Visualization, transformation and reporting with the tidyverse

Kirill Müller, Tobias Schieferdecker, Patrick Schratz

24 November 2019, 20:41 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/
- Scripts and installation instructions: https://github.com/krlmlr/vistransrep-proj/tree/master
  - Prepared scripts: https://github.com/krlmlr/vistransrep-proj/tree/master/script
  - Live code: https://github.com/krlmlr/vistransrep-proj/tree/master/live
- 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
##  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)
##  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)
##  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)
##  evaluate      0.14    2019-05-28 [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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  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)
##  vctrs         0.2.0   2019-07-05 [1] CRAN (R 3.6.1)
##  withr         2.1.2   2018-03-15 [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)
##  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
  - ...

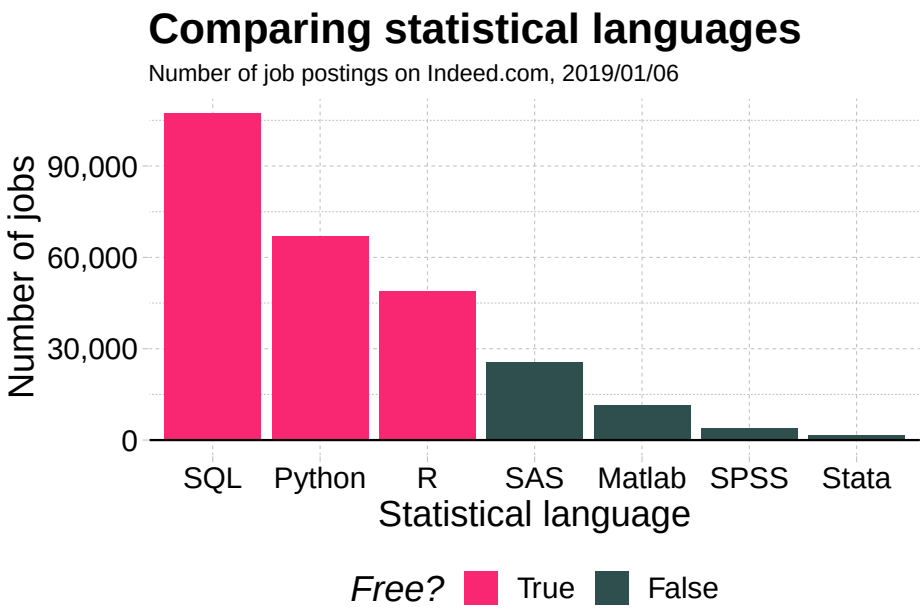
1.2 Overview

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

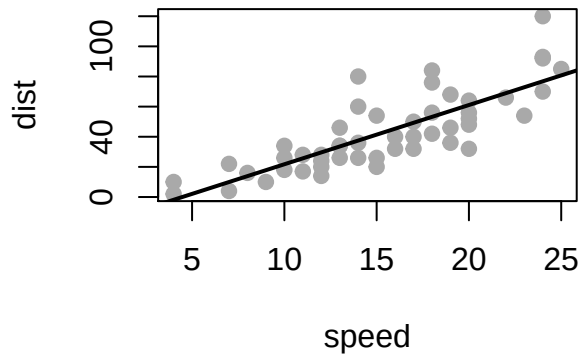
```
fit = lm(dist ~ 1 + speed, data = cars)
summary(fit)
```

```
##
```

```
## 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

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



1.4.3 ggplot2

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

ggplot(data = gapminder,
```

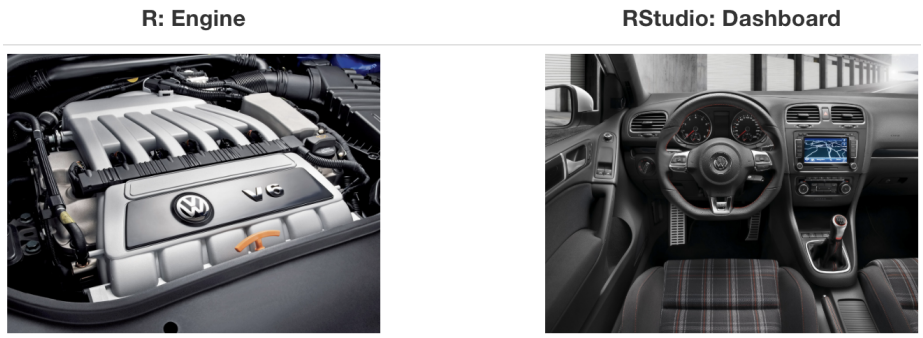
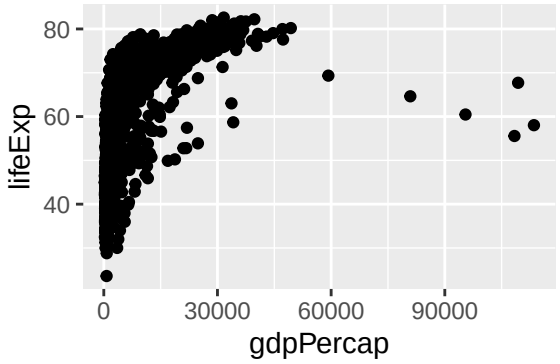


