Machine Learning in R: Workshop Series

Introduction to the Tidyverse

Simon Schölzel

Research Team Berens

2020-08-20 (updated: 2020-09-28)

Agenda

1. Learning Objectives

2. Introduction to the `Tidyverse`

1. What is the `Tidyverse`
2. The Concept of Tidy Data

3. `palmerpenguins`: Palmer Archipelago (Antarctica) Penguin Data

4. The Core `Tidyverse` Packages

1. `magrittr`: A Forward-Pipe Operator for R
2. `tibble`: Simple Data Frames
3. `readr`: Read Rectangular Text Data
4. `tidyr`: Tidy Messy Data
5. `dplyr`: A Grammar of Data Manipulation
6. `purrr`: Functional Programming Tools
7. `ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Learning Objectives 💡

This workshop teaches you important tools for working with rectangular data sets in `R`. It introduces and showcases a suite of package which ease your data science workflow in terms of data import, data cleaning, data transformation and data visualization.

More specifically, after this workshop you should

- be familiar with the main tools of the `Tidyverse` and how it differs from `base R`,
- know your way around in working with the core packages of the `Tidyverse` for importing, tidying, transforming and visualizing data,
- be proficient in processing (*non-tidy*) data of any shape and quality,
- be able to produce high-quality, fully customizable visualizations,
- have improved your your overall data literacy.

Introduction to the `Tidyverse`

What is the `Tidyverse`?

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. ~ tidyverse.org
>
> Its primary goal is to facilitate a conversation between a human and a computer about data. ~ Wickham, H., et al. (2019)

Official Tidyverse Hex Sticker

Hadley Wickham - Chief Scientist @ RStudio
Founding father of the `tidyverse`

What is the `Tidyverse`?

> The tidyverse is an opinionated **collection of R packages** designed for data science. All packages share an underlying design philosophy, grammar, and data structures. ~ tidyverse.org

`Tidyverse` core packages:

- `readr`: data import
- `tibble`: modern data frame object
- `stringr`: working with strings
- `forcats`: working with factors
- `tidyr`: data tidying
- `dplyr`: data manipulation
- `ggplot2`: data visualization
- `purrr`: functional programming

What is the `Tidyverse`?

> The tidyverse is an opinionated **collection of R packages** designed for data science. All packages share an underlying design philosophy, grammar, and data structures. ~ tidyverse.org

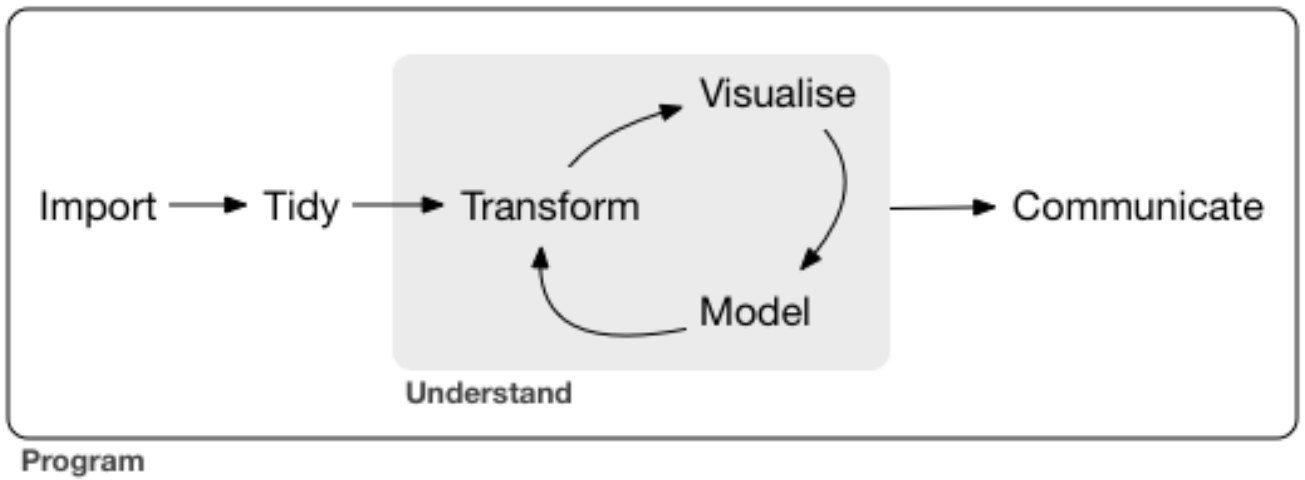
```
install.packages("tidyverse") #equivalent to `install.packages("ggplot2")`, `install.packages("tibble")`, `
library(tidyverse)
```

```
Registered S3 methods overwritten by 'dbplyr':
  method         from
  print.tbl_lazy
  print.tbl_sql
-- Attaching packages --------------------------------------- tidyverse 1.3.0 --
v ggplot2 3.3.2     v purrr   0.3.4
v tibble  3.0.3     v dplyr   1.0.2
v tidyr   1.1.2     v stringr 1.4.0
v readr   1.3.1     v forcats 0.5.0
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

What is the `Tidyverse`?

> The tidyverse is an opinionated collection of R packages designed **for data science**. All packages share an underlying design philosophy, grammar, and data structures. ~ tidyverse.org

These packages are geared towards facilitating the day-2-day data science workflow:



Import: `readr`
Tidy: `tidyr`
Transform: `dplyr`, `forcats`, `stringr`
Visualise: `ggplot2`
Model: `tidymodels` [1]
Communicate: `rmarkdown` [2]
Program: `magrittr`, `purrr`, `tibble`

[1]: For further information refer to our `tidymodels` workshop.
[2]: For further information refer to our `rmarkdown` workshop.

What is the `Tidyverse`?

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an **underlying design philosophy, grammar, and data structures**. ~ tidyverse.org

This underlying design philosophy and grammar boils down to a consistent and easy-to-use API:

- The `tibble` as the core underlying data structure
- Extensive use of the `%>%`-operator for gluing together multiple function calls
- Consistently applied naming conventions (e.g., function names in *snakecase* [Link])
- Consistent structuring of function arguments (e.g., `arg1 = data.frame, arg2 = col names, ...`)
- ...

The `tidyverse` syntax can be viewed as a "dialect" of `R`. When you have familiarized yourself with it, you will be able to easily transfer your knowledge about one function or package to other components of the `tidyverse`. Just like learning a new language.

Note: For further information cf. [2] and [3].

The Concept of Tidy Data

> Tidy data sets are all alike; but every messy data set is messy in its own way.
> ~ Wickham, H./Grolemund, G. (2017)

Tidy Data Principles: The concept of tidy data has been coined by Hadley Wickham in his 2014 paper "Tidy Data"[3]. The concept formulates principles for structuring rectangular, tabular data sets consisting of rows and columns:

1. Each variable is a column.
2. Each observation is a row.
3. Each type of observational unit forms a table.

person	treatment	result
John Smith	a	NA
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

[3]: **Wickham, H. (2014):** Tidy data. Journal of Statistical Software, Vol. 59, No. 10, pp. 1-23.

The Concept of Tidy Data

Violations of the Tidy Data Principles:

1. Column headers are values, not variable names.
2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables.

person	treatmenta	treatmentb
John Smith	NA	2
Jane Doe	16	11

person	col	treatment	result
John Smith	m42	a	NA
Jane Doe	f47	a	16
John Smith	m42	b	2
Jane Doe	f47	b	11

`palmerpenguins`:

Palmer Archipelago (Antarctica) Penguin Data

`palmerpenguins`: Palmer Archipelago (Antarctica) Penguin Data

From here on, to illustrate the functioanilities for the `tidyverse` core packages we use data from the `palmerpenguins` package by Allison Horst.

The package comes with data about penguins observed on islands in the Palmer Archipelago near Palmer Station, Antarctica.

The package requires the `devtools` package to be installed in order to download packages from *GitHub*.

```
#devtools::install_github("allisonhorst/palmerpenguins")
library(palmerpenguins)
```

`palmerpenguins`: Palmer Archipelago (Antarctica) Penguin Data


```
penguins
```

```
> # A tibble: 344 x 8
>    species island bill_length_mm bill_depth_mm flipper_length_~
>    <chr>   <chr>           <dbl>         <dbl>             <dbl>
>  1 Adelie  Torge~           39.1          18.7               181
>  2 Adelie  Torge~           39.5          17.4               186
>  3 Adelie  Torge~           40.3          18                 195
>  4 Adelie  Torge~           NA            NA                  NA
>  5 Adelie  Torge~           36.7          19.3               193
>  6 Adelie  Torge~           39.3          20.6               190
>  7 Adelie  Torge~           38.9          17.8               181
>  8 Adelie  Torge~           39.2          19.6               195
>  9 Adelie  Torge~           34.1          18.1               193
> 10 Adelie  Torge~           42            20.2               190
> # ... with 334 more rows, and 3 more variables: body_mass_g <dbl>,
> #   sex <chr>, year <dbl>
```

`magrittr`:

A Forward-Pipe Operator for R

`magrittr`: A Forward-Pipe Operator for R

`magrittr` comes with a set of operators which improve the readability of your code:

- arrange data operations into an easily readable pipeline of chained commands (left-to-right as opposed to inside-out),
- avoid nested function calls,
- minimize the use of local variable assignments and function definitions, and
- easily add and/or delete steps in your pipeline without breaking the code.

`magrittr`: A Forward-Pipe Operator for R

`magrittr` comes with a set of operators which improve the readability of your code:

- arrange data operations into an easily readable pipeline of chained commands (left-to-right as opposed to inside-out),
- avoid nested function calls,
- minimize the use of local variable assignments and function definitions, and
- easily add and/or delete steps in your pipeline without breaking the code.

The Pipe Operator: `%>%`
The Assignment Operator: `%<>%` (alternative to `<-` at the beginning of a code chunk)
The "Tee" Operator: `%T>%` (used for the side-effect of a function, e.g., plotting or writing)

Note: Find more information about `%>%` by running `vignette("magrittr")`. Type `%>%` using the shortcut: ctrl + shift + M.

`magrittr`: A Forward-Pipe Operator for R

Basic Piping: forward a value or object (LHS) into the next function call (RHS) as **first** argument

```
x %>% f                         #equivalent to f(x)
x %>% f(y)                      #equivalent to f(x, y)
x %>% f %>% g %>% h             #equivalent to h(g(f(x)))
```

`magrittr`: A Forward-Pipe Operator for R

Basic Piping: forward a value or object (LHS) into the next function call (RHS) as **first** argument

```
x %>% f                          #equivalent to f(x)
x %>% f(y)                       #equivalent to f(x, y)
x %>% f %>% g %>% h              #equivalent to h(g(f(x)))
```

Piping with placeholders: forward a value or object (LHS) into the next function call (RHS) as **any** argument

```
x %>% f(.)                       #equivalent to x %>% f
x %>% f(y, .)                    #equivalent to f(y, x)
x %>% f(y, z = .)                #equivalent to f(y, z = x)
x %>% f(y = nrow(.), z = ncol(.)) #equivalent to f(x, y = nrow(x), z = ncol(x))
```

`magrittr`: A Forward-Pipe Operator for R

Basic Piping: forward a value or object (LHS) into the next function call (RHS) as **first** argument

```
x %>% f                          #equivalent to f(x)
x %>% f(y)                       #equivalent to f(x, y)
x %>% f %>% g %>% h              #equivalent to h(g(f(x)))
```

Piping with placeholders: forward a value or object (LHS) into the next function call (RHS) as **any** argument

```
x %>% f(.)                       #equivalent to x %>% f
x %>% f(y, .)                    #equivalent to f(y, x)
x %>% f(y, z = .)                #equivalent to f(y, z = x)
x %>% f(y = nrow(.), z = ncol(.)) #equivalent to f(x, y = nrow(x), z = ncol(x))
```

Building functions and pipelines: a sequence of code starting with the placeholder (`.`) returns a function which can be used to later apply the pipeline to concrete values.

```
f <- . %>% cos %>% sin           #equivalent to f <- function(.) sin(cos(.))
f(20)                            #equivalent to the pipeline 20 %>% cos %>% sin
```

`magrittr`: A Forward-Pipe Operator for R

Question: What is the average body mass index in grams for all penguins observed in the year 2007 (excluding missing values)?

In a pipeless world:

```
mean(subset(penguins, year == 2007)$body_mass_g, na.rm = T)

#alternatively:
peng_bmi_2007 <- subset(penguins, year == 2007)$body_mass_g
mean(peng_bmi_2007, na.rm = T)
```

In a world full of pipes:

```
penguins %>%
  subset(year == 2007) %>%
  .$body_mass_g %>%
  mean(na.rm = T)
```

- Sequential style improves readability!
- Less deciphering of nested function calls!
- No need to store intermediate results!
- Modular modification of pipeline steps!

`tibble`:

Simple Data Frames

`tibble`: Simple Data Frames

`tibble` provides an 'enhanced' data frame object of class `tbl_df`, a so-called *tibble*. *Tibbles* can be created in three different ways:

Create a *tibble* from column vectors with `tibble()`:

```
tibble::tibble(
  x = c("a", "b"),
  y = c(1, 2),
  z = c(T, F)
)
```

Create a *transposed tibble* row by row with `tribble()`:

```
tibble::tribble(
  ~x, ~y,  ~z,
  "a", 1,  T,
  "b", 2,  F
)
```

Create a *tibble* from existing data frames with `as_tibble()` and from named vectors with `enframe()`:

```
df <- data.frame(
  x = c("a", "b"), y = c(1, 2), z = c(T, F)
)
```

```
tibble::as_tibble(df)
```

```
> # A tibble: 2 x 3
>   x         y z
>   <chr> <dbl> <lgl>
> 1 a         1 TRUE
> 2 b         2 FALSE
```

`tibble`: Simple Data Frames

There are three important differences between a `tibble` and a `data.frame` object:

1) **Printing:** `tibble()` prints only the first ten rows and all the columns that fit on the screen as well as a description of the data type.

```
penguins #adjust using print(tibble, n = 5) or options(tibble.print_max = 5)
```

```
> # A tibble: 344 x 8
>    species island bill_length_mm bill_depth_mm flipper_length_~
>    <chr>   <chr>           <dbl>         <dbl>             <dbl>
>  1 Adelie  Torge~           39.1          18.7               181
>  2 Adelie  Torge~           39.5          17.4               186
>  3 Adelie  Torge~           40.3          18                 195
>  4 Adelie  Torge~           NA            NA                  NA
>  5 Adelie  Torge~           36.7          19.3               193
>  6 Adelie  Torge~           39.3          20.6               190
>  7 Adelie  Torge~           38.9          17.8               181
>  8 Adelie  Torge~           39.2          19.6               195
>  9 Adelie  Torge~           34.1          18.1               193
> 10 Adelie  Torge~           42            20.2               190
> # ... with 334 more rows, and 3 more variables: body_mass_g <dbl>,
> #   sex <chr>, year <dbl>
```

`tibble`: Simple Data Frames

There are three important differences between a `tibble` and a `data.frame` object:

2) **Subsetting:** Subsetting a `tibble` (`[]`) always returns another `tibble` and never a vector (in contrast to standard `data.frame` objects).

```
penguins[,"species"] %>%
  class
```

```
> [1] "tbl_df"     "tbl"        "data.frame"
```

```
penguins %>%
  as.data.frame() %>%
  .[,"species"] %>%
  class
```

```
> [1] "character"
```

3) **Partial Matching:** Subsetting with `tibbles` does not allow for partial matching, i.e. you must always provide the whole column name.

Note: Find more information about `tibbles` by running `vignette("tibble")`.

readr:

Read Rectangular Text Data

`readr`: Read Rectangular Text Data

`readr` provides read and write functions for multiple different file formats:

- `read_delim()`: general delimited files
- `read_csv()`: comma separated files
- `read_tsv()`: tab separated files
- `read_fwf()`: fixed width files
- `read_table()`: white-space separated files
- `read_log()`: web log files

Conveniently, the `write_*()` functions work analog. In addition, use the `readxl` package for Excel files or the `haven` package for Stata files.

Note: In most European countries MS Excel is using `;` as the common delimiter, which can be accountet for by simply switching to the `read_csv2()` function.

`readr`: Read Rectangular Text Data

Let's try it out by reading in some penguins data (note that the output of any `read_*()` function is a `tibble` object.). For the purpose of illustrating the `readr` package the `penguins` data is written to a csv-file a priori using `write_csv()`.

```
penguins %>%
  readr::write_csv("./data/penguins.csv")
```

```
penguins <- readr::read_csv("./data/penguins.csv")
```

```
> Parsed with column specification:
> cols(
>   species = col_character(),
>   island = col_character(),
>   bill_length_mm = col_double(),
>   bill_depth_mm = col_double(),
>   flipper_length_mm = col_double(),
>   body_mass_g = col_double(),
>   sex = col_character(),
>   year = col_double()
> )
```

`readr`: Read Rectangular Text Data

Note that `readr` prints the column specifications after importing. By default, `readr` tries to infer the column type (e.g., integer, numeric, character, factor, date, logical) from the first 1000 rows and parses the columns accordingly using the built-in `parse_()` functions.

- Try to make column specifications explicit! You likely get more familiar with your data and see warnings if something changes unexpectedly.

```
readr::read_csv(
  "./data/penguins.csv",
    col_types = cols(
      species = col_character(),
      year = col_datetime(format = "%Y"),
      island = col_skip()
    )
  )
```

`readr`: Read Rectangular Text Data

Note that `readr` prints the column specifications after importing. By default, `readr` tries to infer the column type (e.g., integer, numeric, character, factor, date, logical) from the first 1000 rows and parses the columns accordingly using the built-in `parse_()` functions.

- Parsing only the first 1000 rows is efficient but can lead to erroneous guesses.

```
readr::read_csv(file = "./data/penguins.csv", guess_max = 1001)
```

Note: Find more information and functions on the `readr` cheat sheet.

`readr`: Read Rectangular Text Data

Eventually, you would want to cease using `.xlsx` and `.csv` files as they are not capable of reliably storing your metadata (e.g., data types).

`write_rds()` and `read_rds()` provide a nice alternative for serializing your `R` objects (e.g., `tibbles`, models) and storing them as `.rds` files.

