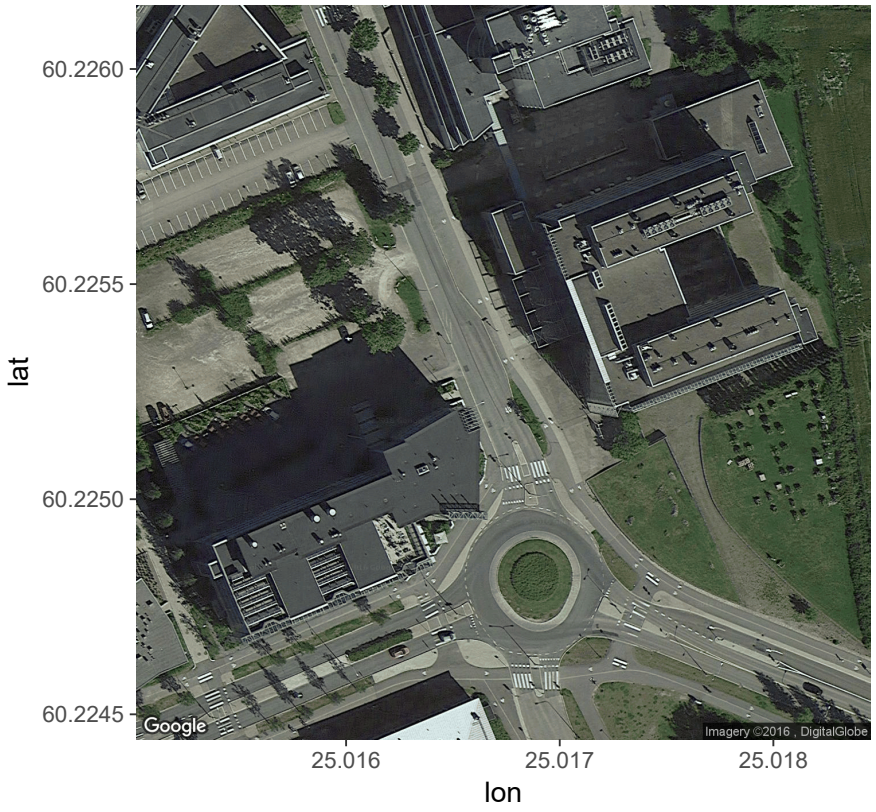
To demonstrate the difference between type `roadmap` and the default type `terrain`, we use the map of Finland. Note that we search for "Oulu" instead of "Finland" as Google Maps takes the position of the label "Finland" as the center of the map, and clips the northern part. By means of `zoom` we override the default automatic zooming onto the city of Oulu.

```
Finland1 <- get_map("Oulu", zoom=5, maptype="terrain")
ggmap(Finland1)

Finland2 <- get_map("Oulu", zoom=5, maptype="roadmap")
ggmap(Finland2)
```


70.0
67.5
65.0
62.5
60.0
lat
20
30
lon
Alta
Tromsø
Murmansk
Мурманск
Sweden
Finland
Gulf of Bothnia
Helsinki
St Petersburg
Санкт-Петербург
Tallinn
Stockholm
Estonia
Baltic Sea
Google
Map data ©2016 Google

We can even search for a street address, and in this case with high zoom value, we can see the building where one of us works:

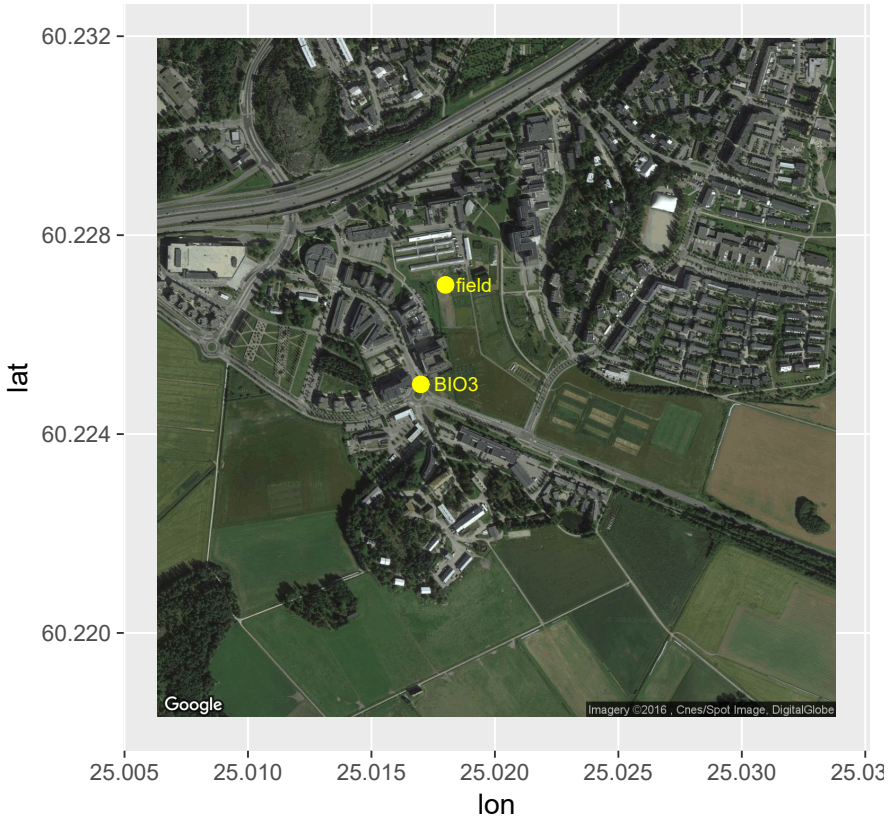
```
BIO3 <- get_map("Viikinkaari 1, 00790 Helsinki",
                zoom=18,
                maptype="satellite")
ggmap(BIO3)
```

