Lecture 3 - Data Wrangling and Visualisation

Andrew Stewart

Andrew.Stewart@manchester.ac.uk

@ajstewart_lang

https://github.com/ajstewartlang

Week	Topic
1	Introduction, Open Science, and Power
2	Introduction to R
3	Data Wrangling and Visualisation
4	General Linear Model - Regression
5	General Linear Model - Regression
6	No Timetabled Lecture - Reading Week
7	Consolidation Lab
8	General Linear Model - ANOVA
9	General Linear Model - ANOVA
10	Tidy Thursday Data Wrangling & Visualisation Challenge
11	Reproducing your Computational Environment using Binder
12	Dynamic, Reproducible Presentations Using xaringan

Semester 1 Assignments

Data wrangling and visualisation – Due around the end of November

ANOVA – Due around mid-January

Last Week

- We had our first introduction to R and RStudio.
- In the second half of class, you went from zero to hero in terms of using R and running a script for some data manipulation and graphing using ggplot.

Some Academic Twitter Accounts on Open Science to Follow

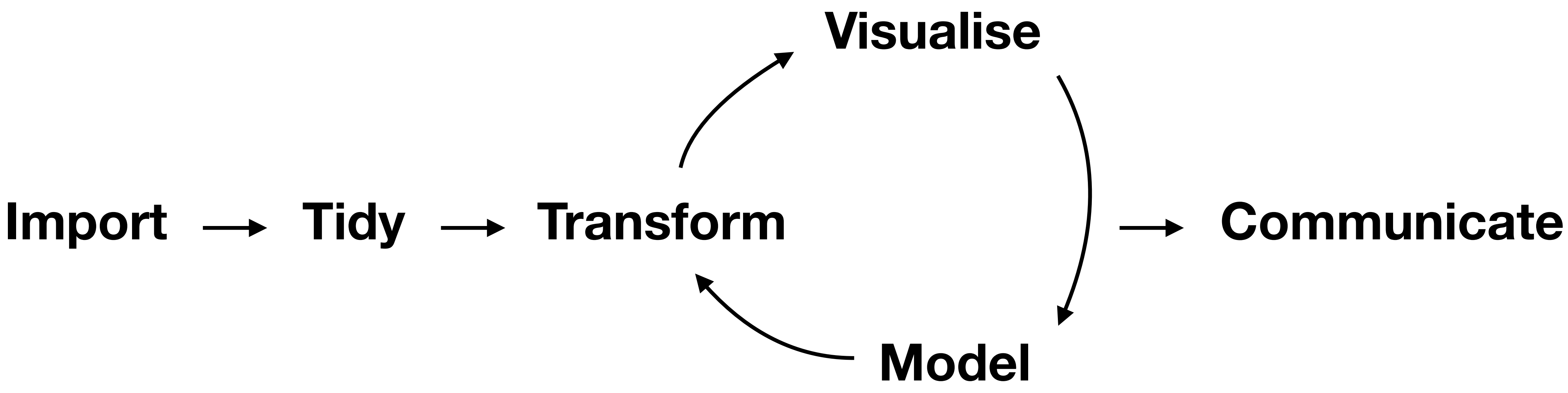
- Brian Nosek (Virginia) - @BrianNosek
- Dorothy Bishop (Oxford) - @deevybee
- Marcus Munafò (Bristol) - @MarcusMunafo
- Chris Chambers (Cardiff) - @chrisdc77
- The UK Reproducibility Network - @UKRepro
- Center for Open Science - @OSFramework

This Week

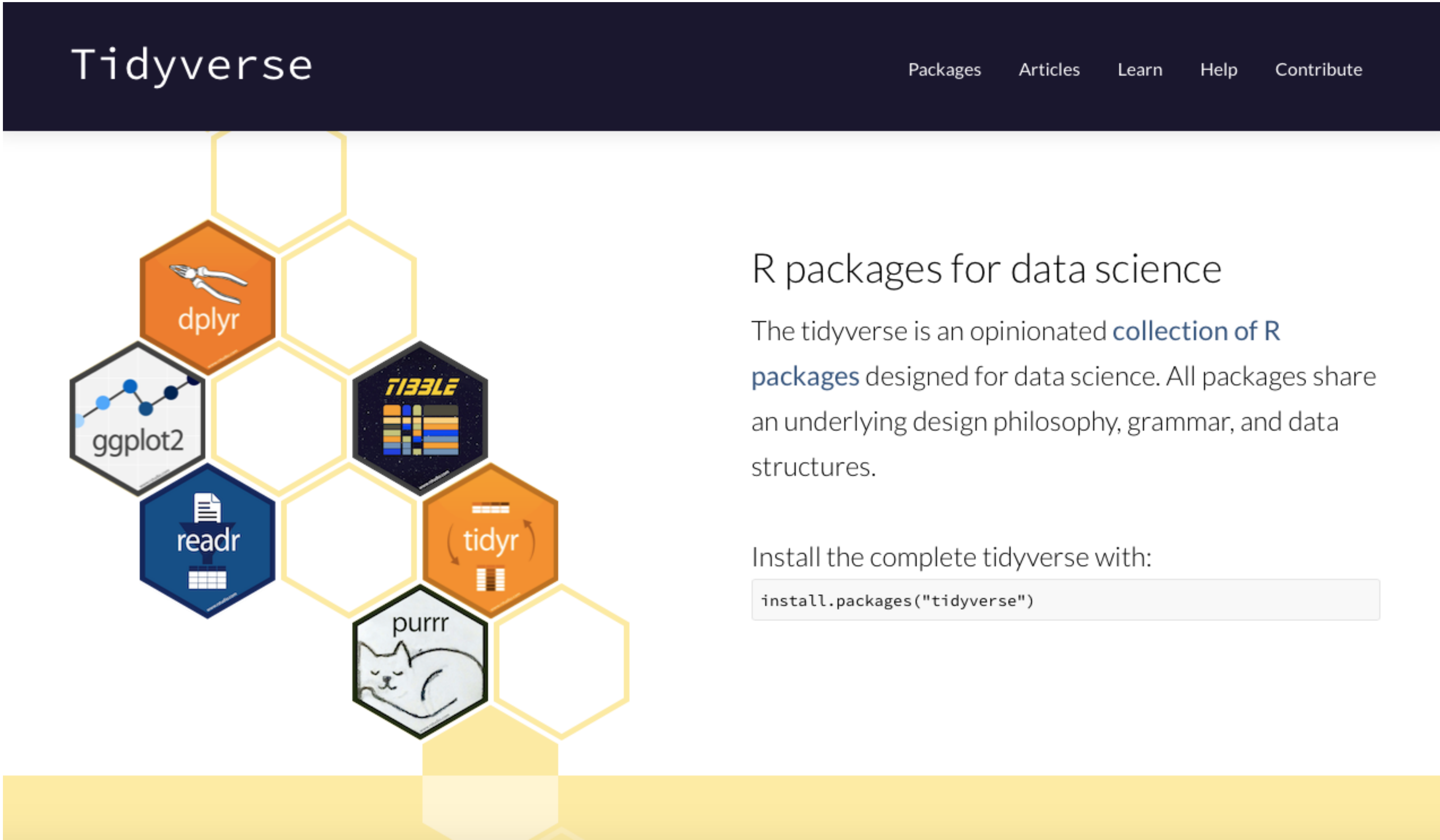
- We’re going to look at some data wrangling - getting your data into the right format and shape for analysis.

- We’re also looking at data visualisation (aka data viz.) - you should always visualise your data (often in more than one way) before you move onto statistical modelling…

Workflow in the Tidyverse (Garrett Grolemund and Hadley Wickham) - from Data to Write-up



https://www.tidyverse.org



R packages for data science

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

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Tidyverse packages

- The Tidyverse contains a number of packages, all containing functions that are designed to ‘play well’ with each other. Packages include `ggplot2`, `dplyr`, and `tidyr`.

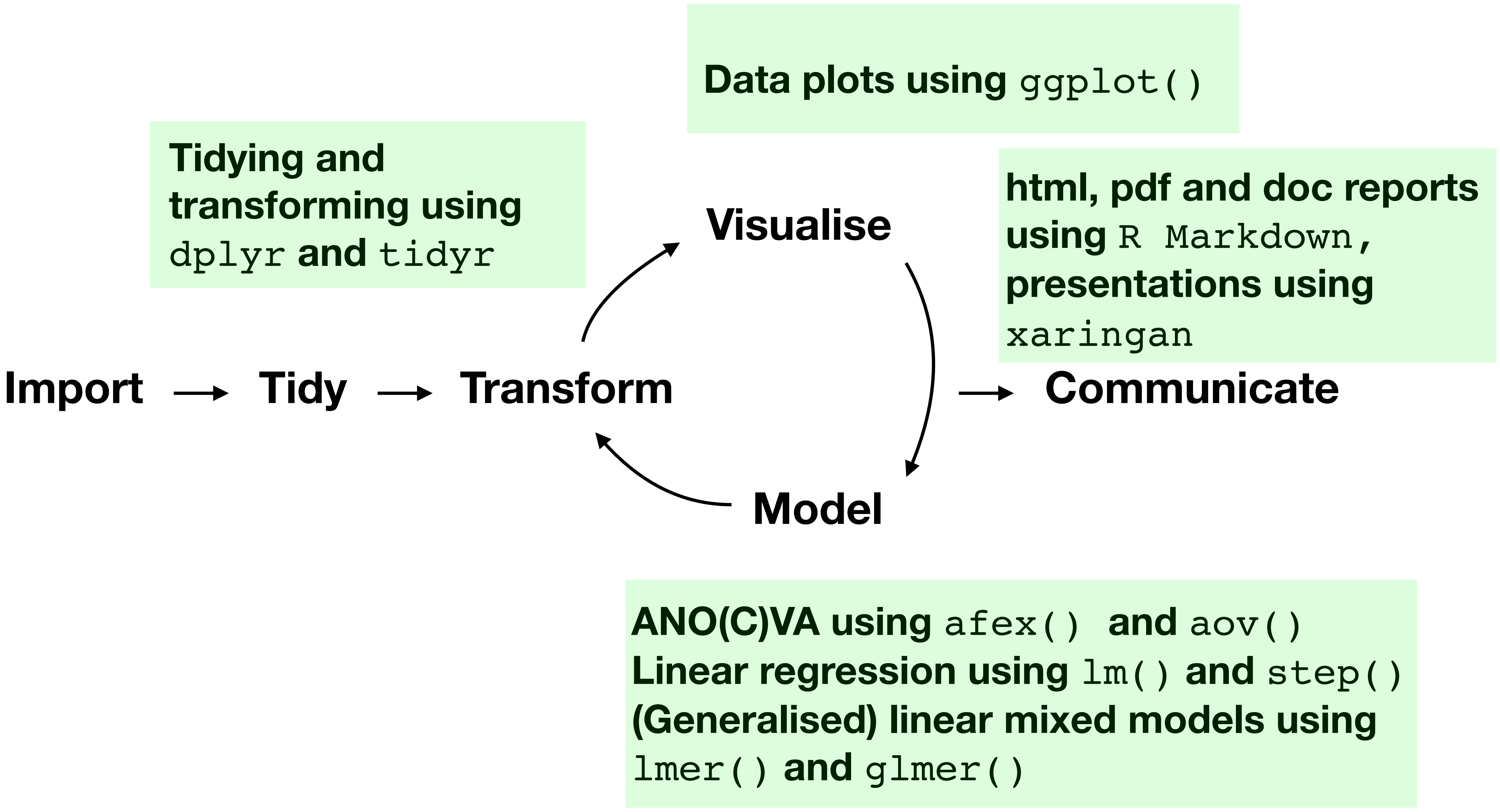
- You can load each package separately with (e.g.)

```
> library(ggplot2)
```

- or load all tidyverse packages with

```
> library(tidyverse)
```

Workflow



Import → Tidy → Transform

Visualise

Model

→ Communicate

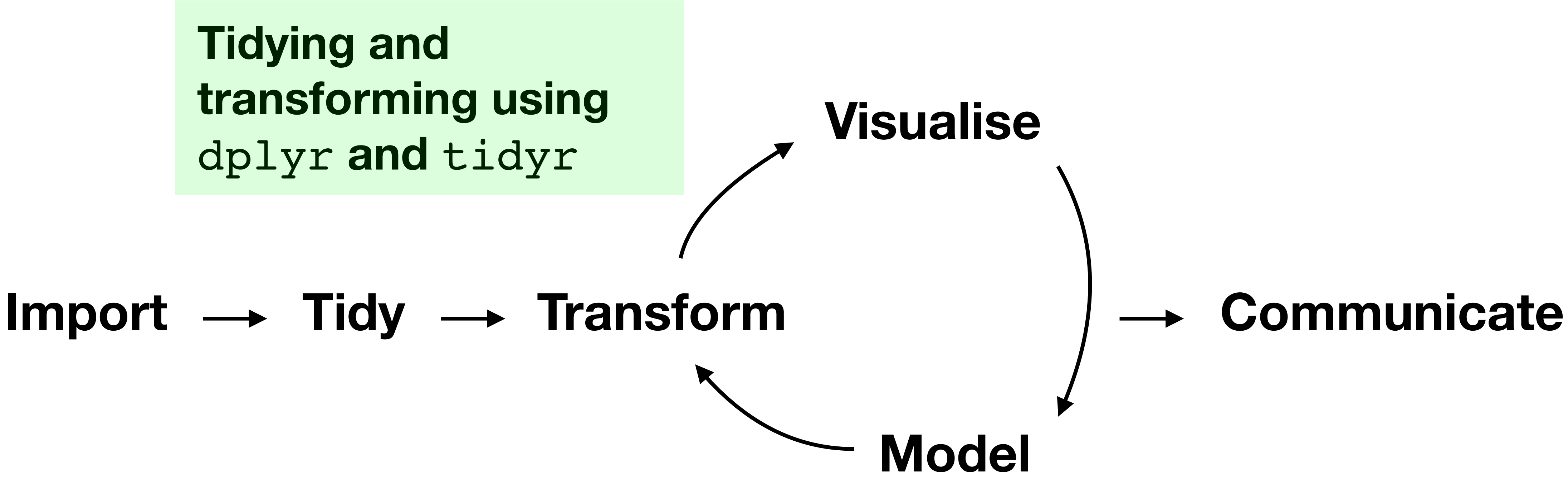
Tidying and transforming using `dplyr` and `tidyr`

Data plots using `ggplot()`

html, pdf and doc reports using `R Markdown`, presentations using `xaringan`

ANO(C)VA using `afex()` and `aov()`
Linear regression using `lm()` and `step()`
(Generalised) linear mixed models using `lmer()` and `glmer()`

Workflow



Tidying and transforming using `dplyr` and `tidyr`

Let's get ready to code...

On your computer, fire up RStudio and install the tidyverse:

```
> install.packages("tidyverse")
```

Then create a new Project in a new folder, and fire up a new script...

On the first line of your script type:

```
library(tidyverse)
```

Then run the script...

Let's now load two datasets that I've created:

```
data1 <- read_csv("https://bit.ly/31Te6HQ")

dataRT <- read_csv("https://bit.ly/2ZflWOr")
```

Tidying and Transforming Data

Imagine we have two rectangular datasets - one (called `data1`) contains a large number of records of individual participants with measures of Working Memory, IQ, and reading comprehension.

If we type into the Console:

```
> data1
```

the data frame is displayed like this…

```
> data1
# A tibble: 10,000 x 4
        id    wm    iq  comp
     <int> <int> <int> <int>
 1       1    43    72    16
 2       2    51   109    18
 3       3    55   107    18
 4       4    38   102    20
 5       5    52   121    17
 6       6    52    92    16
 7       7    47    68    21
 8       8    47    97    23
 9       9    47    93    22
10      10    45   101    17
# … with 9,990 more rows
```

To get more information about the structure of our data frame we can type:

```
> str(data1)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame': 10000 obs. of  4 variables:
 $ id  : int  1 2 3 4 5 6 7 8 9 10 ...
 $ wm  : int  43 51 55 38 52 52 47 47 47 45 ...
 $ iq  : int  72 109 107 102 121 92 68 97 93 101 ...
 $ comp: int  16 18 18 20 17 16 21 23 22 17 ...
```

So we have 10,000 observations with 4 variables associated with each observation - all of them of type integer.

If you ever need help about a function (e.g. str), just type:

```
>?str
```

or

```
>help(str)
```

in the Console window.

Imagine that 48 of these 10,000 people also took part in a reading time experiment and we have their reading data (called `dataRT`) for Simple Sentence and Complex Sentence reading conditions:

```
> str(dataRT)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':  48 obs. of  3 variables:
 $ id             : int  6400 457 8291 4998 2579 9122 1138 5138 5244 3160 ...
 $ simple_sentence : int  1902 1797 2080 1856 1997 1868 2154 1933 1900 1929 ...
 $ complex_sentence: int  2341 2503 2731 2375 2177 2284 2441 2349 2371 2372 ...
```

We are maybe interested in analysing the data of these 48 people in the data frame called `dataRT` but covarying out the effect of IQ captured in our data frame called `data1`.

Problem - how can we combine these two data frames so that we end up with one data frame of 48 people, their reading times plus their individual difference measures?

Manually, in Excel we could open the two data frames as spreadsheets and cut and paste cases where the id number matches…

Probably ok for 48 participants, but what if you had 200 or 2,000?

