Introduction to R

setup, data manipulation and plotting

Alexander Tolios

day 3

preface

housekeeping

Many examples were taken from different online sources:

- Book by Yihui Xie et al.
- Book by Hadley Wickham and Garrett Grolemund
- Presentation by Thomas Lin Pedersen

In addition, many packages from/around the `RStudio` universe have their own cheat sheets.

---

programming concepts for data science

the IDE

When programming: use an IDE (integrated development environment)

- RStudio
- vim/neovim with plugins like e.g. Nvim-R
- Emacs with ESS

Advantages of an IDE in general:

- code completion
- highlighting
- easier access to some often-used function

RStudio

If unsure: use RStudio (very friendly for beginners)

Recommended tweaks (really, do those):

- Tools -> Global Options -> General -> DEACTIVATE 'Restore most recently opened project at startup'
- Tools -> Global Options -> General -> DEACTIVATE 'Restore .RData into workspace at startup'
- Tools -> Global Options -> General -> CHANGE 'Save workspace to .RData on exit: Never'
- Tools -> Global Options -> General -> DEACTIVATE 'Always save history (even when not saving .RData)'

Other interesting options to consider (personal preference):

- Tools -> Global Options -> Appearance -> Editor theme: XXX (try out which one you like, I prefer it dark)
- Tools -> Global Options -> Pane Layout -> XXX (try out which one you like, I prefer 'Source: left, Console: right')
- Tools -> Global Options -> Code -> Editing -> Keybindings: XXX (before you eventually switch to something different, try the keybindings out here, I prefer the vim-keybindings - but do your homework before!)

REPL-style

The read-eval-print-loop (REPL) allows to execute one command after the other and to get the results back to the user.

This allows instantaneous prototyping and is the standard when doing data science work (although it is not limited to scripting languages).

Examples of REPL-use cases:

- all kinds of shells (to interact with the computer)
- typical data science languages (R, Python, Julia, Matlab/Octave, ...)
- interactive debugging

Examples of REPL-abuses:

- An R-script, which doesn't run automatically, because you have to run it line-by-line and sometimes leave stuff out ;)

The goal should ALWAYS be that you can run a script non-interactively and get some results. If this doesn't work your analysis is probably not reproducible!

For an example on how NOT to do it see the R-script `bad_R-script.R`

Basic 'modes' of working in `R`

shell-based only

Only use this if you want to do trivial stuff (e.g. amplicon-calculation)

You could do that without an IDE (since it probably won't help you, but adds a lot of overhead)

R-script

This should be your default way of programming in `R`.

Keep everything you run in your script. Also create a separate folder for your project and store all the data and scripts you need in this folder.

Write **lots** of comments, since you never write a script only for your current self: everything should be understandable also for your future self.

Whenever you are planning to reuse something, assign values to objects. That way you separate input from code, which makes everything easier to read.

---

just a starter. . .

basic stuff to do with R

use R as a simple calculator

```
2 + 2
#> [1] 4
```

This can be useful for quick calculations.

R is more than a calculator

Stuff on functions

```

a <- 3
b <- sqrt(a)
print(b)
#> [1] 1.732051
```

If you don't know how to use a function, look it up in the help-page using `?FUNCTIONNAME`

types of objects in R

Basic R data objects can store a single value

```
# basic R data type consisting of a single value (of the type numeric, integer, complex, logical or cha
a <- 3
print(a)
#> [1] 3
b <- "text"
print(b)
#> [1] "text"
```

---

expand Rs functionality

Using the R-shell you can access functions exported by all the loaded libraries.

To install a package, use the command `install.packages("NAME_OF_PACKAGE")`. Since this command will always install the package (even if it's already installed), it's recommended to use `pacman::p_load(NAME_OF_PACKAGE)` instead.

Packages can be loaded using `library(NAME_OF_PACKAGE)`.

Advanced 'modes' of working in R

Rmarkdown document

Intertwine a markdown-document with chunks of code that gets executed.

Interacting with other languages

You can use different languages for that, not only R.

```
# this is an R-chunk        (starting with ```{r}...```
a <- 2
```

Python and R can easily interact.

```
# this is a Python-chunk    (starting with ```{python}...```
a = 3
print(a)
```

```
#> 3
print(r.a)
#> 2.0
```

Keep in mind that objects created in R have to be called using `r.OBJECTNAME`, otherwise the Python-object will be used.

By using a classical shell (like bash) it's easy to interact with your base operating system.

```
# this is a bash-chunk      (starting with ```{bash}...```
ls -l
```

```
#> total 20
#> -rw------- 1 ruser ruser 20477 Apr 16 15:07 main.Rmd
#> drwx------ 1 ruser ruser   248 Apr 16 15:00 output
```

Obviously the interpreter for each of the languages has to be installed to be used.

