Visualization, transformation and reporting with the tidyverse

Kirill Müller, Tobias Schieferdecker, Patrick Schratz

24 November 2019, 23:42 CET

Contents

Preface

See the controls at the top of the website for searching, font size, editing, and a link to the PDF version of the material.

Links

- This website: https://krlmlr.github.io/vistransrep/book
- Scripts and installation instructions: https://github.com/krlmlr/vistransrep-proj/tree/master
  - Prepared scripts: https://github.com/krlmlr/vistransrep-proj/tree/master/script
- The source project for this material: https://github.com/krlmlr/vistransrep

Package versions used

Click to expand

```
withr::with_options(list(width = 80), print(sessioninfo::session_info()))
```

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 3.6.1 (2017-01-27)
##  os       Ubuntu 16.04.6 LTS
##  system   x86_64, linux-gnu
##  ui       X11
##  language en_US.UTF-8
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       UTC
##  date     2019-11-24
```

```
##
## - Packages -------------------------------------------------------------------
##  package      * version     date       lib source
##  askpass        1.1         2019-01-13 [1] CRAN (R 3.6.1)
##  assertthat     0.2.1       2019-03-21 [1] CRAN (R 3.6.1)
##  backports      1.1.5       2019-10-02 [1] CRAN (R 3.6.1)
##  bookdown       0.16        2019-11-22 [1] CRAN (R 3.6.1)
##  broom          0.5.2       2019-04-07 [1] CRAN (R 3.6.1)
##  cellranger     1.1.0       2016-07-27 [1] CRAN (R 3.6.1)
##  cli            1.1.0       2019-03-19 [1] CRAN (R 3.6.1)
##  codetools      0.2-16      2018-12-24 [3] CRAN (R 3.6.1)
##  colorspace     1.4-1       2019-03-18 [1] CRAN (R 3.6.1)
##  crayon         1.3.4       2017-09-16 [1] CRAN (R 3.6.1)
##  crosstalk      1.0.0       2016-12-21 [1] CRAN (R 3.6.1)
##  data.table     1.12.6      2019-10-18 [1] CRAN (R 3.6.1)
##  DBI            1.0.0       2018-05-02 [1] CRAN (R 3.6.1)
##  dbplyr         1.4.2       2019-06-17 [1] CRAN (R 3.6.1)
##  digest         0.6.23      2019-11-23 [1] CRAN (R 3.6.1)
##  dplyr        * 0.8.3       2019-07-04 [1] CRAN (R 3.6.1)
##  DT             0.10        2019-11-12 [1] CRAN (R 3.6.1)
##  ellipsis       0.3.0       2019-09-20 [1] CRAN (R 3.6.1)
##  evaluate       0.14        2019-05-28 [1] CRAN (R 3.6.1)
##  fansi          0.4.0       2018-10-05 [1] CRAN (R 3.6.1)
##  farver         2.0.1       2019-11-13 [1] CRAN (R 3.6.1)
##  fastmap        1.0.1       2019-10-08 [1] CRAN (R 3.6.1)
##  forcats      * 0.4.0       2019-02-17 [1] CRAN (R 3.6.1)
##  fs             1.3.1       2019-05-06 [1] CRAN (R 3.6.1)
##  generics       0.0.2       2018-11-29 [1] CRAN (R 3.6.1)
##  ggplot2      * 3.2.1       2019-08-10 [1] CRAN (R 3.6.1)
##  ggpubr         0.2.4       2019-11-14 [1] CRAN (R 3.6.1)
##  ggsignif       0.6.0       2019-08-08 [1] CRAN (R 3.6.1)
##  git2r          0.26.1      2019-06-29 [1] CRAN (R 3.6.1)
##  glue           1.3.1       2019-03-12 [1] CRAN (R 3.6.1)
##  gtable         0.3.0       2019-03-25 [1] CRAN (R 3.6.1)
##  haven          2.2.0       2019-11-08 [1] CRAN (R 3.6.1)
##  here         * 0.1         2017-05-28 [1] CRAN (R 3.6.1)
##  hms            0.5.2       2019-10-30 [1] CRAN (R 3.6.1)
##  htmltools      0.4.0       2019-10-04 [1] CRAN (R 3.6.1)
##  htmlwidgets    1.5.1       2019-10-08 [1] CRAN (R 3.6.1)
##  httpuv         1.5.2       2019-09-11 [1] CRAN (R 3.6.1)
##  httr           1.4.1       2019-08-05 [1] CRAN (R 3.6.1)
##  jsonlite       1.6         2018-12-07 [1] CRAN (R 3.6.1)
##  knitr          1.26        2019-11-12 [1] CRAN (R 3.6.1)
##  labeling       0.3         2014-08-23 [1] CRAN (R 3.6.1)
##  later          1.0.0       2019-10-04 [1] CRAN (R 3.6.1)
##  lattice        0.20-38     2018-11-04 [3] CRAN (R 3.6.1)
```

```
##  lazyeval       0.2.2       2019-03-15 [1] CRAN (R 3.6.1)
##  leaflet      * 2.0.3       2019-11-16 [1] CRAN (R 3.6.1)
##  lifecycle      0.1.0       2019-08-01 [1] CRAN (R 3.6.1)
##  lubridate      1.7.4       2018-04-11 [1] CRAN (R 3.6.1)
##  magrittr       1.5         2014-11-22 [1] CRAN (R 3.6.1)
##  MASS           7.3-51.4    2019-03-31 [3] CRAN (R 3.6.1)
##  memoise        1.1.0       2017-04-21 [1] CRAN (R 3.6.1)
##  mime           0.7         2019-06-11 [1] CRAN (R 3.6.1)
##  modelr         0.1.5       2019-08-08 [1] CRAN (R 3.6.1)
##  munsell        0.5.0       2018-06-12 [1] CRAN (R 3.6.1)
##  nlme           3.1-140     2019-05-12 [3] CRAN (R 3.6.1)
##  nycflights13 * 1.0.1       2019-09-16 [1] CRAN (R 3.6.1)
##  openssl        1.4.1       2019-07-18 [1] CRAN (R 3.6.1)
##  pillar         1.4.2       2019-06-29 [1] CRAN (R 3.6.1)
##  pkgconfig      2.0.3       2019-09-22 [1] CRAN (R 3.6.1)
##  plotly         4.9.1       2019-11-07 [1] CRAN (R 3.6.1)
##  plyr           1.8.4       2016-06-08 [1] CRAN (R 3.6.1)
##  promises       1.1.0       2019-10-04 [1] CRAN (R 3.6.1)
##  purrr        * 0.3.3       2019-10-18 [1] CRAN (R 3.6.1)
##  R6             2.4.1       2019-11-12 [1] CRAN (R 3.6.1)
##  RColorBrewer   1.1-2       2014-12-07 [1] CRAN (R 3.6.1)
##  Rcpp           1.0.3       2019-11-08 [1] CRAN (R 3.6.1)
##  readr        * 1.3.1       2018-12-21 [1] CRAN (R 3.6.1)
##  readxl         1.3.1       2019-03-13 [1] CRAN (R 3.6.1)
##  reprex         0.3.0       2019-05-16 [1] CRAN (R 3.6.1)
##  reshape2       1.4.3       2017-12-11 [1] CRAN (R 3.6.1)
##  rlang          0.4.2       2019-11-24 [1] Github (r-lib/rlang@dbcb76f)
##  rmarkdown      1.17        2019-11-13 [1] CRAN (R 3.6.1)
##  rprojroot      1.3-2       2018-01-03 [1] CRAN (R 3.6.1)
##  rstudioapi     0.10        2019-03-19 [1] CRAN (R 3.6.1)
##  rvest          0.3.5       2019-11-08 [1] CRAN (R 3.6.1)
##  scales         1.1.0       2019-11-18 [1] CRAN (R 3.6.1)
##  sessioninfo    1.1.1       2018-11-05 [1] CRAN (R 3.6.1)
##  shiny          1.4.0       2019-10-10 [1] CRAN (R 3.6.1)
##  stringi        1.4.3       2019-03-12 [1] CRAN (R 3.6.1)
##  stringr      * 1.4.0       2019-02-10 [1] CRAN (R 3.6.1)
##  tibble       * 2.1.3       2019-06-06 [1] CRAN (R 3.6.1)
##  tic            0.2.13.9021 2019-11-18 [1] Github (ropenscilabs/tic@9a5f965)
##  tidyr        * 1.0.0       2019-09-11 [1] CRAN (R 3.6.1)
##  tidyselect     0.2.5       2018-10-11 [1] CRAN (R 3.6.1)
##  tidyverse    * 1.3.0       2019-11-21 [1] CRAN (R 3.6.1)
##  utf8           1.1.4       2018-05-24 [1] CRAN (R 3.6.1)
##  vctrs          0.2.0       2019-07-05 [1] CRAN (R 3.6.1)
##  viridisLite    0.3.0       2018-02-01 [1] CRAN (R 3.6.1)
##  withr          2.1.2       2018-03-15 [1] CRAN (R 3.6.1)
##  xaringan       0.13        2019-10-30 [1] CRAN (R 3.6.1)
```

```
##  xfun          0.11       2019-11-12 [1] CRAN (R 3.6.1)
##  xml2          1.2.2      2019-08-09 [1] CRAN (R 3.6.1)
##  xtable        1.8-4      2019-04-21 [1] CRAN (R 3.6.1)
##  yaml          2.2.0      2018-07-25 [1] CRAN (R 3.6.1)
##  zeallot       0.1.0      2018-01-28 [1] CRAN (R 3.6.1)
##
## [1] /home/travis/R/Library
## [2] /usr/local/lib/R/site-library
## [3] /home/travis/R-bin/lib/R/library
```

License

Chapter 1

Introduction

The `tidyverse` has quickly developed over the last years. Its first implementation as a collection of partly older packages was in the second half of 2016. All its packages "share an underlying design philosophy, grammar, and data structures."[1] It is for sure difficult to tell, if "learning the `tidyverse`" is a hard task, since the result of this assessment might differ from person to person. We do believe though, that there are concepts in its approach, which – when grasped – have the potential to increase one's productivity, since code creation will seem more natural. While this might be true for all languages (once you speak it well enough, things go smoothly), in our opinion the `tidyverse` worth exploring in depth, since it is

1. consistent: an especially well designed framework that aims at making data analysis and programming intuitive,
2. evolving: constantly deepened understanding for challenges arising in modern data analysis leads to improving ergonomic user interfaces.

1.1 Speakers

Kirill Müller (@krlmlr)

Patrick Schratz (@pat-s)

- M.Sc. Geoinformatics
- Researcher/Research Engineer at University of **Jena** and **LMU Munich**
- PhD Candidate

- Unix & R enthusiast

[1] citation from tidyverse homepage

Figure 1.1: Patrick Schratz

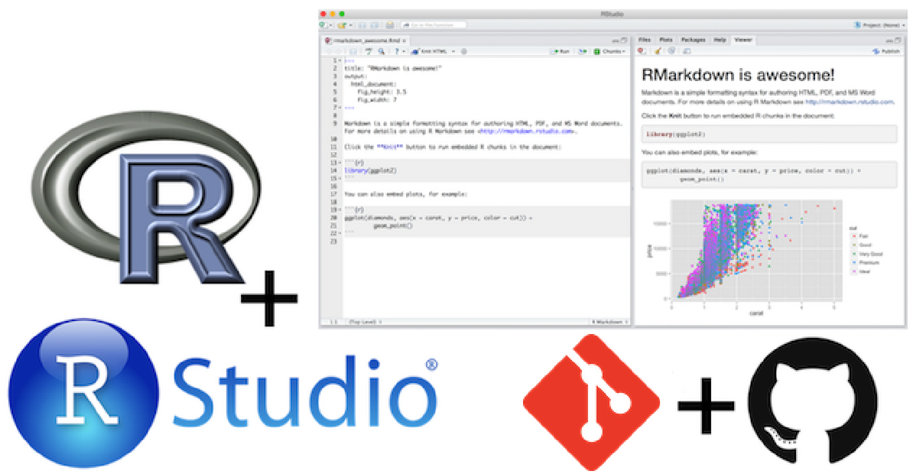


Figure 1.2: R as a toolkit

- Author/Contributor/Maintainer of several R packages:
  - (mlr3, mlr)
  - sperrorest
  - oddsratio
  - xaringan
  - circle
  - RQGIS
  - travis
  - tic
  - …

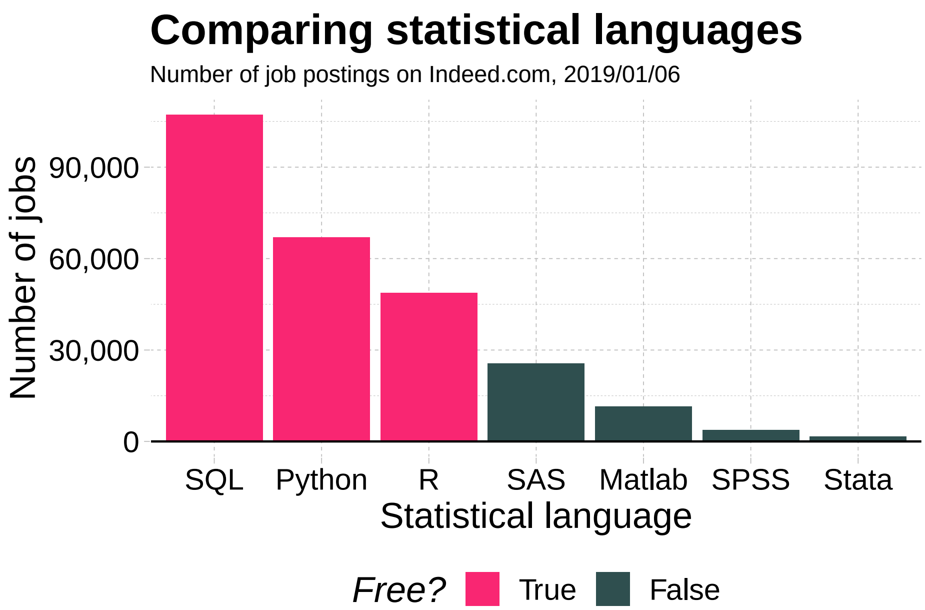
1.2 Overview

This course covers several topics, which everyone working more intently with the `tidyverse` almost inevitably needs to deal with at some point or another. The topics are organized in chapters that contain mostly R code with output and text. In each section, exercises are provided.

1.3 R as a toolkit

- Scriptability → R
- Literate programming (code, narrative, output in one place) → R Markdown
- Version control → Git / GitHub

1.3.1 Why R and RStudio?



1.3.2 Some R basics

- You will load packages at the **start of every new R session**.
  - "Base" R comes with tons of useful built-in functions. It also provides all the tools necessary for you to write your own functions.
  - However, many of R's best data science functions and tools come from external packages written by other users.
- R easily and infinitely parallelizes. For free.
  - Compare the cost of a Stata/MP license, nevermind the fact that you effectively pay per core…

1.4 R code examples

1.4.1 Linear regression

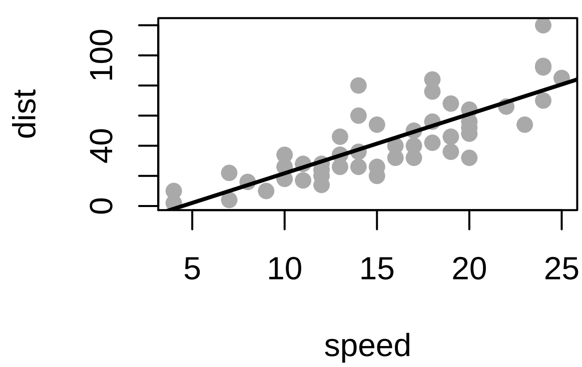
```
fit <- lm(dist ~ 1 + speed, data = cars)
summary(fit)
```

```
##
```

```
## Call:
## lm(formula = dist ~ 1 + 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 ***
## ---
## 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
```

1.4.2 Base R plot

```r
par(mar = c(4, 4, 1, .1)) ## nice plot margins
plot(cars, pch = 19, col = 'darkgray')
abline(fit, lwd = 2)
```



1.4.3 ggplot2

```r
library(ggplot2)
library(gapminder) ## For the gapminder data

ggplot(data = gapminder,
```

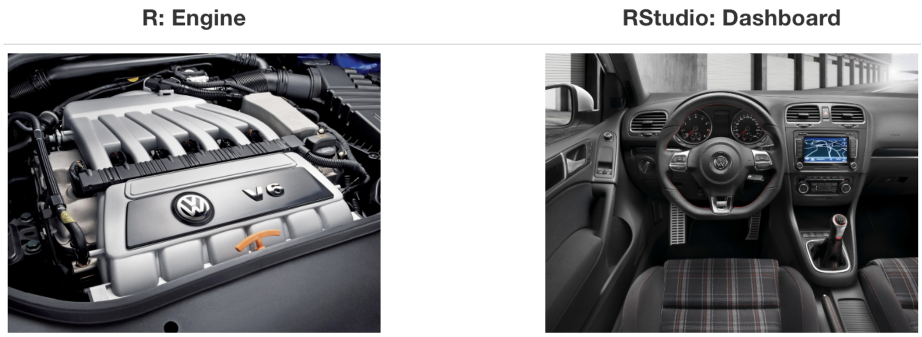
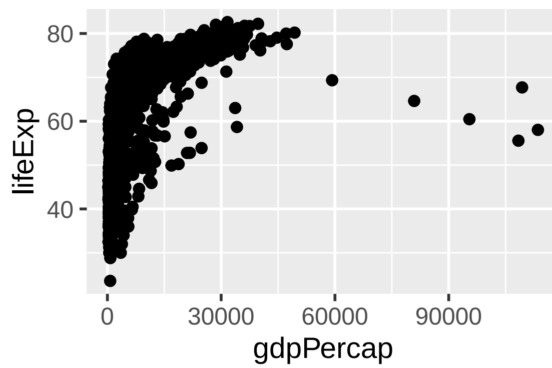


Figure 1.3: Engine vs. dashboard

```
      mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```



1.4.4 gganimate

1.5 R vs. RStudio

- R is a statistical **programming language**
- RStudio is a convenient interface for R (an **integrated development environment**, IDE)
- At its simplest:
  - R is like a car's engine
  - RStudio is like a car's dashboard

R: A new phone	R Packages: Apps you can download
	

Figure 1.4: R versus R packages

1.6 R vs. R packages

- R packages **extend** the functionality of R by providing additional functions, data, and documentation.
- They are written by a world-wide community of R users and can be downloaded for no cost

1.7 R packages

- **CRAN**: A group of people who check that packages fulfill certain standards
- **Mirror**: A location on the web where to download R packages from. Because many thousand people download them daily, the load is distributed on different machines. Pick one which is geographically close to you
- **R base/recommended packages**: The base installation of R ships with a bunch of default packages. In addition, there are some more packages listed as "recommended".