In R, we can use the `inner_join` function from the `dplyr` package where we join the two data frames matched by ID.

```
> dataRT_all <- inner_join(data1, dataRT, by = (c("id")))
> dataRT_all
     id wm  iq comp simple_sentence complex_sentence
1    95 47  94   19            2154             2441
2   400 45 118   18            1824             2456
3   457 42 100   22            1857             2324
4  1138 41  77   18            1902             2341
5  1587 54  67   21            1844             2320
6  1805 52 109   19            2224             2256
7  1864 57 111   19            1880             2391
8  2006 44 110   19            2091             2456
9  2183 55 125   23            1926             2218
10 2318 51  91   21            1960             2440
```

We can use the assignment symbol `<-` to assign the output of this `inner_join` function to a new variable I’m calling `dataRT_all`. We can ask for the structure of this new data frame using the `str()` function:

```
> dataRT_all <- inner_join(data1, dataRT, by = (c("id")))
> str(dataRT_all)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':  48 obs. of  6 variables:
 $ id              : int  95 400 457 1138 1587 1805 1864 2006 2183 2318 ...
 $ wm              : int  47 45 42 41 54 52 57 44 55 51 ...
 $ iq              : int  94 118 100 77 67 109 111 110 125 91 ...
 $ comp            : int  19 18 22 18 21 19 19 19 23 21 ...
 $ simple_sentence : int  2154 1824 1857 1902 1844 2224 1880 2091 1926 1960 ...
 $ complex_sentence: int  2441 2456 2324 2341 2320 2256 2391 2456 2218 2440 ...
```

So we have created a new data frame of 48 participants consisting of their reading times and their individual difference measures from two separate (and different sized) data frames…with one line of code…

Now imagine we find the distributions of reading times for our two conditions are positively skewed (and we discover the residuals are non-normal). We could log transform these two columns and have two new columns in our data frame - let's call them `log_simple` and `log_complex`. We can use the `mutate` function in the `dplyr` package to create two new columns.

```
> data_transformed <- mutate(dataRT_all, log_simple = log(simple_sentence),
log_complex = log(complex_sentence))
> data_transformed
# A tibble: 48 x 8
      id    wm    iq  comp simple_sentence complex_sentence log_simple log_complex
   <int> <int> <int> <int>           <int>            <int>      <dbl>       <dbl>
 1    95    47    94    19            1960             2440       7.58        7.80
 2   400    45   118    18            2186             2200       7.69        7.70
 3   457    42   100    22            1797             2503       7.49        7.83
 4  1138    41    77    18            2154             2441       7.68        7.80
 5  1587    54    67    21            1936             2395       7.57        7.78
 6  1805    52   109    19            1864             2560       7.53        7.85
 7  1864    57   111    19            1930             2540       7.57        7.84
 8  2006    44   110    19            2230             2267       7.71        7.73
 9  2183    55   125    23            1857             2324       7.53        7.75
10  2318    51    91    21            1918             2739       7.56        7.92
# … with 38 more rows
```

Perhaps we have a reason to exclude a particular participant - number 2006 for example. We can use the filter function in `dplyr` to keep those participants where the ID number does not equal 2006.

```
filtered_data <- filter(data_transformed, id != 2006)
```

!= stands for “not equal to”- here are other useful logical operators in R:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to

We can now apply our logical vector to our `dataRT_all` data frame and create a new filtered data frame (which I am calling `filtered_data`):

```
> filtered_data <- filter(data_transformed, id != 2006)
> filtered_data
# A tibble: 47 x 8
      id    wm    iq  comp simple_sentence complex_sentence log_simple log_complex
   <int> <int> <int> <int>           <int>            <int>      <dbl>       <dbl>
 1    95    47    94    19            1960             2440       7.58        7.80
 2   400    45   118    18            2186             2200       7.69        7.70
 3   457    42   100    22            1797             2503       7.49        7.83
 4  1138    41    77    18            2154             2441       7.68        7.80
 5  1587    54    67    21            1936             2395       7.57        7.78
 6  1805    52   109    19            1864             2560       7.53        7.85
 7  1864    57   111    19            1930             2540       7.57        7.84
 8  2183    55   125    23            1857             2324       7.53        7.75
 9  2318    51    91    21            1918             2739       7.56        7.92
10  2324    43   120    20            1891             2426       7.54        7.79
# … with 37 more rows
```

We could then run an ANCOVA over the log transformed RTs while covarying out the individual participant effects…

Problem - imagine our data are in the wrong 'shape' - they are in Wide format (each row is one *participant*) but we need them in Long or Tidy format (each row is one *observation*).

In SPSS, most data will be in Wide format with each experimental condition its own column:

```
> dataRT
# A tibble: 48 x 3
      id simple_sentence complex_sentence
   <int>           <int>            <int>
 1  6400            1902             2341
 2   457            1797             2503
 3  8291            2080             2731
 4  4998            1856             2375
 5  2579            1997             2177
 6  9122            1868             2284
 7  1138            2154             2441
 8  5138            1933             2349
 9  5244            1900             2371
10  3160            1929             2372
# … with 38 more rows
```

For many analyses in R, data need to be in Long format with each row being one observation. So, we want to transform our `dataRT` data frame so it looks like this:

id	condition	rt
…	…	…

To do this we can use the `gather()` function in the `tidyr` package.

```
> data_long <- gather(dataRT, "condition", "rt", c("simple_sentence",
"complex_sentence"))
```

The first parameter is the name of the data frame we want to reshape, the second is the name of the new ‘Key’ column, the third is the name of the new value column and the fourth the names of the columns we want to collapse.

We can use this to create a new data frame called `data_long` which looks like this:

```
> data_long <- gather(dataRT, "condition", "rt", c("simple_sentence", 
"complex_sentence"))
> data_long
# A tibble: 96 x 3
      id condition          rt
   <int> <chr>           <int>
 1  6400 simple_sentence  1902
 2   457 simple_sentence  1797
 3  8291 simple_sentence  2080
 4  4998 simple_sentence  1856
 5  2579 simple_sentence  1997
 6  9122 simple_sentence  1868
 7  1138 simple_sentence  2154
 8  5138 simple_sentence  1933
 9  5244 simple_sentence  1900
10  3160 simple_sentence  1929
# … with 86 more rows
```

And in reverse we can use the `spread()` function to go from Long to Wide data format:

```
> data_wide <- spread(data_long, "condition", "rt")
> data_wide
# A tibble: 48 x 3
      id complex_sentence simple_sentence
   <int>            <int>           <int>
 1    95             2440            1960
 2   400             2200            2186
 3   457             2503            1797
 4  1138             2441            2154
 5  1587             2395            1936
 6  1805             2560            1864
 7  1864             2540            1930
 8  2006             2267            2230
 9  2183             2324            1857
10  2318             2739            1918
# … with 38 more rows
```

We’re now back to where we started with data in Wide format:

This is just a small example of functions in the `dplyr` and `tidyr` packages that allow you to tidy, transform, and reshape your data. All of your code for doing this should appear at the start of your analysis script so that others (and you in 5 years or 5 days time) can see exactly what you did.

This allows for fully reproducible data preparation in the first part of your analysis workflow (important for Open Science and reproducibility).

Generating Descriptives - using `dplyr`

- You can use the `group_by()` and `summarise()` functions in the `dplyr` package to generate descriptives.

- In the following example, we are also using the pipe operator `%>%` which passes a value into an expression or function call from left to right:

```
> data_long %>%
   group_by(condition) %>%
   summarise(Mean = mean(rt), Min = min(rt), Max = max(rt), SD = sd(rt))
# A tibble: 2 x 5
  condition         Mean   Min   Max    SD
  <chr>            <dbl> <int> <int> <dbl>
1 complex_sentence 2405.  2177  2739  132.
2 simple_sentence  1957.  1694  2356  147.
```

Tidying Up Some Real World Messy Data

Let's load another dataset that I created:

```
my_data <- read_csv("https://bit.ly/2KPZEe9")
```

Tidying Up Some Real World Messy Data

- We ran a reaction time experiment with 24 participants and 4 conditions - they are numbered 1-4 in our datafile.

```
> my_data
# A tibble: 96 x 3
   participant condition    rt
         <int>     <int> <int>
 1           1         1   879
 2           1         2  1027
 3           1         3  1108
 4           1         4   765
 5           2         1  1042
 6           2         2  1050
 7           2         3   942
 8           2         4   945
 9           3         1   943
10           3         2   910
# … with 86 more rows
```

- But actually it was a repeated measures design where we had one factor (Prime Type) with two levels (A vs. B) and a second factor (Target Type) with two levels (A vs. B)

- We want to recode our data frame so it better matches our experimental design.

- First we need to recode our 4 conditions like this:

```
# Recode condition columns follows:
# Condition 1 = prime A, target A
# Condition 2 = prime A, target B
# Condition 3 = prime B, target A
# Condition 4 = prime B, target B

my_data <- my_data %>%
            mutate(condition = recode(condition,
           "1" = "primeA_targetA",
           "2" = "primeA_targetB",
           "3" = "primeB_targetA",
           "4" = "primeB_targetB"))
```

- Now our data frame looks like this:

```
> my_data
# A tibble: 96 x 3
   participant condition          rt
         <int> <chr>           <int>
 1           1 primeA_targetA   879
 2           1 primeA_targetB  1027
 3           1 primeB_targetA  1108
 4           1 primeB_targetB   765
 5           2 primeA_targetA  1042
 6           2 primeA_targetB  1050
 7           2 primeB_targetA   942
 8           2 primeB_targetB   945
 9           3 primeA_targetA   943
10           3 primeA_targetB   910
# … with 86 more rows
```

- We then need to separate out our Condition column into two - one for our first factor (Prime), and one for our second factor (Target).

```
> my_data <- separate(my_data, col = "condition", into = c("prime",
"target"), sep = "_")
> my_data
# A tibble: 96 x 4
   participant prime  target     rt
         <int> <chr>  <chr>   <int>
 1           1 primeA targetA   879
 2           1 primeA targetB  1027
 3           1 primeB targetA  1108
 4           1 primeB targetB   765
 5           2 primeA targetA  1042
 6           2 primeA targetB  1050
 7           2 primeB targetA   942
 8           2 primeB targetB   945
 9           3 primeA targetA   943
10           3 primeA targetB   910
# … with 86 more rows
```

- This is looking good - we now have our two factors coded separately and our data are in tidy format (i.e., one observation per row).

- How long would this have taken you in Excel? Would it have been reproducible?

- Perhaps we want to go from the data in long format, to wide format.

```
> my_data <- unite(my_data, col = "condition", c("prime", "target"), sep = "_")
> wide_data <- spread(my_data, key = "condition", value = "rt")
> wide_data
# A tibble: 24 x 5
   participant primeA_targetA primeA_targetB primeB_targetA primeB_targetB
         <int>          <int>          <int>          <int>          <int>
 1           1            879           1027           1108            765
 2           2           1042           1050            942            945
 3           3            943            910            952            900
 4           4            922           1006           1095            988
 5           5            948            908            916           1241
 6           6           1013            950            955           1045
 7           7            930            855           1057            897
 8           8            998            906           1110            952
 9           9            929            949            837            883
10          10            781            865            970            953
# … with 14 more rows
```

- No matter what format your data are in originally, you can use functions from the `dplyr` and `tidyr` packages to quickly get it into whatever format you need for analysis.

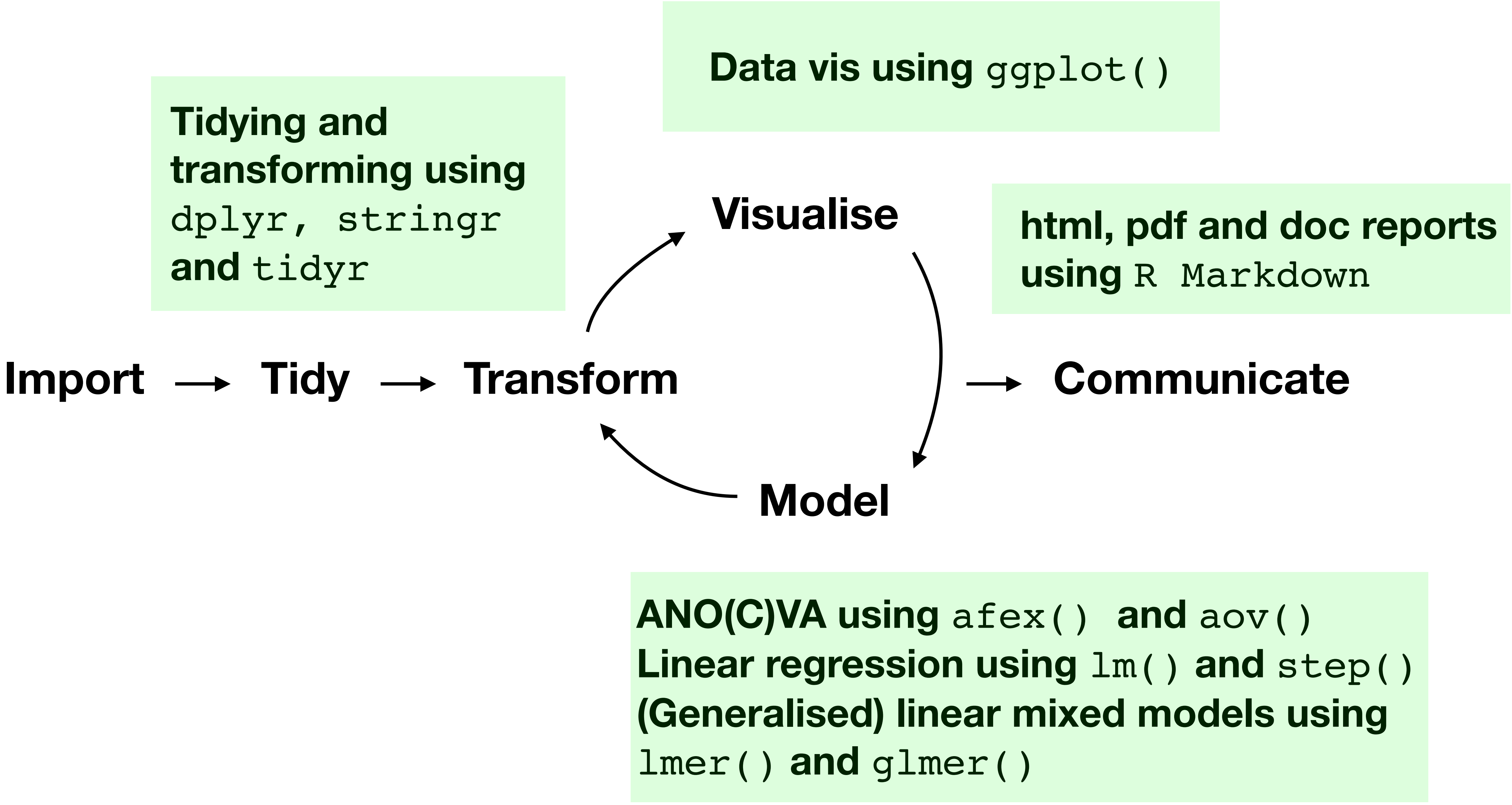
Or using the pipe…

- We could alternatively have used the `%>%` operator to combine all the last few operations which would have avoided the need to create temporary variables.

```
my_data %>%
   unite(col = "condition", c("prime", "target"), sep = "_") %>%
   spread(key = "condition", value = "rt")
```

- Take `my_data` and then unite and then spread…

A Reproducible Workflow



Data vis using `ggplot()`

Tidying and transforming using `dplyr, stringr` and `tidyr`

html, pdf and doc reports using `R Markdown`

Import → Tidy → Transform

Visualise

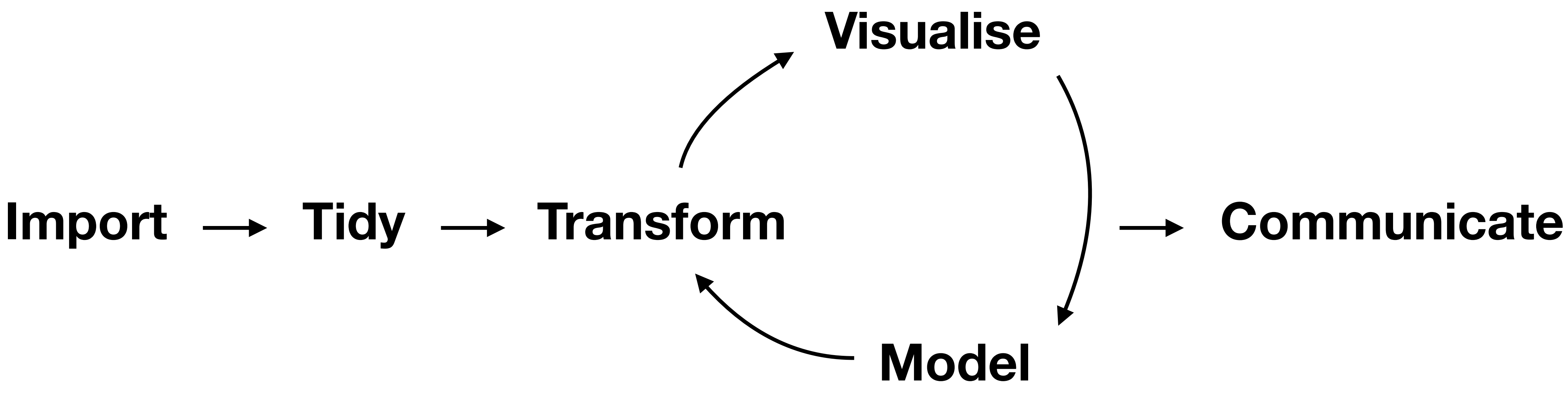
Model

→ Communicate

ANO(C)VA using `afex()` and `aov()`
Linear regression using `lm()` and `step()`
(Generalised) linear mixed models using `lmer()` and `glmer()`

