Data Analysis and Unsupervised Learning

A tour of the `tidyverse`

MAP573 – Julien Chiquet

École Polytechnique, Autumn semester, 2020-2021

https://jchiquet.github.io/MAP573

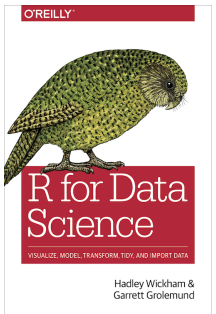
Outline

1. Introduction
2. Structures and types: `tibble`, `forcats`, `stringr`
3. data wrangling: `readr`, `tidyr`, `dplyr`
4. Manipulation: `magrittr`, `purrr`
5. Visualization: `ggplot2`

References

Many ideas/examples inspired/stolen there:

R for data science (Wickham & Grolemund, 2017), http://r4ds.had.co.nz



Tidyverse website, https://www.tidyverse.org/

Core packages

Prerequisites

Data Structures in base R

1. Atomic vector (integer, double, logical, character)
2. Recursive vector (list)
3. Factor
4. Matrix and array
5. Data Frame

R base programming

1. Control Statements (`if/then/else`, `while`, `for`)
2. Functions
3. Functionals (`[x]apply`)
4. Input/output (`read.[x]/write.[x]`)
5. Rstudio IDE

Outline

Tidy data: motivation

Collected data are (never) under a proper canonical format

> *"Happy families are all alike; every unhappy family is unhappy in its own way." – Leo Tolstoy*

> *"Tidy datasets are all alike, but every messy dataset is messy in its own way." – Hadley Wickham*[1]

[1] Rstudio's chief scientific advisor

Tidy data: what?

First, a subjective question

What is the *observation/statistical unit* in your data?

Definition

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.

In tidy data,

1. each variable forms a column,
2. each observation forms a row,
3. each observational unit have its own cell

Tidy data: why?

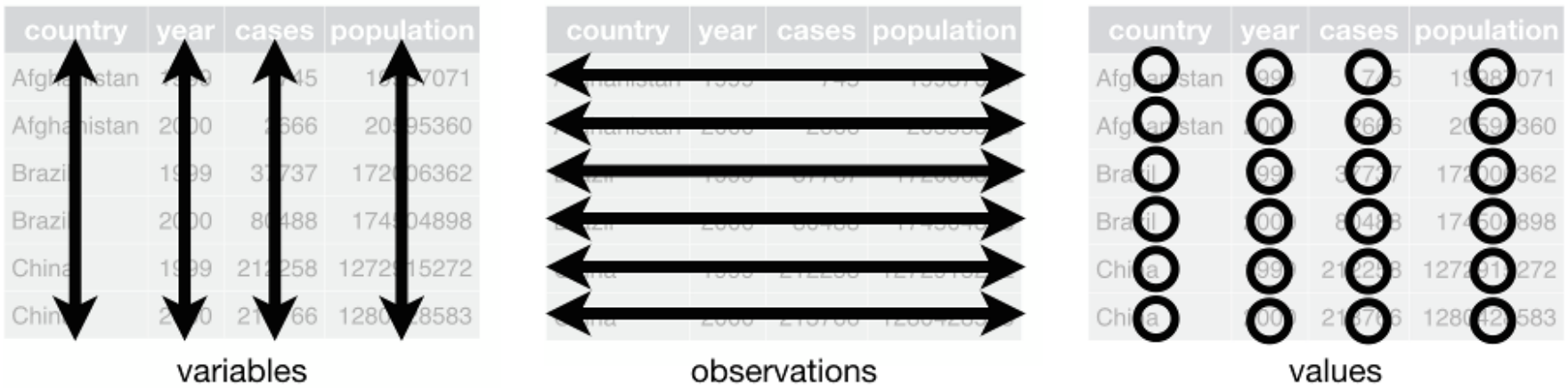


Figure 1: Tidy data

- make manipulation, visualization and modelling easier
- a common structure for all packages
- a philosophy for data representation (beyond the `R` framework)

Tidy or not ?

```
tidyr::table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

Tidy or not ?

```
tidyr::table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

Tidy or not ?

```
tidyr::table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

The process of data analysis

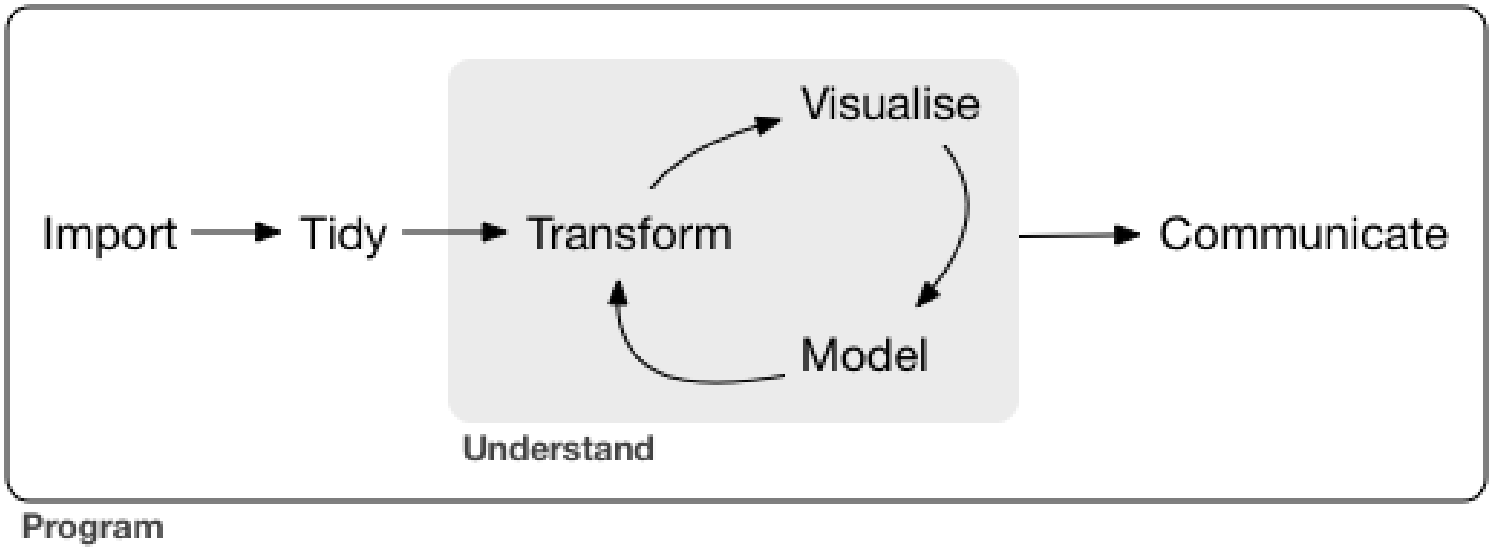


Figure 2: scheme for data analysis process

- **import:** read/load the data
- **tidy:** formating (individuals/variables data frame)
- **transform:** suppression/creation/filtering/selection
- **visualization:** representation and validation
- **model**: statistical fits
- **communication**: diffusion (web/talk/article)

The tidyverse

Definition

- contraction of 'tidy' ("well arranged) and 'universe'.
- an *opinionated collection* of `R` packages designed for data science.
- all packages share an underlying *design philosophy*, *grammar*, and *data structures*

Phylosophy

allows the user to focus on the important statistical questions rather than focusing on the technical aspects of data analysis

Let's have a look

The core `tidyverse` loads `ggplot2`, `tibble`, `tidyr`, `readr`, `purrr`, `stringr`, `forecats`, `dplyr` and others in a fancy and unconflicted way.

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------

## 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
## v purrr   0.3.4

## -- Conflicts ------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Tidyverse dependencies: a lot

```
tidyverse:::tidyverse_deps() %>% knitr::kable()
```

package	cran	local	behind
broom	0.7.0	0.7.0	FALSE
dbplyr	1.4.4	1.4.4	FALSE
dplyr	1.0.2	1.0.2	FALSE
forcats	0.5.0	0.5.0	FALSE
ggplot2	3.3.2	3.3.2	FALSE
haven	2.3.1	2.3.1	FALSE
hms	0.5.3	0.5.3	FALSE
httr	1.4.2	1.4.2	FALSE
jsonlite	1.7.1	1.7.1	FALSE
lubridate	1.7.9	1.7.9	FALSE
magrittr	1.5	1.5	FALSE
modelr	0.1.8	0.1.8	FALSE
pillar	1.4.6	1.4.6	FALSE
purrr	0.3.4	0.3.4	FALSE
readr	1.3.1	1.3.1	FALSE
readxl	1.3.1	1.3.1	FALSE
reprex	0.3.0	0.3.0	FALSE
rlang	0.4.7	0.4.7	FALSE
rvest	0.3.6	0.3.6	FALSE
stringr	1.4.0	1.4.0	FALSE
tibble	3.0.3	3.0.3	FALSE
tidyr	1.1.2	1.1.2	FALSE
xml2	1.3.2	1.3.2	FALSE

Packages roles and overview: types

tibble 

a modern re-imagining of the data frame

stringr 

a cohesive set of functions designed to make working with strings as easy as possible

forcats 

a suite of useful tools that solve common problems with factors

Packages roles and overview: wrangling

readr 

a fast and friendly way to read rectangular data (like csv, tsv, . . .)

tidyr 

a set of functions that help you get to tidy data

dplyr 

a consistent set of verbs that solve the most common data manipulation challenges

Packages roles and overview: manipulation

ggplot2 

a system for declaratively creating graphics, based on The Grammar of Graphics

purrr 

enhances R's functional programming (FP) toolkit

magrittr 

offers a set of operators which make your code more readable

Data analysis with the tidyverse (v1)

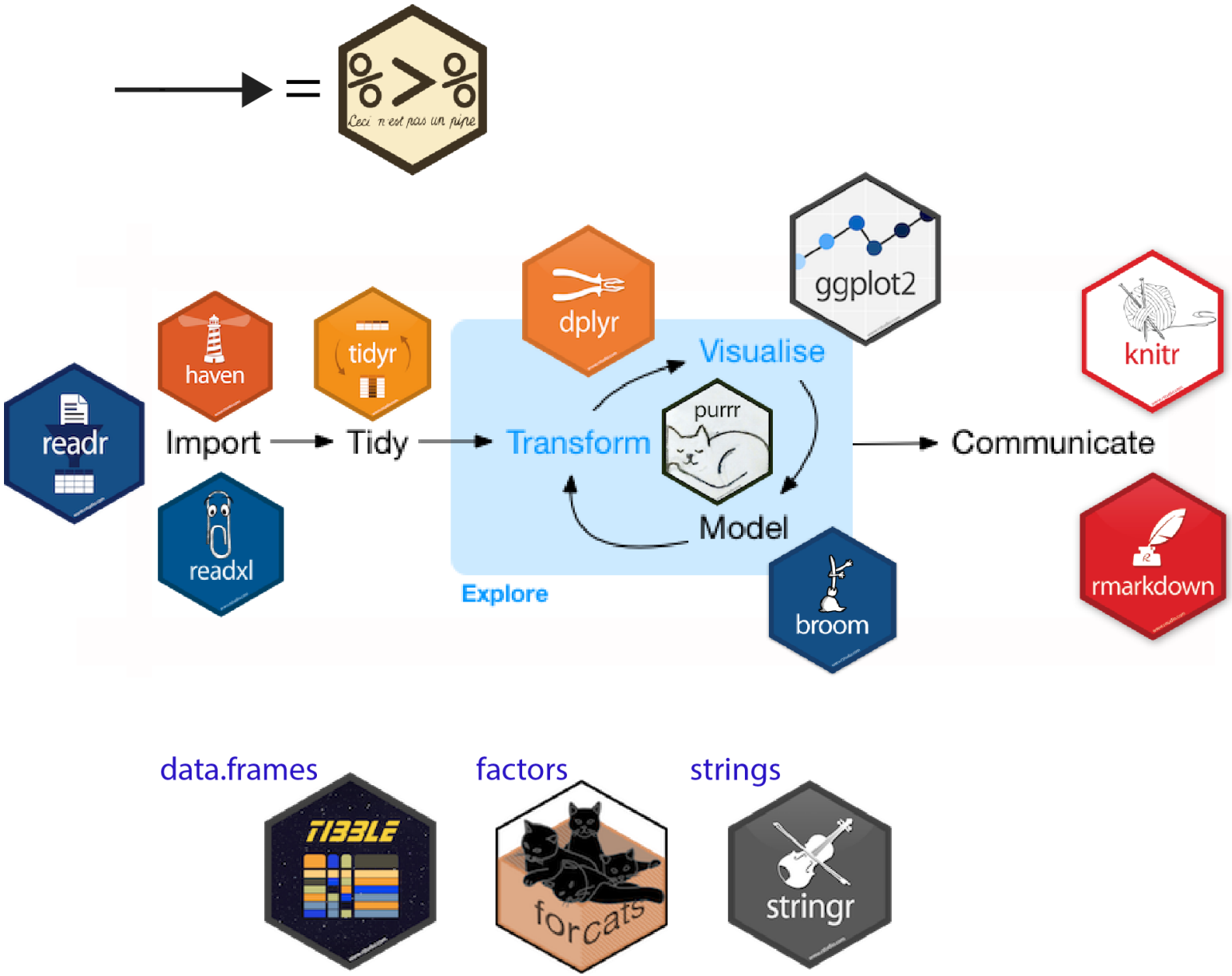


Figure 3: Updated scheme for data analysis process

Data analysis with the tidyverse (v2)

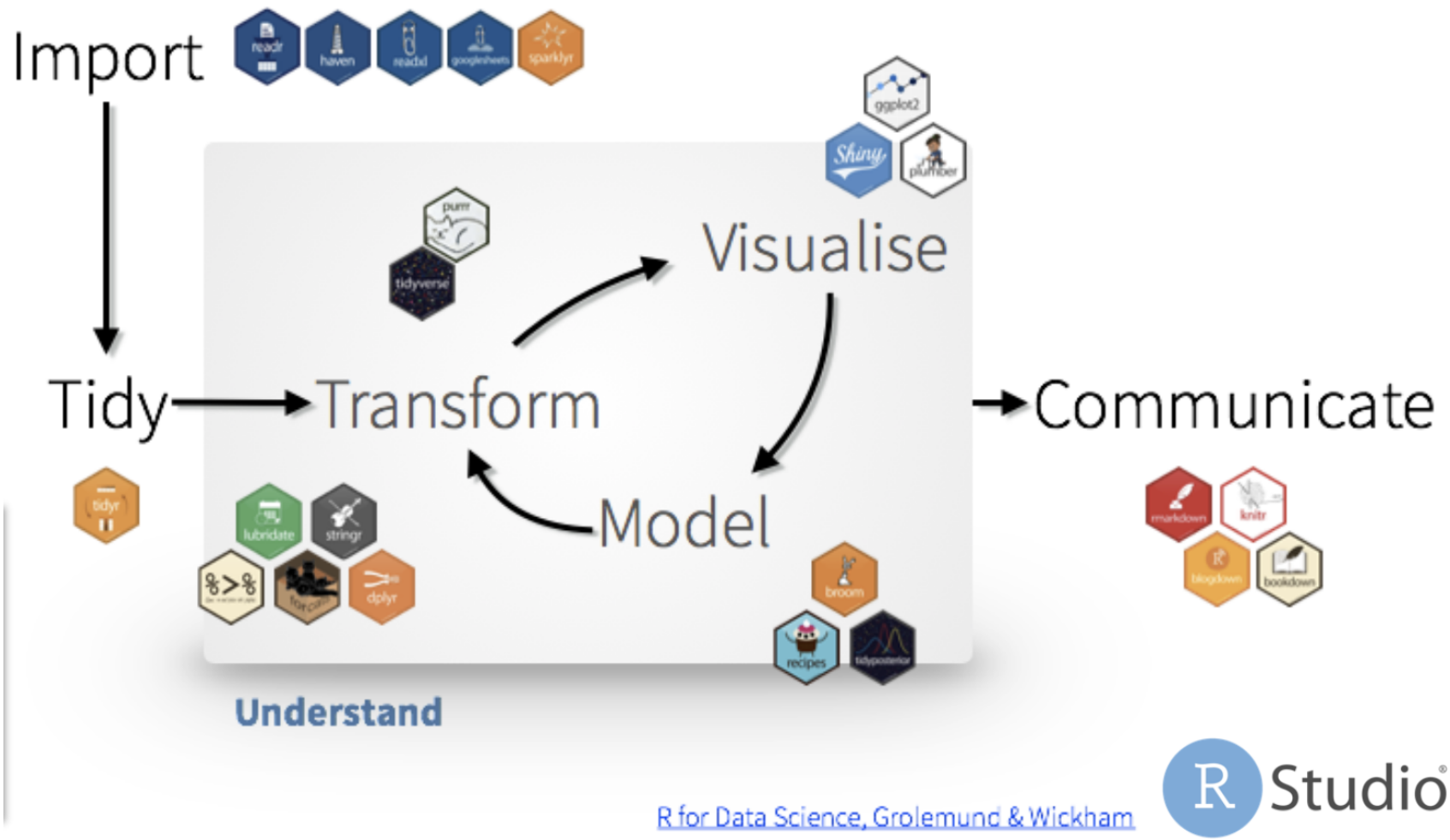


Figure 4: Updated scheme for data analysis process (v2)

Getting help

Rstudio's cheatsheets https://www.rstudio.com/resources/cheatsheets/

For a quick overview of the main features

Stackoverflow https://stackoverflow.com/

For all your specific questions

Tidyverse.org https://www.tidyverse.org

For an exhaustive documentation

Outline

Figure 5: a modern re-imagining of the data frame

`tibble` versus `data.frame`

`tibbles` (or `tbl_df`) are modern reimagining of the `data.frame`,

- *lazy*: do less (e.g. do not change variable names, types, no partial matching)
- *surly*: complain more (e.g. when a variable does not exist)

$\rightsquigarrow$ smarter output, no row-name, no partial matching, more flexible column names, etc.

$\rightsquigarrow$ 100% compatible with `data.frame`

Conversion from a data.frame

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
as_tibble(iris)
```

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

Conversion from a data.frame

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
as_tibble(iris)
```

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

Creating a tibble

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
```

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <dbl> <dbl>
## 1     1     1     2
## 2     2     1     5
## 3     3     1    10
## 4     4     1    17
## 5     5     1    26
```

Column names of a tibble

Names can start by any character. To refer such variables, use the backticks

```
tibble(`:)` = "smile", ` ` = "space", `2000` = "number")
```

```
## # A tibble: 1 x 3
##   `:)`  ` `   `2000`
##   <chr> <chr> <chr>
## 1 smile space number
```

Creating a tibble

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
```

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <dbl> <dbl>
## 1     1     1     2
## 2     2     1     5
## 3     3     1    10
## 4     4     1    17
## 5     5     1    26
```

Column names of a tibble

Names can start by any character. To refer such variables, use the backticks

```
tibble(`:)` = "smile", ` ` = "space", `2000` = "number")
```

```
## # A tibble: 1 x 3
##   `:)`  ` `   `2000`
##   <chr> <chr> <chr>
## 1 smile space number
```

Row names I

Row do not have names in a `tibble`

Solution

- one can use name by adding a specfic column
- `rownames_to_column ()` can help

Example

```
as_tibble(swiss, rownames = "Province")
```

```
## # A tibble: 47 x 7
##    Province Fertility Agriculture Examination Education Catholic
##    <chr>        <dbl>       <dbl>       <int>     <int>    <dbl>
##  1 Courtel~      80.2        17             15        12     9.96
##  2 Delemont      83.1        45.1            6         9    84.8
##  3 Franche~      92.5        39.7            5         5    93.4
##  4 Moutier       85.8        36.5           12         7    33.8
##  5 Neuvevi~      76.9        43.5           17        15     5.16
##  6 Porrent~      76.1        35.3            9         7    90.6
##  7 Broye         83.8        70.2           16         7    92.8
##  8 Glane         92.4        67.8           14         8    97.2
##  9 Gruyere       82.4        53.3           12         7    97.7
## 10 Sarine        82.9        45.2           16        13    91.4
## # ... with 37 more rows, and 1 more variable: Infant.Mortality <dbl>
```

Row names II

Example 2

```
as_tibble(rownames_to_column(swiss))
```

```
## # A tibble: 47 x 7
##    rowname Fertility Agriculture Examination Education Catholic Infant.Mortality
##    <chr>       <dbl>       <dbl>       <int>     <int>    <dbl>             <dbl>
##  1 Courte~      80.2        17             15        12     9.96              22.2
##  2 Delemo~      83.1        45.1            6         9    84.8               22.2
##  3 Franch~      92.5        39.7            5         5    93.4               20.2
##  4 Moutier      85.8        36.5           12         7    33.8               20.3
##  5 Neuvev~      76.9        43.5           17        15     5.16              20.6
##  6 Porren~      76.1        35.3            9         7    90.6               26.6
##  7 Broye        83.8        70.2           16         7    92.8               23.6
##  8 Glane        92.4        67.8           14         8    97.2               24.9
##  9 Gruyere      82.4        53.3           12         7    97.7               21
## 10 Sarine       82.9        45.2           16        13    91.4               24.4
## # ... with 37 more rows
```

Example 3

Row names III

```
my_tibble <- as_tibble(swiss, rownames = "Province")
head(as.data.frame(column_to_rownames(my_tibble, "Province")))
```

```
##              Fertility Agriculture Examination Education Catholic
## Courtelary        80.2        17.0          15        12     9.96
## Delemont          83.1        45.1           6         9    84.84
## Franches-Mnt      92.5        39.7           5         5    93.40
## Moutier           85.8        36.5          12         7    33.77
## Neuveville        76.9        43.5          17        15     5.16
## Porrentruy        76.1        35.3           9         7    90.57
##              Infant.Mortality
## Courtelary               22.2
## Delemont                 22.2
## Franches-Mnt             20.2
## Moutier                  20.3
## Neuveville               20.6
## Porrentruy               26.6
```

Consistency in subsetting

```
df <- data.frame(x = 1:9, y = LETTERS[1:9])
tbl <- tibble(x = 1:9, y = LETTERS[1:9])
```

```
class(df[, 1:2])
```

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

```
class(tbl[, 1:2])
```

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

```
class(df[, 1])
```

```
## [1] "integer"
```

```
class(tbl[, 1])
```

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

Consistency in subsetting

```
df <- data.frame(x = 1:9, y = LETTERS[1:9])
tbl <- tibble(x = 1:9, y = LETTERS[1:9])
```

```
class(df[, 1:2])
```

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

```
class(tbl[, 1:2])
```

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

```
class(df[, 1])
```

```
## [1] "integer"
```

```
class(tbl[, 1])
```

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

Consistency in subsetting

```
df <- data.frame(x = 1:9, y = LETTERS[1:9])
tbl <- tibble(x = 1:9, y = LETTERS[1:9])
```

```
class(df[, 1:2])
```

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

```
class(tbl[, 1:2])
```

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

```
class(df[, 1])
```

```
## [1] "integer"
```

```
class(tbl[, 1])
```

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

List-column

The type `list` is available for a column in `tibble`

- a `tibble` allows cells containing lists
- a `tibble` allows cells containing data frames.