```
penguins <- read_rds(path = "./data/penguins.rds")
```

```
> Rows: 344
> Columns: 8
> $ species           <chr> "Adelie", "Adelie", ...
> $ island            <chr> "Torgersen", "Torgersen", ...
> $ bill_length_mm    <dbl> 39.1, 39.5, ...
> ...
```

Note that

- `write_rds()` can only be used to save one object at a time,
- a loaded `.rds` file must be stored into a new variable, i.e. given a new name,
- `read_rds()` preserves data types.

`tidyr`:

Tidy Messy Data

`tidyr`: Tidy Messy Data

`tidyr` provides several functions that help you to bring your data into *tidy data* format (cf. tidy data), i.e. change the structure of your data set.

Let's again start with our `penguins` data set by looking at the first six observations:

```
penguins %>% head(6)
```

```
> # A tibble: 6 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Torge~           39.1          18.7               181
> 2 Adelie  Torge~           39.5          17.4               186
> 3 Adelie  Torge~           40.3          18                 195
> 4 Adelie  Torge~           NA            NA                  NA
> 5 Adelie  Torge~           36.7          19.3               193
> 6 Adelie  Torge~           39.3          20.6               190
> # ... with 3 more variables: body_mass_g <dbl>, sex <chr>,
> #   year <dbl>
```

Dimensionality: [344, 8]

`tidyr`: Tidy Messy Data

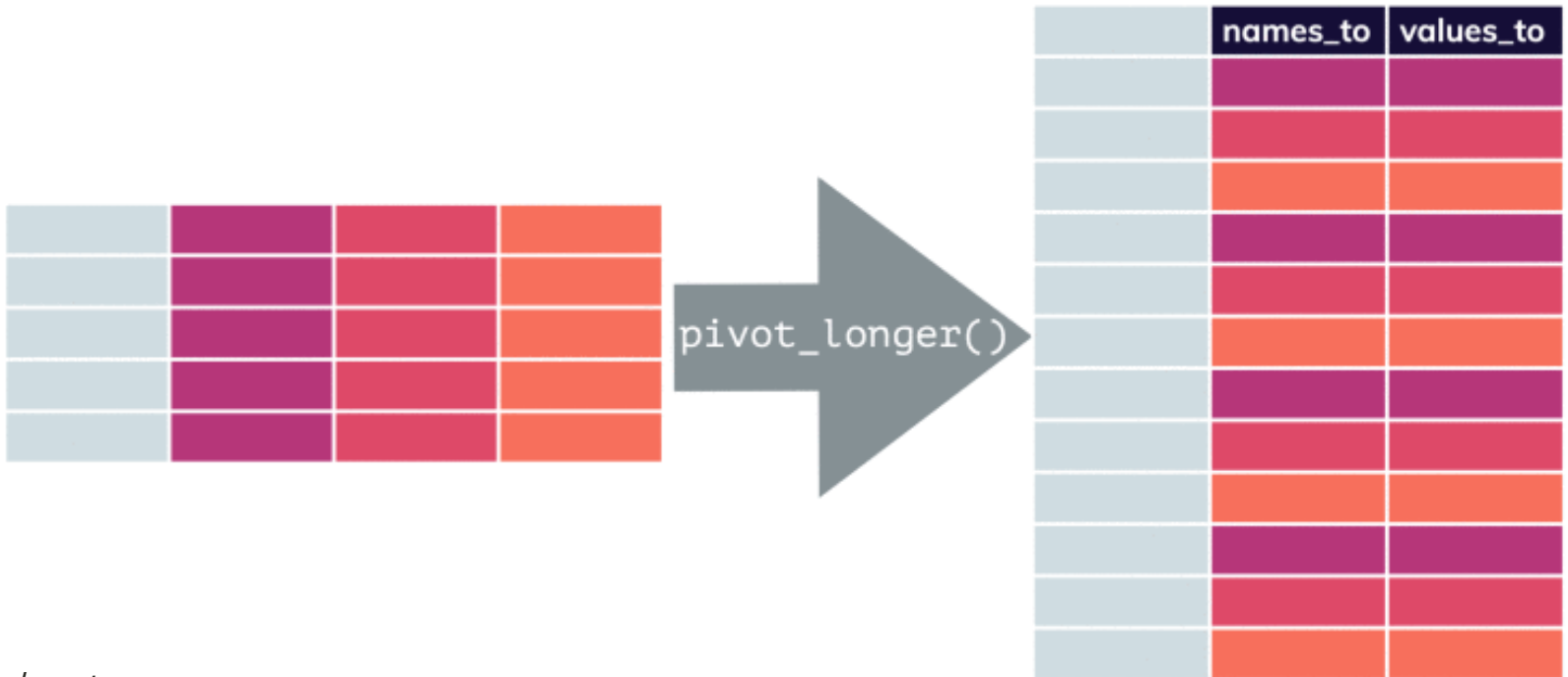
"Pivotting" which converts between long and wide forms using `pivot_longer()` and `pivot_wider()`:

```
tidy_penguins <- penguins %>%
  tidyr::pivot_longer(
    cols = contains("_mm"),
    names_to = "meas_type", values_to = "measurement"
  )

tidy_penguins %>% head(6)
```

```
> # A tibble: 6 x 7
>   species island   body_mass_g sex    year meas_type     measurement
>   <chr>   <chr>          <dbl> <chr> <dbl> <chr>               <dbl>
> 1 Adelie  Torgers~        3750 male   2007 bill_length_~        39.1
> 2 Adelie  Torgers~        3750 male   2007 bill_depth_mm        18.7
> 3 Adelie  Torgers~        3750 male   2007 flipper_leng~       181
> 4 Adelie  Torgers~        3800 fema~  2007 bill_length_~        39.5
> 5 Adelie  Torgers~        3800 fema~  2007 bill_depth_mm        17.4
> 6 Adelie  Torgers~        3800 fema~  2007 flipper_leng~       186
```

Dimensionality: [1032, 7]

`tidyr`: Tidy Messy Data



Source: Allison Hill.
Note: Find more information about `pivot_()` in the pivoting vignette.

`tidyr`: Tidy Messy Data

"Nesting" which groups similar data such that each group becomes a single row in a data frame (the structure is then referred to as *nested data frame*).

```
nested_penguins <- penguins %>%
  tidyr::nest(
    nested_data = c(island, bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g, sex)
  )
```

```
> # A tibble: 9 x 3
>   species    year nested_data
>   <chr>     <dbl> <list>
> 1 Adelie     2007 <tibble [50 x 6]>
> 2 Adelie     2008 <tibble [50 x 6]>
> 3 Adelie     2009 <tibble [52 x 6]>
> 4 Gentoo     2007 <tibble [34 x 6]>
> 5 Gentoo     2008 <tibble [46 x 6]>
> 6 Gentoo     2009 <tibble [44 x 6]>
> 7 Chinstrap  2007 <tibble [26 x 6]>
> 8 Chinstrap  2008 <tibble [18 x 6]>
> 9 Chinstrap  2009 <tibble [24 x 6]>
```

- `nest()` produces a nested data frame with one row per species and year.
- The `nested_data` column contains data frame objects with six columns each and a varying amount of observations.
- Access individual tables from the `nested_data` column using `dpylr::pull()`.
- Revert nesting using `unnest()`.

`tidyr`: Tidy Messy Data

“Rectangling” which disentangles nested data structures (e.g., JSON, HTML) and brings it into *tidy* format (see `unnest()`, `unnest_longer()` and `unnest_wider()`).

"Splitting" and **"combining"** to transform a single character column into multiple columns and vice versa.

```
penguins %>%
  tidyr::unite(col = "spec_gender", c(species, sex), sep = "_", remove = T) %>%
  head(6)
```

```
> # A tibble: 6 x 7
>   spec_gender island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>       <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie_male Torge~           39.1          18.7               181
> 2 Adelie_fem~ Torge~           39.5          17.4               186
> 3 Adelie_fem~ Torge~           40.3          18                 195
> 4 Adelie_NA   Torge~           NA            NA                  NA
> 5 Adelie_fem~ Torge~           36.7          19.3               193
> 6 Adelie_male Torge~           39.3          20.6               190
> # ... with 2 more variables: body_mass_g <dbl>, year <dbl>
```

Revert union using the `separate()` function. Alternatively, use `separate_rows()` if there are two observations for the same variable in one cell.

Excursus: The RStudio "Help"-Pane

02:00

`tidyr`: Tidy Messy Data

"Handling missing values" by making implicit `NA` explicit (`complete()`), by making explicit `NA` implicit (`drop_na()`) or by replacing `NA` with the next/previous value (`fill()`) or a known value (`replace_na()`).

```
incompl_penguins
```

```
> # A tibble: 4 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Gentoo     2008  41.2
> 4 Chinstrap  2007  NA
```

```
incompl_penguins %>%
  tidyr::complete(
    species, year, fill = list(value = NA)
)
```

```
> # A tibble: 6 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Chinstrap  2007  NA
> 4 Chinstrap  2008  NA
> 5 Gentoo     2007  NA
> 6 Gentoo     2008  41.2
```

`tidyr`: Tidy Messy Data

"Handling missing values" by making implicit `NA` explicit (`complete()`), by making explicit `NA` implicit (`drop_na()`) or by replacing `NA` with the next/previous value (`fill()`) or a known value (`replace_na()`).

```
incompl_penguins
```

```
> # A tibble: 4 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Gentoo     2008  41.2
> 4 Chinstrap  2007  NA
```

```
incompl_penguins %>%
  tidyr::drop_na(value)
```

```
> # A tibble: 3 x 3
>   species  year value
>   <chr>   <dbl> <dbl>
> 1 Adelie   2007  41.9
> 2 Adelie   2008  78.4
> 3 Gentoo   2008  41.2
```

`tidyr`: Tidy Messy Data

"Handling missing values" by making implicit `NA` explicit (`complete()`), by making explicit `NA` implicit (`drop_na()`) or by replacing `NA` with the next/previous value (`fill()`) or a known value (`replace_na()`).

```
incompl_penguins
```

```
> # A tibble: 4 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Gentoo     2008  41.2
> 4 Chinstrap  2007  NA
```

```
incompl_penguins %>%
  tidyr::fill(value, .direction = "down")
```

```
> # A tibble: 4 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Gentoo     2008  41.2
> 4 Chinstrap  2007  41.2
```

`tidyr`: Tidy Messy Data

"Handling missing values" by making implicit `NA` explicit (`complete()`), by making explicit `NA` implicit (`drop_na()`) or by replacing `NA` with the next/previous value (`fill()`) or a known value (`replace_na()`).

```
incompl_penguins
```

```
> # A tibble: 4 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Gentoo     2008  41.2
> 4 Chinstrap  2007  NA
```

```
incompl_penguins %>%
  tidyr::replace_na(
    replace = list(
      value = mean(.$value, na.rm = T)
    )
  )
```

```
> # A tibble: 4 x 3
>   species    year value
>   <chr>     <dbl> <dbl>
> 1 Adelie     2007  41.9
> 2 Adelie     2008  78.4
> 3 Gentoo     2008  41.2
> 4 Chinstrap  2007  53.8
```

Note: Find more information and functions on the `tidyr` cheat sheet.

`dplyr`:

A Grammar of Data Manipulation

`dplyr`: A Grammar of Data Manipulation

`dplyr` provides a set of functions for manipulating data frame objects (e.g., `tibbles`) while relying on a consistent grammar. Functions are intuitively represented by "verbs" that reflect the underlying operations and always output a new or modified data frame object.

Operations on rows:

- `filter()` picks rows that meet logical criteria
- `slice()` picks rows based on location
- `arrange()` changes the order of rows

Operations on columns:

- `select()` picks respectively drops certain columns
- `rename()` changes the column names
- `relocate()` changes the order of columns
- `mutate()` changes the values of columns and/or creates new columns

Operations on grouped data:

- `group_by()` based on a set of columns
- `summarise()` reduces a group into a single row

`dplyr`: A Grammar of Data Manipulation

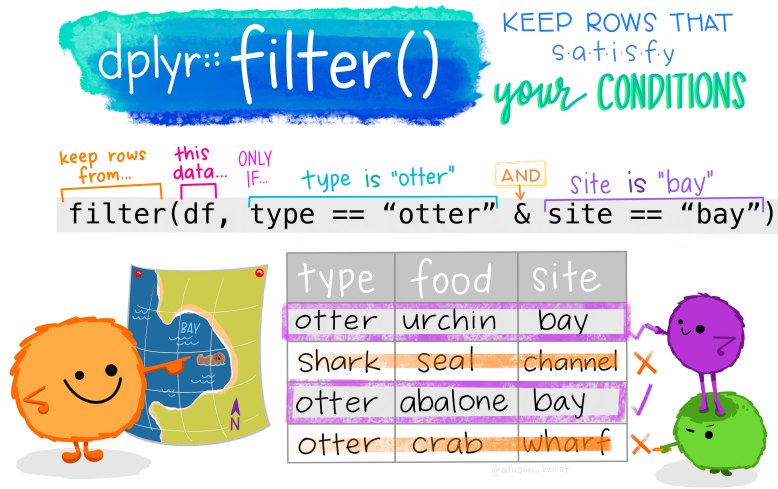
Operations on rows: `filter()` picks rows that meet a logical criteria, i.e. condition

Filter for all penguins of type "Adelie":

```
penguins %>%
  filter(species == "Adelie")
```

Filter for all penguins where the is a missing value in the `bill_length_mm` measurement:

```
penguins %>%
  filter(is.na(bill_length_mm) == T)
```



Filter for all penguins observed in 2008 and where the body mass index lies between 3800 and 4000 grams:

```
penguins %>%
  filter(between(body_mass_g, 3800, 4000) & year < 2008)
```

`dplyr`: A Grammar of Data Manipulation

Operations on rows: `slice()` picks rows based on location

```
penguins %>%
  slice(23:26)
```

```
> # A tibble: 4 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Biscoe           35.9          19.2               189
> 2 Adelie  Biscoe           38.2          18.1               185
> 3 Adelie  Biscoe           38.8          17.2               180
> 4 Adelie  Biscoe           35.3          18.9               187
> # ... with 3 more variables: body_mass_g <dbl>, sex <chr>,
> #   year <dbl>
```

Other `slice_*()` functions:

- `slice_head()` (`slice_tail()`) lets you pick the `n` first (last) observations in the data frame
- `slice_sample()` lets you pick a random sample of `n` observations (with or without replacement)
- `slice_min()` (`slice_max()`) lets you pick the `n` largest (smallest) observations in the data frame

`dplyr`: A Grammar of Data Manipulation

Operations on rows: `arrange()` changes the order of rows

Return the three observed penguins with the smallest body mass:

```
penguins %>%
  arrange(body_mass_g) %>%
  slice_head(n = 3)  #equivalent to using slice_min(order_by = body_mass_g, n = 3)
```

```
> # A tibble: 3 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Chinst~ Dream            46.9          16.6               192
> 2 Adelie  Biscoe           36.5          16.6               181
> 3 Adelie  Biscoe           36.4          17.1               184
> # ... with 3 more variables: body_mass_g <dbl>, sex <chr>,
> #   year <dbl>
```

Return the three observed penguins with the highest body mass:

```
penguins %>%
  arrange(desc(body_mass_g)) %>%
  slice_head(n = 3)  #equivalent to using slice_max(order_by = body_mass_g, n = 3)
```

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `select()` picks respectively drops certain columns

Select the first three columns (by index):

```
penguins %>%
  select(1:3)
```

Select the first three columns (by column name):

```
penguins %>%
  select(species, island, bill_length_mm)
```

Frequently used helper functions:

- `everything()`: select all columns
- `last_col()`: select the last column
- `starts_with()`: select columns which names start with a certain string
- `ends_with()`: select columns which names end with a certain string
- `contains()`: select columns which name contains a certain string
- `where()`: select columns for which a function evaluates to `TRUE`

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `select()` picks respectively drops certain columns

Which columns are returned by the following queries? Try it out at home :)

```
penguins %>%
  select(starts_with("s"))
```

```
penguins %>%
  select(contains("mm"))
```

```
penguins %>%
  select(ends_with("mm"))
```

```
penguins %>%
  select(-contains("mm"))
```

```
penguins %>%
  select(where(is.numeric)) %>%   #equivalent to select(where(~is.numeric(.)))
  select(where(~mean(., na.rm=T) > 1000))
```

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `rename()` changes the column names

Change the name of the column `body_mass_g` (`sex`) to `bm` (`gender`):

```
penguins %>%
  rename(bm = body_mass_g, gender = sex) %>%
  colnames()
```

```
> [1] "species"           "island"            "bill_length_mm"
> [4] "bill_depth_mm"     "flipper_length_mm" "bm"
> [7] "gender"            "year"
```

Convert the name of the columns that include the string "mm" to upper case:

```
penguins %>%
  rename_with(.fn = toupper, .cols = contains("mm")) %>%
  colnames()
```

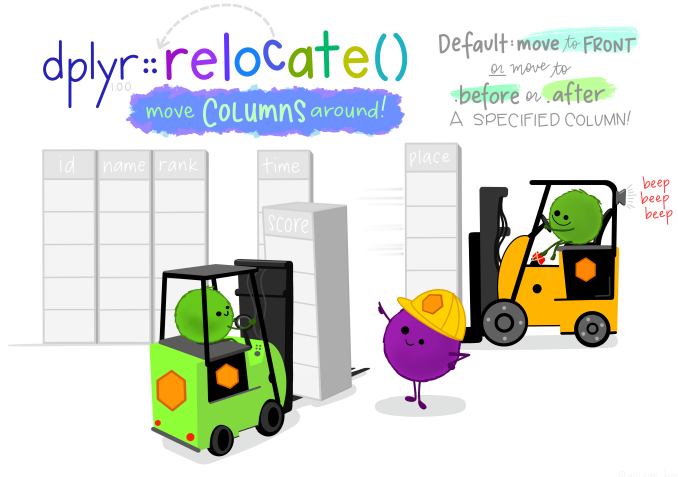
```
> [1] "species"           "island"            "BILL_LENGTH_MM"
> [4] "BILL_DEPTH_MM"     "FLIPPER_LENGTH_MM" "body_mass_g"
> [7] "sex"               "year"
```

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `relocate()` changes the order of columns

Change the order of columns in the `tibble` according to the following scheme:

1. place `species` after `body_mass_g`
2. place `sex` before `species`
3. place `island` after the last column