A Reproducible Workflow

Data vis using `ggplot()`

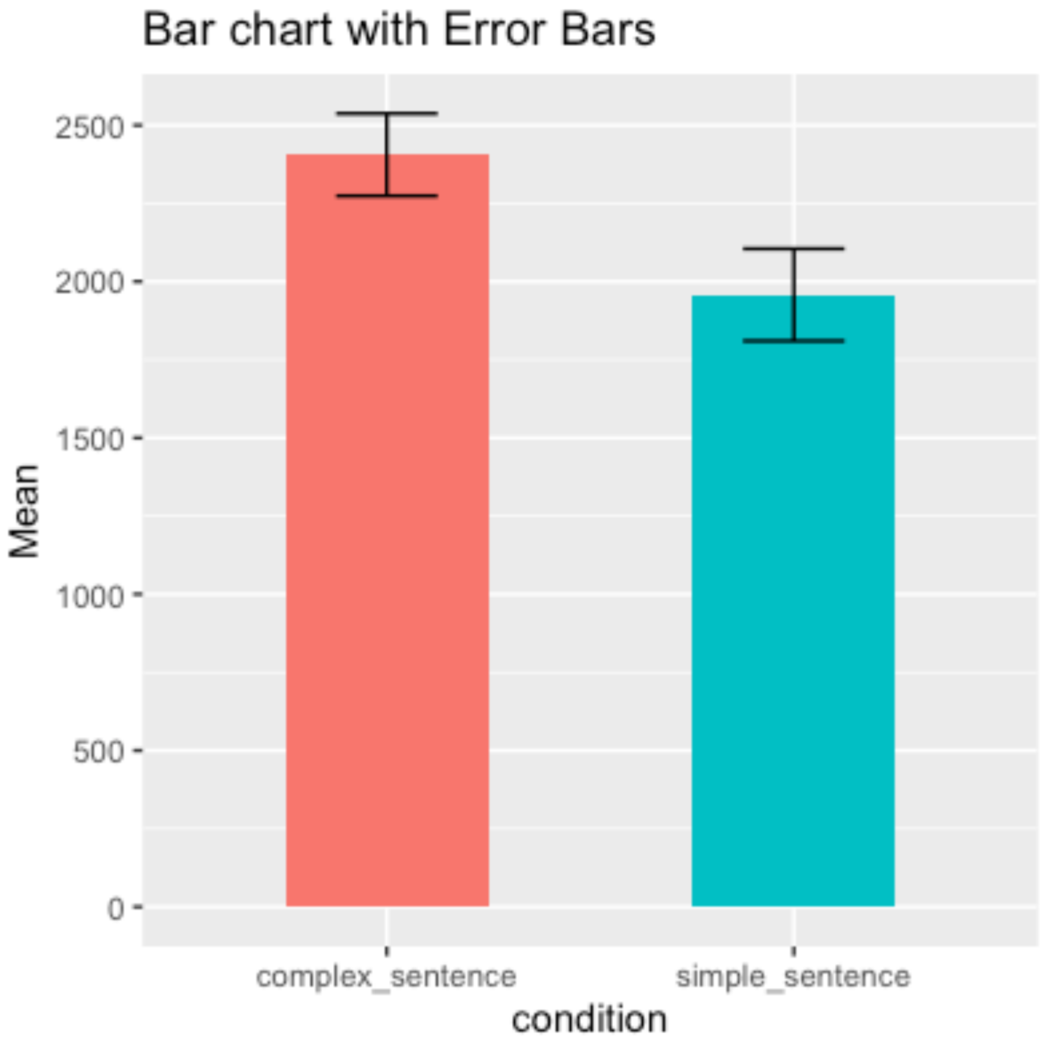


Visualising Your Data

- R has a number of in built (base) graphics functions, but you’re more likely to use functions from within the `ggplot2` packages. `ggplot2` is part of the `tidyverse` so if you have used `library(tidyverse)` then `ggplot2` will already be loaded.

```
> library(ggplot2)
```

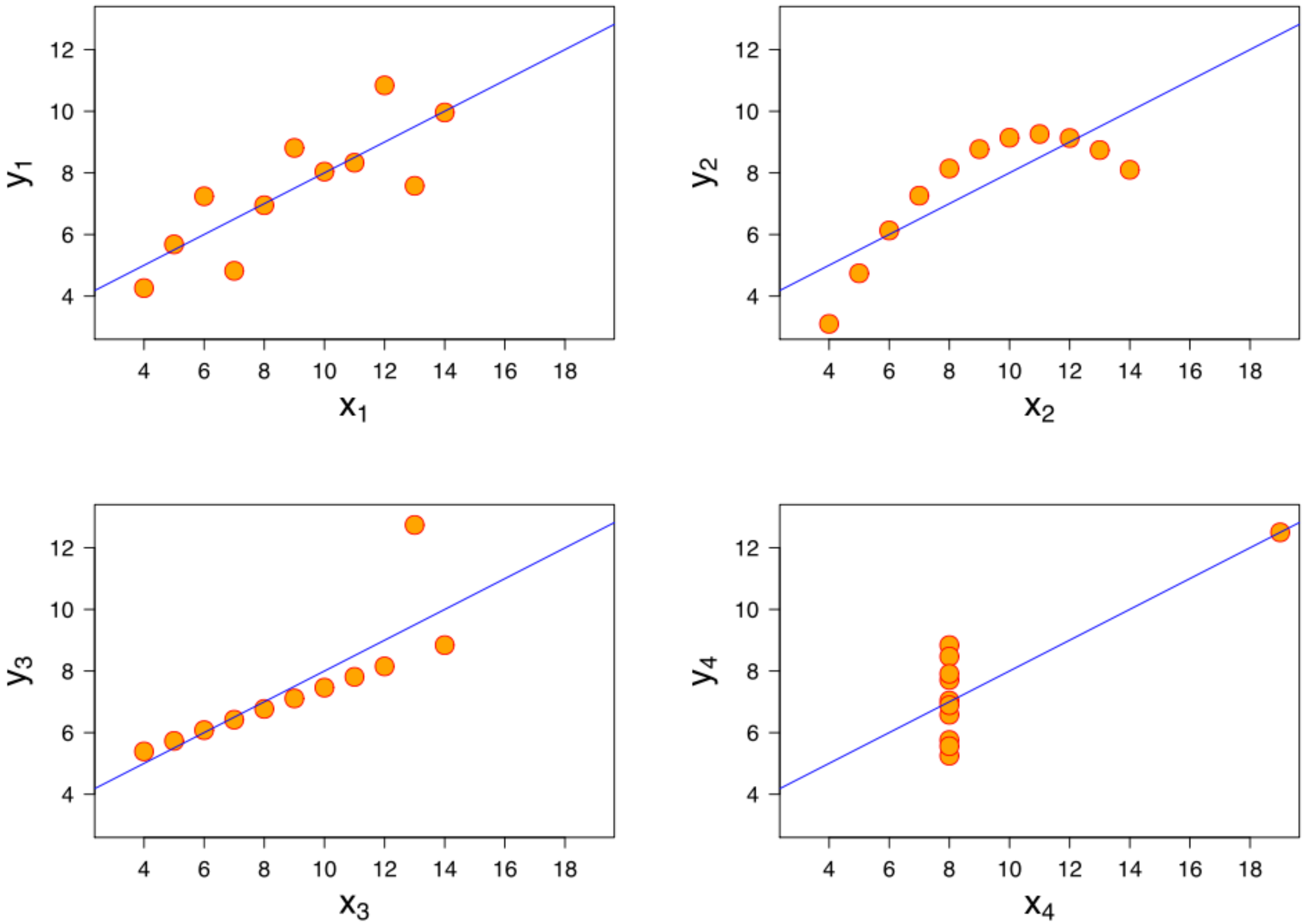
Bar Graphs (bleurgh!)



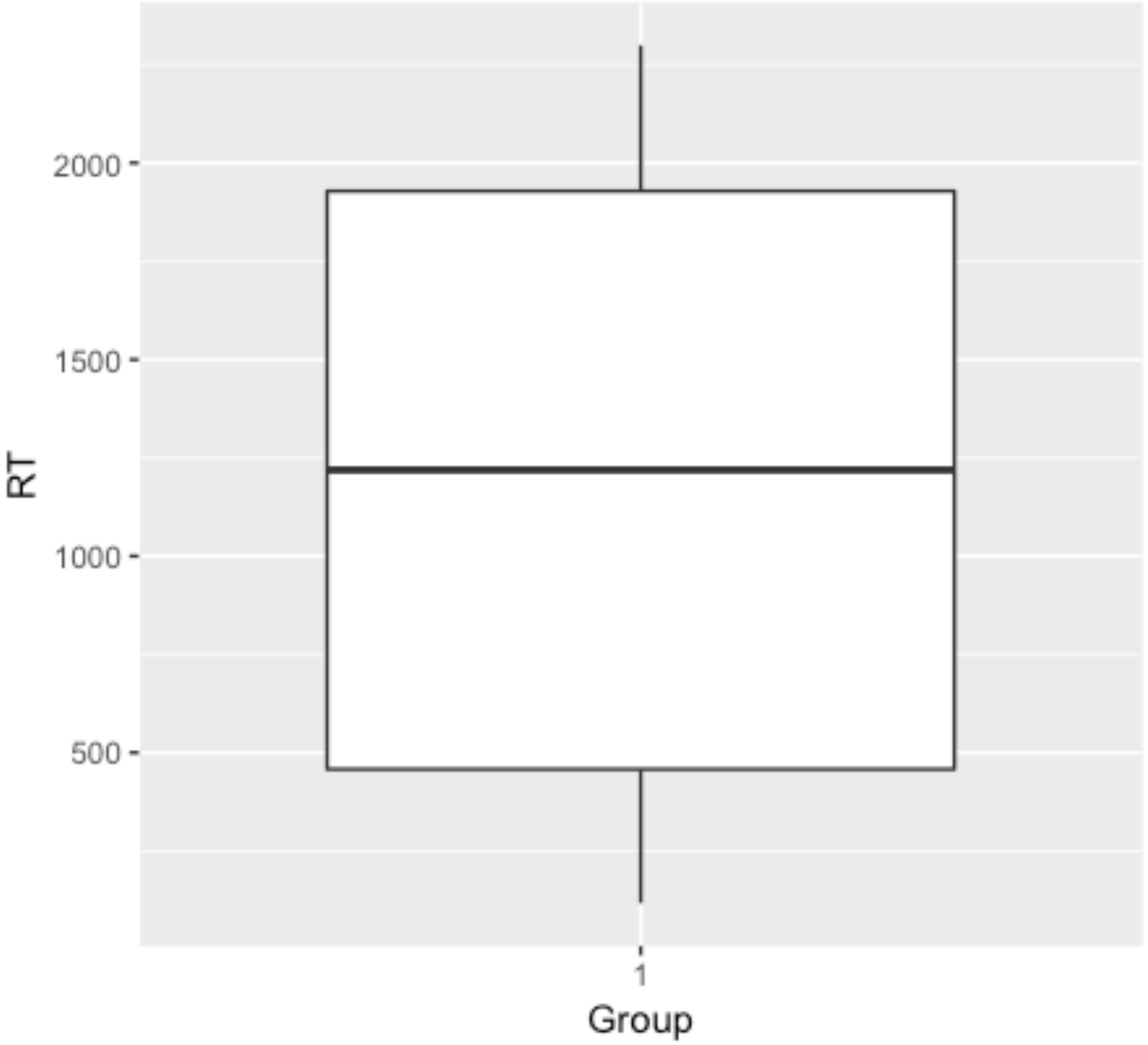
Bar graphs tend to be quite limited in terms of what they communicate. Here they communicate the means for levels of a factor and information about variance. But they don't tell us anything about the *distribution* of the data.

```
data_summ <- data_long %>% group_by(condition) %>% summarise(Mean = mean(rt), sd = sd(rt))

ggplot(data_summ, aes (x = condition, y = Mean, group = condition,
                       fill = condition, ymin = Mean - sd, ymax = Mean + sd)) +
  geom_bar(stat = "identity", width = .5) +
  geom_errorbar(width = .25) +
  ggtitle("Bar chart with Error Bars") +
  guides(fill = FALSE)
```

Anscombe's Quartet

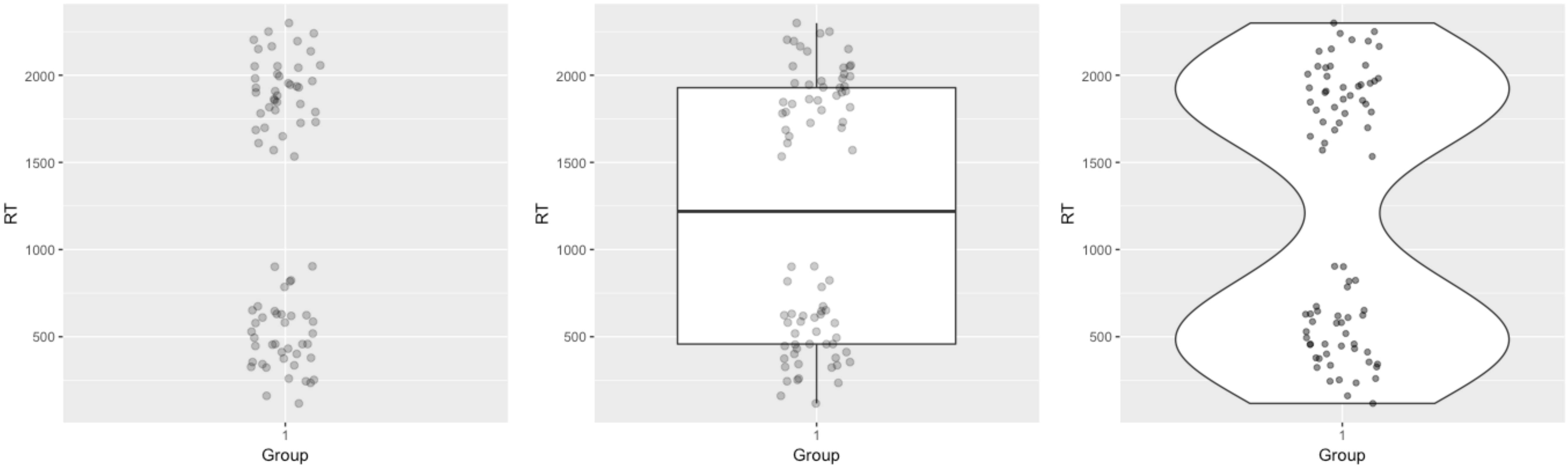


Plots Based on Aggregated Data Can Mislead…



You might make one set of inferences based on this boxplot - maybe a median around 1,250 with the 25th and 75th percentiles being ~480 to ~1,980…

But look more closely at the actual data…

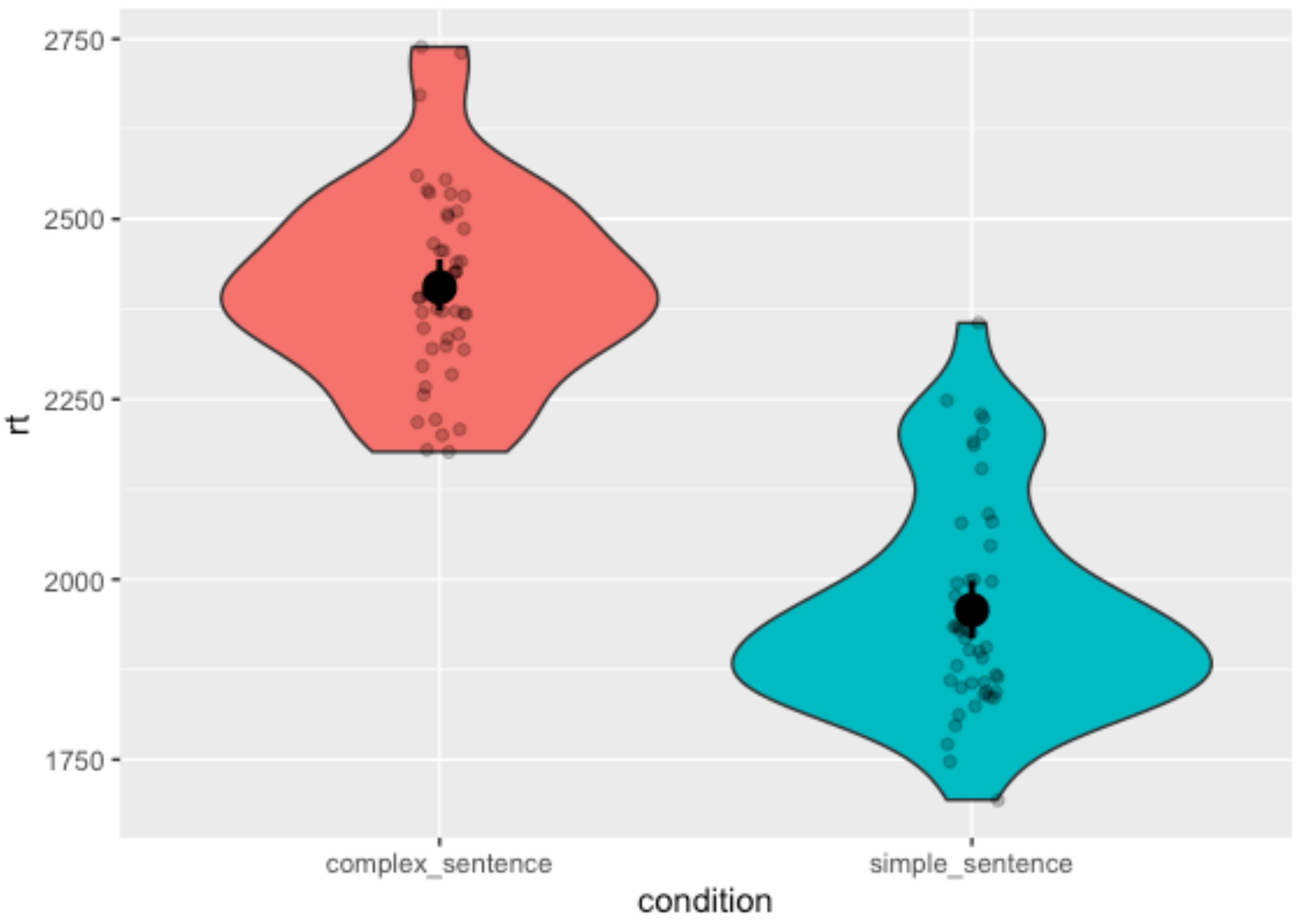


The data are clearly bimodal with no actual data point near the mean. **Distribution shape matters** and we need to capture that in our data visualisations.