```
subset(starwars, select = c('name', 'height', 'mass', 'hair_color', 'films', 'vehicles'))
```

```
## # A tibble: 87 x 6
##    name               height  mass hair_color    films      vehicles
##    <chr>               <int> <dbl> <chr>         <list>     <list>
##  1 Luke Skywalker        172    77 blond         <chr [5]> <chr [2]>
##  2 C-3PO                 167    75 <NA>          <chr [6]> <chr [0]>
##  3 R2-D2                  96    32 <NA>          <chr [7]> <chr [0]>
##  4 Darth Vader           202   136 none          <chr [4]> <chr [0]>
##  5 Leia Organa           150    49 brown         <chr [5]> <chr [1]>
##  6 Owen Lars             178   120 brown, grey   <chr [3]> <chr [0]>
##  7 Beru Whitesun lars    165    75 brown         <chr [3]> <chr [0]>
##  8 R5-D4                  97    32 <NA>          <chr [1]> <chr [0]>
##  9 Biggs Darklighter     183    84 black         <chr [1]> <chr [0]>
## 10 Obi-Wan Kenobi        182    77 auburn, white <chr [6]> <chr [1]>
## # ... with 77 more rows
```

List-column: put a vector in each case

```
head(starwars$films, 4)
```

```
## [[1]]
## [1] "The Empire Strikes Back" "Revenge of the Sith"
## [3] "Return of the Jedi"      "A New Hope"
## [5] "The Force Awakens"
##
## [[2]]
## [1] "The Empire Strikes Back" "Attack of the Clones"
## [3] "The Phantom Menace"      "Revenge of the Sith"
## [5] "Return of the Jedi"      "A New Hope"
##
## [[3]]
## [1] "The Empire Strikes Back" "Attack of the Clones"
## [3] "The Phantom Menace"      "Revenge of the Sith"
## [5] "Return of the Jedi"      "A New Hope"
## [7] "The Force Awakens"
##
## [[4]]
## [1] "The Empire Strikes Back" "Revenge of the Sith"
## [3] "Return of the Jedi"      "A New Hope"
```

Figure 6: a suite of useful tools that solve common problems with factors

`forcats` versus `base` factors

- easy use in conjuction with other tidyverse packages
- correct inconsistent behaviours of `R` base factors facilities

⇝ Slides borrowed to Antoine Bichat
https://abichat.github.io/slides/factors_dates_strings

Convert to factor

Base R

```
fruits <- c("banana", "apple", "mango", "apple", "pear", "apple",
            "banana", "pitaya", "mango", "mango", "apple")
as.factor(fruits) # Use alphabetical order
```

```
##  [1] banana apple  mango  apple  pear   apple  banana pitaya mango  mango
## [11] apple
## Levels: apple banana mango pear pitaya
```

forcats equivalent

```
forcats::as_factor(fruits) # Use appearance order
```

```
##  [1] banana apple  mango  apple  pear   apple  banana pitaya mango  mango
## [11] apple
## Levels: banana apple mango pear pitaya
```

```
## equivalent to factor(countries, levels = unique(countries))
```

Remark

Appearance order increases reproducibility (independent from `locale()`).

Change level names

recode

```
fct_recode(fruits, dragonfruit = "pitaya")
```

```
##  [1] banana      apple       mango       apple       pear        apple
##  [7] banana      dragonfruit mango       mango       apple
## Levels: apple banana mango pear dragonfruit
```

relabel

```
fct_relabel(fruits, stringr::str_to_title)
```

```
##  [1] Banana Apple  Mango  Apple  Pear   Apple  Banana Pitaya Mango  Mango
## [11] Apple
## Levels: Apple Banana Mango Pear Pitaya
```

Remark

When converting from strings to factors, `fct_recode()` and `fct_relabel()` use alphabetical order.

Reorder levels

You can reorder levels:

- manually with `fct_relevel()`,
- by appearance with `fct_inorder()`,
- by frequency with `fct_infreq()`,
- according to another variable with `fct_reorder()`,
- according to the last value of another variable with `fct_reorder2()`,
- randomly with `fct_shuffle()`,
- by reversing order with `fct_rev()`...

Reorder by frequency

```
fct_count(fruits, sort = TRUE)
```

```
## # A tibble: 5 x 2
##   f          n
##   <fct>  <int>
## 1 apple      4
## 2 mango      3
## 3 banana     2
## 4 pear       1
## 5 pitaya     1
```

```
fct_infreq(fruits)
```

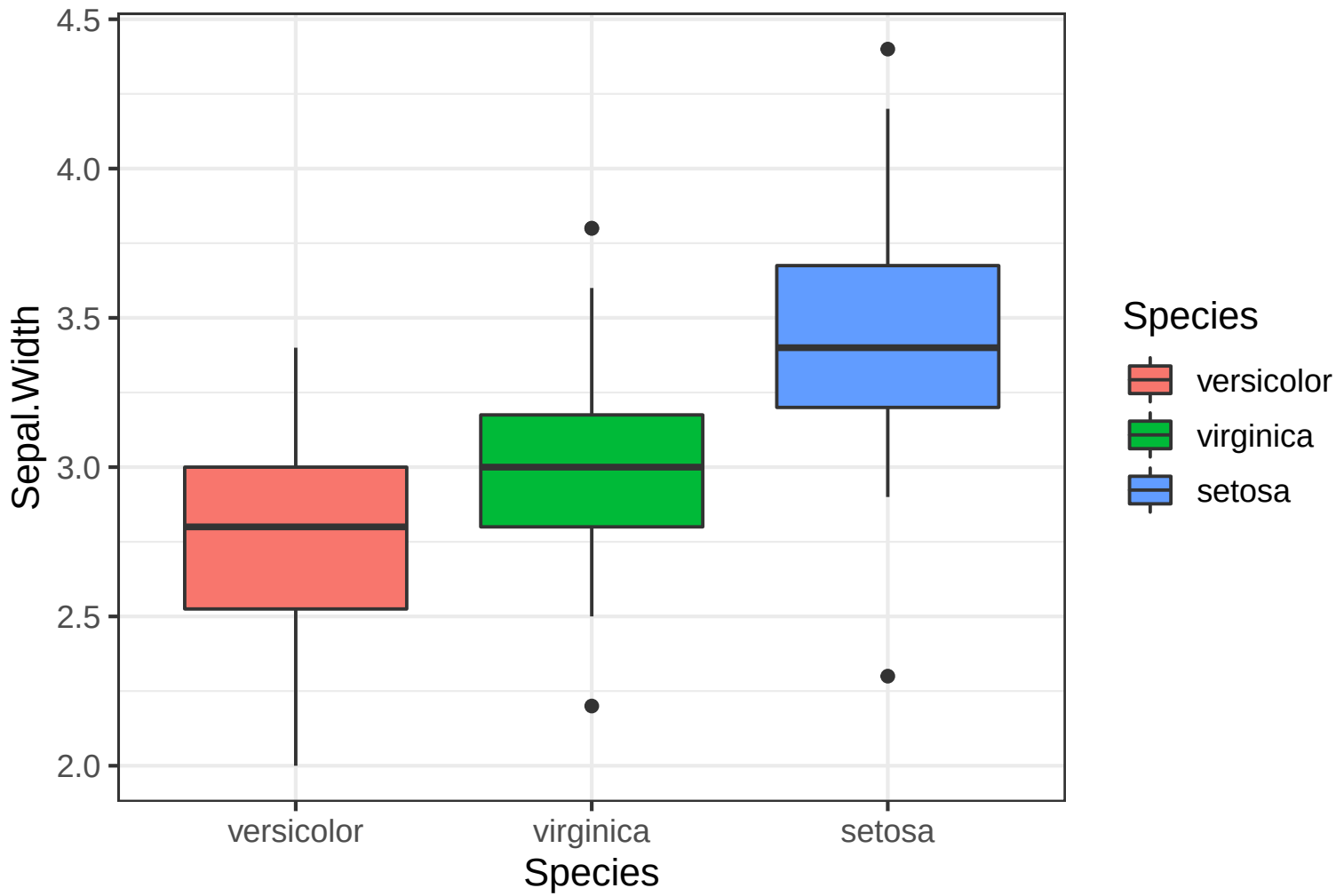
```
##  [1] banana apple  mango  apple  pear   apple  banana pitaya mango  mango
## [11] apple
## Levels: apple mango banana pear pitaya
```

According to another variable

```
levels(iris$Species)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

```
iris$Species <- fct_reorder(iris$Species, iris$Sepal.Width)
ggplot(iris) + aes(x = Species, y = Sepal.Width, fill = Species) + geom_boxplot()
```



Handling many modalities

A data set about nuclear testings around the world (enjoy!):

```
library(tidytuesdayR)
nuclear <- tt_load("2019-08-20")$nuclear_explosion
```

```
##
##  Downloading file 1 of 1: `nuclear_explosions.csv`
```

```
nuclear$type <- stringr::str_to_title(nuclear$type)
fct_count(nuclear$type, sort = TRUE)
```

```
## # A tibble: 20 x 2
##    f             n
##    <fct>     <int>
##  1 Shaft      1015
##  2 Tunnel      310
##  3 Atmosph     185
##  4 Shaft/Gr     85
##  5 Airdrop      78
##  6 Tower        75
##  7 Balloon      62
##  8 Surface      62
##  9 Shaft/Lg     56
## 10 Barge        40
## 11 Ug           32
## 12 Gallery      13
## 13 Rocket       13
## 14 Crater        9
## 15 Uw            8
## 16 Space         4
```

Lump least common factors

```
table(fct_lump(nuclear$type, n = 5))
```

```
##
##  Airdrop  Atmosph    Shaft Shaft/Gr   Tunnel    Other
##       78      185     1015       85      310      378
```

```
table(fct_lump_min(nuclear$type, min = 20))
```

```
##
##  Airdrop  Atmosph  Balloon    Barge    Shaft Shaft/Gr Shaft/Lg  Surface
##       78      185       62       40     1015       85       56       62
##    Tower   Tunnel       Ug    Other
##       75      310       32       51
```

Lump least common factors

```
table(fct_lump(nuclear$type, n = 5))
```

```
##
##  Airdrop  Atmosph    Shaft Shaft/Gr   Tunnel    Other
##       78      185     1015       85      310      378
```

```
table(fct_lump_min(nuclear$type, min = 20))
```

```
##
##  Airdrop  Atmosph  Balloon    Barge    Shaft Shaft/Gr Shaft/Lg  Surface
##       78      185       62       40     1015       85       56       62
##    Tower   Tunnel       Ug    Other
##       75      310       32       51
```

Manually collapse levels

```
nuclear_collapsed <- fct_collapse(nuclear$type,
    Air = c("Atmosph", "Airdrop", "Balloon", "Rocket"),
    Underground = c("Shaft", "Tunnel", "Shaft/Gr", "Shaft/Lg", "Ug", "Gallery"),
    Water = c("Barge", "Uw", "Ship", "Water Su", "Watersur"),
    other_level = "Other")
fct_count(nuclear_collapsed, sort = TRUE)
```

```
## # A tibble: 4 x 2
##   f               n
##   <fct>       <int>
## 1 Underground  1511
## 2 Air           338
## 3 Other         151
## 4 Water          51
```

Figure 7: cohesive set of functions designed to work more easily with

`stringr` versus `base` string utilities

String manipulation is cumbersome in `R` base. However, string plays a big role in many data cleaning and preparation.

- easy use in conjunction with other tidyverse packages
- faster and correct implementations of common string manipulations

See A. Bichat's slides: https://abichat.github.io/slides/factors_dates_strings/#73

{lubridate}

Figure 8: cohesive set of functions designed to make working with date as easy as possible

`lubridate` versus `base R` utilities

Data manipulation is *always* boring[2], whatever the system used...

See A. Bichat's slides: https://abichat.github.io/slides/factors_dates_strings/#38

[2] In my humble opinion...

Outline

{readr}

Figure 9: a fast and friendly way to read rectangular data (like csv, tsv and so on)

- offer coherent/unified functions compared to `base::read.table` and friends
- offer interactive reading
- output `tibble` rather than `data.frame`
- `read_csv`, `read_delim`, `read_rds`, `read_file`, `read_table`, etc

⇝ Read the manual when needed!

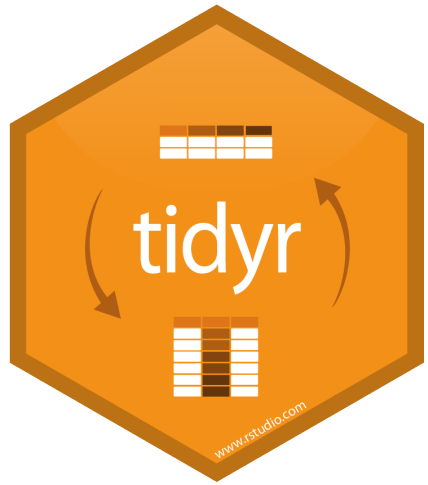


Figure 10: a set of functions that help you get to tidy data

```
library(tidyr)
```

⇝ helps transforming messy data sets into tidy data sets.

- evolution of base function `reshape`
- more useful functions are `spread`/`gather`, `unite`/`separate`, `nest`/`unest`

Grades dataset

```
grades <- tibble(
  Name = c("Tommy", "Mary", "Gary", "Cathy"),
  Sexage = c("m.15", "f.15", "m.16", "f.14"),
  Test1 = c(10, 15, 16, 14),
  Test2 = c(11, 13, 10, 12),
  Test3 = c(12, 13, 17, 10)
  )
grades
```

```
## # A tibble: 4 x 5
##   Name  Sexage Test1 Test2 Test3
##   <chr> <chr>  <dbl> <dbl> <dbl>
## 1 Tommy m.15      10    11    12
## 2 Mary  f.15      15    13    13
## 3 Gary  m.16      16    10    17
## 4 Cathy f.14      14    12    10
```

Name	Sexage	Test1	Test2	Test3
Tommy	m.15	10	11	12
Mary	f.15	15	13	13
Gary	m.16	16	10	17
Cathy	f.14	14	12	10

separate()

Separate one column into multiple columns

```
grades <- separate(grades, Sexage, into = c("Sex", "Age"))
grades
```

```
## # A tibble: 4 x 6
##   Name  Sex   Age   Test1 Test2 Test3
##   <chr> <chr> <chr> <dbl> <dbl> <dbl>
## 1 Tommy m     15       10    11    12
## 2 Mary  f     15       15    13    13
## 3 Gary  m     16       16    10    17
## 4 Cathy f     14       14    12    10
```

Name	Sex	Age	Test1	Test2	Test3
Tommy	m	15	10	11	12
Mary	f	15	15	13	13
Gary	m	16	16	10	17
Cathy	f	14	14	12	10

Remark

The inverse of `separate()` is `unite()`

gather()

Gather columns into key-value pairs

```
grades <- gather(grades, Test1, Test2, Test3, key = Test, value = Grade)
grades
```

```
## # A tibble: 12 x 5
##    Name  Sex   Age   Test  Grade
##    <chr> <chr> <chr> <chr> <dbl>
##  1 Tommy m     15    Test1    10
##  2 Mary  f     15    Test1    15
##  3 Gary  m     16    Test1    16
##  4 Cathy f     14    Test1    14
##  5 Tommy m     15    Test2    11
##  6 Mary  f     15    Test2    13
##  7 Gary  m     16    Test2    10
##  8 Cathy f     14    Test2    12
##  9 Tommy m     15    Test3    12
## 10 Mary  f     15    Test3    13
## 11 Gary  m     16    Test3    17
## 12 Cathy f     14    Test3    10
```

Remark 1

The inverse of `gather()` is `spread()`

Remark 2

`pivot_longer()`/`pivot_wider()`: updates of gather and spread

nest()

Nesting creates a list-column of data frames

```
nest(iris, !Species)
```

```
## # A tibble: 3 x 2
##   Species    data
##   <fct>      <list>
## 1 setosa     <tibble [50 x 4]>
## 2 versicolor <tibble [50 x 4]>
## 3 virginica  <tibble [50 x 4]>
```

Remark

The inverse of `nest()` is `unnest()`

dplyr

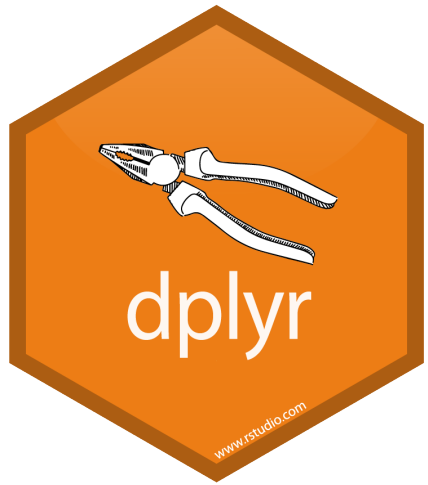


Figure 11: a consistent set of verbs (*a grammar*) that solves the most common data manipulation challenges

Typical operations

- create and pick variables
- pick and reorder observations
- create summaries
- ...

$\rightsquigarrow$ Functions in this package are verbs and work similarly

mtcars dataset

```
data(mtcars)
as_tibble(mtcars)
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

Select rows with `filter()`

Arguments

1. data
2. filtering expressions

Output

- a `tibble`
- **do not modify** the original data

Example

```
filter(mtcars, cyl == 4, mpg > 30)
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

Reorder rows with `arrange()`

Principle

works like `filter()` but reorder rows according to a series of conditions

Example

```
as_tibble(arrange(mtcars, desc(carb), mpg))
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  15       8  301    335  3.54  3.57  14.6     0     1     5     8
##  2  19.7     6  145    175  3.62  2.77  15.5     0     1     5     6
##  3  10.4     8  472    205  2.93  5.25  18.0     0     0     3     4
##  4  10.4     8  460    215  3     5.42  17.8     0     0     3     4
##  5  13.3     8  350    245  3.73  3.84  15.4     0     0     3     4
##  6  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  7  14.7     8  440    230  3.23  5.34  17.4     0     0     3     4
##  8  15.8     8  351    264  4.22  3.17  14.5     0     1     5     4
##  9  17.8     6  168.   123  3.92  3.44  18.9     1     0     4     4
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

Selecting columns with `select()` I

Similar to `base::subsect(, select = c("","") )`

With names

can be quoted or unquoted

```
as_tibble(select(mtcars, mpg, 'wt', cyl))
```

```
## # A tibble: 32 x 3
##      mpg    wt   cyl
##    <dbl> <dbl> <dbl>
##  1  21    2.62     6
##  2  21    2.88     6
##  3  22.8  2.32     4
##  4  21.4  3.22     6
##  5  18.7  3.44     8
##  6  18.1  3.46     6
##  7  14.3  3.57     8
##  8  24.4  3.19     4
##  9  22.8  3.15     4
## 10  19.2  3.44     6
## # ... with 22 more rows
```

Selecting columns with `select()` II

With indexes

```
as_tibble(select(mtcars, 1,2,5:7))
```

```
## # A tibble: 32 x 5
##      mpg   cyl  drat    wt  qsec
##    <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  3.9   2.62  16.5
##  2  21       6  3.9   2.88  17.0
##  3  22.8     4  3.85  2.32  18.6
##  4  21.4     6  3.08  3.22  19.4
##  5  18.7     8  3.15  3.44  17.0
##  6  18.1     6  2.76  3.46  20.2
##  7  14.3     8  3.21  3.57  15.8
##  8  24.4     4  3.69  3.19  20
##  9  22.8     4  3.92  3.15  22.9
## 10  19.2     6  3.92  3.44  18.3
## # ... with 22 more rows
```

Selecting columns with `select()` III

With scoping

See `starts_with`, `end_with`, `matches`, `any_of`, `last_col`, `everything` ...

```
as_tibble(select(mtcars, contains('t')))
```

```
## # A tibble: 32 x 2
##     drat    wt
##    <dbl> <dbl>
##  1  3.9   2.62
##  2  3.9   2.88
##  3  3.85  2.32
##  4  3.08  3.22
##  5  3.15  3.44
##  6  2.76  3.46
##  7  3.21  3.57
##  8  3.69  3.19
##  9  3.92  3.15
## 10  3.92  3.44
## # ... with 22 more rows
```

Renaming columns with `rename()`

`rename()` keeps all variables

```
as_tibble(rename(iris, petal_length = Petal.Length))
```

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

Renaming columns with `select()`

Renaming can be done with `select()`

`select()` only keeps the variables specified

```
as_tibble(select(iris, petal_length = Petal.Length))
```

```
## # A tibble: 150 x 1
##    petal_length
##           <dbl>
##  1          1.4
##  2          1.4
##  3          1.3
##  4          1.5
##  5          1.4
##  6          1.7
##  7          1.4
##  8          1.5
##  9          1.4
## 10          1.5
## # ... with 140 more rows
```

Add new variables with `mutate()`

`mutate` keeps the existing variables

```
as_tibble(
  mutate(mtcars,
         cyl2 = 2 * cyl,
         cyl4 = 2 * cyl2,
         disp = disp * 0.0163871,
         drat = NULL)
)
```

```
## # A tibble: 32 x 12
##      mpg   cyl  disp    hp    wt  qsec    vs    am  gear  carb  cyl2  cyl4
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  2.62   110  2.62  16.5     0     1     4     4    12    24
##  2  21       6  2.62   110  2.88  17.0     0     1     4     4    12    24
##  3  22.8     4  1.77    93  2.32  18.6     1     1     4     1     8    16
##  4  21.4     6  4.23   110  3.22  19.4     1     0     3     1    12    24
##  5  18.7     8  5.90   175  3.44  17.0     0     0     3     2    16    32
##  6  18.1     6  3.69   105  3.46  20.2     1     0     3     1    12    24
##  7  14.3     8  5.90   245  3.57  15.8     0     0     3     4    16    32
##  8  24.4     4  2.40    62  3.19  20       1     0     4     2     8    16
##  9  22.8     4  2.31    95  3.15  22.9     1     0     4     2     8    16
## 10  19.2     6  2.75   123  3.44  18.3     1     0     4     4    12    24
## # ... with 22 more rows
```

Add new variables with `transmute()`

`transmute` drops the existing variables

```
as_tibble(
  transmute(mtcars,
         cyl2 = 2 * cyl,
         cyl4 = 2 * cyl2,
         disp = disp * 0.0163871,
         drat = NULL)
)
```

```
## # A tibble: 32 x 3
##     cyl2  cyl4  disp
##    <dbl> <dbl> <dbl>
##  1    12    24  2.62
##  2    12    24  2.62
##  3     8    16  1.77
##  4    12    24  4.23
##  5    16    32  5.90
##  6    12    24  3.69
##  7    16    32  5.90
##  8     8    16  2.40
##  9     8    16  2.31
## 10    12    24  2.75
## # ... with 22 more rows
```

Create summary statistics with `summarise()`

Reduction is done by means of statistical functions, among which

- Center: `mean()`, `median()`
- Spread: `sd()`, `IQR()`, `mad()`
- Range: `min()`, `max()`, `quantile()`
- Position: `first()`, `last()`, `nth()`,
- Count: `n()`, `n_distinct()`
- Logical: `any()`, `all()`

Example

```
summarise(mtcars, Mean_mpg = mean(mpg),  Var_disp = var(disp))
```

```
##   Mean_mpg Var_disp
## 1 20.09062  15360.8
```

group rows according to factors with group_by()

group_by() does not do much visible expect creating a grouped data frame with type grouped_df

```
group_by(mtcars, cyl,am)
```

```
## # A tibble: 32 x 11
## # Groups:   cyl, am [6]
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

Remark

ungroup() performs the reverse operation.

Combine summarise() and group_by()

Magic of group_by() comes true when used in conjunction with summarise()

```
grp_mtcars <- group_by(mtcars, cyl, carb)
summarise(grp_mtcars, Count = n(),  Mean_mpg = mean(mpg),  Var_disp = var(disp))
```

```
## # A tibble: 9 x 5
## # Groups:   cyl [3]
##     cyl  carb Count Mean_mpg Var_disp
##   <dbl> <dbl> <int>    <dbl>    <dbl>
## 1     4     1     5     27.6    457.
## 2     4     2     6     25.9    732.
## 3     6     1     2     19.8    544.
## 4     6     4     4     19.8     19.3
## 5     6     6     1     19.7     NA
## 6     8     2     4     17.2   1886.
## 7     8     3     3     16.3      0
## 8     8     4     6     13.2   3341.
## 9     8     8     1     15       NA
```

Common remarks and extension

Remarks

Most primitive in `dyplr` do no modify the original table

Scoping

`rename`, `filter`, `select`, `summarise`, etc. all have *scoped* variant

- `rename_all()`: apply operation on all variables
- `rename_at()`: apply an operation on a subset of *specified* variables
- `rename_if()`:) apply an operation on the subset of *predicated* variables

Others verbs/functions

- `distinct()` subset unique rows
- `relocate()` change column position
- `top_n()`, `sample_n()` ... select rows
- `n()`, `if_else()` ... vector function for selection
- any function from `stringr` (`str_to_title`, `toupper`, etc.)