"base" packages are managed by the R core team and will only be updated for every R release.

Packages listed as "recommended" inherit the attributes of being widely used and having a long history in the R community.

```
##     Package Priority
## 1      base     base
## 2  compiler     base
```

```
## 3  datasets     base
## 4  graphics     base
## 5 grDevices     base
## 6      grid     base
## 7   methods     base
## 8  parallel     base
```

```
##       Package    Priority
## 1        boot recommended
## 2       class recommended
## 3     cluster recommended
## 4   codetools recommended
## 5     foreign recommended
## 6  KernSmooth recommended
## 7     lattice recommended
## 8        MASS recommended
## 9      Matrix recommended
## 10       mgcv recommended
##  [ reached 'max' / getOption("max.print") -- omitted 2 rows ]
```

1.8 .Rprofile

- File in your home directory `~/.Rprofile`
- Will be executed before every R session starts
- Useful to set global options and for loading of often used packages

1.9 .Renviron

- File in your home directory `~/.Renviron`
- Used to set environment variables
- Used to store "Access tokens" (Github, CI provider, C++ flags)

1.10 RStudio

→ Exists to **boost** your productivity

→ Change the defaults to your liking so you *actually* can be **productive**

→ Keybindings = productivity

Since RStudio v1.3 a portable JSON settings file exists.

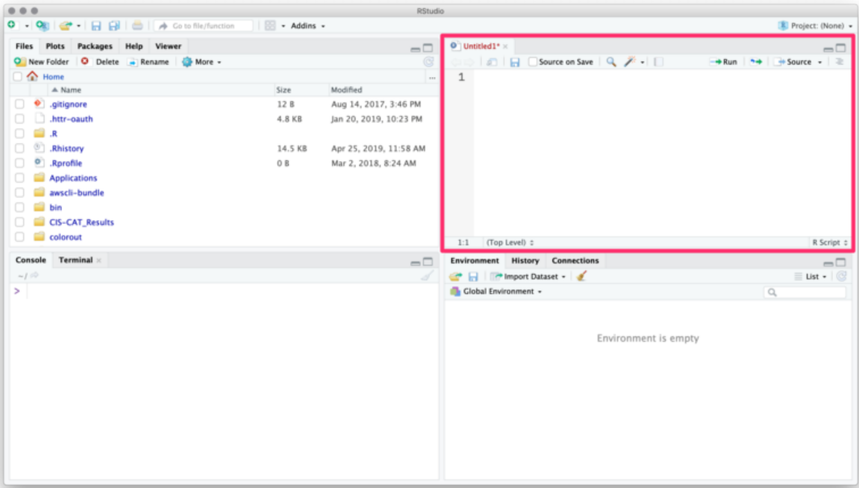


Figure 1.5: Source pane

If you want to have sane settings without much hassle, you can execute the following R code: `source("https://bit.ly/rstudio-pat")`

This code will change/overwrite your existing RStudio settings and

- set custom keybindings
- move the console panel to the top-right (by default bottom-left)
- Enable/Disable some core settings to have a better overall experience

---

R scripts (source code) are written in the *Source* pane (Editor).

(Source of all following RStudio screenshots: https://github.com/edrubin/EC525S19)

You can use the menubar or ++N / +CTRL+N to create new R scripts.

To execute commands from your R script, use +Enter / CTRL+Enter.

RStudio will execute the command in the console.

You can see the new object in the *Environment* pane.

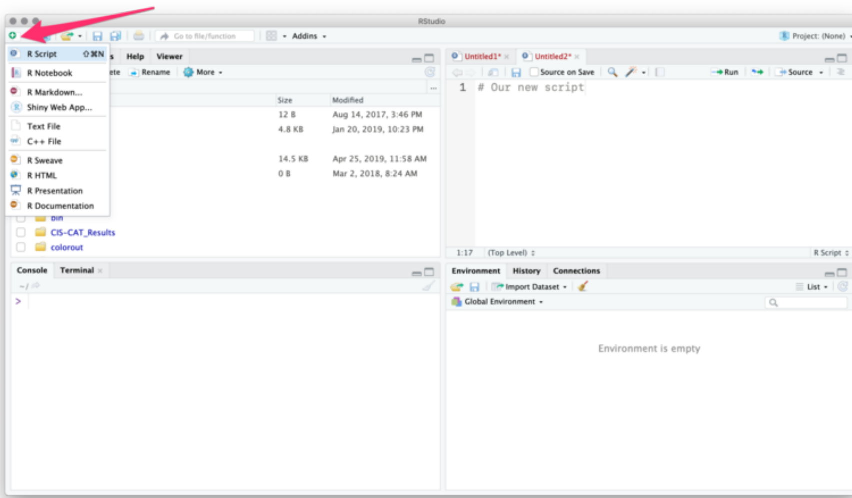


Figure 1.6: New script

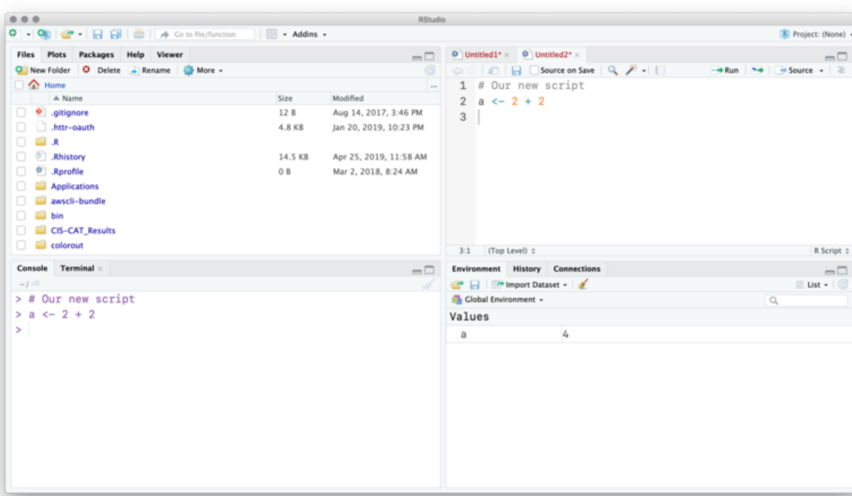


Figure 1.7: Execute commands

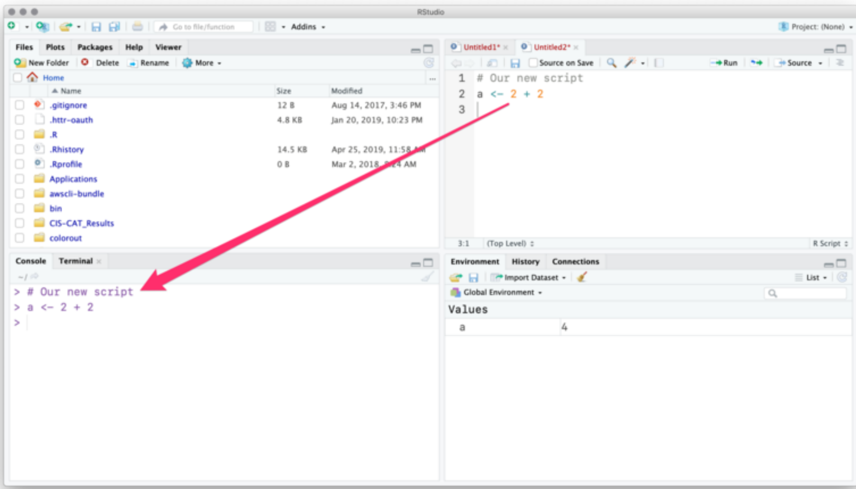


Figure 1.8: Console output

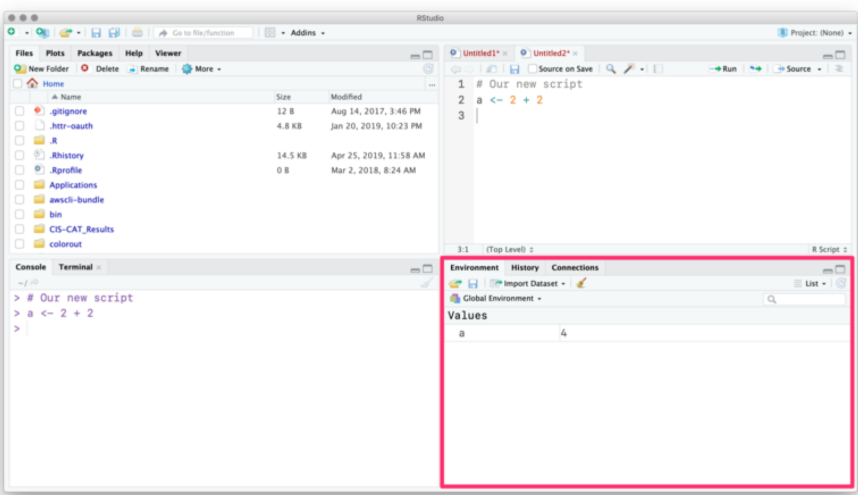


Figure 1.9: Environment pane

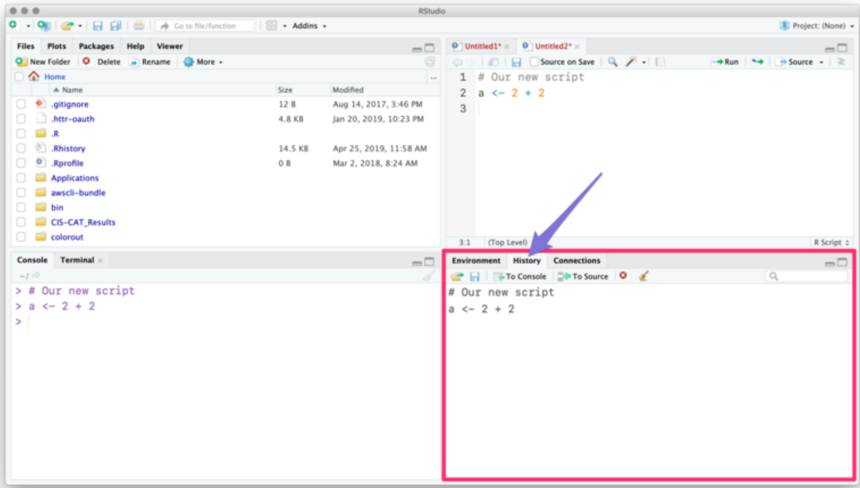


Figure 1.10: History pane

The *History* tab records your old commands.

The *Files* pane is the file explorer.

The *Plots* pane/tab shows... plots.

Packages shows installed packages

Packages shows installed packages and whether they are *loaded*.

The *Help* tab shows help documentation (also accessible via `?`).

Finally, you can customize the actual layout

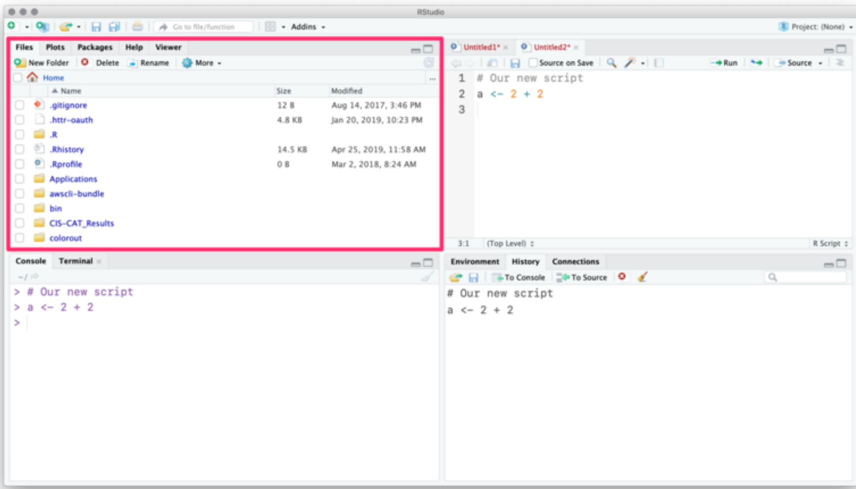


Figure 1.11: Files pane

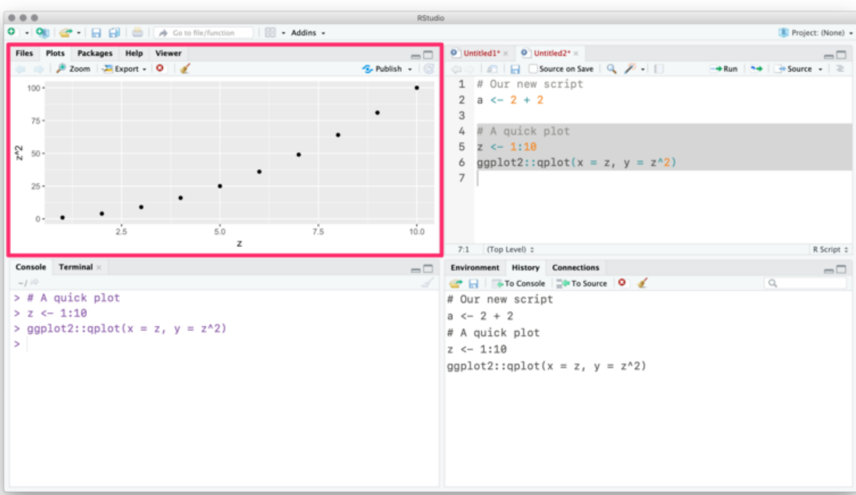


Figure 1.12: Plots pane

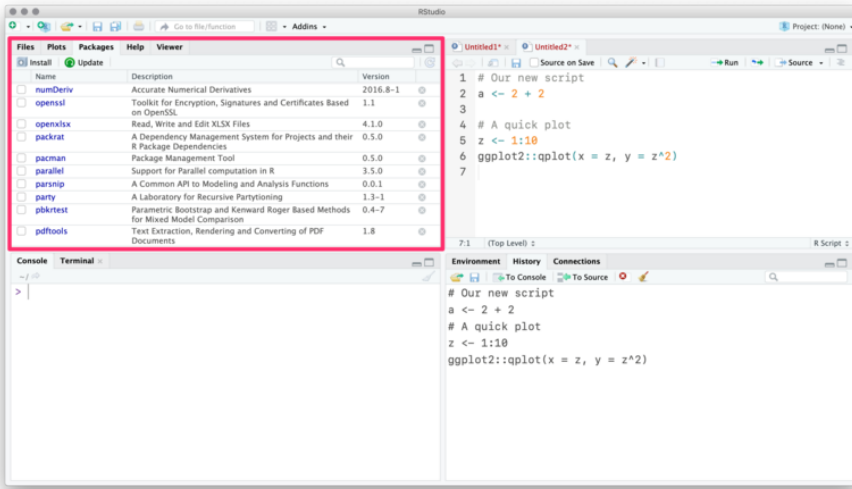


Figure 1.13: Packages pane

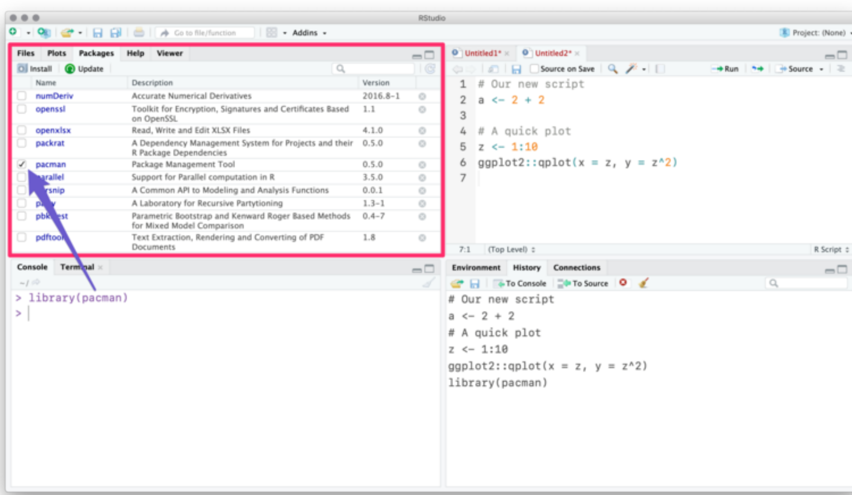


Figure 1.14: Loaded and installed packages

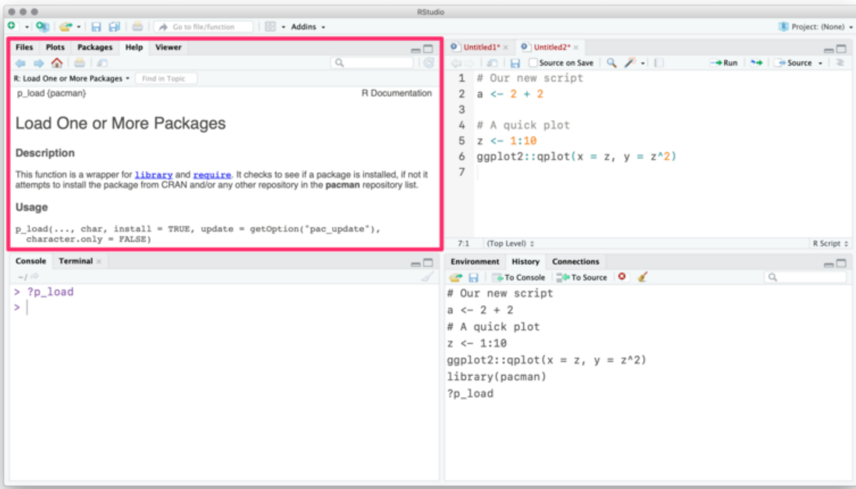


Figure 1.15: Help pane

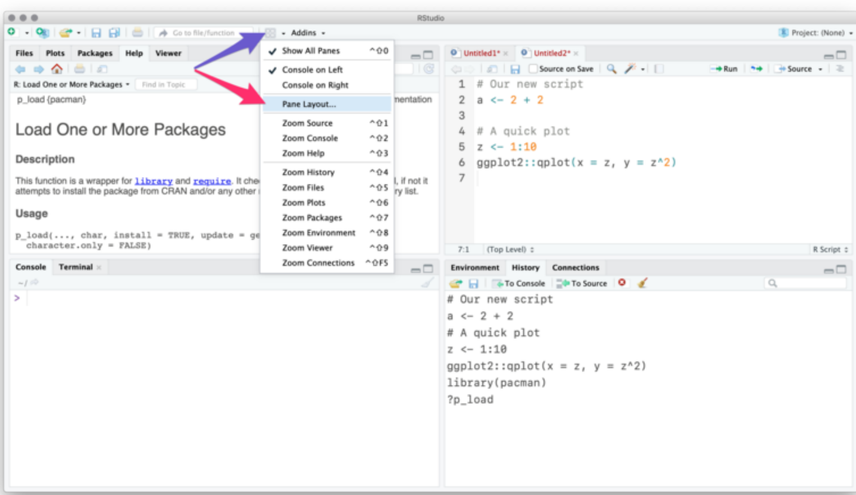


Figure 1.16: Customize layout

1.11 RStudio Addins

RStudio can be further enhanced by so called “addins”. These are clickable snippets that execute certain actions in RStudio.