You try...

```
data2 <- read_csv("https://bit.ly/31UgcXT")

ggplot(data2, aes(x = group, y = rt)) +
  geom_boxplot()

ggplot(data2, aes(x = group, y = rt)) +
  geom_jitter(size = 2, width = .1, alpha = .25)

ggplot(data2, aes(x = group, y = rt)) +
  geom_boxplot() +
  geom_jitter(size = 2, width = .1, alpha = .25)

ggplot(data2, aes(x = group, y = rt)) +
   geom_violin() +
   geom_jitter(width = .1, alpha = .5)
```

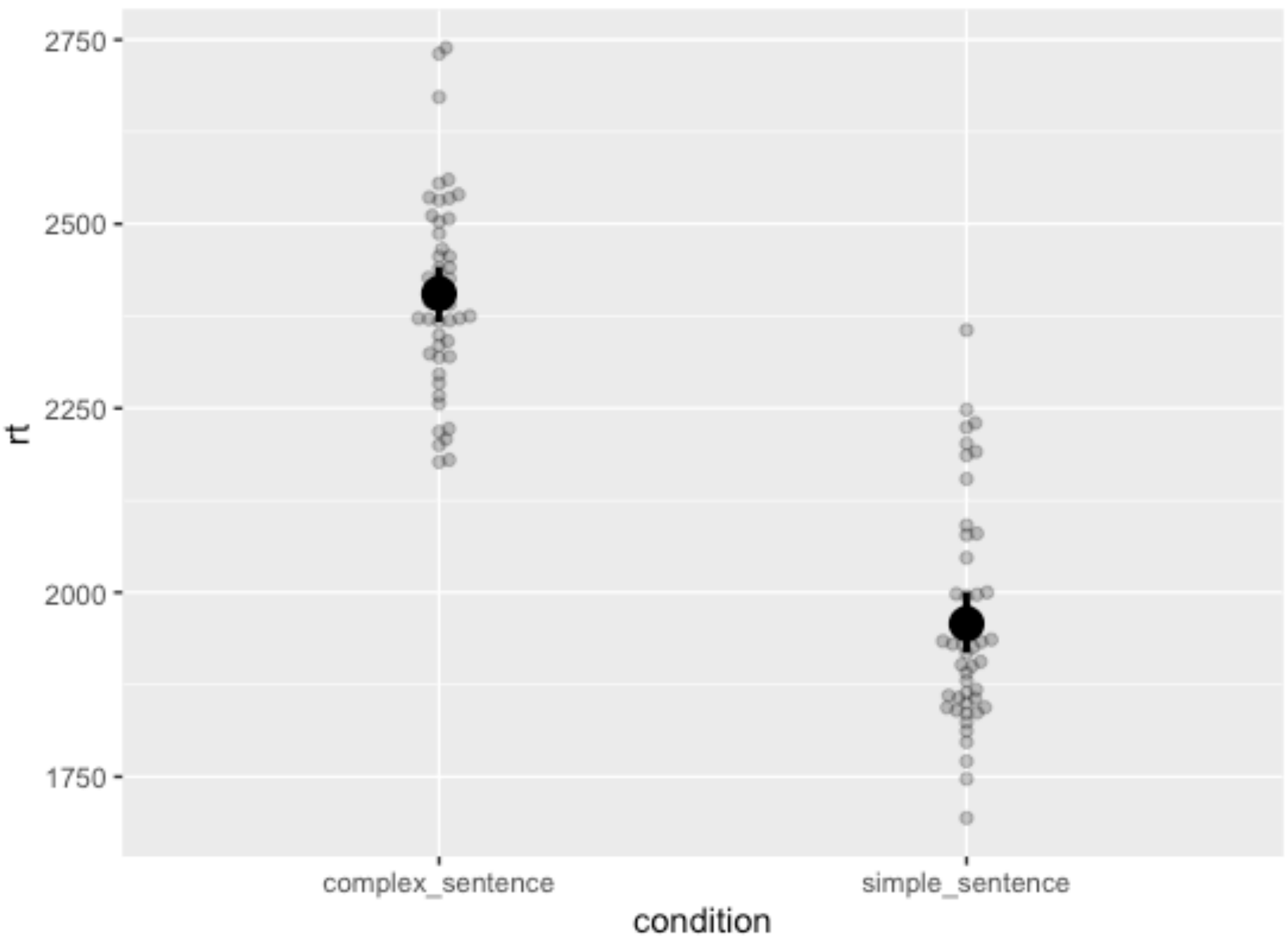
Violin Plots



Violin plots tell us about the distribution of the data. The width at any point corresponds to the *density* of the data at that value.

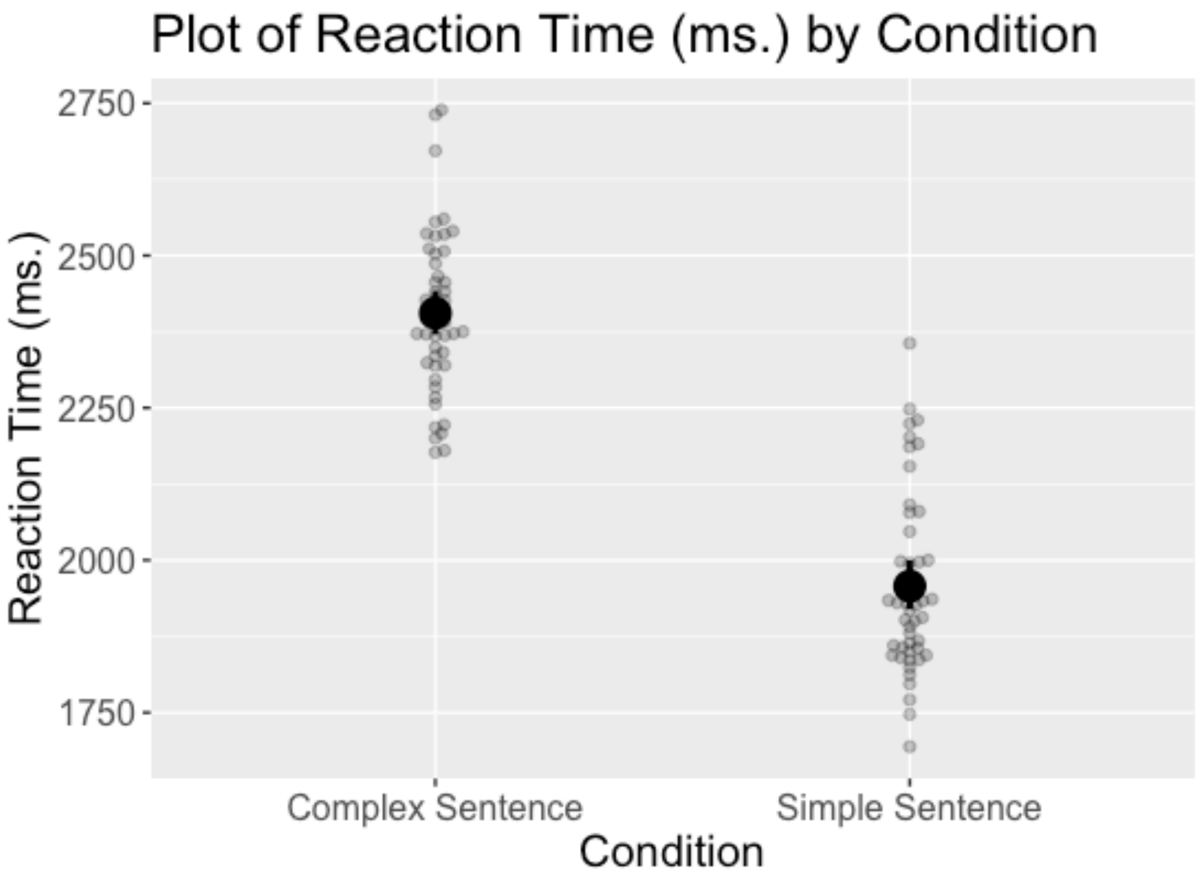
```
ggplot(data_long, aes(x = condition, y = rt,
   group = condition, fill = condition)) +
   geom_violin() +
   geom_jitter(alpha = .25, position = position_jitter(0.05)) +
   guides(colour = FALSE, fill = FALSE) +
   stat_summary(fun.data = "mean_cl_boot", colour = "black", size = 1)
```

Beeswarm Plots



```
ggplot(data_long, aes(x = condition, y = rt, group = condition, fill = condition)) +
  geom_beeswarm(alpha = .25) +
  guides(colour = FALSE, fill = FALSE) +
  stat_summary(fun.data = "mean_cl_boot", colour = "black", size = 1)
```

Tidying up our labels

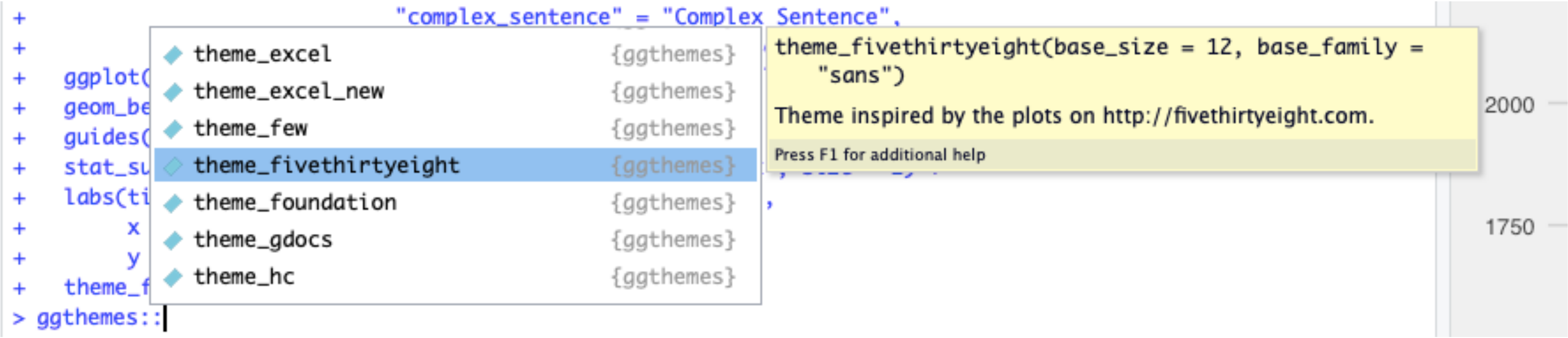


```
data_long %>%
  mutate(Condition = recode(condition,
                            "complex_sentence" = "Complex Sentence",
                            "simple_sentence" = "Simple Sentence")) %>%
  ggplot(aes(x = Condition, y = rt, group = Condition, fill = Condition)) +
  geom_beeswarm(alpha = .25) +
  guides(colour = FALSE, fill = FALSE) +
  stat_summary(fun.data = "mean_cl_boot", colour = "black", size = 1) +
  labs(title = "Plot of Reaction Time (ms.) by Condition",
       x = "Condition",
       y = "Reaction Time (ms.)") +
  theme(text = element_text(size = 15))
```

Themes

- The ggthemes package has lots of pre-built ggplot themes that we can apply to our ggplot visualisations.

```
>library(ggthemes)
```



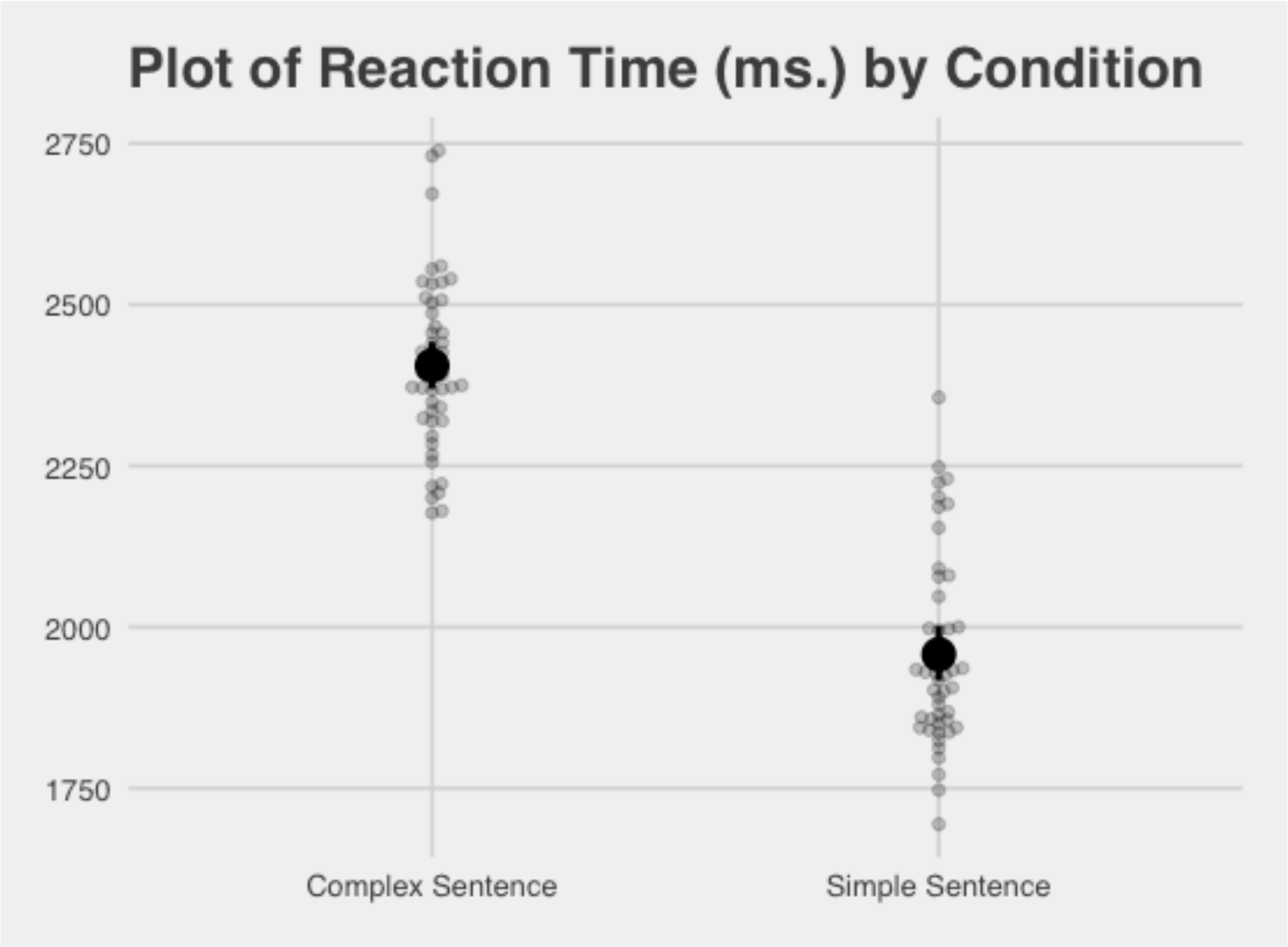
```
+                         "complex_sentence" = "Complex Sentence",
+
+   ggplot(
+   geom_be
+   guides(
+   stat_su
+   labs(ti
+         x
+         y
+   theme_f
> ggthemes::
```

```
theme_excel            {ggthemes}
theme_excel_new        {ggthemes}
theme_few              {ggthemes}
theme_fivethirtyeight  {ggthemes}
theme_foundation       {ggthemes}
theme_gdocs            {ggthemes}
theme_hc               {ggthemes}
```

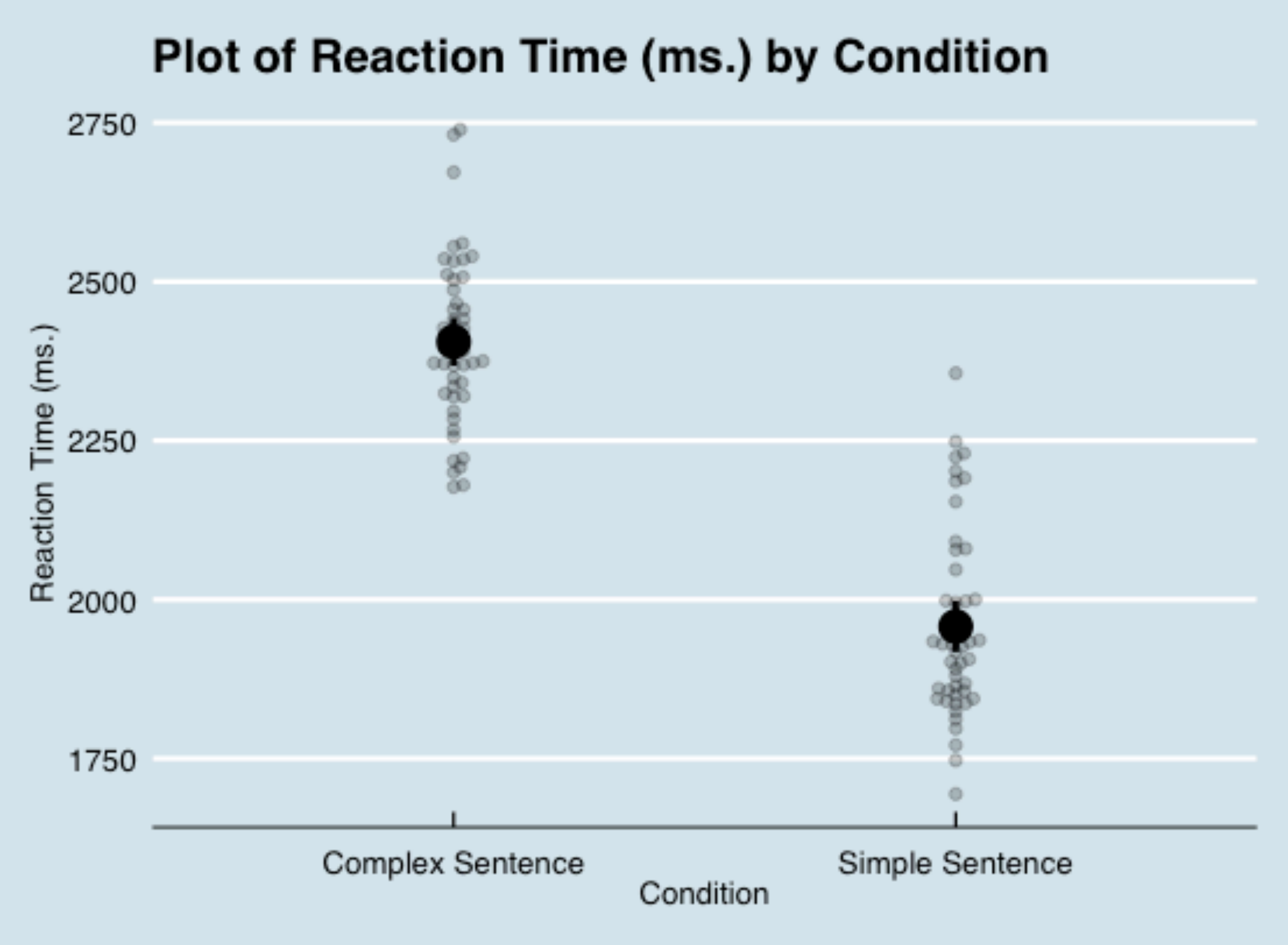
```
theme_fivethirtyeight(base_size = 12, base_family =
    "sans")
```

Theme inspired by the plots on http://fivethirtyeight.com.