Figure 1.3: Engine vs. dashboard

```
      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

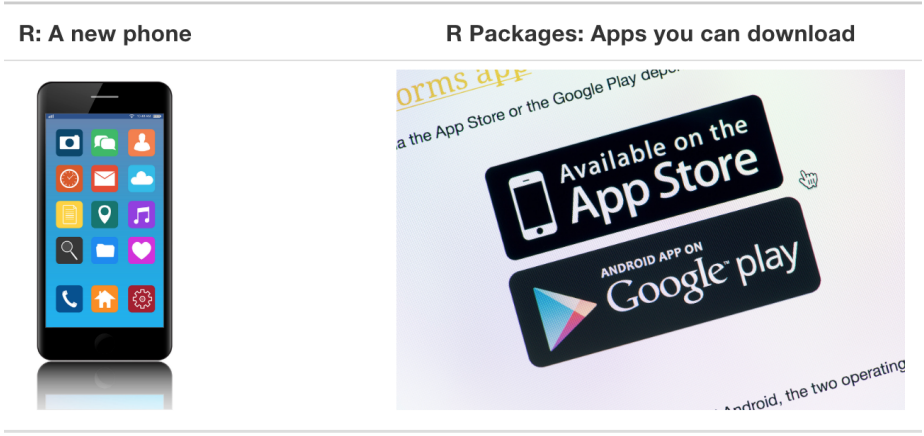


Figure 1.4: R versus R packages

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.

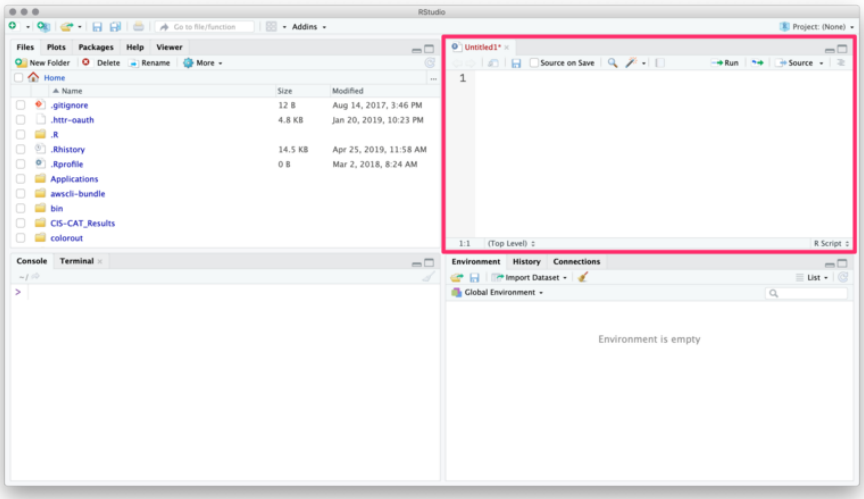


Figure 1.5: Source pane

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.

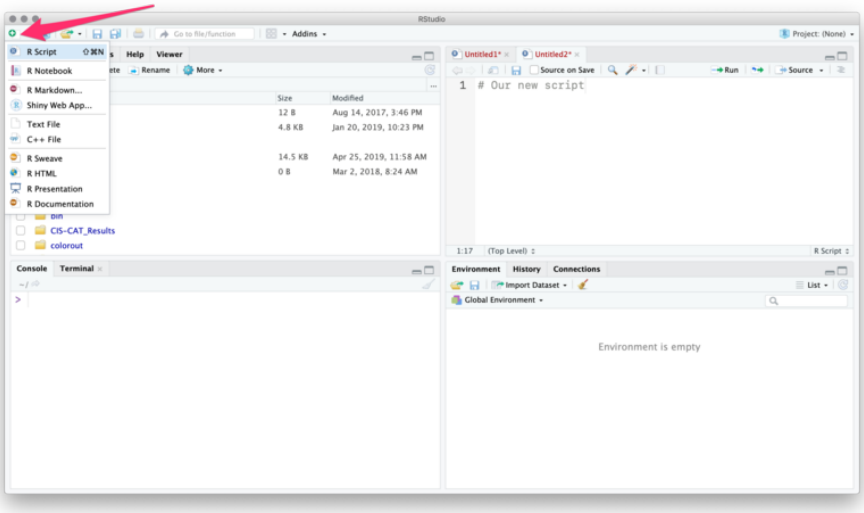


Figure 1.6: New script

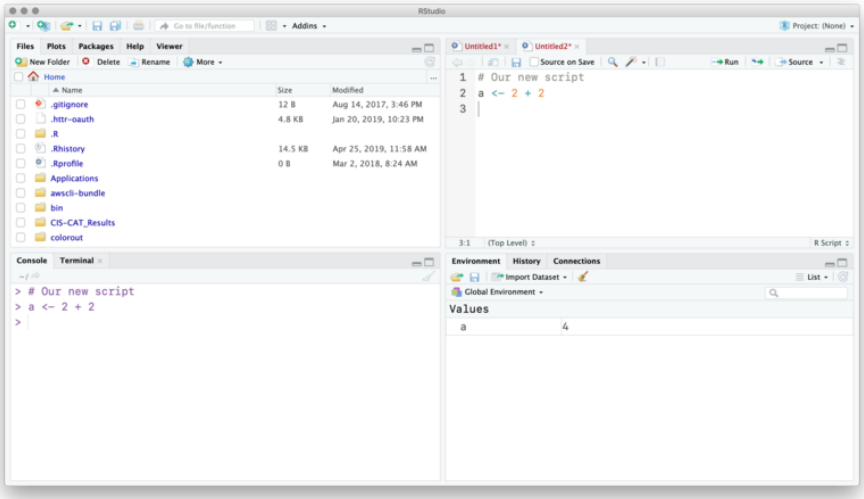


Figure 1.7: Execute commands

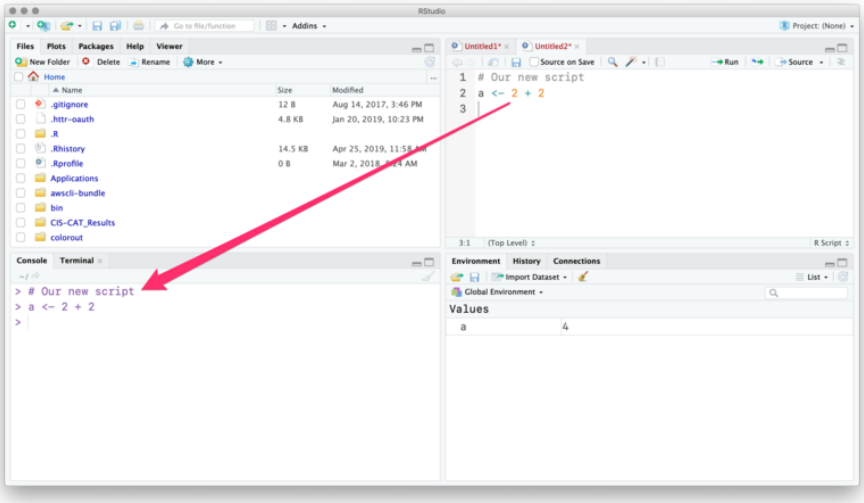


Figure 1.8: Console output

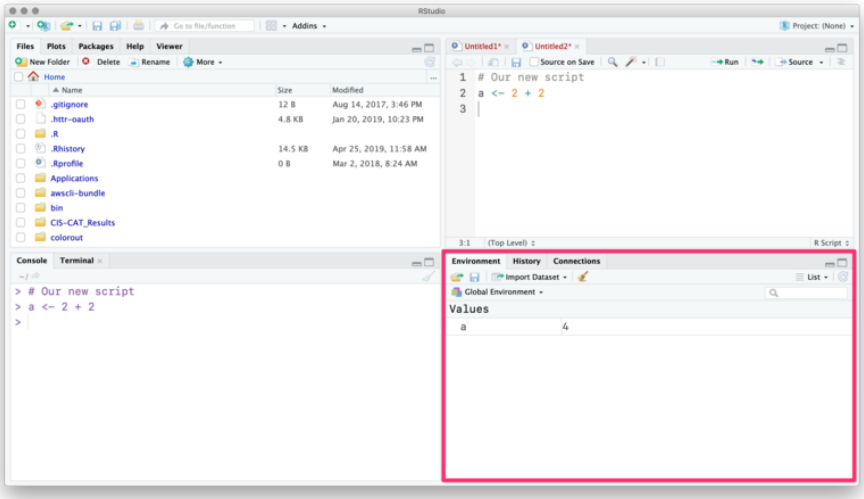


Figure 1.9: Environment pane

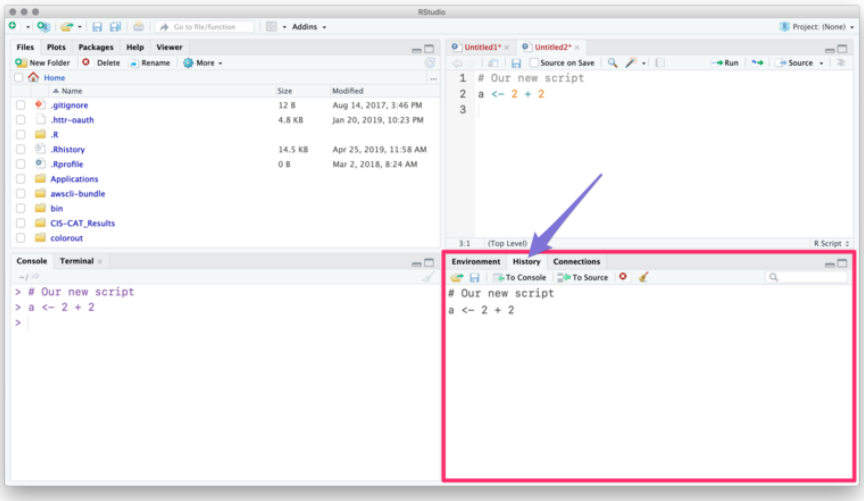


Figure 1.10: History pane

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

