Visualization 101 2026

Data Visualization In R

Boris Shor

2026-05-20

Table of contents

1 Visualization 101 2026

This book is the working text for a five-day course on data visualization in R. It is designed to be read, run, edited, rendered, and reused. The goal is not only to make attractive graphs, but to make visual choices that clarify an argument, reveal a pattern, and leave a reproducible record of how the result was made.

Visualization is a sequence of choices. A plot emphasizes some comparisons and pushes others into the background. It chooses what counts as the main unit, what deserves color or space, what should be labeled directly, what can remain implicit, and what should be excluded because it distracts from the point. The course therefore treats visualization as part of analysis rather than as decoration added at the end.

R is the statistical language used throughout the course. RStudio Desktop and Posit Cloud are the interfaces used to write and run code. The tidyverse is the central programming framework: it supplies a readable grammar for importing, reshaping, summarizing, and plotting data. `ggplot2`, the tidyverse visualization package, is the main plotting system.

Quarto is the document system used for the course files. A Quarto document can contain prose, code, output, figures, tables, citations, and links in one place. The same source document can produce HTML, PDF, slides, dashboards, and reports. When the data or code changes, the outputs can be regenerated from the source instead of manually updated across Word documents, spreadsheets, and slide decks.

Literate programming also supports replication. Code and interpretation live together, which makes it easier to return to a project after a long break, share work with collaborators, and identify exactly how a result was produced. Writing the explanation forces decisions to be made explicit while they are still fresh.

1.1 Schedule

Day	Main Focus	Chapters
1	Workflow, Quarto, data import, and an introduction to `ggplot2`	2–3, start 4
2	Core `ggplot2`, separating and comparing data, distributions	finish 4, 5–6b

Day	Main Focus	Chapters
3	Lines, bars, annotation, and from question to final figure	7–8
4	Interactivity, dashboards, and tables	9–10
5	Model visualization, mapping, and agentic AI	11–13

Each day has five instructional sessions. The written chapters contain more material than the minimum live path so that faster pacing, review, and post-course study are supported.

1.2 Setup

The Posit Cloud project should already include the packages needed for the course. The file `install_packages.R` automates installation if the materials are moved to a local computer.

Run `check_setup.R` to verify that the packages and data files are available.

1.3 Files

The main book chapters are the numbered `.qmd` files in this project. Data files are under `Data/`. Standalone examples are under `Demos/`. Short exercises and solutions are under `Exercises/`.

The course examples use files included with the project, such as `Data/gapminder/gapminder.csv`.

2 Start Here: Workflow, Quarto, And RStudio

2.1 Learning Goals

By the end of this chapter, the main goals are:

- open and navigate the course project
- understand the relationship among R, RStudio, Posit Cloud, and Quarto
- run code chunks inside a `.qmd` file
- render a Quarto document to HTML
- use source mode, visual mode, and the document outline intentionally

2.2 The Course Workflow

This course uses R for computation, RStudio for the working interface, and Quarto for the documents that combine prose, code, and output. The goal is not only to make individual plots. The goal is to build a reproducible workflow where the text explains the question, the code performs the work, and the output records what happened.

During class, most work will happen inside `.qmd` files like this one. A Quarto file is a plain-text document that can contain headings, paragraphs, lists, links, code chunks, tables, figures, and notes about interpretation. When Quarto renders the file, it runs the code chunks and combines the results with the surrounding text.

This is different from working only in the console. The console is useful for quick tests, but it does not automatically preserve the reasoning around the command. A Quarto document keeps the analysis in order: what question was being asked, what data were used, what code was run, what output appeared, and what conclusion followed.

2.3 Literate Programming

Donald Knuth — who wrote *The Art of Computer Programming* and built the TeX typesetting system — coined the term *literate programming* in 1984. The idea is to write code primarily for human readers rather than for computers. Explanation and code go in the same document, in the same order as the reasoning, so a reader can follow both at once.

Most statistical work has not been done this way. The standard workflow for a long time has been: run the analysis in Stata or R, save the plots and tables to a folder somewhere, open Word, write the paper, load the figures in, open PowerPoint, build the slides, load the figures in again. Get a comment from a co-author, go back to Stata, re-run something, save a new version of the figure, find all the places it appears in Word and PowerPoint, replace them. Repeat. The analysis and the document are separate objects that have to be kept in sync by hand.

This works, after a fashion, but it is slow, it is error-prone, and it makes replication hard. A table in the slides may not match the one in the paper. A figure may have been made with an earlier cut of the data. Six months later, you may not be able to reconstruct exactly which version of the code produced a given number.

Quarto solves this by putting everything in one place. The prose, the code, the figures, and the tables all live in a single `.qmd` file. Render it and you get a paper, a slide deck, or a report — whatever the YAML specifies. Change the data or revise the analysis, render again, and all the outputs update at once. Nothing is pasted in from elsewhere and nothing gets out of sync.

It is worth saying something about LaTeX here, because it has been the standard for scientific publishing for decades, and for good reason. LaTeX produces beautiful output. The typesetting is precise, the mathematical notation is unmatched, and the journal templates for economics, political science, sociology, psychology, statistics, and most other quantitative disciplines are built for it. Many journals still expect it.

The problem is that LaTeX is genuinely hard to use and remarkably unforgiving. One missing brace and the document fails to compile with an error message that tells you very little. The syntax is arcane. Managing bibliographies, cross-references, and figure placement requires learning a small universe of packages. Collaborating on a LaTeX document means everyone needs a working installation and the patience to debug it. This is not like Word, where you type, save, and look at what you have.

Quarto gives you most of what LaTeX offers without requiring you to write it directly. Quarto uses Markdown as its writing format — plain text with simple conventions for headings, bold, lists, and links — and is the successor to R Markdown, which many researchers in these fields will have encountered. R Markdown established the idea of mixing R code with prose in a single document; Quarto takes that further, adding support for Python and other languages, a more consistent syntax, and better tools for books, websites, and presentations. When you render a `.qmd` file to PDF, Quarto calls LaTeX under the hood — so the output looks like a LaTeX document, journal templates work, and equations render properly — but you write in Markdown. You get the output quality without the brittleness. When you need to go further into LaTeX directly, it is there, but for most purposes you will not need to.

The tidyverse extends this thinking down to the code itself. Hadley Wickham designed `dplyr` and `ggplot2` to be read as well as run. A pipeline written with `|>` reads left to right: take this data, filter it, reshape it, summarize it by group. A `ggplot2` call names explicitly what each variable does in the plot and why. The code is not just instructions for R — it is a record of the decisions made during the analysis, which is exactly what Knuth had in mind.

The `Examples/Party Rolls` folder shows this in a real project, covered in more detail later in this chapter.

2.4 Plain R Scripts

The simplest kind of R source file is a plain R script. A plain R script uses the `.R` extension and contains only R code. It is useful for setup tasks, repeated checks, helper functions, and other code that does not need surrounding prose or rendered output.

This project includes a few plain R scripts:

```
install_packages.R
check_setup.R
```

The file `check_setup.R` is a good example. It checks whether the course packages and course data files are available. Opening that file and clicking **Source** runs the whole script. The result is a quick setup check, not a rendered document.

Use `.R` scripts when the goal is to run code directly. Use `.qmd` files when the goal is to combine explanation, code, and output.

2.5 The Console

The console is the fastest place to test a command, inspect an object, or rerun something without rendering a document.

Two console habits are especially helpful:

- the up arrow cycles through recent commands
- `Ctrl + R` searches command history in many RStudio setups

The console is not a replacement for a source file. If a command becomes part of the analysis, move it into a `.qmd` or `.R` file so it is preserved.

2.6 R, RStudio, Posit Cloud, And Quarto

These tools have different jobs:

- R is the programming language that reads data, transforms it, estimates models, and makes graphics.

- RStudio is the interface where you edit files, run code, inspect objects, view plots, and render documents.
- Posit Cloud is a browser-based version of the RStudio workflow that avoids many local installation problems.
- Quarto is the publishing system that turns `.qmd` source files into HTML, PDF, Word, slides, dashboards, and books.

R Markdown was the predecessor used in earlier versions of this course. Quarto keeps the same basic idea, but it is now the preferred format for this version of the workshop. If you have used `.Rmd` files before, the transition should feel familiar: a `.qmd` file still mixes text and code. The main difference is that Quarto uses a more general and consistent document system.

2.7 Project Files And Portable Paths

The examples in this course assume that the project has been opened in RStudio or Posit Cloud. Files are organized inside the project in folders such as `Data/`, `Demos/`, and `Exercises/`.

The data examples use file names like these:

```
Data/gapminder/gapminder.csv
Data/penguins/penguins.csv
Demos/presentation-demo.qmd
Exercises/04-scatterplot-solution.R
```

Avoid file names that depend on a personal Desktop, Downloads folder, or another location on one person's computer.

Keeping files inside the project matters because the course should work in Posit Cloud, on another computer, or after being archived for later use.

2.8 The RStudio Layout

RStudio usually has four main panes:

- the source pane, where `.qmd` and `.R` files are edited
- the console, where commands can be typed and run directly
- the environment and history pane, where current objects and previous commands are listed
- the files, plots, packages, help, and viewer pane

The exact pane arrangement can be changed under Global Options. The important habit is to know which pane is doing which job. The source pane is for durable work. The console is for quick tests. The environment pane shows what objects currently exist. The viewer pane is often where rendered HTML output appears.