They aim to make repetitive tasks easier and to save you time. There is an addin called addinslist which lists all available addins. It can be installed as a normal package from CRAN:

```
install.packages("addinslist")
```

To have an addin available in RStudio after installation, RStudio needs to be restarted.

1.12 RStudio projects

Without a project, you will need to define **long** file paths which **only exist on your machine**.

```
sample_df <- read.csv("/Users/<yourname>/somewhere/on/this/machine/sample.csv")
```

With a project, R automatically references the project’s folder as the current working directory.

From there on, you can use *relative paths* to point to files.

```
sample_df <- read.csv("sample.csv")
```

Double-plus bonus: The *here* package extends *RStudio project* philosophy even more and helps in cases when not using RStudio (e.g. on the command line).

1.13 Alternatives to RStudio

- Using R directly in the terminal via radian (optimized R console interpreter)
- R is supported in other “general purpose IDE’s” (VScode, Sublime Text, Atom, Vim, etc.)

Chapter 2

Visualization Basics {ggplot2}

> Embracing the grammar of graphics.

This chapter discusses plotting with the ggplot2 package.

2.1 Introduction

Click here to show setup code.

```
library(tidyverse)
```

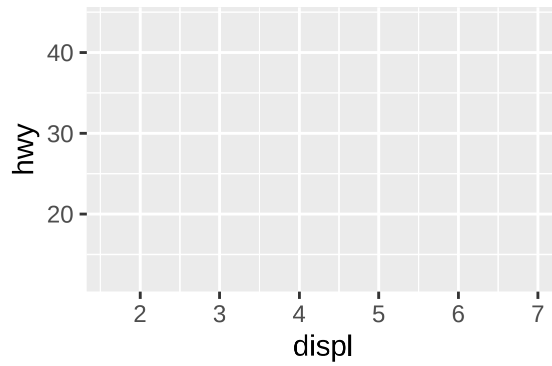
In the {tidyverse} the standard package for visualization is {ggplot2}. The functions of this package follow a quite unique logic (the "grammar of graphics") and therefore require a special syntax. In this section we want to give a short introduction, how to get started with {ggplot2}.

2.1.1 Creating the plot skeleton: `ggplot()`

The main function in the package is `ggplot()`, which prepares/creates a graph. By setting the arguments of the function, you can:

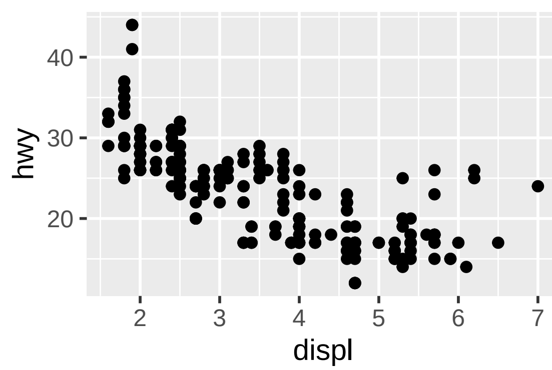
1. Choose the dataset to be plotted (argument `data`)
2. Choose the mapping of the variables to the axes (or further forms of setting apart data) in the argument `mapping`. This argument takes the result of the function `aes()`, which you will get to know in many different examples.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
)
```



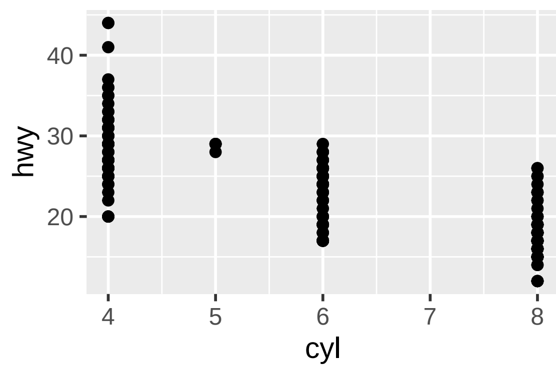
This created only an empty plot, because we did not tell {ggplot2} which geometry we want to use to display the variables we set in the `ggplot()` call. We do this by adding (with the help of the `+` operator after the `ggplot()`-call) a different function starting with `geom_` to provide this information.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point()
```



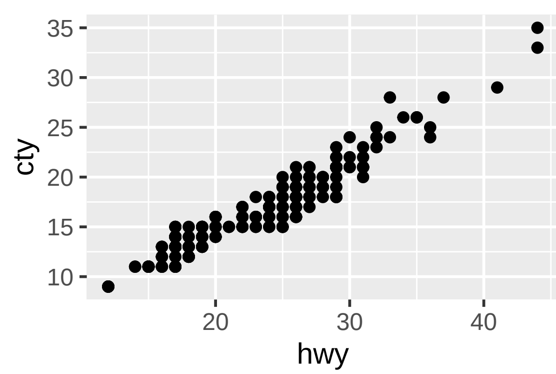
This is maybe the most basic plot you can create. To map a different variable than `disp` to the x-axis, change the respective variable name in the `aes()` argument.

```
ggplot(
  data = mpg,
  mapping = aes(x = cyl, y = hwy)
) +
  geom_point()
```



You can exchange the variables to be plotted freely, without changing anything else to the rest of the code.

```
ggplot(
  data = mpg,
  mapping = aes(x = hwy, y = cty)
) +
  geom_point()
```



Always good to have: The *ggplot2* cheatsheet (https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf).

2.1.2 What is a "statistical graphic"?

Wilkinson (2005) defines a grammar to describe the basic elements of a statistical graphic:

> "[…] a statistical graphic is a mapping from data to aesthetic attributes (colour, shape, size) of geometric objects (points, line, bars)."
>
> (Wickham, 2009)

2.1.3 Terminology

- **Data:** The data to visualize – consists of variables and observations.
- **Geoms:** Geometric objects which represent the data (points, lines, polygons, etc.).
- **Mappings:** Match variables with aesthetic attributes of the (geometric) objects.
- **Scales:** Mapping of the "data units" to "physical units" of the geometric objects (e.g. length, diameter or color); defines the *legend*.
- **Coord:** System of coordinates, mapping of the data to a two dimensional plain of the graphic; defines the *axes* and *grid*.
- **Stats:** Statistical transformation of the data (5 point summary, classification, etc.).
- **Facetting:** Division and illustration of data subsets, also known as "Trellis" images.

2.1.4 The Grammar of graphics …

is …

a formal guideline which describes the dependencies between all elements of a statistical graphic.

isn't …

- a manual which tells us *which graphic* should be created for a given question.
- a specification *how* a statistical graphic should look like.

2.1.5 About {ggplot2}

```
## Package: ggplot2
## Version: 3.2.1
```

```
## Title: Create Elegant Data Visualisations Using the Grammar of Graphics
## Depends: R (>= 3.2)
## Imports: digest, grDevices, grid, gtable (>= 0.1.1), lazyeval, MASS, mgcv,
##          reshape2, rlang (>= 0.3.0), scales (>= 0.5.0), stats, tibble,
##          viridisLite, withr (>= 2.0.0)
## License: GPL-2 | file LICENSE
## URL: http://ggplot2.tidyverse.org, https://github.com/tidyverse/ggplot2
## BugReports: https://github.com/tidyverse/ggplot2/issues
## Encoding: UTF-8
## Author: Hadley Wickham [aut, cre], Winston Chang [aut], Lionel Henry [aut],
##          Thomas Lin Pedersen [aut], Kohske Takahashi [aut], Claus Wilke [aut],
##          Kara Woo [aut], Hiroaki Yutani [aut], RStudio [cph]
## Maintainer: Hadley Wickham <hadley@rstudio.com>
##
## -- File:
```

2.2 `geom_*` functions

`geom_*` functions are added to the main `ggplot()` call via the "+" operator and (usually) placed on a new line.

A list of all available "geoms" can be found here:

https://ggplot2.tidyverse.org/reference/#section-layer-geoms

The most popular ones are

- `geom_point()`
- `geom_histogram()`
- `geom_boxplot()`
- `geom_bar()`

The `geom_*` family can be divided into three parts:

One variable plots

- `geom_hist()`
- `geom_bar()`
- etc.

Two variable plots

- `geom_point()`
- `geom_line()`
- `geom_boxplot()`

- etc.

Three variables plots

- `geom_raster()`
- `geom_sf()`
- `geom_tile()`
- etc.

2.2.1 Arguments

```
ggplot(data, mapping = aes(), ...) +
  geom_XXX(mapping = NULL, data = NULL, stat, ...)
```

`geom_*` functions have the same basic arguments as `ggplot()`. In addition, they come with more arguments specific to the respective "geom".

stat

The `stat` parameter defines a statistical transformation:

- if set to `"identity"`: No transformation
- if set to `boxplot`: Boxplot transformation
- etc.

position

The same applies to the `position` argument. In the example below, points are not adjusted and just visualized where they appear in the data.

In the case of boxplots, a special position arrangement function is used to arrange everything nicely: `position_dodge2()` (here denoted by `position = "dodge2"`).

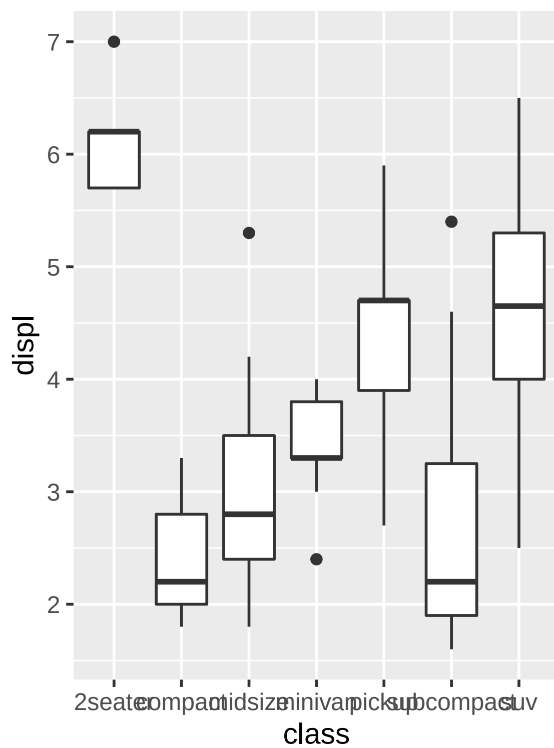
```
geom_point(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", ..., na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)

geom_boxplot(mapping = NULL, data = NULL, stat = "boxplot",
  position = "dodge2", ..., outlier.colour = NULL,
  outlier.color = NULL, outlier.fill = NULL, outlier.shape = 19,
  outlier.size = 1.5, outlier.stroke = 0.5, outlier.alpha = NULL,
  notch = FALSE, notchwidth = 0.5, varwidth = FALSE, na.rm = FALSE,
  show.legend = NA, inherit.aes = TRUE)
```

`geom_boxplot()` needs one variable to be of class `character` or `factor` (better) to initiate the grouping.

```
class(mpg$class)
```

```
## [1] "character"
```

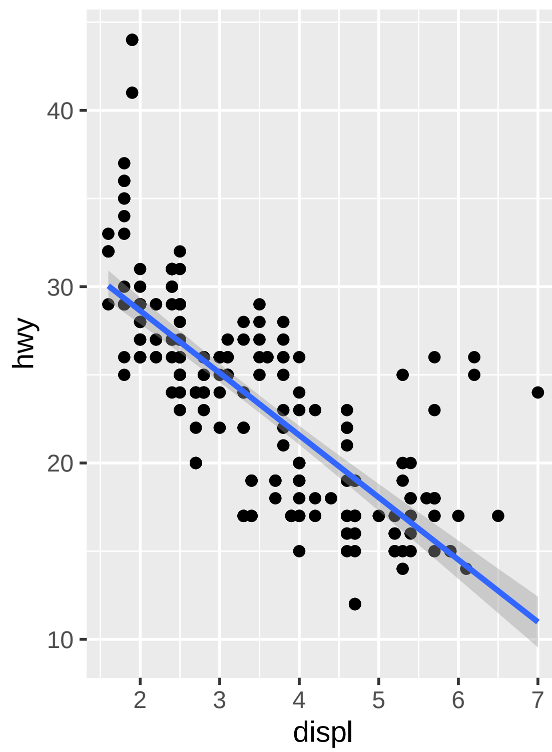
```
ggplot(mpg, aes(x = class, y = displ)) +
  geom_boxplot()
```



2.2.2 Combining geoms

Multiple `geom_*` functions can be used in one plot. A combination that is often used together is `geom_point()` and `geom_smooth()`

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(method = "lm")
```



Unless specified differently in the `geom_*()` call, all geoms will use the same variables.

2.2.3 Summary

The modular principle of `ggplot2` enables:

- the combination of any geometric objects (geoms).
- a high flexibility and customizability

An extensive description of all geometric objects can be found on the `ggplot2` website https://ggplot2.tidyverse.org/reference/.

2.3 Export & saving

The default way to export plots in {{ggplot2}} is by using `ggsave()`.

It differs slightly from other "exporting" functions in R because it comes with some smart defaults:

> ggsave() is a convenient function for saving a plot. It defaults to **saving the last plot** that you displayed, using the size of the current graphics device. It also **guesses the type** of graphics device from the extension.

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

```
ggsave("mtcars.pdf")
```

```
## Saving 3 x 2 in image
```

```
ggsave("mtcars.png")
```

```
## Saving 3 x 2 in image
```

This might seem natural to you but is is not. Let's compare base R and {{ggplot2}}.

2.3.1 Base R vs. {{ggplot2}}

In base R

- one needs to open a specific graphic device first
- then create the plot
- and close the graphic device again.

```
png("Plot.png")
plot(mpg$displ, mpg$hwy)
dev.off()
```

```
ggplot(mpg, aes(disply, hwy)) +
  geom_point()
ggsave("Plot.png")
```

Base R plotting functions come with suboptimal defaults

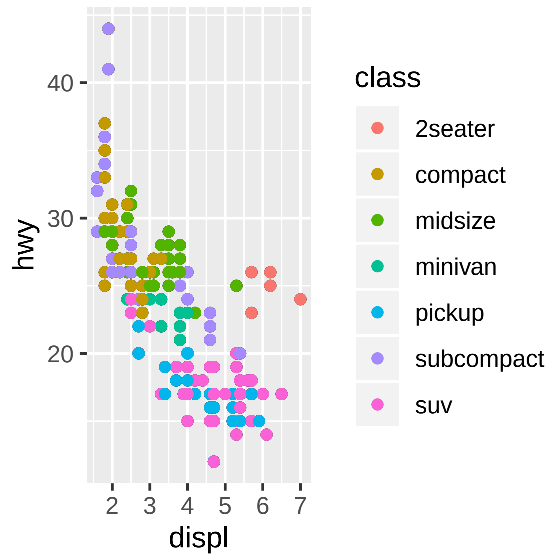
- saving in pixels (differs on every monitors)
- saving as a square image
- no option to specify the DPI (dots per inch)

2.3.2 Storing the plot as an R object

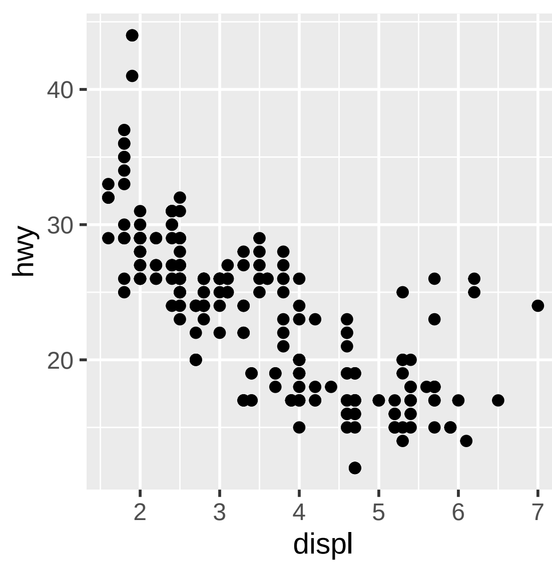
One of the major advantages of `ggplot()` is that you can save a plot as an R object and modify it later.

This is not possible with base R plots.