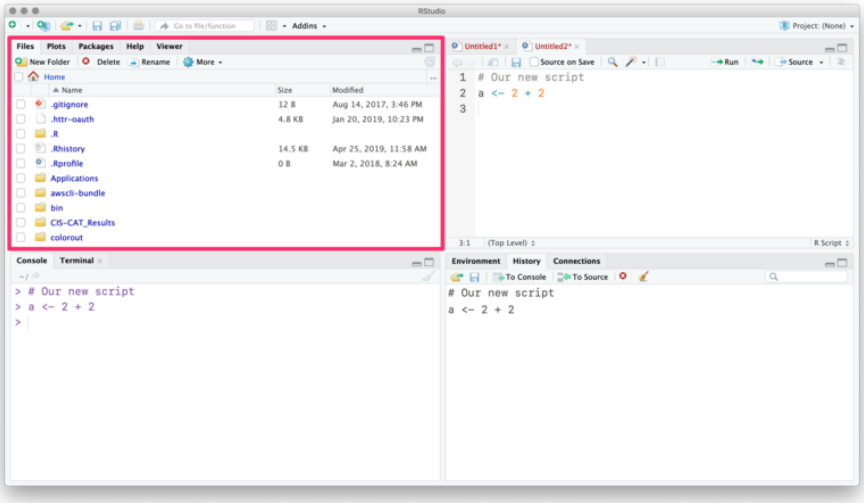


Figure 1.11: Files pane

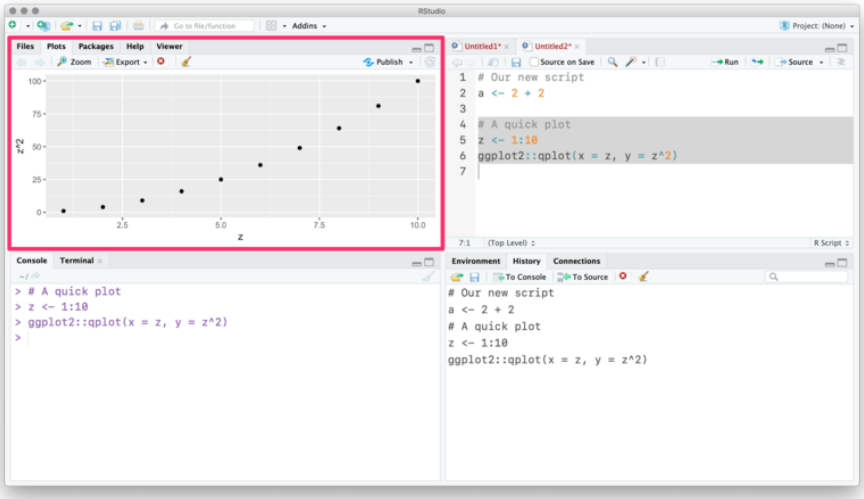


Figure 1.12: Plots pane

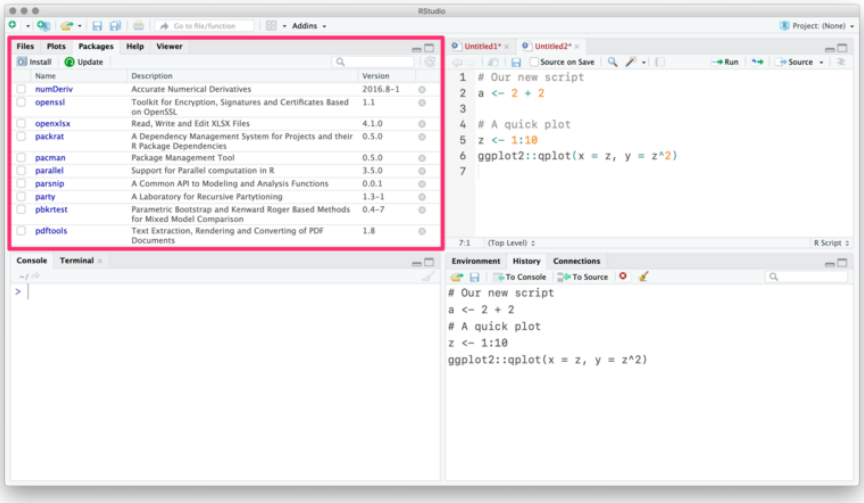


Figure 1.13: Packages pane

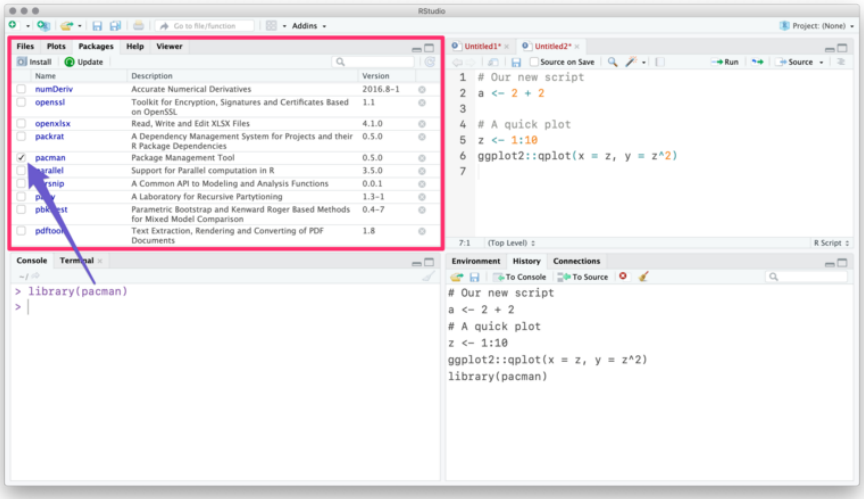


Figure 1.14: Loaded and installed packages

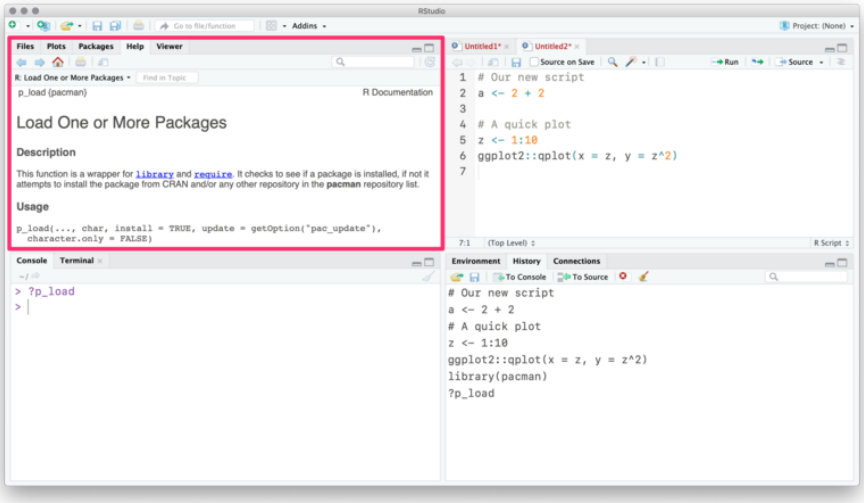


Figure 1.15: Help pane

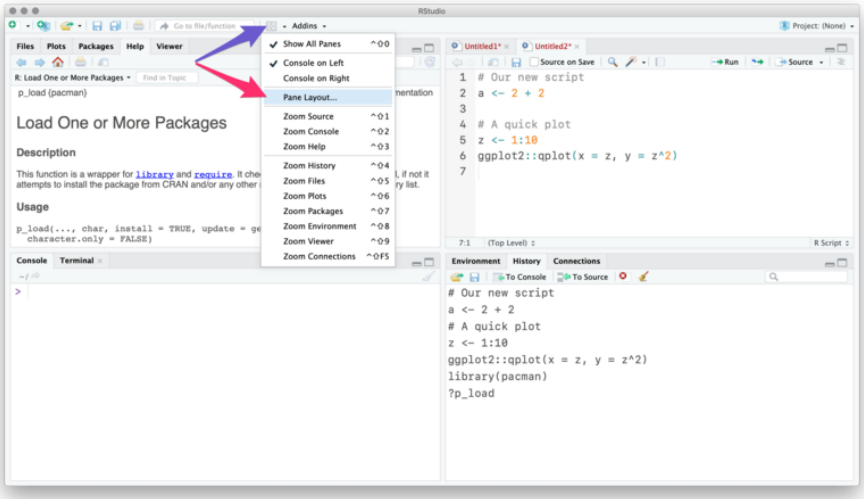


Figure 1.16: Customize layout

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

> Embracing the grammar of graphics.

This chapter discusses plotting with the ggplot2 package.

2.1 Basics for visualisation in R using {ggplot2}

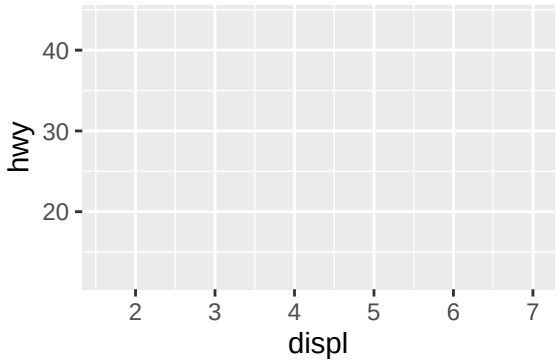
Click here to show setup code.