2.9 RStudio Settings

A few RStudio settings can make workshop work more comfortable:

- turn off restoring `.RData` into the workspace at startup
- set saving the workspace on exit to "Never"
- turn on rainbow parentheses if available
- use the document outline for long `.qmd` files
- choose a source editor font and size that are comfortable for several hours of reading

These settings do not change the analysis, but they reduce friction. The main principle is that the project files should define the work, not a hidden workspace image from a previous session.

2.10 Source Mode And Visual Mode

RStudio can edit Quarto documents in source mode or visual mode.

Source mode shows the underlying Markdown and code exactly as written. It is the most transparent mode for learning because it makes headings, chunks, chunk options, links, and lists visible.

Visual mode is a more word-processor-like interface. It can be helpful for editing prose, tables, footnotes, and links. It is still editing the same `.qmd` file, but it hides some of the raw syntax while you write.

Both modes are useful. During this course, source mode is usually the clearest mode for understanding how the document works. Visual mode can be helpful later when revising narrative text.

2.11 Markdown Basics

Quarto uses Markdown for ordinary text formatting. A few patterns cover most needs:

```
# First-level heading

## Second-level heading

Regular paragraph text.

- a bullet point
- another bullet point

**bold text**

*italic text*

[Quarto website](https://quarto.org)
```

The number of `#` symbols controls the heading level. Headings also create the document outline, which makes long lessons easier to navigate.

2.12 Code Chunks

An R code chunk starts with three backticks and `{r}`. A useful chunk also has a short name after `r`, then ends with three backticks.

In the source file, the opening line would be ```` ```{r ch02-code-chunks-1} ```` and the closing line would be ```` ``` ````. The R code between those lines could be:

```
1 + 1
```

The code inside the chunk can be run interactively, and it can also be run during rendering. In RStudio, you can run a chunk with the green play button or with keyboard shortcuts. You can insert a new chunk from the menu or by typing the chunk delimiters yourself.

If that chunk runs, the result would be:

```
[1] 2
```

Chunks can contain more than one line. For example:

```
x <- 10
y <- 25
x + y
```

The assignment operator `<-` stores a value in an object. In the example above, `x` stores `10` and `y` stores `25`. After those objects exist, R can use them in later commands.

2.13 Chunk Names And Navigation

Most real code chunks in these course files have a name. The name appears in the opening line of the chunk. For example, this opening line names a chunk `ch02-first-calculation`:

```
# opening line in the source file:
# ```{r ch02-first-calculation}
```

Chunk names make long documents easier to navigate. In RStudio, the document outline can show both section headings and code chunks, so a descriptive chunk name makes it easier to jump to a specific part of the file. Chunk names also make error messages easier to interpret because Quarto can report which chunk failed.

Chunk names should be short, unique, and descriptive. A name such as `ch04-first-scatterplot` is more useful than a name such as `chunk1`.

2.14 Chunk Options

Chunk options control how code behaves during rendering. In Quarto, options are often written as special comment lines at the top of the chunk.

This chunk runs but hides the code:

```
[1] "2026-05-20"
```

This chunk displays code without running it:

```
Sys.Date()
```

`eval: false` is important for examples that require credentials, internet access, private files, or intentional manual action. The code remains visible as a template, but it does not run automatically.

2.15 Rendering

Rendering turns a source `.qmd` file into an output document. For this course, HTML is the primary output because it is easy to view in a browser and works well for code, figures, tables, and links.

When you render a document, Quarto starts from the top, runs the chunks in order, and inserts the output where it belongs. If a chunk fails, rendering stops. That is useful because it exposes broken code early.

The render button in RStudio is the easiest way to start. Open the file and click **Render** to create the output document. PDF output is also useful for papers, handouts, and print-ready reports, but it depends on a working TeX installation. Posit Cloud's tidyverse template already has TeX installed, so PDF rendering works without any additional setup. On a local machine without TeX, the `tinytex` package provides a lightweight installation:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

2.16 A Real Workflow: From Presentation to Published Paper

The `Examples/Party Rolls` folder shows this in practice. The files trace one research project from early results to publication:

- `rolls23.Rmd` — a presentation written in R Markdown. Rendering it produces `rolls23.pdf`, a slide deck.
- `agenda_leviathan23.Rmd` — the manuscript version of the same project, with different formatting and tables. Rendering it produces `agenda_leviathan23.pdf`, the submitted manuscript.
- `shor and kistner 2023.pdf` — the final published version, typeset by the journal.

The presentation and the manuscript share the same underlying code. When the analysis changed, it changed in one place and both outputs updated.

2.17 Output Formats

Quarto can produce many output formats from the same source file:

- HTML documents and books
- PDF reports
- Word documents

- RevealJS slides
- dashboards
- websites

2.17.1 Presentation Demo

A presentation file uses a `revealjs` format in the YAML header. Second-level headings become slides.

```
---
title: "Presentation Demo"
format:
  revealjs:
    theme: simple
---
```

```
## Slide Title

Slide text goes here.
```

The course includes a standalone demo at `Demos/presentation-demo.qmd`. Code chunks work the same way as in a chapter, but slides usually hide the code and show the output.

2.17.2 Paper Or Report Demo

A paper-style document adds a bibliography file and uses `article` as the document class. Citations use keys from the `.bib` file.

```
---
title: "Paper Demo"
author: "Name"
bibliography: paper-example.bib
format:
  html: default
  pdf: default
---
```

The course includes `Demos/paper-example.qmd` and `Demos/paper-example.bib`. Citations are written as `[@wickham_ggplot2_2016]`.

2.18 A First Reproducible Example

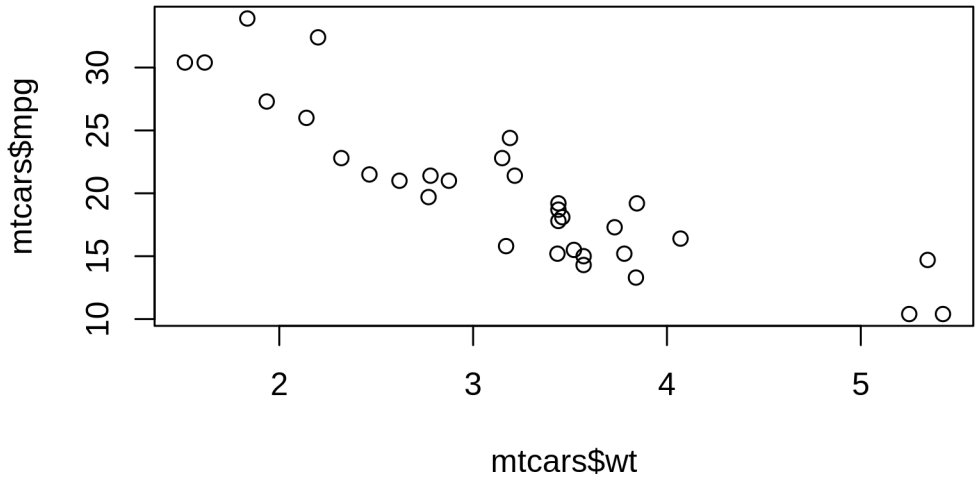
The built-in `mtcars` dataset is available in base R, so it is useful for a first test that does not depend on course data files.

```
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

This command creates a simple base R plot:

```
plot(mtcars$wt, mtcars$mpg)
```



The plot is not polished, but it confirms that R is running, code chunks are executing, and graphical output appears in the document.

2.19 Reading Output Carefully

A rendered document is not automatically correct just because it renders. Rendering answers one question: did the code run from top to bottom? You still have to ask whether the data are the right data, whether the variables mean what you think they mean, whether missing values were handled correctly, and whether the plot supports the claim being made.

This course will return to that distinction throughout the week. Reproducibility is necessary, but it is not enough. A reproducible mistake is still a mistake.

2.20 Short Exercise

Create a new code chunk below this paragraph. In that chunk:

1. store the number `100` in an object named `start_value`
2. store the number `25` in an object named `change_value`
3. subtract `change_value` from `start_value`

```
# Write your code here.
```

2.21 What Comes Next

The next chapter begins the data workflow: importing files, inspecting tibbles, reshaping data, and using the core tidyverse verbs that prepare data for visualization.

3 Data Import And The Tidyverse

3.1 Learning Goals

By the end of this chapter, the main goals are:

- read local data included with the project
- recognize the difference between a data frame and a tibble
- inspect a dataset before plotting it
- distinguish wide data from long data
- reshape data with `pivot_longer()` and `pivot_wider()`
- use the core tidyverse verbs for common data preparation tasks
- recognize which import examples should not run automatically because they require credentials or external services

3.2 Load Packages

The `tidyverse` is a collection of packages for importing, transforming, and visualizing data. Loading it makes packages such as `readr`, `dplyr`, `tidyr`, `ggplot2`, `stringr`, and `tibble` available.

```
library(tidyverse)
library(skimr)
```

3.3 Importing Local Data

Many projects begin with a local CSV file. CSV files are plain-text tables where columns are separated by commas. They are common because nearly every spreadsheet, statistical package, and database can export them.

The preferred tidyverse function is `read_csv()` from the `readr` package. Compared with base R's `read.csv()`, it usually prints more clearly, guesses column types more consistently, and returns a tibble.

The course data live under `Data/`. The example below reads the Gapminder CSV file included with the project.