Output formats using Rmarkdown

Rmarkdown can be used to create documents or presentations. Rmarkdown-options can be defined either in the `yaml-header` or using commands which get executed when compiling the document (which are therefore dependend on your choice of output format).

Most common output formats are:

- pdf documents (using LaTeX)
- html documents (using html)
- pdf presentations (using LaTeX Beamer class)
- html presentations (using ioslides or slidy)
- html/js presentations (using reveal.js)
- other formats (e.g. Microsoft Powerpoint, Microsoft Word)

The conversion is performed using the awesome `pandoc` engine.

For details refer to the R Markdown: The Definitive Guide-book by Yihui Xie et al.

Take care when using pdf-output: this often needs many dependencies (e.g. huge parts of the LaTeX development environment).

Compile Rmarkdown-documents

For the compilation of Rmarkdown-documents into an output format the packages `rmarkdown` and `knitr` are used.

For the analyses to be reproducible, specify the compilation command manually:

```r
rmarkdown::render(input = "YOURFILENAME.Rmd")
```

That way you can ensure that it can be run from within a script.

Markdown in general

Markdown is a (or the most) lightweight markup language for document generation.

Be aware that multiple markdown syntaxes exist and that they are NOT compatible!

Relevant examples of markdown commands:

```
#                   Heading level 1
##                  Heading level 2

*italic* or _italic_
**bold** or __bold__
***italic and bold*** or ___italic and bold___

![Figure legend](PATH_TO_IMAGE.png)

* or - or +         Unordered lists
  * or - or +       (with indentation): unordered sublists
1. or 2.            or any other number: ordered lists
   1. or 2.         or any other number: ordered sublists

`stuff`:            stuff to be formated as code

--- or *** or ___   horizontal ruler

[some text](https://example.org): a hyperlink
[some text](https://example.org "a title"): a hyperlink with a title

>                   Quotes
>>                  Subquotes

\*                  escaping special characters (in this case the `*`)

     (4 spaces):    ignore all markdown syntax, just display the text as is
                    (this is how I formated this paragraph)
```

Information on Rmarkdown can be found in Yihuis' book.

---

importing and exporting data

importing data

Several different possibilities of importing data in R exist.

Almost all of them can be solved using the `rio`-package.

```r
rio::import(file = "output/penguins.csv")       %>% as_tibble() %>% head(n = 4)
#> # A tibble: 4 x 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
#> 1 Adelie  Torgersen           39.1          18.7               181        3750
#> 2 Adelie  Torgersen           39.5          17.4               186        3800
#> 3 Adelie  Torgersen           40.3          18                 195        3250
#> 4 Adelie  Torgersen           NA            NA                  NA          NA
#> # i 2 more variables: sex <chr>, year <int>
rio::import(file = "output/penguins.rds")       %>% as_tibble() %>% head(n = 4)
#> # A tibble: 4 x 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
#> 1 Adelie  Torgersen           39.1          18.7               181        3750
#> 2 Adelie  Torgersen           39.5          17.4               186        3800
#> 3 Adelie  Torgersen           40.3          18                 195        3250
#> 4 Adelie  Torgersen           NA            NA                  NA          NA
#> # i 2 more variables: sex <fct>, year <int>
```

---

basic tidy

operations on dataframes

The ‘tidyverse’ is a collection of packages designed to work with dataframes (or tibbles). The main package is `dplyr`.

selecting columns

Use the `select` command to keep or discard specific columns in a dataframe.

```
dplyr::select(iris, Sepal.Length, Species)
#> # A tibble: 150 x 2
#>    Sepal.Length Species
#>           <dbl> <fct>
#>  1          5.1 setosa
#>  2          4.9 setosa
#>  3          4.7 setosa
#>  4          4.6 setosa
#>  5          5   setosa
#>  6          5.4 setosa
#>  7          4.6 setosa
#>  8          5   setosa
#>  9          4.4 setosa
#> 10          4.9 setosa
#> # i 140 more rows

dplyr::select(iris, -Petal.Width)
#> # A tibble: 150 x 4
#>    Sepal.Length Sepal.Width Petal.Length Species
#>           <dbl>       <dbl>        <dbl> <fct>
#>  1          5.1         3.5          1.4 setosa
#>  2          4.9         3            1.4 setosa
#>  3          4.7         3.2          1.3 setosa
#>  4          4.6         3.1          1.5 setosa
#>  5          5           3.6          1.4 setosa
#>  6          5.4         3.9          1.7 setosa
#>  7          4.6         3.4          1.4 setosa
#>  8          5           3.4          1.5 setosa
#>  9          4.4         2.9          1.4 setosa
#> 10          4.9         3.1          1.5 setosa
#> # i 140 more rows
```