```
penguins %>%
  relocate(species, .after = body_mass_g) %>%
  relocate(sex, .before = species) %>%
  relocate(island, .after = last_col()) %>%
  colnames()
```

```
> [1] "bill_length_mm"    "bill_depth_mm"     "flipper_length_mm"
> [4] "body_mass_g"       "sex"               "species"
> [7] "year"              "island"
```

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `mutate()` changes the values of columns and/or creates new columns

Engineer a new `bm_kg` variable which is measured in kilo grams:

```
penguins %>%
  mutate(
    bm_kg = body_mass_g / 1000,
    .keep = "all",
    .after = body_mass_g) %>%
  slice_head(n = 3)
```

```
> # A tibble: 3 x 9
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>            <dbl>
> 1 Adelie  Torge~           39.1          18.7              181
> 2 Adelie  Torge~           39.5          17.4              186
> 3 Adelie  Torge~           40.3          18                195
> # ... with 4 more variables: body_mass_g <dbl>, bm_kg <dbl>,
> #   sex <chr>, year <dbl>
```

- Use the `.keep` argument to specify which columns to keep after manipulation.
- Use the `.before`/`.after`) arguments to specify the position of the new column.
- For overriding a given column simply use the same column name.
- For keeping only the new column use `dplyr::transmute()`.

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `mutate()` changes the values of columns and/or creates new columns

Engineer a *one-hot encoded* `sex` dummy:

```
penguins %>%
  mutate(
    sex = case_when(
      sex == "male" ~ 1,
      sex == "female" ~ 0),
    .keep = "all") %>%
  slice_head(n = 3)
```

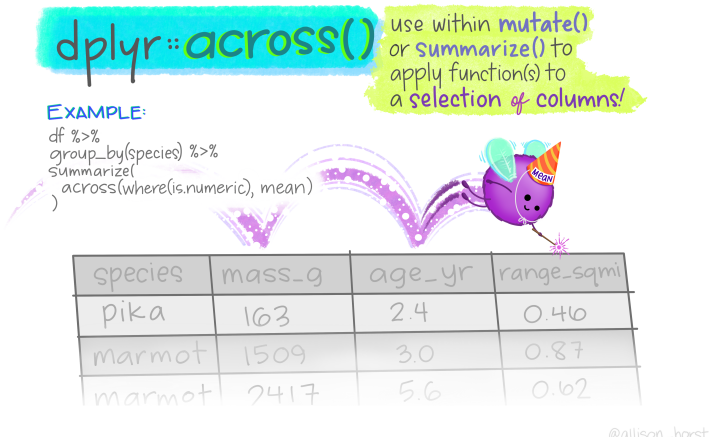

```
> # A tibble: 3 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Torge~           39.1          18.7               181
> 2 Adelie  Torge~           39.5          17.4               186
> 3 Adelie  Torge~           40.3          18                 195
> # ... with 3 more variables: body_mass_g <dbl>, sex <dbl>,
> #   year <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `mutate()` changes the values of columns and/or creates new columns

Engineer measurement variables that are measured in meters:

```
penguins %>%
  mutate(
    across(contains("mm"), ~ . / 1000),
    .keep = "all")
```



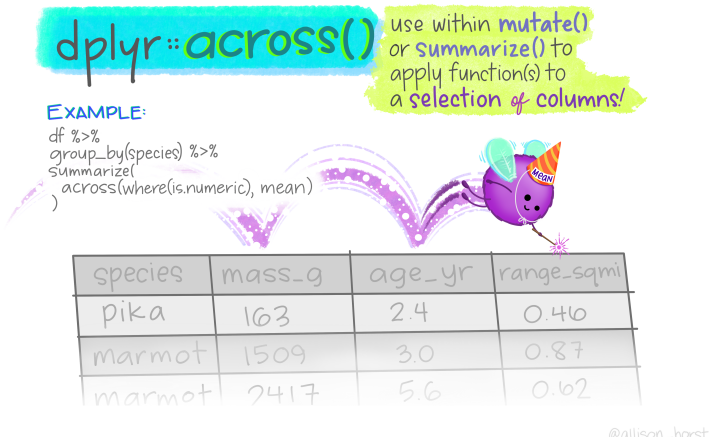

```
> # A tibble: 344 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>            <dbl>
> 1 Adelie  Torge~         0.0391        0.0187            0.181
> 2 Adelie  Torge~         0.0395        0.0174            0.186
> 3 Adelie  Torge~         0.0403        0.018             0.195
> # ... with 341 more rows, and 3 more variables: body_mass_g <dbl>,
> #   sex <chr>, year <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on columns: `mutate()` changes the values of columns and/or creates new columns

Define `species`, `island` and `sex` as a categorical variables (*factor*):

```
penguins %>%
  mutate(
    across(
      where(is.character), as.factor),
      .keep = "all")
```




```
> # A tibble: 344 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <fct>   <fct>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Torge~           39.1          18.7               181
> 2 Adelie  Torge~           39.5          17.4               186
> 3 Adelie  Torge~           40.3          18                 195
> # ... with 341 more rows, and 3 more variables: body_mass_g <dbl>,
> #   sex <fct>, year <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns

```
penguins %>%
  group_by(species)
```

Use `group_keys()`, `group_indices()` and `group_vars()` to access grouping keys, group indices per row and grouping variables.

```
> # A tibble: 344 x 8
> # Groups:   species [3]
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Torge~           39.1          18.7               181
> 2 Adelie  Torge~           39.5          17.4               186
> 3 Adelie  Torge~           40.3          18                 195
> 4 Adelie  Torge~           NA            NA                  NA
> 5 Adelie  Torge~           36.7          19.3               193
> 6 Adelie  Torge~           39.3          20.6               190
> 7 Adelie  Torge~           38.9          17.8               181
> 8 Adelie  Torge~           39.2          19.6               195
> # ... with 336 more rows, and 3 more variables: body_mass_g <dbl>,
> #   sex <chr>, year <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns

Under the hood `group_by` changes the representation of our `tibble` and transforms it into a grouped data frame (`grouped_df`). This allows us to operate on the three subgroups individually using `summarise()`.

Operations on grouped data: `summarise()` reduces a group into a single row

```
penguins %>%
  group_by(species) %>% #univariate
  summarise(count = n(), .groups = "drop")
```

```
> # A tibble: 3 x 2
>   species   count
>   <chr>     <int>
> 1 Adelie      152
> 2 Chinstrap    68
> 3 Gentoo      124
```

```
penguins %>%
  group_by(species, year) %>% #bivariate
  summarise(count = n(), .groups = "drop")
```

```
> # A tibble: 9 x 3
>   species    year count
>   <chr>     <dbl> <int>
> 1 Adelie     2007    50
> 2 Adelie     2008    50
> 3 Adelie     2009    52
> 4 Chinstrap  2007    26
> # ... with 5 more rows
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

```
penguins %>%
  group_by(species) %>%
  summarise(
    across(contains("mm"), ~mean(., na.rm = T), .names = "{.col}_avg"),
    .groups = "drop")
```

```
> # A tibble: 3 x 4
>   species   bill_length_mm_avg bill_depth_mm_a~ flipper_length_mm_a~
>   <chr>                  <dbl>            <dbl>                <dbl>
> 1 Adelie                  38.8             18.3                 190.
> 2 Chinstrap               48.8             18.4                 196.
> 3 Gentoo                  47.5             15.0                 217.
```

Using `group_by()`, followed by `summarise()` and `ungroup()` reflects the **split-apply-combine paradigm** in data analysis: Split the data into partitions, apply some function to the data and then merge the results.

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

Note: Instead of using `ungroup()` you may also set the `.groups` argument in `summarise()` equal to "drop".

But never forget to ungroup your data, otherwise you may run into errors later on in your analysis!

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

Stacked `group_by()`: If `.add = T` is omitted, the second `group_by()` overrides the first.

```
penguins %>%
  group_by(species) %>%
  group_by(year, .add = T)  #equivalent to group_by(species, year)
```

```
> # A tibble: 344 x 8
> # Groups:   species, year [9]
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Torge~           39.1          18.7               181
> 2 Adelie  Torge~           39.5          17.4               186
> 3 Adelie  Torge~           40.3          18                 195
> 4 Adelie  Torge~           NA            NA                  NA
> 5 Adelie  Torge~           36.7          19.3               193
> 6 Adelie  Torge~           39.3          20.6               190
> # ... with 338 more rows, and 3 more variables: body_mass_g <dbl>,
> #   sex <chr>, year <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

Apply multiple functions: Provide a list of `purrr`-style functions to `across()`.

```
penguins %>%
  group_by(species) %>%
  summarise(
    across(
      contains("mm"),
      list(avg = ~mean(., na.rm = T), sd = ~sd(., na.rm = T)),
      .names = "{.col}_{.fn}"),
    .groups = "drop")
```

```
> # A tibble: 3 x 7
>   species bill_length_mm_~ bill_length_mm_~ bill_depth_mm_a~
>   <chr>              <dbl>            <dbl>            <dbl>
> 1 Adelie              38.8             2.66             18.3
> 2 Chinst~             48.8             3.34             18.4
> 3 Gentoo              47.5             3.08             15.0
> # ... with 3 more variables: bill_depth_mm_sd <dbl>,
> #   flipper_length_mm_avg <dbl>, flipper_length_mm_sd <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

Changed behavior of `mutate()`: Summary functions, e.g., `mean()` or `sd()` now operate on partitions of the data instead of on the whole data frame.

```
penguins %>%
  group_by(species) %>%
  mutate(stand_bm = (body_mass_g - mean(body_mass_g, na.rm = TRUE)) /
           sd(body_mass_g, na.rm = TRUE))
```

```
> # A tibble: 344 x 9
> # Groups:   species [3]
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>            <dbl>
> 1 Adelie  Torge~           39.1          18.7              181
> 2 Adelie  Torge~           39.5          17.4              186
> 3 Adelie  Torge~           40.3          18                195
> 4 Adelie  Torge~           NA            NA                 NA
> # ... with 340 more rows, and 4 more variables: body_mass_g <dbl>,
> #   sex <chr>, year <dbl>, stand_bm <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

`group_by()` a transformed column: Provide a `mutate()`-like expression in your `group_by()` statement.

```
bm_breaks <- mean(penguins$body_mass_g, na.rm = T) -
  (-3:3) * sd(penguins$body_mass_g, na.rm = T)

penguins %>%
  group_by(species, bm_cat = cut(body_mass_g, breaks = bm_breaks)) %>%
  summarise(count = n(), .groups = "drop")
```

```
> # A tibble: 12 x 3
>   species   bm_cat            count
>   <chr>     <fct>             <int>
> 1 Adelie    (2.6e+03,3.4e+03]    39
> 2 Adelie    (3.4e+03,4.2e+03]    87
> 3 Adelie    (4.2e+03,5e+03]      25
> 4 Adelie    <NA>                  1
> 5 Chinstrap (2.6e+03,3.4e+03]    11
> # ... with 7 more rows
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

Changed behavior of `filter()`: Filters can now operate on partitions of the data instead of on the whole data frame.

```
penguins %>%
  group_by(species, island) %>%
  filter(flipper_length_mm == max(flipper_length_mm, na.rm = T))
```

```
> # A tibble: 5 x 8
> # Groups:   species, island [5]
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Adelie  Dream            40.8          18.9               208
> 2 Adelie  Biscoe           41            20                 203
> 3 Adelie  Torge~           44.1          18                 210
> 4 Gentoo  Biscoe           54.3          15.7               231
> 5 Chinst~ Dream            49            19.6               212
> # ... with 3 more variables: body_mass_g <dbl>, sex <chr>,
> #   year <dbl>
```

`dplyr`: A Grammar of Data Manipulation

Operations on grouped data: `group_by()` groups rows based on a set of columns and `summarise()` reduces a group into a single row

Nesting of grouped data: Usually, you will find it more intuitive to use `group_by()` followed by `nest()` to produce a nested data frame compared to the example on slide 27.

```
penguins %>%
  group_by(species, year) %>%
  nest
```

```
> # A tibble: 9 x 3
> # Groups:   species, year [9]
>   species  year data
>   <chr>   <dbl> <list>
> 1 Adelie   2007 <tibble [50 x 6]>
> 2 Adelie   2008 <tibble [50 x 6]>
> 3 Adelie   2009 <tibble [52 x 6]>
> 4 Gentoo   2007 <tibble [34 x 6]>
> 5 Gentoo   2008 <tibble [46 x 6]>
> # ... with 4 more rows
```

Note: Find more information about `group_by()` by running `vignette("grouping")`.

`dplyr`: A Grammar of Data Manipulation

Some more selected `dplyr` functions:

`dplyr::distinct()`: selects only unique rows

```
penguins %>%
  distinct(species, island)
```

`dplyr::pull()`: extracts single columns as vectors

```
penguins %>%
  pull(year) #equivalent to penguins$year
```

`dplyr::if_else()`: applies a vectorized if-else-statement

```
penguins %>%
  select(species, island, body_mass_g) %>%
  mutate(penguin_size = if_else(body_mass_g < 3500, "tiny penguin", "big penguin"))
```

`dplyr`: A Grammar of Data Manipulation

Some more selected `dplyr` functions:

`dplyr::lag()` and `dplyr::lead()`: shifts column values by an offset `n`

```
penguins %>%
  select(species, body_mass_g) %>%
  mutate(
    lagged_bm = lag(body_mass_g, n = 1),
    lead_bm = lead(body_mass_g, n = 2))
```

`dplyr::left_join()`, `dplyr::right_join()`, `dplyr::inner_join()` and `dplyr::full_join()`: merge different data frames by matching rows based on keys (similar to joins performed in SQL).

Note: Find more information about `dplyr` by running vignette("dplyr") and consulting the official cheat sheet (may be not reflect updates included in `dplyr v1.0.0`).

purrr:

Functional Programming Tools

`purrr`: Functional Programming Tools

`purrr` facilitates *functional programming* (FP) with data frame objects (e.g., `tibbles`) in `R`. Whenever you would normally refer to a `for`-loop for solving an iterative problem, the family of `map_*()` functions allows you to rephrase your problem as a data manipulation pipeline.

Three types of `map_*()` function:

- `map(.x, .f, ...)` takes the input `.x` and applies `.f` to each element in `.x`.
- `group_map(.data, .f, ...)` takes a grouped `tibble` and applies `.f` to each subgroup.
- `map2(.x, .y, .f, ...)` takes the inputs `.x` and `.y` and applies `.f` to `.x` and `.y` in parallel.
- `pmap(.l, .f, ...)` takes a list `.l` of inputs and applies `.f` to each element in `.l` in parallel.

By default `map()` returns a list. If you want to be more explicit about the output you may refer to

- `map_lgl()` to receive an output type logical,
- `map_chr()` to receive an output type character,
- `map_int()` to receive an output type integer,
- `map_dbl()` to receive an output type double,
- `map_df()` to receive a data frame output.

The input `.x` to any `map()_*` function can be either a vector, list or data frame.