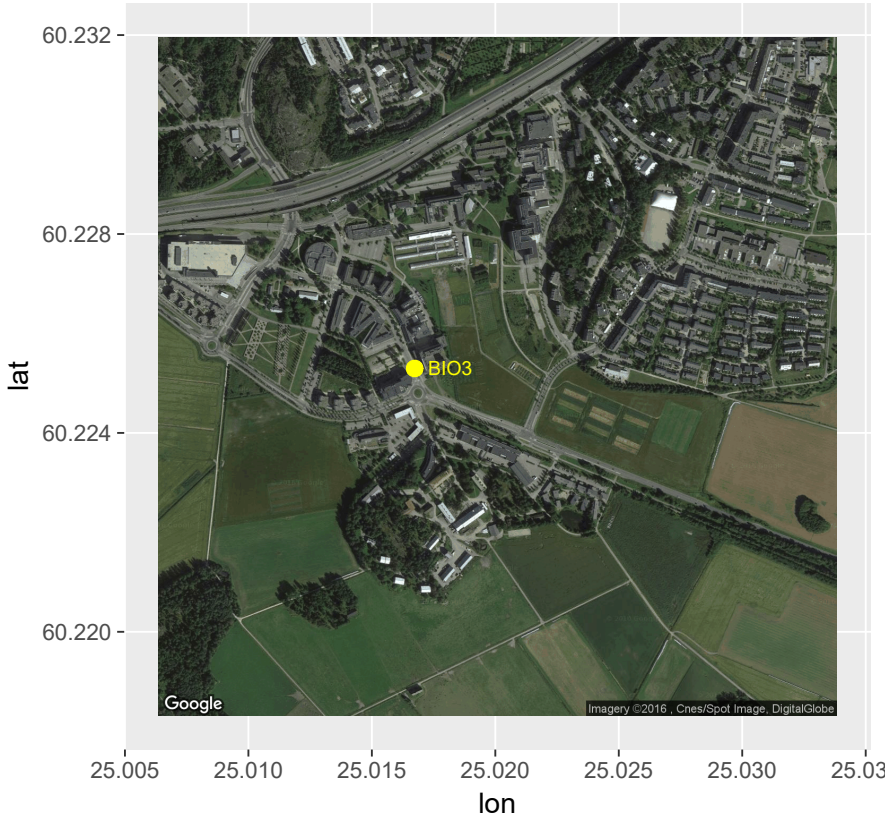


We will now show a simple example of plotting data on a map, first by explicitly giving the coordinates, and in the second example we show how to fetch from Google Maps coordinate values that can be then plotted. We use function `geocode`. In one example we use `geom_point` and `geom_text`, while in the second example we use `annotate`, but either approach could have been used for both plots:

```
viikki <- get_map("Viikki, 00790 Helsinki",
                  zoom=15,
                  maptype="satellite")

our_location <- data.frame(lat=c(60.225, 60.227),
                           lon=c(25.017, 25.018),
                           label=c("BIO3", "field"))
ggmap(viikki, extent = "normal") +
```

```
  geom_point(data=our_location, aes(y=lat, x=lon),
             size=4, colour="yellow") +
  geom_text(data=our_location, aes(y=lat, x=lon, label=label),
            hjust=-0.3, colour="yellow")

our_geocode <- geocode("Viikinkaari 1, 00790 Helsinki")
ggmap(viikki, extent = "normal") +
  annotate(geom="point",
           y=our_geocode[ 1, "lat"], x=our_geocode[ 1, "lon"],
           size=4, colour="yellow") +
  annotate(geom="text",
           y=our_geocode[ 1, "lat"], x=our_geocode[ 1, "lon"],
           label="BIO3", hjust=-0.3, colour="yellow")
```





Using `get_map` from package `ggmap` for drawing a world map is not possible at the time of writing. In addition a worked out example of how to plot shape files, and how to download them from a repository is suitable as our final example. We also show how to change the map projection. The example is adapted from a blog post at `http://rpsychologist.com/working-with-shapefiles-projections-and-world-maps-in-ggplot`.

We start by downloading the map data archive files from `http://www.naturalearthdata.com` which is available in different layers. We only use three of the available layers: 'physical' which describes the coastlines and a grid and bounding box, and 'cultural' which gives country borders. We save them in a folder with name 'maps', which is expected to already exist. After downloading each file, we unzip it.

```
getwd()

## [1] "D:/aphalo/Documents/Own_manuscripts/Books/using-r"

oldwd <- setwd("./maps")

url_path <-
#  "http://www.naturalearthdata.com/download/110m/"
 "http://www.naturalearthdata.com/http//www.naturalearthdata.com/download/110m/"

download.file(paste(url_path,
                    "physical/ne_110m_land.zip",
                    sep=""), "ne_110m_land.zip")
unzip("ne_110m_land.zip")

download.file(paste(url_path,
                    "cultural/ne_110m_admin_0_countries.zip",
                    sep=""), "ne_110m_admin_0_countries.zip")
unzip("ne_110m_admin_0_countries.zip")

download.file(paste(url_path,
                    "physical/ne_110m_graticules_all.zip",
                    sep=""), "ne_110m_graticules_all.zip")
unzip("ne_110m_graticules_all.zip")

setwd(oldwd)
```

We list the layers that we have downloaded.

```
ogrListLayers(dsn="./maps")

## [1] "ne_110m_admin_0_countries"
## [2] "ne_110m_graticules_1"
## [3] "ne_110m_graticules_10"
## [4] "ne_110m_graticules_15"
## [5] "ne_110m_graticules_20"
## [6] "ne_110m_graticules_30"
## [7] "ne_110m_graticules_5"
## [8] "ne_110m_land"
## [9] "ne_110m_wgs84_bounding_box"
## attr(,"driver")
## [1] "ESRI Shapefile"
## attr(,"nlayers")
## [1] 9
```

Next we read the layer for the coastline, and use `fortify` to convert it into a data frame. We also create a second version of the data using the Robinson projection.

```
wmap <- readOGR(dsn="./maps", layer="ne_110m_land")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_land"
## with 127 features
## It has 2 fields

wmap.data <- fortify(wmap)

## Regions defined for each Polygons

wmap_robin <- spTransform(wmap, CRS("+proj=robin"))
wmap_robin.data <- fortify(wmap_robin)

## Regions defined for each Polygons
```

We do the same for country borders,

```
countries <- readOGR("./maps", layer="ne_110m_admin_0_countries")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_admin_0_countries"
## with 177 features
## It has 63 fields

countries.data <- fortify(countries)

## Regions defined for each Polygons

countries_robin <- spTransform(countries, CRS("+init=ESRI:54030"))
countries_robin.data <- fortify(countries_robin)

## Regions defined for each Polygons
```