filtering rows

With the `filter` command you can keep or discard rows according to a function.

```
dplyr::filter(iris, Species == "setosa")
#> # A tibble: 50 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # i 40 more rows

dplyr::filter(iris, Sepal.Length < 5)
#> # A tibble: 22 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          4.9         3            1.4         0.2 setosa
#>  2          4.7         3.2          1.3         0.2 setosa
#>  3          4.6         3.1          1.5         0.2 setosa
#>  4          4.6         3.4          1.4         0.3 setosa
#>  5          4.4         2.9          1.4         0.2 setosa
#>  6          4.9         3.1          1.5         0.1 setosa
#>  7          4.8         3.4          1.6         0.2 setosa
#>  8          4.8         3            1.4         0.1 setosa
#>  9          4.3         3            1.1         0.1 setosa
#> 10          4.6         3.6          1           0.2 setosa
#> # i 12 more rows
```

Rows can also be chosen by position, by order or a specific variable or randomly.

```
dplyr::slice_head(iris, prop = 0.1)
#> # A tibble: 15 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> 11          5.4         3.7          1.5         0.2 setosa
#> 12          4.8         3.4          1.6         0.2 setosa
#> 13          4.8         3            1.4         0.1 setosa
#> 14          4.3         3            1.1         0.1 setosa
#> 15          5.8         4            1.2         0.2 setosa

dplyr::slice_max(iris, order_by = Sepal.Width, n = 3)
#> # A tibble: 3 x 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>
#> 1          5.7         4.4          1.5         0.4 setosa
#> 2          5.5         4.2          1.4         0.2 setosa
#> 3          5.2         4.1          1.5         0.1 setosa

dplyr::slice_sample(iris, n = 3)
#> # A tibble: 3 x 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>
#> 1          6.7         3.3          5.7         2.1 virginica
#> 2          6.5         3            5.5         1.8 virginica
#> 3          6.5         3.2          5.1         2   virginica
```

create new variables

New variables can be created using information from preexisting ones.

```
dplyr::mutate(iris, new_column = Sepal.Length + 2 * Petal.Width)
#> # A tibble: 150 x 6
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species new_column
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>        <dbl>
#>  1          5.1         3.5          1.4         0.2 setosa         5.5
#>  2          4.9         3            1.4         0.2 setosa         5.3
#>  3          4.7         3.2          1.3         0.2 setosa         5.1
#>  4          4.6         3.1          1.5         0.2 setosa         5
#>  5          5           3.6          1.4         0.2 setosa         5.4
#>  6          5.4         3.9          1.7         0.4 setosa         6.2
#>  7          4.6         3.4          1.4         0.3 setosa         5.2
#>  8          5           3.4          1.5         0.2 setosa         5.4
#>  9          4.4         2.9          1.4         0.2 setosa         4.8
#> 10          4.9         3.1          1.5         0.1 setosa         5.1
#> # i 140 more rows
```

---

combining multiple functions

Try to create as few objects in your global environment as possible (and if you create them, get rid of them if you don't need them any more using `rm`).

The best way to do that is by chaining commands together using the `pipe`.

Cave: the `pipe` is different in different programming languages (e.g. `|` in most programming languages and all classical `UNIX system shells` and `%>%` in `GNU R`). Also `R` pipes objects, while most other classical shells pipe raw text!

---

summarize data

group data according to a variable

A grouping doesn't change the data by itself but allows subsequent commands to be used on groups.

```
iris %>%
  dplyr::group_by(Species)
#> # A tibble: 150 x 5
#> # Groups:   Species [3]
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # i 140 more rows
```

perform summaries

Summaries can be used to perform actions on grouped variables.

```
iris %>%
  dplyr::group_by(Species) %>%
  dplyr::summarize(mean_sepal_length = mean(Sepal.Length))
#> # A tibble: 3 x 2
#>   Species    mean_sepal_length
#>   <fct>                  <dbl>
#> 1 setosa                  5.01
#> 2 versicolor              5.94
#> 3 virginica               6.59
```

Multiple summarization steps can be performed at the same time.

```
iris %>%
  dplyr::group_by(Species) %>%
  dplyr::summarize(mean_sepal_length = mean(Sepal.Length),
                   median_Petal_width = median(Petal.Width),
                   n = n())
#> # A tibble: 3 x 4
#>   Species    mean_sepal_length median_Petal_width     n
#>   <fct>                    <dbl>              <dbl> <int>
#> 1 setosa                    5.01               0.2     50
#> 2 versicolor                5.94               1.3     50
#> 3 virginica                 6.59               2       50
```