```
library(tidyverse)
```

In the {tidyverse} the standard package for visualisation 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}.

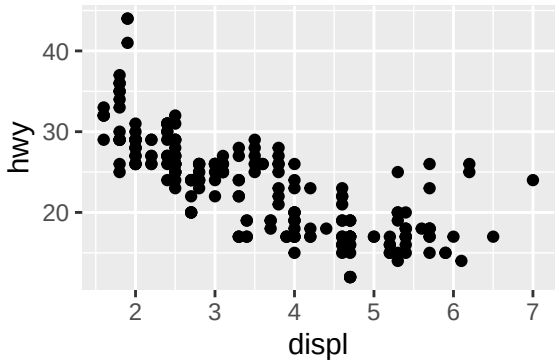
The main function in the package is `ggplot()`, which prepares/creates a graph. By setting the arguments of the function, you can: 1. choose the dataset to be plotted in the argument `data` 2. choose the mapping of the variables to the axes (or further forms of setting apart data) in the argument `mapping`. 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)
)
```



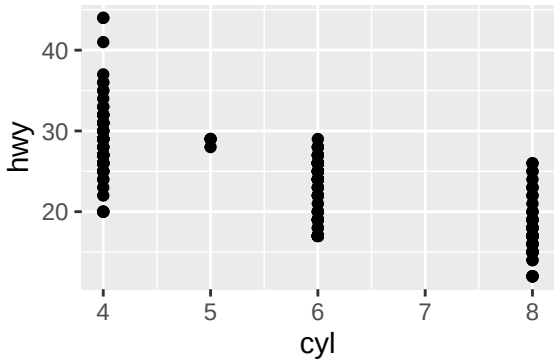
This created only an empty plot, because we did not tell {ggplot2}, which geometry we want to use to display the result. We do this by adding (literally using `+` after the `ggplot()`-call) a different function starting with `geom_` to provide this information.

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



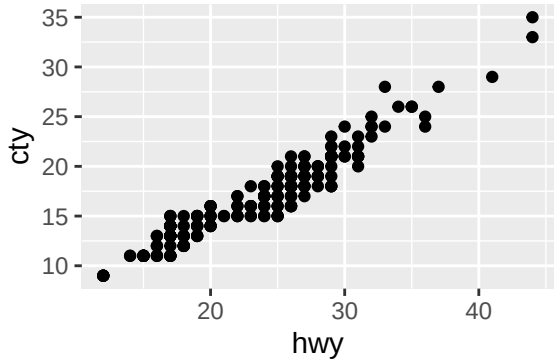
This is maybe the most basic plot you can create. To map a different variable than `disp` to the x-axis, change the respective argument in `aes()`.

```
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()
```



2.2 Tweaks and tricks

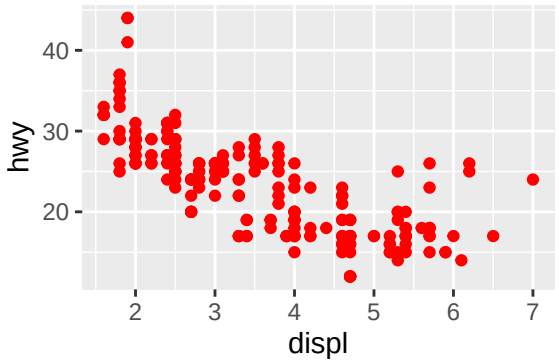
Click here to show setup code.

```
library(tidyverse)
```

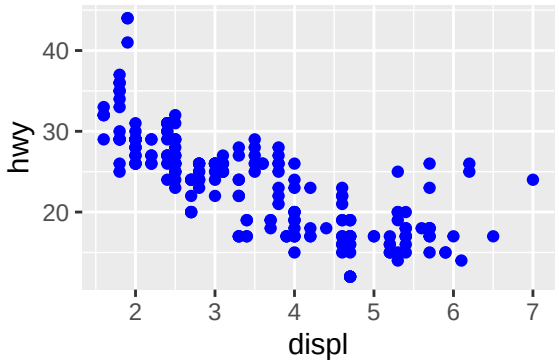
Change the color of the plotted data by setting the `color` argument in the `geom()`-function to a character variable with the color name.

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

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

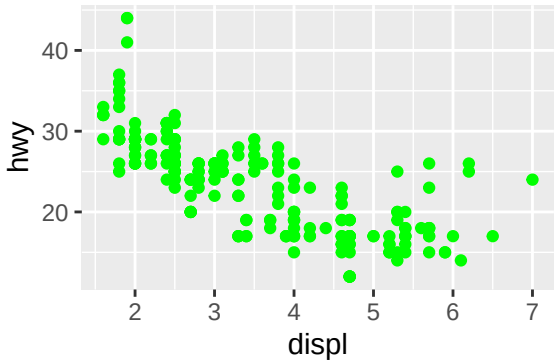



```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(
    color = "blue"
  )
```



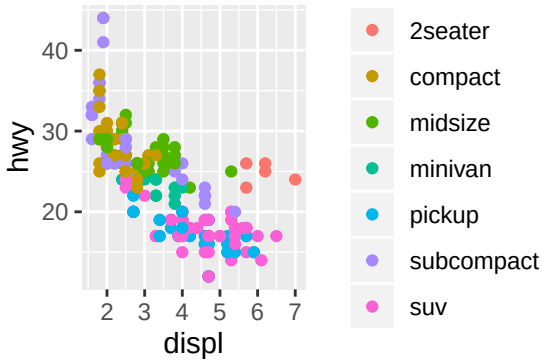

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(
    color = "green"
```

```
  )
```



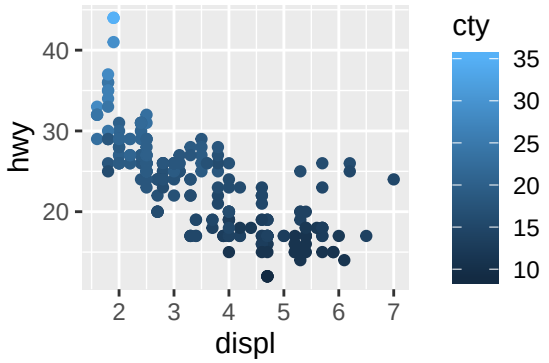
Different colors can be mapped to the values of a variable as a further aesthetic property of the plot. The `class` variable is discrete and leads to a discrete color scale.

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



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

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

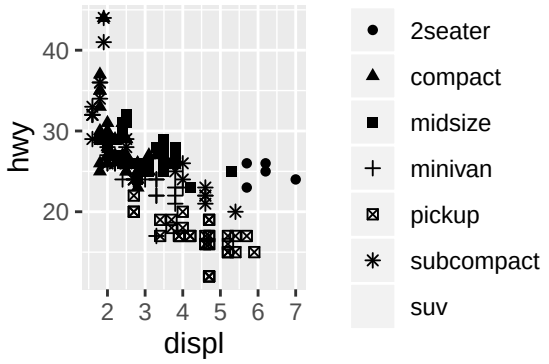


One more degree of freedom is the shape of the symbols to be plotted. The warning indicates that a plot should not use too many different shapes.

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

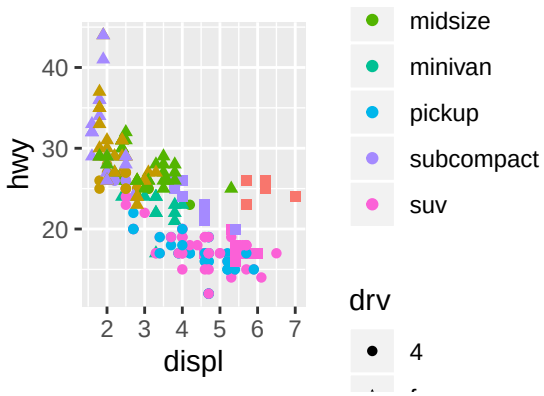
```
## Warning: The shape palette can deal with a maximum of 6
## discrete values because more than 6 becomes difficult
## to discriminate; you have 7. Consider specifying
## shapes manually if you must have them.
```

```
## Warning: Removed 62 rows containing missing values
## (geom_point).
```



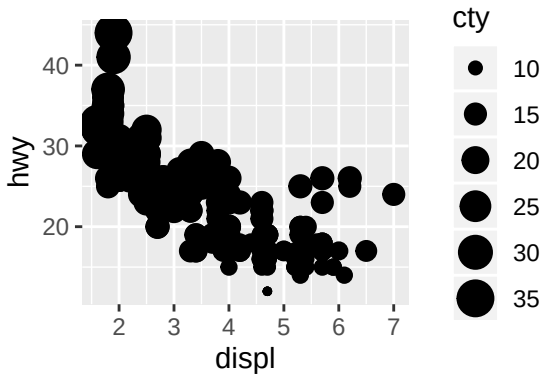
Variables with fewer classes are fine. Color and shape can be combined too.

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



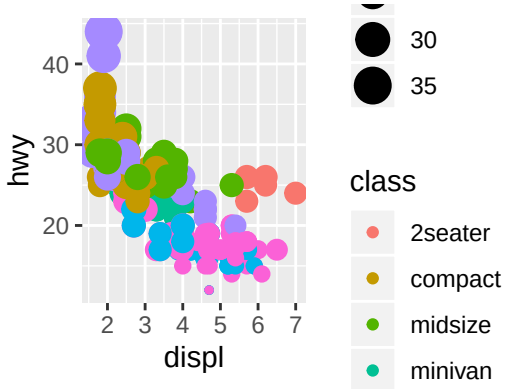
And last but not least, the size of the plotted symbols can be linked to numeric values of the mapped variable.