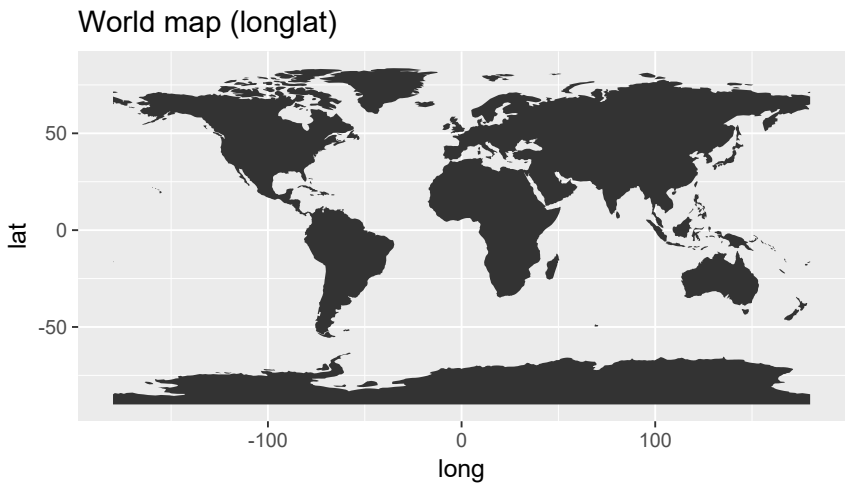
and for the graticule at 15° intervals, and the bounding box.

```
grat <- readOGR("./maps", layer="ne_110m_graticules_15")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_graticules_15"
## with 35 features
## It has 5 fields
## Integer64 fields read as strings:  degrees scalerank

grat.data <- fortify(grat)
grat_robin <- spTransform(grat, CRS("+proj=robin"))
grat_robin.data <- fortify(grat_robin)

bbox <- readOGR("./maps", layer="ne_110m_wgs84_bounding_box")

## OGR data source with driver: ESRI Shapefile
```

```
## Source: "./maps", layer: "ne_110m_wgs84_bounding_box"
## with 1 features
## It has 2 fields

bbox.data <- fortify(bbox)

## Regions defined for each Polygons

bbox_robin <- spTransform(bbox, CRS("+proj=robin"))
bbox_robin.data <- fortify(bbox_robin)

## Regions defined for each Polygons
```

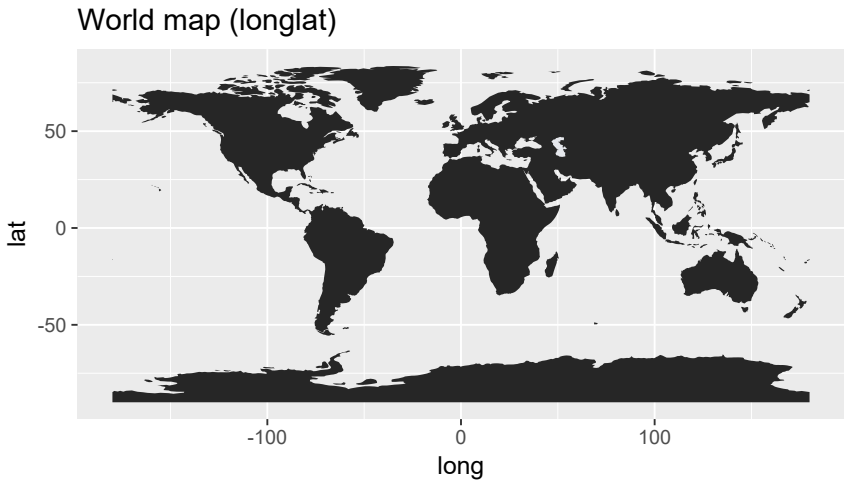
Now we plot the world map of the coastlines, on a longitude and latitude scale, as a `ggplot` using `geom_polygon`.

```
ggplot(wmap.data, aes(long,lat, group=group)) +
  geom_polygon() +
  labs(title="World map (longlat)") +
  coord_equal()
```



There is one noticeable problem in the map shown above: the Caspian sea is missing. We need to use aesthetic `fill` and a manual scale to correct this.

```
ggplot(wmap.data, aes(long,lat, group=group, fill=hole)) +
  geom_polygon() +
  labs(title="World map (longlat)") +
  scale_fill_manual(values=c("#262626", "#e6e8ed"),
                    guide="none") +
  coord_equal()
```



When plotting a map using a projection, many default elements of the `ggplot` theme need to be removed, as the data is no longer in units of degrees of latitude and longitude and axes and their labels are no longer meaningful.