---

ggplot2

ggplot - an overview

components of a ggplot-plot

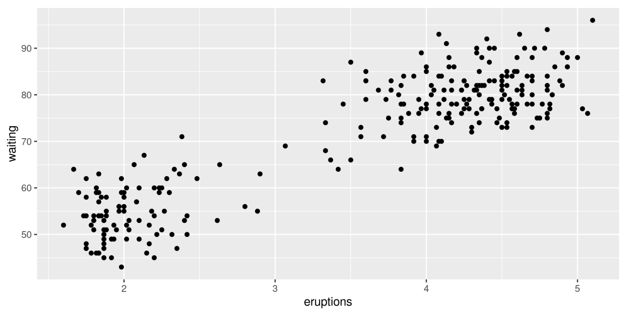
- *ggplot()*: the `ggplot`-function is needed. You can supply different arguments here, if you want them to be true for every part of the plot:
  - *data*: A dataset. The data needs to be *tidy*.
  - *mapping*: A set of aesthetic mappings, specified using the aes() function and combined with the plot defaults as described in aesthetic mappings. This links the data to the graphical properties in the geometries.
- *geometries*: The name of the geometric object to use to draw each observation. All geoms take aesthetics as parameters. If you supply an aesthetic (e.g. x, y, size, colour) as a parameter, it will not be scaled, allowing you to control the appearance of the plot. You only need to set one of stat and geom: every geom has a default stat.
- *statistics*: The name of the statistical tranformation to use. A statistical transformation performs statistical summaries and is key to histograms and smoothes. To keep the data as is, use the "identity" stat. You only need to set one of stat and geom: every stat a default geom.
- *scales*: Translation between variable ranges and property ranges (e.g. from numbers to positions on the plot).
- *coordinates*: Define the mapping of the aesthetics to the plot.
- *facets*: Define multiple panel ("small multiples").
- *themes*: Add additional design elements to a plot.

---

data and geoms

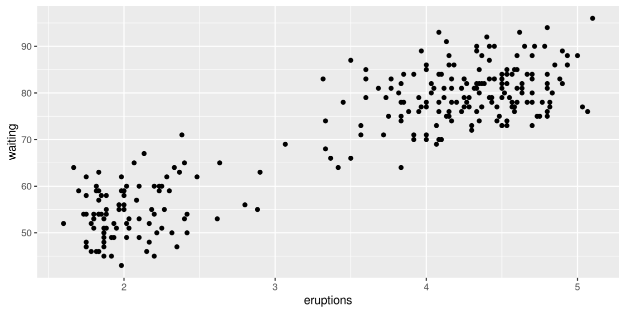
data can be added directly as part of the ggplot-object ...

```
data("faithful")

ggplot(data = faithful,
       mapping = aes(x = eruptions,
                     y = waiting)) +
  geom_point()
```



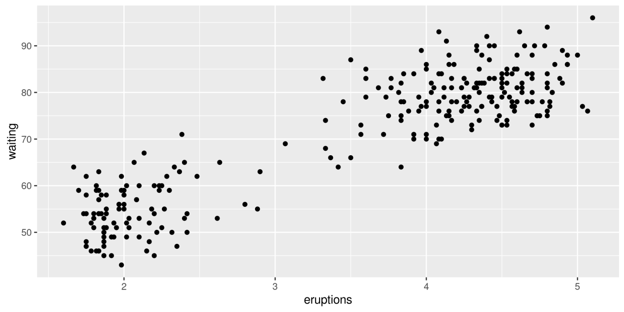
... or be piped into it ...

```
faithful %>%
  ggplot(mapping = aes(x = eruptions,
                       y = waiting)) +
  geom_point()
```



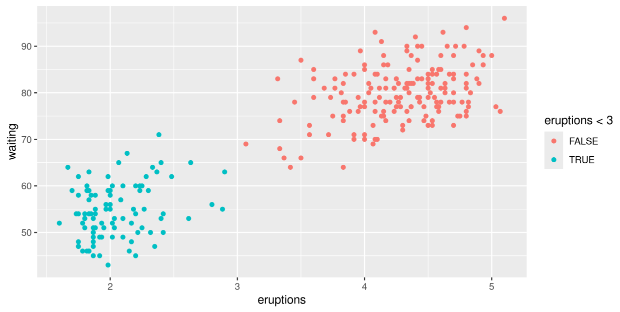
... or as part of the geometries

```


ggplot() +
  geom_point(mapping = aes(x = eruptions,
                           y = waiting),
             data = faithful)
```