More on https://dplyr.tidyverse.org/reference/index.html

Exercises in dplyr vs base R

Exercises adapted from UseR 2017 on data. table

```
set.seed(20170703L)## toy data.frames
DF1 = data.frame(id = sample(1:2, 9L, TRUE),
                 code = sample(letters[1:3], 9, TRUE),
                 valA = 1:9, valB = 10:18,
                 stringsAsFactors = FALSE)
DF2 = data.frame(id = c(3L, 1L, 1L, 2L, 3L),
                 code = c("b", "a", "c", "c", "d"),
                 mul = 5:1, stringsAsFactors = FALSE)
## corresponding data tibble
TB1 <- as_tibble(DF1)
TB2 <- as_tibble(DF2)
```

Question 1

Subset all rows where id column equals 1 & code column is not equal to "c"

base

```
base::subset(DF1, id == 1 & code != "c")
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 4  1    a    4   13
```

```
with(DF1, DF1[id == 1 & code != "c",])
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 4  1    a    4   13
```

dplyr

```
filter(TB1, id == 1 & code != "c")
```

```
## # A tibble: 3 x 4
##      id code  valA  valB
##   <int> <chr> <int> <int>
## 1     1 a         1    10
## 2     1 a         2    11
## 3     1 a         4    13
```

Question 1

Subset all rows where id column equals 1 & code column is not equal to "c"

base

```
base::subset(DF1, id == 1 & code != "c")
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 4  1    a    4   13
```

```
with(DF1, DF1[id == 1 & code != "c",])
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 4  1    a    4   13
```

dplyr

```
filter(TB1, id == 1 & code != "c")
```

```
## # A tibble: 3 x 4
##      id code   valA  valB
##   <int> <chr> <int> <int>
## 1     1 a         1    10
## 2     1 a         2    11
## 3     1 a         4    13
```

Question 1

Subset all rows where id column equals 1 & code column is not equal to "c"

base

```
base::subset(DF1, id == 1 & code != "c")
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 4  1    a    4   13
```

```
with(DF1, DF1[id == 1 & code != "c",])
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 4  1    a    4   13
```

dplyr

```
filter(TB1, id == 1 & code != "c")
```

```
## # A tibble: 3 x 4
##      id code   valA  valB
##   <int> <chr> <int> <int>
## 1     1 a         1    10
## 2     1 a         2    11
## 3     1 a         4    13
```

Question 2

Select valA and valB columns from DF1

base R

```
DF1[, c("valA", "valB")]
```

```
##   valA valB
## 1    1   10
## 2    2   11
## 3    3   12
## 4    4   13
## 5    5   14
## 6    6   15
## 7    7   16
## 8    8   17
## 9    9   18
```

dplyr

```
select(TB1, valA, valB)
```

```
## # A tibble: 9 x 2
##    valA  valB
##   <int> <int>
## 1     1    10
## 2     2    11
## 3     3    12
## 4     4    13
## 5     5    14
## 6     6    15
```

Question 2

Select valA and valB columns from DF1

base R

```
DF1[, c("valA", "valB")]
```

```
##   valA valB
## 1    1   10
## 2    2   11
## 3    3   12
## 4    4   13
## 5    5   14
## 6    6   15
## 7    7   16
## 8    8   17
## 9    9   18
```

dplyr

```
select(TB1, valA, valB)
```

```
## # A tibble: 9 x 2
##    valA  valB
##   <int> <int>
## 1     1    10
## 2     2    11
## 3     3    12
## 4     4    13
## 5     5    14
## 6     6    15
```

Question 2

Select valA and valB columns from DF1

base R

```
DF1[, c("valA", "valB")]
```

```
##   valA valB
## 1    1   10
## 2    2   11
## 3    3   12
## 4    4   13
## 5    5   14
## 6    6   15
## 7    7   16
## 8    8   17
## 9    9   18
```

dplyr

```
select(TB1, valA, valB)
```

```
## # A tibble: 9 x 2
##    valA  valB
##   <int> <int>
## 1     1    10
## 2     2    11
## 3     3    12
## 4     4    13
## 5     5    14
## 6     6    15
```

Question 3

Get sum(valA) and sum(valB) for id $>$ 1 as a 1-row, 2-col data.frame

base R

```
colSums(DF1[ DF1$id > 1, c("valA", "valB")])
```

```
## valA valB
##   30   66
```

dplyr

```
TB1 %>% filter(id > 1) %>% select(valA, valB) %>% summarise_all(sum)
```

```
## # A tibble: 1 x 2
##    valA  valB
##   <int> <int>
## 1    30    66
```

Question 3

Get sum(valA) and sum(valB) for id > 1 as a 1-row, 2-col data.frame

base R

```
colSums(DF1[ DF1$id > 1, c("valA", "valB")])
```

```
## valA valB
##   30   66
```

dplyr

```
TB1 %>% filter(id > 1) %>% select(valA, valB) %>% summarise_all(sum)
```

```
## # A tibble: 1 x 2
##    valA  valB
##   <int> <int>
## 1    30    66
```

Question 3

Get sum(valA) and sum(valB) for id > 1 as a 1-row, 2-col data.frame

base R

```
colSums(DF1[ DF1$id > 1, c("valA", "valB")])
```

```
## valA valB
##   30   66
```

dplyr

```
TB1 %>% filter(id > 1) %>% select(valA, valB) %>% summarise_all(sum)
```

```
## # A tibble: 1 x 2
##    valA  valB
##   <int> <int>
## 1    30    66
```

Question 4

Replace valB with valB+1 for all rows where code == "c"

base R

```
DF1$valB[DF1$code == "c"] = DF1$valB[DF1$code == "c"] + 1
DF1
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 3  1    c    3   13
## 4  1    a    4   13
## 5  1    c    5   15
## 6  2    a    6   15
## 7  2    a    7   16
## 8  2    c    8   18
## 9  2    b    9   18
```

dplyr

```
mutate(TB1, valB = ifelse(code == "c", valB + 1,valB))
```

```
## # A tibble: 9 x 4
##      id code   valA  valB
##   <int> <chr> <int> <dbl>
## 1     1 a         1    10
## 2     1 a         2    11
## 3     1 c         3    13
## 4     1 a         4    13
## 5     1 c         5    15
```

Question 4

Replace valB with valB+1 for all rows where code == "c"

base R

```
DF1$valB[DF1$code == "c"] = DF1$valB[DF1$code == "c"] + 1
DF1
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 3  1    c    3   13
## 4  1    a    4   13
## 5  1    c    5   15
## 6  2    a    6   15
## 7  2    a    7   16
## 8  2    c    8   18
## 9  2    b    9   18
```

dplyr

```
mutate(TB1, valB = ifelse(code == "c", valB + 1,valB))
```

```
## # A tibble: 9 x 4
##      id code   valA  valB
##   <int> <chr> <int> <dbl>
## 1     1 a         1    10
## 2     1 a         2    11
## 3     1 c         3    13
## 4     1 a         4    13
## 5     1 c         5    15
```

Question 4

Replace valB with valB+1 for all rows where code == "c"

base R

```
DF1$valB[DF1$code == "c"] = DF1$valB[DF1$code == "c"] + 1
DF1
```

```
##   id code valA valB
## 1  1    a    1   10
## 2  1    a    2   11
## 3  1    c    3   13
## 4  1    a    4   13
## 5  1    c    5   15
## 6  2    a    6   15
## 7  2    a    7   16
## 8  2    c    8   18
## 9  2    b    9   18
```

dplyr

```
mutate(TB1, valB = ifelse(code == "c", valB + 1,valB))
```

```
## # A tibble: 9 x 4
##      id code   valA  valB
##   <int> <chr> <int> <dbl>
## 1     1 a         1    10
## 2     1 a         2    11
## 3     1 c         3    13
## 4     1 a         4    13
## 5     1 c         5    15
```

Question 5

Add a new column valC column with values equal to valB^2 - valA^2

base R

```
DF1 <- transform(DF1, valC = valB^2 - valA^2)
## DF1$valC <- DF1$valB^2 - DF1$valA^2 # alternate solution
DF1
```

```
##   id code valA valB valC
## 1  1    a    1   10   99
## 2  1    a    2   11  117
## 3  1    c    3   13  160
## 4  1    a    4   13  153
## 5  1    c    5   15  200
## 6  2    a    6   15  189
## 7  2    a    7   16  207
## 8  2    c    8   18  260
## 9  2    b    9   18  243
```

dplyr

```
TB1 <- mutate(TB1, valC = valB^2 - valA^2)
TB1
```

```
## # A tibble: 9 x 5
##      id code   valA  valB  valC
##   <int> <chr> <int> <int> <dbl>
## 1     1 a         1    10    99
## 2     1 a         2    11   117
## 3     1 c         3    12   135
```

Question 5

Add a new column valC column with values equal to valB^2 - valA^2

base R

```
DF1 <- transform(DF1, valC = valB^2 - valA^2)
## DF1$valC <- DF1$valB^2 - DF1$valA^2 # alternate solution
DF1
```

```
##   id code valA valB valC
## 1  1    a    1   10   99
## 2  1    a    2   11  117
## 3  1    c    3   13  160
## 4  1    a    4   13  153
## 5  1    c    5   15  200
## 6  2    a    6   15  189
## 7  2    a    7   16  207
## 8  2    c    8   18  260
## 9  2    b    9   18  243
```

dplyr

```
TB1 <- mutate(TB1, valC = valB^2 - valA^2)
TB1
```

```
## # A tibble: 9 x 5
##      id code   valA  valB  valC
##   <int> <chr> <int> <int> <dbl>
## 1     1 a         1    10    99
## 2     1 a         2    11   117
## 3     1 c         3    12   135
```

Question 5

Add a new column valC column with values equal to valB^2 - valA^2

base R

```
DF1 <- transform(DF1, valC = valB^2 - valA^2)
## DF1$valC <- DF1$valB^2 - DF1$valA^2 # alternate solution
DF1
```

```
##   id code valA valB valC
## 1  1    a    1   10   99
## 2  1    a    2   11  117
## 3  1    c    3   13  160
## 4  1    a    4   13  153
## 5  1    c    5   15  200
## 6  2    a    6   15  189
## 7  2    a    7   16  207
## 8  2    c    8   18  260
## 9  2    b    9   18  243
```

dplyr

```
TB1 <- mutate(TB1, valC = valB^2 - valA^2)
TB1
```

```
## # A tibble: 9 x 5
##      id code   valA  valB  valC
##   <int> <chr> <int> <int> <dbl>
## 1     1 a         1    10    99
## 2     1 a         2    11   117
## 3     1 c         3    12   135
```

Question 6

Get sum(valA) and sum(valB) grouped by id and code (i.e., for each unique combination of id,code)

base

```
aggregate(. ~ id + code, DF1, sum)
```

```
##   id code valA valB valC
## 1  1    a    7   34  369
## 2  2    a   13   31  396
## 3  2    b    9   18  243
## 4  1    c    8   28  360
## 5  2    c    8   18  260
```

```
aggregate(DF1[, c("valA", "valB")], list(DF1$id, DF1$code), sum)
```

```
##   Group.1 Group.2 valA valB
## 1       1       a    7   34
## 2       2       a   13   31
## 3       2       b    9   18
## 4       1       c    8   28
## 5       2       c    8   18
```

dplyr

```
TB1 %>% group_by(id, code) %>% summarise_all(sum)
```

```
## # A tibble: 5 x 5
## # Groups:   id [2]
##   id    code  valA  valB  valC
```

Question 6

Get sum(valA) and sum(valB) grouped by id and code (i.e., for each unique combination of id,code)

base

```
aggregate(.~ id + code, DF1, sum)
```

```
##   id code valA valB valC
## 1  1    a    7   34  369
## 2  2    a   13   31  396
## 3  2    b    9   18  243
## 4  1    c    8   28  360
## 5  2    c    8   18  260
```

```
aggregate(DF1[, c("valA", "valB")], list(DF1$id, DF1$code), sum)
```

```
##   Group.1 Group.2 valA valB
## 1       1       a    7   34
## 2       2       a   13   31
## 3       2       b    9   18
## 4       1       c    8   28
## 5       2       c    8   18
```

dplyr

```
TB1 %>% group_by(id, code) %>% summarise_all(sum)
```

```
## # A tibble: 5 x 5
## # Groups:   id [2]
```

Question 6

Get sum(valA) and sum(valB) grouped by id and code (i.e., for each unique combination of id,code)

base

```
aggregate(.~ id + code, DF1, sum)
```

```
##   id code valA valB valC
## 1  1    a    7   34  369
## 2  2    a   13   31  396
## 3  2    b    9   18  243
## 4  1    c    8   28  360
## 5  2    c    8   18  260
```

```
aggregate(DF1[, c("valA", "valB")], list(DF1$id, DF1$code), sum)
```

```
##   Group.1 Group.2 valA valB
## 1       1       a    7   34
## 2       2       a   13   31
## 3       2       b    9   18
## 4       1       c    8   28
## 5       2       c    8   18
```

dplyr

```
TB1 %>% group_by(id, code) %>% summarise_all(sum)
```

```
## # A tibble: 5 x 5
## # Groups:   id [2]
```

Question 7

Get sum(valA) and sum(valB) grouped by id for id >= 2 & code %in% c("a", "c")

base

```
aggregate(.~ id , subset(DF1, id >=2 & code %in% c("a","c"), -code), sum)
```

```
##   id valA valB valC
## 1  2   21   49  656
```

dplyr

```
TB1 %>%
  group_by(id) %>%
  filter(id >=2, code %in% c("a", "c")) %>%
  select(-code, -valC) %>%
  summarise_all(sum)
```

```
## # A tibble: 1 x 3
##      id  valA  valB
##   <int> <int> <int>
## 1     2    21    48
```

Question 7

Get sum(valA) and sum(valB) grouped by id for id >= 2 & code %in% c("a", "c")

base

```
aggregate(.~ id , subset(DF1, id >=2 & code %in% c("a","c"), -code), sum)
```

```
##   id valA valB valC
## 1  2   21   49  656
```

dplyr

```
TB1 %>%
  group_by(id) %>%
  filter(id >=2, code %in% c("a", "c")) %>%
  select(-code, -valC) %>%
  summarise_all(sum)

## # A tibble: 1 x 3
##      id  valA  valB
##   <int> <int> <int>
## 1     2    21    48
```

Question 7

Get sum(valA) and sum(valB) grouped by id for id >= 2 & code %in% c("a", "c")

base

```
aggregate(.~ id , subset(DF1, id >=2 & code %in% c("a","c"), -code), sum)
```

```
##   id valA valB valC
## 1  2   21   49  656
```

dplyr

```
TB1 %>%
  group_by(id) %>%
  filter(id >=2, code %in% c("a", "c")) %>%
  select(-code, -valC) %>%
  summarise_all(sum)
```

```
## # A tibble: 1 x 3
##      id  valA  valB
##   <int> <int> <int>
## 1     2    21    48
```

Question 8

Replace valA with max(valA)-min(valA) grouped by code

base

```
DF1 <- transform(DF1, valA = rep(tapply(valA, code, function(x) diff(range(x)))[code]))
DF1
```

```
##   id code valA valB valC
## 1  1    a    6   10   99
## 2  1    a    6   11  117
## 3  1    c    5   13  160
## 4  1    a    6   13  153
## 5  1    c    5   15  200
## 6  2    a    6   15  189
## 7  2    a    6   16  207
## 8  2    c    5   18  260
## 9  2    b    0   18  243
```

dplyr

```
TB1 <- TB1 %>% group_by(code) %>% mutate(valA= max(valA)-min(valA))
TB1
```

```
## # A tibble: 9 x 5
## # Groups:   code [3]
##      id code   valA  valB  valC
##   <int> <chr> <int> <int> <dbl>
## 1     1 a         6    10    99
## 2     1 a         6    11   117
## 3     1 c         5    12   135
```

Question 8

Replace valA with max(valA)-min(valA) grouped by code

base

```
DF1 <- transform(DF1, valA = rep(tapply(valA, code, function(x) diff(range(x)))[code]))
DF1
```

```
##   id code valA valB valC
## 1  1    a    6   10   99
## 2  1    a    6   11  117
## 3  1    c    5   13  160
## 4  1    a    6   13  153
## 5  1    c    5   15  200
## 6  2    a    6   15  189
## 7  2    a    6   16  207
## 8  2    c    5   18  260
## 9  2    b    0   18  243
```

dplyr

```
TB1 <- TB1 %>% group_by(code) %>% mutate(valA= max(valA)-min(valA))
TB1
```

```
## # A tibble: 9 x 5
## # Groups:   code [3]
##      id code   valA  valB  valC
##   <int> <chr> <int> <int> <dbl>
## 1     1 a         6    10    99
## 2     1 a         6    11   117
## 3     1 c         5    12   135
```

Question 8

Replace valA with max(valA)-min(valA) grouped by code

base

```
DF1 <- transform(DF1, valA = rep(tapply(valA, code, function(x) diff(range(x)))[code]))
DF1
```

```
##   id code valA valB valC
## 1  1    a    6   10   99
## 2  1    a    6   11  117
## 3  1    c    5   13  160
## 4  1    a    6   13  153
## 5  1    c    5   15  200
## 6  2    a    6   15  189
## 7  2    a    6   16  207
## 8  2    c    5   18  260
## 9  2    b    0   18  243
```

dplyr

```
TB1 <- TB1 %>% group_by(code) %>% mutate(valA= max(valA)-min(valA))
TB1
```

```
## # A tibble: 9 x 5
## # Groups:   code [3]
##      id code   valA  valB  valC
##   <int> <chr> <int> <int> <dbl>
## 1     1 a         6    10    99
## 2     1 a         6    11   117
## 3     1 c         5    12   135
```

Question 9

Create a new col named valD with max(valB)-min(valA) grouped by code

base

```
DF1 <- transform(DF1, valD = by(DF1, code, function(x) max(x$valB) - min(x$valA))[code])
DF1
```

```
##   id code valA valB valC valD
## 1  1    a    6   10   99   10
## 2  1    a    6   11  117   10
## 3  1    c    5   13  160   13
## 4  1    a    6   13  153   10
## 5  1    c    5   15  200   13
## 6  2    a    6   15  189   10
## 7  2    a    6   16  207   10
## 8  2    c    5   18  260   13
## 9  2    b    0   18  243   18
```

dplyr

```
TB1 <- TB1 %>% group_by(code) %>% mutate(valD= max(valB)-min(valA))
TB1
```

```
## # A tibble: 9 x 6
## # Groups:   code [3]
##      id code   valA  valB  valC  valD
##   <int> <chr> <int> <int> <dbl> <int>
## 1     1 a         6    10    99    10
## 2     1 a         6    11   117    10
## 3     1 c         5    12   135    12
```

Question 9

Create a new col named valD with max(valB)-min(valA) grouped by code

base

```
DF1 <- transform(DF1, valD = by(DF1, code, function(x) max(x$valB) - min(x$valA))[code])
DF1
```

```
##   id code valA valB valC valD
## 1  1    a    6   10   99   10
## 2  1    a    6   11  117   10
## 3  1    c    5   13  160   13
## 4  1    a    6   13  153   10
## 5  1    c    5   15  200   13
## 6  2    a    6   15  189   10
## 7  2    a    6   16  207   10
## 8  2    c    5   18  260   13
## 9  2    b    0   18  243   18
```

dplyr

```
TB1 <- TB1 %>% group_by(code) %>% mutate(valD= max(valB)-min(valA))
TB1
```

```
## # A tibble: 9 x 6
## # Groups:   code [3]
##      id code   valA  valB  valC  valD
##   <int> <chr> <int> <int> <dbl> <int>
## 1     1 a         6    10    99    10
## 2     1 a         6    11   117    10
## 3     1 c         5    12   135    12
```

Question 9

Create a new col named valD with max(valB)-min(valA) grouped by code

base

```
DF1 <- transform(DF1, valD = by(DF1, code, function(x) max(x$valB) - min(x$valA))[code])
DF1
```

```
##   id code valA valB valC valD
## 1  1    a    6   10   99   10
## 2  1    a    6   11  117   10
## 3  1    c    5   13  160   13
## 4  1    a    6   13  153   10
## 5  1    c    5   15  200   13
## 6  2    a    6   15  189   10
## 7  2    a    6   16  207   10
## 8  2    c    5   18  260   13
## 9  2    b    0   18  243   18
```

dplyr

```
TB1 <- TB1 %>% group_by(code) %>% mutate(valD= max(valB)-min(valA))
TB1
```

```
## # A tibble: 9 x 6
## # Groups:   code [3]
##      id code   valA  valB  valC  valD
##   <int> <chr> <int> <int> <dbl> <int>
## 1     1 a         6    10    99    10
## 2     1 a         6    11   117    10
## 3     1 c         5    12   135    12
```

Outline

{magrittr}

Figure 12: a set of operators which make your code more readable

```
library(magrittr)
```

Provides the following operators

- Pipe %>%
- Reassignment pipe %<>%
- T-Pipe %T>%

Motivation: make Tom eat an apple

Everyday language

Tom eats an apple

Subject - Verb - Complement

Programming language

```
eat(Tom, apple)
```

Verb - Subject - Complement

Pipes

$\rightsquigarrow$ get closer to everyday language in your code
$\rightsquigarrow$ clearly expressing a sequence of multiple operations

Pipe %>%

- when you read code, %>% is pronounced "then"
- the keybord shortcut for %>% is Ctrl + shift + M

Objective

- Helps writing `R` code which is easy to read (and thus, easy to understand)
- `x %>% f()` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`
- `x %>% f(y,.)` is équivalent to `f(y,x)`

Example

```
2^mean(log(seq_len(10), base = 2), na.rm = TRUE)
```

```
## [1] 4.528729
```

```
10 %>%
  seq_len() %>%
  log(base = 2) %>%
  mean(na.rm = TRUE) %>%
  {2^.}
```

```
## [1] 4.528729
```

Pipe %>%

- when you read code, %>% is pronounced "then"
- the keybord shortcut for %>% is Ctrl + shift + M

Objective

- Helps writing `R` code which is easy to read (and thus, easy to understand)
- `x %>% f()` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`
- `x %>% f(y,.)` is équivalent to `f(y,x)`

Example

```
2^mean(log(seq_len(10), base = 2), na.rm = TRUE)
```

```
## [1] 4.528729
```

```
10 %>%
  seq_len() %>%
  log(base = 2) %>%
  mean(na.rm = TRUE) %>%
  {2^.}
```

```
## [1] 4.528729
```

Pipe %>%

- when you read code, %>% is pronounced "then"
- the keybord shortcut for %>% is Ctrl + shift + M

Objective

- Helps writing `R` code which is easy to read (and thus, easy to understand)
- `x %>% f()` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`
- `x %>% f(y,.)` is équivalent to `f(y,x)`

Example

```
2^mean(log(seq_len(10), base = 2), na.rm = TRUE)
```

```
## [1] 4.528729
```

```
10 %>%
  seq_len() %>%
  log(base = 2) %>%
  mean(na.rm = TRUE) %>%
  {2^.}
```

```
## [1] 4.528729
```

Exercise

Consider

```
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
```

Compute the logarithm of `x`, return suitably lagged and iterated differences, compute the exponential function and round the result

1. In base `R`
2. Using %>%

(Re)assignment pipe %<>%

For affectation, `magrittr` provides the operator %<>% which allows to replace code like

```
mtcars <- mtcars%>% transform(cyl = cyl * 2)
```

by

```
mtcars %<>% transform(cyl = cyl * 2)
```

T-pipe %T>%

Problem with functions requiring early side effects along succession of %>%

- you might want to plot or print and object
- such function do not send back anything and break the pipe

Solution

- to overcome such an issue, use the "tee" pipe %T>%
- works like %>% except that it sends left side in place of right side of the expression
- "tee"because it looks like a pipe with a T shape