- **Vector:** Iteration over vector entries
- **List:** Iteration over list elements
- **Data frame:** Iteration over columns

`purrr`: Functional Programming Tools

Use Case: Applying the z-normalization to multiple variables

First, write a *named function* for performing z-normalization that takes a vector `.x` as input.

```
z_transform <- function(.x) {
  mean <- mean(.x); sd <- sd(.x); return( (.x - mean) / sd )
}
```

Second, draw samples from the `penguin` data set and store them as double vectors in a list.

```
samples <- list(
  sample1 = slice_sample(penguins, n = 10)$bill_length_mm,
  sample2 = slice_sample(penguins, n = 10)$bill_depth_mm,
  sample3 = slice_sample(penguins, n = 10)$flipper_length_mm)
```

```
> $sample1
>  [1] 49.0 42.0 47.5 37.8 38.6 41.3 37.6 40.6 45.6 51.1
>
> $sample2
>  [1] 15.8 17.1 16.5 18.6 18.8 17.0 13.9 13.7 18.4 16.2
>
> $sample3
>  [1] 198 201 186 199 184 215 193 220 190 188
```

`purrr`: Functional Programming Tools

Use Case: Applying the z-normalization to multiple variables

Third, perform the z-normalization using a `for`-loop.

```
for (s in samples) {
  print(z_transform(s))
}
```

Or simply perform the z-normalization using `map()`.

```
map(.x = samples, .f = ~z_transform(.x)) #equivalent to map(samples, z_transform)
```

```
> $sample1
>  [1]  1.2087184 -0.2277890  0.9008954 -1.0896935 -0.9255212
>  [6] -0.3714398 -1.1307366 -0.5150905  0.5109862  1.6396707
>
> $sample2
>  [1] -0.4456688  0.2785430 -0.0557086  1.1141720  1.2255892
>  [6]  0.2228344 -1.5041322 -1.6155494  1.0027548 -0.2228344
>
> $sample3
>  [1]  0.04976251  0.29857503 -0.94548760  0.13270001 -1.11136262
>  [6]  1.45970016 -0.36492504  1.87438770 -0.61373757 -0.77961258
```

`purrr`: Functional Programming Tools

Use Case: Applying the z-normalization to multiple variables

Or perform the z-normalization using `map()` but use an *anonymous function*.

```
map(
  .x = samples,
  .f = function(.x) {
    (.x - mean(.x, na.rm = T)) / sd(.x, na.rm = T)
  })
```

Or perform the z-normalization using `map()` but use an *`purrr`-style function*.

```
map(
  .x = samples,
  .f = ~(.x - mean(.x, na.rm = T)) / sd(.x, na.rm = T))
```

`purrr`: Functional Programming Tools

`purrr`: Functional Programming Tools

Check the data types of my columns:

```
penguins %>%
  map_df(class)
```

```
> # A tibble: 1 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>  <chr>          <chr>         <chr>
> 1 charac~ chara~ numeric        numeric       numeric
> # ... with 3 more variables: body_mass_g <chr>, sex <chr>,
> #   year <chr>
```

Check the number of missing values per column:

```
penguins %>%
  map_df(~sum(is.na(.)))
```

```
> # A tibble: 1 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>     <int>  <int>          <int>         <int>             <int>
> 1       0      0              2             2                 2
> # ... with 3 more variables: body_mass_g <int>, sex <int>,
> #   year <int>
```

`purrr`: Functional Programming Tools

Check the number of distinct values per column:

```
penguins %>%
  map_df(n_distinct)
```

```
> # A tibble: 1 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>     <int>  <int>          <int>         <int>             <int>
> 1       3      3            165            81                56
> # ... with 3 more variables: body_mass_g <int>, sex <int>,
> #   year <int>
```

`purrr`: Functional Programming Tools

Check the highest value in each subset of the data (e.g., largest `flipper_length_mm` per `sex`):

```
penguins %>%
  drop_na %>%
  group_by(sex) %>%
  group_map(~slice_max(., flipper_length_mm, n = 1), .keep = T)
```

```
> [[1]]
> # A tibble: 1 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Gentoo  Biscoe           46.9          14.6               222
> # ... with 3 more variables: body_mass_g <dbl>, sex <chr>,
> #   year <dbl>
>
> [[2]]
> # A tibble: 1 x 8
>   species island bill_length_mm bill_depth_mm flipper_length_~
>   <chr>   <chr>           <dbl>         <dbl>             <dbl>
> 1 Gentoo  Biscoe           54.3          15.7               231
> # ... with 3 more variables: body_mass_g <dbl>, sex <chr>,
> #   year <dbl>
```

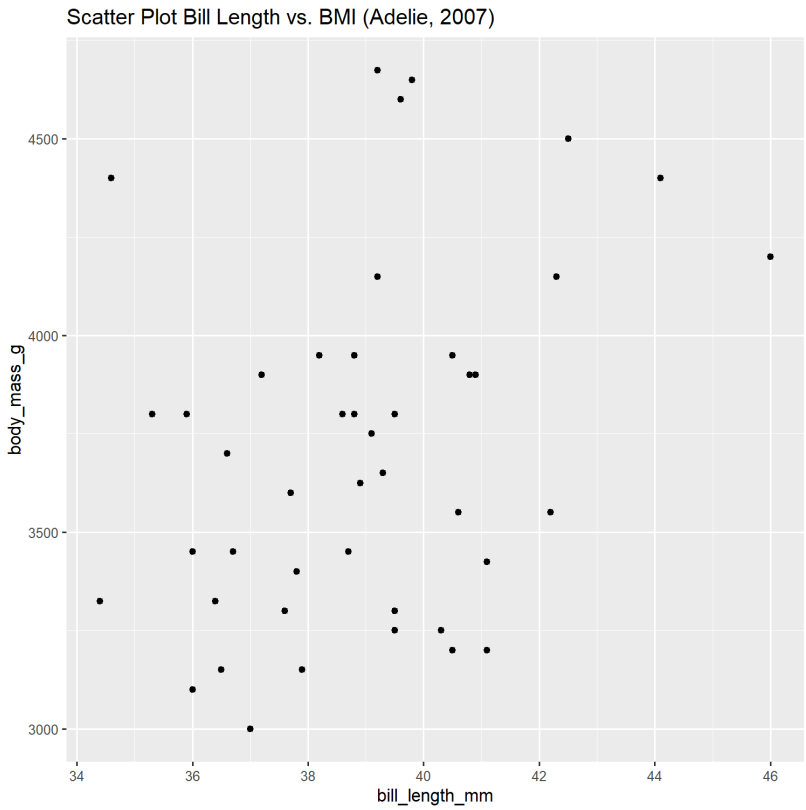
`purrr`: Functional Programming Tools

`map()` also comes in handy, if you like to produce a series of identical plots, each depicting a separate subset of the underlying data:

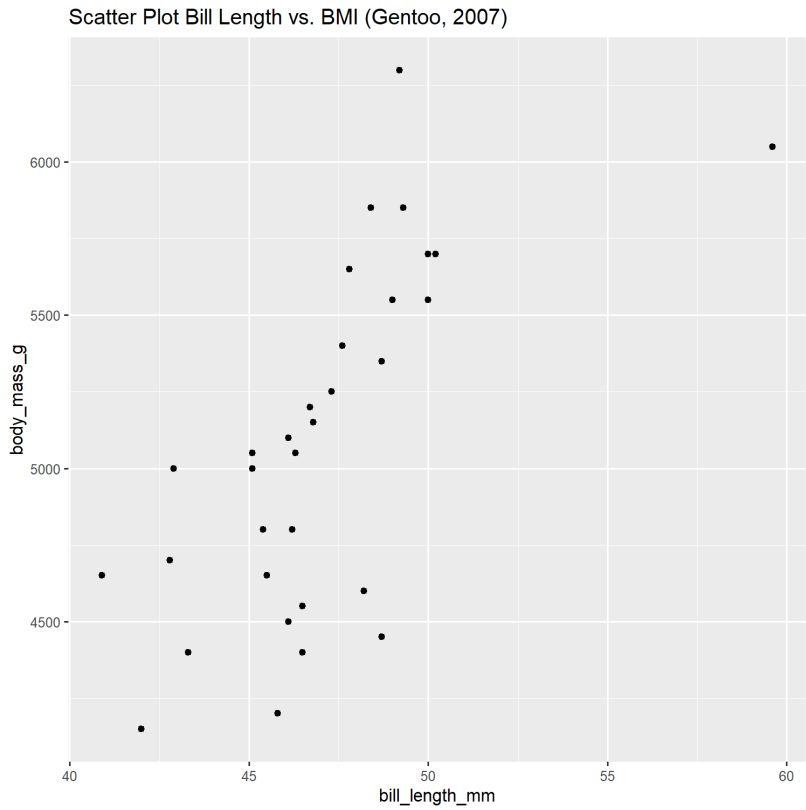
```
species <- penguins %>% distinct(species, year) %>% pull(species) #.x argument for map()
years <- penguins %>% distinct(species, year) %>% pull(year)      #.y argument for map()

penguin_plots <- map2(
  .x = species,
  .y = years,
  .f = ~{
    penguins %>%
      drop_na %>%
      filter(species == .x, year == .y) %>%
      ggplot() +
        geom_point(aes(x = bill_length_mm, y = body_mass_g)) +
        labs(title = glue::glue("Scatter Plot Bill Length vs. BMI ({.x}, {.y})"))
    })
```

purrr: Functional Programming Tools


```
penguin_plots[[1]]
```




```
penguin_plots[[4]]
```



`purrr`: Functional Programming Tools

Finally, `map()` is really powerful in the context of modeling. In the following we fit a linear regression model for each `species-island` subset.

First, we create a nested data frame that contains a `tibble` of data for each `species-island` combination.

```
nested_penguins <- penguins %>%
  drop_na %>%
  group_by(species, island) %>%
  nest
```

```
> # A tibble: 5 x 3
> # Groups:   species, island [5]
>   species   island    data
>   <chr>     <chr>     <list>
> 1 Adelie    Torgersen <tibble [47 x 6]>
> 2 Adelie    Biscoe    <tibble [44 x 6]>
> 3 Adelie    Dream     <tibble [55 x 6]>
> 4 Gentoo    Biscoe    <tibble [119 x 6]>
> 5 Chinstrap Dream     <tibble [68 x 6]>
```

Note: For accessing elements in a nested `tibble` you may use the `pluck()`. For example, for accessing the first `tibble` in the column `data`, you may run `nested_penguins %>% pluck("data", 1)`.

`purrr`: Functional Programming Tools

Second, we fit a linear model to each data subset. In our model `body_mass_g` is regressed (`~`) on all other variables (denoted by a dot in the `lm()` formula).

```
nested_penguins <- nested_penguins %>%
  mutate(lin_reg = map(.x = data, .f = ~lm(body_mass_g ~ ., data = .x)))
```

```
> # A tibble: 5 x 4
> # Groups:   species, island [5]
>   species   island    data               lin_reg
>   <chr>     <chr>     <list>             <list>
> 1 Adelie    Torgersen <tibble [47 x 6]>  <lm>
> 2 Adelie    Biscoe    <tibble [44 x 6]>  <lm>
> 3 Adelie    Dream     <tibble [55 x 6]>  <lm>
> 4 Gentoo    Biscoe    <tibble [119 x 6]> <lm>
> 5 Chinstrap Dream     <tibble [68 x 6]>  <lm>
```

`purrr`: Functional Programming Tools

Third, for each linear model, we generate a model summary using `summary()` and extract the model coefficients as a `tibble`. Finally, we use `unnest()` to receive a tidy data frame.

```
nested_penguins %>%
  mutate(coefs = map(lin_reg, ~summary(.x) %>% .$coefficients %>% as_tibble)) %>%
  select(-data, -lin_reg) %>%
  unnest(coefs)
```

```
> # A tibble: 30 x 6
> # Groups:   species, island [5]
>   species island    Estimate `Std. Error` `t value` `Pr(>|t|)`
>   <chr>   <chr>        <dbl>        <dbl>     <dbl>      <dbl>
> 1 Adelie  Torgersen 449264.      130401.       3.45     0.00133
> 2 Adelie  Torgersen      4.20        17.3      0.243    0.809
> 3 Adelie  Torgersen    -62.0         54.6     -1.14     0.263
> 4 Adelie  Torgersen     15.5          8.74     1.77     0.0838
> # ... with 26 more rows
```

Note: Here we eventually dropped the `lin_reg` and `data` columns, otherwise we would have carried around a lot of redundant data which may exceed our memory storage capacity very quickly.

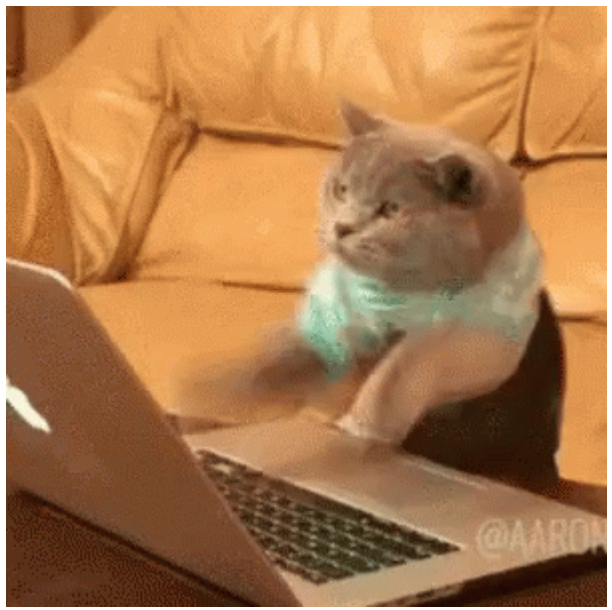
`purrr`: Functional Programming Tools

How you may probably feel right now

`purrr`: Functional Programming Tools

How you may probably feel right now

How you do after mastering the intricacies of FP



Note: For a great tutorial that helps you master the notion of functional programming with `R` see [5].

`purrr`: Functional Programming Tools

Finally, `purrr` also provides convenient wrapper functions for **error handling**. These come in handy if your are iterating over a very large data set and your program would simply stop if an error occurs and you would loose the whole progress.

For example, at some point you might want train a separate prediction model (`lm`) for each unique value of `species` (Adelie, Gentoo, Chinstrap). Unfortunately, the following code is throwing an error...

```
grouped_penguins <- penguins %>%
  mutate(across(c(sex, island), as.factor)) %>%
  group_by(species)
```

```
grouped_penguins %>%
  group_map(.f = ~lm(flipper_length_mm ~ bill_length_mm + island, data = .))
```

```
> Error in `contrasts<-`(`*tmp*`, value = contr.funs[1 + isOF[nn]]) :
> contrasts can be applied only to factors with 2 or more levels
```

🤔 **Which group is eventually responsible to the error?**

`purrr`: Functional Programming Tools

`purrr::possibly()`: returns a list containing the function's result respectively a user-defined value (`otherwise`) if an error occurs

```
possibly_lm <- possibly(.f = lm, otherwise = "Error message")

grouped_penguins %>%
  group_map(.f = ~possibly_lm(flipper_length_mm ~ bill_length_mm + island, data = .))
```

```
> [[1]]
>
> Call:
> .f(formula = ..1, data = ..2)
>
> Coefficients:
>     (Intercept)   bill_length_mm      islandDream  islandTorgersen
>        157.5591           0.8014           1.3159           2.4199
>
>
> [[2]]
> [1] "Error message"
>
> [[3]]
> [1] "Error message"
```

Note: Use `purrr::discard(. == "Error message")` (`purrr::keep()`) at the end of the pipeline to drop (keep) function calls that yielded an error. These work like `dplyr::select()` and `dplyr::filter()` in the context of `tibbles`.

`purrr`: Functional Programming Tools

`purrr::safely()`: returns a named list containing the function's result (or `otherwise` if an error occurs) as well as an error object that captures the error message

```
safely_lm <- safely(.f = lm, otherwise = NULL)

grouped_penguins %>%
  group_map(.f = ~safely_lm(flipper_length_mm ~ bill_length_mm + island, data = .))
```

- Use `purrr::map(., "result")` at the end of the pipeline to access the results of each function call stored in the list.
- Use `purrr::map(., "error")` at the end of the pipeline to access the errors of each function call stored in the list.

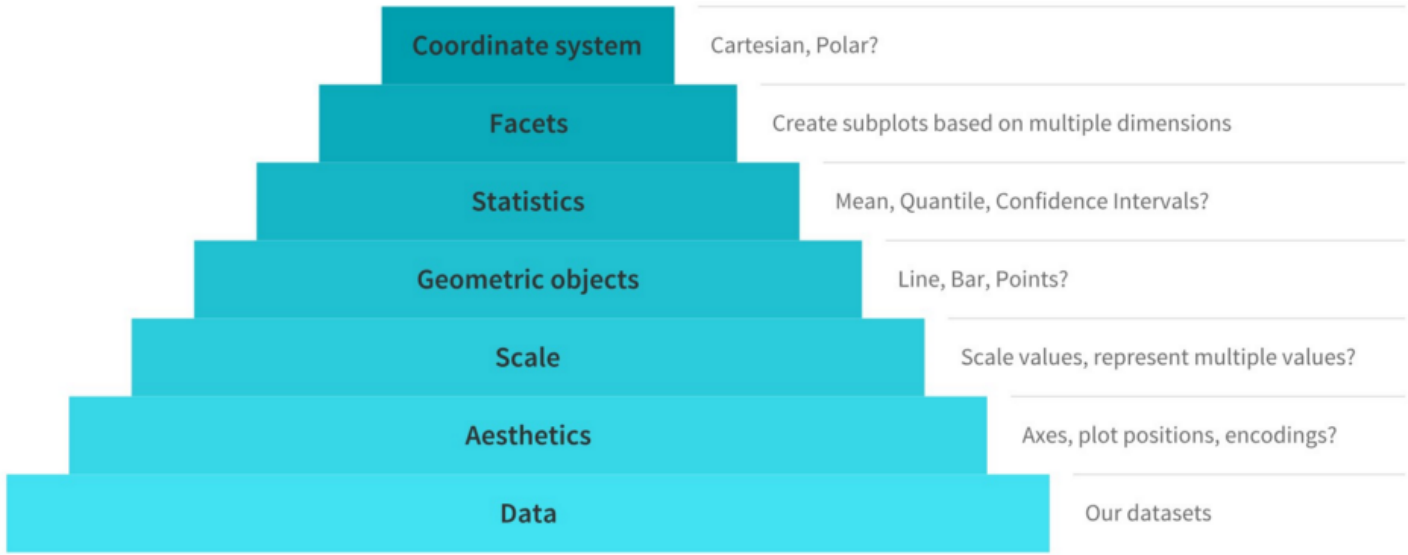
*Note: Similarly, use **`purrr::quietly()`** to return a named list containing not only the function's results and error but also other kinds of output, such as warnings or messages.*

`ggplot2`:

Create Elegant Data Visualisations Using the Grammar of Graphics

`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

`ggplot2` is Hadley Wickham's reimplementation [6] of the 2005 published *The Grammar of Graphics* by Leland Wilkinson [7]. It provides a large amount of functions for generating high-quality graphs in layer-based fashion and has even sparked a whole ecosystem of 'gg'-style visualization packages (*ggverse*).



Source: [8]

ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Data: The data set (usually a `tibble`) from which to select the variables that are about to be plotted. It is specified by the first argument in `ggplot()` and thus predestined to be piped into our plot pipeline.

Univariate example:

```
penguins %>%
  ggplot(data = .) #equivalent to ggplot()
```

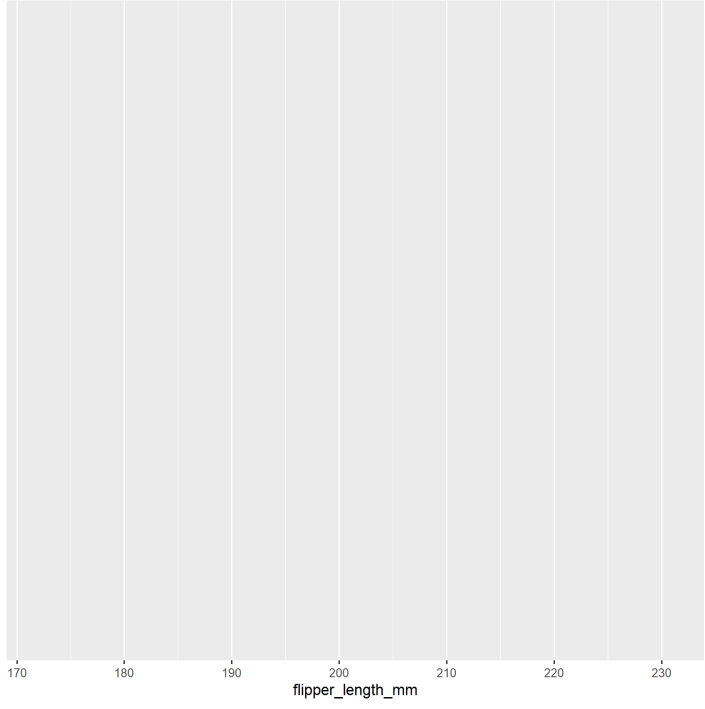
Note: Again, a good go-to-guide when crafting visualisations with `ggplot2` is the official cheat sheet.