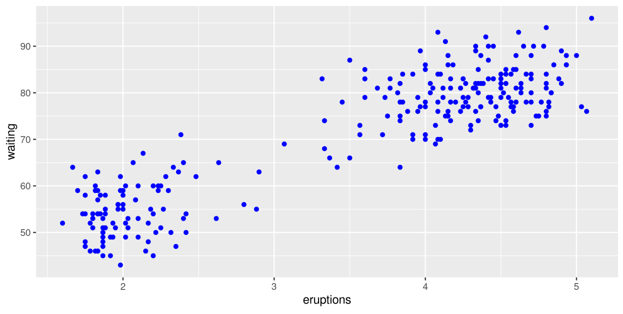
add command to the mapping to be linked to the data

```
ggplot(faithful) +
  geom_point(aes(x = eruptions,
                 y = waiting,
                 colour = eruptions < 3))
```



add command outside of the mapping to be interpreted without link to the data

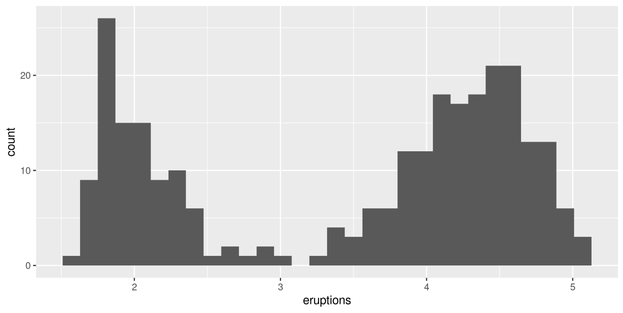
```
ggplot(faithful) +
  geom_point(aes(x = eruptions,
                 y = waiting),
             colour = "blue")
```



the number of mappings needed depends on the used geom

Some geoms only need a single mapping and will calculate the rest for you

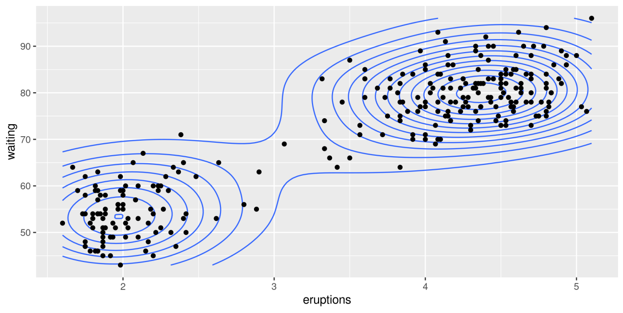
```
ggplot(faithful) +
  geom_histogram(aes(x = eruptions))
```



plots are build from the bottom up

geoms are drawn in the order they are added. The point layer is thus drawn on top of the density contours in the example below.

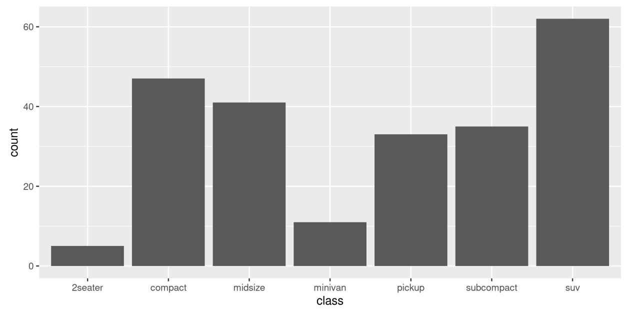
```
ggplot(faithful, aes(x = eruptions,
                     y = waiting)) +
  geom_density_2d() +
  geom_point()
```



Stat

Every geom has a stat. This is why new data (`count`) can appear when using `geom_bar()`.

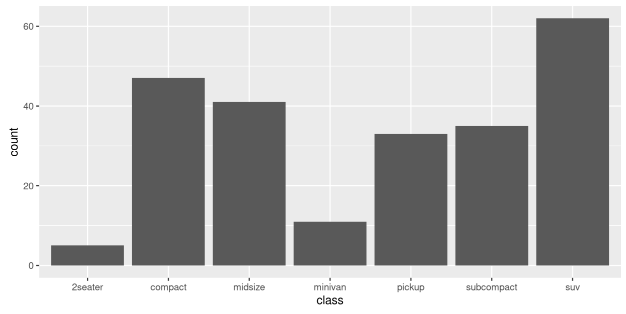
```
data("mpg")
ggplot(mpg) +
  geom_bar(aes(x = class))
```



Overwriting stat

If we have precomputed count we don't want any additional computations to perform and we use the `identity` stat to leave the data alone.