```
p <- ggplot(mpg, aes(displ, hwy)) +
  geom_point()

p + geom_point(aes(color = class))
```




```
print(p)
```



```r
str(p)
```

```
## List of 9
##  $ data       :Classes 'tbl_df', 'tbl' and 'data.frame': 234 obs. of  11 variables:
##   ..$ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
##   ..$ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
##   ..$ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##   ..$ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##   ..$ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
##   ..$ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##   ..$ drv         : chr [1:234] "f" "f" "f" "f" ...
##   ..$ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
##   ..$ hwy         : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
##   ..$ fl          : chr [1:234] "p" "p" "p" "p" ...
##   ..$ class       : chr [1:234] "compact" "compact" "compact" "compact" ...
##  $ layers     :List of 1
##   ..$ :Classes 'LayerInstance', 'Layer', 'ggproto', 'gg' <ggproto object: Class LayerInstance,
##     aes_params: list
##     compute_aesthetics: function
##     compute_geom_1: function
##     compute_geom_2: function
##     compute_position: function
##     compute_statistic: function
##     data: waiver
##     draw_geom: function
##     finish_statistics: function
##     geom: <ggproto object: Class GeomPoint, Geom, gg>
##         aesthetics: function
##         default_aes: uneval
##         draw_group: function
##         draw_key: function
##         draw_layer: function
##         draw_panel: function
##         extra_params: na.rm
##         handle_na: function
##         non_missing_aes: size shape colour
##         optional_aes:
##         parameters: function
##         required_aes: x y
##         setup_data: function
##         use_defaults: function
##         super:  <ggproto object: Class Geom, gg>
##     geom_params: list
##     inherit.aes: TRUE
##     layer_data: function
```

```
##     map_statistic: function
##     mapping: NULL
##     position: <ggproto object: Class PositionIdentity, Position, gg>
##         compute_layer: function
##         compute_panel: function
##         required_aes:
##         setup_data: function
##         setup_params: function
##         super:  <ggproto object: Class Position, gg>
##     print: function
##     setup_layer: function
##     show.legend: NA
##     stat: <ggproto object: Class StatIdentity, Stat, gg>
##         aesthetics: function
##         compute_group: function
##         compute_layer: function
##         compute_panel: function
##         default_aes: uneval
##         extra_params: na.rm
##         finish_layer: function
##         non_missing_aes:
##         parameters: function
##         required_aes:
##         retransform: TRUE
##         setup_data: function
##         setup_params: function
##         super:  <ggproto object: Class Stat, gg>
##     stat_params: list
##     super:  <ggproto object: Class Layer, gg>
##  $ scales     :Classes 'ScalesList', 'ggproto', 'gg' <ggproto object: Class ScalesLi
##     add: function
##     clone: function
##     find: function
##     get_scales: function
##     has_scale: function
##     input: function
##     n: function
##     non_position_scales: function
##     scales: list
##     super:  <ggproto object: Class ScalesList, gg>
##  $ mapping    :List of 2
##   ..$ x: language ~displ
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   ..$ y: language ~hwy
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   ..- attr(*, "class")= chr "uneval"
```

```
## $ theme      : list()
## $ coordinates:Classes 'CoordCartesian', 'Coord', 'ggproto', 'gg' <ggproto object: Class Coord
##     aspect: function
##     backtransform_range: function
##     clip: on
##     default: TRUE
##     distance: function
##     expand: TRUE
##     is_free: function
##     is_linear: function
##     labels: function
##     limits: list
##     modify_scales: function
##     range: function
##     render_axis_h: function
##     render_axis_v: function
##     render_bg: function
##     render_fg: function
##     setup_data: function
##     setup_layout: function
##     setup_panel_params: function
##     setup_params: function
##     transform: function
##     super:  <ggproto object: Class CoordCartesian, Coord, gg>
## $ facet      :Classes 'FacetNull', 'Facet', 'ggproto', 'gg' <ggproto object: Class FacetNull,
##     compute_layout: function
##     draw_back: function
##     draw_front: function
##     draw_labels: function
##     draw_panels: function
##     finish_data: function
##     init_scales: function
##     map_data: function
##     params: list
##     setup_data: function
##     setup_params: function
##     shrink: TRUE
##     train_scales: function
##     vars: function
##     super:  <ggproto object: Class FacetNull, Facet, gg>
## $ plot_env   :<environment: R_GlobalEnv>
## $ labels     :List of 2
##   ..$ x: chr "displ"
##   ..$ y: chr "hwy"
## - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

2.3.3 Best practices for exporting

Some best practices:

- Use a reasonable high DPI. A value of "300" is ok in most cases.
- Save in "inches" and not in "pixels". The latter always differs on screens with different resolutions (`png()` uses pixels by default.)
- Always specify a file name to ensure the right plot is chosen. Do not rely on the default behavior of `ggsave()` (even though it might seem convenient) which takes the last visualized plot.
- An alternative to `ggsave()` is `cowplot::save_plot()` which comes with sensible defaults for multi-plot arrangements.

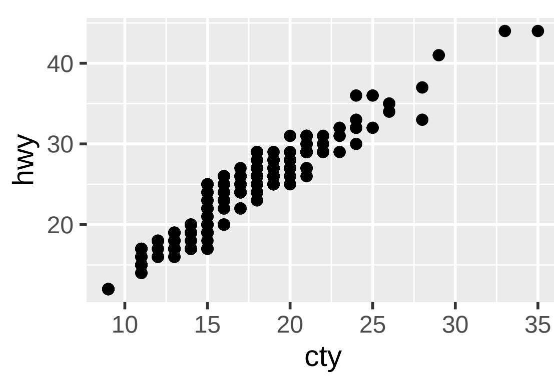
2.4 Two variable plots

"Two variable plots" can be split into sub-categories:

- Continuous X and Y
- Continuous X and discrete Y (and vice-versa)
- Discrete X and Y

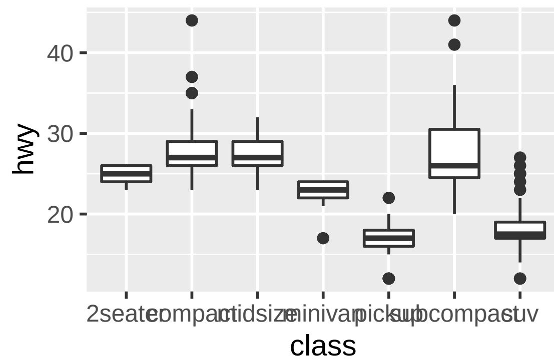
2.4.1 Continuous X and Y

```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point()
```



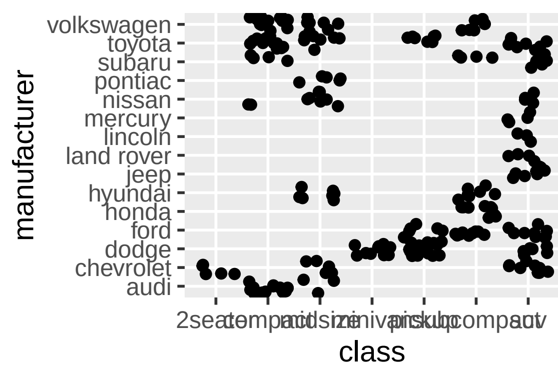
2.4.2 Discrete X and continuous Y

```
ggplot(mpg, aes(x = class, y = hwy)) +
  geom_boxplot()
```



2.4.3 Discrete X and Y

```
ggplot(mpg, aes(x = class, y = manufacturer)) +
  geom_jitter()
```



2.5 One variable plots

This type of plots visualizes ONE variable in a certain way.

To do this in a 2D space, a **statistical transformation** of the variable is required for the missing axis.

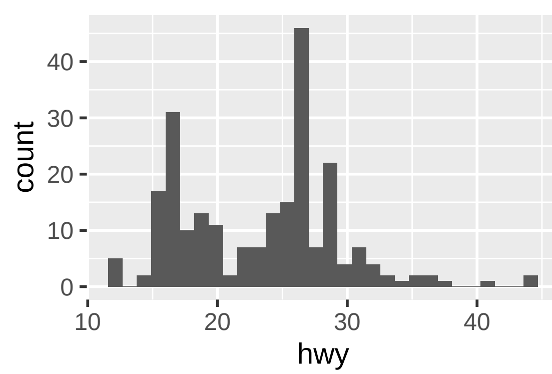
2.5.1 Continuous variables

- Histogram: Most common way - grouping the variable into equal bins
- `geom_density()`, `geom_freq()`, `geom_dotplot()` and `geom_area()` are mainly doing the same as `geom_hist()`

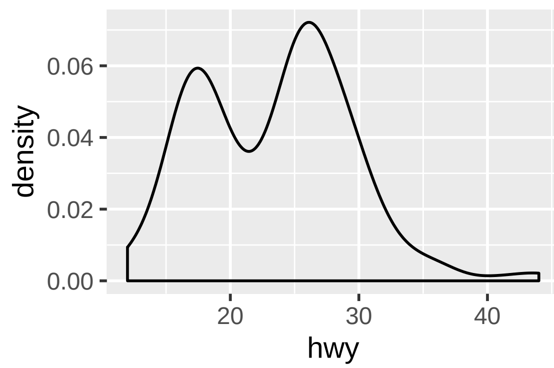
We supply only *one* variable to the `mapping` argument with the help of `aes()`. This one is automatically grouped into 30 bins.

```
ggplot(mpg, aes(x = hwy)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with
## `binwidth`.
```



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

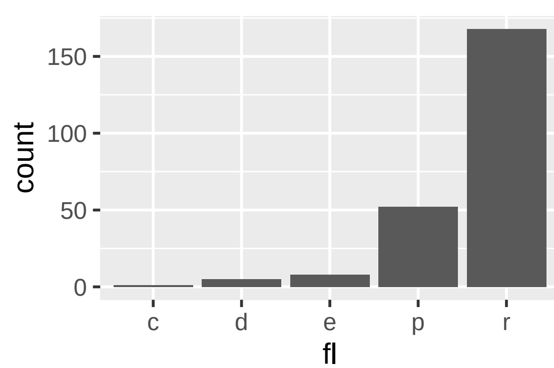


2.5.2 Discrete variables

For discrete data, there is actually only one visualization method - the bar plot.

Note the difference of `geom_bar()` compared to `geom_hist()`.

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



2.6 Colors and shape

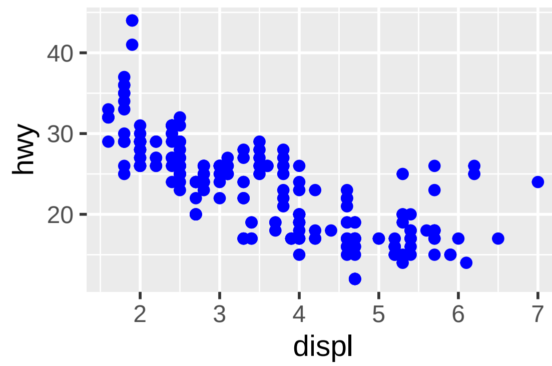
Click here to show setup code.

```
library(tidyverse)
```

2.6.1 Static colors

There are many ways to set a color for a specific geom. The simplest is to set all observations of a geom to a dedicated color, supplied as a character value.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(
    color = "blue"
  )
```



2.6.2 Dynamic colors

Dynamic colors, which depend on a variable of the dataset, need to be passed within an `aes()` call. A direct specification like in the example above with `color = "blue"` only works for static colors.

Good to know: While it is possible to include `color = class` directly in the `aes()` call of the `ggplot()` function, it is recommended to set it within the particular geom. This is for two reasons:

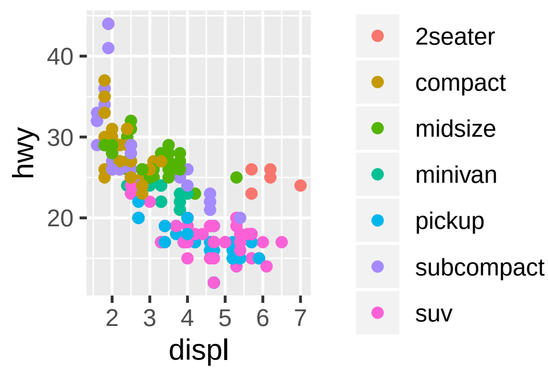
- When working with multiple geoms, you can use different mappings for each geom without any possibility of conflicts
- When reading the code, it becomes more clear which settings apply to which geoms

Discrete

Different colors can be mapped to the values of a variable by supplying a variable of the dataset. The `class` variable is discrete and leads to a discrete color scale.

```
ggplot(
  data = mpg,
```

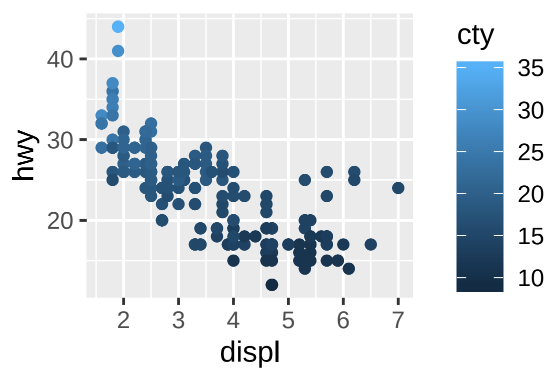
```
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(aes(color = class))
```



Continuous

The `cty` attribute is continuous, the color scale is adapted accordingly.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(aes(color = cty))
```

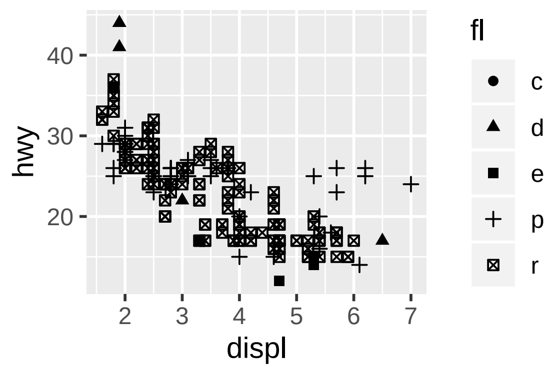


2.6.3 Shape

One more degree of freedom is the shape of the symbols to be plotted.

```
ggplot(
  data = mpg,
```

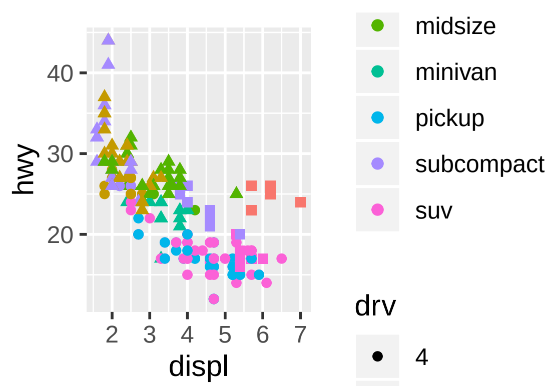
```
  mapping = aes(
    x = displ,
    y = hwy
  )
) +
  geom_point(aes(shape = fl))
```



2.6.4 Combining color and shape

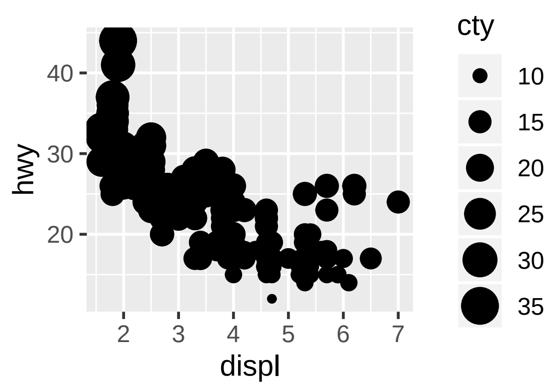
Color and shape can be combined.

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy,
  )
) +
  geom_point(aes(color = class, shape = drv))
```



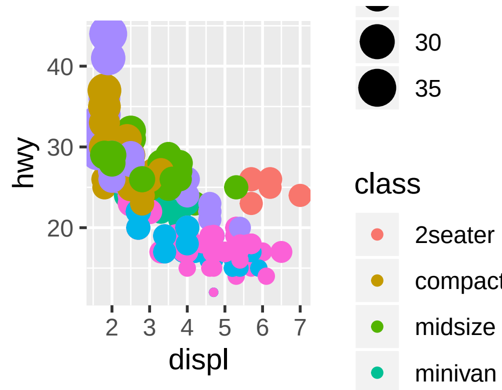
And last but not least, the size of the plotted symbols can be linked to numeric values of the mapped variable.

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy,
    size = cty
  )
) +
  geom_point()
```



You can mix different aesthetic mappings in order to produce a plot with densely packed information. However, be aware that adding too much information to a plot does not necessarily make it better.

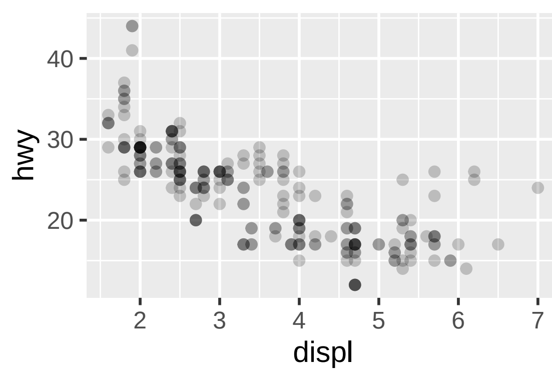
```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy,
    color = class,
    size = cty
  )
) +
  geom_point()
```



2.6.5 Transparency

Semi-transparency is another way to better display your data when observations are overlapping. This is useful to get an impression of how many data points share the same coordinates.

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy
  )
) +
  geom_point(alpha = 0.2)
```



2.6.6 What can go wrong

If you try to specify a color in the `mapping`-argument of the main `ggplot()` call, you will face an error since a mapping of a variable to an aesthetic is expected.

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy,
    color = blue
  )
) +
  geom_point()
```

```
## Error in FUN(X[[i]], ...): object 'blue' not found
```

R treats objects without quotation marks in a special way, expecting them to be variables. Since `blue` is not a variable of `mpg`, this did not work. Use quotation marks if you mean a string, as opposed to a variable or object name.

```
mpg
```

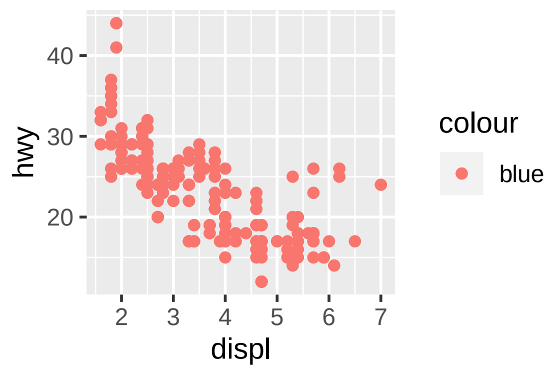
```
## # A tibble: 234 x 11
##   manufacturer model displ  year   cyl trans drv     cty   hwy
##   <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int>
## 1 audi         a4      1.8  1999     4 auto~ f        18    29
## 2 audi         a4      1.8  1999     4 manu~ f        21    29
## 3 audi         a4      2    2008     4 manu~ f        20    31
## # ... with 231 more rows, and 2 more variables: fl <chr>,
## #   class <chr>
```

```
"mpg"
```

```
## [1] "mpg"
```

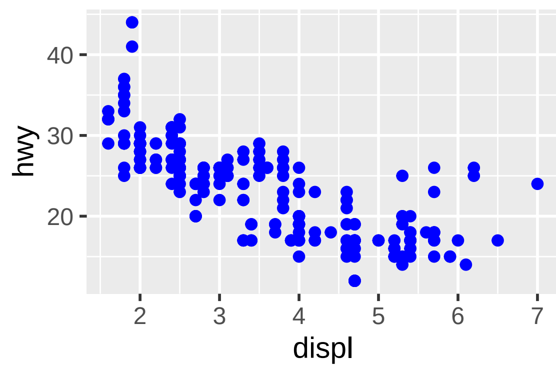
So what if we pass the color as a character variable?