`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Aesthetics: Mappings that describe how variables in my data are mapped to aesthetic attributes of the plot, such as axes, shapes, sizes or colors.

Univariate example:

```
penguins %>%
  ggplot(
    aes(x = flipper_length_mm))
```

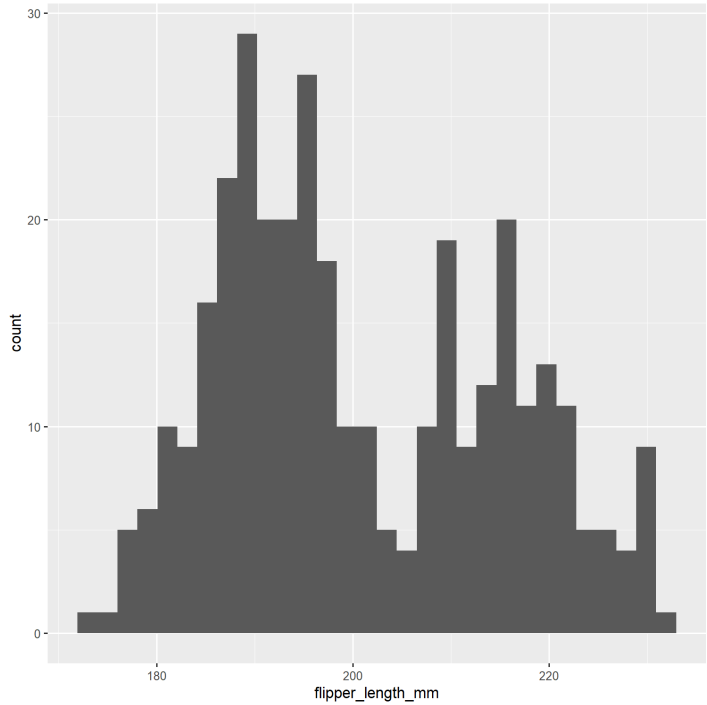


`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Geoms: Geometric objects that determine your overall plot type, e.g., bar, lines, points, boxplots. They specify the graphical representation of your data.

Univariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm)) +
    geom_histogram(na.rm = TRUE)
```



`ggplot2` comes with very decent *default* settings. Each `geom_*()` comes with its own options for customizing the geom, e.g.,

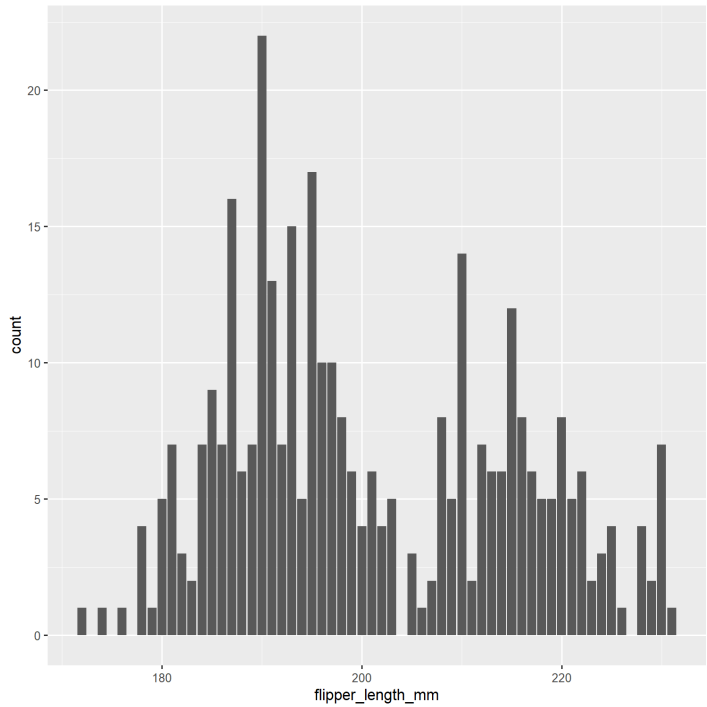
- change number of bins with `bins` argument,
- change number binwidth with `binwidth` argument.

`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Geoms: Geometric objects that determine your overall plot type, e.g., bar, lines, points, boxplots. They specify the graphical representation of your data.

Univariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm)) +
    geom_bar(na.rm = TRUE)
```

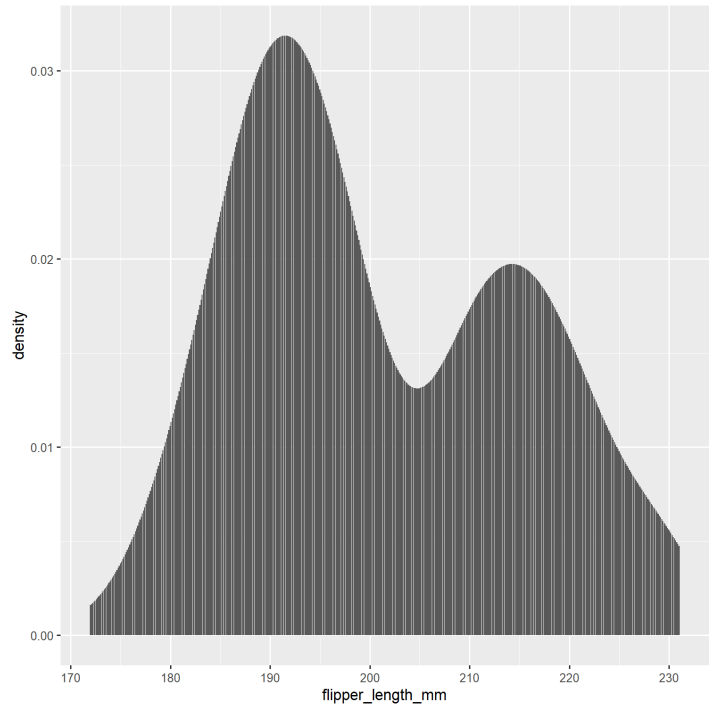


ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

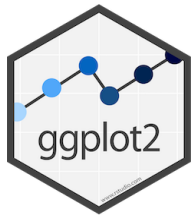
Stats: Statistical transformations provide a summary of the data. They can be used to transform a given variable without changing the plot type (i.e. geom).

Univariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm)) +
    geom_bar(
      stat = "density",
      na.rm = TRUE)
```



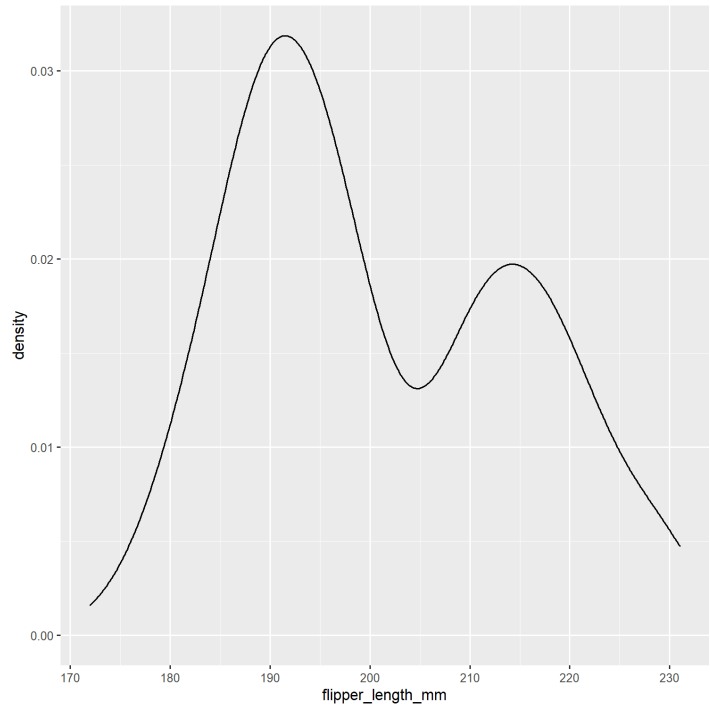
ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics



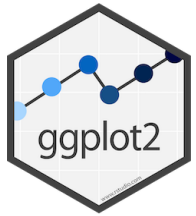
Stats: Statistical transformations provide a summary of the data. They can be used to transform a given variable without changing the plot type (i.e. geom).

Univariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm)) +
    geom_density(na.rm = TRUE)
```



`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

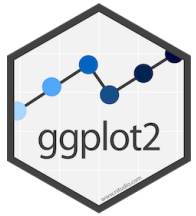


Data: The data set (usually a `tibble`) from which to select the variables that are about to be plotted. It is specified by the first argument in `ggplot()` and thus predestined to be piped into our plot pipeline.

Bivariate example:

```
penguins %>%
  ggplot()
```

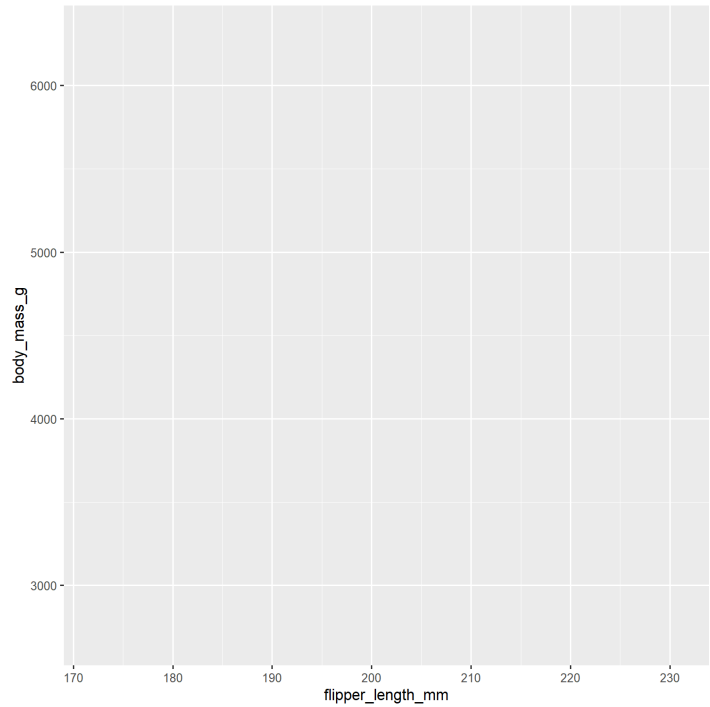
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



Aesthetics: Mappings that describe how variables in my data are mapped to aesthetic attributes of the plot, such as axes, shapes, sizes or colors.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g))
```

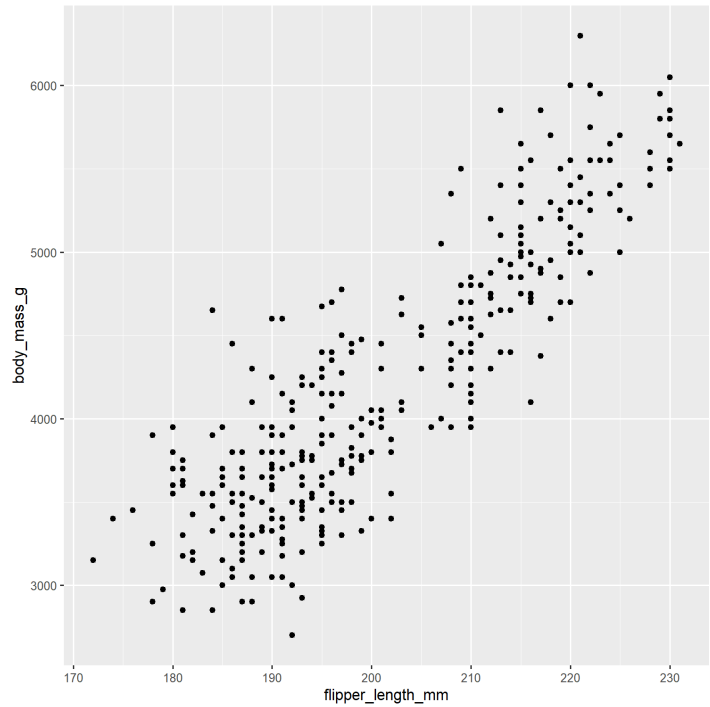


ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

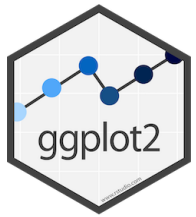
Geoms: Geometric objects that determine your overall plot type, e.g., bar, lines, points, boxplots. They specify the graphical representation of your data.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(na.rm = TRUE)
```



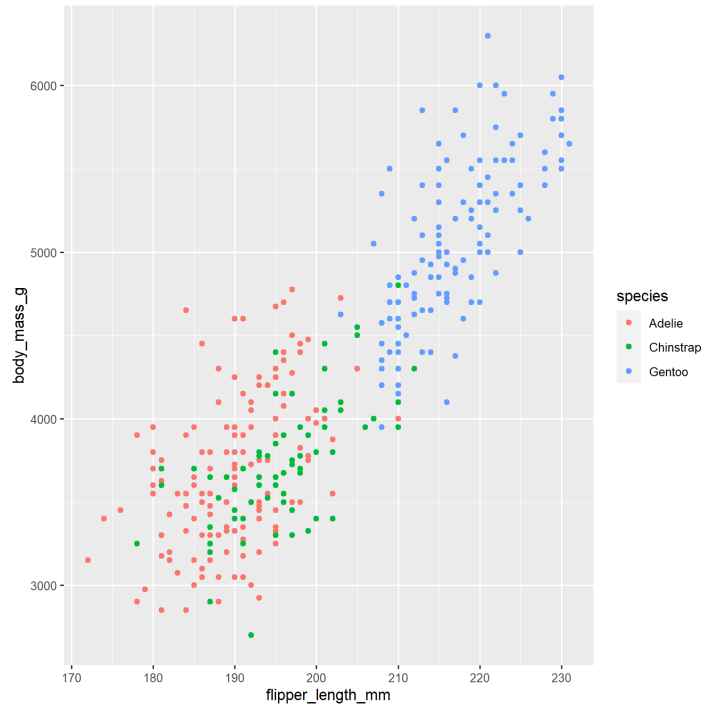
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



There are multiple ways of changing the color, shape or size aesthetics. Remember that using the `aes()` argument **maps** variable values to your aesthetic. The behavior differs for discrete vs. continuous variables.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(
      aes(color = species),
      na.rm = TRUE)
```

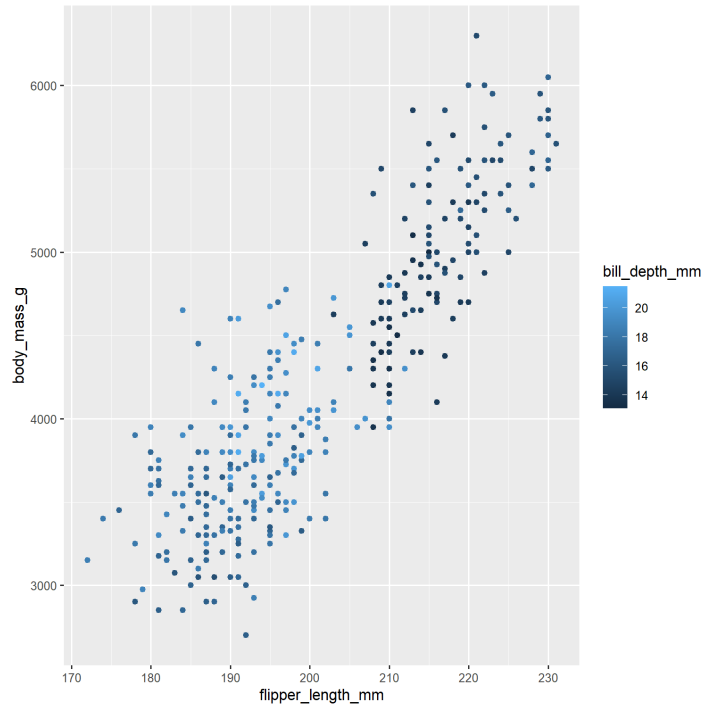


`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

There are multiple ways of changing the color, shape or size aesthetics. Remember that using the `aes()` argument **maps** variable values to your aesthetic. The behavior differs for discrete vs. continuous variables.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(
      aes(color = bill_depth_mm),
      na.rm = TRUE)
```



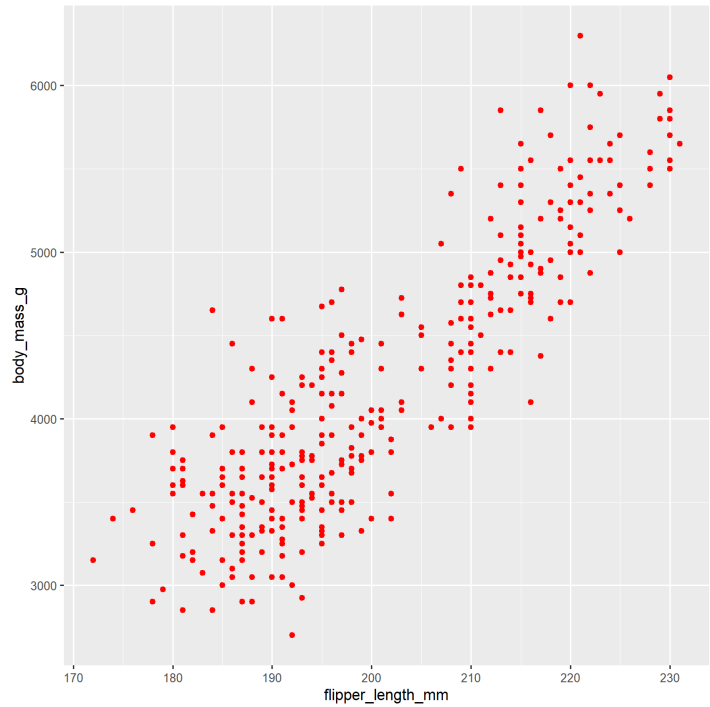
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