T-pipe %T>%: example without T

```
rnorm (100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
```

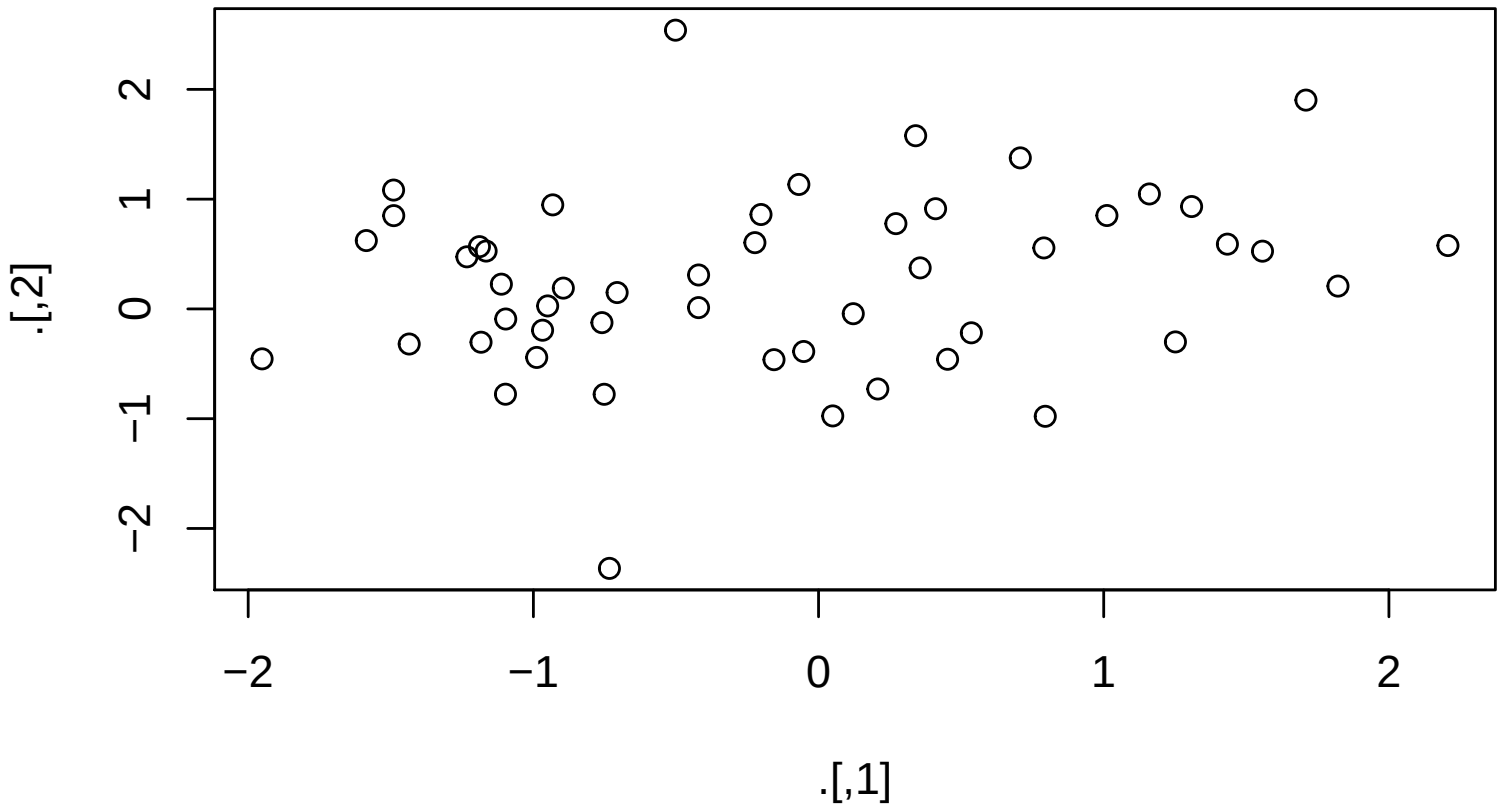


Figure 13: plot of bivariate Gaussian sample

```
##  NULL
```

T-pipe %T>%: example with T

```
rnorm (100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
```

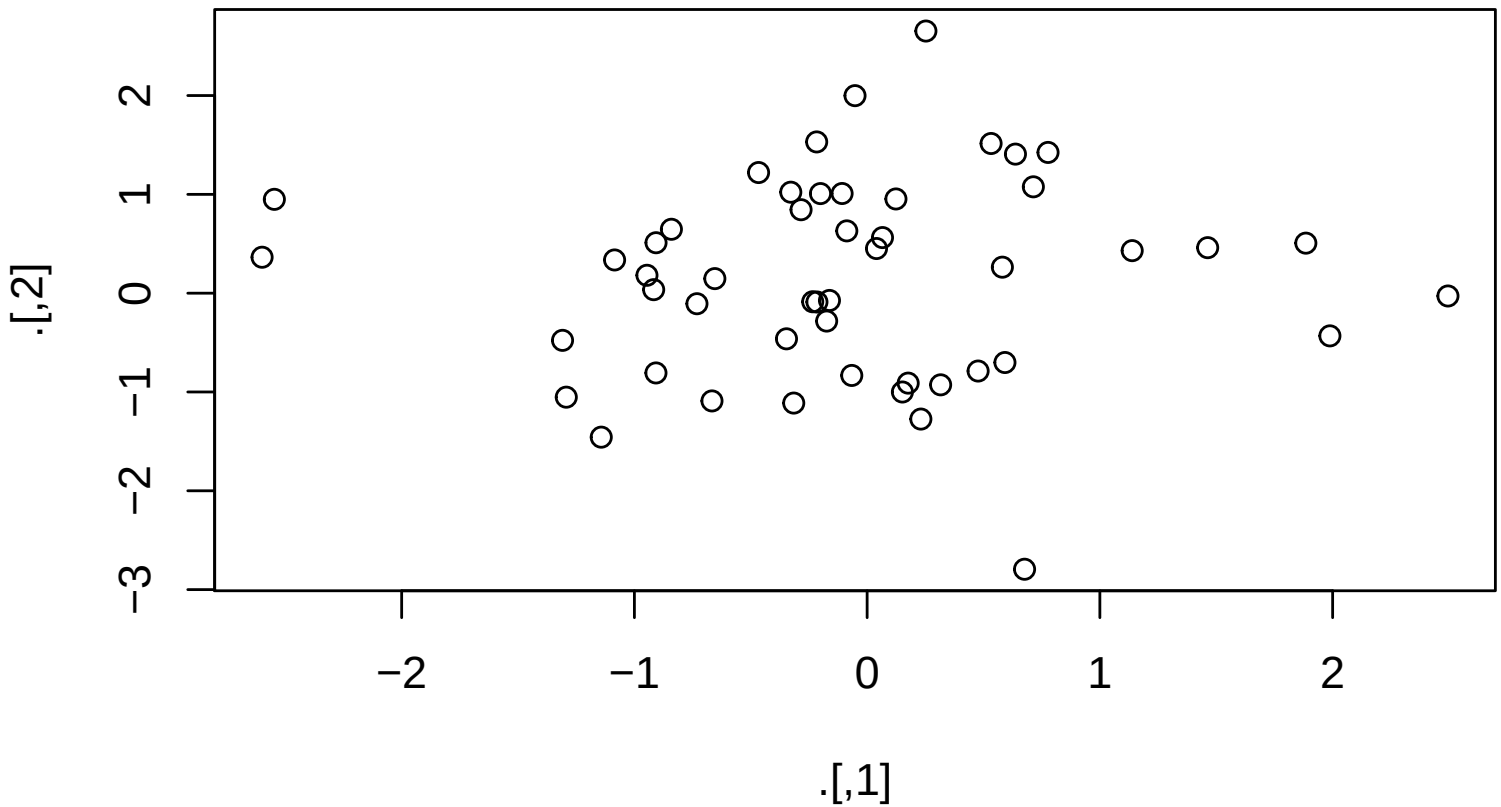


Figure 14: plot of bivariate Gaussian sample

```
##  num [1:50, 1:2] -0.216 1.463 -0.667 -2.6 -0.347 ...
```

Exposition Operator %$%

When working with functions that do not take data argumentbut still useful in a pipeline, e.g., when your data is first processed and then passed into the function.

Example

```
iris %>%
  subset(Sepal.Length > mean(Sepal.Length)) %$%
  cor(Sepal.Length, Sepal.Width)
```

```
## [1] 0.3361992
```

When not to use the pipe

Consider other solutions when

Pipes contain too many steps

Create *intermediate* objects with meaningful names

Multiple inputs or outputs are required

E.g., when several objects need to *combine* together

Complex dependance structures exists between your entries

Pipes are fundamentally *linear*: expressing complex relationships with them yield confusing code.

Functional programming: general idea

Definition

R is *a Functional programming language* with *first class functions*: it places functions at the same level as variables. One can

1. *pass functions as arguments* to other functions
2. *return functions* from other functions
3. *store functions in data structures*

Ingredients

In R the tools/ingredients allowing functional programming are

- *closure*, i.e. functions that output functions
- *list*, in which you can store functions
- *anonymous functions*, i.e. function that does not deserve a name
- *functionals*, i.e. functions that take function(s) as input

$\rightsquigarrow$ we briefly review some of R's functional tools, which you probably already use

Functional programming: general idea

Definition

R is *a Functional programming language* with *first class functions*: it places functions at the same level as variables. One can

1. *pass functions as arguments* to other functions
2. *return functions* from other functions
3. *store functions in data structures*

Ingredients

In R the tools/ingredients allowing functional programming are

- *closure*, i.e. functions that output functions
- *list*, in which you can store functions
- *anonymous functions*, i.e. function that does not deserve a name
- *functionals*, i.e. functions that take function(s) as input

$\rightsquigarrow$ we briefly review some of R's functional tools, which you probably already use

Figure 15: enhances R's functional programming (FP) toolkit

Some slides borrowed to - Happy dev with {purrr}
https://colinfay.me/happy-dev-purrr/ (C. Fay) - Modeling in the tidyverse
https://abichat.github.io/slides/modelingtidyverse/ (A. Bichat)

Functional programming "phylosophy"

- Never use `for` loops

Base R

`*apply()` functions and friends for lists and data frames

- `apply`, `lapply()`, `sapply()`, `tapply()`, `mapply()`, `...`
- `Map()`, `Reduce()`
- `Vectorize()`

Tidyverse variants

`map()` function and variants

- `map()`, `map_dbl()`, `map_chr()`, `map_lgl()`, `...`
- `map2()`, `map2_dbl()`, `map2_chr()`, `map2_lgl()`, `...`
- `pmap()`, `pmap_dbl()`, `pmap_chr()`, `pmap_lgl()`, `...`
- `reduce()`

map*()

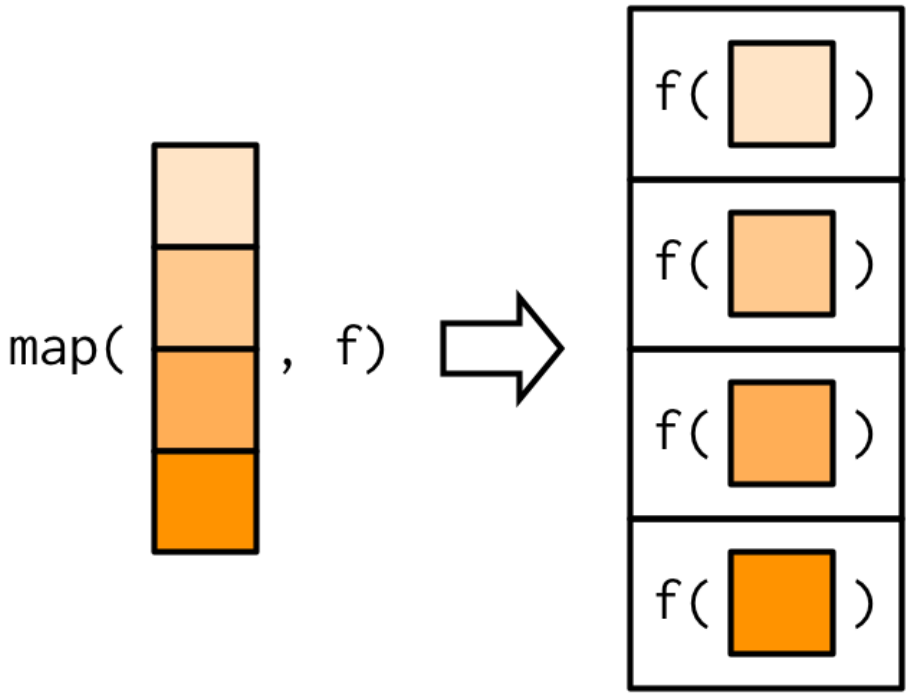


Figure 16: map

- `map()` applies the function to each element and returns a list
- `map_lgl()`, `map_int()`, `map_dbl()` and `map_chr()` applies the function to each element and returns a atomic vector of the corresponding type

map*(): examples I

```
map(1:5, rnorm)
```

```
## [[1]]
## [1] 0.6786101
##
## [[2]]
## [1] -0.8344392 -1.0176957
##
## [[3]]
## [1] -0.4142210  1.6200032 -0.5265256
##
## [[4]]
## [1] -0.1924844  1.0187318  0.1121379  1.4153514
##
## [[5]]
## [1] -0.4068349  1.0349484  1.4408870 -1.1577407  0.6536112
```

```
map(1:5, rnorm) %>%
  map_dbl(mean)
```

```
## [1] -0.61699888 -0.00437652  0.18621952 -0.60010996  0.16600898
```

map*() (additional arguments)

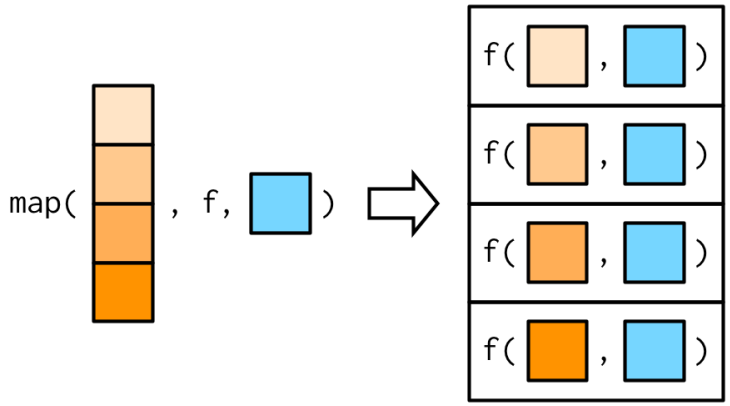


Figure 17: map

```
map(1:4, rnorm, n = 6)
```

```
## [[1]]
## [1] 2.3709584 0.4353018 1.3631284 1.6328626 1.4042683 0.8938755
##
## [[2]]
## [1] 3.511522 1.905341 4.018424 1.937286 3.304870 4.286645
##
## [[3]]
## [1] 1.6111393 2.7212112 2.8666787 3.6359504 2.7157471 0.3435446
##
## [[4]]
## [1] 1.559533 5.320113 3.693361 2.218692 3.828083 5.214675
```

map*()

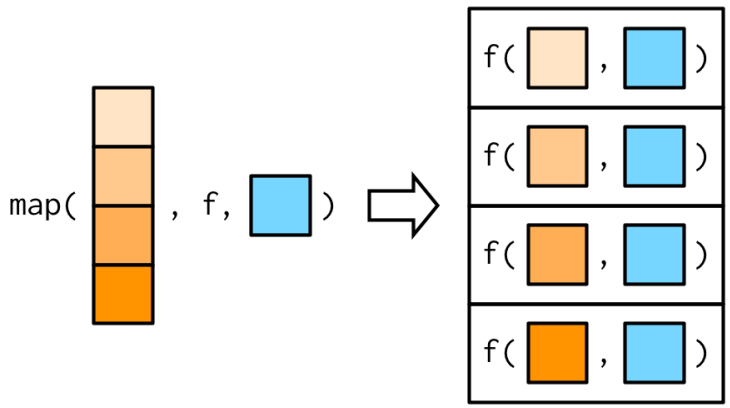


Figure 18: map

```
map(1:4, rnorm, n = 6) %>%
  map_dbl(function(x) x[2])
```

```
## [1] 0.4353018 1.9053410 2.7212112 5.3201133
```

map*()

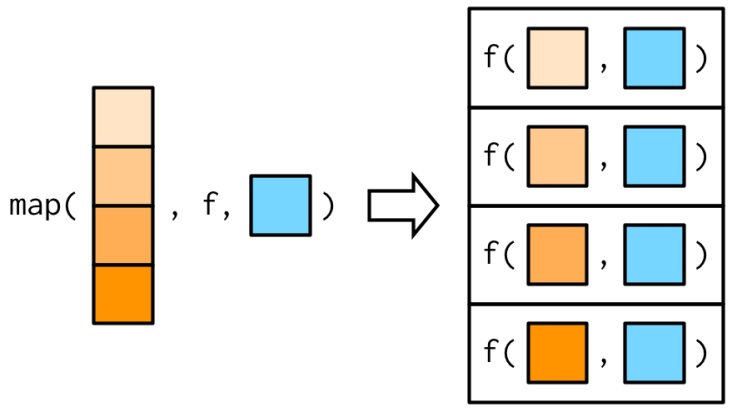


Figure 19: map

```
map(1:4, rnorm, n = 6) %>%
  map_dbl(~ .[2])
```

```
## [1] 0.4353018 1.9053410 2.7212112 5.3201133
```

map*()

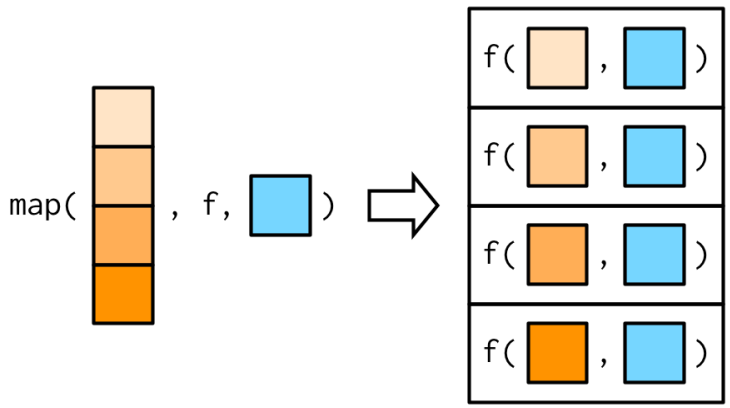


Figure 20: map

```
map(1:4, rnorm, n = 6) %>%
  map_dbl(2)
```

```
## [1] 0.4353018 1.9053410 2.7212112 5.3201133
```

Examples

Split a data frame into pieces, fit a model to each piece, compute the summary, then extract the R2.

```
mtcars %>%
  split(.$cyl) %>% # from base R
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

A more complicated example

```
iris %>%
  group_by(Species) %>%
  nest() %>%
  mutate(model = map(data, ~ lm(data = ., Sepal.Length ~ Petal.Length))) %>%
  mutate(Summary = map(model, summary)) %>%
  mutate(`R squared` = map_dbl(Summary, ~ .$r.squared))
```

```
## # A tibble: 3 x 5
## # Groups:   Species [3]
##   Species    data              model  Summary    `R squared`
##   <fct>      <list>            <list> <list>           <dbl>
## 1 setosa     <tibble [50 x 4]> <lm>   <smmry.lm>      0.0714
## 2 versicolor <tibble [50 x 4]> <lm>   <smmry.lm>      0.569
## 3 virginica  <tibble [50 x 4]> <lm>   <smmry.lm>      0.747
```

Advantages of `map()` vs the `apply()` family

- Stable and consistent grammar

```
apply(X, MARGIN, FUN, ...)
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)
...
```

VS

```
map(.x, .f, ...)
map_if(.x, .p, .f, ...)
map_chr(.x, .f, ...)
map_int(.x, .f, ...)
map_dbl(.x, .f, ...)
...
```

Advantages of `map()` vs the `apply()` family

- Type stability

```
sapply(iris$Sepal.Length, as.numeric) %>% class()
```

```
## [1] "numeric"
```

```
sapply(iris$Sepal.Length, as.data.frame) %>% class()
```

```
## [1] "list"
```

VS

```
map_dbl(iris$Sepal.Length, as.numeric) %>% class()
```

```
## [1] "numeric"
```

```
map_df(iris$Sepal.Length, as.data.frame) %>% class()
```

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

Advantages of `map()` vs the `apply()` family

- Anonymous functions & verbosity

```
lapply(list, function(x) x + 2)
mapply(function(x, y) x + y, list1, list2)
lapply(list, function(x) x[[2]])
lapply(list, function(x) x$foo)
```

VS

```
map(list, ~ . + 2)
map2(list1, list2, ~ .x + .y)
map(list, 2)
map(list, "foo")
```

map2*()

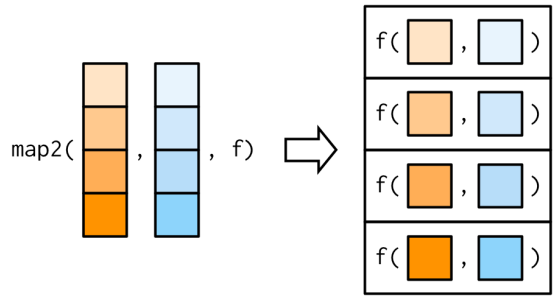


Figure 21: map2

map2() and map2_*() are variants of map() and map_*() which work with two arguments

```
map2(1:4, c(2, 5, 5, 10), runif, n = 5)
```

```
## [[1]]
## [1] 1.970967 1.618838 1.333427 1.346748 1.398485
##
## [[2]]
## [1] 4.354078 2.116809 4.246386 4.031830 2.513793
##
## [[3]]
## [1] 3.522176 4.028826 4.351215 4.965634 4.519089
##
## [[4]]
## [1] 7.398931 9.098138 5.136844 5.627720 8.968951
```

pmap*()

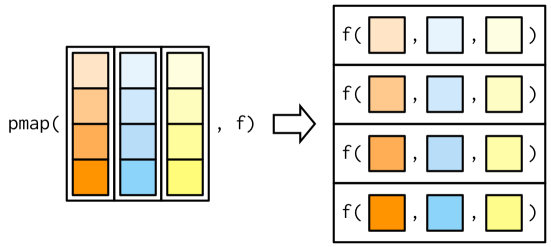


Figure 22: pmap

pmap() and pmap_*() are generalized versions of map() and map_*() which work with any number of arguments

```
pmap(list(n = c(2, 3, 2, 5), min = 1:4, max = c(2, 5, 5, 10)), runif)
```

```
## [[1]]
## [1] 1.693205 1.240545
##
## [[2]]
## [1] 2.128966 2.421437 2.649156
##
## [[3]]
## [1] 3.958797 3.394821
##
## [[4]]
## [1] 8.316135 4.047308 6.252940 7.086446 4.009423
```

reduce()

```
rerun(4, sample(1:10, 6))
```

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

reduce()

```
rerun(4, sample(1:10, 6))
```

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