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy,
    color = "blue"
  )
) +
  geom_point()
```



At least there was no error, but now the constant value `blue` is mapped to the first default color of the color mapping, which happens to be red. We could have been fooled, if it had been blue. Recall, it is best to specify geom related mappings with the respective geom function.

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy
  )
) +
  geom_point(
    color = "blue"
  )
```



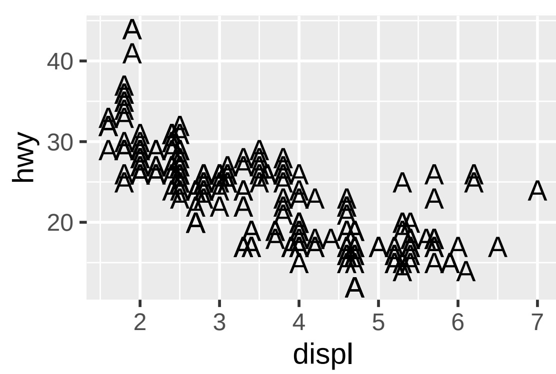
2.7 Labels

Click here to show setup code.

```
library(tidyverse)
```

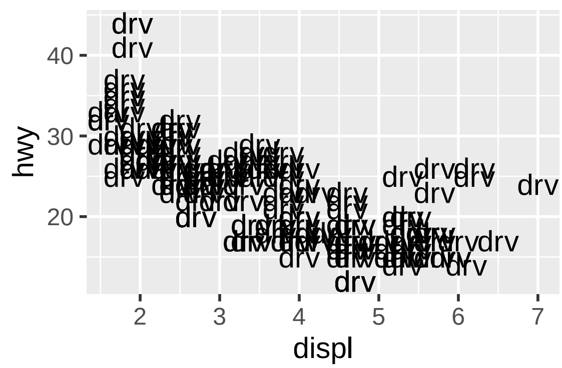
For character variables there is further way of integrating its value to a plot. `geom_text()` takes a `label` argument, which influences the plot in the following way.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_text(label = "A")
```



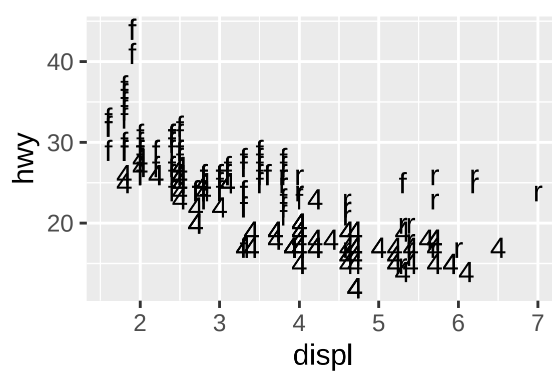
Let's try to map this argument to a variable (here: `drv`) of our dataset in the `mapping` argument of `ggplot()`.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_text(label = "drv")
```



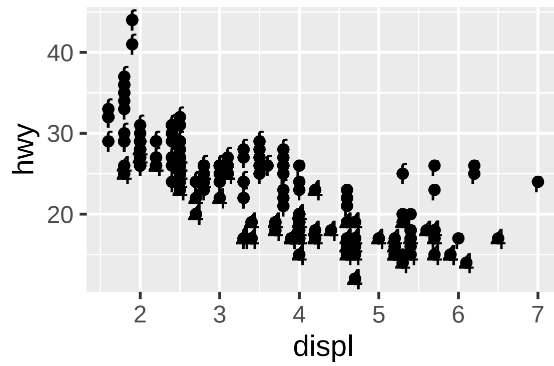
Right, of course we need to pass the variable without quotation marks, otherwise it is interpreted as a (constant) character variable. When changing this, a vector with the values of the variable is passed on to `geom_text()`. This is one way of including the values of character variables in a plot.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_text(aes(label = drv))
```



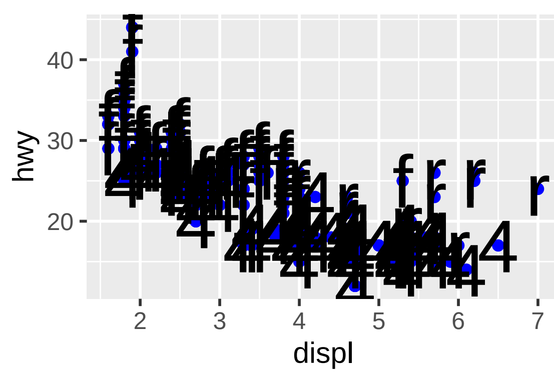
When adding more than one `geom()`-function, multiple geometries are added to the plot. However, because `geom_point()` has no support for passing a label, we can only use this mapping in `geom_text()`.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point() +
  geom_text(aes(label = drv))
```



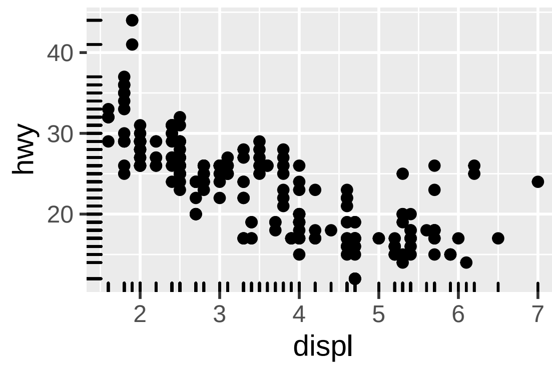
Since this looks just slightly odd, let's try to make it more apparent, what is happening.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(color = "blue") +
  geom_text(aes(label = drv), size = 10)
```




```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
```

```
  geom_point() +
  geom_rug()
```



Chapter 3

Transformation

> Using a consistent grammar of data manipulation.

This chapter discusses data transformation with the dplyr package.

3.1 Package: {conflicted}

Click here to show setup code.

```
library(tidyverse)
library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

This section is dedicated to show you the basic building blocks (i.e. functions) of data analysis in R within the {tidyverse}. The package providing these is {dplyr}.

Before starting, we would like to mention the package {conflicted}, which when loaded, will help detecting functions of the same name from different packages (an error is thrown in case of such situations). It furthermore helps to resolve these situations, by allowing you to choose, the function of which package you prefer (`conflicted::conflict_prefer()`). You can see an example in the setup code.

3.2 Filtering: `dplyr::filter()`

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

During this lecture we will be working with data from the package {nycflights13}, which contains flights in the year 2013 with their departure in New York City (airports: JFK, LGA or EWR) to destinations in the United States, Puerto Rico, and the American Virgin Islands.

```
flights
```

```
## # A tibble: 336,776 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## # ... with 3.368e+05 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

```
?flights
```

```
?flights
```

The function `dplyr::filter()` helps you to reduce your dataset to the observations (rows) of interest. The filter condition can use any of the dataset's variables and needs to be a logical expression.

```
flights %>%
  filter(dep_time < 600)
```

```
## # A tibble: 8,730 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## # ... with 8,727 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
```

```
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

If you use one or more variables of the dataset in the filter condition, a vectorized evaluation of the condition is taking place. Generally you can provide any logical vector with a length equal to the number of rows (or alternatively equal to 1, if you want to keep/drop all rows).

```
flights %>%
  filter(is.na(dep_time))
```

```
## # A tibble: 8,255 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1       NA           1630        NA       NA
## 2  2013     1     1       NA           1935        NA       NA
## 3  2013     1     1       NA           1500        NA       NA
## # ... with 8,252 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Use `&` or multiple filters to return only rows that match both criteria:

```
flights %>%
  filter(dep_time < 600 & arr_time > 2200)
```

```
## # A tibble: 0 x 19
## # ... with 19 variables: year <int>, month <int>, day <int>,
## #   dep_time <int>, sched_dep_time <int>, dep_delay <dbl>,
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
## #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
flights %>%
  filter(dep_time >= 700 & arr_time < 800)
```

```
## # A tibble: 10,654 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1     1929           1920         9        3
## 2  2013     1     1     1939           1840        59       29
## 3  2013     1     1     2058           2100        -2        8
## # ... with 1.065e+04 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
```

```
## #   time_hour <dttm>
```

```
flights %>%
  filter(dep_time >= 700) %>%
  filter(arr_time < 800)
```

```
## # A tibble: 10,654 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1     1929           1920         9        3
## 2  2013     1     1     1939           1840        59       29
## 3  2013     1     1     2058           2100        -2        8
## # ... with 1.065e+04 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Use `|` to return all rows that match either criterion or both:

```
flights %>%
  filter(dep_time < 600 | arr_time > 2200)
```

```
## # A tibble: 40,879 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## # ... with 4.088e+04 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

3.3 Sort rows: `dplyr::arrange()`

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

The function `dplyr::arrange()` sorts the rows of the dataset according to the values of the variable(s) you are providing.

```
flights %>%
  arrange(dep_time)
```

```
## # A tibble: 336,776 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1    13        1           2249        72      108
## 2  2013     1    31        1           2100       181      124
## 3  2013    11    13        1           2359         2      442
## # ... with 3.368e+05 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

When providing multiple variables as arguments for `...` (the ellipsis), the dataset is first sorted accorcing to the values of the first variable. Wherever these values occur more than once, another sorting takes place within those groups, according to the second variable you provided. The same rule applies for every further variable you add to `arrange()`.

```
flights %>%
  arrange(dep_time, dep_delay)
```

```
## # A tibble: 336,776 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013    11    13        1           2359         2      442
## 2  2013    12    16        1           2359         2      447
## 3  2013    12    20        1           2359         2      430
## # ... with 3.368e+05 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

You can combine `filter()` and `arrange()`.

```
flights %>%
  filter(dep_time < 600) %>%
  filter(month >= 10) %>%
  arrange(dep_time, dep_delay) %>%
  view()
```

```
## # A tibble: 1,894 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1   2013    11    13        1           2359         2      442
## 2   2013    12    16        1           2359         2      447
## 3   2013    12    20        1           2359         2      430
## # ... with 1,891 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

You can use `arrange()` with arbitrary expressions.

```
flights %>%
  filter(month == 4) %>%
  filter(day == 1) %>%
  arrange(is.na(dep_time)) %>%
  view()
```

```
## # A tibble: 970 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1   2013     4     1      454            500        -6      636
## 2   2013     4     1      509            515        -6      743
## 3   2013     4     1      526            530        -4      812
## # ... with 967 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

The reason for the result you just saw in the view of the filtered dataset is, that the binary result of the expression (`TRUE`, `FALSE`) is sorted `FALSE` first (lexicographically).

Let's give it a twist:

```
flights %>%
  filter(month == 4) %>%
  filter(day == 1) %>%
  arrange(!is.na(dep_time)) %>%
  view()
```

```
## # A tibble: 970 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1   2013     4     1       NA           1125        NA       NA
## 2   2013     4     1       NA           1545        NA       NA
```

```
## 3  2013     4     1       NA            850        NA       NA
## # ... with 967 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Sorting the dataset according to which flights arrived earliest on April 1, 2013:

```
flights %>%
  filter(month == 4) %>%
  filter(day == 1) %>%
  arrange(arr_time) %>%
  view()
```

```
## # A tibble: 970 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     4     1     2243           2245        -2        6
## 2  2013     4     1     2056           1925        91        8
## 3  2013     4     1     2216           2100        76        9
## # ... with 967 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Invert the sorting by either…

```
flights %>%
  filter(month == 4) %>%
  filter(day == 1) %>%
  arrange(-arr_time) %>%
  view()
```

```
## # A tibble: 970 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     4     1     2027           2032        -5     2358
## 2  2013     4     1     2151           1930       141     2358
## 3  2013     4     1     2252           2245         7     2358
## # ... with 967 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

… or:

```
flights %>%
  filter(month == 4) %>%
  filter(day == 1) %>%
  arrange(desc(arr_time)) %>%
  view()
```

```
## # A tibble: 970 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     4     1     2027           2032        -5     2358
## 2  2013     4     1     2151           1930       141     2358
## 3  2013     4     1     2252           2245         7     2358
## # ... with 967 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

You can mix sorting in an ascending and a descending manner:

```
flights %>%
  filter(month == 4) %>%
  filter(day == 1) %>%
  arrange(dep_time, desc(arr_time)) %>%
  view()
```

```
## # A tibble: 970 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     4     1      454            500        -6      636
## 2  2013     4     1      509            515        -6      743
## 3  2013     4     1      526            530        -4      812
## # ... with 967 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

3.4 The pipe

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)
```

```
library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

We already heavily used it today, but what exactly are the characteristics of `%>%`, better known as "the pipe"?

```
early_flights <-
  flights %>%
  filter(dep_time < 600)
```

The above is just another way of writing:

```
early_flights <- filter(flights, dep_time < 600)
```

The manual describes this operator in detail:

```
?"%>%"
```

With the pipe, code can be read in a natural way, from left to right. The following snippet extracts

1. all early flights
2. from October till December,
3. ordered by departure time and then departure delay
4. and displays it.

Note how the reading corresponds to the code.

```
flights %>%
  filter(dep_time < 600) %>%
  filter(month >= 10) %>%
  arrange(dep_time, dep_delay) %>%
  view()
```

```
## # A tibble: 1,894 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013    11    13        1           2359         2      442
## 2  2013    12    16        1           2359         2      447
## 3  2013    12    20        1           2359         2      430
## # ... with 1,891 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

This is possible, because all transformation verbs (`filter()`, `arrange()`, `view()`) accept the main input (a tibble) as the first argument and also return a tibble.

The following three codes are equivalent, but are more difficult to write, to read and to maintain.

Naming is hard. Trying to give each intermediate result a name is exhausting. Introducing an additional step in this sequence of operations is prone to errors.

```
early_flights <- filter(flights, dep_time < 600)
early_flights_oct_dec <- filter(early_flights, month >= 10)
early_flights_oct_dec_sorted <- arrange(early_flights_oct_dec, dep_time, dep_delay)
view(early_flights_oct_dec_sorted)
```

```
## # A tibble: 1,894 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013    11    13        1           2359         2      442
## 2  2013    12    16        1           2359         2      447
## 3  2013    12    20        1           2359         2      430
## # ... with 1,891 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

We can keep using the same variable, e.g. `x`, to avoid naming. This adds noise compared to the pipe.

```
x <- flights
x <- filter(x, dep_time < 600)
x <- filter(x, month >= 10)
x <- arrange(x, dep_time, dep_delay)
view(x)
```

```
## # A tibble: 1,894 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013    11    13        1           2359         2      442
## 2  2013    12    16        1           2359         2      447
## 3  2013    12    20        1           2359         2      430
## # ... with 1,891 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

We can avoid intermediate variables. This disconnects the verbs from their arguments and is very difficult to read.

```
view(
  arrange(
    filter(
      filter(
        flights,
        dep_time < 600
      ),
      month >= 10
    ),
    dep_time, dep_delay
  )
)
```

```
## # A tibble: 1,894 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013    11    13        1           2359         2      442
## 2  2013    12    16        1           2359         2      447
## 3  2013    12    20        1           2359         2      430
## # ... with 1,891 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

3.4.1 Further advantages

When working on a code chunk consisting of subsequent transformations connected by pipes, it can be useful to end the pipeline with either `I` or `view()`.

```
flights %>%
  filter(dep_time < 600) %>%
  filter(month >= 10) %>% I
```

```
## # A tibble: 1,894 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
## * <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013    10     1      447            500       -13      614
## 2  2013    10     1      522            517         5      735
## 3  2013    10     1      536            545        -9      809
## # ... with 1,891 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

```
##arrange(dep_time, dep_delay) %>%
##view()
```

Once the chunk does what you expect it to do, do not forget to remove the `I` or `view()` call.

```
try(
  arrange(dep_time, dep_delay) %>%
  view()
)
```

```
## Error in arrange(dep_time, dep_delay) : object 'dep_time' not found
```

To rearrange rows, you can use the shortcut `Alt + Cursor up/down`. In a piped expression, no further editing is necessary!

3.5 Pick columns: `dplyr::select()`

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

With `dplyr::select()` you can (de-)select and/or rename columns of your dataset. The basic operation is like in the following examples:

```
flights %>%
  select(year, month, day)
```

```
## # A tibble: 336,776 x 3
##    year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## # ... with 3.368e+05 more rows
```

```
flights %>%
  select(-year)
```

```
## # A tibble: 336,776 x 18
```

```
##   month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int>    <int>          <int>     <dbl>    <int>
## 1     1     1      517            515         2      830
## 2     1     1      533            529         4      850
## 3     1     1      542            540         2      923
## # ... with 3.368e+05 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Renaming works by addressing an existing column on the right hand side of an equality sign and providing the new name of the column on its left hand side.

```
flights %>%
  select(
    year, month, day,
    departure_delay = dep_delay,
    arrival_delay = arr_delay
  )
```

```
## # A tibble: 336,776 x 5
##    year month   day departure_delay arrival_delay
##   <int> <int> <int>           <dbl>         <dbl>
## 1  2013     1     1               2            11
## 2  2013     1     1               4            20
## 3  2013     1     1               2            33
## # ... with 3.368e+05 more rows
```

With backticks, it is possible, but not advised, to use arbitrary characters (including spaces) in column names:

```
flights_with_spaces <-
  flights %>%
  select(
    year, month, day,
    `Departure delay` = dep_delay,
    `Arrival delay` = arr_delay
  ) %>%
  filter(
    `Arrival delay` < 0
  )
```

Address them in the same way, if the dataset already has such variables:

```
flights_with_spaces %>%
  select(
    year, month, day,
    dep_delay = `Departure delay`,
```

```
    arr_delay = `Arrival delay`
  )
```

```
## # A tibble: 188,933 x 5
##    year month   day dep_delay arr_delay
##   <int> <int> <int>     <dbl>     <dbl>
## 1  2013     1     1        -1       -18
## 2  2013     1     1        -6       -25
## 3  2013     1     1        -3       -14
## # ... with 1.889e+05 more rows
```

The {janitor} package helps fixing issues with colum names automatically.