By specifying the `color` argument outside of the `aes()` argument, we **set** the color without considering the values of any other variable.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(
      color = "red",
      na.rm = TRUE)
```

For truly customized coloring plots you may refer to HTML color codes (also called *hex codes*, e.g., `#ff0000` for red) instead of specifying colors by name.

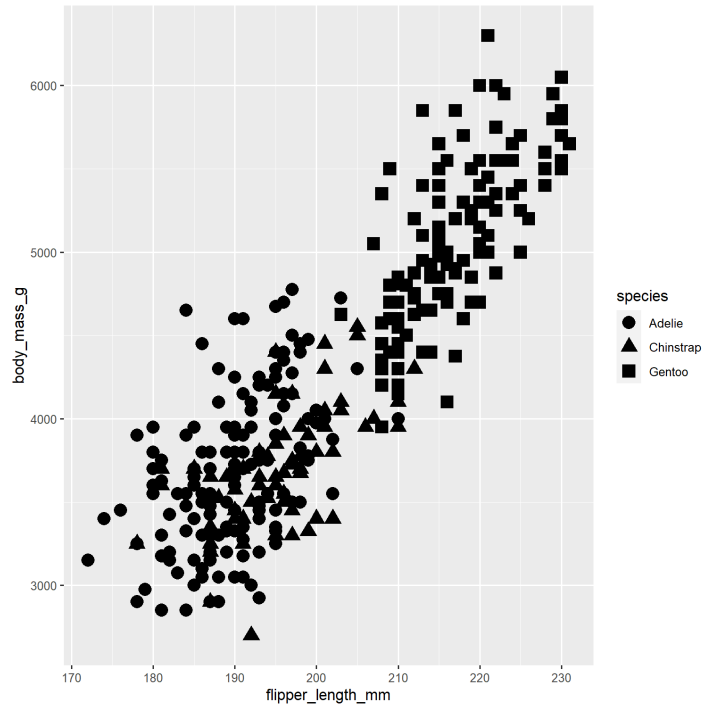


`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

We can do the same for changing the `shape` and `size` of our data points. Either by mapping them to the values of another variable or by setting them manually outside of the `aes()` argument.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(
      aes(shape = species),
      size = 4,
      na.rm = TRUE)
```



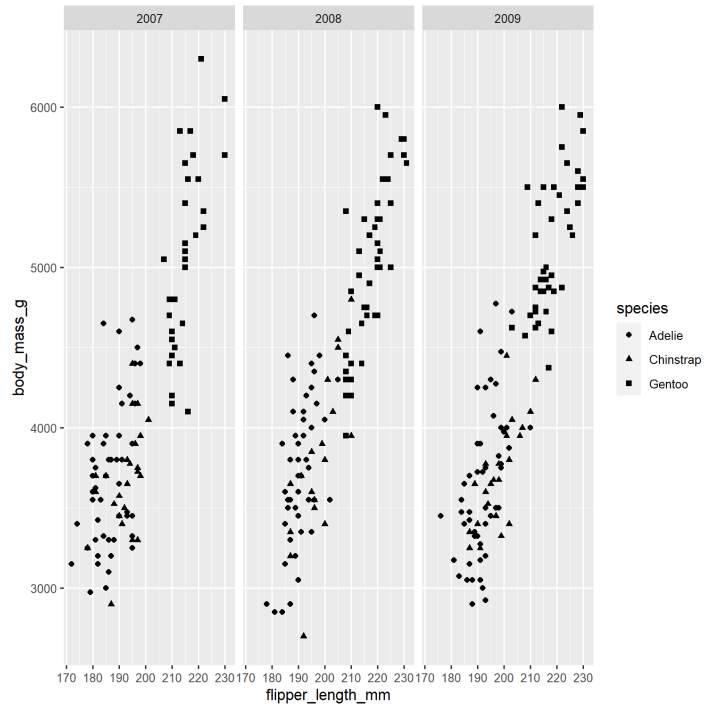
`ggplot2` provides 24 available shapes for customizing your plot. When specifying the shape outside of `aes()` you may refer to the shape's index (see shape overview).

ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

Facets: Facets allow you to split your plot into multiple subplots based on the levels of one or more variables not yet part of your plot.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(aes(shape = species),
               na.rm = TRUE) +
    facet_wrap(~year)
```

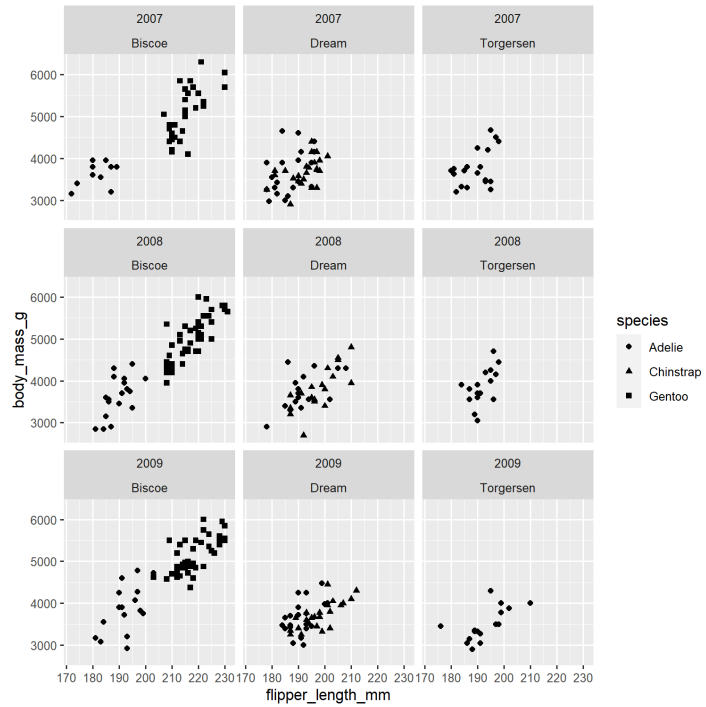


ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

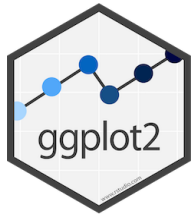
Facets: Facets allow you to split your plot into multiple subplots based on the levels of one or more variables not yet part of your plot.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(aes(shape = species),
               na.rm = TRUE) +
    facet_wrap(~year + island)
```



`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

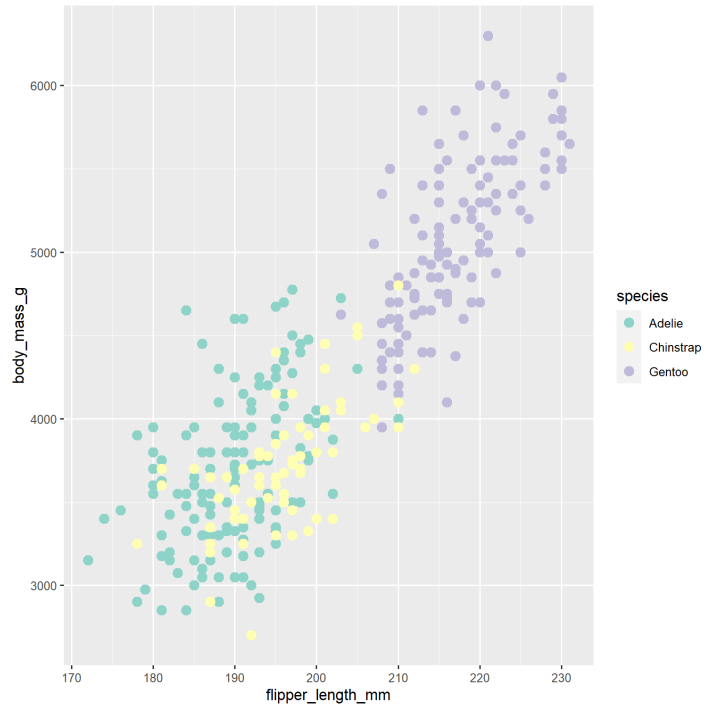


Scales: Scales control the aesthetic mappings by overriding the *default* settings. For example, they allow to refine the presentation of x- and y-axis, labels or color palettes.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(aes(color = species), size = 3,
               na.rm = TRUE) +
    scale_colour_brewer(palette = "Set3")
```

The family of `scale_colour_*()` functions enables you to adjust the values of your `color` aesthetic ex post (e.g., `scale_colour_brewer()` to select a palette from the famous ColorBrewer project).



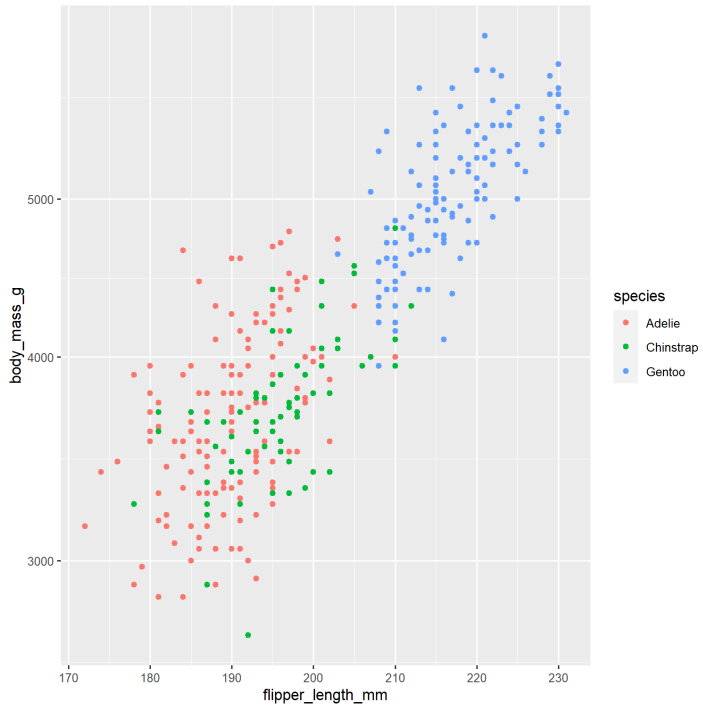
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Scales: Scales control the aesthetic mappings by overriding the *default* settings. For example, they allow to refine the presentation of x- and y-axis, labels or color palettes.

Bivariate example:

```
penguins %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(aes(color = species),
               na.rm = TRUE) +
    scale_y_log10()
```

Or use the `scale_*_log10()` functions to improve the readability of your plot in the presence of high-variance variables.



`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Other examples: Boxplots for numeric variables

```
penguins_long <- penguins %>%
  pivot_longer(
    cols = contains("mm"),
    names_to = "var",
    values_to = "val") %>%
  drop_na
```

- Use `dplyr::pivot_longer()` to bring data frame into *long* format.
- Take care of missing values using `dplyr::drop_na()` to avoid error messages.

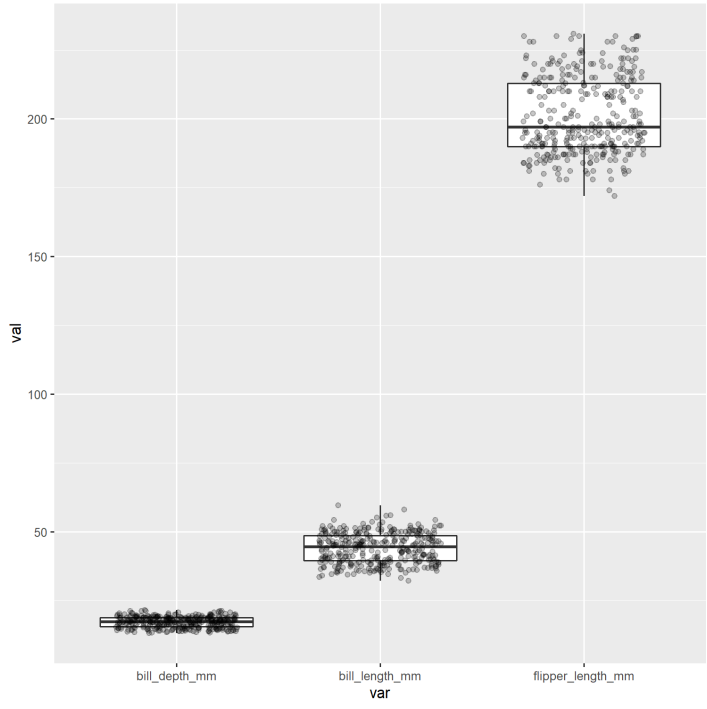
```
> # A tibble: 999 x 7
>   species island    body_mass_g sex     year var               val
>   <chr>   <chr>           <dbl> <chr>  <dbl> <chr>             <dbl>
> 1 Adelie  Torgersen        3750 male    2007 bill_length_mm     39.1
> 2 Adelie  Torgersen        3750 male    2007 bill_depth_mm      18.7
> 3 Adelie  Torgersen        3750 male    2007 flipper_length_mm 181
> 4 Adelie  Torgersen        3800 female  2007 bill_length_mm     39.5
> # ... with 995 more rows
```

`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Other examples: Boxplots for numeric variables

```
penguins_long %>%
  ggplot(aes(x = var, y = val)) +
    geom_boxplot(na.rm = TRUE) +
    geom_jitter(alpha = 0.22, width = 0.3)
```

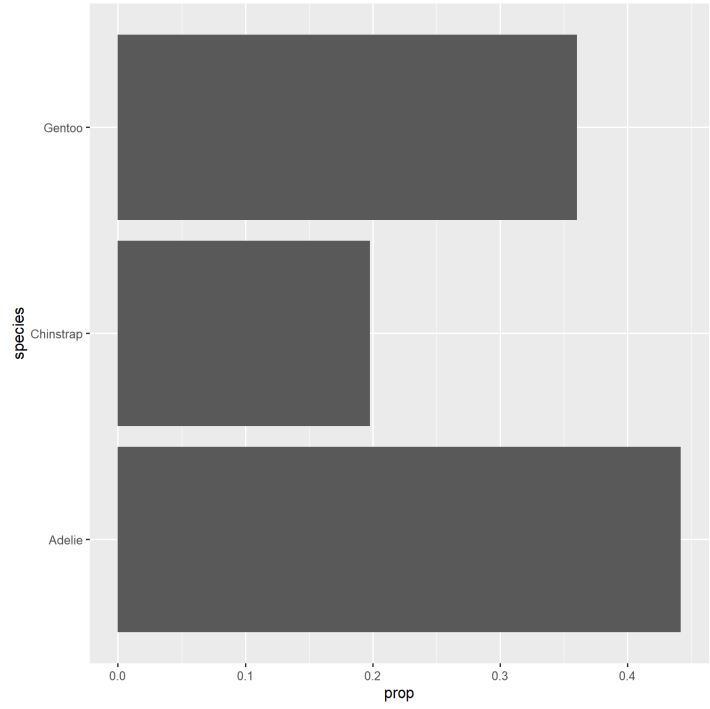
- Use `geom_jitter()` to induce some random noise to the data points to prevent overlapping.
- Control transparency of the respective plot element via the `alpha` aesthetic.



`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

Other examples: Ordered bar chart

```
plot <- penguins %>%
  dplyr::count(species) %>%
  dplyr::mutate(prop = n / sum(n)) %>%
  ggplot()

plot +
  geom_col(aes(x = prop, y = species))
```



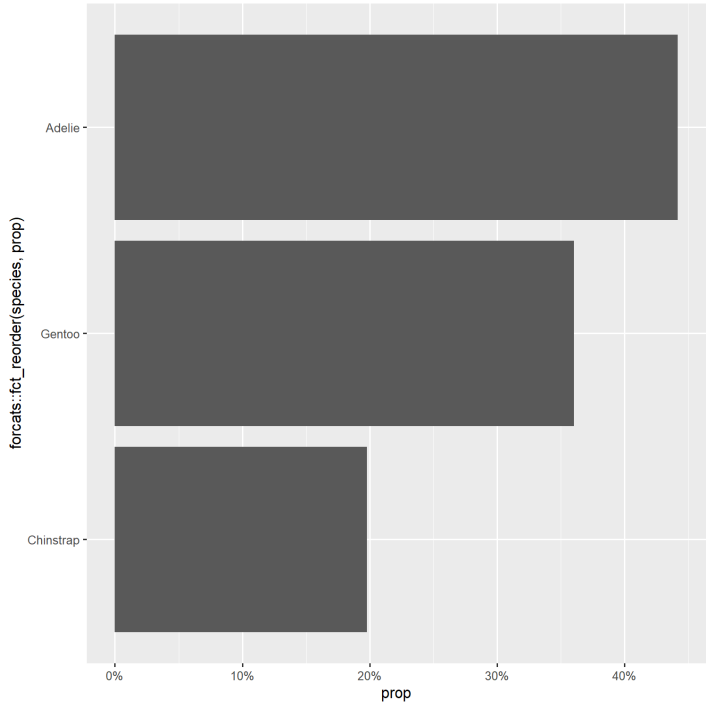
- You can easily store an `ggplot` object in a user-defined variable.
- Use `dplyr::count()` as shortcut for `group_by()` and `summarise(n = n())`.
- You can either set `aes()` and `data` in `ggplot()` (*global*) or in `geom_*()` (*local*). In the latter case the data and mappings are only active on the *geom*-level.

`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

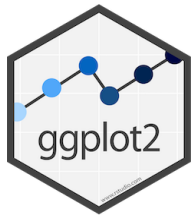
Other examples: Ordered bar chart