Press F1 for additional help

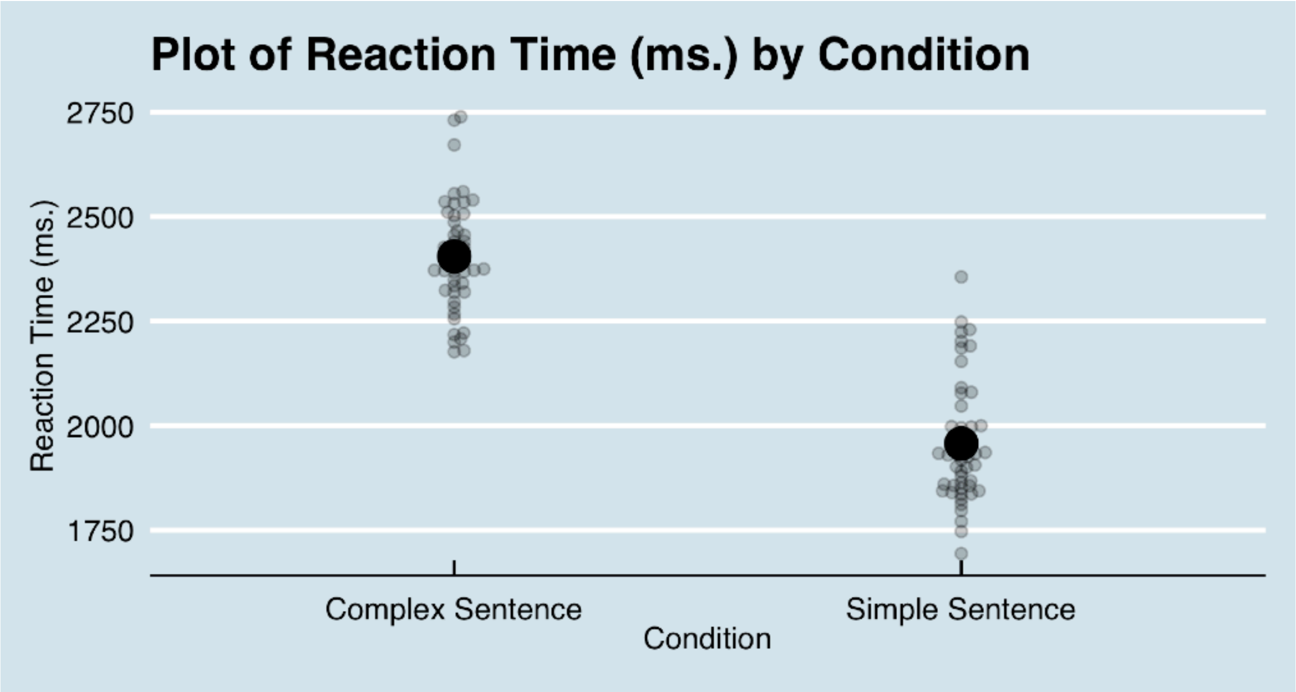


```
data_long %>%
  mutate(Condition = recode(condition,
                            "complex_sentence" = "Complex Sentence",
                            "simple_sentence" = "Simple Sentence")) %>%
  ggplot(aes(x = Condition, y = rt, group = Condition, fill = Condition)) +
  geom_beeswarm(alpha = .25) +
  guides(colour = FALSE, fill = FALSE) +
  stat_summary(fun.data = "mean_cl_boot", colour = "black", size = 1) +
  labs(title = "Plot of Reaction Time (ms.) by Condition",
       x = "Condition",
       y = "Reaction Time (ms.)") +
  theme_fivethirtyeight()
```



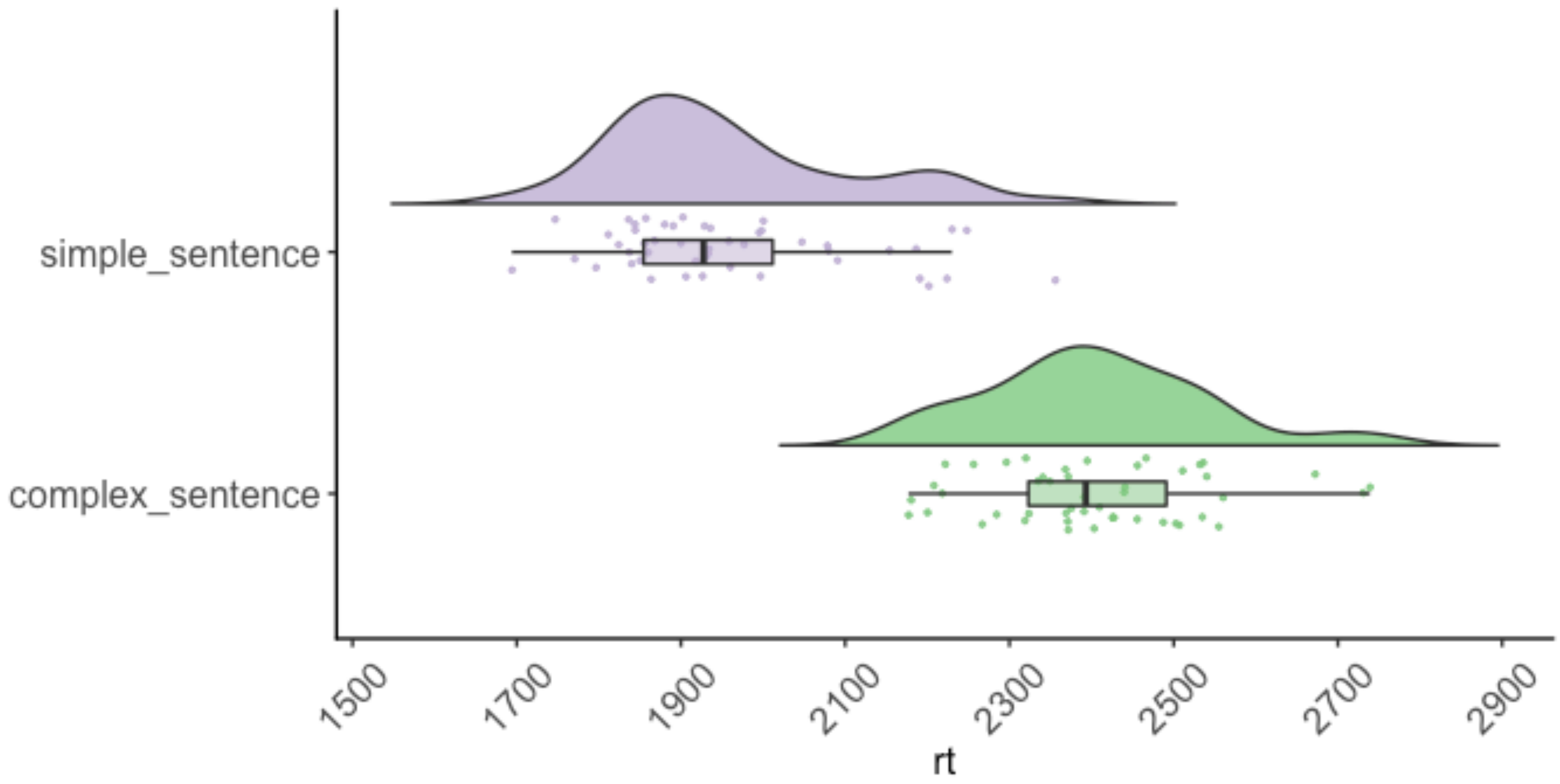
```
data_long %>%
  mutate(Condition = recode(condition,
                            "complex_sentence" = "Complex Sentence",
                            "simple_sentence" = "Simple Sentence")) %>%
  ggplot(aes(x = Condition, y = rt, group = Condition, fill = Condition)) +
  geom_beeswarm(alpha = .25) +
  guides(colour = FALSE, fill = FALSE) +
  stat_summary(fun.data = "mean_cl_boot", colour = "black", size = 1) +
  labs(title = "Plot of Reaction Time (ms.) by Condition",
       x = "Condition",
       y = "Reaction Time (ms.)") +
  theme_economist()
```

Changing Plot Dimensions



```
my_plot <- data_long %>%
  mutate(Condition = recode(condition,
                           "complex_sentence" = "Complex Sentence",
                           "simple_sentence" = "Simple Sentence")) %>%
  ggplot(aes(x = Condition, y = rt, group = Condition, fill = Condition)) +
  geom_beeswarm(alpha = .25) +
  guides(colour = FALSE, fill = FALSE) +
  stat_summary(fun.data = "mean_cl_boot", colour = "black", size = 1) +
  labs(title = "Plot of Reaction Time (ms.) by Condition",
       x = "Condition",
       y = "Reaction Time (ms.)") +
  theme_economist()

ggsave("my_plot.png", my_plot, height = 8, width = 15, units = "cm")
```

Raincloud Plots



Developed by Micah Allen (UCL), raincloud plots allow you to see the raw data, and the shape of the distribution alongside a box plot (capturing the median, 25th and 75th percentiles as hinges, and 1.5 * IQR from the hinges as the whisker length.)

A Variety of Plots Using the Same Dataset

We're going to use the built-in dataset 'mpg' to build a variety of plots. First, let's find out about the data by using the `head` function to view the first part of the data.

```
> head(mpg)
# A tibble: 6 x 11
  manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
  <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compact
2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compact
3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compact
4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compact
5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compact
6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p     compact
```

We can explore the data further by asking for all the possibilities in each column using the `unique` function. For example, we can check to see how many different types of cars there are. Note that the $ after the dataset name allows us to refer to a column in the mpg dataset.

```
> unique(mpg$manufacturer)
 [1] "audi"       "chevrolet"  "dodge"      "ford"       "honda"      "hyundai"    "jeep"
 [8] "land rover" "lincoln"    "mercury"    "nissan"     "pontiac"    "subaru"     "toyota"
[15] "volkswagen"
```

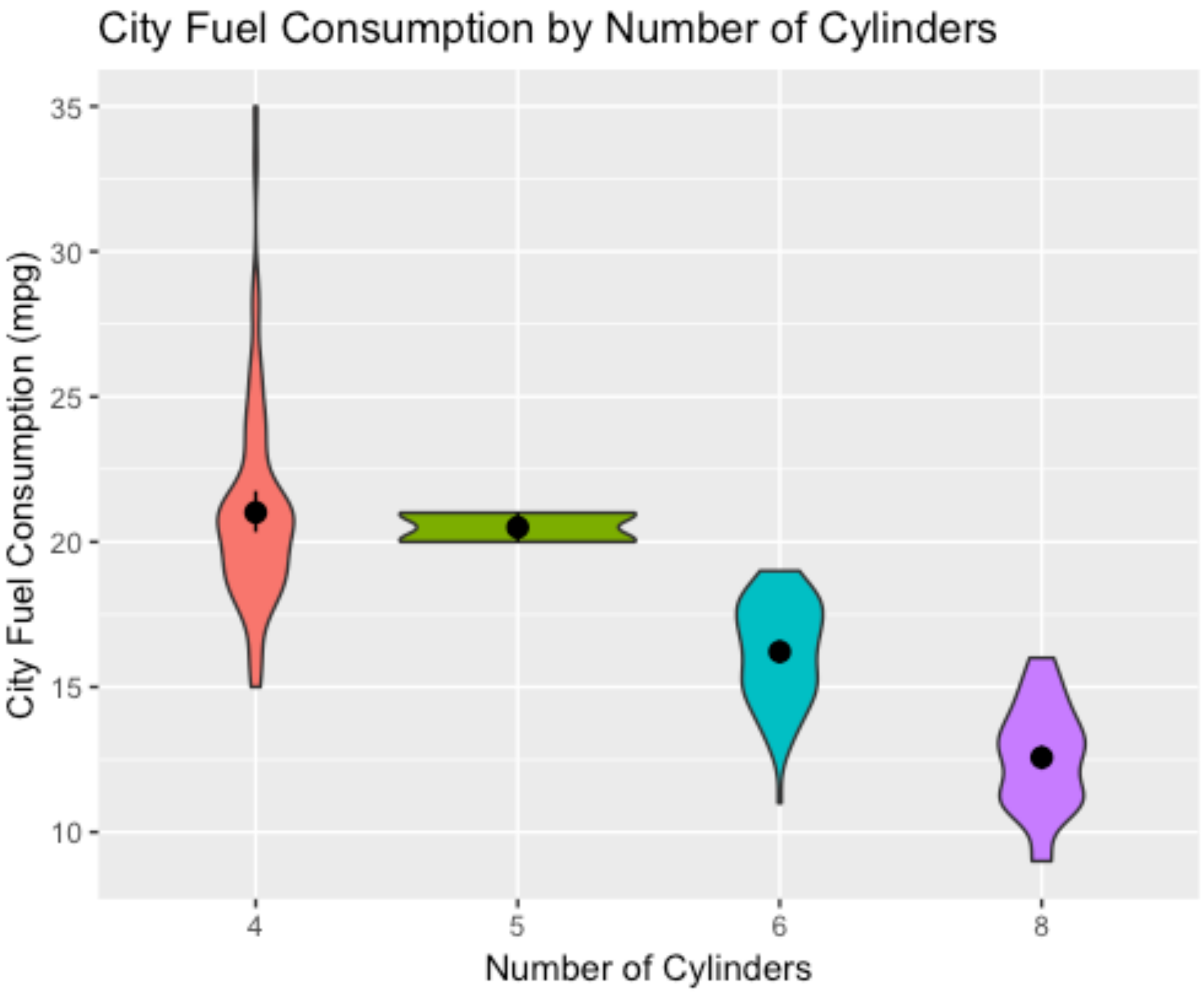
We can use the `length` function to give us the total number of unique possibilities:

```
> length(unique(mpg$manufacturer))
[1] 15
```

Let’s look at a whole bunch of different visualisations using the mpg data set…

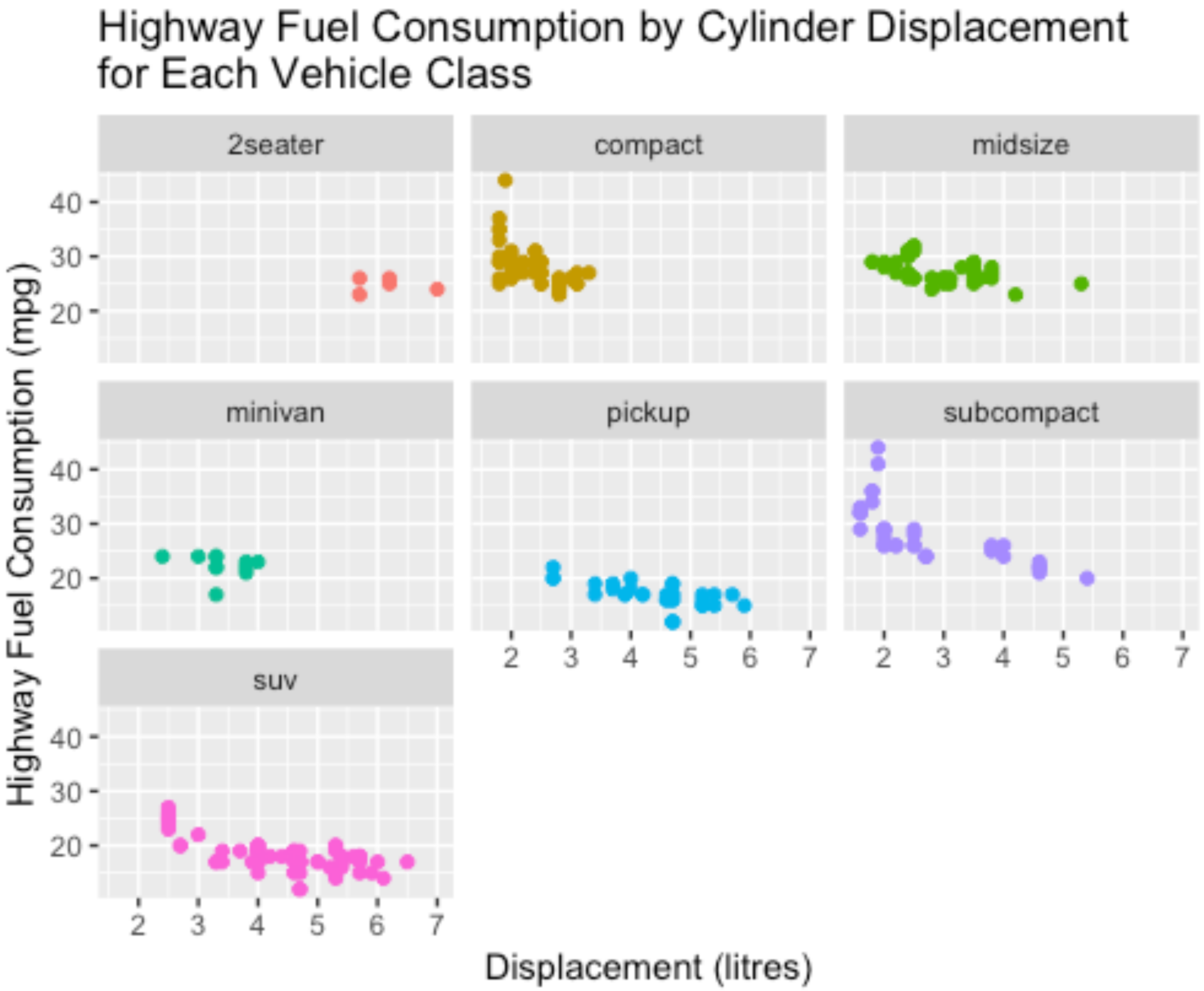
This illustrates the idea that there is not one ‘correct’ way to visualise the data, but rather that your choice of visualisation will be influenced by the question you’re investigating, or the story you’re wanting to tell…