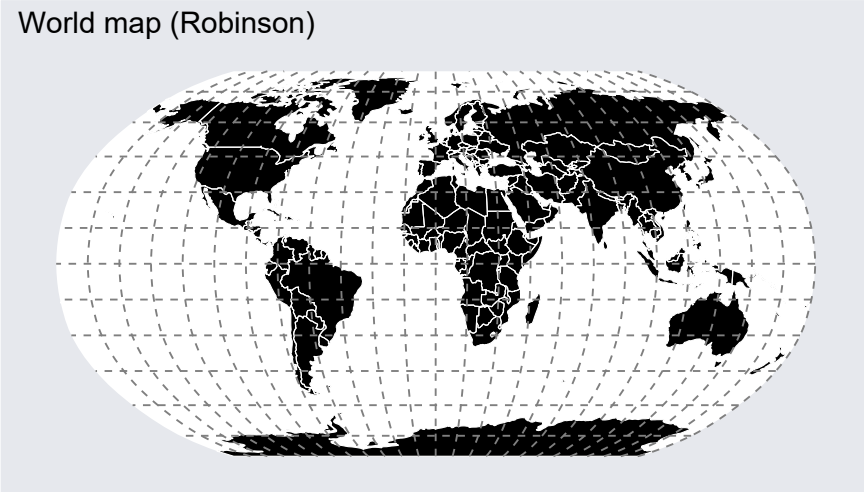
```
theme_map_opts <-
  list(theme(panel.grid.minor = element_blank(),
             panel.grid.major = element_blank(),
             panel.background = element_blank(),
             plot.background = element_rect(fill="#e6e8ed"),
             panel.border = element_blank(),
             axis.line = element_blank(),
             axis.text.x = element_blank(),
             axis.text.y = element_blank(),
             axis.ticks = element_blank(),
             axis.title.x = element_blank(),
             axis.title.y = element_blank()))
```

Finally we plot all the layers using the Robinson projection. This is still a `ggplot` and consequently one can plot data on top of the map, being aware of the transformation of the scale needed to make the data location match locations in a map using a certain projection.

```
ggplot(bbox_robin.data, aes(long,lat, group=group)) +
  geom_polygon(fill="white") +
  geom_polygon(data=countries_robin.data,
               aes(long,lat, group=group,
                   fill=hole)) +
  geom_path(data=countries_robin.data,
            aes(long,lat, group=group, fill=hole),
```

```
            color="white",
            size=0.3) +
  geom_path(data=grat_robin.data,
            aes(long, lat, group=group, fill=NULL),
            linetype="dashed",
            color="grey50") +
  labs(title="World map (Robinson)") +
  coord_equal() +
  theme_map_opts +
  scale_fill_manual(values=c("black", "white"),
                    guide="none")
```

```
## Warning: Ignoring unknown aesthetics: fill
## Warning: Ignoring unknown aesthetics: fill
```




```
try(detach(package:ggmap))
try(detach(package:rgdal))
```

9 If and when R needs help

9.1 Introduction

S (R is sometimes called Gnu S) is a very well designed language, but it was designed with a specific purpose in mind. No tool can handle efficiently and gracefully every task one may want to throw at it. R is no different. However, R is good at working together with other languages and tools. As for most tasks in life, for data analysis one may find the best tools for our toolbox scattered around the place and coming from different sources. What is important in any tool pack is that all the tools can be used together in an efficient and 'fluid' combination. In this final chapter, I will give an overview of what is available, giving only minimal examples, as a teaser for readers to explore on their own what may be most useful for their jobs.