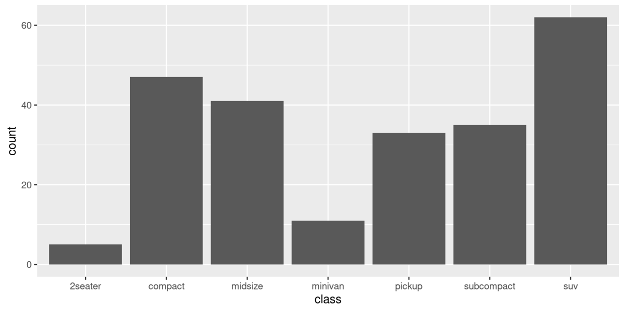
```
mpg_counted <- mpg %>%
  count(class, name = "count")

ggplot(mpg_counted) +
  geom_bar(aes(x = class,
               y = count),
           stat = "identity")
```



Geom / Stat combinations

Most obvious geom/stat combinations have a dedicated geom constructor. The one above is available directly as `geom_col()`.

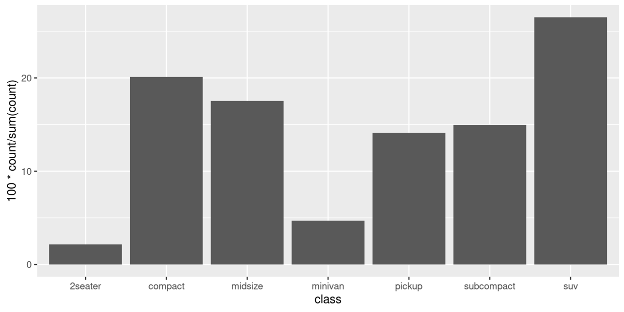
```
ggplot(mpg_counted) +
  geom_col(aes(x = class,
               y = count))
```



calculate values of stat

Values calculated by the stat (e.g. the counting inside the "stat_count"-stat, which is the default when using "geom_bar") is available with the `after_stat()` function inside `aes()`. You can do all sorts of computations inside that.

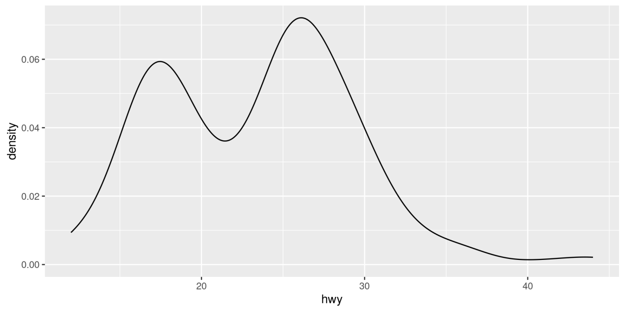
```
ggplot(mpg) +
  geom_bar(aes(x = class,
               y = after_stat(100 * count / sum(count))))
```



use default stats by selecting a geom

Many stats provide multiple variations of the same calculation, and provides a default (here, `density`).

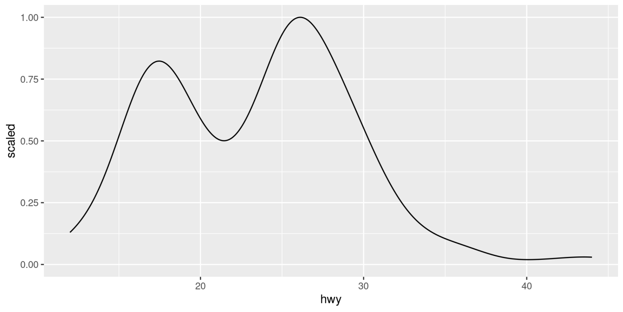
```
ggplot(mpg) +
  geom_density(aes(x = hwy))
```



change default stat by hand

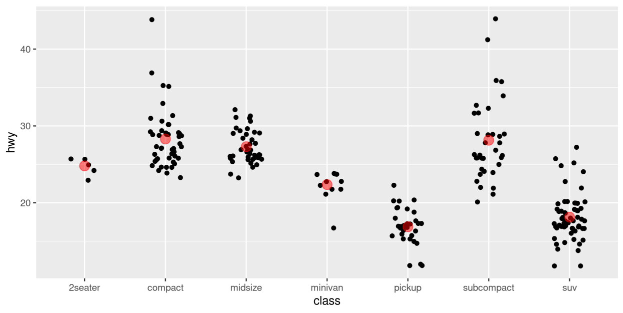
For some calculations the `after_stat()` function must be changed by hand.

```
ggplot(mpg) +
  geom_density(aes(x = hwy,
                   y = after_stat(scaled)))
```



You can also use `stat_summary()` to add a big, transparent red dot at the mean `hwy` for each group.