```
gap <- read_csv("Data/gapminder/gapminder.csv", show_col_types = FALSE)

gap
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <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.
# i 1,694 more rows
```

The object name `gap` now points to a table in memory. The assignment operator `<-` reads as "store the result on the right in the object named on the left." Object names should be short enough to type but descriptive enough to understand later.

3.4 Tibbles And First Inspection

A tibble is the tidyverse version of a data frame. It is still a rectangular table with rows and columns, but it prints in a more controlled way. A tibble tells you the number of rows and columns, shows variable types, and avoids flooding the console with every row of a large dataset.

Start every new dataset with a quick inspection:

```
glimpse(gap)
```

```
Rows: 1,704
Columns: 6
$ country   <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan", ~
```

```
$ continent <chr> "Asia", "Asia", "Asia", "Asia", "Asia", "Asia", "Asia", "Asi~
$ year      <dbl> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, ~
$ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40.8~
$ pop       <dbl> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, 12~
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134, ~
```

```
skim(gap)
```

Table 3.1: Data summary

Name	gap
Number of rows	1704
Number of columns	6
_____	
Column type frequency:	
character	2
numeric	4
_____	
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
country	0	1	4	24	0	142	0
continent	0	1	4	8	0	5	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
year	0	1	1979.50	17.27	1952.00	1965.75	1979.50	1993.25	2007.0	
lifeExp	0	1	59.47	12.92	23.60	48.20	60.71	70.85	82.6	
pop	0	1	29601212.32	106157896.70	60011.00	2793664.00	7023595.50	19585221.75	1318683096.0	
gdpPercap	0	1	7215.33	9857.45	241.17	1202.06	3531.85	9325.46	113523.1	

`glimpse()` comes from `dplyr`, which is loaded as part of the tidyverse. It gives a compact structural view of the data. `skim()` comes from the `skimr` package and gives a broader summary, including missingness and descriptive statistics. These checks are not just housekeeping. They help identify the unit of observation, variable types, unexpected missing values, and suspicious ranges before those problems enter a plot.

3.5 Base R Import And Inspection

Base R can also import data. The syntax is common in older examples and in some online answers:

```
gap_base <- read.csv("Data/gapminder/gapminder.csv")
str(gap_base)
head(gap_base)
```

3.6 Other Local File Types

CSV is only one possible format. Social science projects often use Excel, Stata, SPSS, or SAS files. The syntax below should be treated as a template. These chunks are non-executing unless the named files are present and the relevant packages are installed.

3.6.1 Excel

The function `read_excel()` comes from the `readxl` package.

```
library(readxl)

gap_excel <- read_excel(
  path = "Data/gapminder/gapminder.xlsx",
  sheet = "gapminder"
)

gap_excel
```

3.6.2 Stata

The functions for Stata, SPSS, and SAS files below come from the `haven` package.

```
library(haven)

state_legislatures <- read_dta("Data/state_policy/shor_mccarty_state_data.dta")

state_legislatures
```

3.6.3 SPSS

The example uses a small SPSS file that is installed with the `haven` package.

```
library(haven)

path <- system.file("examples", "iris.sav", package = "haven")
survey_data <- read_sav(path)

survey_data
```

3.6.4 SAS

The example uses a small SAS file that is installed with the `haven` package.

```
library(haven)

path <- system.file("examples", "iris.sas7bdat", package = "haven")
admin_data <- read_sas(path)

admin_data
```

3.7 Wide And Long Data

Data can be organized in more than one shape. A *wide* dataset has one row per unit and a separate column for each measurement (`gdpPercap`, `lifeExp`). A *long* dataset stacks the measurement names into one column and the values into another. Wide data is often easier for people to read; long data is often easier for `ggplot2` and group-wise summaries because the measurement name itself can be mapped to color, facets, or groups.

`pivot_longer()` and `pivot_wider()` from `tidyr` move data between the two shapes.

```
wide_data <- tibble(
  country = c("Afghanistan", "Brazil"),
  year = c(2007, 2007),
  gdpPercap = c(974.6, 9065.8),
  lifeExp = c(43.8, 72.4)
)

long_data <- wide_data |>
```

```
  pivot_longer(
    cols = c(gdpPercap, lifeExp),
    names_to = "variable",
    values_to = "value"
  )

long_data
```

```
# A tibble: 4 x 4
  country      year variable   value
  <chr>       <dbl> <chr>      <dbl>
1 Afghanistan  2007 gdpPercap  975.
2 Afghanistan  2007 lifeExp     43.8
3 Brazil       2007 gdpPercap 9066.
4 Brazil       2007 lifeExp     72.4
```

```
long_data |>
  pivot_wider(names_from = variable, values_from = value)
```

```
# A tibble: 2 x 4
  country      year gdpPercap lifeExp
  <chr>       <dbl>     <dbl>   <dbl>
1 Afghanistan  2007      975.    43.8
2 Brazil       2007     9066.    72.4
```

The pipe operator `|>` means "take the result so far and send it into the next function." It lets a sequence of data steps read in the same order as the logic of the analysis.

3.8 Core Tidyverse Verbs

Most data preparation before plotting uses a small set of verbs:

- `filter()` keeps rows
- `select()` keeps columns
- `mutate()` creates or modifies variables
- `group_by()` defines groups
- `summarize()` collapses rows into summaries
- joins combine information from two tables

These verbs come from `dplyr`, which is loaded as part of the tidyverse. They are useful because they correspond to ordinary analytical questions. Which cases belong in this plot? Which variables matter? Do we need a new rate, logged value, or category? What is the summary by group? Does another table contain information that should be merged in?

3.9 `filter()`: Keep Rows

`filter()` keeps observations that satisfy logical conditions. The operator `==` tests exact equality. The operator `%in%` tests whether a value belongs to a set.

```
gap |>
  filter(year == 2007)
```

```
# A tibble: 142 x 6
   country     continent  year lifeExp       pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>     <dbl>     <dbl>
 1 Afghanistan Asia       2007    43.8  31889923      975.
 2 Albania     Europe     2007    76.4   3600523     5937.
 3 Algeria     Africa     2007    72.3  33333216     6223.
 4 Angola      Africa     2007    42.7  12420476     4797.
 5 Argentina   Americas   2007    75.3  40301927    12779.
 6 Australia   Oceania    2007    81.2  20434176    34435.
 7 Austria     Europe     2007    79.8   8199783    36126.
 8 Bahrain     Asia       2007    75.6    708573    29796.
 9 Bangladesh  Asia       2007    64.1 150448339     1391.
10 Belgium     Europe     2007    79.4  10392226    33693.
# i 132 more rows
```

```
gap |>
  filter(continent %in% c("Americas", "Europe"))
```

```
# A tibble: 660 x 6
   country continent  year lifeExp     pop gdpPercap
   <chr>   <chr>     <dbl>   <dbl>   <dbl>     <dbl>
 1 Albania Europe     1952    55.2 1282697     1601.
 2 Albania Europe     1957    59.3 1476505     1942.
 3 Albania Europe     1962    64.8 1728137     2313.
 4 Albania Europe     1967    66.2 1984060     2760.
 5 Albania Europe     1972    67.7 2263554     3313.
 6 Albania Europe     1977    68.9 2509048     3533.
```

```
 7 Albania Europe     1982    70.4 2780097     3631.
 8 Albania Europe     1987    72   3075321     3739.
 9 Albania Europe     1992    71.6 3326498     2497.
10 Albania Europe     1997    73.0 3428038     3193.
# i 650 more rows
```

Use `==` for one value. Use `%in%` when several values are acceptable.

3.10 `select()`: Keep Columns

`select()` keeps variables. This is useful when a dataset has many columns but the next step only needs a few.

```
gap |>
  select(country, continent, year, lifeExp, gdpPercap)
```

```
# A tibble: 1,704 x 5
   country     continent  year lifeExp gdpPercap
   <chr>       <chr>     <dbl>   <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8      779.
 2 Afghanistan Asia       1957    30.3      821.
 3 Afghanistan Asia       1962    32.0      853.
 4 Afghanistan Asia       1967    34.0      836.
 5 Afghanistan Asia       1972    36.1      740.
 6 Afghanistan Asia       1977    38.4      786.
 7 Afghanistan Asia       1982    39.9      978.
 8 Afghanistan Asia       1987    40.8      852.
 9 Afghanistan Asia       1992    41.7      649.
10 Afghanistan Asia       1997    41.8      635.
# i 1,694 more rows
```

Rows are observations. Columns are variables. `filter()` works on rows; `select()` works on columns.

3.11 `mutate()`: Create Variables

`mutate()` creates new variables or rewrites existing ones. In the next example, `round()` makes GDP per capita easier to read, and `log10()` creates a transformed variable that will be useful for plotting.

```r
gap |>
  mutate(
    gdp_pc_round = round(gdpPercap),
    log_gdp_pc = log10(gdpPercap)
  ) |>
  select(country, gdpPercap, gdp_pc_round, log_gdp_pc)
```

```
# A tibble: 1,704 x 4
   country     gdpPercap gdp_pc_round log_gdp_pc
   <chr>           <dbl>        <dbl>      <dbl>
 1 Afghanistan      779.          779       2.89
 2 Afghanistan      821.          821       2.91
 3 Afghanistan      853.          853       2.93
 4 Afghanistan      836.          836       2.92
 5 Afghanistan      740.          740       2.87
 6 Afghanistan      786.          786       2.90
 7 Afghanistan      978.          978       2.99
 8 Afghanistan      852.          852       2.93
 9 Afghanistan      649.          649       2.81
10 Afghanistan      635.          635       2.80
# i 1,694 more rows
```

`round(gdpPercap)` rounds each value in the `gdpPercap` column. `log10(gdpPercap)` takes the base-10 logarithm. The original variable remains unless you explicitly overwrite it.

3.12 `group_by()` And `summarize()`: Aggregate Data

Many plots need data at a different level than the imported file. For example, the Gapminder data are country-year observations, but we may want a continent-level summary.