Violin Plots



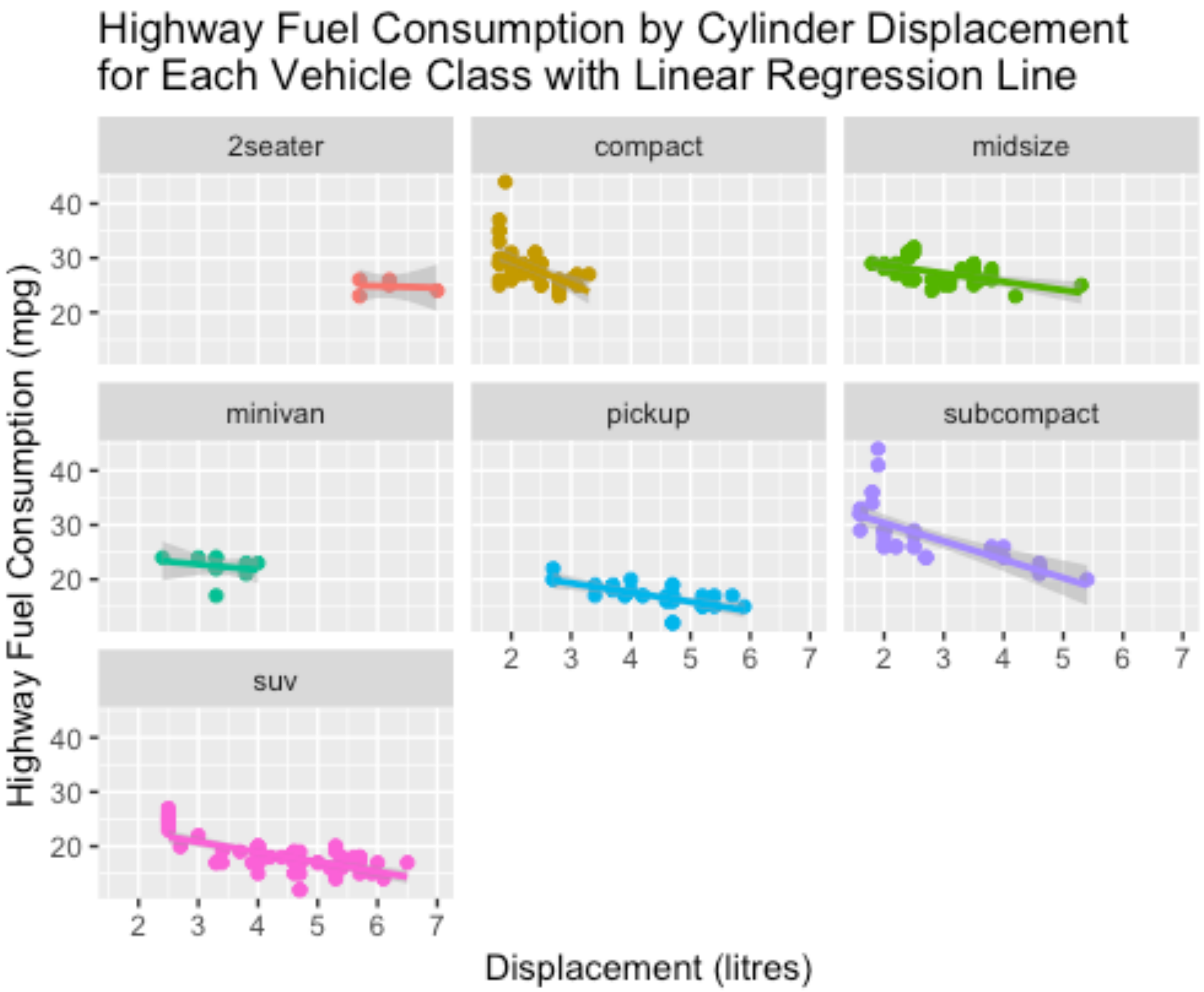
```
ggplot(mpg, aes(x = factor(cyl), y = cty, fill = factor(cyl))) +
  geom_violin() +
  guides(colour = FALSE, fill = FALSE) +
  stat_summary(fun.data = mean_cl_boot, colour = "black", size = .5) +
  labs(title = "City Fuel Consumption by Number of Cylinders",
       x = "Number of Cylinders",
       y = "City Fuel Consumption (mpg)")
```

Faceting

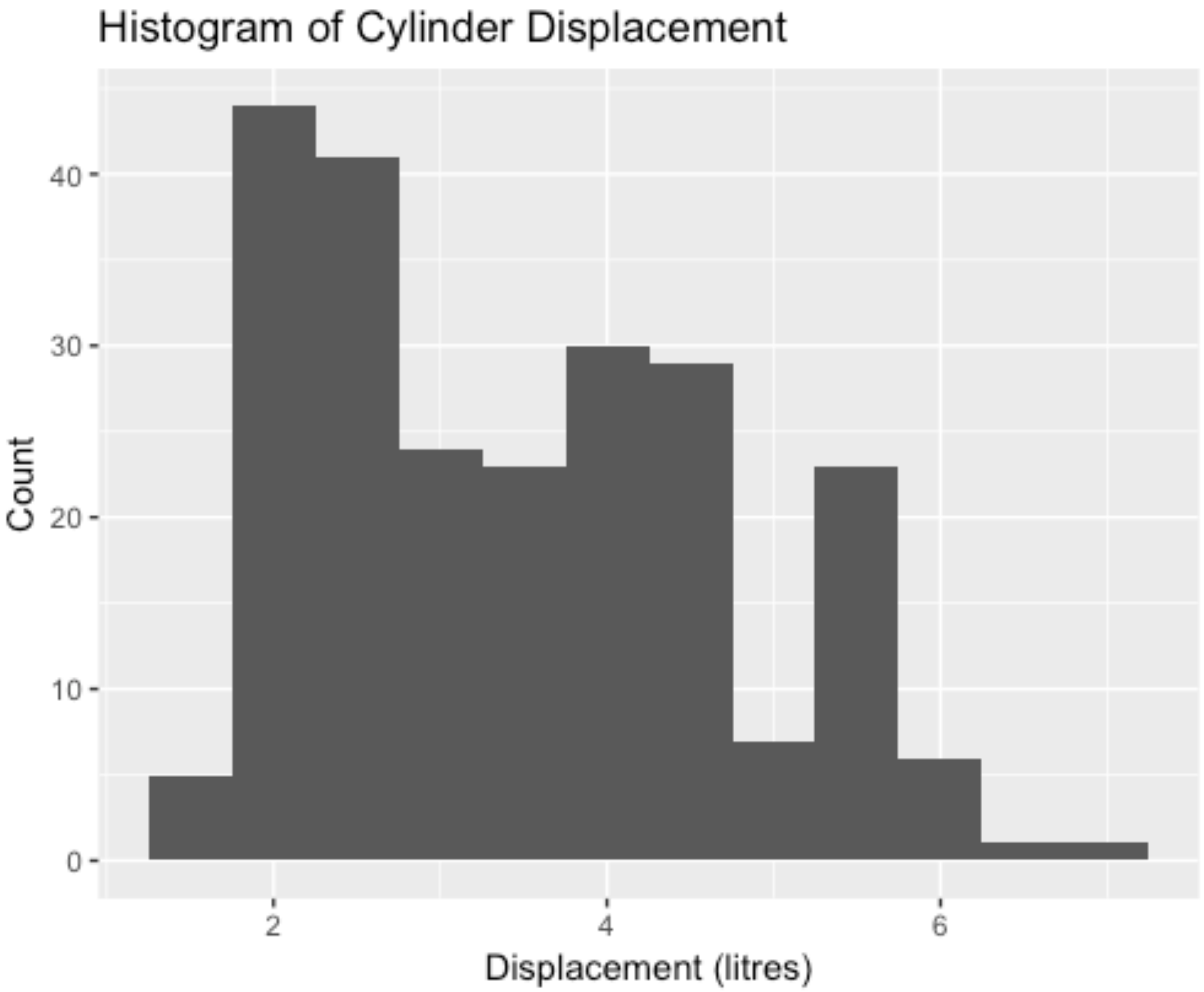


```
ggplot(mpg, aes(x = displ, y = hwy, colour = class)) +
  geom_point() +
  facet_wrap(~ class) +
  guides(colour = FALSE) +
  labs(title = "Highway Fuel Consumption by Cylinder Displacement \nfor Each Vehicle Class",
       x = "Displacement (litres)",
       y = "Highway Fuel Consumption (mpg)")
```

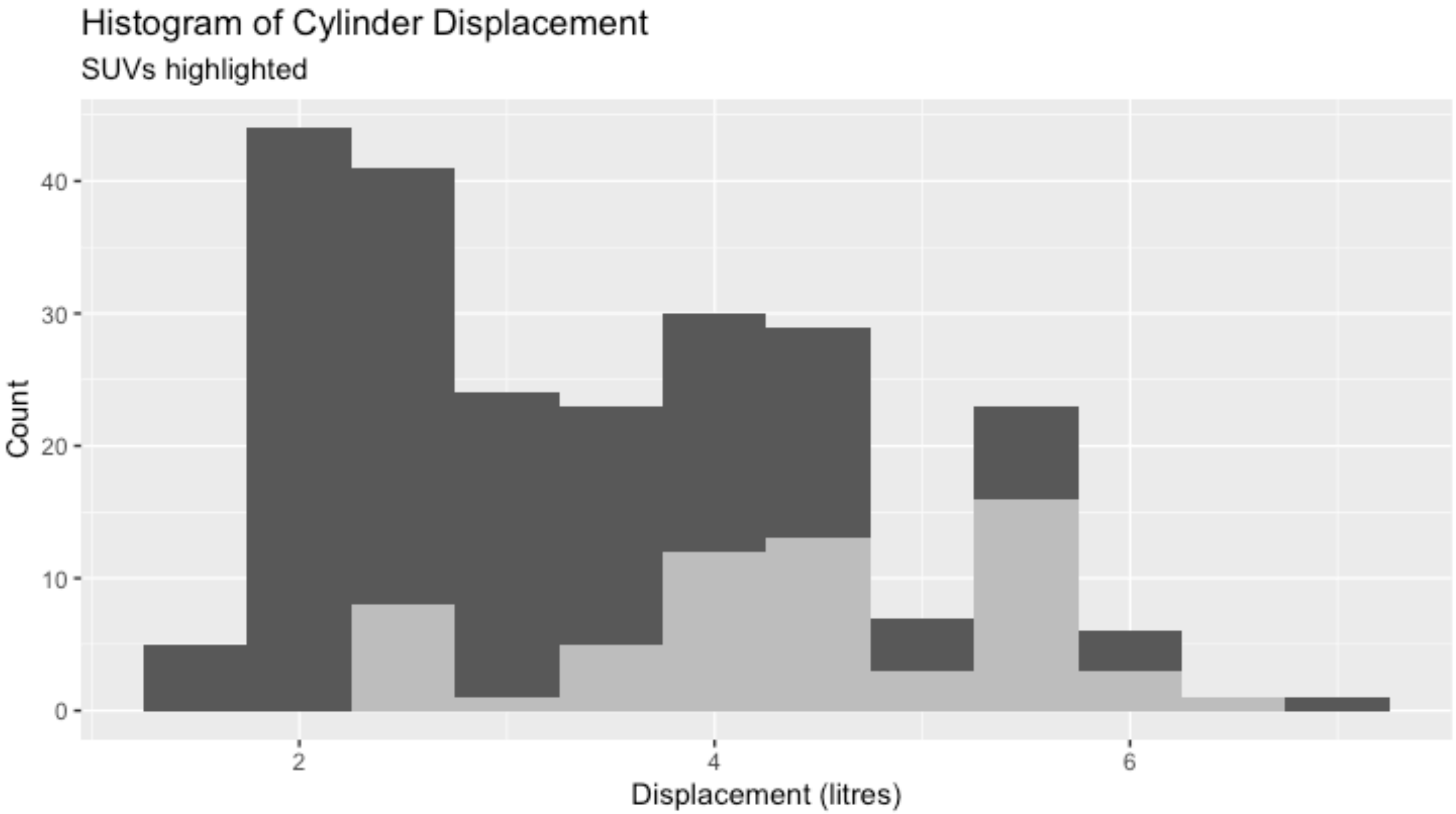
Adding a regression line



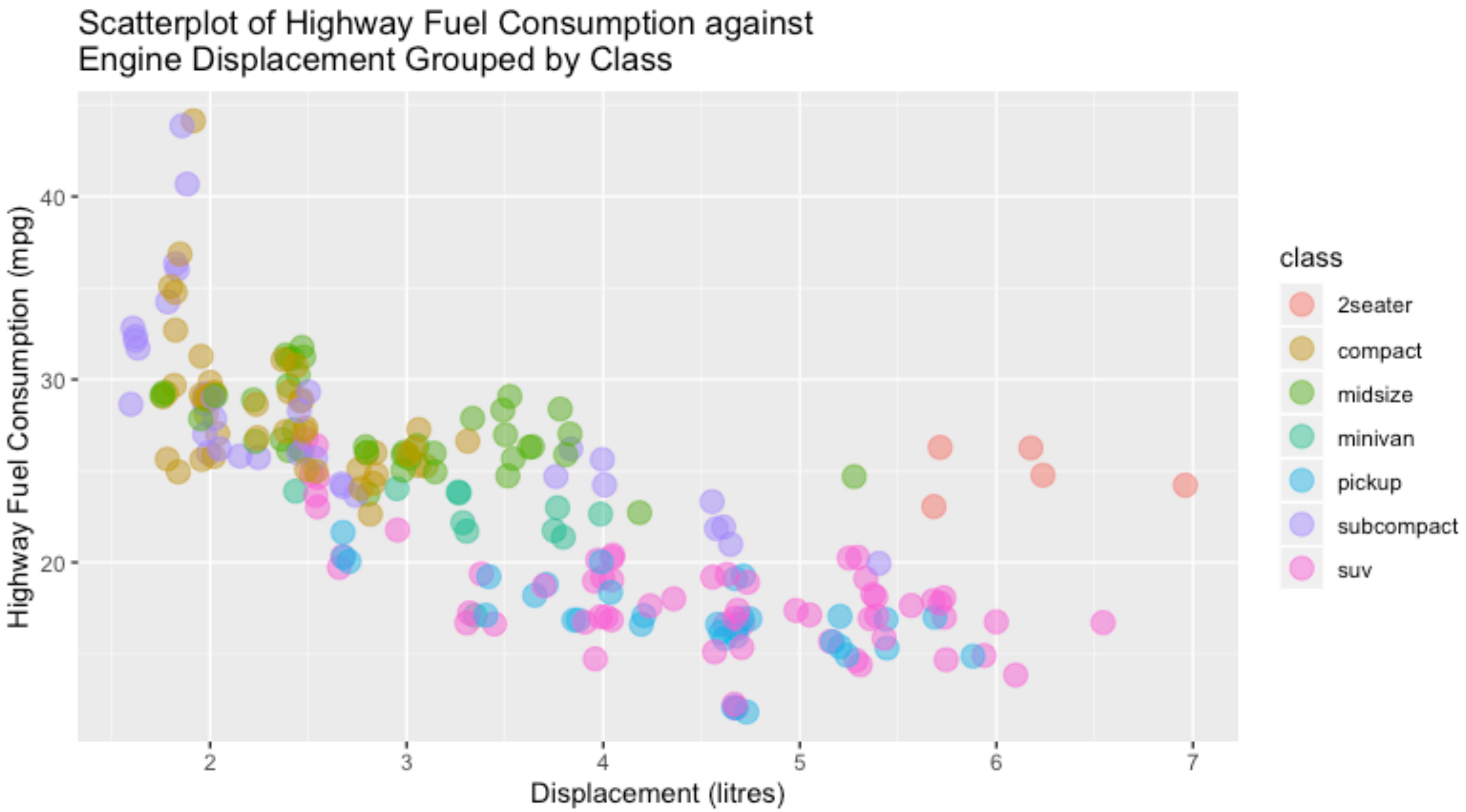
```
ggplot(mpg, aes(x = displ, y = hwy, colour = class)) +
  geom_point() +
  facet_wrap(~ class) +
  guides(colour = FALSE) +
  geom_smooth(method = "lm") +
  labs(title = "Highway Fuel Consumption by Cylinder Displacement \nfor Each Vehicle Class",
       x = "Displacement (litres)",
       y = "Highway Fuel Consumption (mpg)")
```