```
rerun(4, sample(1:10, 6)) %>%
  reduce(intersect)
```

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

Outline

ggplot2

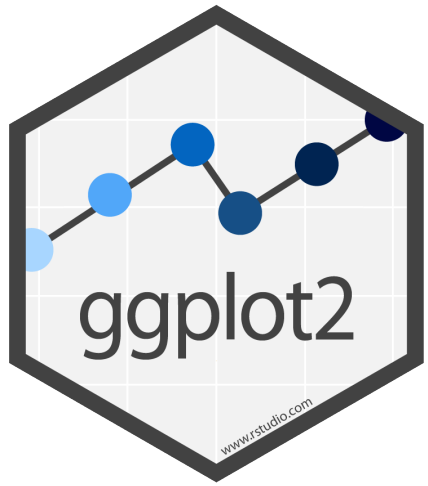
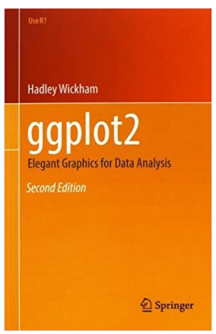


Figure 23: a system for declaratively creating graphics, based on The Grammar of Graphics

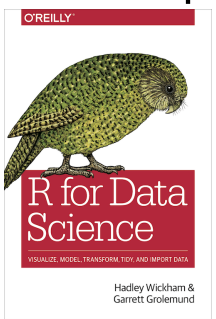
Fully documented (Wickham, 2016) http://ggplot2.tidyverse.org/



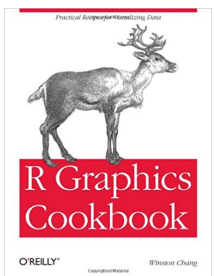
Learning ggplot2

R for data science (Wickham & Grolemund, 2017), http://r4ds.had.co.nz

See chapters *data visualisation* and *graphics for communication*



R for data science (Chang, 2012), http://www.cookbook-r.com/Graphs/

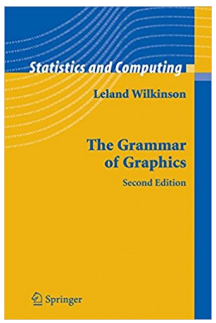


This course

A short introduction, mostly based on examples

ggplot2: grammar of graphics

Implements the grammar of graphics (Wilkinson, 2006)



Elements that composes a the grammar of ggplot

- a data set (*data*),
- a graphical projection/mapping (*aes*),
- a geometrical representation (*geom*),
- a statistical transformation (*stats*),
- a coordinate system (coord),
- some scales (*scale*),
- some groupings (*facet*).

Grammar of Graphics in ggplot: summary

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
                  stat = <STAT>, position = <POSITION>) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

⇝ any plot can be described by a combination of these 7 parameters.

ggplot2: standard steps

Supply data and specify mapping

with functions `ggplot()` and `aes()`

Create a layer

Combine data, mapping, a geometric object, a stat (statistical transformation) and a position adjustment

- by using `geom()` (overide the statistical transformation and position)
- by using `stat()` (specifying a statistical transformation with `stat`)
- add layers to the current ggplot object with the + operator

Adjustements

- the position (`position_`)
- the coordinate system (`coord_`)
- some annotations (`annotation_`)
- faceting (`facet_`)

Example: good old `iris` data set

```
as_tibble(iris)
```

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

Initializing the plot object

Supply data and mapping

All layers use a common data set and common set of aesthetics

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length))
```

Supply data

All layers use a common data set, but with specific aesthetics

```
ggplot(data = iris)
```

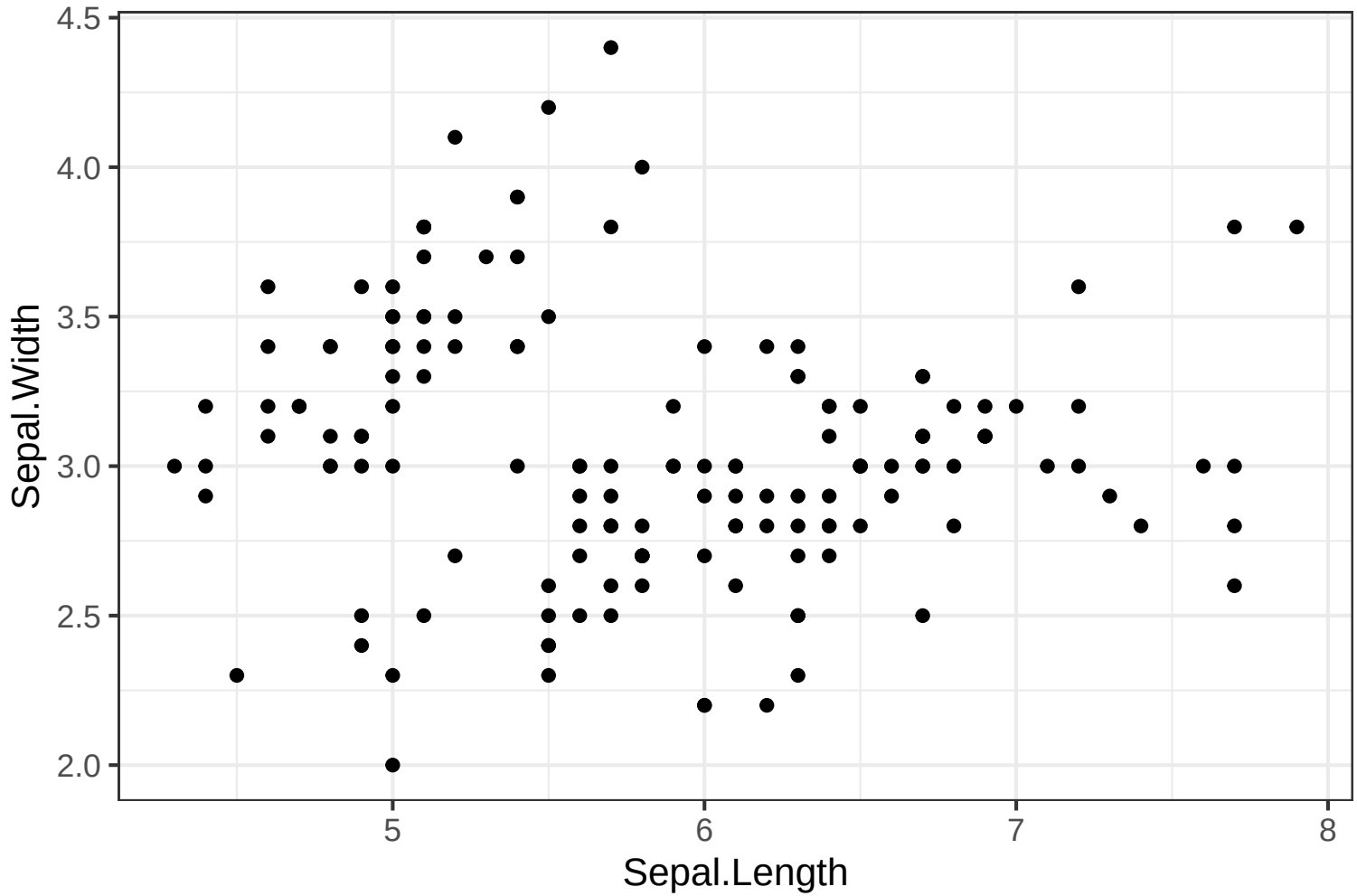
Simple initialization

Each layer use a specific data set

```
ggplot()
```

Add a layer: scatterplot

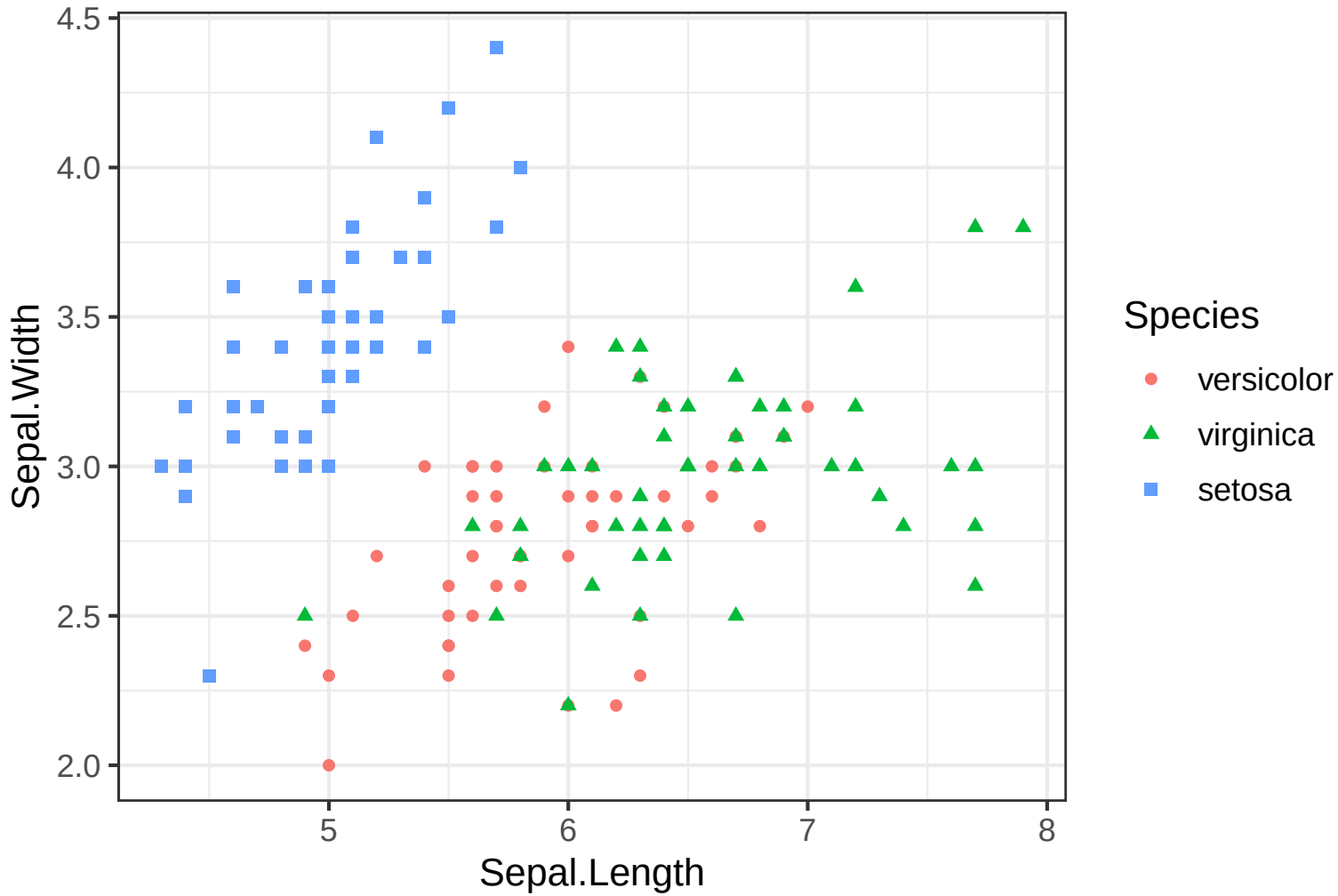
```
ggplot(iris) + geom_point(aes(x = Sepal.Length, y = Sepal.Width))
```



Add a layer: scatterplot + annotation

some aesthetic are optional

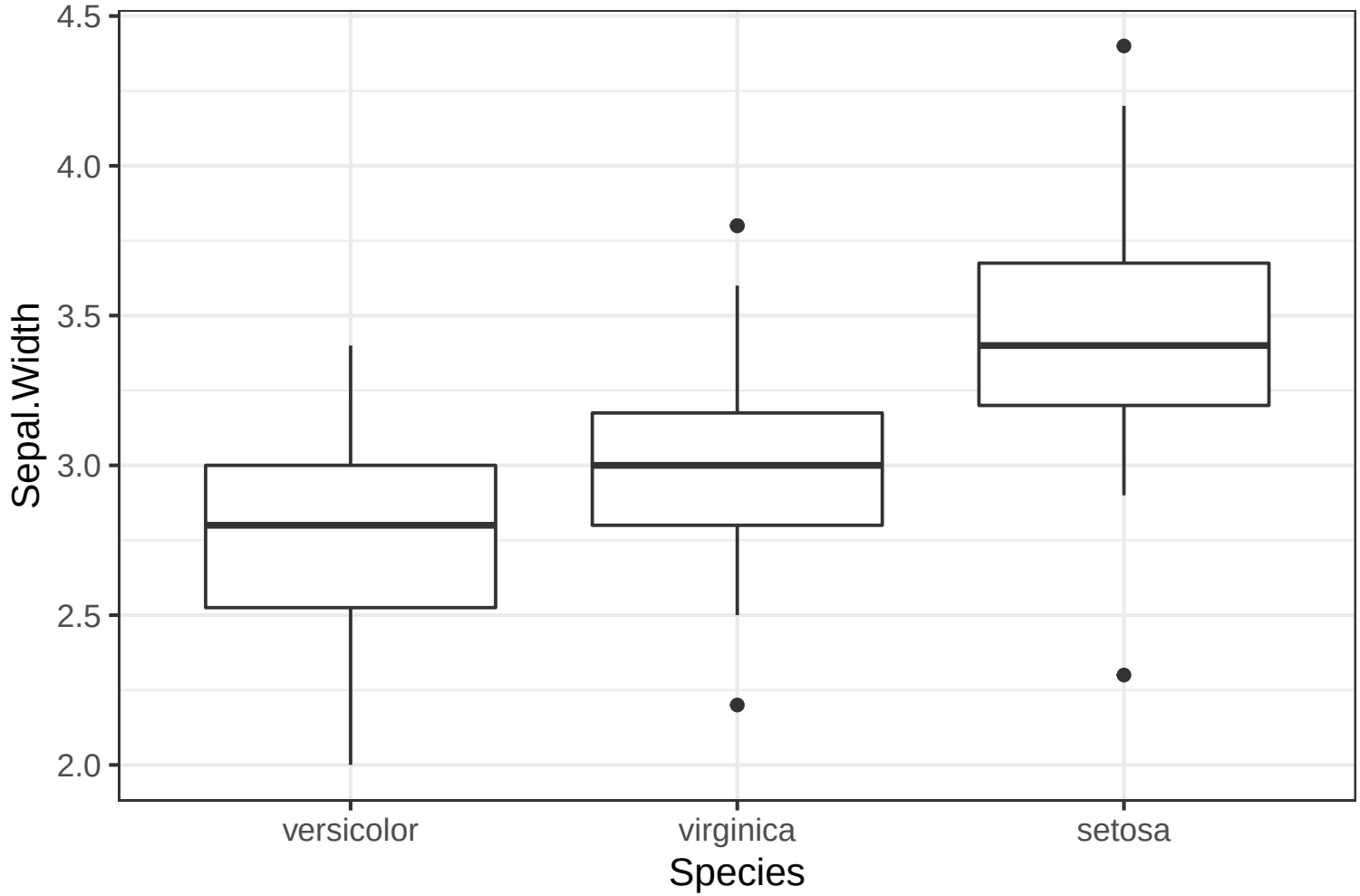
```
ggplot(iris) + geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species, shape = Speci
```



Add a layer: boxplot

the aes depends on the geometry (here, a factor is expected for `x`)

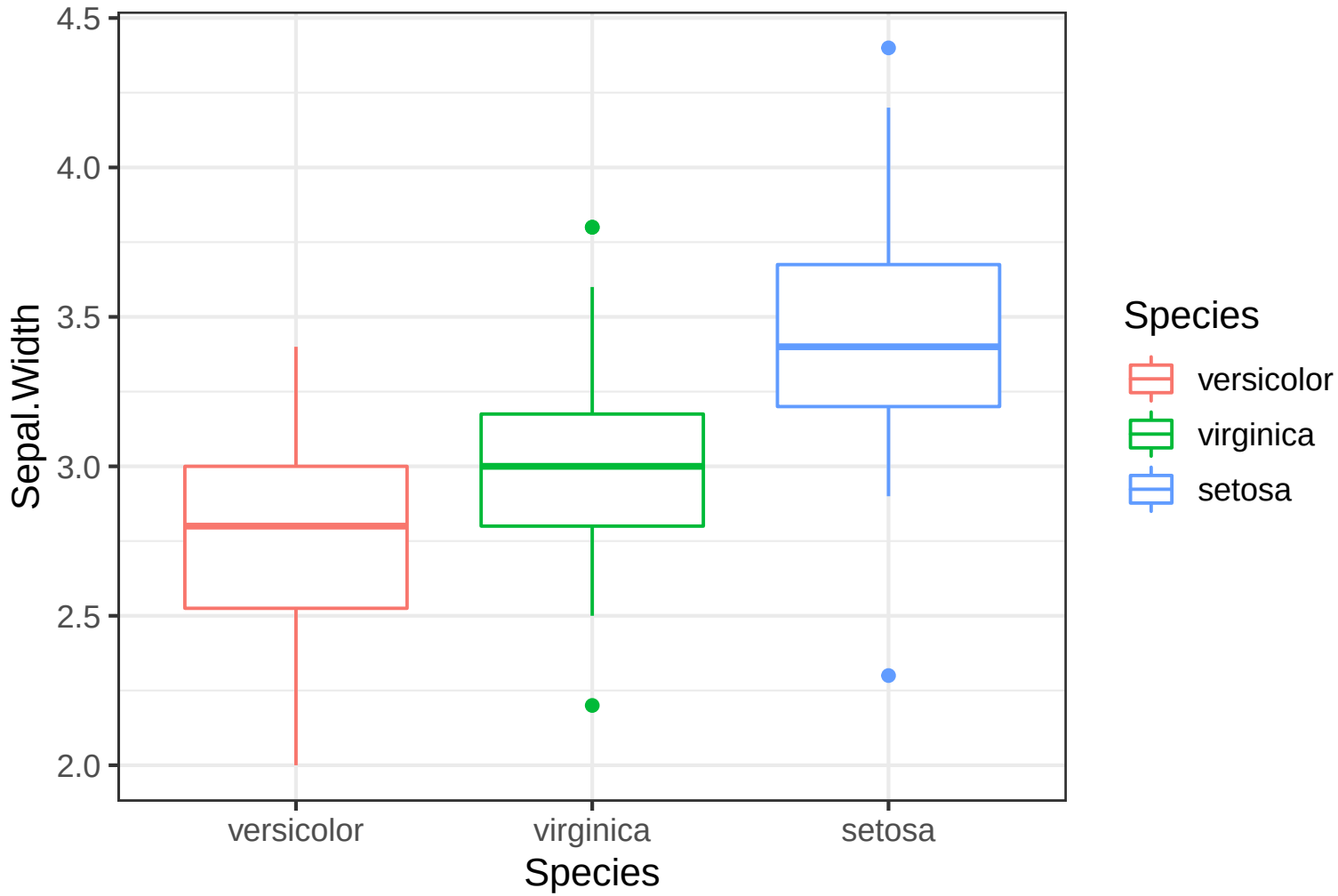
```
ggplot(iris) + geom_boxplot(aes(x = Species, y = Sepal.Width))
```



Add a layer: boxplot + annotation

the aes depends on the geometry (here, a factor is expected for `x`)

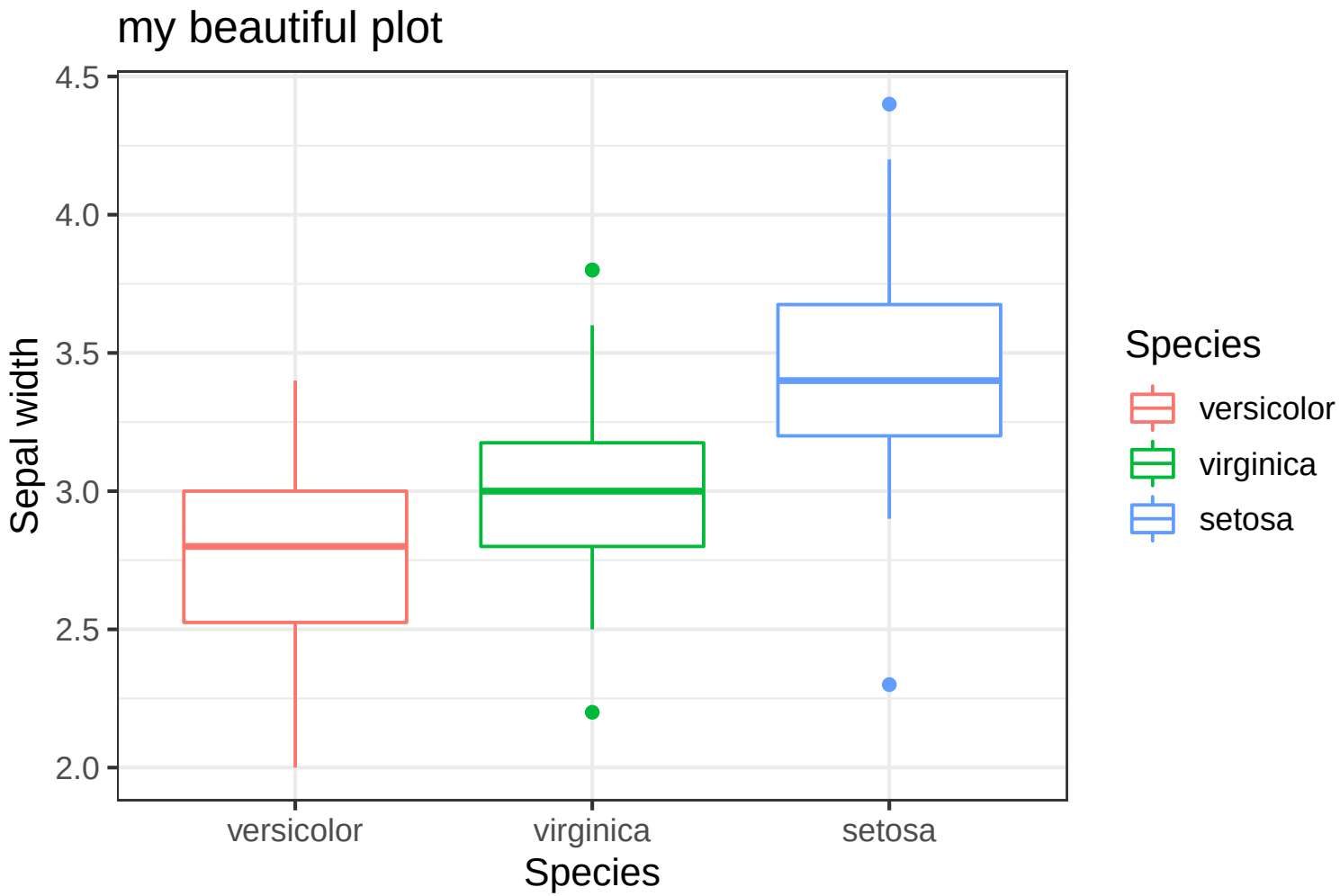
```
ggplot(iris) + geom_boxplot(aes(x = Species, y = Sepal.Width, color = Species))
```



Add a layer: boxplot + annotation (Cont'd)

the aes depends on the geometry (here, a factor is expected for `x`)