```r
continent_summary <- gap |>
  group_by(continent) |>
  summarize(
    median_life_exp = median(lifeExp, na.rm = TRUE),
    median_gdp_pc = median(gdpPercap, na.rm = TRUE),
    n_countries = n_distinct(country)
  )

continent_summary
```

```
# A tibble: 5 x 4
  continent median_life_exp median_gdp_pc n_countries
  <chr>               <dbl>         <dbl>       <int>
1 Africa               47.8         1192.          52
2 Americas             67.0         5466.          25
3 Asia                 61.8         2647.          33
4 Europe               72.2        12082.          30
5 Oceania              73.7        17983.           2
```

`na.rm = TRUE` tells functions such as `median()` to ignore missing values. Without it, one missing value can make a summary missing.

3.13 Readable Pipelines

Long pipelines are easier to read when each verb gets its own line and arguments are indented consistently.

```
gap |>
  filter(year == 2007) |>
  mutate(gdp_pc_round = round(gdpPercap)) |>
  group_by(continent) |>
  summarize(
    median_life_exp = median(lifeExp, na.rm = TRUE),
    median_gdp_pc = median(gdp_pc_round, na.rm = TRUE)
  ) |>
  arrange(desc(median_life_exp))
```

```
# A tibble: 5 x 3
  continent median_life_exp median_gdp_pc
  <chr>               <dbl>         <dbl>
1 Oceania              80.7         29810
2 Europe               78.6         28054
3 Americas             72.9          8948
4 Asia                 72.4          4471
5 Africa               52.9          1452
```

The line breaks are not just style. They make the data workflow visible. Each line is one transformation.

3.14 Joins

Joins combine two tables using one or more key variables. A common pattern is to keep all rows from the main table and add matching columns from a smaller lookup table. That is a `left_join()`, another `dplyr` function.

```
region_notes <- tibble(
  continent = c("Africa", "Americas", "Asia", "Europe", "Oceania"),
  example_note = c(
    "Many countries, large range of outcomes",
    "Includes North, Central, and South America",
    "Large population range",
    "High median life expectancy in many examples",
    "Small number of countries in many teaching datasets"
  )
)

continent_summary |>
  left_join(region_notes, by = "continent")
```

```
# A tibble: 5 x 5
  continent median_life_exp median_gdp_pc n_countries example_note
  <chr>               <dbl>         <dbl>       <int> <chr>
1 Africa               47.8         1192.          52 Many countries, large ran~
2 Americas             67.0         5466.          25 Includes North, Central, ~
3 Asia                 61.8         2647.          33 Large population range
4 Europe               72.2        12082.          30 High median life expectan~
5 Oceania              73.7        17983.           2 Small number of countries~
```

Rows without a match in the lookup table would remain, but the new variables would be `NA`. That behavior is usually safer than silently dropping observations.

3.15 Short Exercise

Use the built-in `mtcars` dataset. Write a pipeline that:

1. converts `mtcars` to a tibble with row names stored as a column named `car`
2. groups by `cyl`
3. calculates average `mpg` and maximum `wt`
4. returns one row per value of `cyl`

```
# Write your code here.
```

3.16 Extras: Google Sheets

Google Sheets can be useful when data are collaboratively maintained. These examples require authentication and should not run automatically in a rendered course book.

```
library(googlesheets4)

sheet_data <- read_sheet(
  ss = "PASTE-GOOGLE-SHEET-ID-OR-URL-HERE",
  sheet = "Sheet1"
)

sheet_data
```

```
library(googlesheets4)

ss <- gs4_create("Visualization 101 2026 Example Sheet")

mtcars |>
  as_tibble(rownames = "car") |>
  sheet_write(ss = ss, sheet = "mtcars")
```

3.17 Census Data With `tidycensus`

Some data sources are best accessed through specialized packages. These examples are templates, not automatically running chunks.

The `tidycensus` package wraps the U.S. Census Bureau's APIs. The two most common entry points are the American Community Survey (`get_acs()`) and the decennial census (`get_decennial()`). A free Census API key is required. Request one at https://api.census.gov/data/key_signup.html, then register it once on your machine with `census_api_key("YOUR_KEY", install = TRUE)`. After that, the key is available to every R session and the line can be removed from teaching files.

```
library(tidycensus)

#census_api_key("2c5ba48be91062db570c7fc5fa49dbca03306c33", install = TRUE)
```

`load_variables()` lists the variables available in a given dataset and year. Caching the result avoids re-downloading the table.

```
acs_vars <- load_variables(2020, "acs5", cache = TRUE) |>
  distinct(concept, .keep_all = TRUE)

acs_vars
```

If `load_variables()` errors with a JSON parse message like `invalid char in json text. <html>`, the Census Bureau returned an HTML error page instead of the variable table. That usually means the requested year is not yet published for that dataset, the Census server is temporarily down, or the request was rate-limited. Try a slightly older year (for ACS 5-year, two years behind the current year is usually safe), re-run the call after a minute, or check https://www.census.gov/data/developers/data-sets.html for the dataset's release schedule.

The `variables` argument of `get_acs()` accepts named variable codes. Naming the entries gives readable column names instead of raw codes like `B19013_001`. `output = "wide"` puts each variable in its own column, which is usually easier to plot from than the default long format.

```
acs_raw <- get_acs(
  geography = "congressional district",
  variables = c(
    median_income = "B19013_001",
    total_pop = "B01003_001"
  ),
  year = 2022,
  survey = "acs5",
  output = "wide"
)

acs_raw
```

The `GEOID` column is a concatenation of geographic identifiers. For congressional districts it combines the two-digit state FIPS code with the two-digit district number. Splitting it with `str_sub()` gives joinable columns.

```
acs <- acs_raw |>
  mutate(
    fips = str_sub(GEOID, 1, 2),
    cd = str_sub(GEOID, 3, 4)
  )
```

```
acs
```

State-level summaries are simpler because the `GEOID` is just the state FIPS code. The wide format makes it easy to compare one variable against another.

```
acs_state <- get_acs(
  geography = "state",
  variables = c(
    median_income = "B19013_001",
    median_rent = "B25064_001"
  ),
  year = 2022,
  survey = "acs5",
  output = "wide"
)

acs_state |>
  summarize(
    n_states = n(),
    cor_income_rent = cor(median_incomeE, median_rentE, use = "complete.obs")
  )
```

Note that the wide variable names get an `E` suffix for estimates and an `M` suffix for the margin of error. Both are usually returned. For most teaching purposes the `E` columns are enough; in published work the `M` columns should be acknowledged.

The decennial census uses `get_decennial()` instead. It returns counts rather than estimates and does not need a `survey` argument.

```
pop_2020 <- get_decennial(
  geography = "state",
  variables = c(total_pop = "P1_001N"),
  year = 2020,
  sumfile = "pl"
)

pop_2020
```

3.18 Extras: Qualtrics And Canvas

Survey platforms and learning management systems are also possible sources of data. The main point is that API-based imports should be treated as retrieval steps, not as code that runs every time a teaching document renders.

```
library(qualtRics)

raw_survey <- fetch_survey(
  surveyID = "YOUR-SURVEY-ID",
  label = FALSE,
  convert = FALSE
)
```

```
library(rcanvas)

courses <- get_course_list()
```

3.19 What Comes Next

The next chapter uses these data ideas to build plots. `ggplot2` works best when the data are already in a clear rectangular form and the variables needed for the plot have been inspected, selected, transformed, or summarized.

4 Core ggplot2

4.1 Learning Goals

By the end of this chapter, the main goals are:

- compare base R graphics with `ggplot2`
- understand the grammar of graphics at a practical level
- build plots from `ggplot()`, `aes()`, and geoms
- use the `+` operator to build a visualization pipeline
- distinguish mapped aesthetics from fixed aesthetics
- add scatterplot layers, smoothers, log scales, and labels
- apply built-in themes and color palettes
- use continuous color and size scales
- label points with text using `geom_text()` and `geom_text_repel()`

4.2 Load Packages

`ggplot2` is part of the tidyverse. The `scales` package is useful for readable axis labels. `RColorBrewer` is used later in the color palettes section. `ggrepel` provides `geom_text_repel()` for non-overlapping text labels.