```
plot +
  geom_col(
    aes(
      x = prop,
      y = forcats::fct_reorder(species, prop))) +
  scale_x_continuous(
    labels = scales::label_percent(1.))
```

- Use `fct_reorder()` from the `forcats` package to reorder the levels of `species` by their relative frequency (`prop`).
- Finally, `scales::label_percent(1.)` formats the axis as percentages, rounded to the first value after the digit.

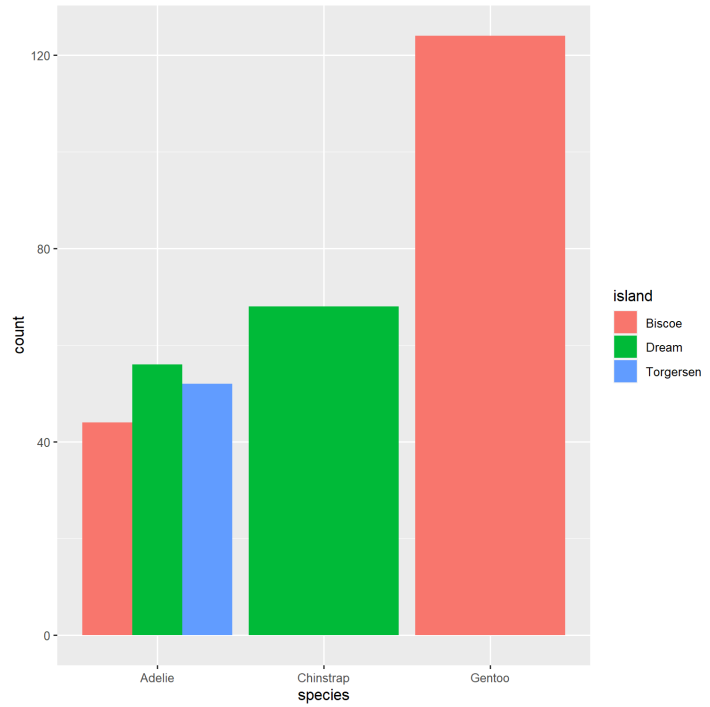


`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



Other examples: Adjacent bar chart

```
penguins %>%
  ggplot(aes(x = species)) +
    geom_bar(
      aes(
        fill = island),
        position = "dodge")
```

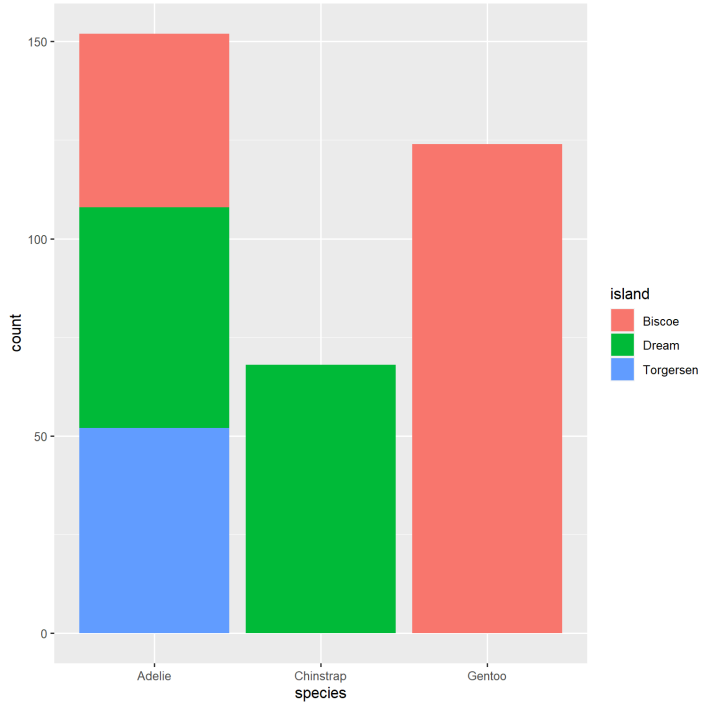


Note: `geom_col()` takes a `x` and `y` argument, whereas `geom_bar()` only takes a variable `x` and computes the `y`-quantity internally (e.g., the number of observations).

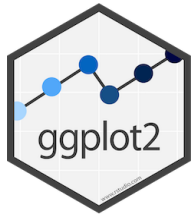
ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

Other examples: Stacked bar chart

```
penguins %>%
  ggplot(aes(x = species)) +
    geom_bar(
      aes(
        fill = island),
        position = "stack")
```



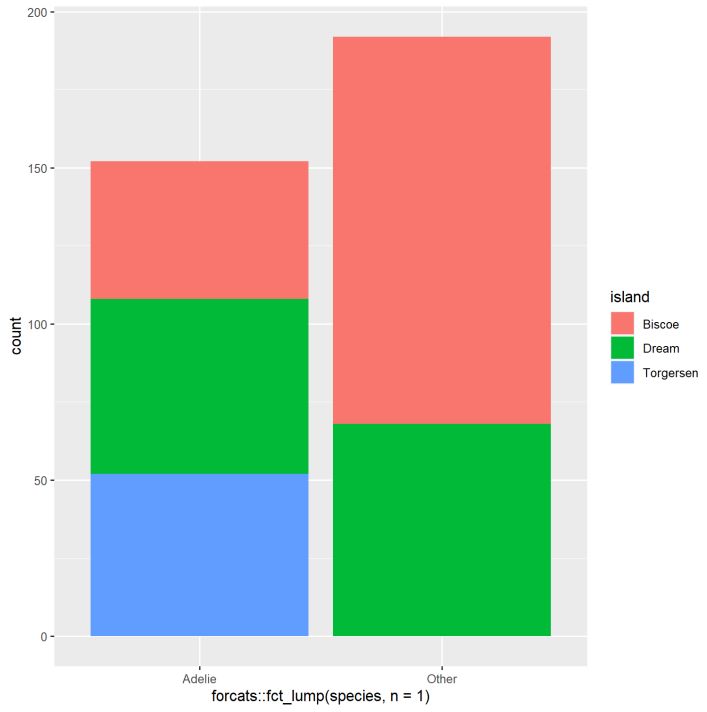
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



Other examples: Stacked bar chart

```
penguins %>%
  ggplot(aes(
    x = forcats::fct_lump(species, n = 1))) +
    geom_bar(
      aes(
        fill = island),
        position = "stack")
```

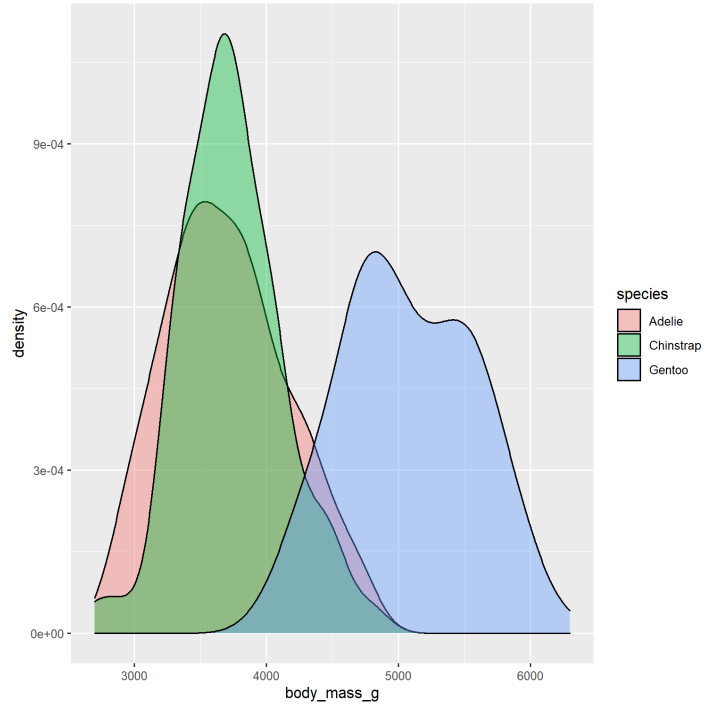
In this crude example we use `forcats::fct_lump()` to lumpy together all factor levels except the `n = 1` level(s) with the highest number of observations.



ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

Other examples: High-quality density plot

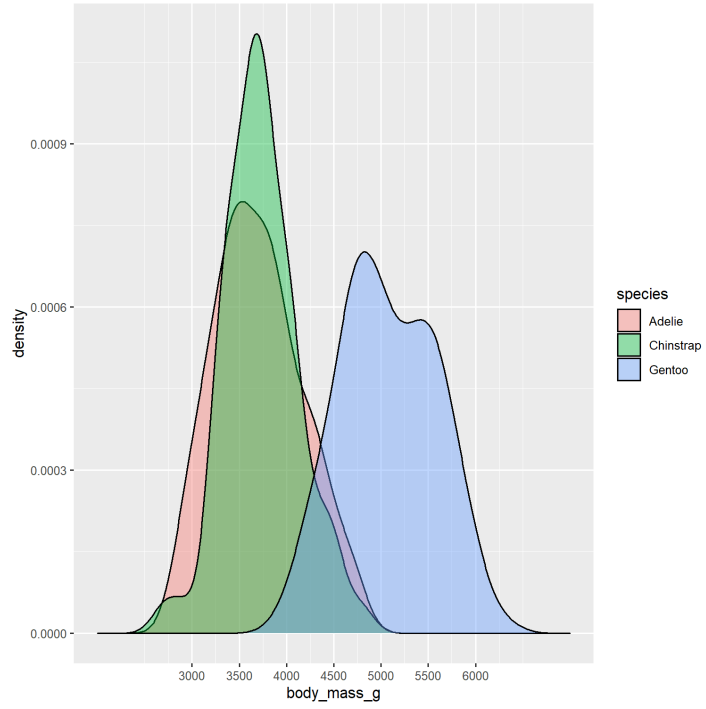
```
p <- penguins %>%
  ggplot(
    aes(x = body_mass_g)) +
    geom_density(
      aes(fill = species),
      na.rm = T,
      alpha = 0.4)

p
```



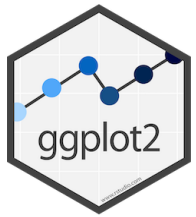
ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

Other examples: High-quality density plot

```
p <- p +
  scale_x_continuous(
    breaks = seq(from = 3000, to = 6000, by = 500)
    limits = c(2000, 7000)) +
  scale_y_continuous(
    labels = scales::label_comma(accuracy = 0.0001

p
```

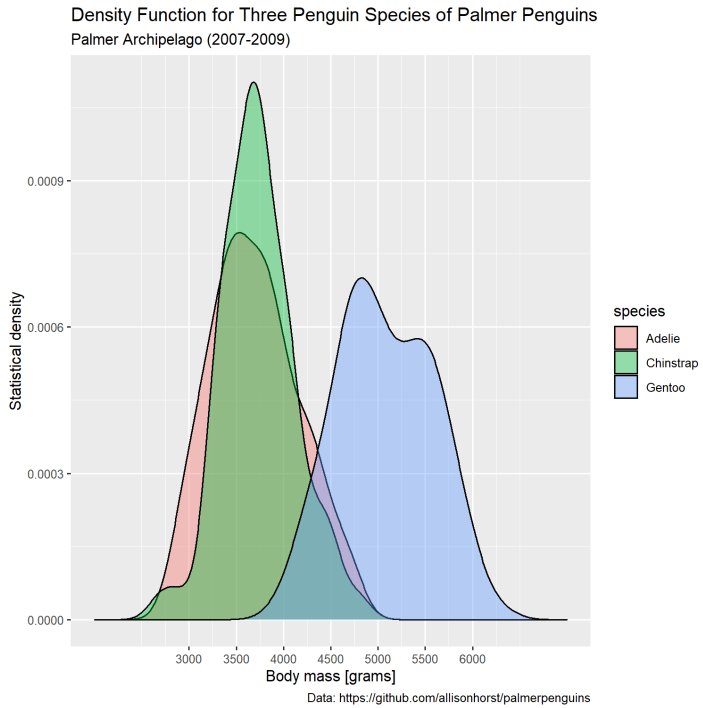


`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

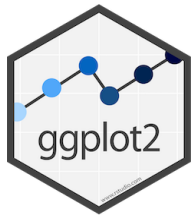


Other examples: High-quality density plot

```
p <- p +
  labs(
    title = "Density Function for Three Penguin Sp
    subtitle = "Palmer Archipelago (2007-2009)",
    caption = "Data: https://github.com/allisonhor
    x = "Body mass [grams]",
    y = "Statistical density"
  )

p
```



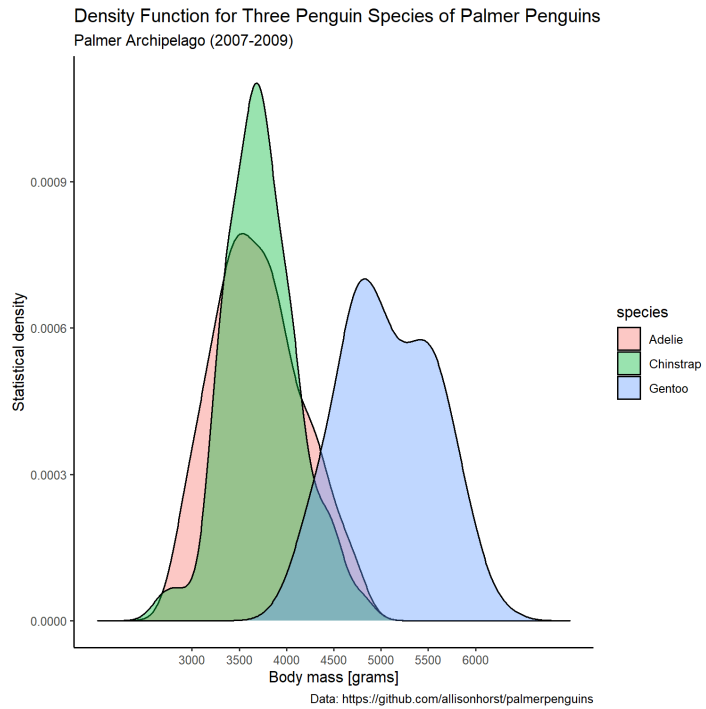
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



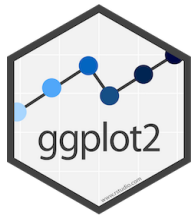
Other examples: High-quality density plot

```
p <- p +
  theme_classic() #also: theme_minimal()

p
```

The `theme` function allows to customize all elements of your plot which are not immediately related to your data, e.g., titles, labels, fonts, background, or legends. `ggplot2` also comes with a set of predefined themes (`theme_*()`) .

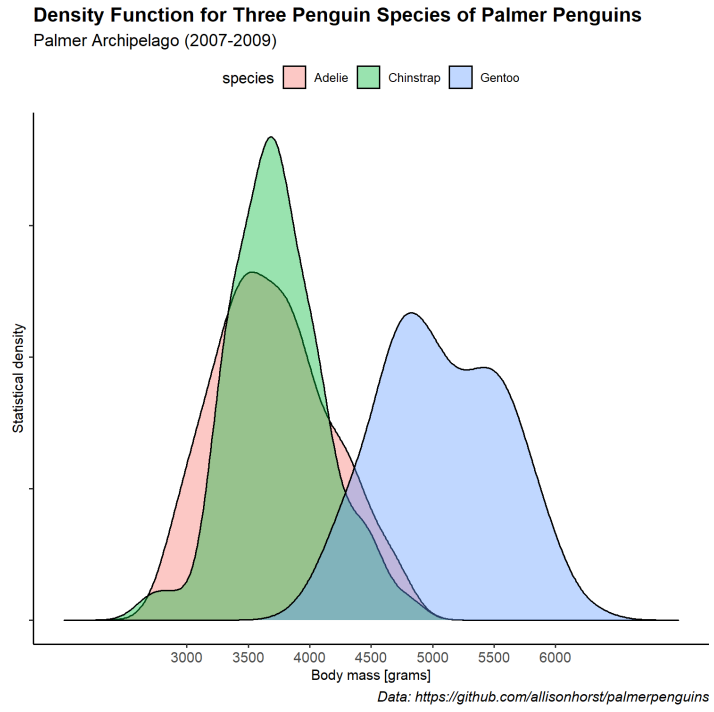


ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics

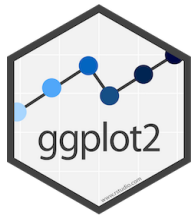


Other examples: High-quality density plot

```
p <- p +
  theme(
    legend.position = "top",
    plot.title = element_text(size = 14, face = "b
    plot.subtitle = element_text(size = 12),
    plot.caption = element_text(size = 10, face =
    axis.text.x = element_text(size = 10),
    axis.text.y = element_blank(),
    axis.title = element_text(size = 10),
  )

p
```



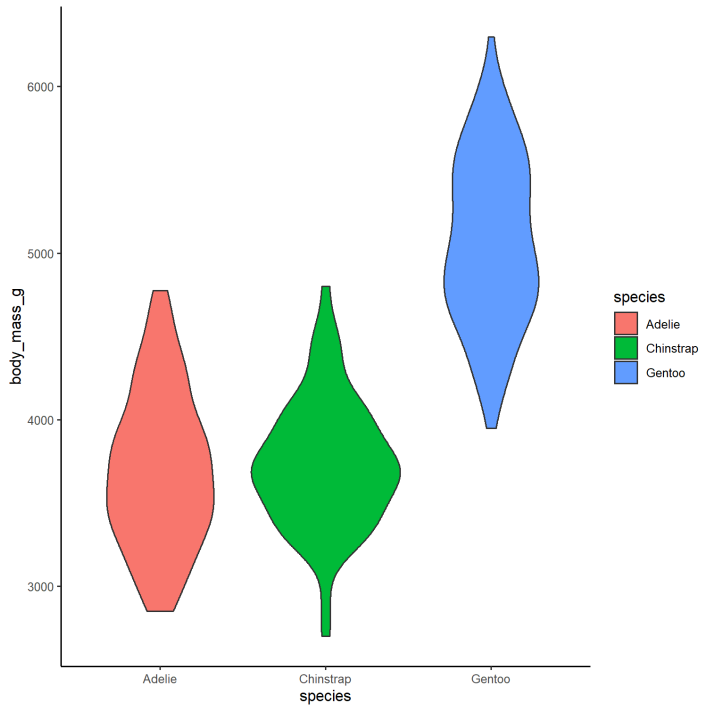
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



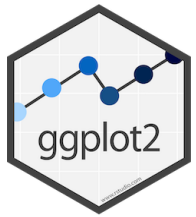
Other examples: Violin Plot