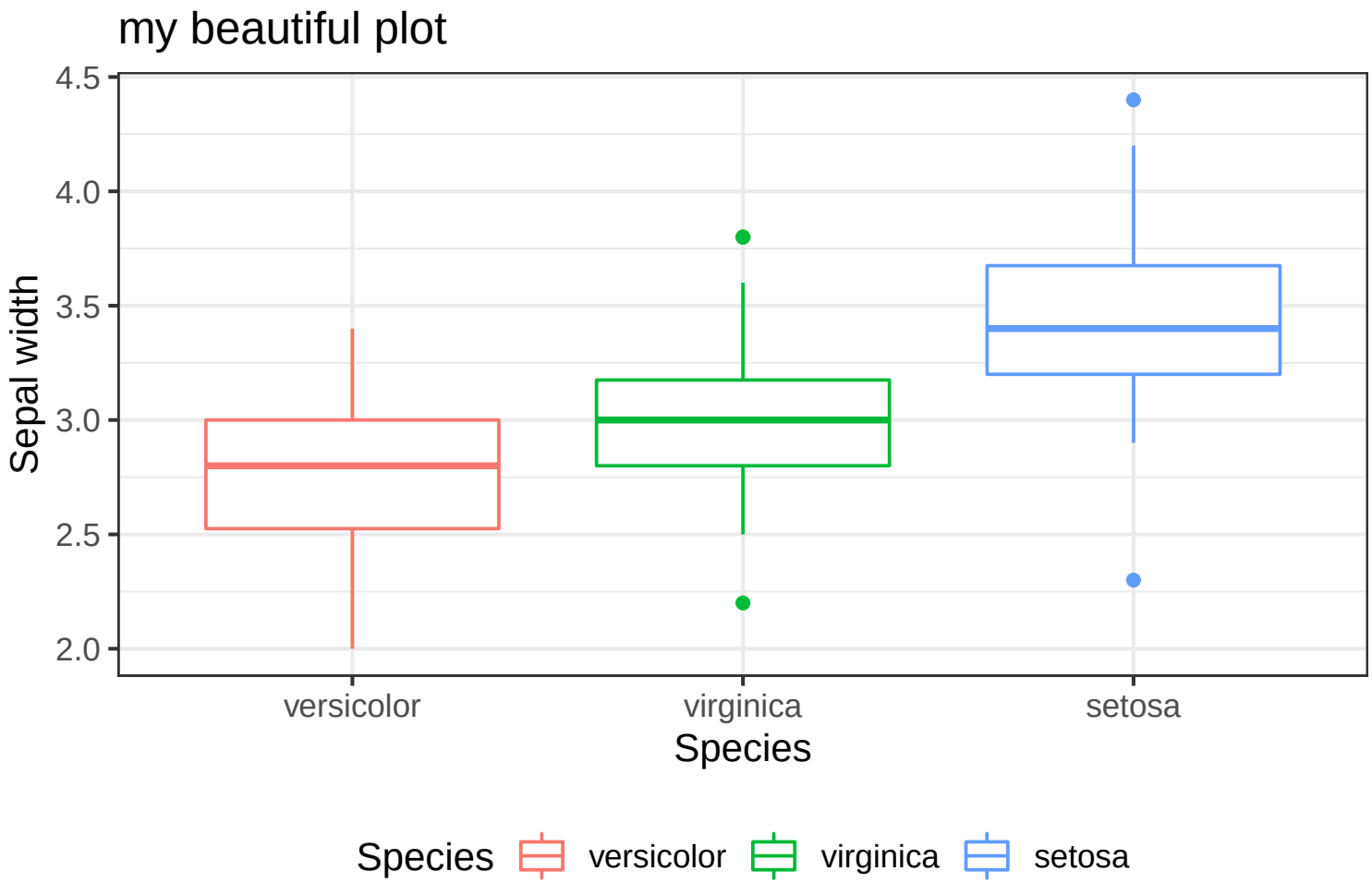
```
ggplot(iris) +
  geom_boxplot(aes(x = Species, y = Sepal.Width, color = Species)) +
  labs("species", y = "Sepal width", title = "my beautiful plot")
```



Add a layer: boxplot + annotation (Cont'd)

the aes depends on the geometry (here, a factor is expected for `x`)

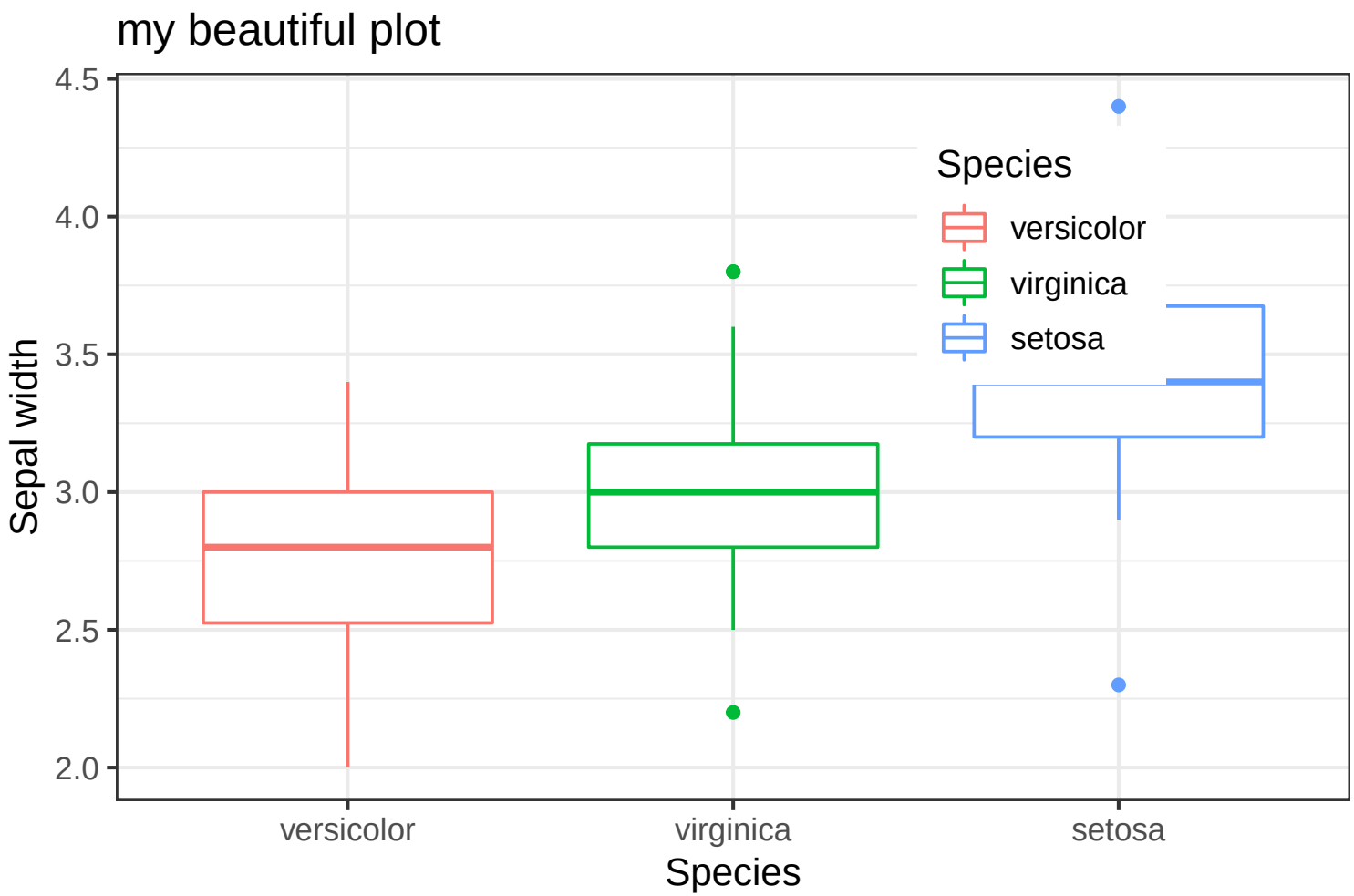
```
ggplot(iris) +
  geom_boxplot(aes(x = Species, y = Sepal.Width, color = Species)) +
  labs("species", y = "Sepal width", title = "my beautiful plot") +
  theme(legend.position = "bottom")
```



Add a layer: boxplot + annotation (Cont'd)

the aes depends on the geometry (here, a factor is expected for `x`)

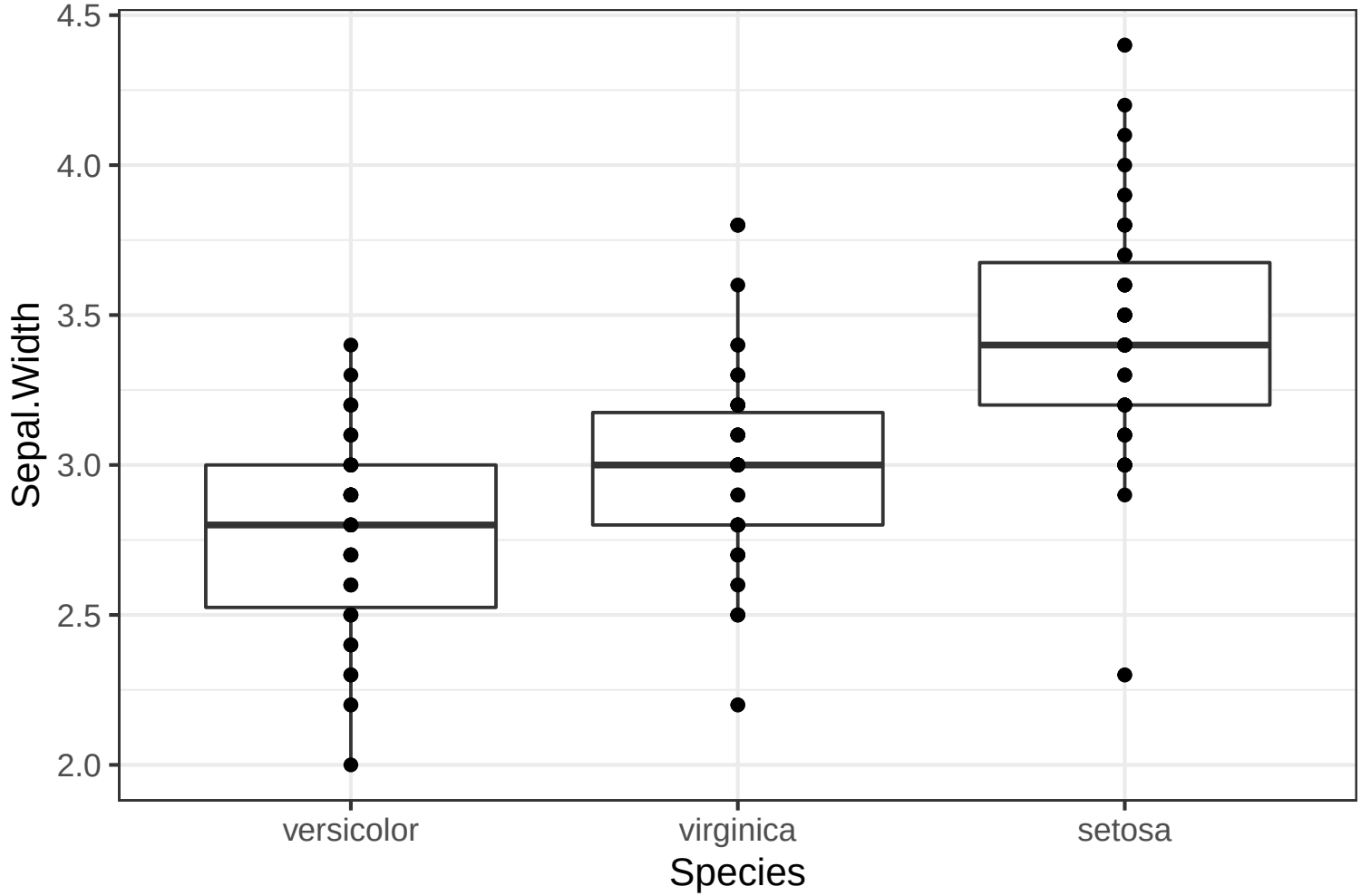
```
ggplot(iris) +
  geom_boxplot(aes(x = Species, y = Sepal.Width, color = Species)) +
  labs("species", y = "Sepal width", title = "my beautiful plot") +
  theme(legend.position = c(.75, .75))
```



Add several layers: boxplot + points

Note how I changed the use of `ggplot` since aesthetic where common to both

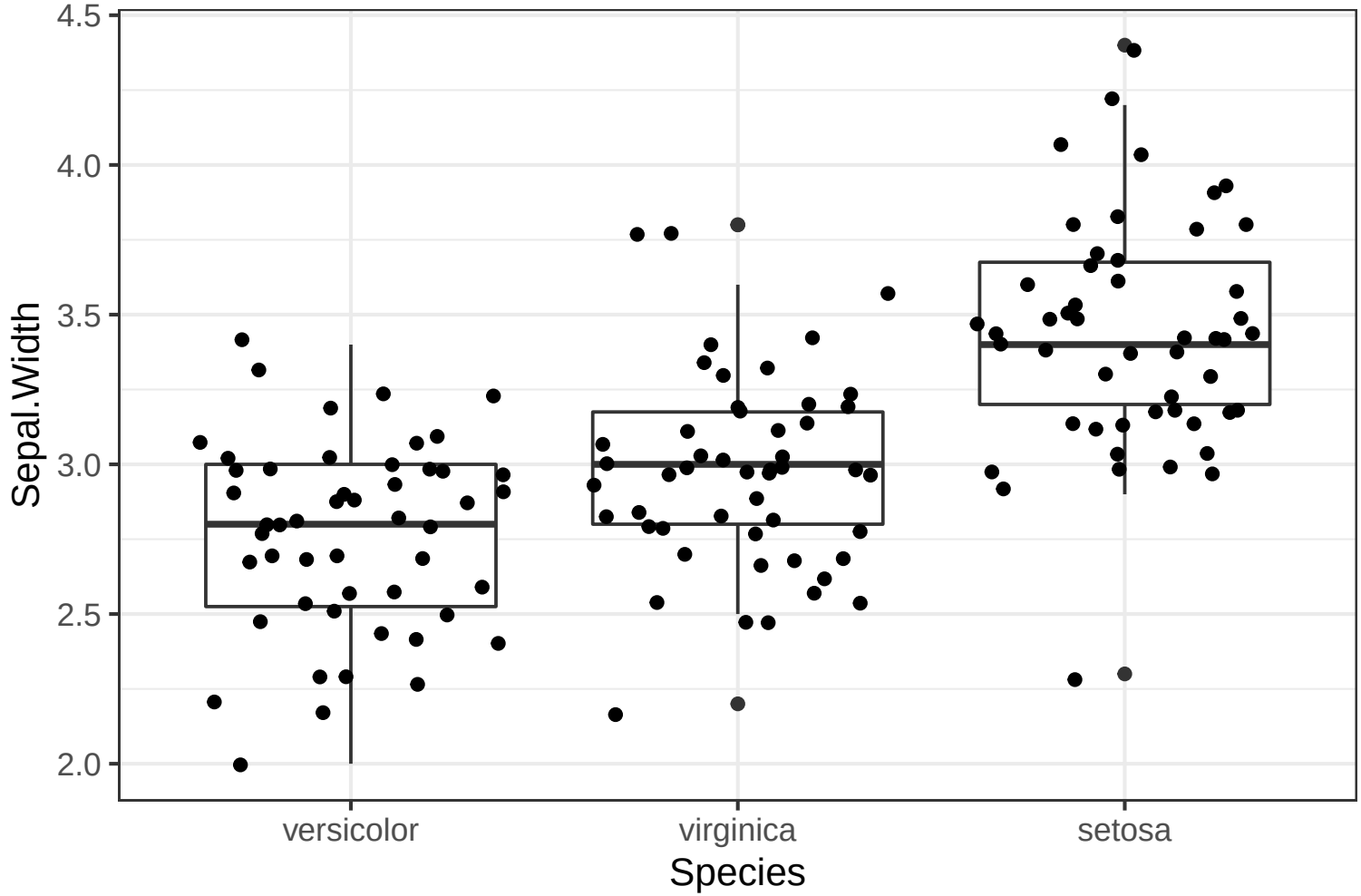
```
ggplot(iris, aes(x = Species, y = Sepal.Width)) + geom_boxplot() + geom_point()
```



Add several layers: boxplot + jitter

Note how I changed the use of `ggplot` since aesthetic where common to both

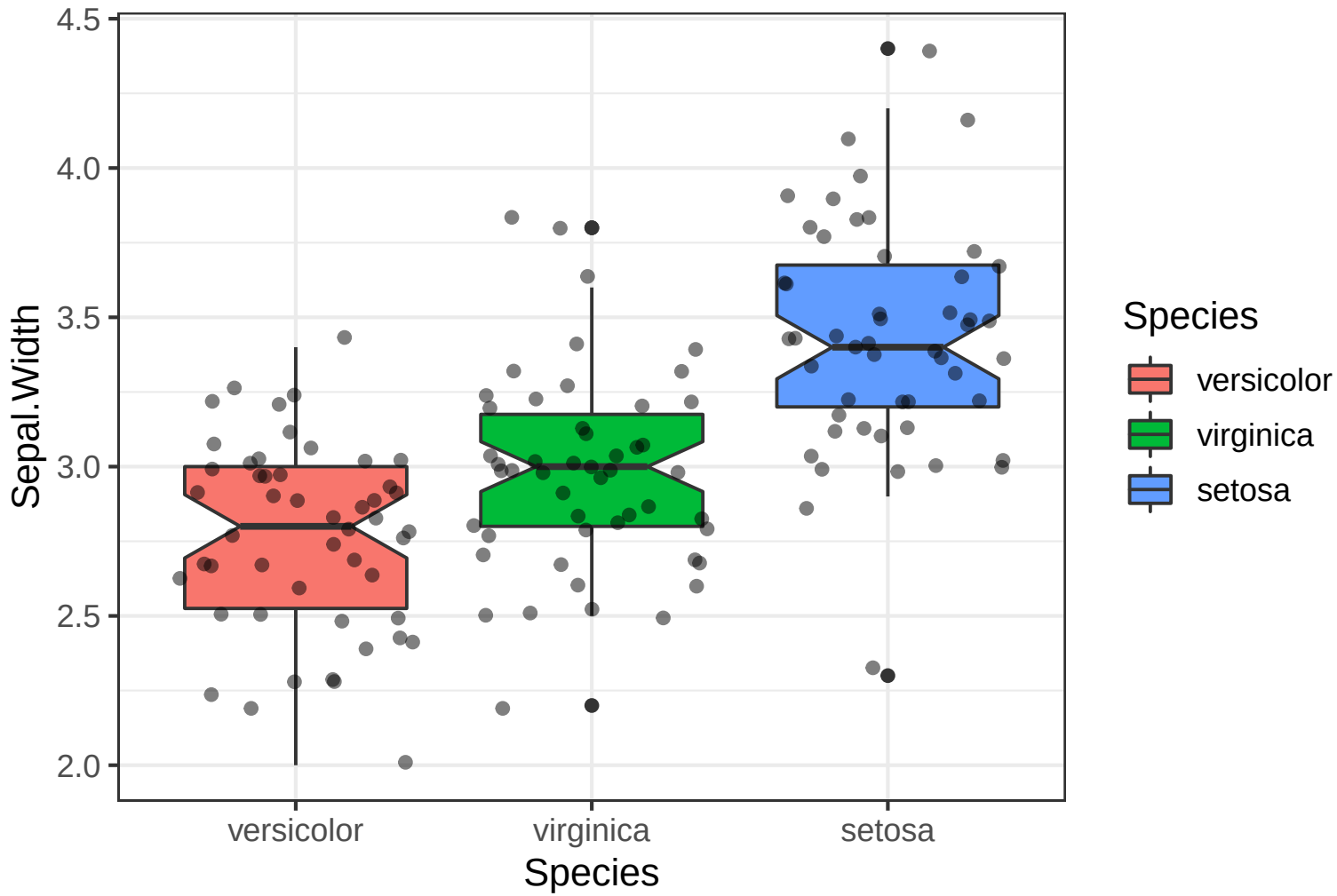
```
ggplot(iris, aes(x = Species, y = Sepal.Width)) + geom_boxplot() + geom_jitter()
```



Add several layers: boxplot + jitter

Note how I changed the use of `ggplot` since aesthetic where common to both

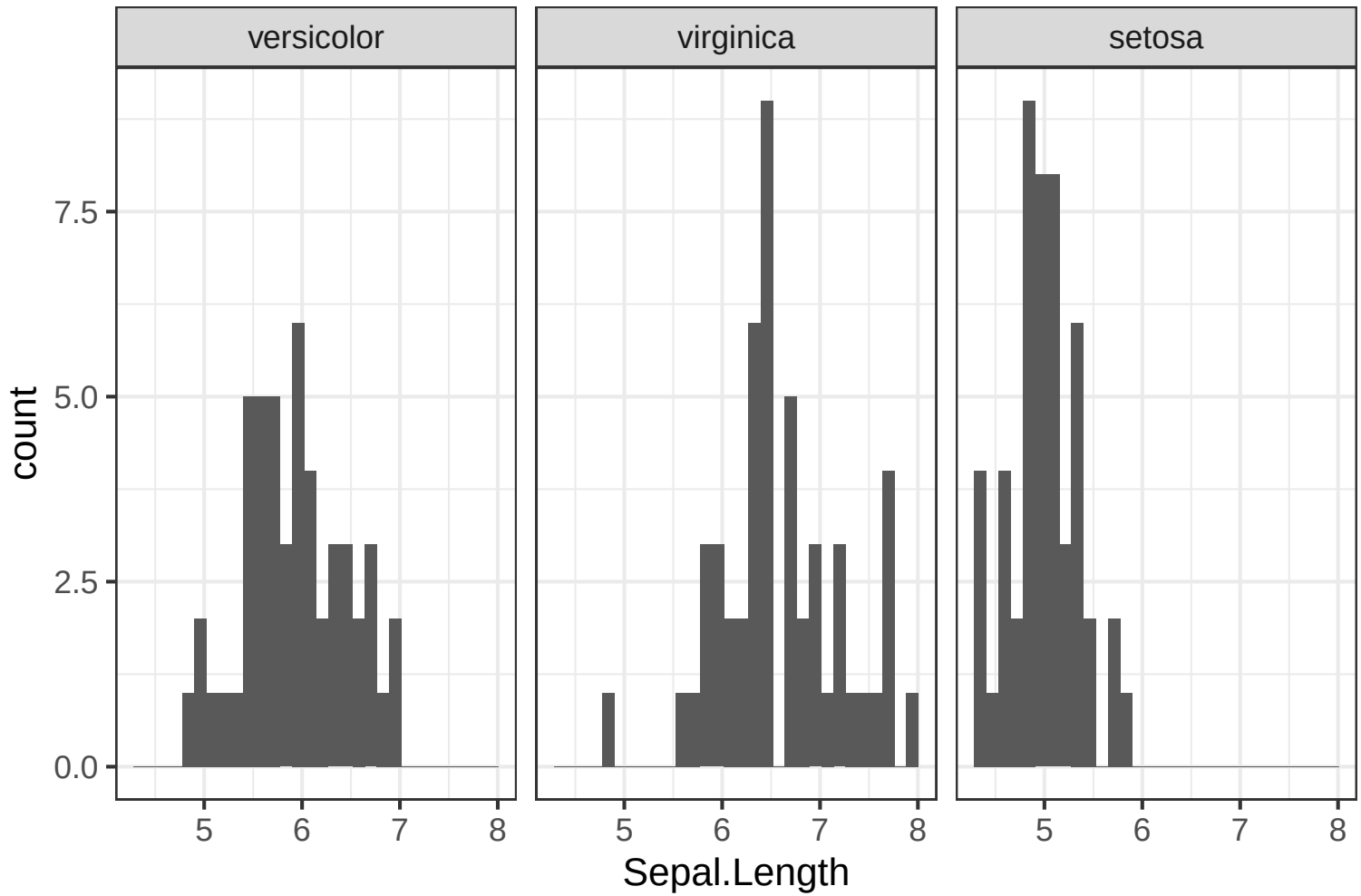
```
ggplot(iris, aes(x = Species, y = Sepal.Width)) +
  geom_boxplot(aes(fill = Species), notch = TRUE) +
  geom_jitter(alpha = .5)
```