Select helpers allow selecting multiple related columns conveniently.

```
flights %>%
  select(origin, dest, ends_with("_time"))
```

```
## # A tibble: 336,776 x 7
##   origin dest  dep_time sched_dep_time arr_time sched_arr_time
##   <chr>  <chr>    <int>          <int>    <int>          <int>
## 1 EWR    IAH        517            515      830            819
## 2 LGA    IAH        533            529      850            830
## 3 JFK    MIA        542            540      923            850
## # ... with 3.368e+05 more rows, and 1 more variable:
## #   air_time <dbl>
```

3.6 Create new columns based on old ones: `dplyr::mutate()`

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

```
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Will prefer dplyr::lag over any other package
```

With `dplyr::mutate()` you can add new columns to a table, e.g. making use of the already existing variables.

How much faster than the scheduled time did the pilots manage to fly:

```
flights %>%
  mutate(recovery = dep_delay - arr_delay)
```

```
## # A tibble: 336,776 x 20
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## # ... with 3.368e+05 more rows, and 13 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>, recovery <dbl>
```

This is another building block added to the toolset:

```
flights %>%
  mutate(recovery = dep_delay - arr_delay) %>%
  select(dep_delay, arr_delay, recovery)
```

```
## # A tibble: 336,776 x 3
##   dep_delay arr_delay recovery
##       <dbl>     <dbl>    <dbl>
## 1         2        11       -9
## 2         4        20      -16
## 3         2        33      -31
## # ... with 3.368e+05 more rows
```

Work with the newly created variable just like with the original ones:

```
flights %>%
  mutate(recovery = dep_delay - arr_delay) %>%
  select(dep_delay, arr_delay, recovery) %>%
  arrange(recovery)
```

```
## # A tibble: 336,776 x 3
##   dep_delay arr_delay recovery
##       <dbl>     <dbl>    <dbl>
## 1        -2       194     -196
## 2        -2       179     -181
## 3       180       345     -165
## # ... with 3.368e+05 more rows
```

Assign the results to new variables. The old ones remain unchanged.

```
recovery_data <-
  flights %>%
```

```
  mutate(recovery = dep_delay - arr_delay) %>%
  select(dep_delay, arr_delay, recovery) %>%
  arrange(recovery)

recovery_data
```

```
## # A tibble: 336,776 x 3
##   dep_delay arr_delay recovery
##       <dbl>     <dbl>    <dbl>
## 1        -2       194     -196
## 2        -2       179     -181
## 3       180       345     -165
## # ... with 3.368e+05 more rows
```

Let's look at a single airplane:

```
flights %>%
  filter(tailnum == "N14228") %>%
  select(year, month, day, dep_time, arr_time) %>%
  view()
```

```
## # A tibble: 111 x 5
##    year month   day dep_time arr_time
##   <int> <int> <int>    <int>    <int>
## 1  2013     1     1      517      830
## 2  2013     1     8     1435     1717
## 3  2013     1     9      717      812
## # ... with 108 more rows
```

Adding the departure time of the *next* flight to the current row, respectively, using `mutate()` with `lead()`:

```
flights %>%
  filter(tailnum == "N14228") %>%
  select(year, month, day, dep_time, arr_time) %>%
  mutate(lead_dep_time = lead(dep_time)) %>%
  view()
```

```
## # A tibble: 111 x 6
##    year month   day dep_time arr_time lead_dep_time
##   <int> <int> <int>    <int>    <int>         <int>
## 1  2013     1     1      517      830          1435
## 2  2013     1     8     1435     1717           717
## 3  2013     1     9      717      812          1143
## # ... with 108 more rows
```

The opposite effect to `lead()` can be realized using `lag()`:

```
flights %>%
  filter(tailnum == "N14228") %>%
  select(year, month, day, dep_time, arr_time) %>%
  mutate(lag_arr_time = lag(arr_time)) %>%
  view()
```

```
## # A tibble: 111 x 6
##    year month   day dep_time arr_time lag_arr_time
##   <int> <int> <int>    <int>    <int>        <int>
## 1  2013     1     1      517      830           NA
## 2  2013     1     8     1435     1717          830
## 3  2013     1     9      717      812         1717
## # ... with 108 more rows
```

There is even a use-case for this in our little example. How long has our airplane been absent from NYC airports between each of its flights out?

```
flights %>%
  filter(tailnum == "N14228") %>%
  select(year, month, day, dep_time, arr_time) %>%
  mutate(lag_arr_time = lag(arr_time)) %>%
  mutate(ground_time = dep_time - lag_arr_time) %>%
  view()
```

```
## # A tibble: 111 x 7
##    year month   day dep_time arr_time lag_arr_time ground_time
##   <int> <int> <int>    <int>    <int>        <int>       <int>
## 1  2013     1     1      517      830           NA          NA
## 2  2013     1     8     1435     1717          830         605
## 3  2013     1     9      717      812         1717       -1000
## # ... with 108 more rows
```

The negative values occur because not everything happens on the same day, implying that our method is still in need of some refinement. Nevertheless, let's continue.

A frequently used workflow is creating a helper variable at some point in the pipeline and then dropping it later on:

```
flights %>%
  filter(tailnum == "N14228") %>%
  select(year, month, day, dep_time, arr_time) %>%
  mutate(lag_arr_time = lag(arr_time)) %>%
  mutate(ground_time = dep_time - lag_arr_time) %>%
  select(-lag_arr_time)
```

```
## # A tibble: 111 x 6
##    year month   day dep_time arr_time ground_time
```

```
##    <int> <int> <int>    <int>    <int>       <int>
## 1  2013     1     1      517      830          NA
## 2  2013     1     8     1435     1717         605
## 3  2013     1     9      717      812       -1000
## # ... with 108 more rows
```

Let's work some more with the flight data of our special plane.

```
flights %>%
  filter(tailnum == "N14228") %>%
  view()
```

```
## # A tibble: 111 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     8     1435           1440        -5     1717
## 3  2013     1     9      717            700        17      812
## # ... with 108 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

The total air time of a plane up to and including a given flight can be calculated with `base::cumsum()`:

```
flights %>%
  filter(tailnum == "N14228") %>%
  mutate(cum_air_time = cumsum(air_time)) %>%
  select(air_time, cum_air_time) %>%
  view()
```

```
## # A tibble: 111 x 2
##   air_time cum_air_time
##      <dbl>        <dbl>
## 1      227          227
## 2      150          377
## 3       39          416
## # ... with 108 more rows
```

Creating a "flag" variable with `mutate()` which shows if a flight was on time or not:

```
flights %>%
  filter(tailnum == "N14228") %>%
  mutate(delayed = if_else(arr_delay > 0, "delayed", "on time")) %>%
  select(arr_delay, delayed)
```

```
## # A tibble: 111 x 2
##   arr_delay delayed
##       <dbl> <chr>
## 1        11 delayed
## 2       -29 on time
## 3        -3 on time
## # ... with 108 more rows
```

A more straightforward way to get the same (or at least a very similar and probably easier to work with) result:

```
flights %>%
  filter(tailnum == "N14228") %>%
  mutate(delayed = arr_delay > 0) %>%
  select(arr_delay, delayed)
```

```
## # A tibble: 111 x 2
##   arr_delay delayed
##       <dbl> <lgl>
## 1        11 TRUE
## 2       -29 FALSE
## 3        -3 FALSE
## # ... with 108 more rows
```

… easier to work with, because `filter()` can directly take logical arguments:

```
flights %>%
  filter(tailnum == "N14228") %>%
  mutate(delayed = arr_delay > 0) %>%
  select(arr_delay, delayed) %>%
  filter(delayed)
```

```
## # A tibble: 39 x 2
##   arr_delay delayed
##       <dbl> <lgl>
## 1        11 TRUE
## 2        39 TRUE
## 3        54 TRUE
## # ... with 36 more rows
```

Negation for inverse filtering:

```
flights %>%
  filter(tailnum == "N14228") %>%
  mutate(delayed = arr_delay > 0) %>%
  select(arr_delay, delayed) %>%
  filter(!delayed)
```

```
## # A tibble: 72 x 2
```

```
##   arr_delay delayed
##       <dbl> <lgl>
## 1       -29 FALSE
## 2        -3 FALSE
## 3       -20 FALSE
## # ... with 69 more rows
```

These are the flights that had no delay:

```
on_time_flights <-
  flights %>%
  filter(tailnum == "N14228") %>%
  mutate(delayed = arr_delay > 0) %>%
  select(arr_delay, delayed) %>%
  filter(!delayed)
```

3.7 Summarize data (by groups): `dplyr::summarize()`, `dplyr::group_by()` + `dplyr::ungroup()`

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference

## [conflicted] Will prefer dplyr::filter over any other package
```

```
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Removing existing preference

## [conflicted] Will prefer dplyr::lag over any other package
```

Often we want to draw just conclusions from larger datasets by gaining insight by using statistical (or other) methods for summarizing – and thus drastically reducing – the data: How much time did all planes spend in the air?

```
flights %>%
  select(air_time) %>%
  mutate(total_air_time = sum(air_time, na.rm = TRUE))
```

```
## # A tibble: 336,776 x 2
##   air_time total_air_time
##      <dbl>          <dbl>
```

```
## 1      227       49326610
## 2      227       49326610
## 3      160       49326610
## # ... with 3.368e+05 more rows
```

The `mutate()` call adds a new variable with the same value across all rows. To reduce the result to a single row, use `summarize()`:

```
flights %>%
  summarize(total_air_time = sum(air_time, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   total_air_time
##            <dbl>
## 1       49326610
```

Simple counts can be computed with `n()` inside `summarize()`:

```
flights %>%
  summarize(n = n())
```

```
## # A tibble: 1 x 1
##        n
##    <int>
## 1 336776
```

A variety of aggregate functions is supported:

```
flights %>%
  summarize(median = median(air_time, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   median
##    <dbl>
## 1    129
```

It's possible to produce two different summarizations at once:

```
flights %>%
  summarize(
    n = n(),
    mean_air_time = mean(air_time, na.rm = TRUE),
    median_air_time = median(air_time, na.rm = TRUE)
  )
```

```
## # A tibble: 1 x 3
##        n mean_air_time median_air_time
##    <int>         <dbl>           <dbl>
## 1 336776          151.             129
```

The `summarize()` verb gains its full power in grouped operations. Surround with `group_by()` and `ungroup()` to compute summaries in groups defined by common values in one or more columns. In the next example, the same summary is computed separately for each origin airport.

```
flights %>%
  group_by(origin) %>%
  summarize(
    n = n(),
    mean_air_time = mean(air_time, na.rm = TRUE),
    median_air_time = median(air_time, na.rm = TRUE)
  ) %>%
  ungroup()
```

```
## # A tibble: 3 x 4
##   origin      n mean_air_time median_air_time
##   <chr>   <int>         <dbl>           <dbl>
## 1 EWR    120835          153.             130
## 2 JFK    111279          178.             149
## 3 LGA    104662          118.             115
```

The next example splits the data into one group for each day.

```
flights %>%
  group_by(year, month, day) %>%
  summarize(
    n = n(),
    mean_air_time = mean(air_time, na.rm = TRUE),
    median_air_time = median(air_time, na.rm = TRUE)
  ) %>%
  ungroup()
```

```
## # A tibble: 365 x 6
##    year month   day     n mean_air_time median_air_time
##   <int> <int> <int> <int>         <dbl>           <dbl>
## 1  2013     1     1   842          170.             149
## 2  2013     1     2   943          162.             148
## 3  2013     1     3   914          157.             148
## # ... with 362 more rows
```

For quick exploration, the names of the new columns can be omitted:

```
flights %>%
  group_by(year, month, day) %>%
  summarize(
    n(),
    mean(air_time, na.rm = TRUE),
    median(air_time, na.rm = TRUE)
  ) %>%
```

```
  ungroup()
```

```
## # A tibble: 365 x 6
##    year month   day `n()` `mean(air_time, n~ `median(air_time~
##   <int> <int> <int> <int>              <dbl>             <dbl>
## 1  2013     1     1   842               170.               149
## 2  2013     1     2   943               162.               148
## 3  2013     1     3   914               157.               148
## # ... with 362 more rows
```

```
TRUE
```

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

```
TRUE
```

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

3.8 Summary-plots

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

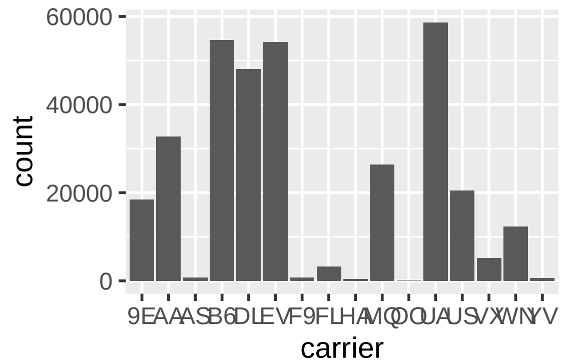
```
## [conflicted] Removing existing preference

## [conflicted] Will prefer dplyr::filter over any other package
```

```
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Removing existing preference

## [conflicted] Will prefer dplyr::lag over any other package
```

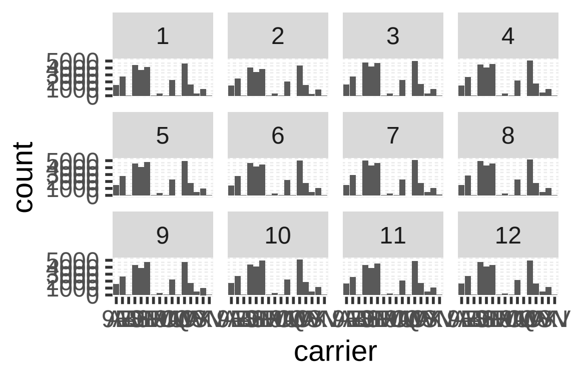
Potentially surprisingly, `mutate()` can also work with the results of a `ggplot()` call. Let's approach this step by step. Here is a basic barplot of `flights$carrier`:

```
flights %>%
  ggplot(aes(x = carrier)) +
  geom_bar()
```



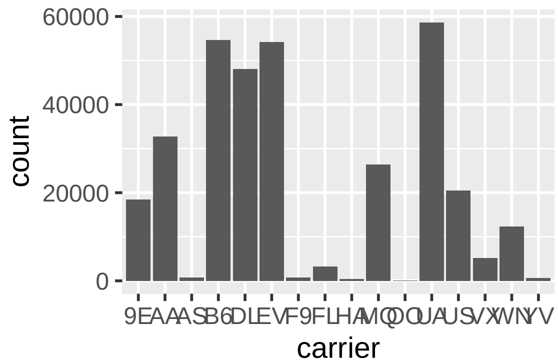
Same with one facet per month:

```
flights %>%
  ggplot(aes(x = carrier)) +
  geom_bar() +
  facet_wrap(~month)
```



We can extract a function that takes any data and produces a barplot of the variable `carrier`:

```
plot_fun <- function(data) {
  data %>%
    ggplot(aes(x = carrier)) +
    geom_bar()
}

plot_fun(flights)
```



The result of `ggplot()` is first and foremost an object. Only when R tries to display it on the console a method is triggered, which causes it to show the graph in the "Viewer". Therefore, we can use the `group_by` – `summarize()` – `ungroup()` pattern to produce one plot per group and store it in a new column:

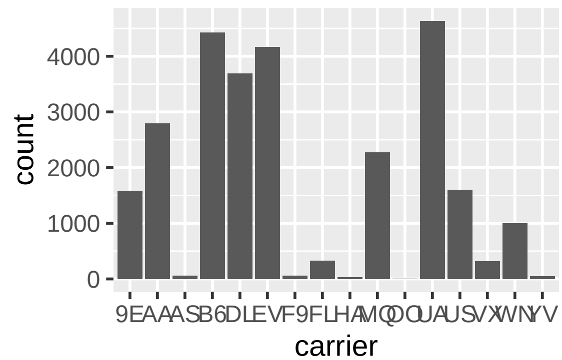
```
plot_df <-
  flights %>%
  group_by(month) %>%
  summarize(
    plot = list(plot_fun(tibble(carrier)))
  ) %>%
  ungroup()

plot_df
```

```
## # A tibble: 12 x 2
##   month plot
##   <int> <list>
## 1     1 <gg>
## 2     2 <gg>
## 3     3 <gg>
## # ... with 9 more rows
```

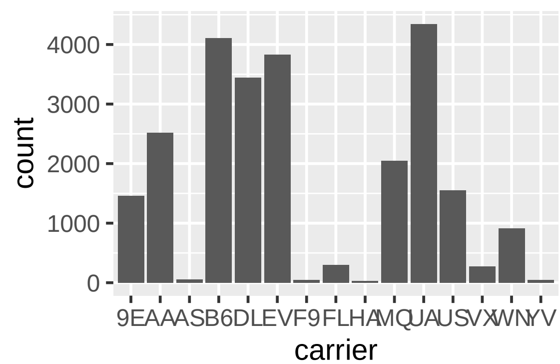
When using `dplyr::pull()` (this function "extracts" a variable from a `data.frame` and returns it as a normal vector), each of the plots will be subsequently displayed in your "Viewer".

```
plot_df %>%
  pull()
```

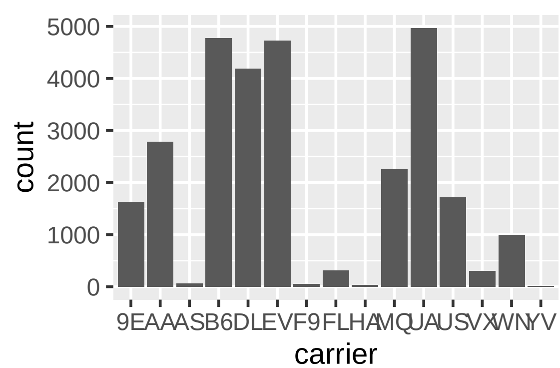
```
## [[1]]
```


count
4000
3000
2000
1000
0
9E AA AS B6 DL EV F9 FL HA MQ OO UA US VX WN YV
carrier