```
penguins %>%
  ggplot(
    aes(x = species, y = body_mass_g)) +
    geom_violin(aes(fill = species), na.rm = T) +
    theme_classic()

ggsave("./img/violin-plot.PNG",
       device = "png", dpi = 300)
```

- `geom_violin()` creates a cross-over version of a box-plot and a density plot, particularly suitable for visualizing continuous variables.
- `ggsave()` writes the most recent plot to disk.



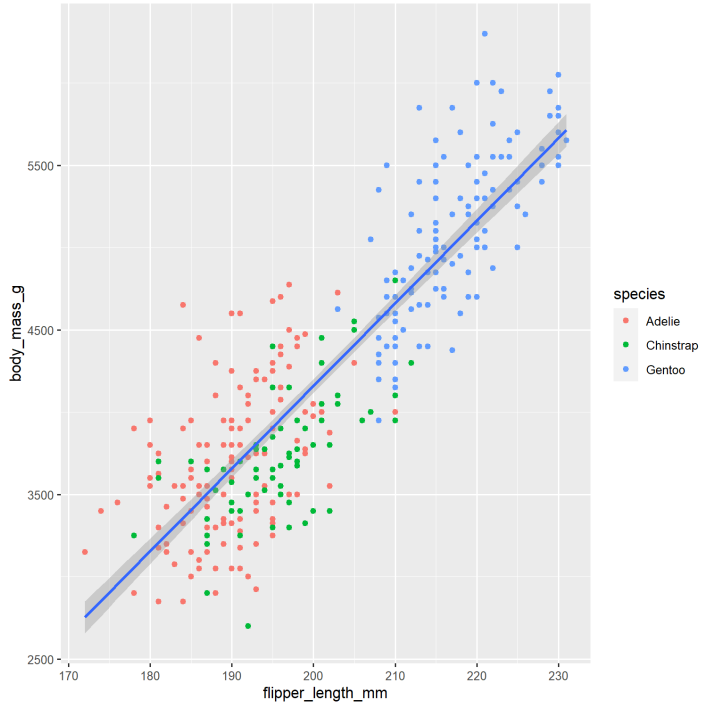
`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics



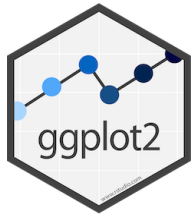
Other examples: Lines of Best Fit

```
penguins %>%
  drop_na %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g)) +
    geom_point(aes(color = species)) +
    geom_smooth(method = "lm", se = T)
```

- Use `geom_smooth()` to fit a smooth line to visualise the relationship between `x` and `y`.
- For the `method` argument specify one of: *lm* (linear model), *glm* (generalized linear model), *gam* (generalized additive model), *loess* (local regression).
- Set the `se` argument to `TRUE` to show standard error bands (i.e. uncertainty)!



`ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics

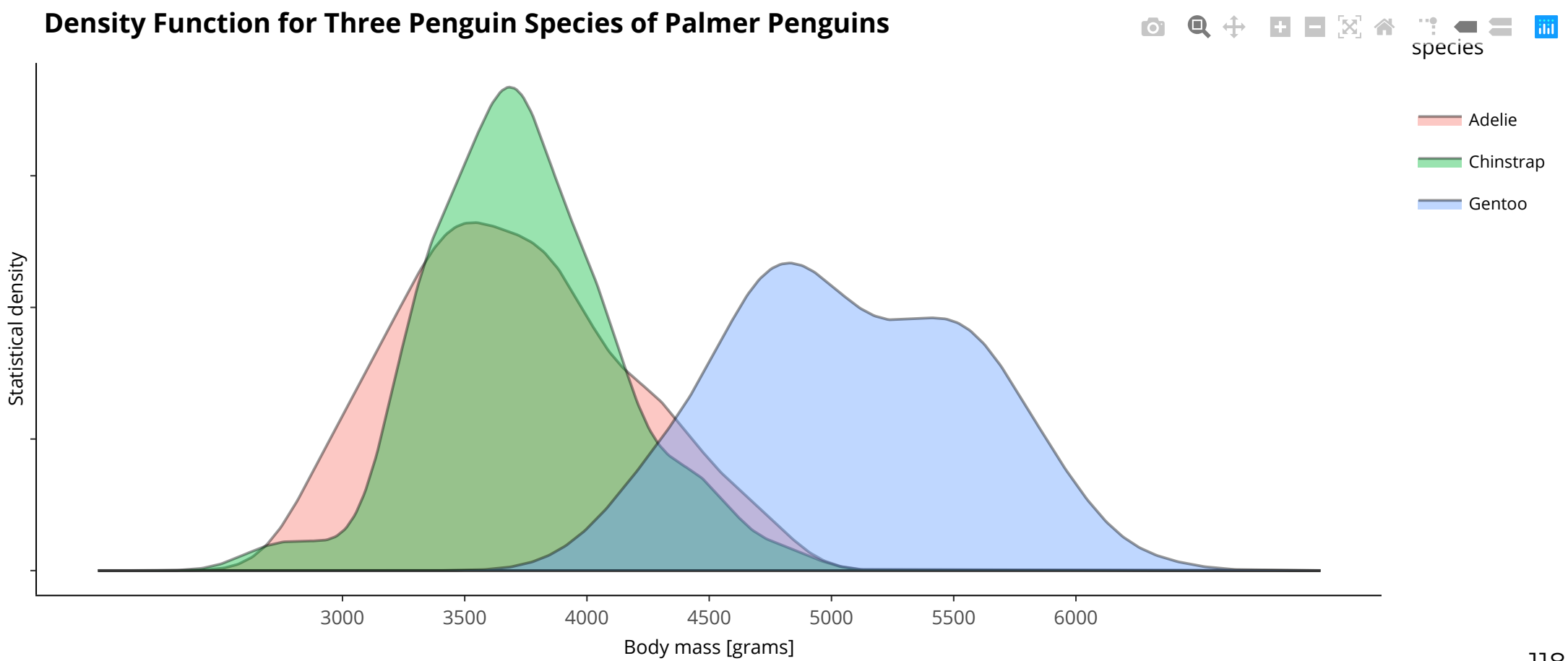


By now, there is a whole ecosystem (aka the **ggverse** [9]) of amazing packages around `ggplot2` that further extent its capabilities:

- `scales`: Scale Functions for Visualization
- `ggtext`: Improved Text Rendering Support for `ggplot2`
- `ggraph`: An Implementation of Grammar of Graphics for Graphs and Networks
- `plotly`: Create Interactive Web Graphics via `plotly.js`
- `patchwork`: The Composer of Plots
- `ggforce`: Accelerating `ggplot2`
- ...

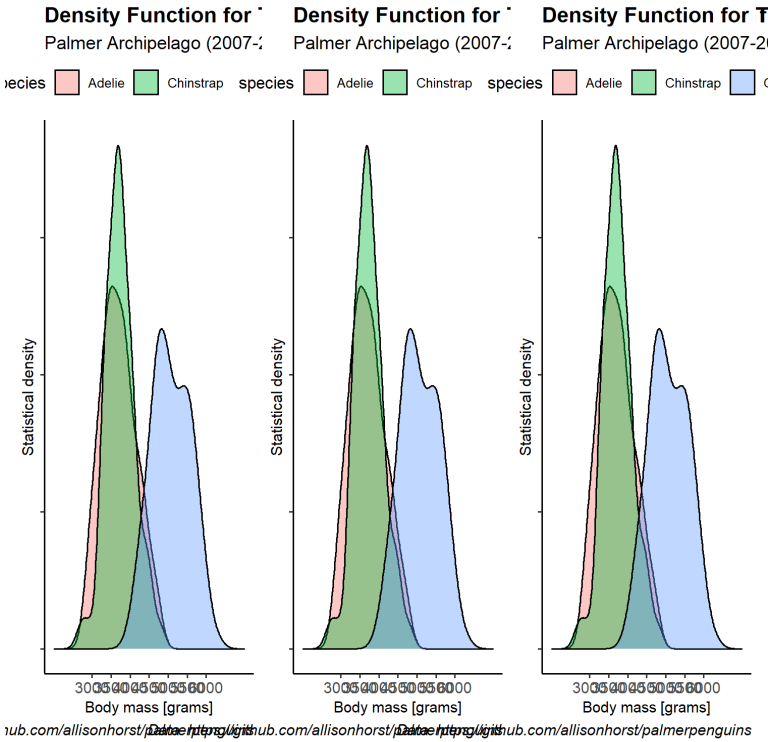
`plotly`: Interactive Web Graphics

```
plotly::ggplotly(p)
```

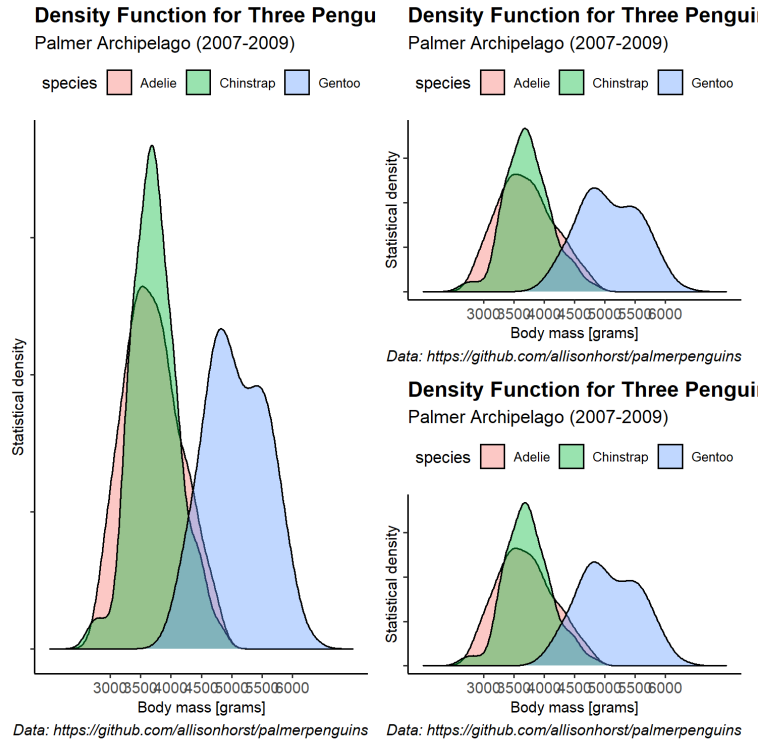


patchwork: The Composer of Plots


```
library(patchwork)
p + p + p
```

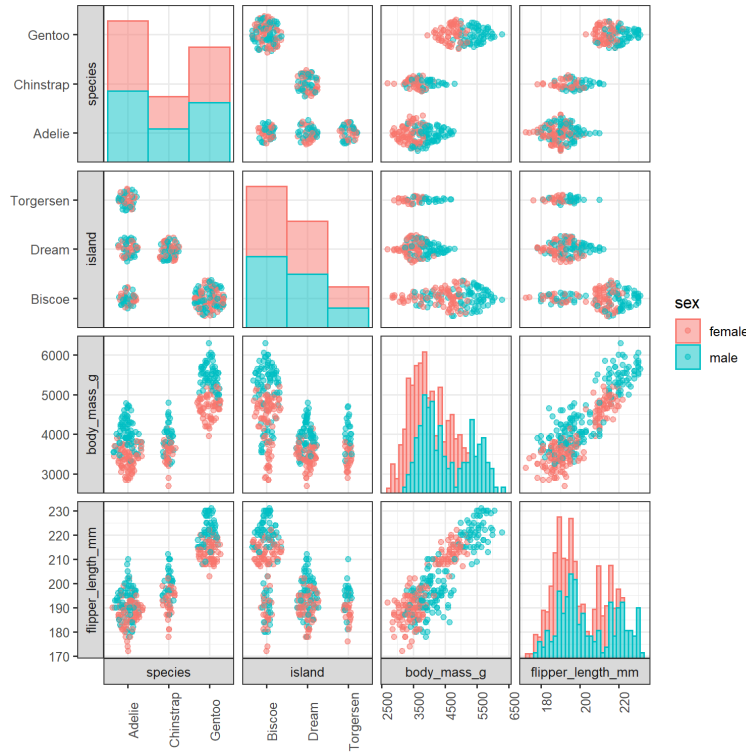



```
library(patchwork)
p + (p / p)
```



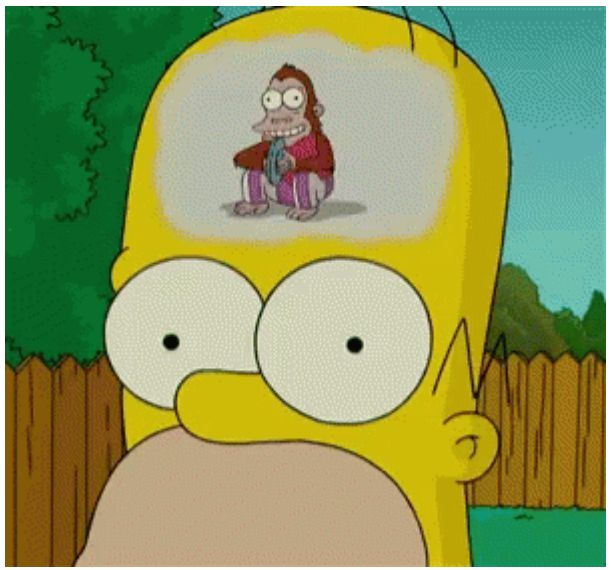
ggforce: Accelerating ggplot2


```
penguins %>%
  drop_na %>%
  ggplot(aes(x = .panel_x, y = .panel_y, col = sex
    ggforce::geom_autopoint(alpha = 0.5) +
    ggforce::geom_autohistogram(alpha = 0.5) +
    ggforce::facet_matrix(
      rows = vars(species, island, body_mass_g, fl
      switch = "both", layer.diag = 2) +
    theme_bw() +
    theme(axis.text.x = element_text(angle = 90))
```

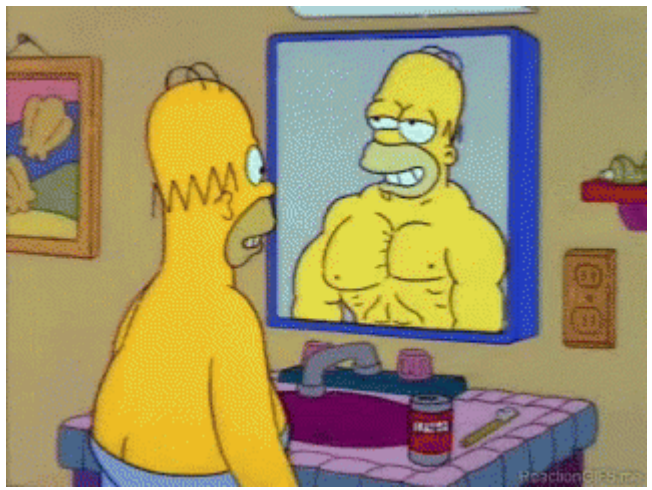


Thank You!

Right now:



After having mastered the Tidyverse:



But not all is great in the `Tidyverse` - be aware of its downsides.

References

[1]: **Wickham, H., et al. (2019):** Welcome to the Tidyverse. Journal of Open Source Software, Vol. 4, 2019 No. 43, p. 1686. URL: https://joss.theoj.org/papers/10.21105/joss.01686.pdf.

[2]: **Tidyverse team (2020):** Tidyverse Design Guide. URL: https://design.tidyverse.org/.

[3]: **Wickham, H. (2019):** The Tidy Tools Manifesto. URL: https://cran.r-project.org/web/packages/tidyverse/vignettes/manifesto.html.

[4]: **Wickham, H./Grolemund, G. (2017):** R for Data Science: Visualize, Model, Transform, Tidy, and Import Data. URL: https://r4ds.had.co.nz/tidy-data.html, chapter 12 (Tidy data).

[5]: **Barter, R. (2019):** Learn to purrr. URL: http://www.rebeccabarter.com/blog/2019-08-19_purrr/#simplest-usage-repeated-looping-with-map.

[6]: **Wickham, H. (2010):** A Layered Grammar of Graphics. Journal of Computational and Graphical Statistics, Vol. 19, 2010, No. 1, pp. 3-28.

[7]: **Wilkinson, L. (2005):** The Grammar of Graphics. Springer: New York 2005.

References

[8]: **Sarkar, D. (2018):** A Comprehensive Guide to the Grammar of Graphics for Effective Visualization of Multidimensional Data. towardsdatascience 2018-09-12. URL: https://towardsdatascience.com/a-comprehensive-guide-to-the-grammar-of-graphics-for-effective-visualization-of-multi-dimensional-1f92b4ed4149).

[9]: **Gahner, E. (2020):** Awesome `ggplot2`. GitHub 2020-09-25. URL: https://github.com/erikgahner/awesome-ggplot2.

Further Resources

Best read for starting in the `tidyverse`:
Wickham, H./Grolemund, G. (2017): R for Data Science: Visualize, Model, Transform, Tidy, and Import Data. URL: https://r4ds.had.co.nz/tidy-data.html.

Additional resource for diving deeper into the world of `ggplot2`:
Wickham, H/Navarro, D./Lin Pedersen, T. (2020): ggplot2: Elegant Graphics for Data Analysis. 3rd. edition, Online Publication 2020. URL: https://ggplot2-book.org/.

Stay up-to-date with recent developments in the `tidyverse`: https://www.tidyverse.org/blog/

Watch live-coding sessions related to the TidyTuesday Project, e.g., the episodes by David Robinson:
https://www.youtube.com/user/safe4democracy/videos

Credits

Educational resources are inspired by workshop materials of Garrett Grolemund and blog posts by Mine Çetinkaya-Rundel of the RStudio Education team.

`tidyverse` artworks and illustration are provided by Allison Horst.