Faceting

Note how I changed the use of ggplot since aesthetic where common to both

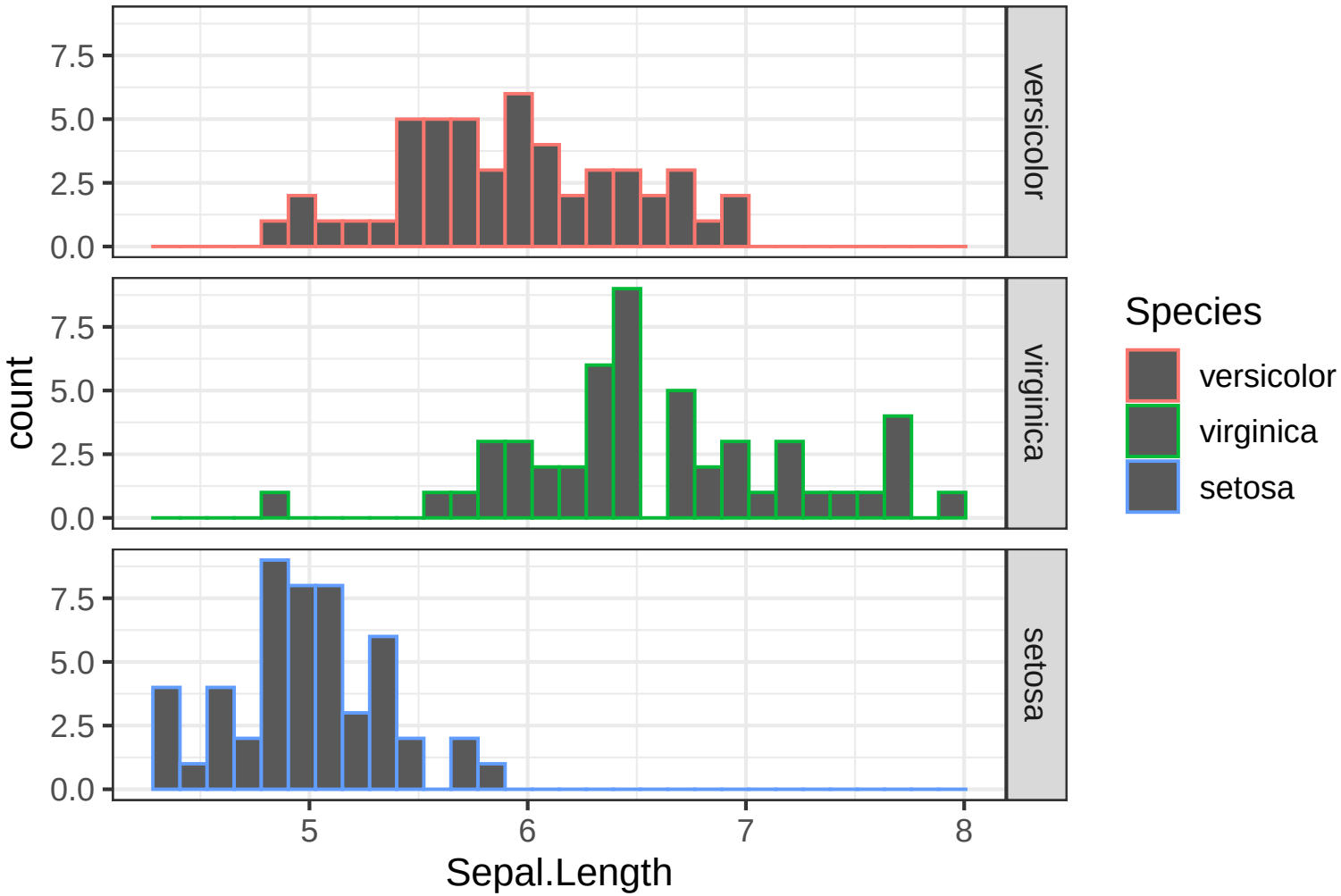
```
ggplot(iris) + geom_histogram(aes(x = Sepal.Length)) + facet_grid( . ~ Species)
```



Faceting (Cont'd)

Note how I changed the use of `ggplot` since aesthetic where common to both

```
ggplot(iris) + geom_histogram(aes(x = Sepal.Length, color = Species)) + facet_grid(Species ~ .)
```



Use ggplot2 in conjonction with other packages

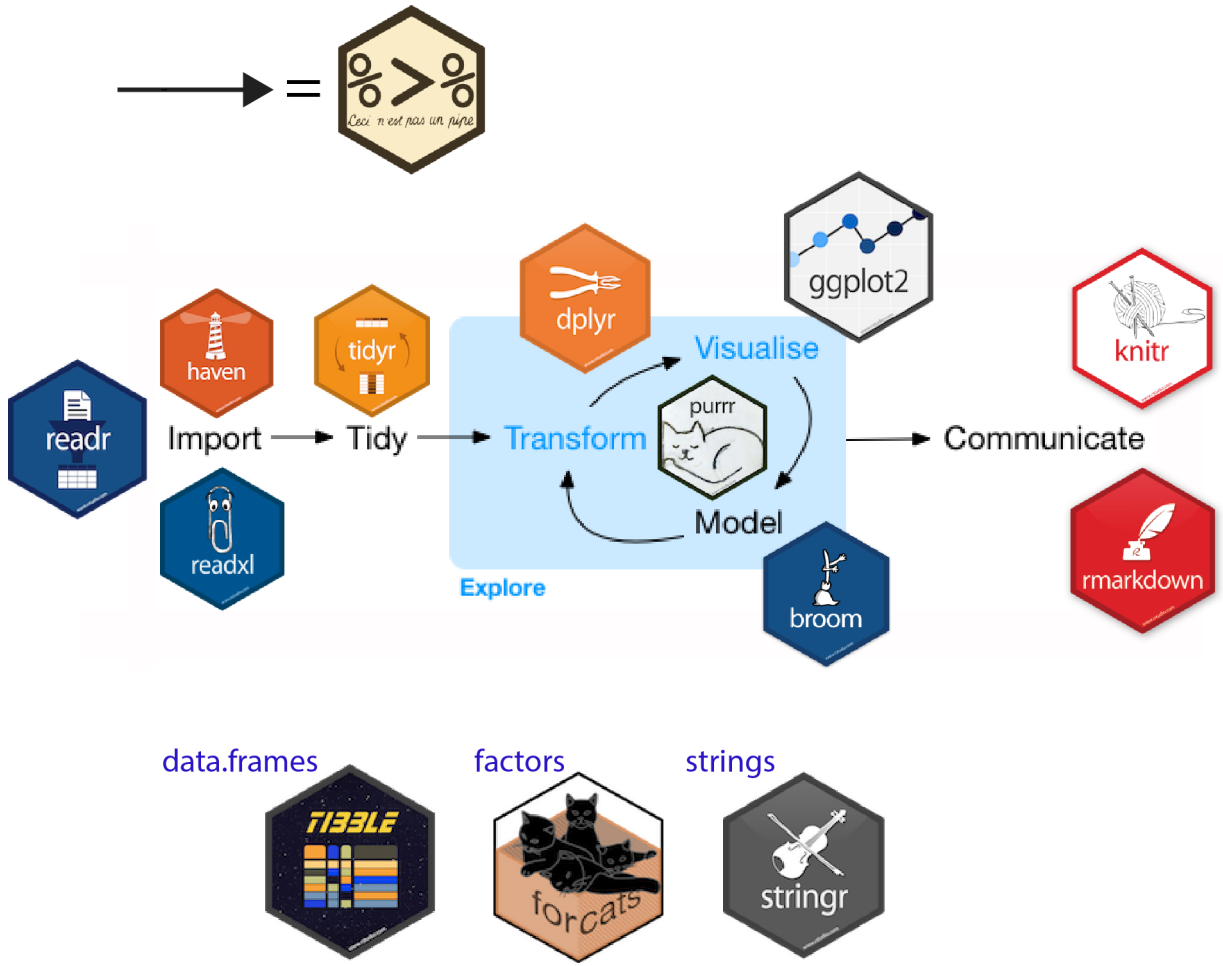


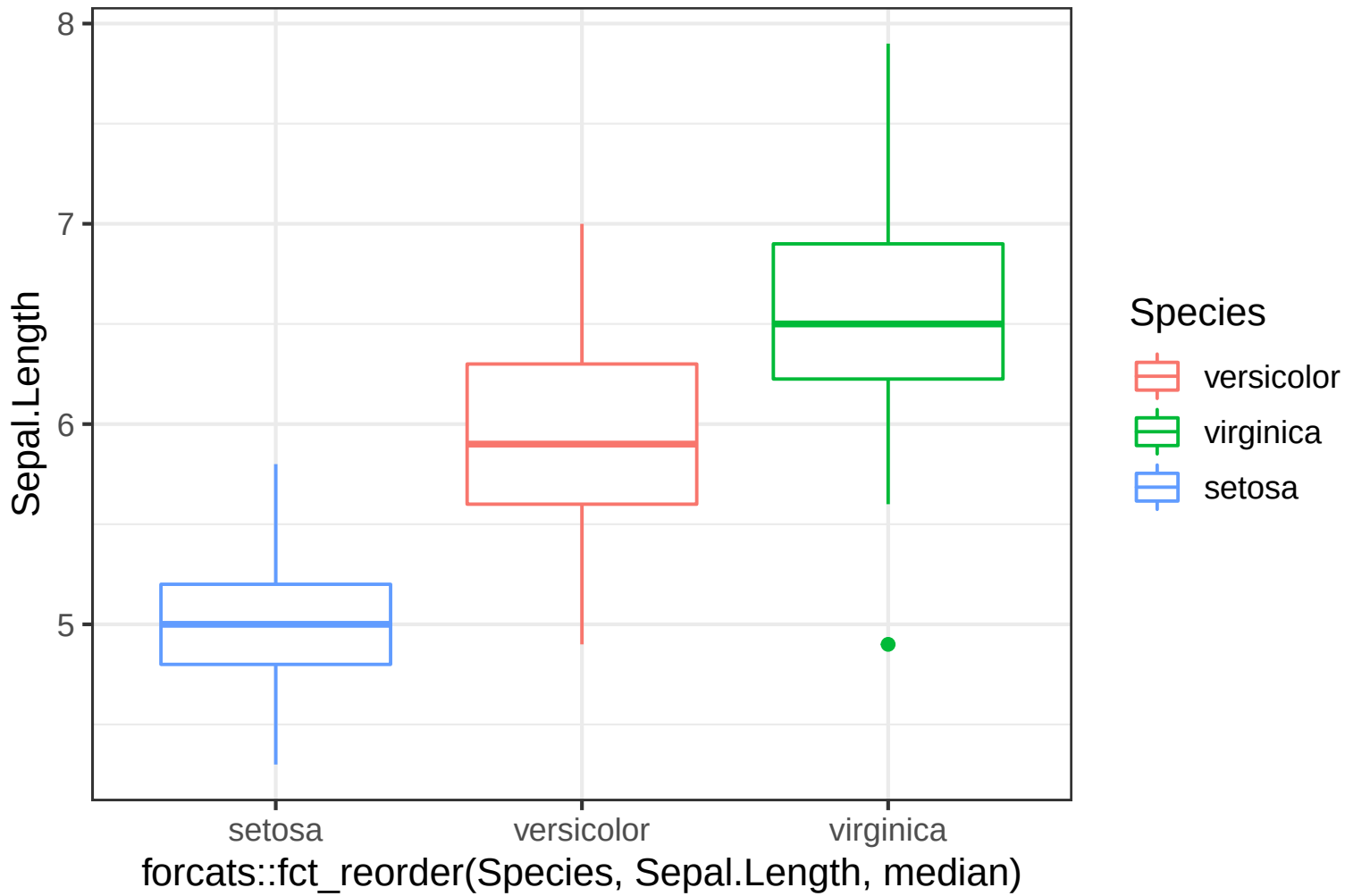
Figure 24: Remember the data process scheme?

The model/visualization part need constant wrangling/new arragement of your data set

Example: `forcats`

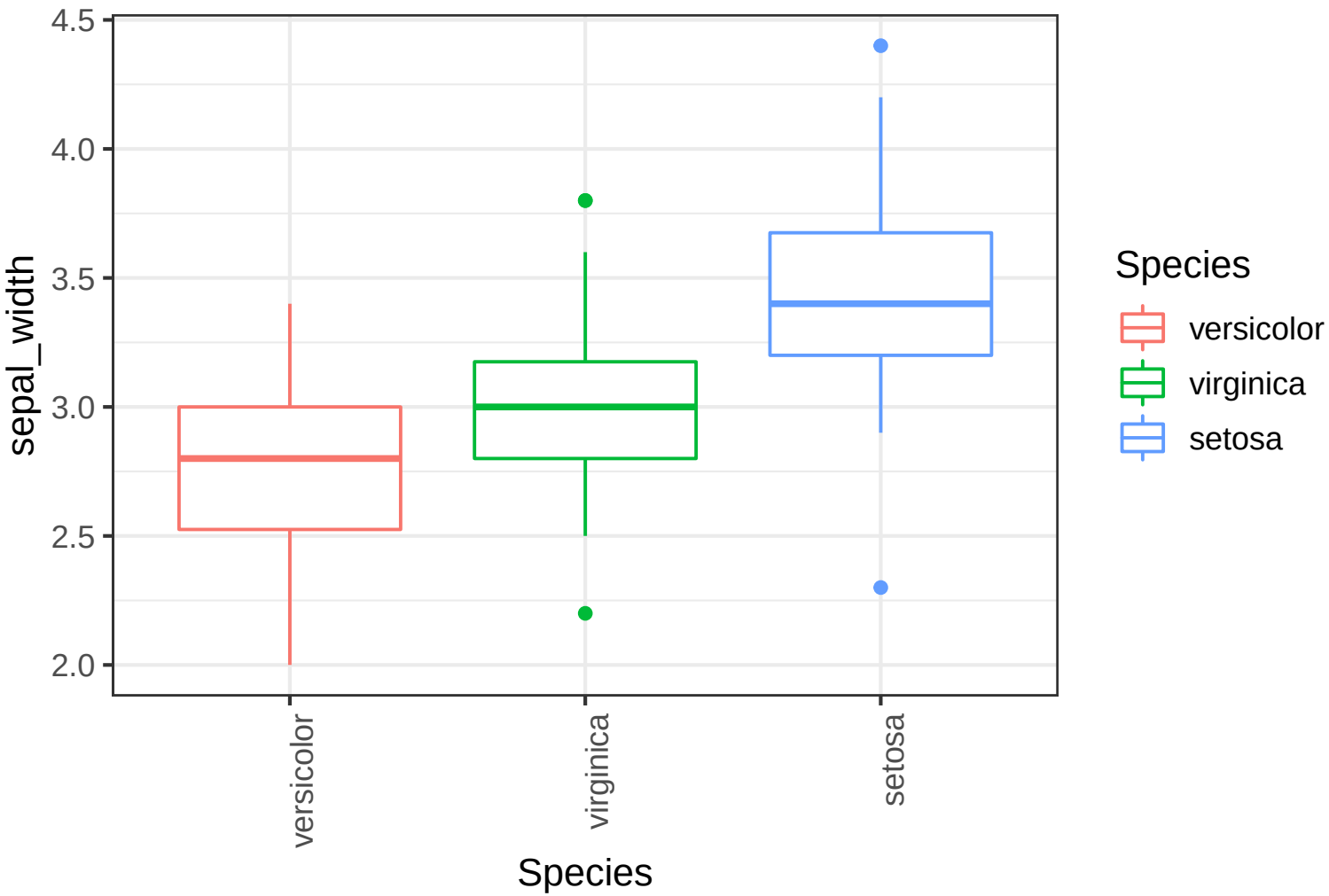
Automatically more readable graphs

```
ggplot(iris) +
  geom_boxplot(aes(
    x = forcats::fct_reorder(Species, Sepal.Length, median),
    y = Sepal.Length, color = Species))
```



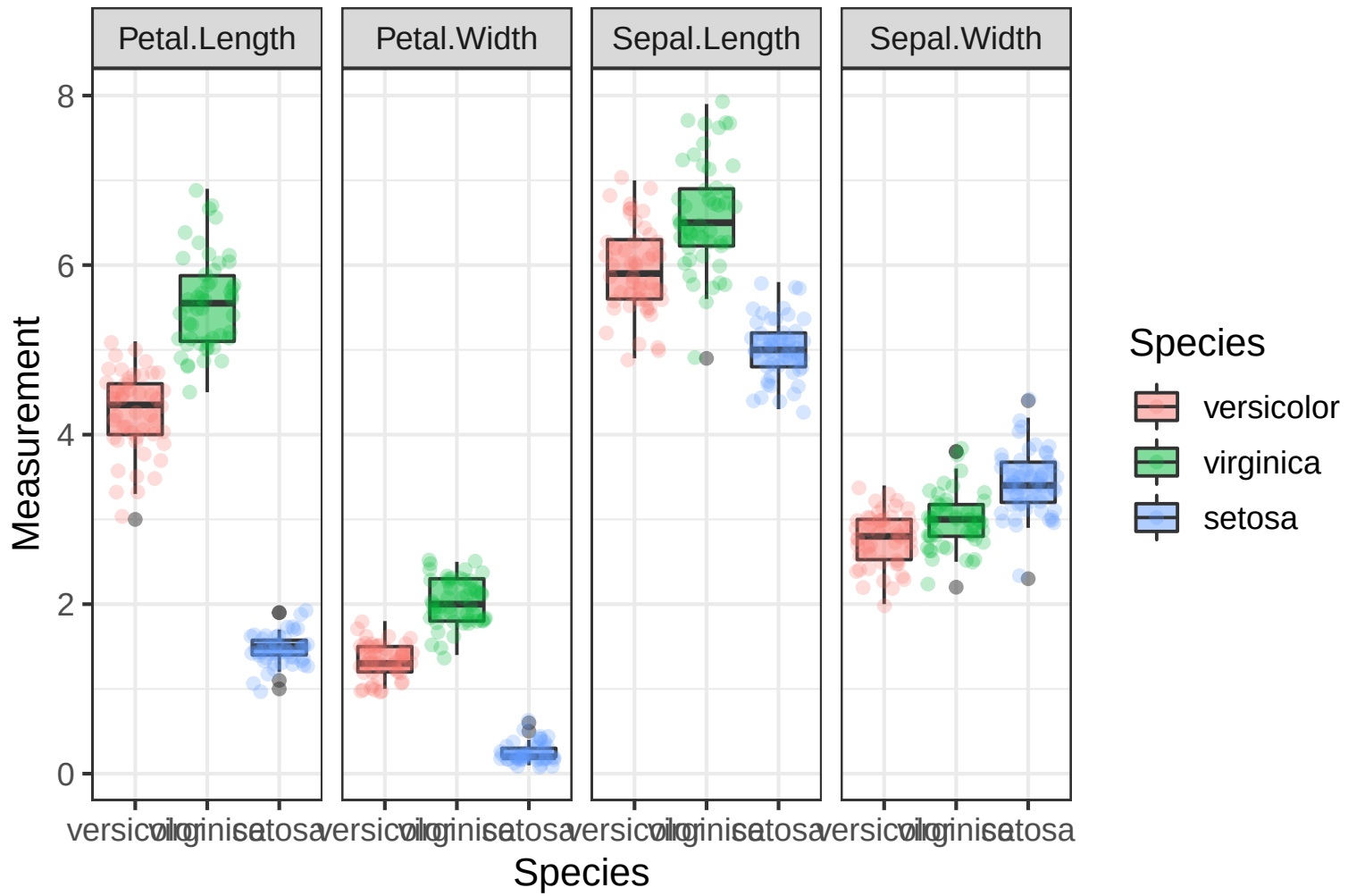
Example: dplyr + %>% for renaming before ploting

```
iris %>% rename(sepal_length = Sepal.Length,
                sepal_width = Sepal.Width,
                petal_length = Petal.Length,
                pepal_width = Petal.Width) %>%
  ggplot() + geom_boxplot(aes(x = Species, y = sepal_width, color = Species)) +
  theme(axis.text.x=element_text(angle=90,hjust=1))
```



Example: dplyr + %>% for gathering new data

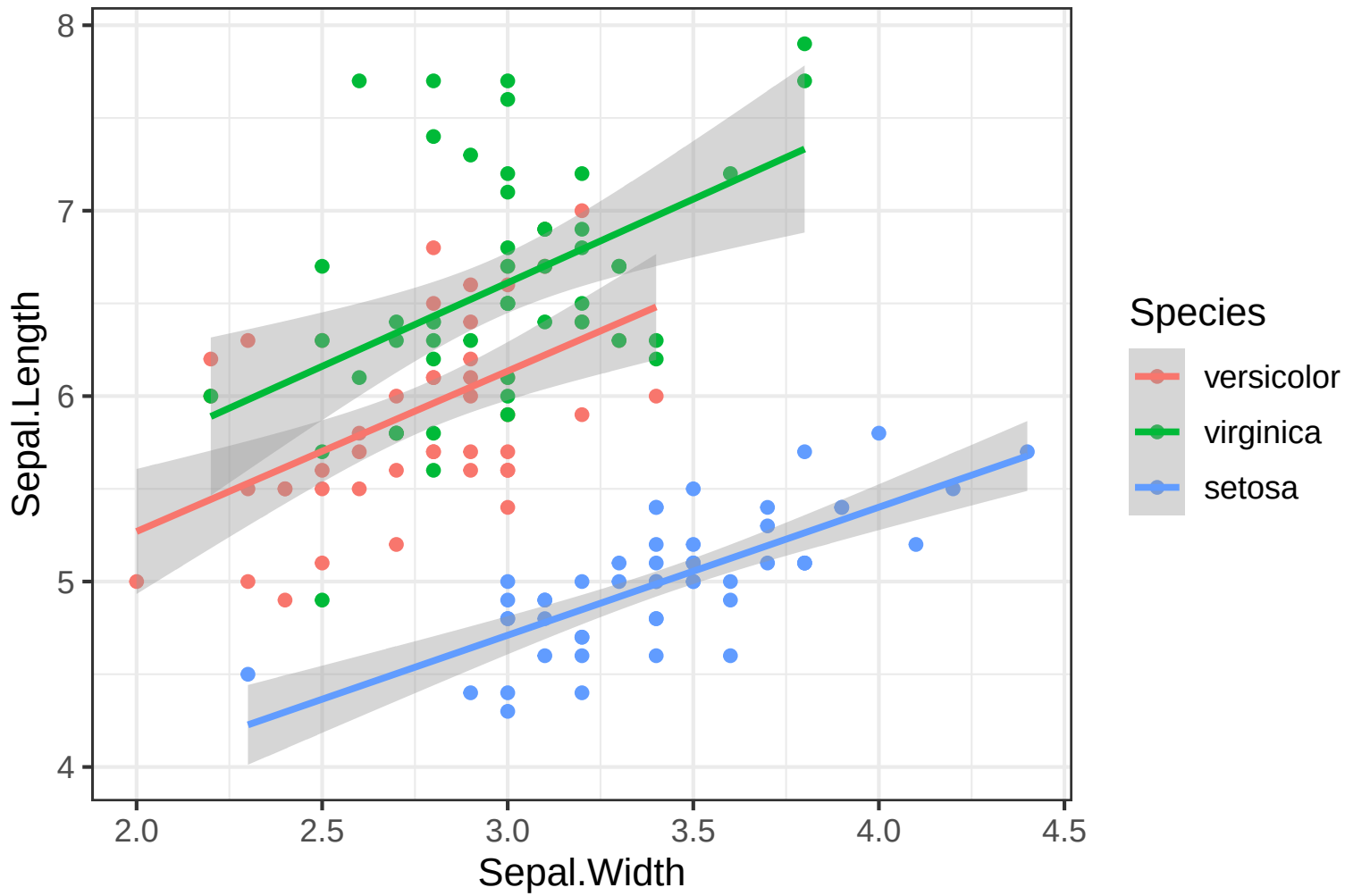
```
iris %>%
  gather(key = "Attribute", value = "Measurement", -Species) %>%
  ggplot(aes(x = Species, y = Measurement)) + geom_boxplot(aes(fill = Species), alpha = .5) +
  geom_jitter(aes(color = Species), alpha = 0.25) + facet_grid(. ~ Attribute)
```