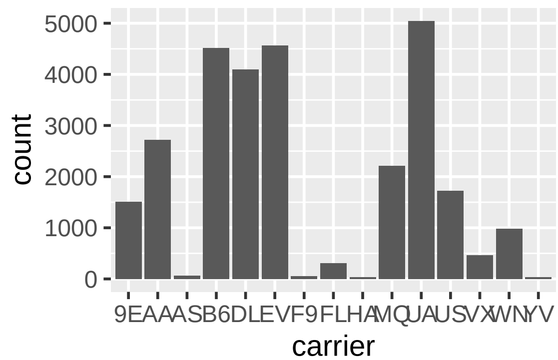
```
##
## [[2]]
```


count
4000
3000
2000
1000
0
9E AA AS B6 DL EV F9 FL HA MQ UA US VX WN YV
carrier

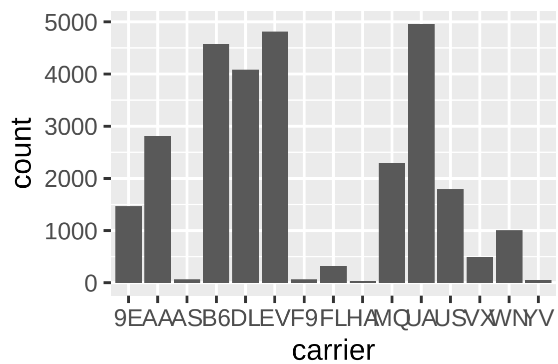
```
##
## [[3]]
```


count
5000
4000
3000
2000
1000
0
9E AA AS B6 DL EV F9 FL HA MQ UA US VX WN YV
carrier

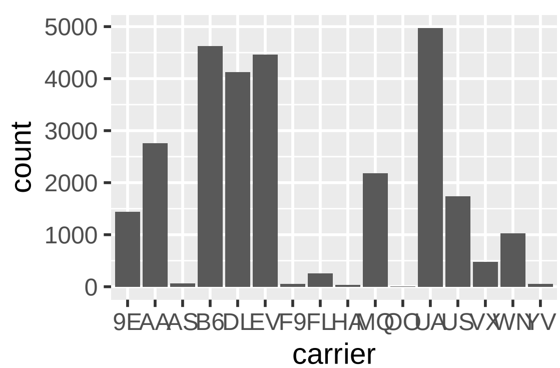
```
##
## [[4]]
```



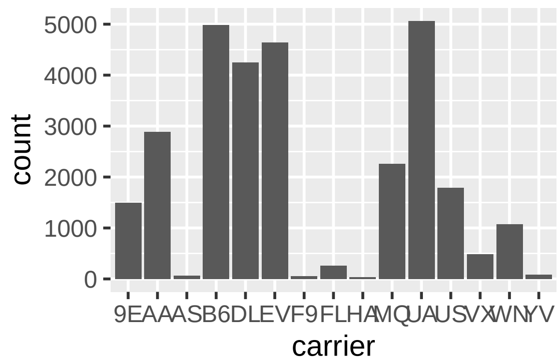

```
##
## [[5]]
```



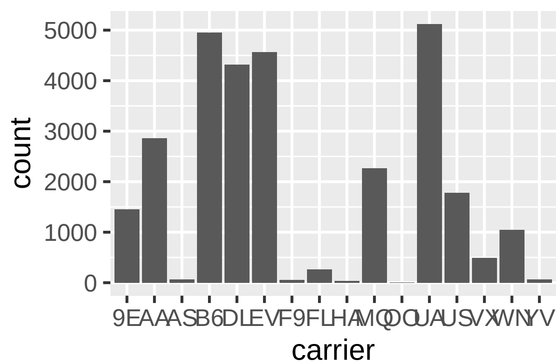

```
##
## [[6]]
```



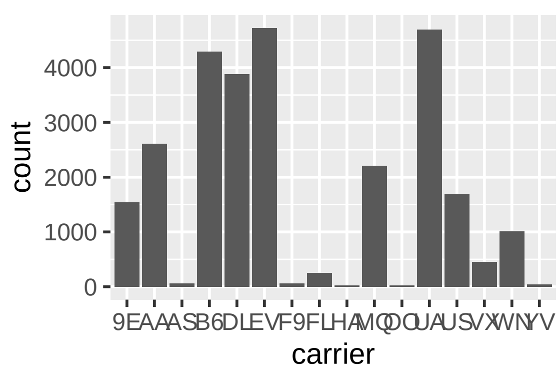

```
##
## [[7]]
```


count
5000
4000
3000
2000
1000
0
9E AA AS B6 DL EV F9 FL HA MQ UA US VX WN YV
carrier

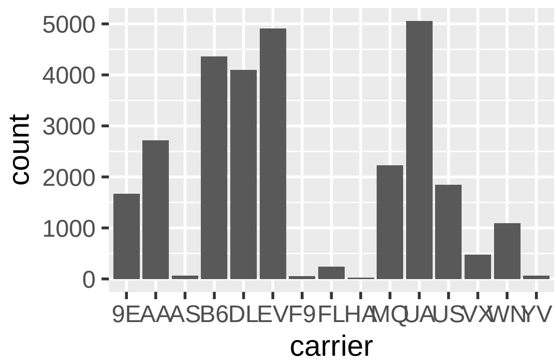
```
##
## [[8]]
```


count
5000
4000
3000
2000
1000
0
9E AA AS B6 DL EV F9 FL HA MQ OO UA US VX WN YV
carrier

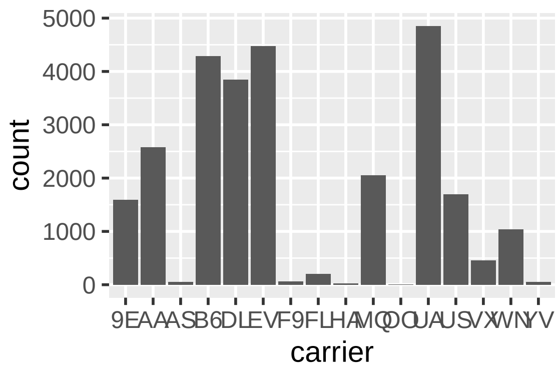
```
##
## [[9]]
```


count
4000
3000
2000
1000
0
9E AA AS B6 DL EV F9 FL HA MQ OO UA US VX WN YV
carrier

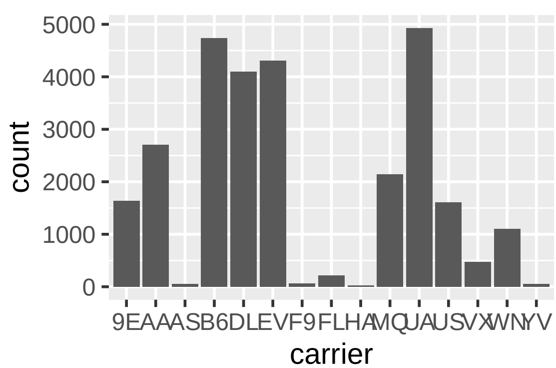
```
##
## [[10]]
```




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




```
##
## [[12]]
```



Use the left arrow to click through the different plots.

Chapter 4

Import

> Ingesting data.

This chapter discusses data import with RStudio, with the help of the readr, readxl, and rio packages.

4.1

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

```
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::lag over any other package
```

```
library(readr)
example1 <-
  read_delim(
    "data/example1.csv",
    ";",
```

```
    escape_double = FALSE, trim_ws = TRUE
  )
```

```
## Parsed with column specification:
## cols(
##   col1 = col_double(),
##   col2 = col_character(),
##   col3 = col_character()
## )
```

```
view(example1)
```

```
## # A tibble: 2 x 3
##    col1 col2  col3
##   <dbl> <chr> <chr>
## 1   1   a     X
## 2   2.5 b     Y
```

4.2 Import many files

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(here)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

```
files <- dir(path = here("data"), pattern = "[.]xlsx$", full.names = TRUE)
files
```

```
## [1] "/home/travis/build/krlmlr/vistransrep/book/data/example6a.xlsx"
## [2] "/home/travis/build/krlmlr/vistransrep/book/data/example6b.xlsx"
## [3] "/home/travis/build/krlmlr/vistransrep/book/data/example6c.xlsx"
```

```
files %>%
  rio::import_list(setclass = class(tibble()), rbind = TRUE)
```

```
## # A tibble: 6 x 5
##      id  col1 col2  col3  `_file`
##   <dbl> <dbl> <chr> <chr> <chr>
```

```
## 1      1    1    a     X     /home/travis/build/krlmlr/vistransr~
## 2      1    2.5  b     Y     /home/travis/build/krlmlr/vistransr~
## 3      2    1.5  c     Z     /home/travis/build/krlmlr/vistransr~
## # ... with 3 more rows
```

```
list_of_tables <- rio::import_list(files, setclass = class(tibble()))
list_of_tables
```

```
## $example6a
## # A tibble: 2 x 4
##       id  col1 col2  col3
##    <dbl> <dbl> <chr> <chr>
## 1      1    1    a     X
## 2      1    2.5  b     Y
##
## $example6b
## # A tibble: 2 x 4
##       id  col1 col2  col3
##    <dbl> <dbl> <chr> <chr>
## 1      2    1.5  c     Z
## 2      2    2    d     W
##
## $example6c
## # A tibble: 2 x 4
##       id  col1 col2  col3
##    <dbl> <dbl> <chr> <chr>
## 1      3    4    g     J
## 2      3    3.5  f     H
```

```
list_of_tables$example6b
```

```
## # A tibble: 2 x 4
##       id  col1 col2  col3
##    <dbl> <dbl> <chr> <chr>
## 1      2    1.5  c     Z
## 2      2    2    d     W
```

```
try(
  list_of_tables$example6b <-
    list_of_tables$example6b %>%
    mutate(...) %>%
    select(...)
)
```

```
## Error in function_list[[i]](value) : '...' used in an incorrect context
```

```
all_tables <- bind_rows(list_of_tables, .id = "path")
all_tables
```

```
## # A tibble: 6 x 5
##   path          id  col1 col2  col3
##   <chr>      <dbl> <dbl> <chr> <chr>
## 1 example6a      1   1   a     X
## 2 example6a      1   2.5 b     Y
## 3 example6b      2   1.5 c     Z
## # ... with 3 more rows
```

```
all_tables %>%
  filter(path == "example6b") %>%
  summarize(mean(col1), first(col2))
```

```
## # A tibble: 1 x 2
##   `mean(col1)` `first(col2)`
##          <dbl> <chr>
## 1         1.75 c
```

```
all_tables %>%
  group_by(path) %>%
  summarize(mean(col1), first(col2)) %>%
  ungroup()
```

```
## # A tibble: 3 x 3
##   path       `mean(col1)` `first(col2)`
##   <chr>             <dbl> <chr>
## 1 example6a          1.75 a
## 2 example6b          1.75 c
## 3 example6c          3.75 g
```

```
files %>%
  map_dfr(~ readxl::read_excel(.))
```

```
## # A tibble: 6 x 4
##      id  col1 col2  col3
##   <dbl> <dbl> <chr> <chr>
## 1     1   1   a     X
## 2     1   2.5 b     Y
## 3     2   1.5 c     Z
## # ... with 3 more rows
```

Chapter 5

Tidying

> Rows, columns, cells.

This chapter discusses pivoting and data tidying with the help of the tidyr package.

5.1 Pivoting

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

```
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::lag over any other package
```

Pivoting describes operations that help rearrange data in different ways. The following two tables contain the same data arranged differently.

```
table1
```

```
## # A tibble: 6 x 4
##   country      year cases population
```

```
##   <chr>       <int> <int>      <int>
## 1 Afghanistan  1999   745   19987071
## 2 Afghanistan  2000  2666   20595360
## 3 Brazil       1999 37737  172006362
## # ... with 3 more rows
```

```
table2
```

```
## # A tibble: 12 x 4
##   country      year type           count
##   <chr>       <int> <chr>          <int>
## 1 Afghanistan  1999 cases            745
## 2 Afghanistan  1999 population 19987071
## 3 Afghanistan  2000 cases           2666
## # ... with 9 more rows
```

Both tables contain `country` and `year` column that describe the source of the measurements. The "wider" version, `table1`, contains two columns that hold the number of cases (of a disease) and the population for the corresponding country in the corresponding year. In the "longer" version, `table2`, the number of cases and the population are stored in the same `count` column, with the `type` column defining the measurement.

Somewhat counter-intuitively, "longer-form" data is often better suited for analyzing data. "Wider-form" data makes better use of screen space, but may be more difficult to work with.

The following example computes the maximum number of cases and population for each country. For the wider form, this requires repeating the same expression for all columns. This may work with two columns but becomes tedious once more measurements are added.

```
table1 %>%
  group_by(country) %>%
  summarize(
    max_cases = max(cases),
    max_population = max(population)
  ) %>%
  ungroup()
```

```
## # A tibble: 3 x 3
##   country     max_cases max_population
##   <chr>           <int>          <int>
## 1 Afghanistan      2666       20595360
## 2 Brazil          80488      174504898
## 3 China          213766     1280428583
```

The `_at` family of functions helps iterating over columns, but all columns still need to be enumerated. (Specifying ranges of columns is rather brittle.)

```
table1 %>%
  group_by(country) %>%
  summarize_at(
    vars(cases, population),
    max
  ) %>%
  ungroup()
```

```
## # A tibble: 3 x 3
##   country      cases population
##   <chr>        <int>      <int>
## 1 Afghanistan   2666   20595360
## 2 Brazil       80488  174504898
## 3 China       213766 1280428583
```

If the data is in the "longer" form, it is sufficient to include `type` in the grouping variables. The same code works for arbitrary number of measurements.

```
table2 %>%
  group_by(country, type) %>%
  summarize(
    max = max(count)
  ) %>%
  ungroup()
```

```
## # A tibble: 6 x 3
##   country     type            max
##   <chr>       <chr>         <int>
## 1 Afghanistan cases          2666
## 2 Afghanistan population 20595360
## 3 Brazil      cases         80488
## # ... with 3 more rows
```

The following examples give a gentle introduction into pivoting.

5.1.1 Convert to longer form

The `pivot_longer()` function takes a "wider-form" dataset and converts it to an equivalent dataset with more rows.

```
table1
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>       <int> <int>      <int>
## 1 Afghanistan  1999   745   19987071
## 2 Afghanistan  2000  2666   20595360
```

```
## 3 Brazil       1999 37737  172006362
## # ... with 3 more rows
```

```
table1 %>%
  pivot_longer(-c(country, year))
```

```
## # A tibble: 12 x 4
##   country      year name          value
##   <chr>       <int> <chr>         <int>
## 1 Afghanistan  1999 cases           745
## 2 Afghanistan  1999 population 19987071
## 3 Afghanistan  2000 cases          2666
## # ... with 9 more rows
```

The `-c(...)` notation indicates that all column except `country` and `year` are to be transformed into longer form. The column names become the contents of the new `name` column, the values are available in the `value` column.

The result of this operation isn't strictly equivalent to `table2`, we need to rename and sort differently. Alternatively, the `names_to` and `values_to` arguments allow specifying the names of the new columns.

```
table1 %>%
  pivot_longer(-c(country, year)) %>%
  rename(type = name, count = value) %>%
  arrange(country, year, type)
```

```
## # A tibble: 12 x 4
##   country      year type          count
##   <chr>       <int> <chr>         <int>
## 1 Afghanistan  1999 cases           745
## 2 Afghanistan  1999 population 19987071
## 3 Afghanistan  2000 cases          2666
## # ... with 9 more rows
```

```
table1 %>%
  pivot_longer(
    -c(country, year),
    names_to = "type",
    values_to = "count"
  ) %>%
  arrange(country, year, type)
```

```
## # A tibble: 12 x 4
##   country      year type          count
##   <chr>       <int> <chr>         <int>
## 1 Afghanistan  1999 cases           745
## 2 Afghanistan  1999 population 19987071
## 3 Afghanistan  2000 cases          2666
```

```
## # ... with 9 more rows
```

5.1.2 Convert to wider form

The `pivot_wider()` form does the inverse: it creates a dataset with fewer rows. If the `name` and `value` columns are named differently, these columns can be provided via the `names_from` and `values_from` arguments.

```
table2
```

```
## # A tibble: 12 x 4
##   country      year type          count
##   <chr>       <int> <chr>         <int>
## 1 Afghanistan  1999 cases           745
## 2 Afghanistan  1999 population 19987071
## 3 Afghanistan  2000 cases          2666
## # ... with 9 more rows
```

```
table2 %>%
  pivot_wider(names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>       <int> <int>      <int>
## 1 Afghanistan  1999   745   19987071
## 2 Afghanistan  2000  2666   20595360
## 3 Brazil       1999 37737  172006362
## # ... with 3 more rows
```

```
table2 %>%
  rename(name = type, value = count) %>%
  pivot_wider()
```

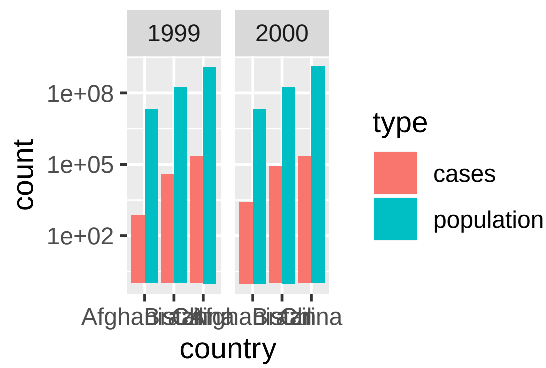
```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>       <int> <int>      <int>
## 1 Afghanistan  1999   745   19987071
## 2 Afghanistan  2000  2666   20595360
## 3 Brazil       1999 37737  172006362
## # ... with 3 more rows
```

5.1.3 Use cases

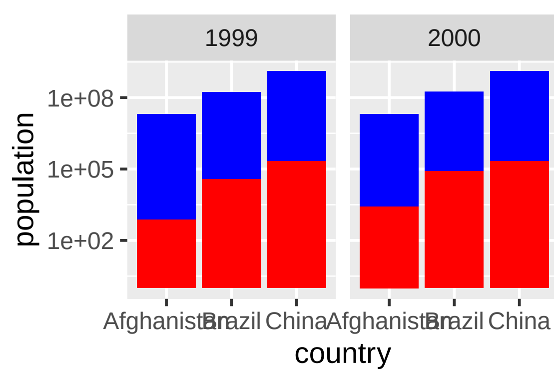
Data in "longer" form usually works better for plotting the values side by side, e.g. by assigning the type of value to an aesthetic. Recall that each row in the data produces one geometric object in the corresponding layer. For a bar chart that shows cases and population side by side, mapped to the y aesthetic, the "longer" form is more natural.