9.2 R's limitations and strengths

9.2.1 Using the best tool for each job

9.2.2 R is great, but

9.2.3 On choice versus fashion

9.2.4 On finding one's own way around

9.2.5 Getting around performance issues

9.2.6 Re-using code writing in other languages

9.3 C++

9.4 FORTRAN and C

9.5 Phyton

9.6 Java

9.7 Javascript

9.8 sh, bash

10 Further reading about R

10.1 Introductory texts

Dalgaard 2008; Teetor 2011; Zuur et al. 2009

10.2 Texts on specific aspects

Chang 2013; Everitt and Hothorn 2011; Faraway 2004; Faraway 2006; Fox 2002; Fox and Weisberg 2010

10.3 Advanced texts

Ihaka and Gentleman 1996; Matloff 2011; Murrell 2011; Pinheiro and Bates 2000; Wickham 2014, 2015; Xie 2013

Bibliography

Anscombe, F. J. (1973). "Graphs in Statistical Analysis". In: *The American Statistician* 27.1, p. 17. DOI: 10.2307/2682899. URL: http://dx.doi.org/10.2307/2682899 (cit. on p. 178).

Chang, W. (2013). *R Graphics Cookbook*. 1-2. Sebastopol: O'Reilly Media, p. 413. ISBN: 9781449316952. URL: http://medcontent.metapress.com/index/A65RM03P4874243N.pdf (cit. on pp. 73, 265).

Dalgaard, P. (2008). *Introductory Statistics with R*. Springer, p. 380. ISBN: 0387790543 (cit. on p. 265).

Everitt, B. and T. Hothorn (2011). *An Introduction to Applied Multivariate Analysis with R*. Springer, p. 288. ISBN: 1441996494. URL: http://www.amazon.co.uk/Introduction-Applied-Multivariate-Analysis-Use/dp/1441996494 (cit. on p. 265).

Faraway, J. J. (2004). *Linear Models with R*. Boca Raton, FL: Chapman & Hall/CRC, p. 240. URL: http://www.maths.bath.ac.uk/~jjf23/LMR/ (cit. on p. 265).

Faraway, J. J. (2006). *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. Chapman & Hall/CRC Taylor & Francis Group, p. 345. ISBN: 158488424X (cit. on p. 265).

Fox, J. (2002). *An {R} and {S-Plus} Companion to Applied Regression*. Thousand Oaks, CA, USA: Sage Publications. URL: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion/index.html (cit. on p. 265).

Fox, J. and H. S. Weisberg (2010). *An R Companion to Applied Regression*. SAGE Publications, Inc, p. 472. ISBN: 141297514X. URL: http://www.amazon.com/An-R-Companion-Applied-Regression/dp/141297514X (cit. on p. 265).

Ihaka, R. and R. Gentleman (1996). "R: A Language for Data Analysis and Graphics". In: *J. Comput. Graph. Stat.* 5, pp. 299–314 (cit. on p. 265).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843. URL: http://www.amazon.com/The-Art-Programming-Statistical-Software/dp/1593273843 (cit. on p. 265).

Murrell, P. (2011). *R Graphics, Second Edition (Chapman & Hall/CRC The R Series)*. CRC Press, p. 546. ISBN: 1439831769. URL: http://www.amazon.com/Graphics-Second-Edition-Chapman-Series/dp/1439831769 (cit. on p. 265).

Pinheiro, J. C. and D. M. Bates (2000). *Mixed-Effects Models in S and S-Plus*. New York: Springer (cit. on p. 265).

Teetor, P. (2011). *R Cookbook*. 1st ed. Sebastopol: O'Reilly Media, p. 436. ISBN: 9780596809157 (cit. on p. 265).

Wickham, H. (2014). *Advanced R*. Chapman & Hall/CRC The R Series. CRC Press. ISBN: 9781466586970. URL: https://books.google.fi/books?id=G5PNBQAAQBAJ (cit. on p. 265).

– (2015). *R Packages*. O'Reilly Media. ISBN: 9781491910542. URL: https://books.google.fi/books?id=eqOxBwAAQBAJ (cit. on p. 265).

Xie, Y. (2013). *Dynamic Documents with R and knitr (Chapman & Hall/CRC The R Series)*. Chapman and Hall/CRC, p. 216. ISBN: 1482203537. URL: http://www.amazon.com/Dynamic-Documents-knitr-Chapman-Series/dp/1482203537 (cit. on p. 265).

Zuur, A. F., E. N. Ieno, and E. Meesters (2009). *A Beginner's Guide to R*. 1st ed. Springer, p. 236. ISBN: 0387938362. URL: http://www.amazon.com/Beginners-Guide-Use-Alain-Zuur/dp/0387938362 (cit. on p. 265).

Index