```
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.

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



Choosing a specific color in the `mapping`-argument of `ggplot()` does not work, since a mapping of a variable to an aesthetic is expected. Let's try passing a color anyway...

```
try(
  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>
```

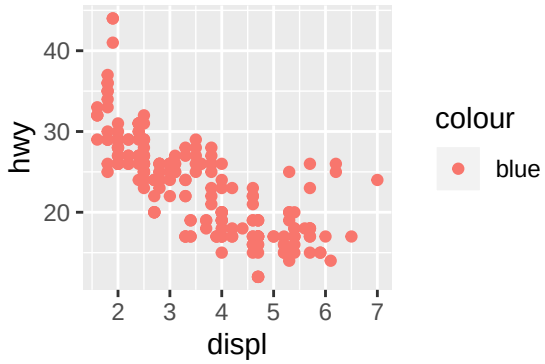
```
"mpg"
```

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

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

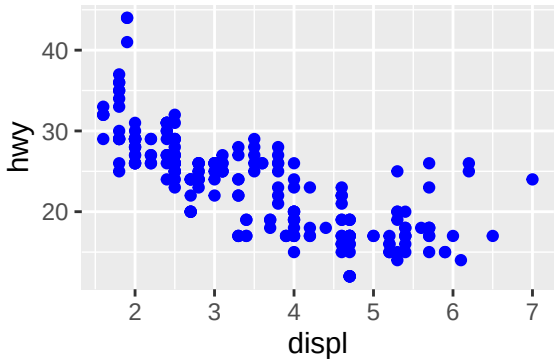
```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
```

```
    y = hwy,
    color = "blue"
  )
) +
  geom_point()
```



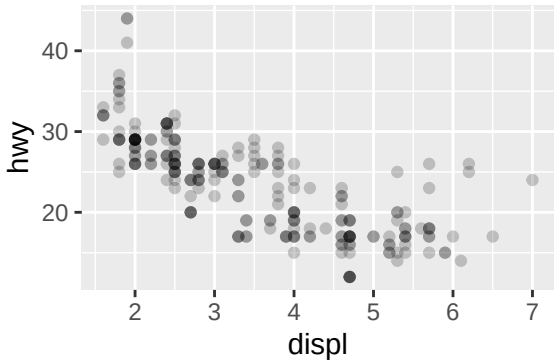
At least there was no error, but now the constant value `blue` is mapped to the first default color of the color mapping, which happens to be red. We could have been fooled, if it had been blue. Manual aesthetics must be specified in the geom:

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



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

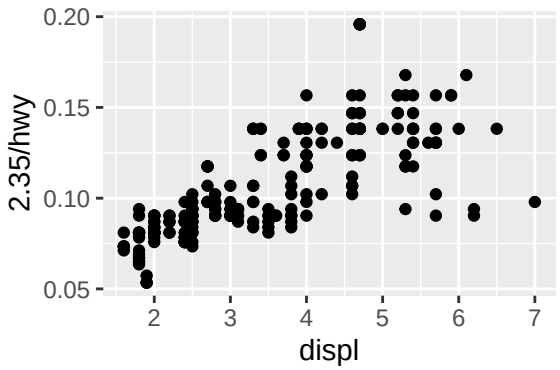
```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = hwy
  )
) +
  geom_point(alpha = 0.2)
```



Within the `aes()`-function you can not only provide the bare variable of the data set, but you can also pass a function of a variable.

```
ggplot(
  data = mpg,
  mapping = aes(
    x = displ,
    y = 2.35 / hwy
  )
```

```
) +
  geom_point()
```



Trying to map aesthetics in the `geom()`-function, does not work.

```
try(
  ggplot(
    data = mpg,
    mapping = aes(
      x = displ,
      y = 2.35 / hwy
    )
  ) +
    geom_point(color = class)
)
```

```
## Error in rep(value[[k]], length.out = n): attempt to replicate an object of type 'bu
```

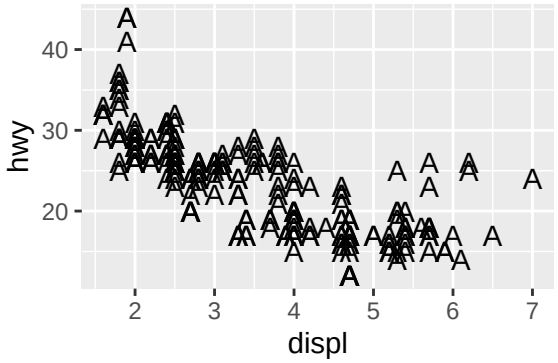
2.3 Labels and layers

Click here to show setup code.

```
library(tidyverse)
```

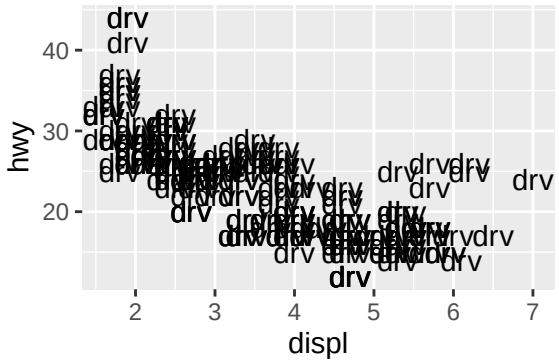
For character variables there is further way of integrating its value to a plot. `geom_text()` takes a `label` argument, which influences the plot in the following way.

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



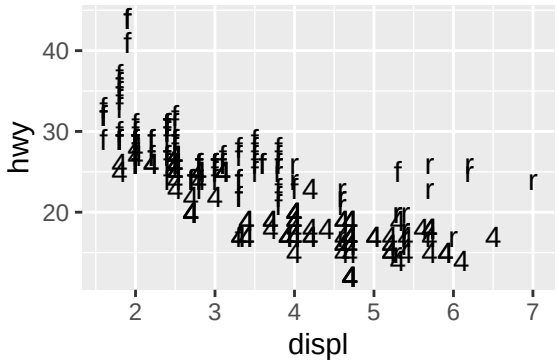
Let's try to map this argument to a variable (here: `drv`) of our dataset in the `mapping` argument of `ggplot()`.

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



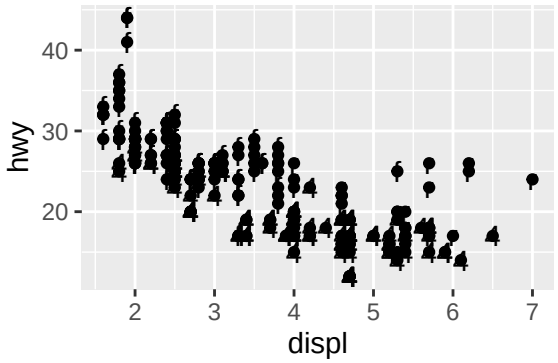
Right, of course we need to pass the variable without quotation marks, otherwise it is interpreted as a (constant) character variable. When changing this, a vector with the values of the variable is passed on to `geom_text()`. This is one way of including the values of character variables in a plot.