- table2 form requires only one layer, the fill color is determined automatically, the legend is created automatically
- table1 requires two layers, manual assignment of fill color, and manual creation of legend (not shown)

```
table2 %>%
  ggplot() +
  geom_col(aes(country, count, fill = type), position = "dodge") +
  facet_wrap(~year) +
  scale_y_log10()
```



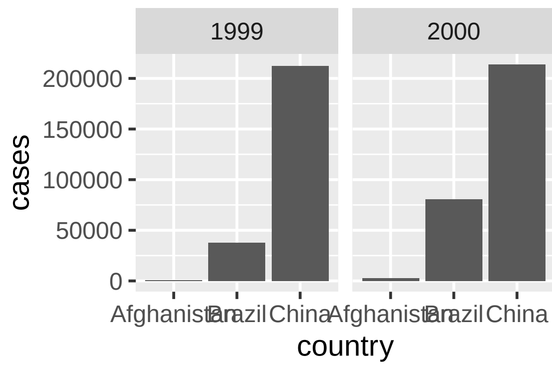

```
table1 %>%
  ggplot() +
  geom_col(aes(country, population), position = "dodge", fill = "blue") +
  geom_col(aes(country, cases), position = "dodge", fill = "red") +
  facet_wrap(~year) +
  scale_y_log10()
```



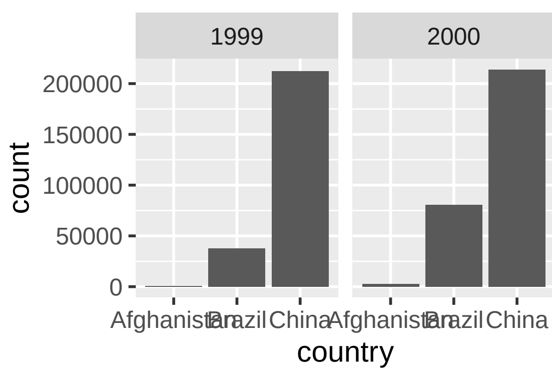
On the other hand, iIf only a single measurement needs to be plotted, the "wider" form is easier to work with.

- `table1` only requires selecting the correct column
- `table2` requires a `filter()`

```
table1 %>%
  ggplot() +
  geom_col(aes(country, cases)) +
  facet_wrap(~year)
```




```
table2 %>%
  filter(type == "cases") %>%
  ggplot() +
  geom_col(aes(country, count)) +
  facet_wrap(~year)
```



The "wider" form is also the only way to map different measures to different aesthetics, e.g. to correlate values.

```
table1 %>%
  ggplot() +
  geom_point(aes(cases, population, color = factor(year), shape = country)) +
```

```
  scale_x_log10() +
  scale_y_log10()
```

5.1.4 Combining vertically

A different view on the same data is given in the two tables `table4a` and `table4b`.

```
table4a
```

```
## # A tibble: 3 x 3
##   country      `1999` `2000`
## * <chr>         <int>  <int>
## 1 Afghanistan     745   2666
## 2 Brazil        37737  80488
## 3 China        212258 213766
```

```
table4b
```

```
## # A tibble: 3 x 3
##   country          `1999`     `2000`
## * <chr>             <int>      <int>
## 1 Afghanistan    19987071   20595360
## 2 Brazil        172006362  174504898
## 3 China        1272915272 1280428583
```

The `bind_rows()` function combines these two parts into a single table. The `.id = "type"` setting ensures that the input datasets gain different tags in the new `type` column.

```
table4 <-
  bind_rows(
    cases = table4a,
    population = table4b,
```

```
    .id = "type"
  )
table4
```

```
## # A tibble: 6 x 4
##   type  country     `1999` `2000`
##   <chr> <chr>        <int>  <int>
## 1 cases Afghanistan    745   2666
## 2 cases Brazil       37737  80488
## 3 cases China       212258 213766
## # ... with 3 more rows
```

As before, `pivot_longer()` helps converting the results into something similar to `table2`. The result isn't quite the same yet, can you spot the difference?

```
table4 %>%
  pivot_longer(c(`1999`, `2000`))
```

```
## # A tibble: 12 x 4
##   type  country     name  value
##   <chr> <chr>       <chr> <int>
## 1 cases Afghanistan 1999    745
## 2 cases Afghanistan 2000   2666
## 3 cases Brazil      1999  37737
## # ... with 9 more rows
```

5.1.5 Tidy data

From "R for data science":

> In a tidy dataset,
>
> 1. each variable must have its own column.
> 2. each observation must have its own row.
> 3. each value must have its own cell.

The following example shows a case that violates the first two rules: WHO data arranged for optimal use of screen space. The column names define, in addition to the measurement type `new_sp`, `new_sn`, `new_ep` and `newrel`, the age and sex stratum of the corresponding measurements. One single `pivot_longer()` call transforms the data into a longer-form version with four measurement columns and one row for each age/sex stratum. The `names_pattern` is a regular expression that defines what part of the column name is stored where. (Regular expressions are a powerful tool for parsing text data, out of scope for this lecture but very much worth looking into.) The `names_to` sequence defines, for each `()` group in `names_pattern`, if the data encoded in the column name is stored in a new column or if it is kept as column name.

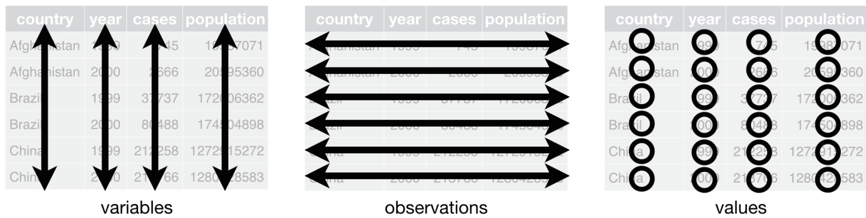


Figure 5.1: Tidy data

```r
who %>%
  view()
```

```
## # A tibble: 7,240 x 60
##   country iso2  iso3   year new_sp_m014 new_sp_m1524
##   <chr>   <chr> <chr> <int>       <int>        <int>
## 1 Afghan~ AF    AFG    1980          NA           NA
## 2 Afghan~ AF    AFG    1981          NA           NA
## 3 Afghan~ AF    AFG    1982          NA           NA
## # ... with 7,237 more rows, and 54 more variables:
## #   new_sp_m2534 <int>, new_sp_m3544 <int>,
## #   new_sp_m4554 <int>, new_sp_m5564 <int>, new_sp_m65 <int>,
## #   new_sp_f014 <int>, new_sp_f1524 <int>,
## #   new_sp_f2534 <int>, new_sp_f3544 <int>,
## #   new_sp_f4554 <int>, new_sp_f5564 <int>, new_sp_f65 <int>,
## #   new_sn_m014 <int>, new_sn_m1524 <int>,
## #   new_sn_m2534 <int>, new_sn_m3544 <int>,
## #   new_sn_m4554 <int>, new_sn_m5564 <int>, new_sn_m65 <int>,
## #   new_sn_f014 <int>, new_sn_f1524 <int>,
## #   new_sn_f2534 <int>, new_sn_f3544 <int>,
## #   new_sn_f4554 <int>, new_sn_f5564 <int>, new_sn_f65 <int>,
## #   new_ep_m014 <int>, new_ep_m1524 <int>,
## #   new_ep_m2534 <int>, new_ep_m3544 <int>,
## #   new_ep_m4554 <int>, new_ep_m5564 <int>, new_ep_m65 <int>,
## #   new_ep_f014 <int>, new_ep_f1524 <int>,
## #   new_ep_f2534 <int>, new_ep_f3544 <int>,
## #   new_ep_f4554 <int>, new_ep_f5564 <int>, new_ep_f65 <int>,
## #   newrel_m014 <int>, newrel_m1524 <int>,
## #   newrel_m2534 <int>, newrel_m3544 <int>,
## #   newrel_m4554 <int>, newrel_m5564 <int>, newrel_m65 <int>,
## #   newrel_f014 <int>, newrel_f1524 <int>,
## #   newrel_f2534 <int>, newrel_f3544 <int>,
## #   newrel_f4554 <int>, newrel_f5564 <int>, newrel_f65 <int>
```

```r
who_longer <-
  who %>%
  pivot_longer(
    -(country:year),
    names_pattern = "([a-z_]+)_(.)([0-9]+)",
    names_to = c(".value", "sex", "age")
  )

who_longer
```

```
## # A tibble: 101,360 x 10
##   country iso2  iso3   year sex   age   new_sp new_sn new_ep
##   <chr>   <chr> <chr> <int> <chr> <chr>  <int>  <int>  <int>
## 1 Afghan~ AF    AFG    1980 m     014       NA     NA     NA
## 2 Afghan~ AF    AFG    1980 m     1524      NA     NA     NA
## 3 Afghan~ AF    AFG    1980 m     2534      NA     NA     NA
## # ... with 1.014e+05 more rows, and 1 more variable:
## #   newrel <int>
```

```r
who_longer %>%
  count(sex, age)
```

```
## # A tibble: 14 x 3
##   sex   age       n
##   <chr> <chr> <int>
## 1 f     014    7240
## 2 f     1524   7240
## 3 f     2534   7240
## # ... with 11 more rows
```

5.2 Separating and uniting

Click here to show setup code.

```r
library(tidyverse)
library(nycflights13)

library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

```r
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::lag over any other package
```

The `table3` table violates the third principle of tidy data: each cell contains two values.

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## # ... with 3 more rows
```

The `separate()` verb offers a convenient way to deal with this situation, including automatic type conversion.

```
table3 %>%
  separate(rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>       <int> <chr> <chr>
## 1 Afghanistan  1999 745   19987071
## 2 Afghanistan  2000 2666  20595360
## 3 Brazil       1999 37737 172006362
## # ... with 3 more rows
```

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/", convert = TRUE)
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>       <int> <int>      <int>
## 1 Afghanistan  1999   745   19987071
## 2 Afghanistan  2000  2666   20595360
## 3 Brazil       1999 37737  172006362
## # ... with 3 more rows
```

The inverse is offered by `unite()`. The data in `table5` stores year data in two columns.

```
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
## * <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
```

```
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## # ... with 3 more rows
```

```
table5 %>%
  unite("year", c(century, year))
```

```
## # A tibble: 6 x 3
##   country     year  rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 19_99 745/19987071
## 2 Afghanistan 20_00 2666/20595360
## 3 Brazil      19_99 37737/172006362
## # ... with 3 more rows
```

The result needs a few tweaks to finally resemble `table3`.

```
table5 %>%
  unite("year", c(century, year), sep = "")
```

```
## # A tibble: 6 x 3
##   country     year  rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000  2666/20595360
## 3 Brazil      1999  37737/172006362
## # ... with 3 more rows
```

```
table5 %>%
  unite("year", c(century, year), sep = "") %>%
  mutate(year = as.numeric(year))
```

```
## # A tibble: 6 x 3
##   country      year rate
##   <chr>       <dbl> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## # ... with 3 more rows
```

See the help for further details.

```
?separate
?unite
```

5.2.1 Parsing numbers

```
thousand_separator <-
  tribble(
    ~num,
    "1'000.00",
    "2'000'000.00"
  )

thousand_separator
```

```
## # A tibble: 2 x 1
##   num
##   <chr>
## 1 1'000.00
## 2 2'000'000.00
```

```
thousand_separator %>%
  separate(num, into = c("num"))
```

```
## Warning: Expected 1 pieces. Additional pieces discarded in 2
## rows [1, 2].
```

```
## # A tibble: 2 x 1
##   num
##   <chr>
## 1 1
## 2 2
```

```
thousand_separator %>%
  mutate(num = str_replace_all(num, "[^-0-9.]", "")) %>%
  mutate(num = as.numeric(num))
```

```
## # A tibble: 2 x 1
##       num
##     <dbl>
## 1    1000
## 2 2000000
```

5.3

Click here to show setup code.

```
library(tidyverse)
library(nycflights13)
```

```r
library(conflicted)
conflict_prefer("filter", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::filter over any other package
```

```r
conflict_prefer("lag", "dplyr")
```

```
## [conflicted] Removing existing preference
```

```
## [conflicted] Will prefer dplyr::lag over any other package
```

```r
table2 %>%
  xtabs(count ~ ., .) %>%
  ftable()
```

```
##                   type      cases population
## country     year
## Afghanistan 1999              745   19987071
##             2000             2666   20595360
## Brazil      1999            37737  172006362
##             2000            80488  174504898
## China       1999           212258 1272915272
##             2000           213766 1280428583
```

```r
table2 %>%
  xtabs(count ~ ., .) %>%
  ftable(col.vars = c("year", "type"))
```

```
##             year       1999                    2000
##             type      cases population        cases population
## country
## Afghanistan             745   19987071         2666   20595360
## Brazil                37737  172006362        80488  174504898
## China                212258 1272915272       213766 1280428583
```

```r
?`tidyr-package`
```

```r
NA
```

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

Chapter 6

Best practices

R code is often organized in packages that can be installed from centralized repositories such as CRAN or GitHub. If you are new to writing R packages, this course cannot give a complete introduction into packages. It is still useful to embrace some very few concepts of R packages to gain access to a vast toolbox and also organize your code in a standardized way familiar to other users. With the first steps in place, the road to your first R package may become less steep.

- Create a `DESCRIPTION` file to declare dependencies and allow easy reloading of the functions you define
- Store your functions in `.R` files in the `R/` directory in your project
  - Scripts that you execute live in `script/` or a similar directory
- Use roxygen2 to document your functions close to the source
- Write tests for your functions, e.g. with testthat

See R packages for a more comprehensive treatment.

6.1 DESCRIPTION

Create and open a new RStudio project. Then, create a `DESCRIPTION` file with `usethis::use_description()`:

```
# install.packages("usethis")
usethis::use_description()
```

Double-check success:

```
# install.packages("devtools")
devtools::load_all()
```

Declare that your project requires the tidyverse and the here package:

```
usethis::use_package("here")
# Currently doesn't work, add manually
# https://github.com/r-lib/usethis/issues/760
# usethis::use_package("tidyverse")
```

6.2 R

With a `DESCRIPTION` file defined, create a new `.R` file and save it in the `R/` directory. (Create this directory if it does not exist.) Create a function in this file, save the file:

```
hi <- function(text = "Hello, world!") {
  print(text)
  invisible(text)
}
```

Do not source the file.

Restart R (with Ctrl + Shift + F10 in RStudio).

Run `devtools::load_all()` again, you can use the shortcut Ctrl + Shift + L or Cmd + Shift + L in RStudio.

Check that you can run `hi()` in the console:

```
hi()
```

```
## [1] "Hello, world!"
```

```
hi("Wow!")
```

```
## [1] "Wow!"
```

Edit the function:

```
hi <- function(text = "Wow!") {
  print(text)
  invisible(text)
}
```

Save the file, but do not source it.

Run `devtools::load_all()` again, you can use the shortcut Ctrl + Shift + L or Cmd + Shift + L in RStudio.

Check that the new implementation of `hi()` is active:

```
hi()
```

```
## [1] "Wow!"
```

All functions that are required for your project are stored in this directory. Do not store executable scripts, use a `script/` directory.

6.3 roxygen2

The following intuitive annotation syntax is a standard way to create documentation for your functions:

```
#' Print a welcome message
#'
#' This function prints "Wow!", or a custom text, on the console.
#'
#' @param text The text to print, "Wow!" by default.
#'
#' @return The `text` argument, invisibly.
#'
#' @examples
#' hi()
#' hi("Hello!")
hi <- function(text = "Wow!") {
  print(text)
  invisible(text)
}
```

This annotation can be rendered to a nicely looking HTML page with the roxygen2 and pkgdown packages. All you need to do is provide (and maintain) it.

6.4 testthat

Automated tests make sure that the functions you write today continue working tomorrow. Create your first test with `usethis::use_test()`:

```
# install.packages("testthat")
usethis::use_test("hi")
```

The file `tests/testthat/test-hi.R` is created, with the following contents:

```
test_that("multiplication works", {
  expect_equal(2 * 2, 4)
})
```

Replace this predefined text with a test that makes more sense for us:

```r
test_that("hi() works", {
  expect_output(hi(), "Wow")
  expect_output(hi("Hello"), "Hello")
})
```

Run the new test with `devtools::test()`, you can use the shortcut Ctrl + Shift + T or Cmd + Shift + T in RStudio.

Check that the test actually detects failures by modifying the implementation of `hi()` and rerunning the test:

```r
hi <- function(text = "Oops!") {
  print(text)
  invisible(text)
}
```

Run the new test with `devtools::test()`, you can use the shortcut Ctrl + Shift + T or Cmd + Shift + T in RStudio. One test should be failing now.

Chapter 7

- R for data science: https://r4ds.had.co.nz/
- Row oriented workflows: https://github.com/jennybc/row-oriented-workflows#readme
- Advanced R: http://adv-r.had.co.nz/
- Tidy evaluation: https://tidyeval.tidyverse.org/
- R packages: http://r-pkgs.had.co.nz/
- roxygen2: Vignettes in https://cran.r-project.org/package=roxygen2, especially:
  - Introduction to roxygen2
  - Generating Rd files for an overview of available tags
  - Write R documentation in Markdown
- How R searches and finds stuff: http://blog.obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/
- What they forgot to teach you: https://whattheyforgot.org/
- Parallel processing with a purrr-like interface: https://davisvaughan.github.io/furrr/
- Tidyverse principles: https://principles.tidyverse.org/
- Recursive lists to use in teaching and examples: https://github.com/jennybc/repurrrsive