```
library(tidyverse)
library(scales)
library(RColorBrewer)
library(ggrepel)
```

4.3 Data For This Chapter

The main examples use the local Gapminder country-year dataset in `Data/gapminder/gapminder.csv`.

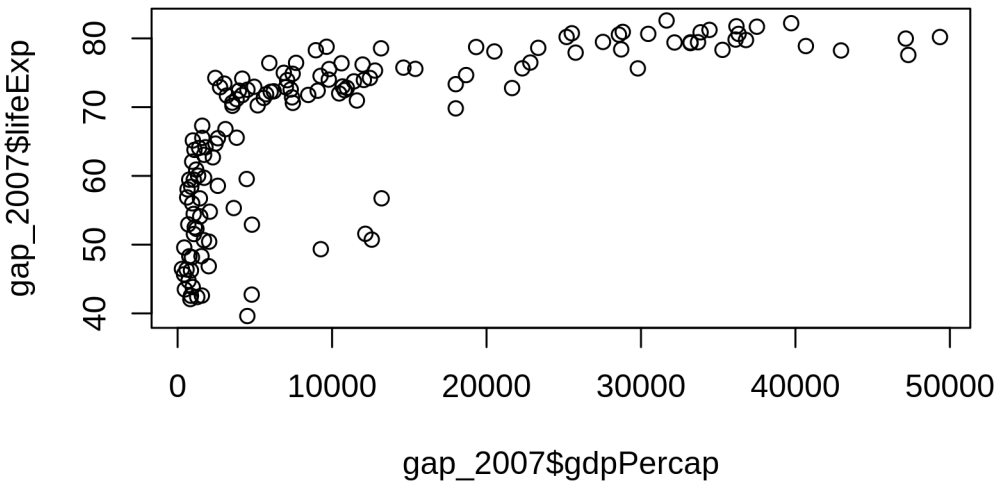
```r
gap <- read_csv("Data/gapminder/gapminder.csv", show_col_types = FALSE)

gap_2007 <- gap |>
  filter(year == max(year, na.rm = TRUE))

gap_2007
```

```
# A tibble: 142 x 6
   country     continent  year lifeExp       pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>     <dbl>     <dbl>
 1 Afghanistan Asia       2007    43.8  31889923      975.
 2 Albania     Europe     2007    76.4   3600523     5937.
 3 Algeria     Africa     2007    72.3  33333216     6223.
 4 Angola      Africa     2007    42.7  12420476     4797.
 5 Argentina   Americas   2007    75.3  40301927    12779.
 6 Australia   Oceania    2007    81.2  20434176    34435.
 7 Austria     Europe     2007    79.8   8199783    36126.
 8 Bahrain     Asia       2007    75.6    708573    29796.
 9 Bangladesh  Asia       2007    64.1 150448339     1391.
10 Belgium     Europe     2007    79.4  10392226    33693.
# i 132 more rows
```

4.4 Comparing Base R Graphics And `ggplot2`

Base R has plotting functions built in. They are useful for quick checks and appear in many older examples.

```r
plot(gap_2007$gdpPercap, gap_2007$lifeExp)
```



That plot works, but the labels are not very informative, the x-axis is hard to read, and customization quickly becomes a matter of remembering many arguments.

A base R histogram shows a similar pattern:

```
hist(gap_2007$lifeExp, breaks = 8)
```

The problem is not that base R cannot make plots. The problem is that complex plots become difficult to revise. `ggplot2` gives us a more structured way to build visualizations.

4.5 The Grammar Of Graphics

A `ggplot2` figure is built in layers. Every plot needs three minimum elements before any layer can sit on top of them:

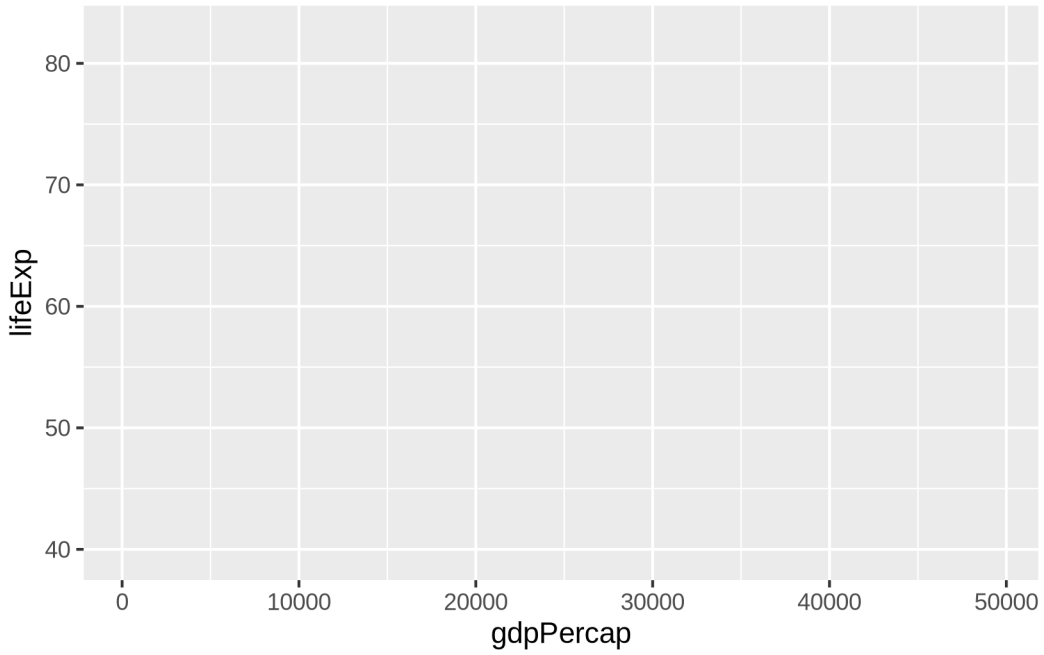
- data: the table being plotted
- mappings: the links between variables and visual features
- geoms: the visual marks, such as points, lines, bars, or histograms

This separation is what makes `ggplot2` powerful. You can change the data, the mapping, or the geom without rewriting the whole plot from scratch.

4.6 The Base Layer: `ggplot()` And `aes()`

The `ggplot()` function starts a plot. The `aes()` function defines aesthetic mappings. Aesthetic mappings connect variables in the data to visual properties such as x-position, y-position, color, size, shape, or transparency.

```r
ggplot(
  data = gap_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)
)
```

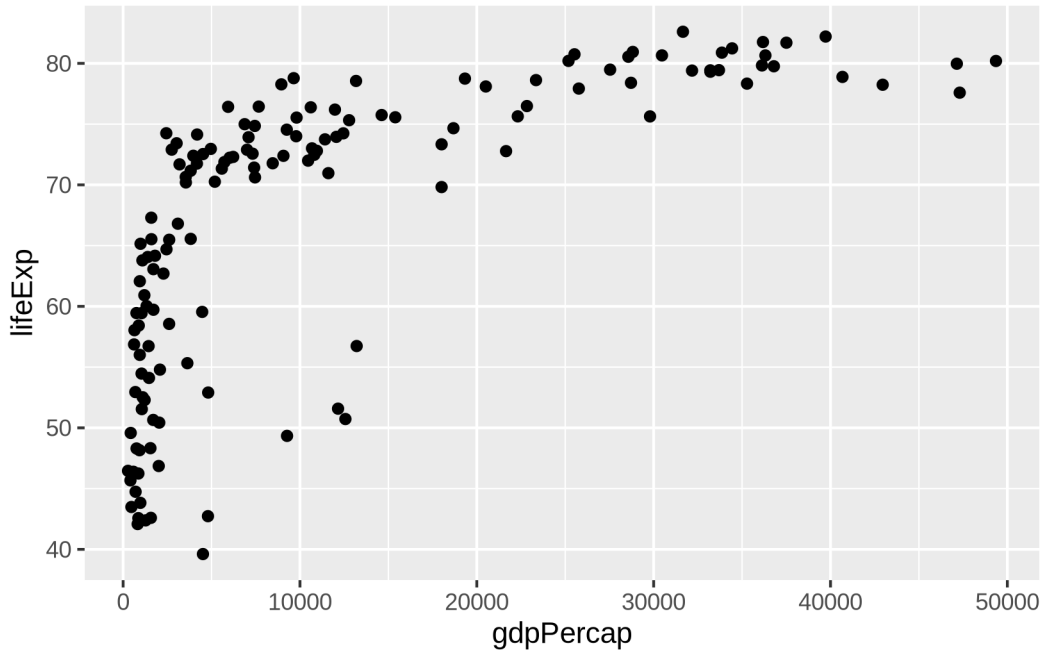


This is the **base layer**. The data and mappings are set, the axes are drawn, but nothing is plotted yet because no geom has been added. Every later layer — points, lines, smoothers, labels, scales — sits on top of this base.

4.7 Add A Geom Layer

`geom_point()` draws points. Because each row in `gap_2007` is a country, each point is a country.

```r
ggplot(
  data = gap_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)
) +
  geom_point()
```

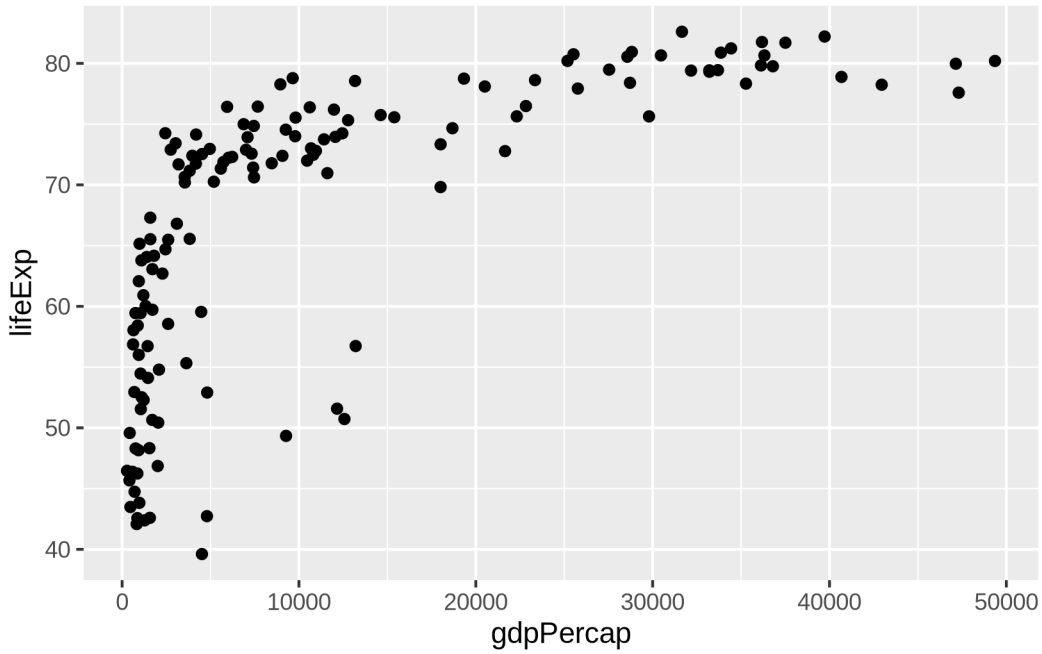


The + operator means "add the next plot component." In `ggplot2`, line breaks usually put + at the end of the line so R knows the plot is not finished.

4.8 Saving A Plot Object

It is often useful to store the base plot and add layers later.