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



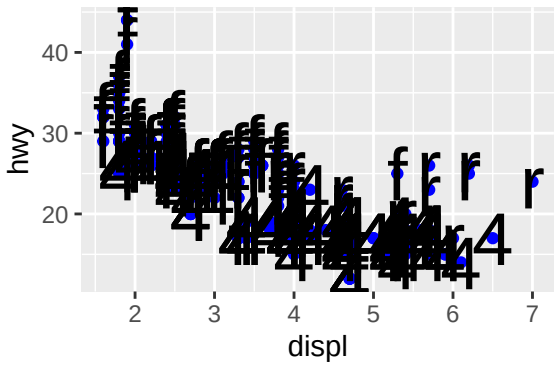
By adding more than one `geom()`-function, more than one geometry is added to the plot.

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



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

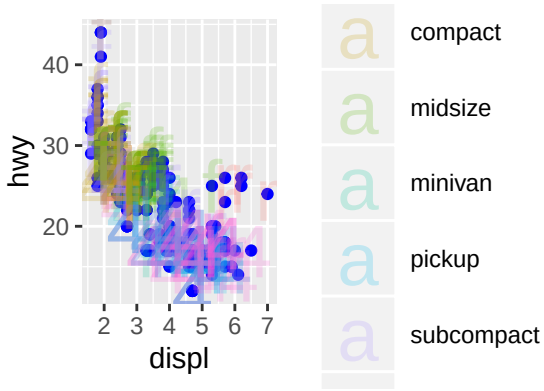
```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy, label = drv)
) +
  geom_point(color = "blue") +
  geom_text(size = 10)
```



It is also possible ot specify the mapping in the `geom()`-function. This way, when adding more than one geometry, you can choose different specifications for the mapping in each of the `geom()`-functions.

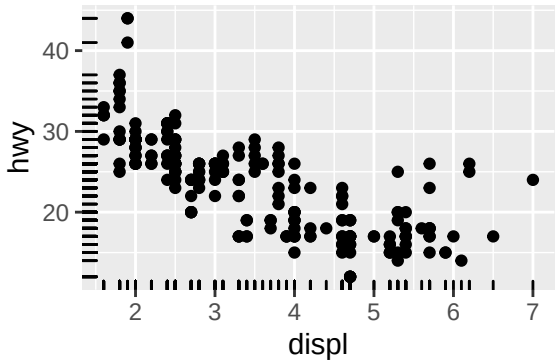
```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point(color = "blue") +
  geom_text(
    mapping = aes(color = class, label = drv),
```

```
    size = 10,
    alpha = 0.2
  )
```



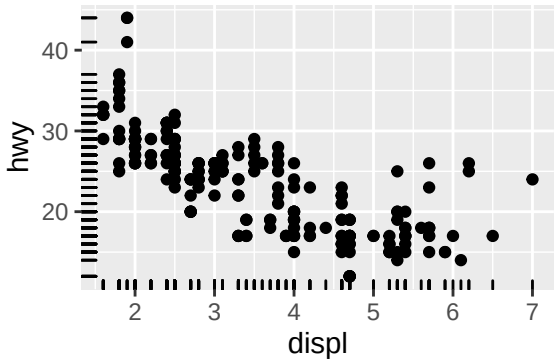
This can be dangerous sometimes though. A good approach is always to define the mapping globally in `ggplot()` when possible.

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



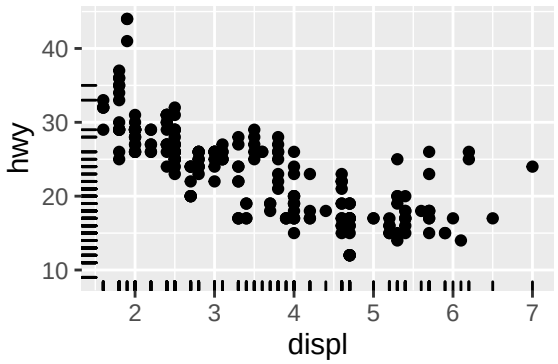
This would be redundant:

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



And here we have a mismatch between the two geometries, since the y-mapping is referring to different variables of the dataset.

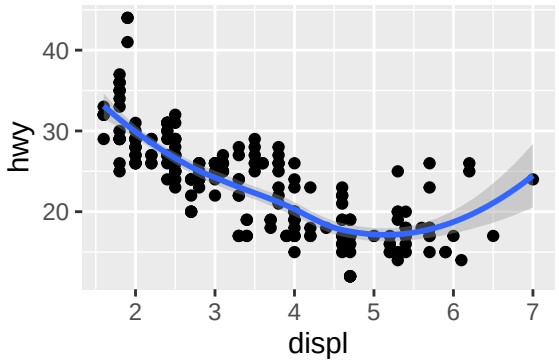
```
ggplot(
  data = mpg
) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_rug(mapping = aes(x = displ, y = cty))
```



Displaying a scatter plot with an overlaid smooth curve fitted to the data is made very easy by the geometry function `geom_smooth()`

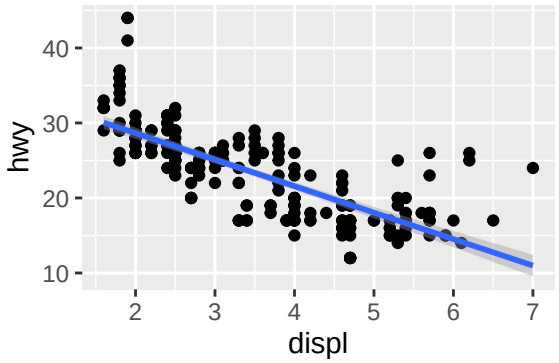
```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy)
) +
  geom_point() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



The default for the smoothing method is set to `auto`, which translates to a LOESS (locally estimated scatterplot smoothing) method, when dealing with fewer than 1000 observations. Changing the method to "lm", a linear model will be fitted and displayed in your plot.

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



2.4 Statistical summaries

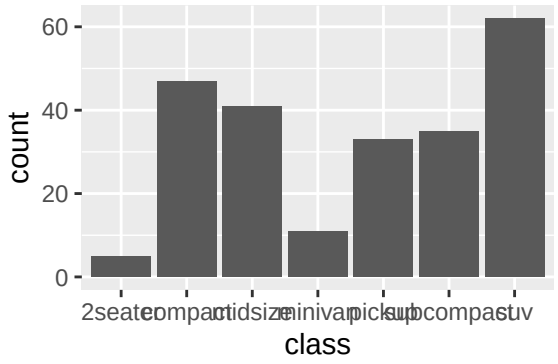
Click here to show setup code.

```
library(tidyverse)
```

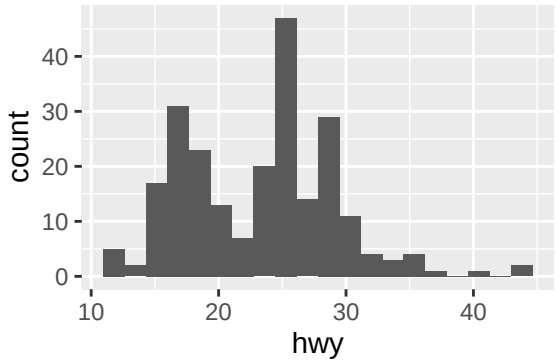
Some `geom()`-functions do some transformations and calculations behind the scenes before the data are visualized. One such function is `geom_smooth()` from

the last section. Also `geom_bar()` does some transformations first. It just needs a mapping for the x-axis. Once that is done, the y-values displayed for each value of the x-variable are the numbers of of times that the x-value occurs in the dataset. So basically the different values are counted.

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

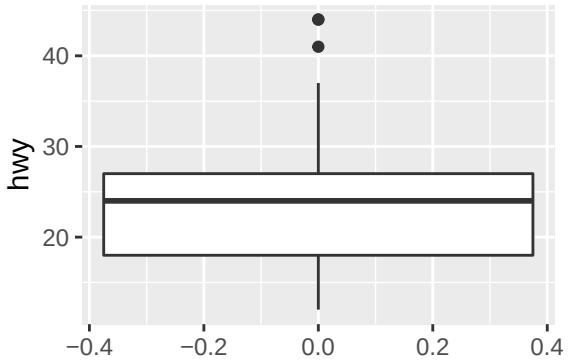