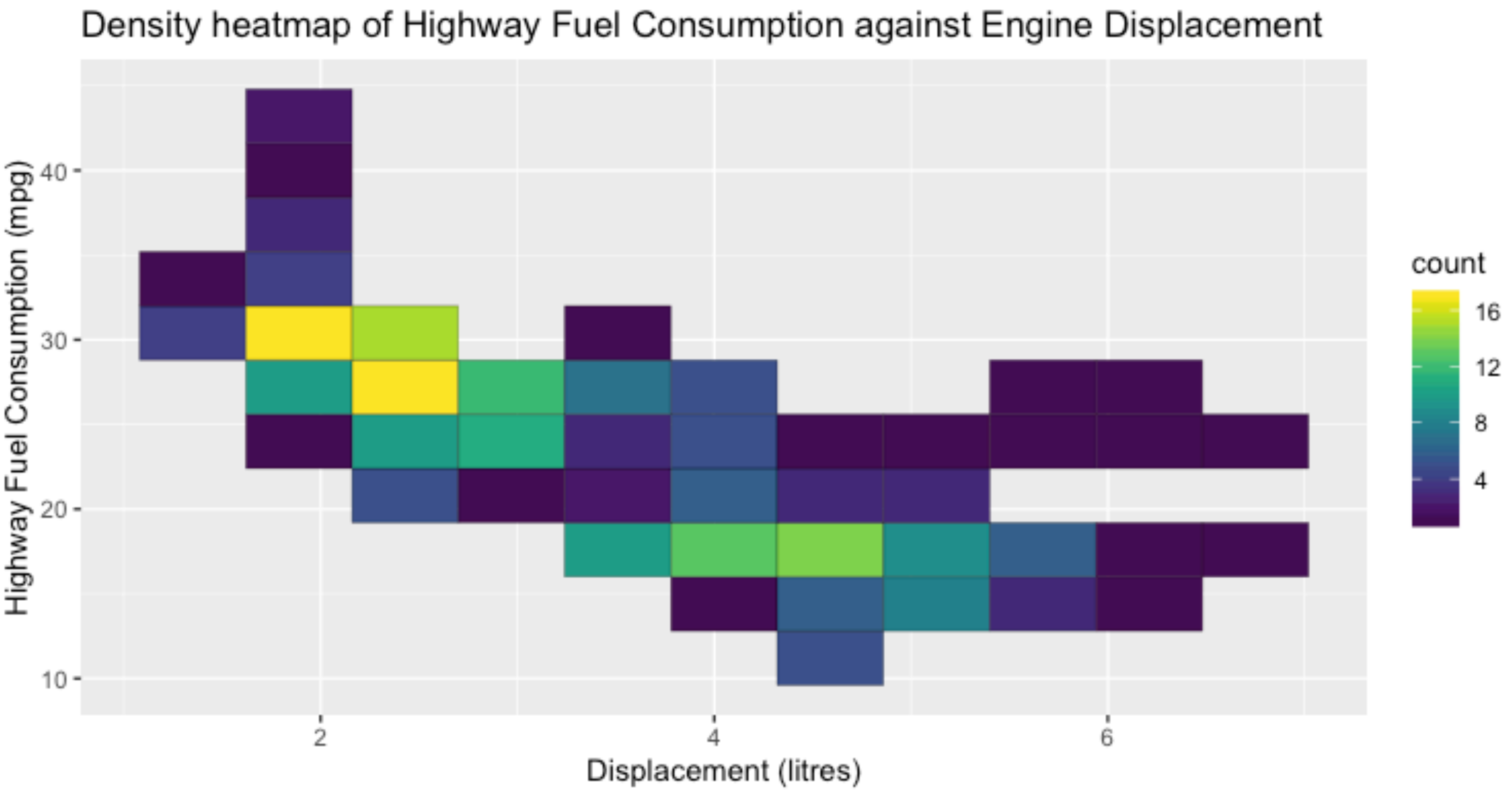
```
ggplot(mpg, aes(x = displ)) +
  geom_histogram(binwidth = .5) +
  guides(fill = FALSE) +
  labs(title = "Histogram of Cylinder Displacement",
       x = "Displacement (litres)",
       y = "Count")
```



```
ggplot(mpg, aes(x = displ)) +
  geom_histogram(binwidth = .5) +
  geom_histogram(data = filter(mpg, class == "suv"), fill = "grey", binwidth = .5) +
  guides(fill = FALSE) +
  labs(title = "Histogram of Cylinder Displacement",
       subtitle = "SUVs highlighted",
       x = "Displacement (litres)",
       y = "Count")
```



```
ggplot(mpg, aes(x = displ, y = hwy, colour = class)) +
  geom_jitter(width = 0.05, alpha = .5, size = 4) +
  labs(title = "Scatterplot of Highway Fuel Consumption against \nEngine
Displacement Grouped by Class",
       x = "Displacement (litres)",
       y = "Highway Fuel Consumption (mpg)")
```



```
ggplot(mpg, aes(x = displ, y = hwy)) +
  stat_bin2d(bins = 10, colour = "black") +
  scale_fill_viridis() +
  labs(title = "Density heatmap of Highway Fuel Consumption against Engine Displacement",
       x = "Displacement (litres)",
       y = "Highway Fuel Consumption (mpg)")
```

Plotting Time Series Data

We can install the `gapminder` package which contains lots of interesting data about about life expectancy, population size, GDP for lots of countries collected over lots of years.

```
> gapminder
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

Animated visualisations

For datasets with time series information, we might think it could be easier to tell our data story if we animate by time.

The `gganimate` package allows us to create animated visualisations from within R which we can import or embed in R Markdown documents.

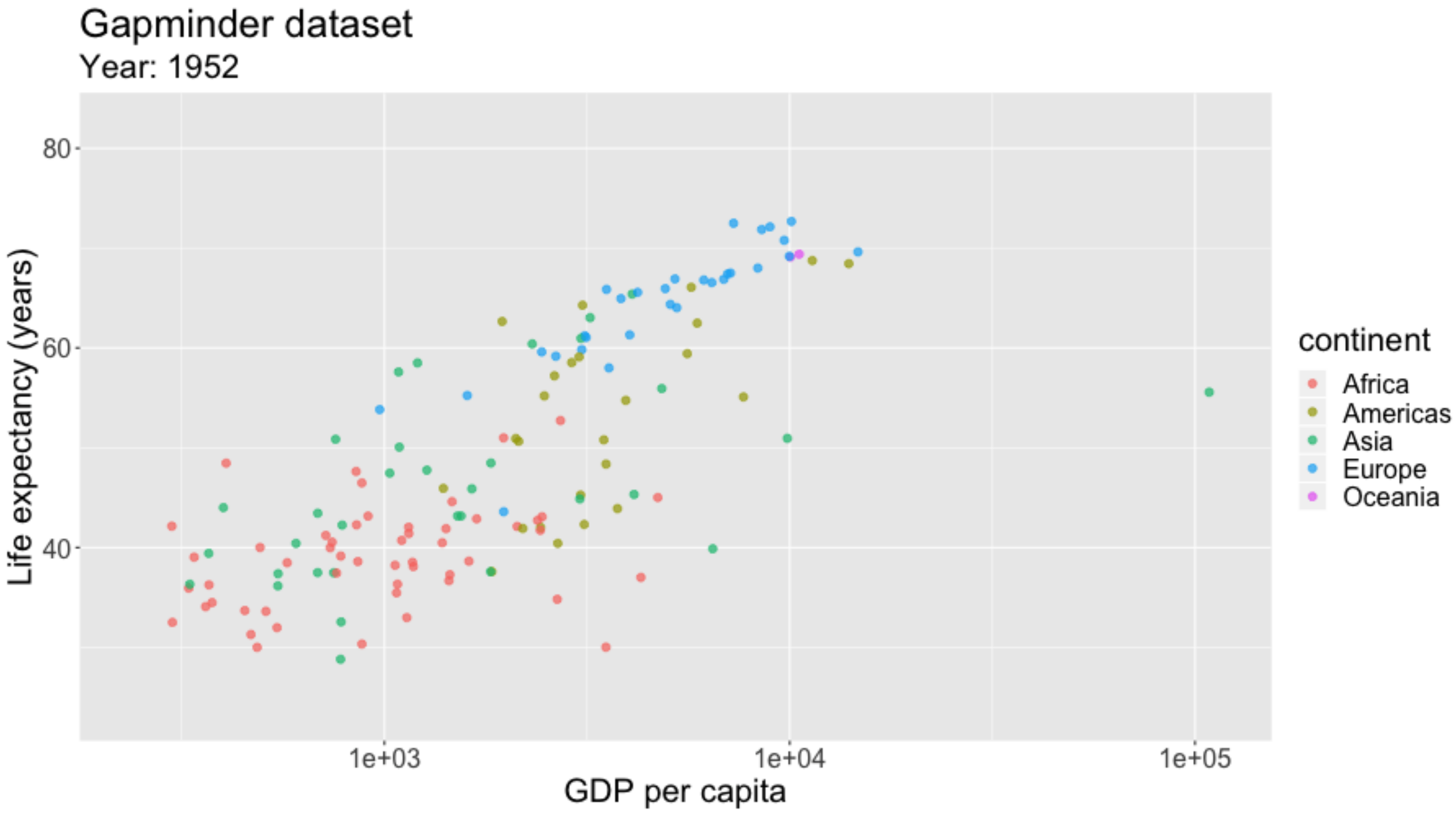
We need to install it via the usual route:

```
> install.packages("gganimate")
```

and then load it when we want to use it:

```
> library(gganimate)
```

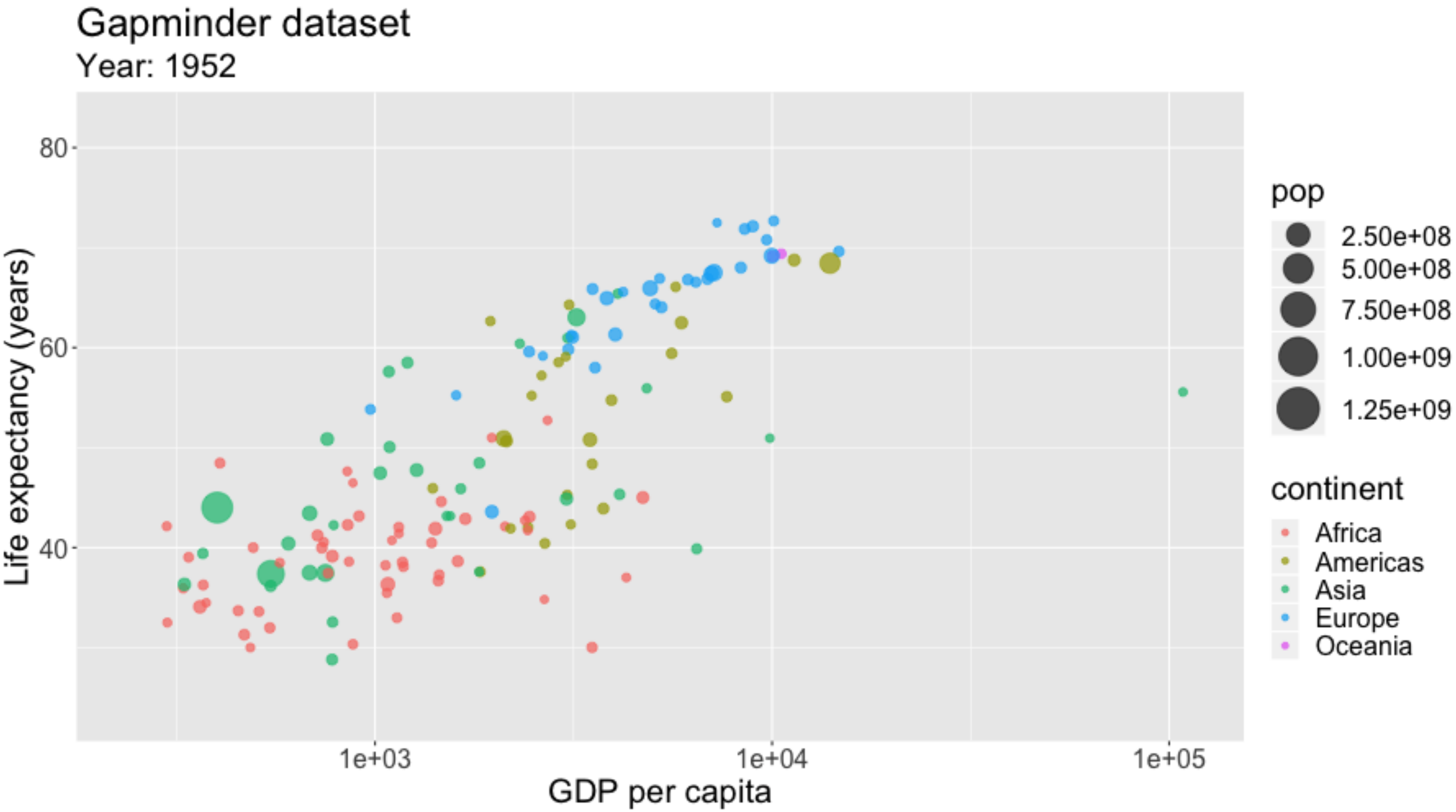
Animated Time Series Data



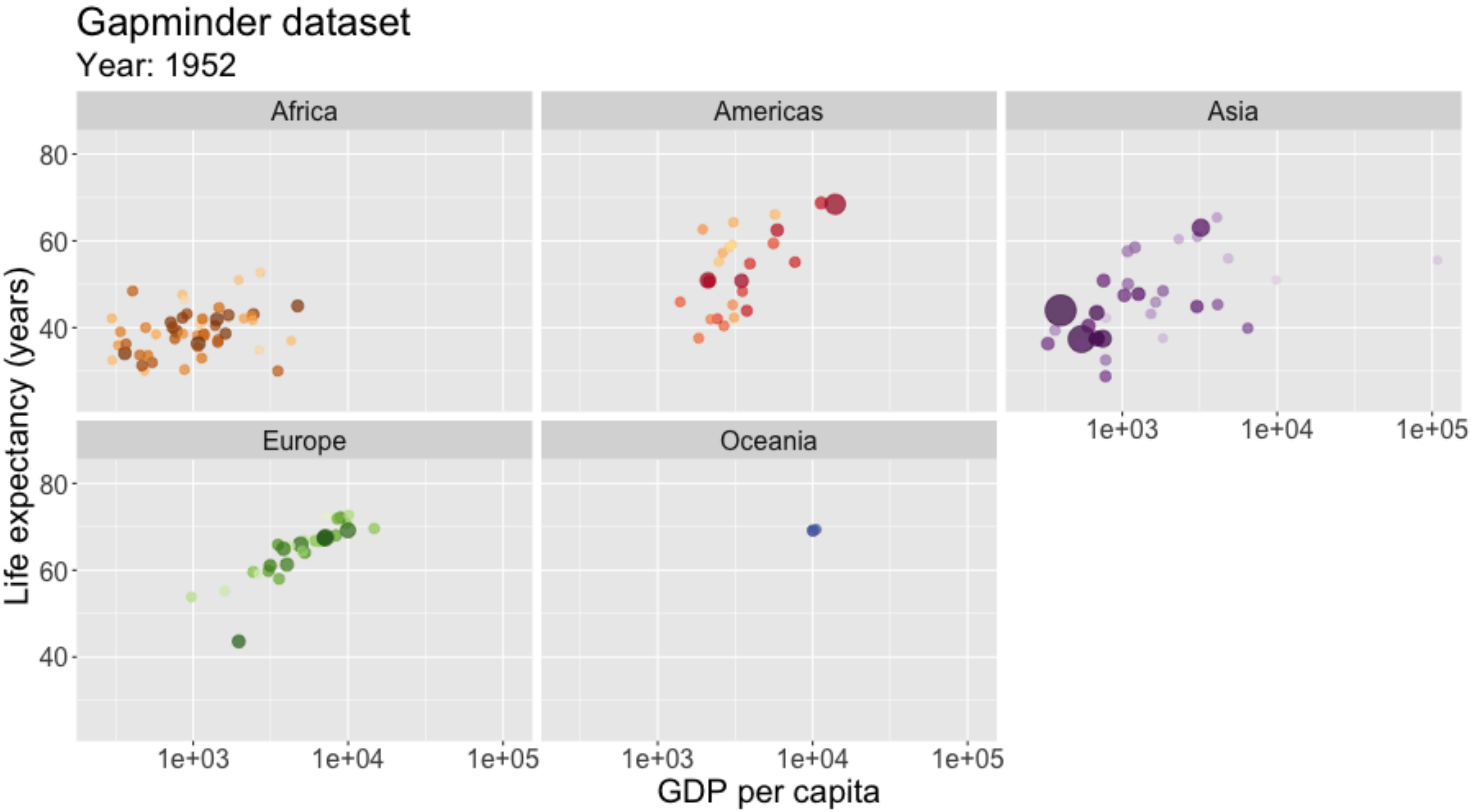
Visualising Data with 5 Variables Simultaneously

Animated Time Series Data

Now with a representation of population size.

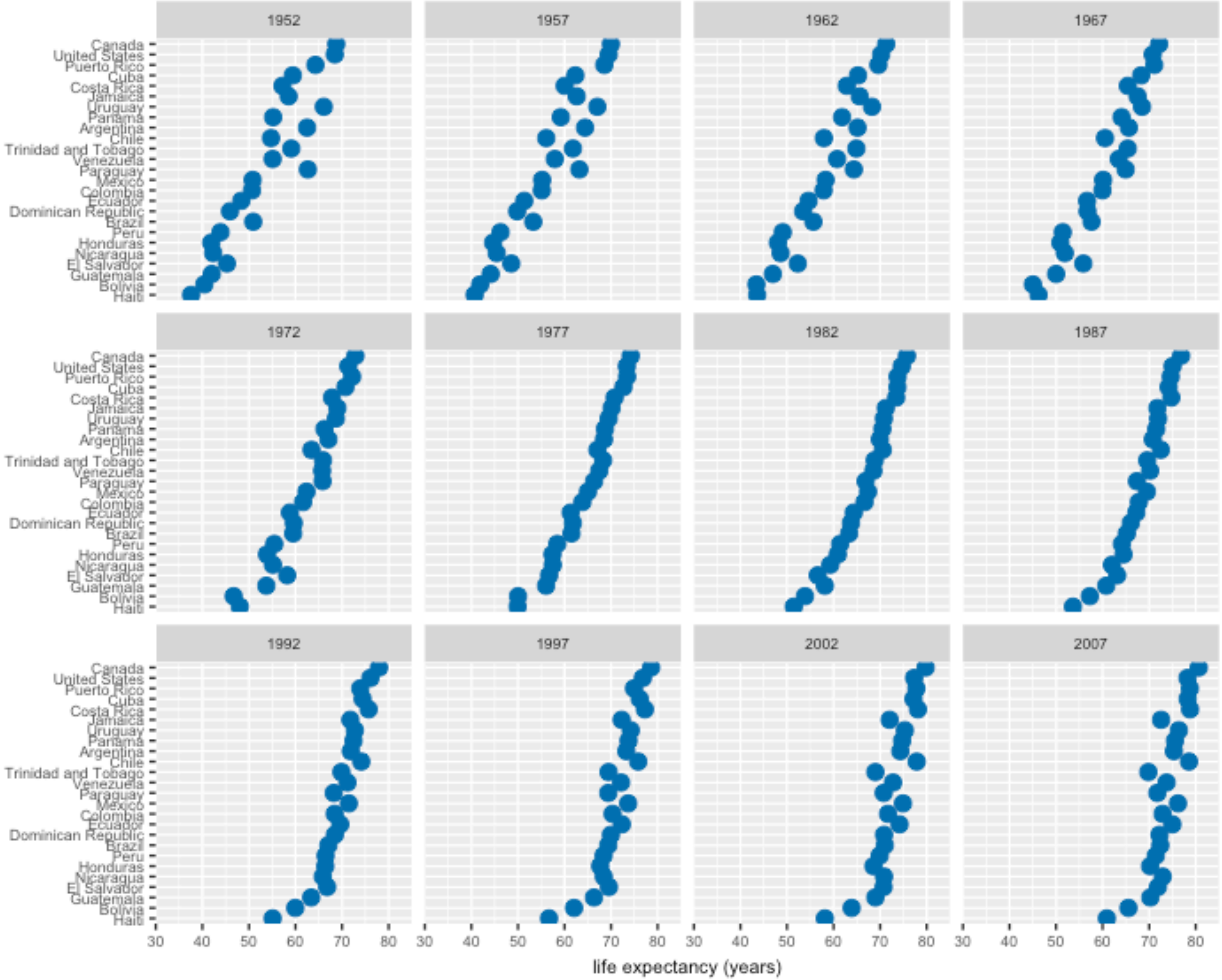


Separately by Continent

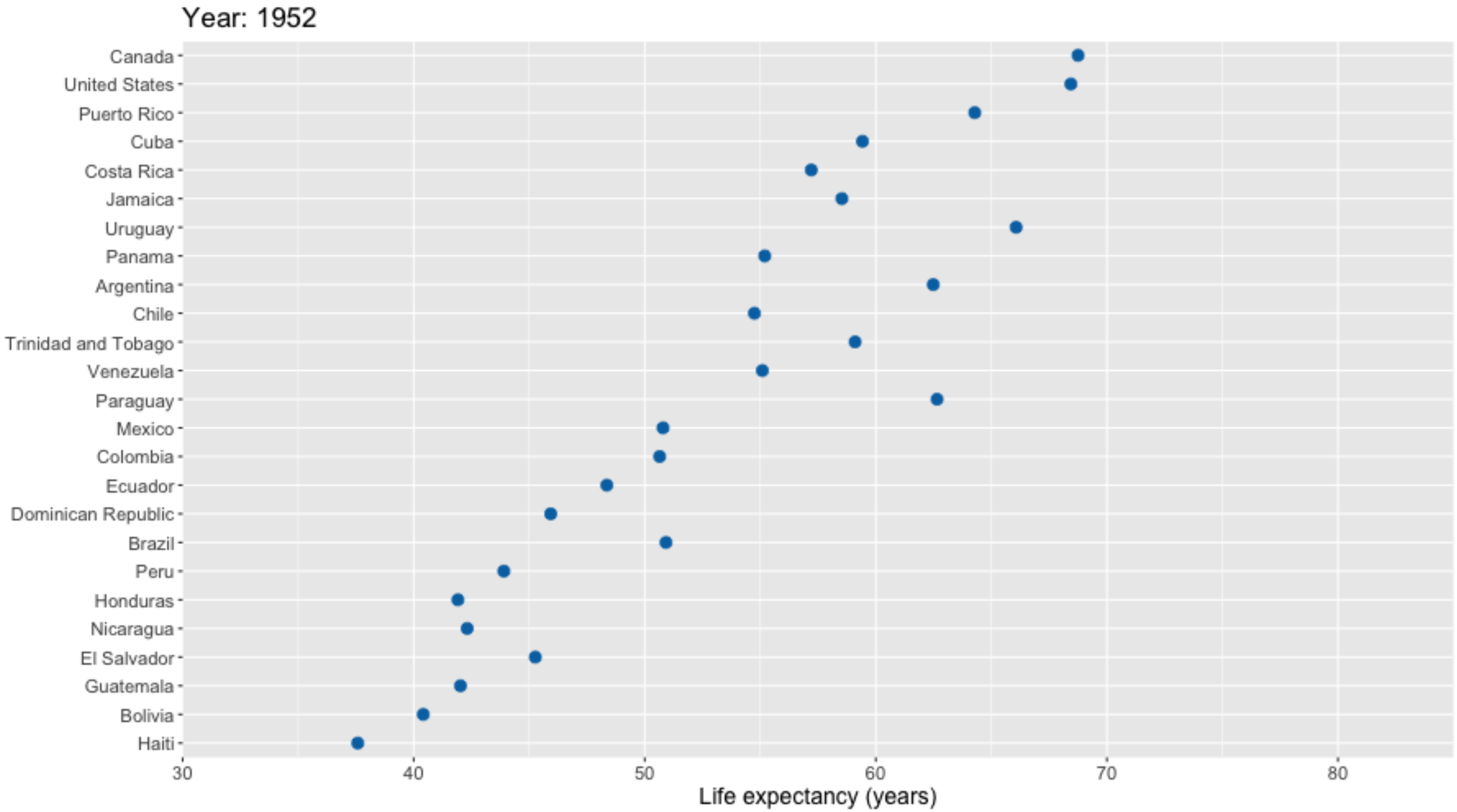


https://github.com/thomasp85/gganimate

Life Expectancy - Americas - Static



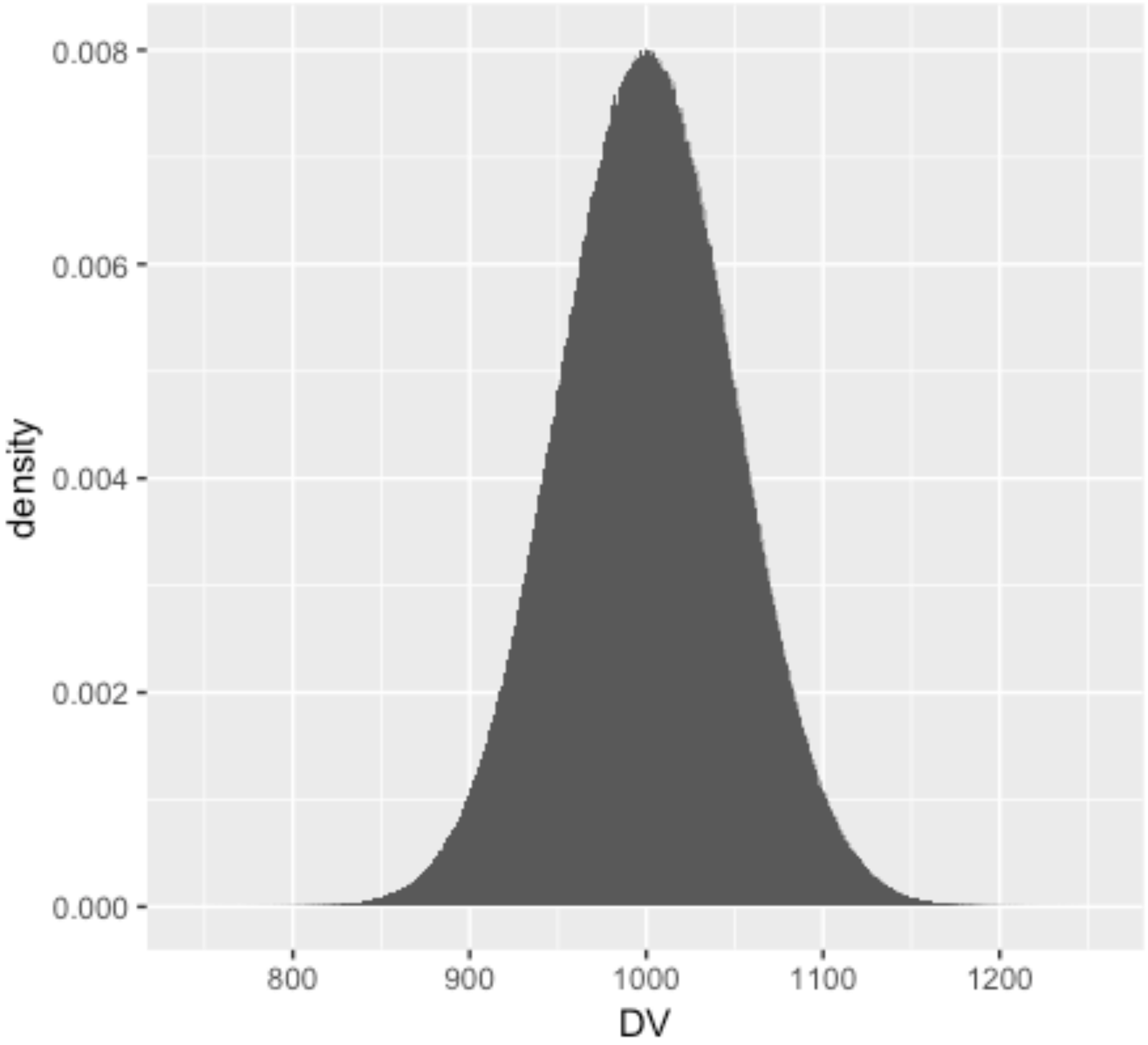
Life Expectancy - Americas - Animated



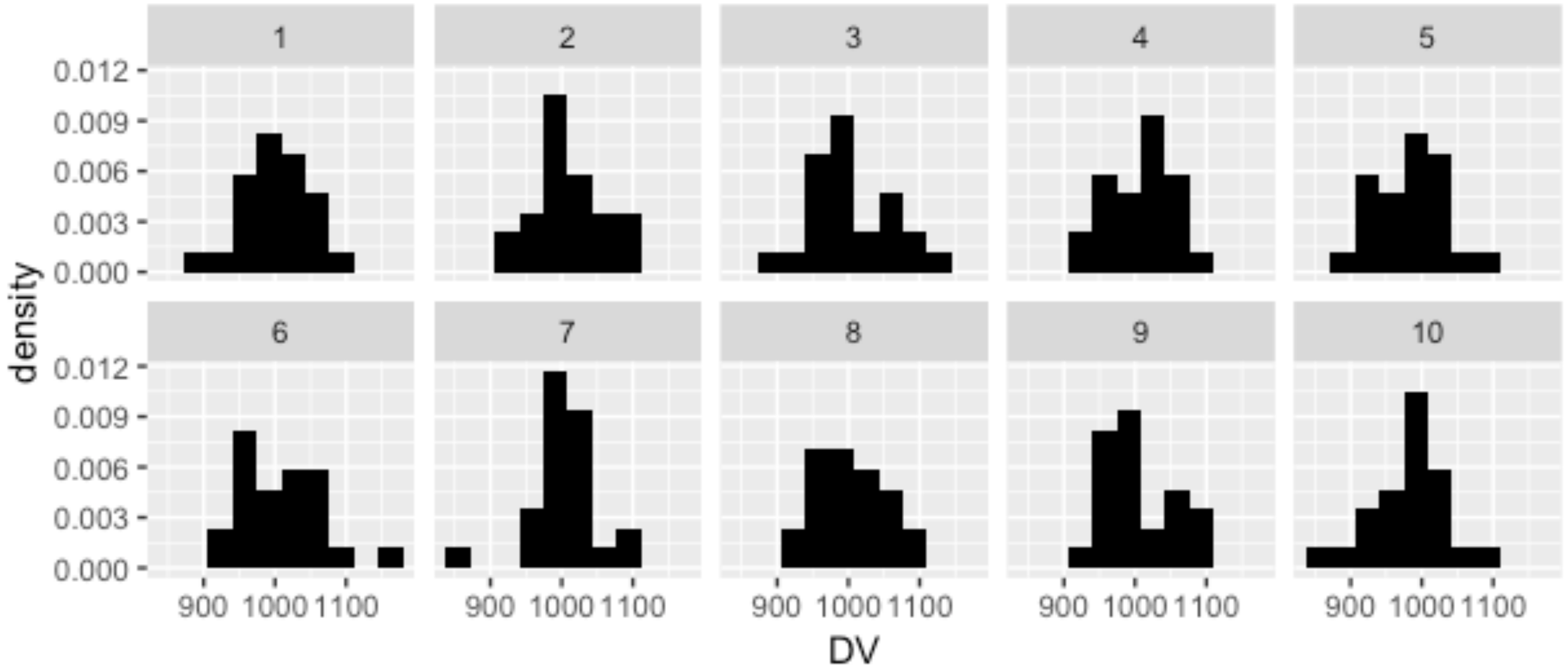
Although we have data only once every 5 years, the `gganimate` package interpolates between each census date to provide a smoother animation.

Using animation and data vis. to understand statistical concepts

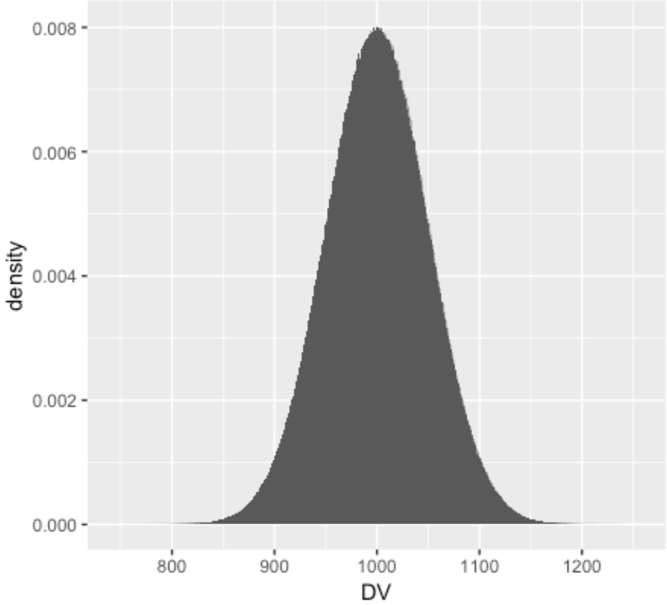
When we sample from a population, we are taking a sample of data points from the population distribution - the population could look something like this:



If our sample sizes are small, few sample distributions actually look like the population from which they're drawn and most sample means are a little different from the population mean:



None of these look much like:



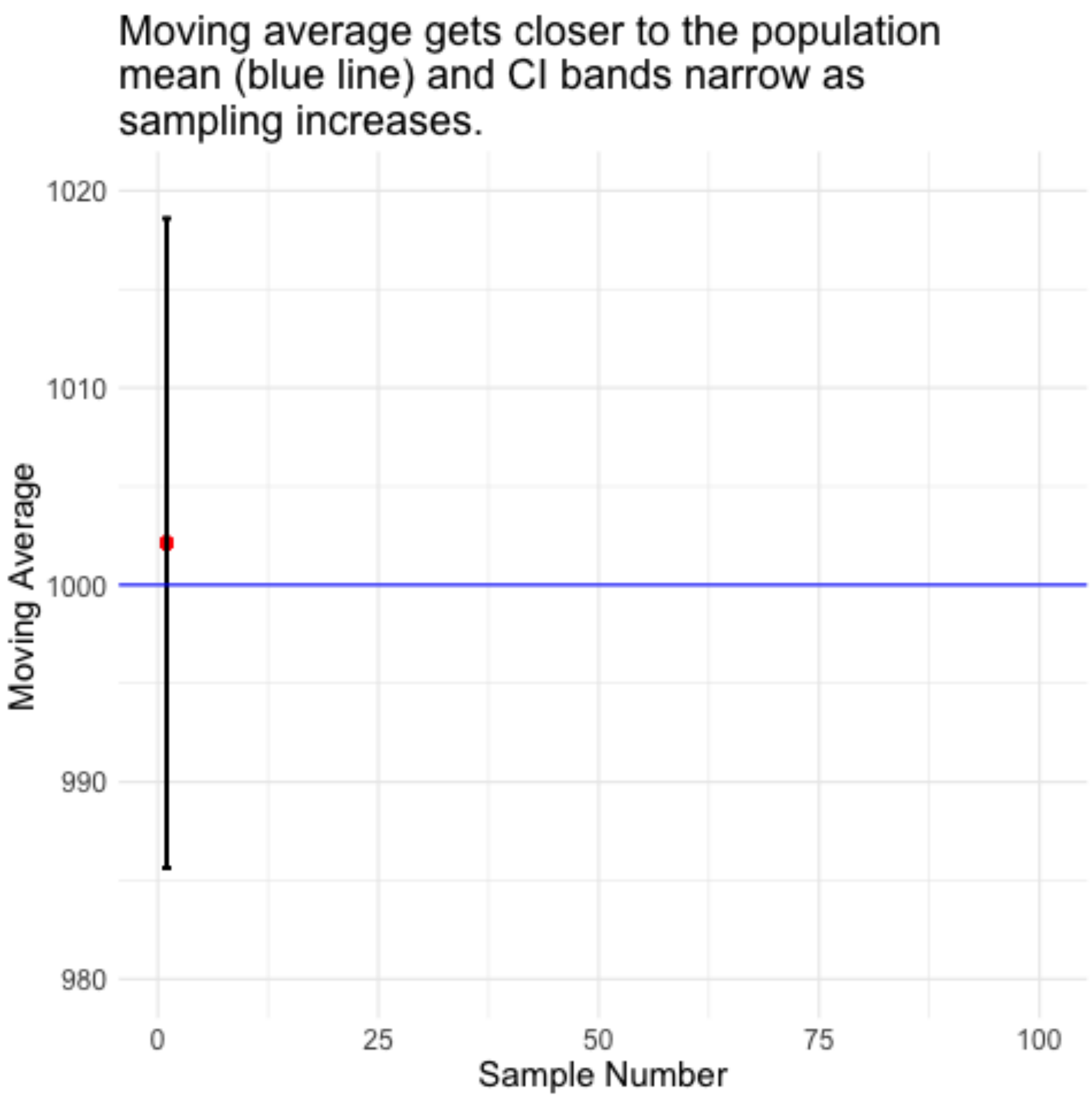
- When we take a sample from a population the mean of the sample may be quite different from the mean of the population (aka sampling error).

- If I take two samples, and work out the mean of these two samples, I should have a better estimate of the population mean than if I just looked at the mean of one of the samples.

- If I take three samples, work out the mean of these three samples etc. etc.

- The more samples (each with their own mean) we draw from the population, the closer we get to the true mean of the population. As sampling increases, the 95% CI bands around the mean narrow.

- So, animation can be used not just to communicate information, but also principles…



The Key Question

There is no such thing as the *best* way of visualising data - the method you choose will be determined by (e.g.) the type of data you have, the message you want to communicate, and the type of audience you will be communicating with.

Animations can be helpful, but they involve data being presented at a pace that might not suit the viewer - probably best suited for communicating time series data.