```
p_gap <- ggplot(
  data = gap_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)
)

p_gap +
  geom_point()
```



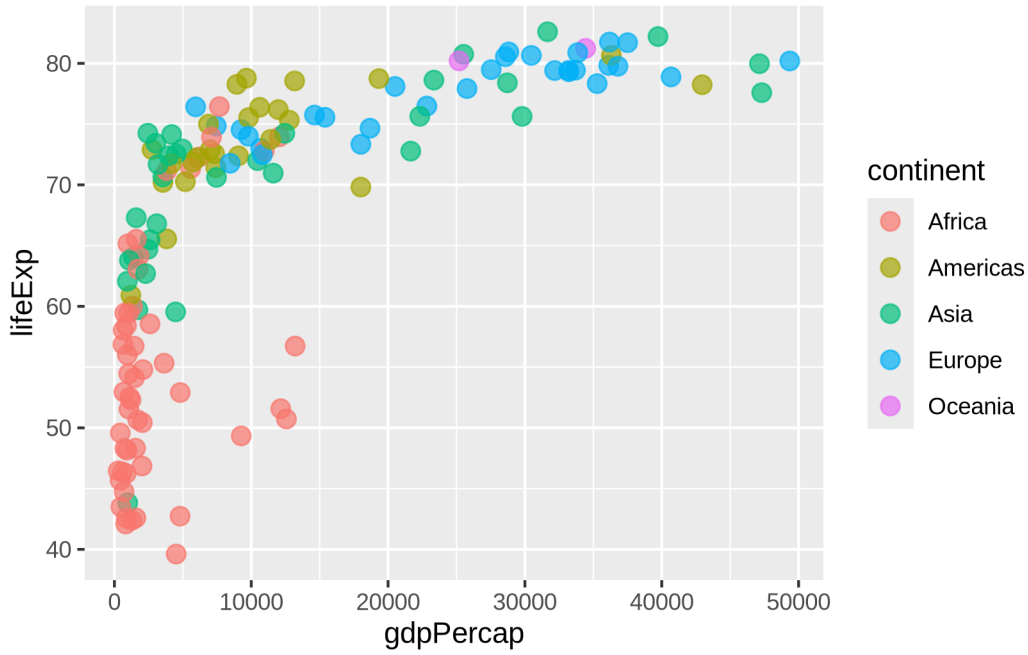
The object `p_gap` stores the data and default mappings. It does not lock the plot into one final form. We can keep adding layers.

4.9 Mapping Versus Setting

One of the most important `ggplot2` distinctions is the difference between mapping and setting.

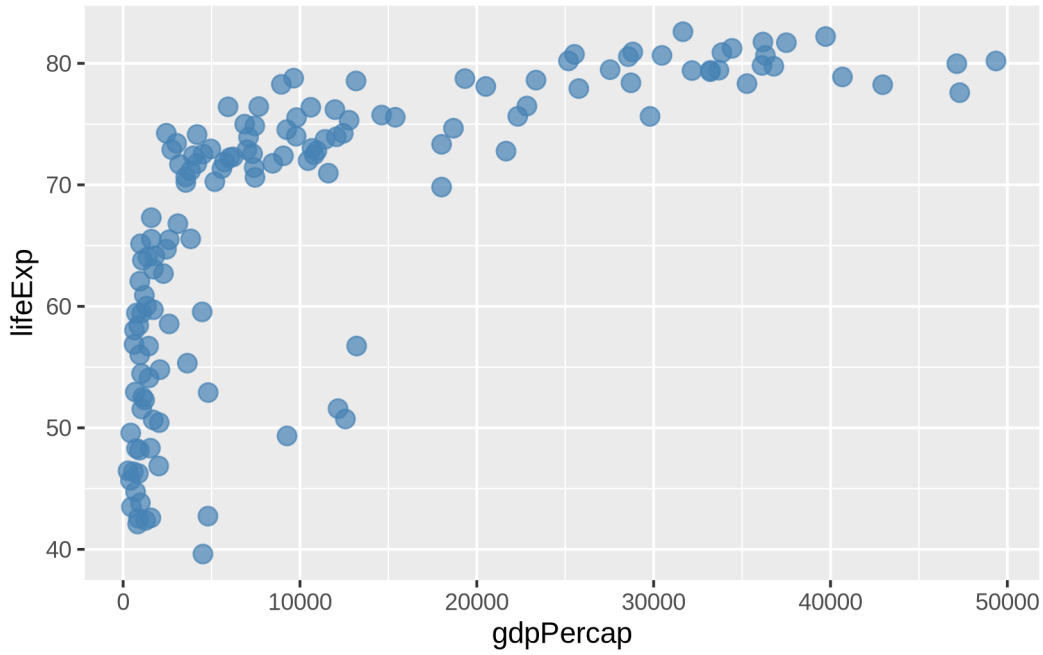
When an aesthetic is mapped, it goes inside `aes()` and represents data. In the next plot, color means continent.

```
p_gap +
  geom_point(aes(color = continent), size = 3, alpha = 0.7)
```



When an aesthetic is set, it stays fixed and goes outside `aes()`. In the next plot, all points are the same color. The color does not encode information.

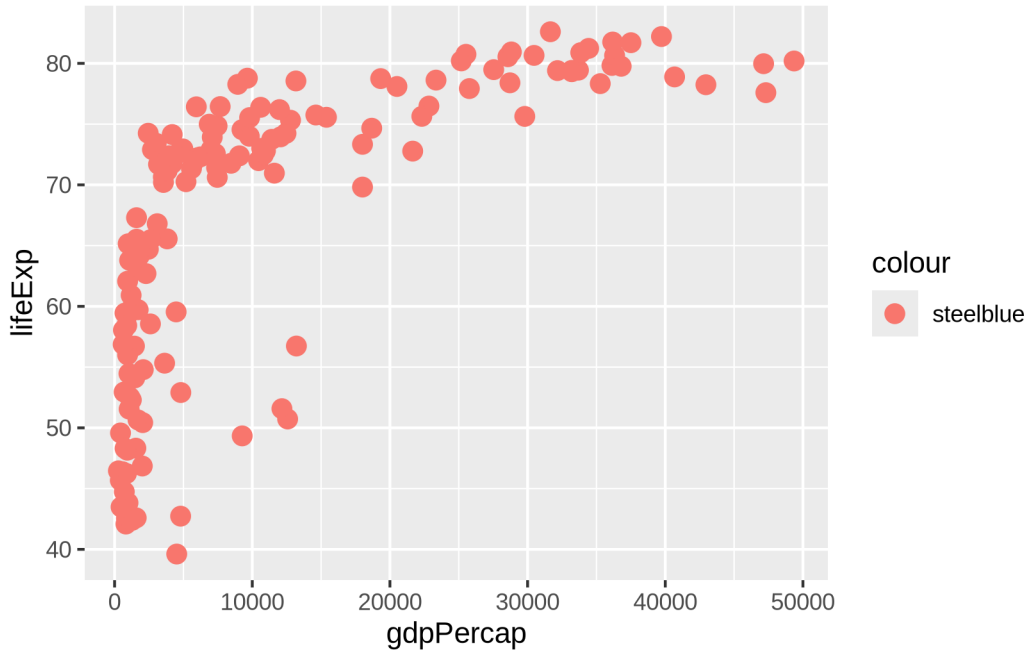
```
p_gap +
  geom_point(color = "steelblue", size = 3, alpha = 0.7)
```



This distinction matters because viewers interpret visual differences. If color varies because it is mapped to a variable, it carries meaning. If color is fixed, it is a design choice.

A common mistake is to put a fixed color inside `aes()`:

```
p_gap +
  geom_point(aes(color = "steelblue"), size = 3)
```



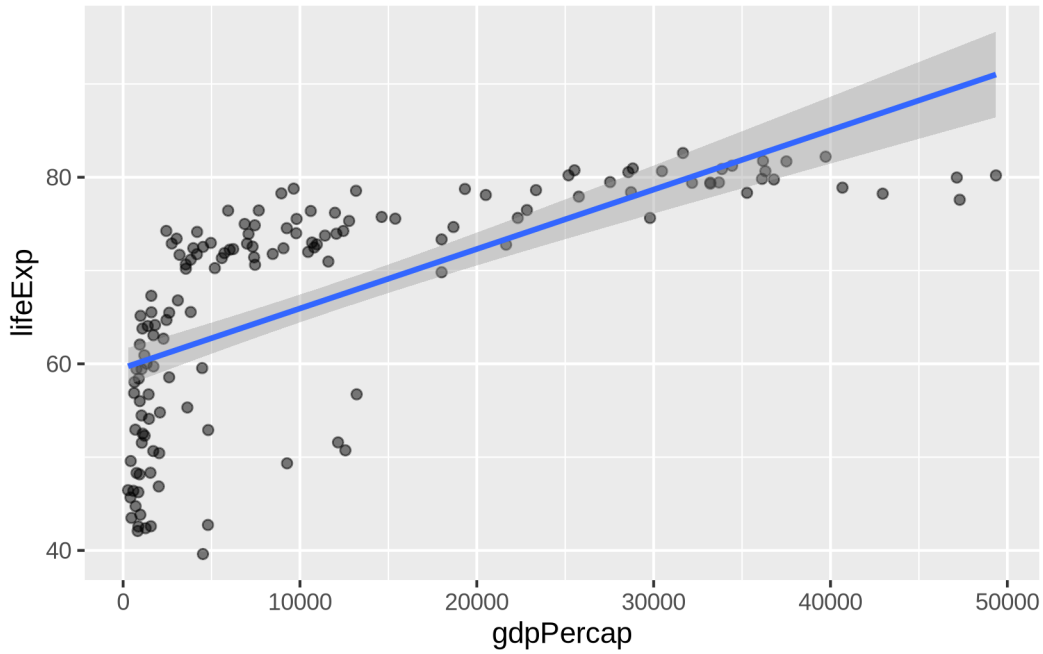
In that plot, `"steelblue"` is treated like a one-value variable, not as a fixed color instruction. The result is a legend that does not help the reader. The fix is to move fixed settings outside `aes()`.

4.10 Smoothers

Scatterplots show individual observations. Smoothers add a summary of the relationship.

A linear smoother fits a straight line:

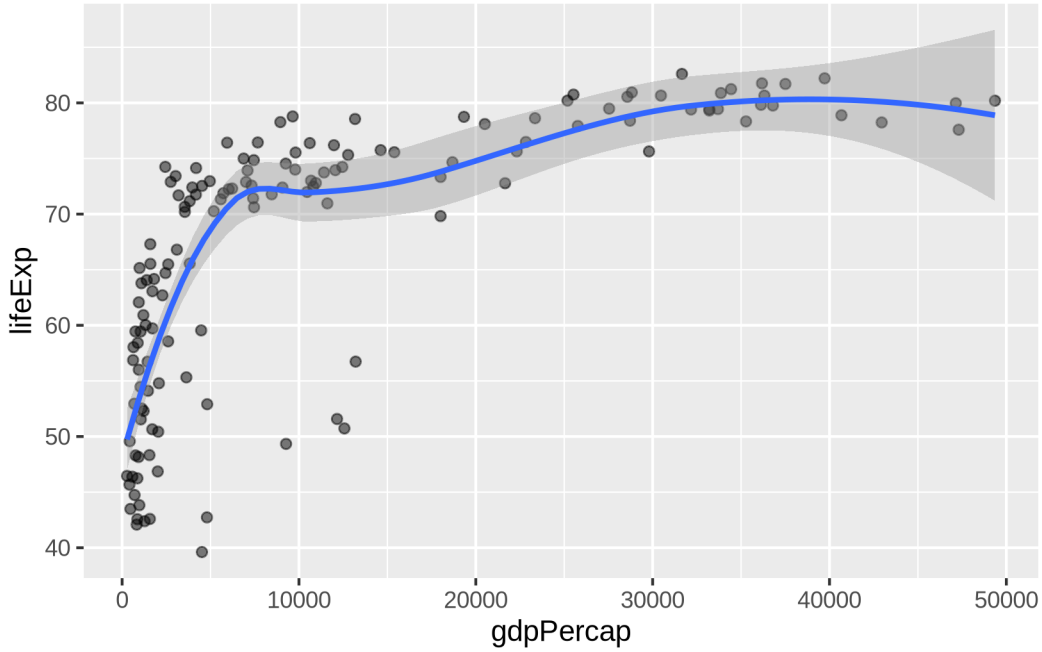
```
p_gap +
  geom_point(alpha = 0.5) +
  geom_smooth(method = "lm", formula = y ~ x)
```



`method = "lm"` asks for a linear model. The `formula = y ~ x` part says to model the y-axis variable as a function of the x-axis variable. In this simple two-variable plot, that is the default relationship anyway, but writing it explicitly prevents `ggplot2` from printing an informational message about the formula it is using.

A loess smoother is more flexible:

```
p_gap +
  geom_point(alpha = 0.5) +
  geom_smooth(method = "loess", formula = y ~ x)
```



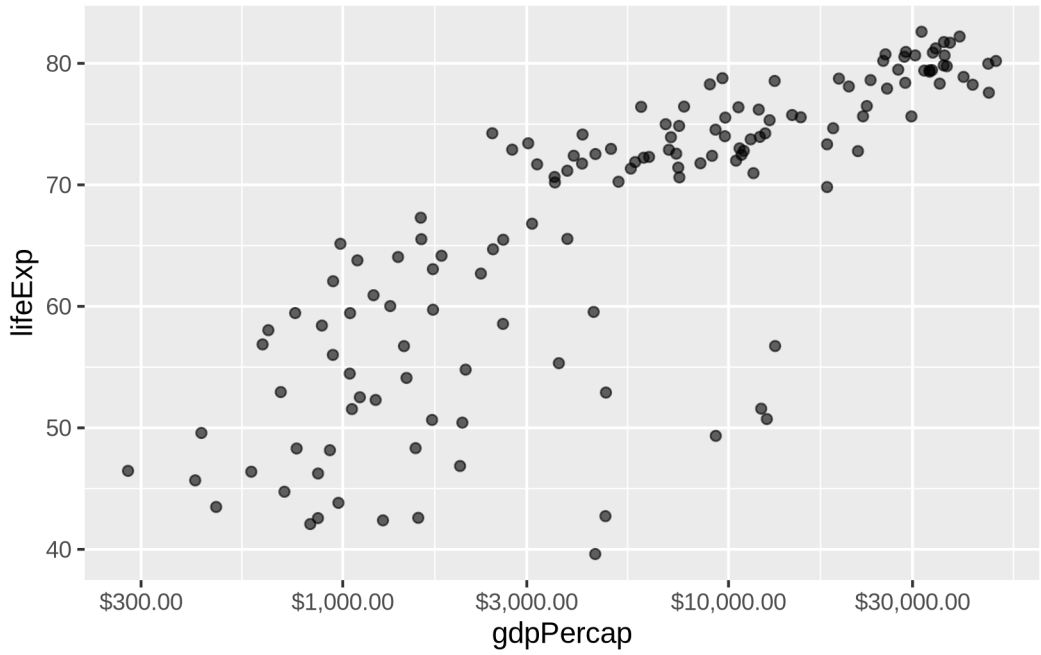
`method = "loess"` asks for a locally smoothed curve rather than a straight line. It can follow curvature, but it is still a descriptive summary of the plotted relationship, not proof of a causal pattern.

Neither smoother is automatically correct. A straight line is easy to interpret but may miss curvature. A loess curve can reveal bends in the data but is more descriptive than explanatory. The right choice depends on the purpose of the plot.

4.11 Log Scales

GDP per capita is usually highly skewed. A few very high values can compress most observations into the left side of the plot. A log scale spreads the lower and middle values out while keeping the order intact.

```
p_gap +
  geom_point(alpha = 0.6) +
  scale_x_log10(labels = label_dollar())
```

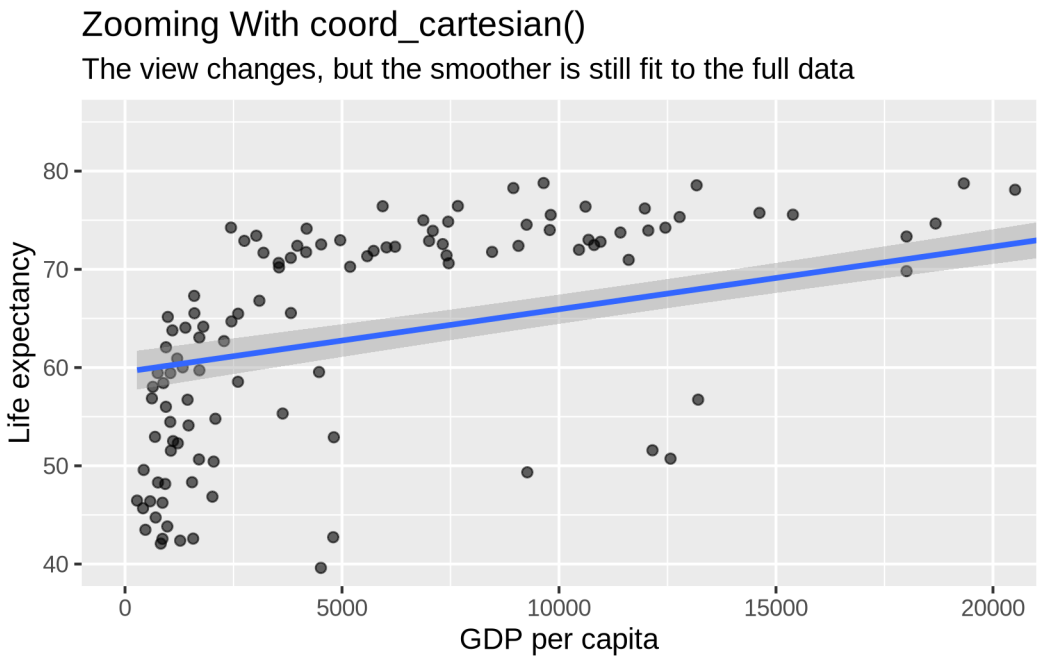


The command `scale_x_log10()` changes the x-axis, not the underlying data object. The argument `labels = label_dollar()` formats the axis as dollars instead of scientific notation.

4.12 Zooming Without Dropping Data

Sometimes the goal is to zoom in on part of a plot. `coord_cartesian()` changes the visible plotting window without removing observations from the data before statistics are calculated.