```
ggplot(
  data = mpg,
  mapping = aes(x = hwy)
) +
  geom_histogram(bins = 20)
```

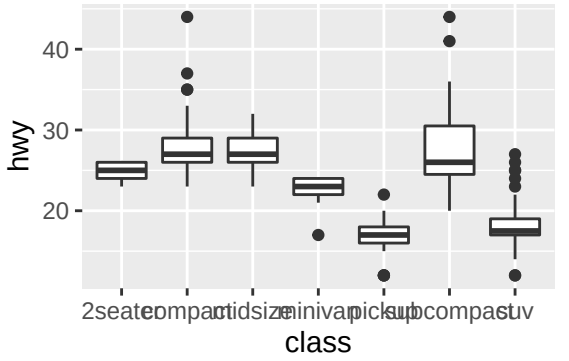



```
ggplot(
  data = mpg,
  mapping = aes(y = hwy)
) +
```

```
  geom_boxplot()
```

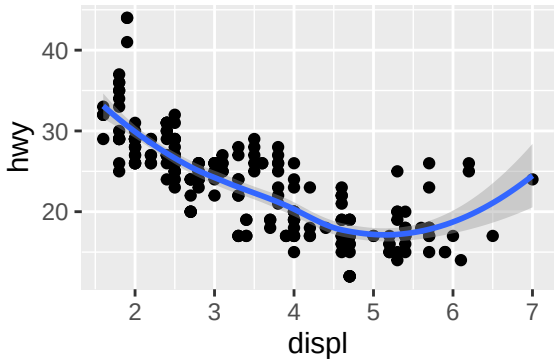



```
ggplot(
  data = mpg,
  mapping = aes(x = class, y = hwy)
) +
  geom_boxplot()
```




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

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



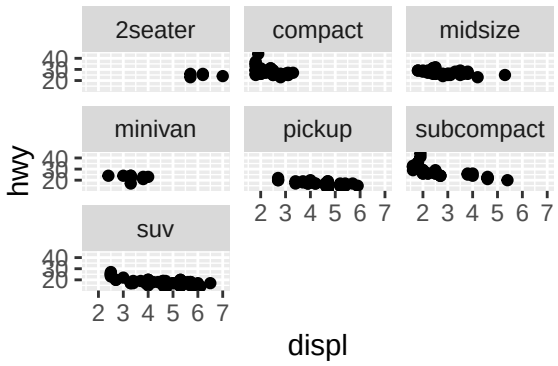
2.5 Facet plots

Click here to show setup code.

```
library(tidyverse)
```

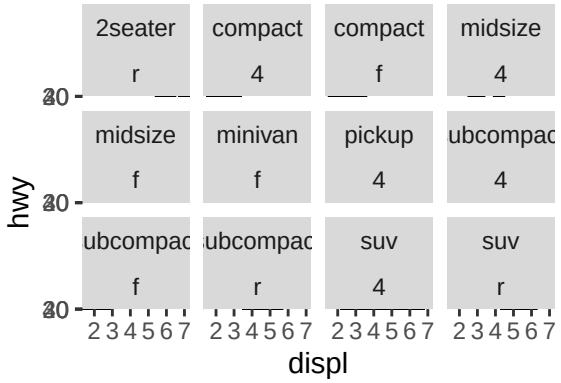
One helpful feature that {ggplot2} offers is the facet plot. This means that for each value of a given variable a different sub-plot is created. The relevant functions for this are `facet_wrap()` and `facet_grid()`. The former aligns all sub-plots from left to right and from top to bottom by default.

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



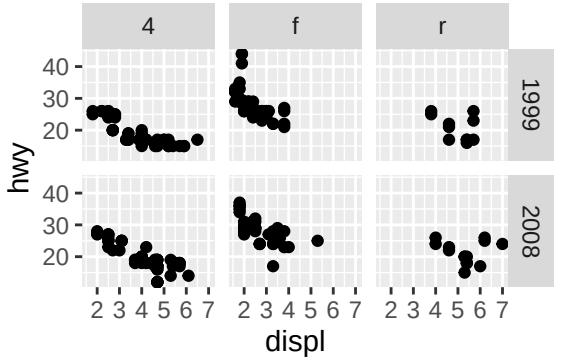
Combinations of variables are also supported, and result in a sub-plot for each unique combination with valid data points.

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



The latter arranges all sub-plots in a grid and requires two variables.

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



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
```

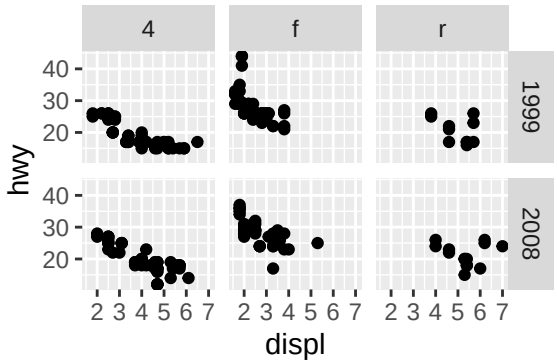
This section is dedicated to show you the basic building blocks (i.e. functions) of data analysis in R within the {tidyverse}. The package providing these is {dplyr}.

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.

Last time we stopped with facet-plots, e.g. like this one here:

```
my_mpg <- mpg

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

```
) +
  geom_point() +
  facet_grid(year ~ drv)
```



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()
```

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

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

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()
```

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

```
flights %>%
  filter(month == 4) %>%
```

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

Invert the sorting by either...

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

... or:

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

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()
```

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()
```

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)
```

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)
```

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
  )
)
```

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()
```

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()
```

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()
```

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()
```

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()
```

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()
```

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")
```

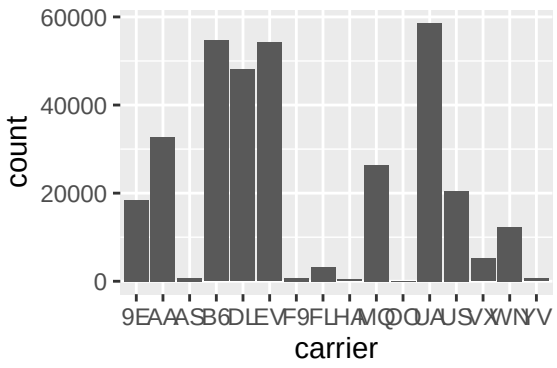
```
## [conflicted] Removing existing preference

## [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
```

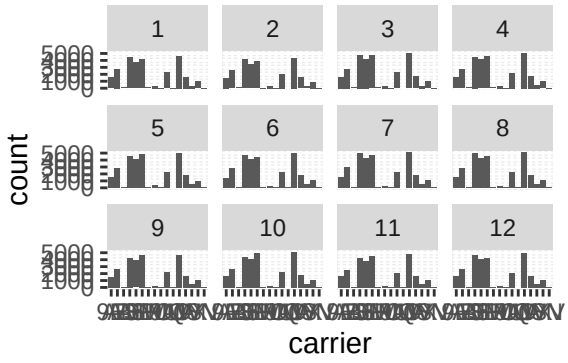
Potentially surprisingly, `mutate()` can also work with the results of a `ggplot()` call. Let's approach this step by step. Here is a basic barplot of `flights$carrier`:

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



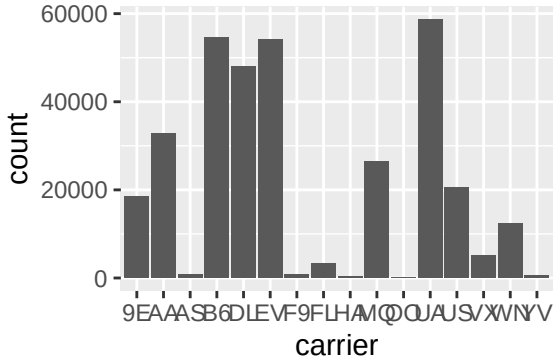
Same with one facet per month:

```
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)))
```

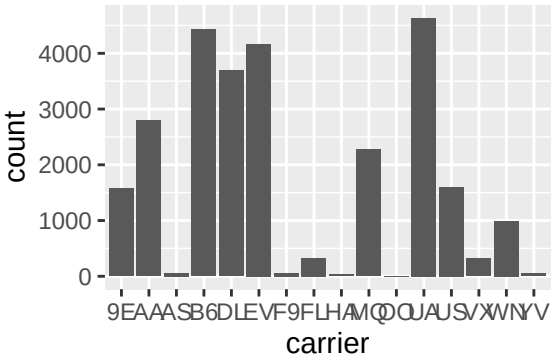
```
  ) %>%
  ungroup()

plot_df
```

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

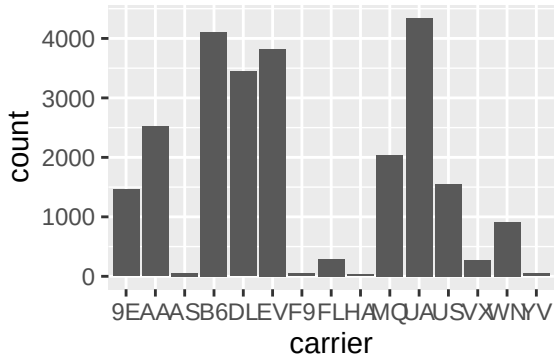
When using `dplyr::pull()` (this function "extracts" a variable from a `data.frame` and returns it as a normal vector), each of the plots will be subsequently displayed in your "Viewer".

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

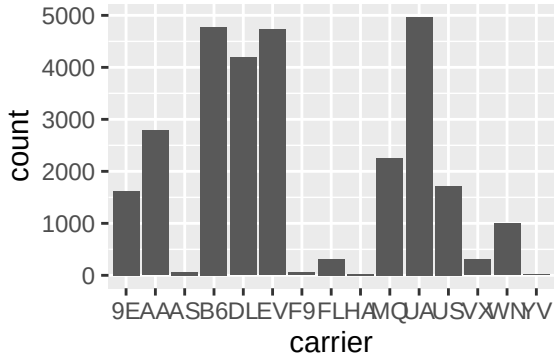
```
## [[1]]
```



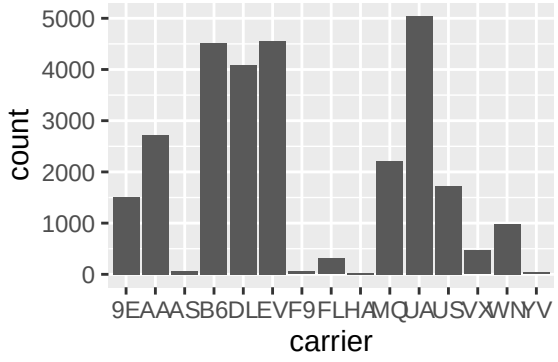

```
##
## [[2]]
```


4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

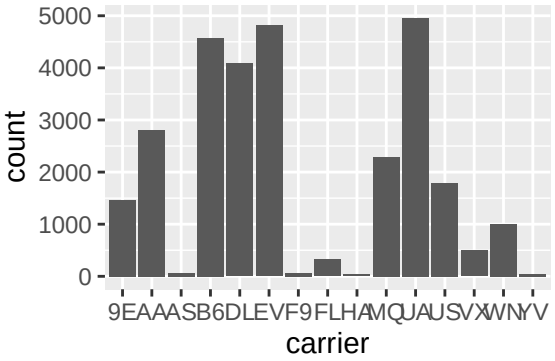
```
##
## [[3]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

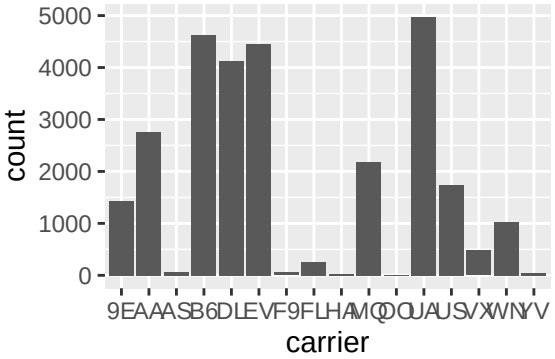
```
##
## [[4]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

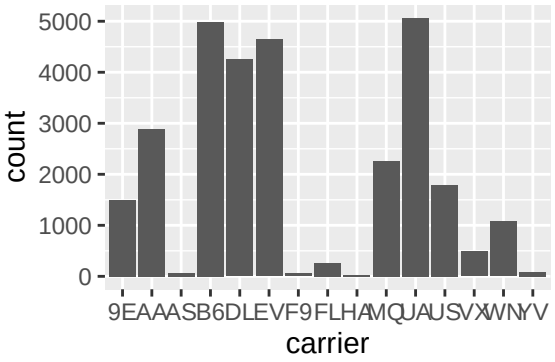
```
##
## [[5]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

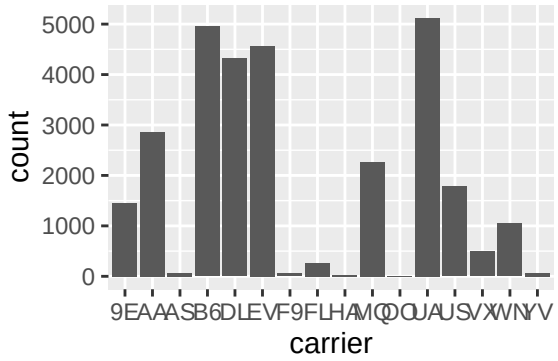
```
##
## [[6]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

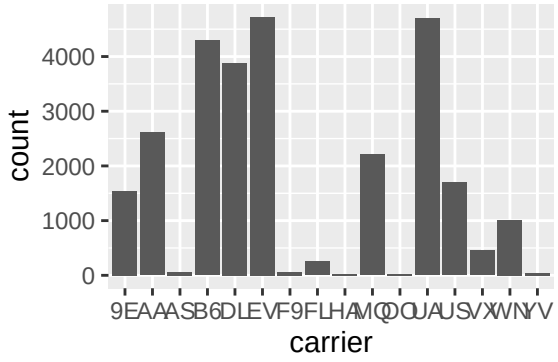
```
##
## [[7]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

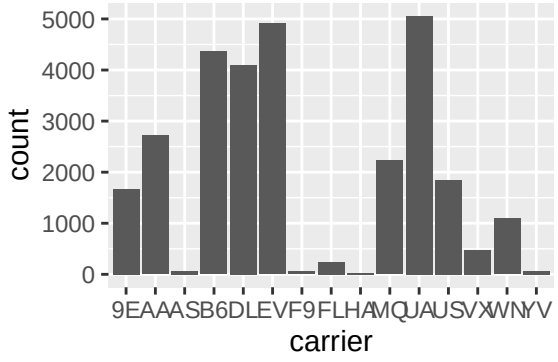
```
##
## [[8]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQDOUAUSVXWNYV
carrier

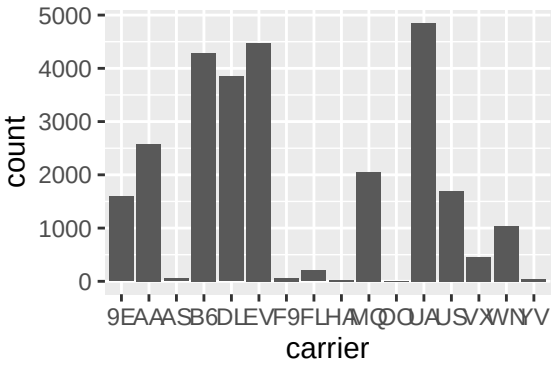
```
##
## [[9]]
```


4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQDOUAUSVXWNYV
carrier

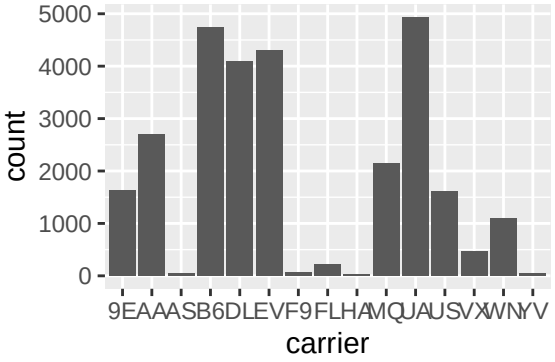
```
##
## [[10]]
```


5000
4000
3000
2000
1000
0
count
9EAAASB6DLEVF9FLHAMQUAUSVXWNYV
carrier

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




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



Use the left arrow to click through the different plots.

Chapter 4

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.

4.1 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")
```

4.2 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.

4.3 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.

4.4 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:

```
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:

```
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 5

- 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