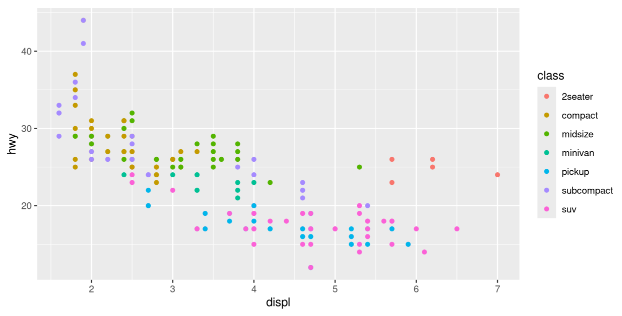
```
ggplot(mpg,
       aes(x = class,
           y = hwy)) +
  geom_jitter(width = 0.2) +
  stat_summary(geom = "point",
               size = 4,
               alpha = 0.5,
               colour = "red",
               fun = mean)
```



Scales

Scales define how the mapping you specify inside `aes()` should happen. All mappings have an associated scale even if not specified.

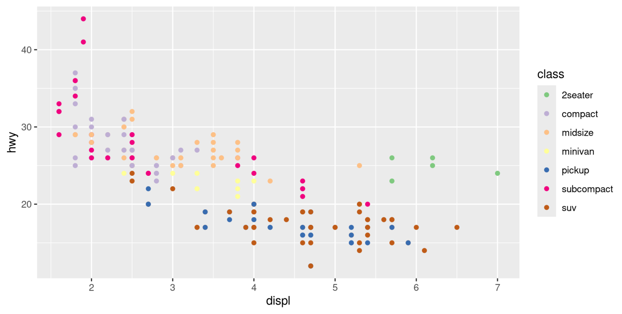
```
ggplot(mpg) +
  geom_point(aes(x = displ,
                 y = hwy,
                 colour = class))
```



Adding a scale explicitly

All scales follow the same naming conventions.

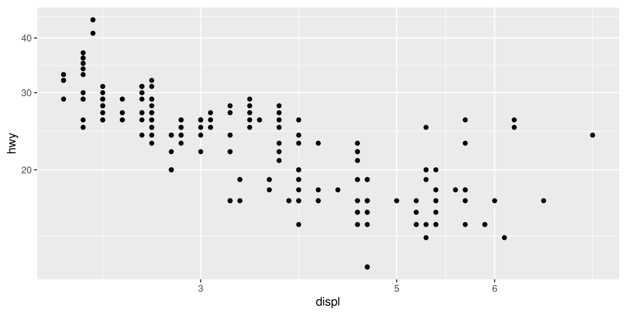
```
ggplot(mpg) +
  geom_point(aes(x = displ,
                 y = hwy,
                 colour = class)) +
  scale_colour_brewer(type = "qual")
```



If you want to use different colors use `RColorBrewer::display.brewer.all()` to see all the different palettes. More color palettes are available using the `paletteer` package (see here for all available color palettes from this package).

Positional mappings (x and y) also have associated scales

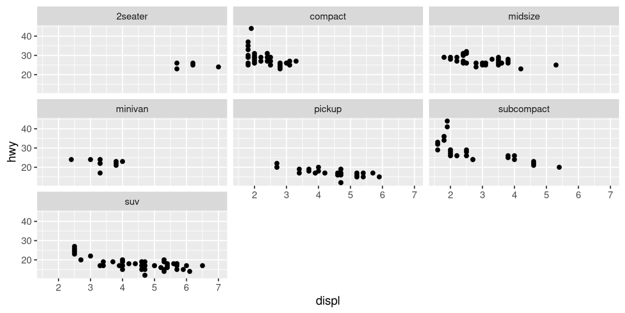
```
ggplot(mpg) +
  geom_point(aes(x = displ,
                 y = hwy)) +
  scale_x_continuous(breaks = c(3, 5, 6)) +
  scale_y_continuous(trans = "log10")
```



Facets

The facet defines how data is split among panels. The default facet (`facet_null()`) puts all the data in a single panel, while `facet_wrap()` and `facet_grid()` allows you to specify different types of small multiples.

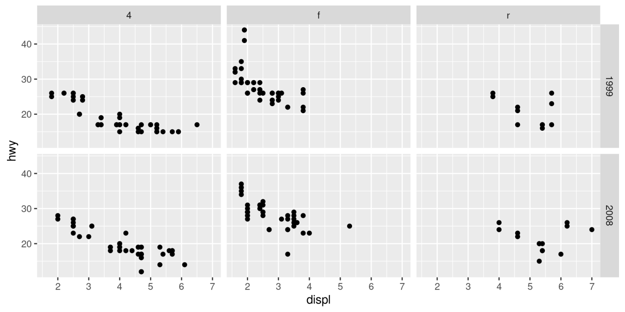
```r
ggplot(mpg) +
  geom_point(aes(x = displ,
                 y = hwy)) +
  facet_wrap(~ class)
```



multiple facets

More than one facet can be used. One of the great things about facets is that they share the axes between the different panels.