Add a model layer

Adjust one linear model per species

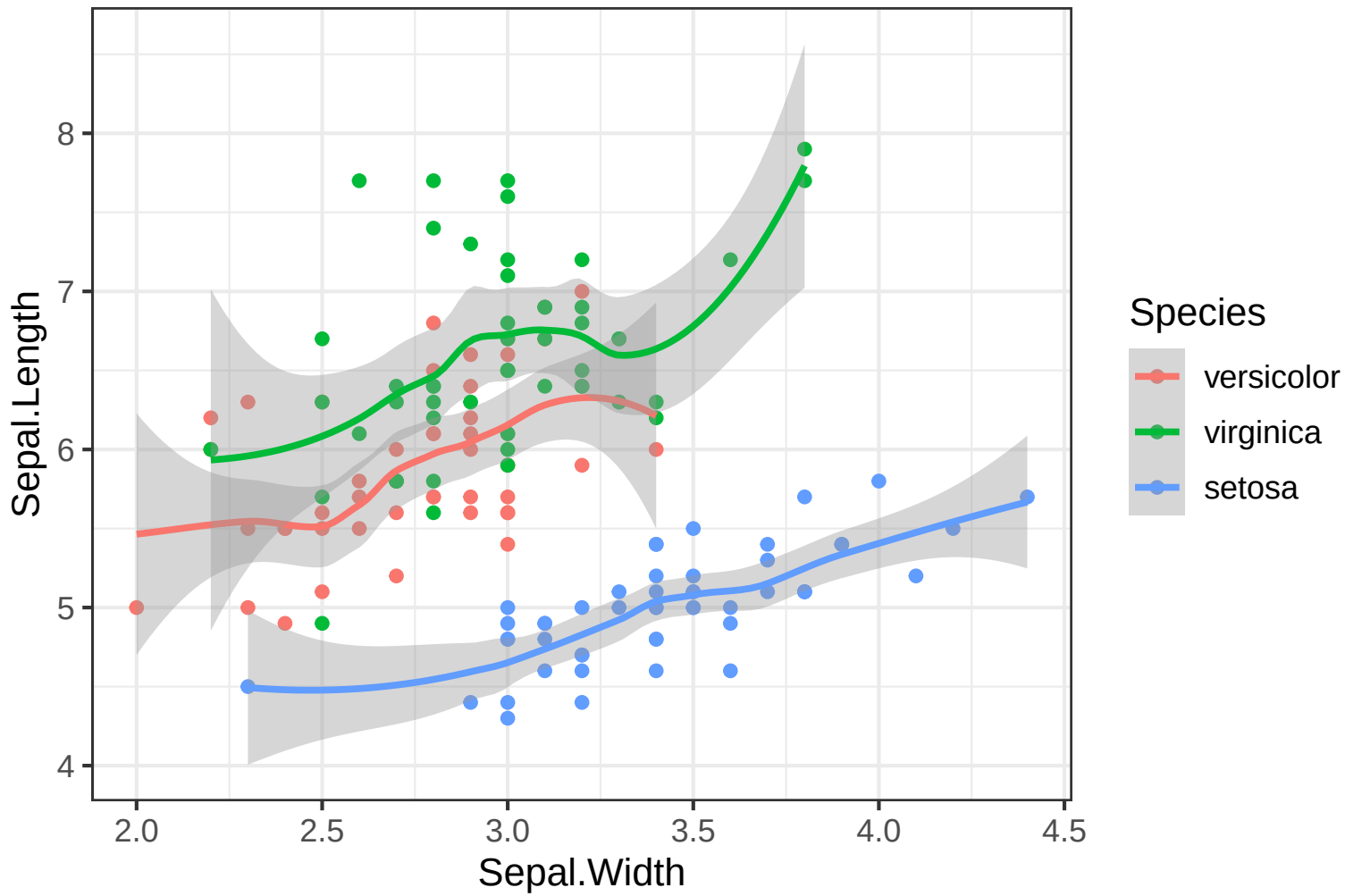
```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, colour = Species)) +
  geom_point() +  geom_smooth(method = lm)
```



Add a model layer (Cont'd)

Adjust one nonlinear model per species

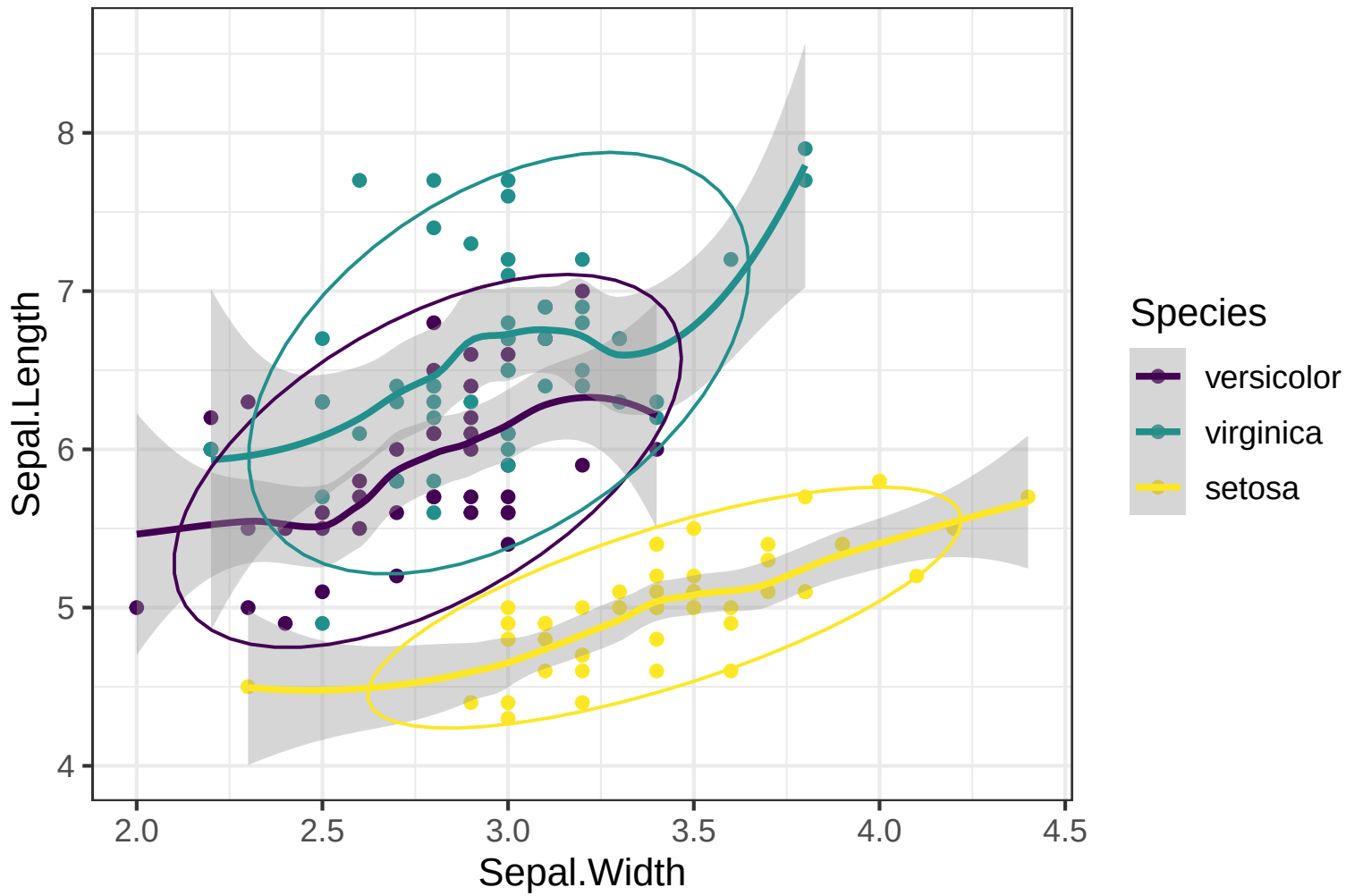
```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point() +  geom_smooth(method = loess)
```



Add model + stat layers and colorblind pallete

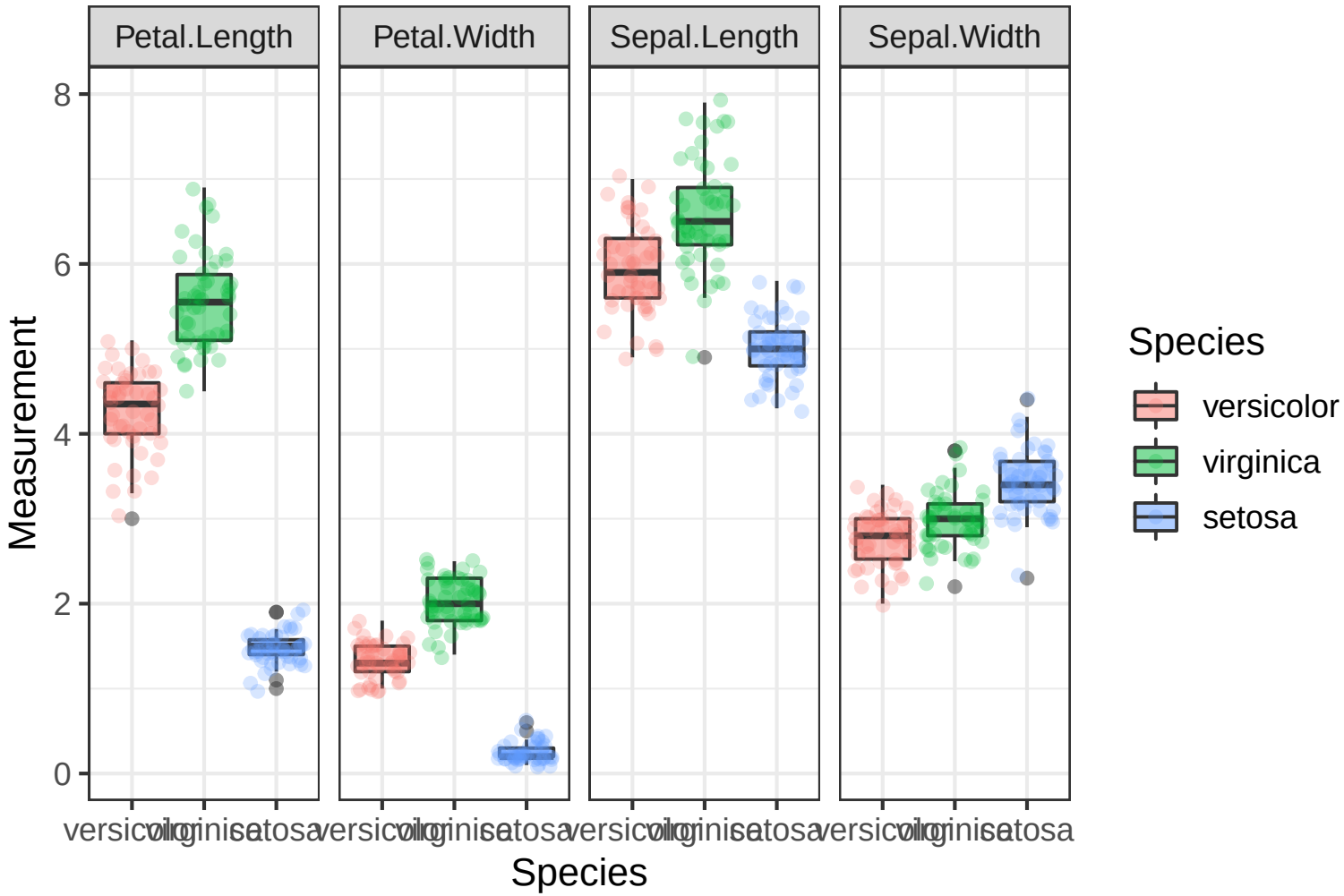
Adjust one nonlinear model per species

```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point() +  geom_smooth(method = loess) +
  stat_ellipse() + viridis::scale_color_viridis(discrete = TRUE)
```



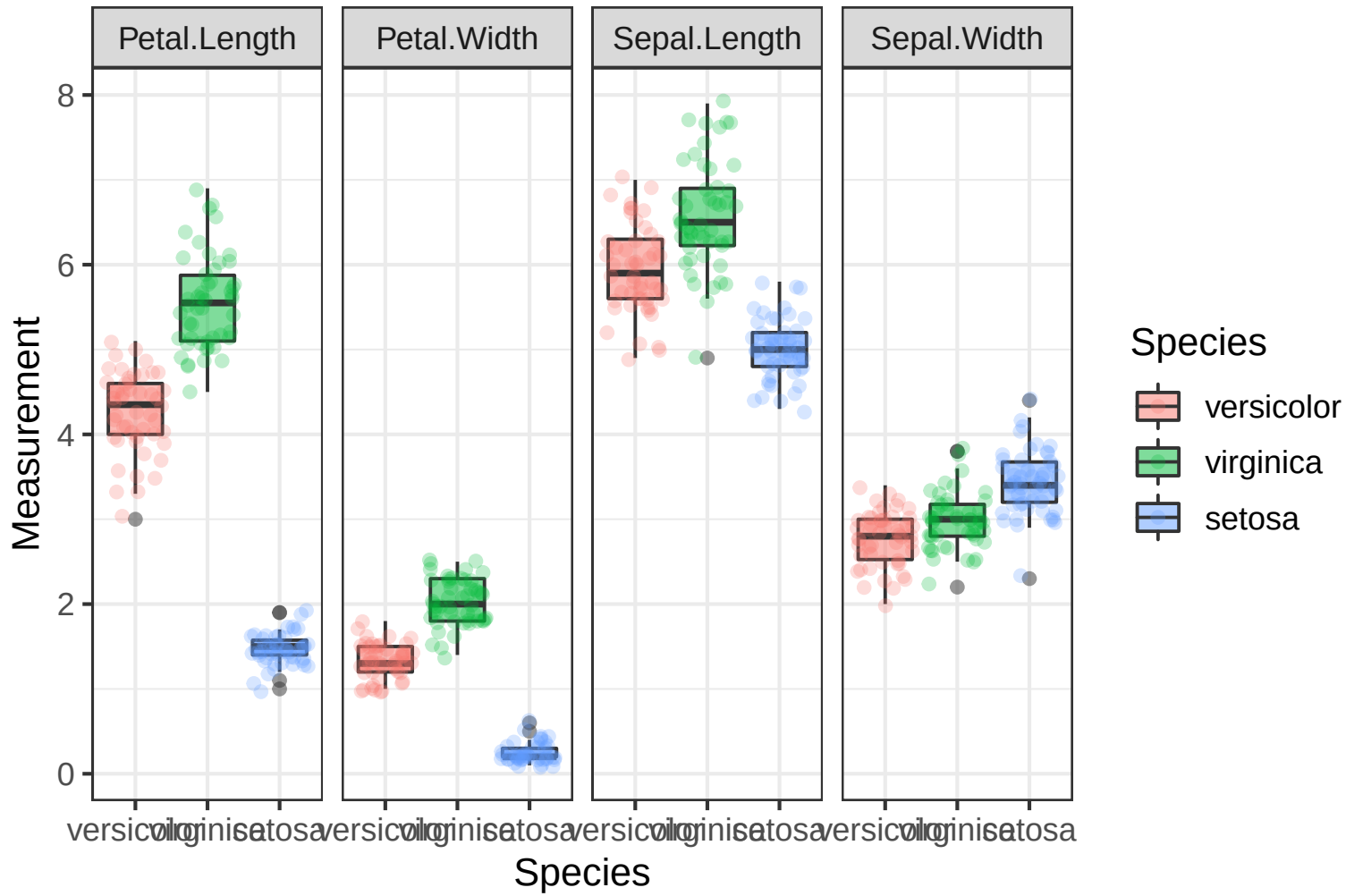
Transform coordinate: log-scales

```
ggplot(diamonds, aes(carat, price)) +
  geom_point()
```



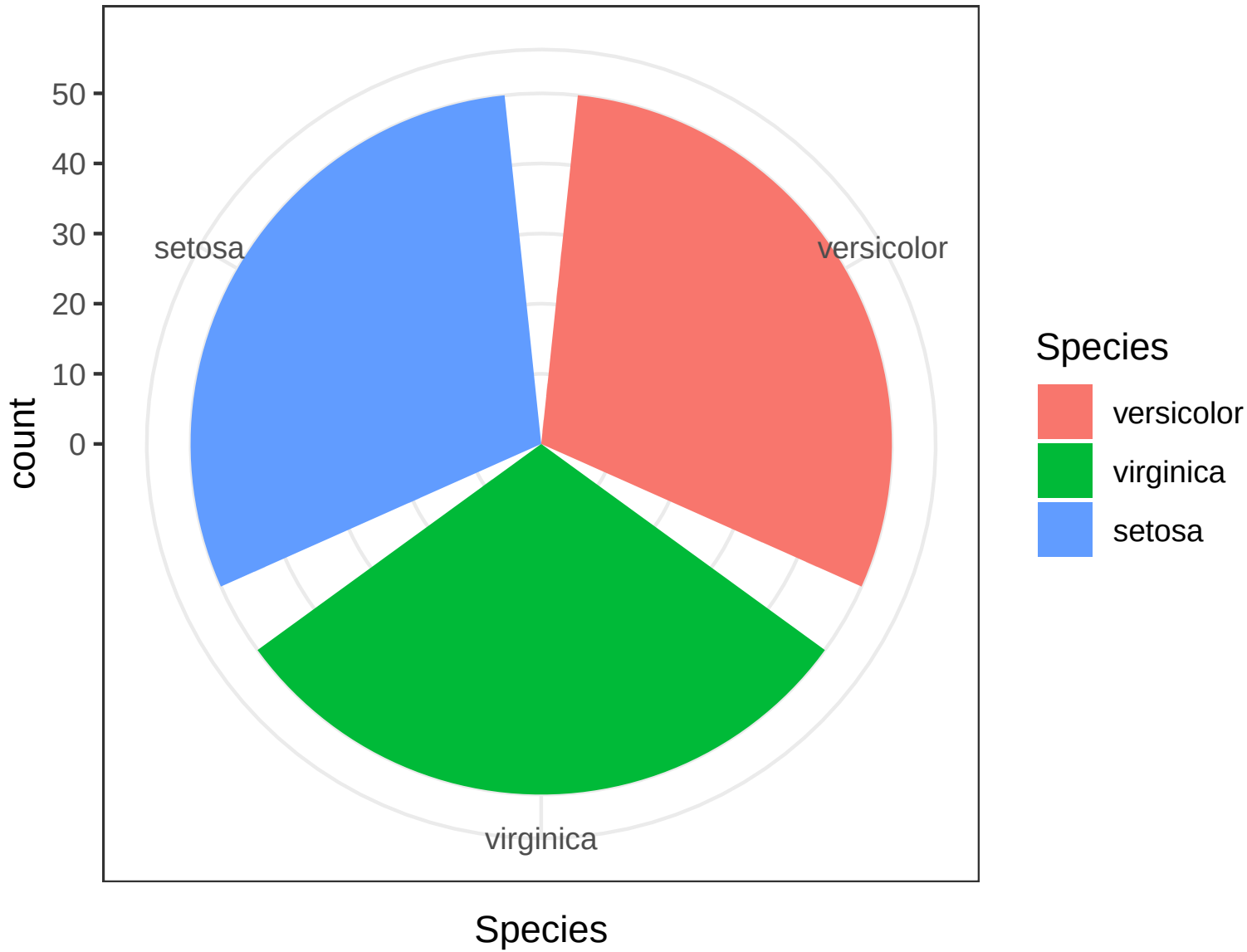
Transform coordinate: log-scales

```
ggplot(diamonds, aes(carat, price)) +
  geom_point() + coord_trans(x = "log10") + coord_trans(y = "log10")
```



Changing coordinate system: polar

```
ggplot(iris, mapping = aes(x = Species, fill = Species)) +
  geom_bar() + coord_polar() + theme_bw()
```

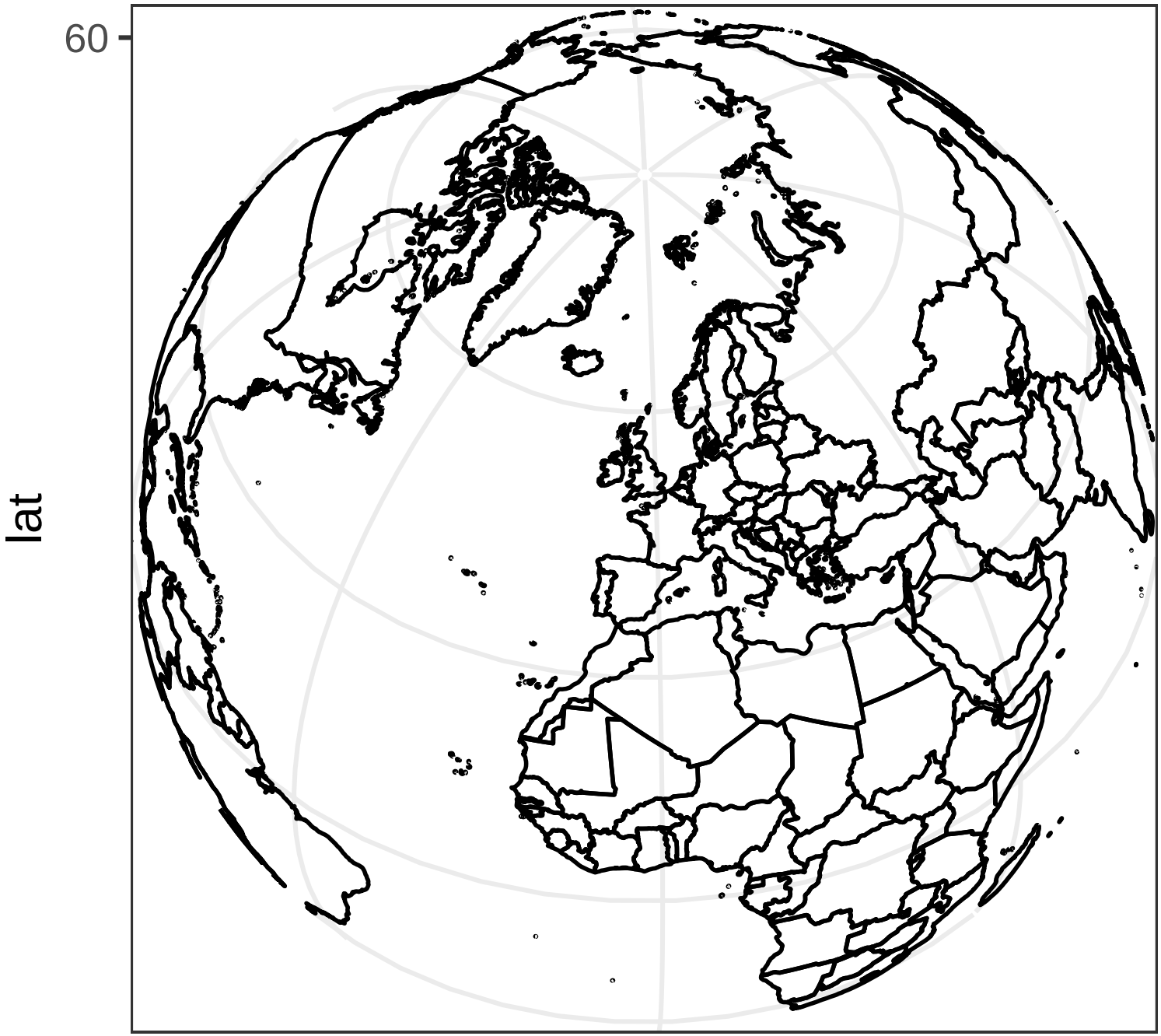


Changing coordinate system: maps I

```
# World map, using geom_path instead of geom_polygon
world <- map_data("world")
worldmap <- ggplot(world, aes(x = long, y = lat, group = group)) +
  geom_path() +
  scale_y_continuous(breaks = (-2:2) * 30) +
  scale_x_continuous(breaks = (-4:4) * 45)

# Orthographic projection centered on Paris
worldmap + coord_map("ortho",orientation = c(48, -2, 0))
```

Changing coordinate system: maps II



References

Chang, W. (2012). *R graphics cookbook: Practical recipes for visualizing data*. "O'Reilly Media, Inc.". Retrieved from http://www.cookbook-r.com/Graphs/

R Core Team. (2017). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Wickham, H. (2014). *Advanced r*. CRC Press. Retrieved from http://adv-r.had.co.nz/

Wickham, H. (2016). *Ggplot2: Elegant graphics for data analysis*. Springer. Retrieved from http://ggplot2.tidyverse.org/reference/

Wickham, H., & Grolemund, G. (2017). *R for data science: Visualize, model, transform, tidy, and import data*. "O'Reilly Media, Inc.". Retrieved from http://r4ds.had.co.nz

Wilkinson, L. (2006). *The grammar of graphics*. Springer Science & Business Media.