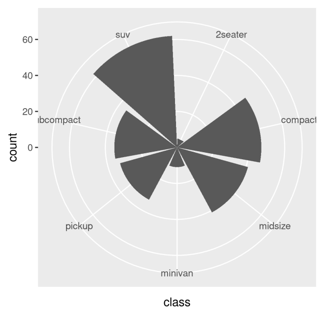
```r
ggplot(mpg) +
  geom_point(aes(x = displ,
                 y = hwy)) +
  facet_grid(year ~ drv)
```



Coordinates

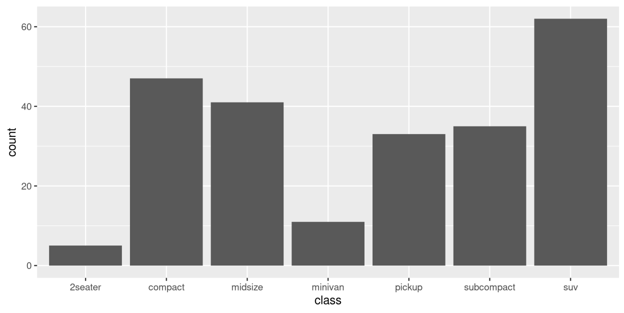
The coordinate system is the fabric you draw your layers on in the end. The default `coord_cartesian` provides the standard rectangular x-y coordinate system. Changing the coordinate system can have dramatic effects.

```r
ggplot(mpg) +
  geom_bar(aes(x = class)) +
  coord_polar()
```



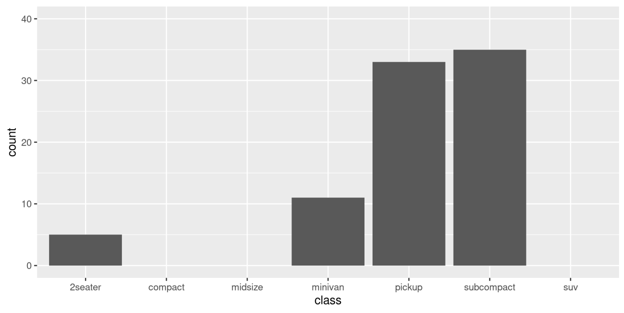
If you want to zoom into your data ...

```
ggplot(mpg) +
  geom_bar(aes(x = class))
```



... you can do this using the scale ...

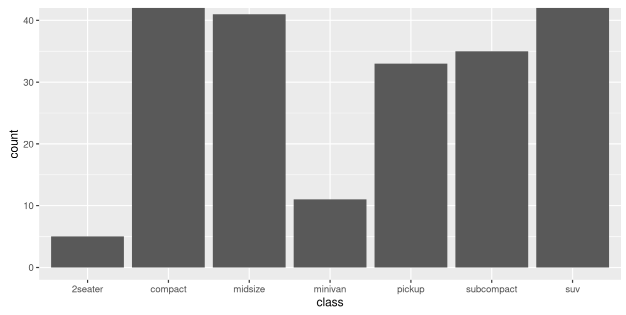
```
ggplot(mpg) +
  geom_bar(aes(x = class)) +
  scale_y_continuous(limits = c(0, 40))
```



But it might not yield the results you want.

... but you should do it using the coord.

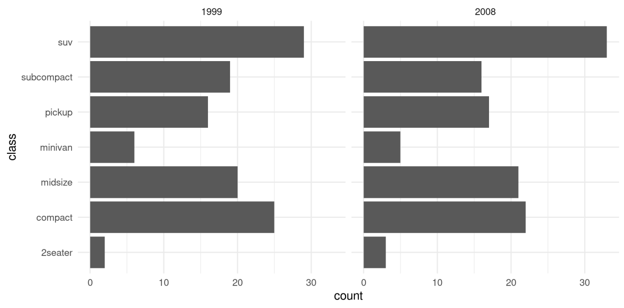
```
ggplot(mpg) +
  geom_bar(aes(x = class)) +
  coord_cartesian(ylim = c(0, 40))
```



Themes

Theming defines the feel and look of your final visualisation and is something you will normally defer to the final polishing of the plot. It is very easy to change looks with a prebuild theme.

```r
ggplot(mpg) +
  geom_bar(aes(y = class)) +
  facet_wrap(~ year) +
  theme_minimal()
```


class
suv
subcompact
pickup
minivan
midsize
compact
2seater
count