```
p_gap +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x) +
  coord_cartesian(xlim = c(0, 20000), ylim = c(40, 85)) +
  labs(
    title = "Zooming With coord_cartesian()",
    subtitle = "The view changes, but the smoother is still fit to the full data",
    x = "GDP per capita",
    y = "Life expectancy"
  )
```

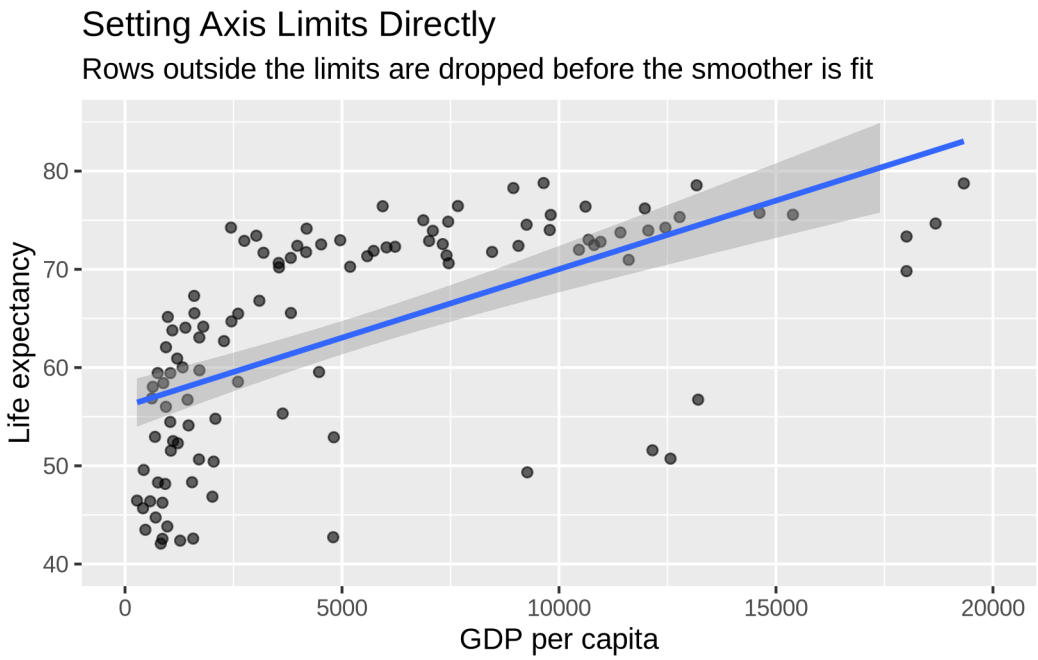


This differs from filtering the data before plotting. Filtering changes which rows are used. `coord_cartesian()` keeps the data and changes only the displayed region.

4.13 Setting Axis Limits Directly

Axis limits can also be set directly with scale functions. This looks similar to zooming, but it is not the same. Direct scale limits drop observations outside the limits before the plot is drawn. If a smoother or other statistical layer is included, it is fit only to the remaining visible data.

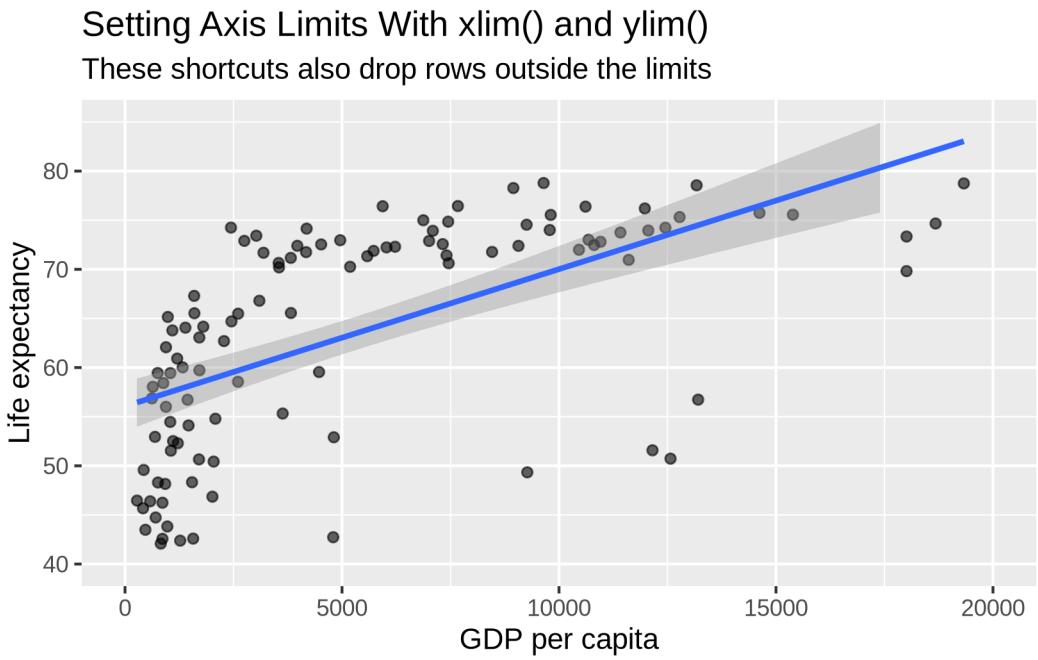
```
p_gap +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x) +
  scale_x_continuous(limits = c(0, 20000)) +
  scale_y_continuous(limits = c(40, 85)) +
  labs(
    title = "Setting Axis Limits Directly",
    subtitle = "Rows outside the limits are dropped before the smoother is fit",
    x = "GDP per capita",
    y = "Life expectancy"
  )
```



The warnings are expected: `ggplot2` is reporting that observations outside the requested scale limits were removed.

The shortcut functions `xlim()` and `ylim()` do the same kind of direct scale limiting.

```
p_gap +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x) +
  xlim(0, 20000) +
  ylim(40, 85) +
  labs(
    title = "Setting Axis Limits With xlim() and ylim()",
    subtitle = "These shortcuts also drop rows outside the limits",
    x = "GDP per capita",
    y = "Life expectancy"
  )
```

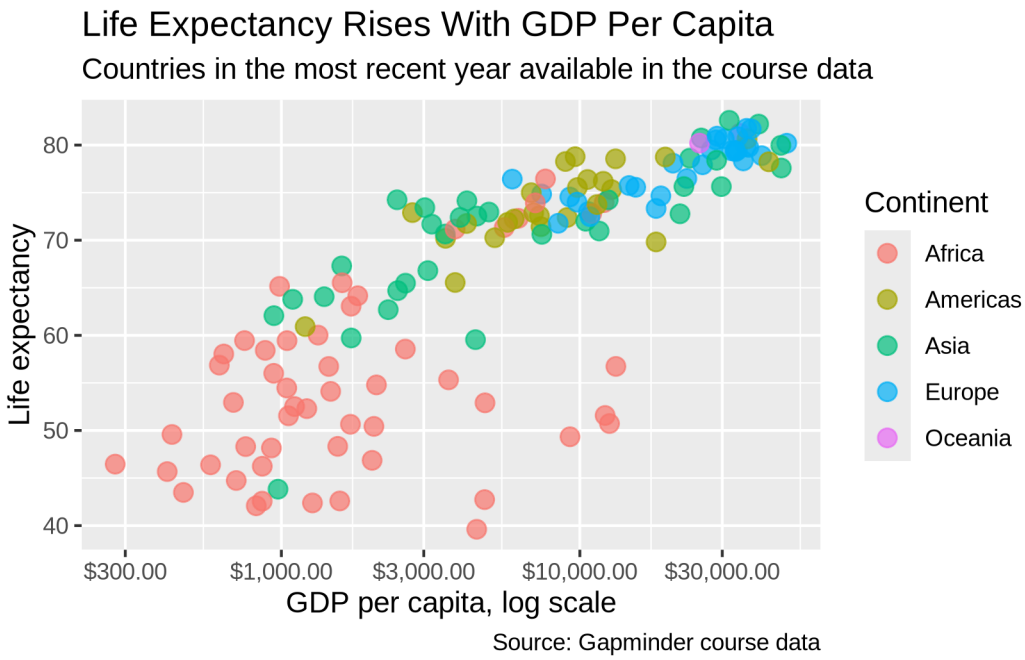


For most zooming tasks, `coord_cartesian()` is safer because it changes the view without changing which rows are used by the statistical layers. Direct scale limits are useful when the intention is to remove values outside the range from both the display and the plotted calculations.

4.14 Labels With `labs()`

Plots should be understandable outside the code chunk. `labs()` adds titles, subtitles, axis labels, legend labels, and captions.

```
p_gap +
  geom_point(aes(color = continent), alpha = 0.7, size = 3) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "Life Expectancy Rises With GDP Per Capita",
    subtitle = "Countries in the most recent year available in the course data",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent",
    caption = "Source: Gapminder course data"
  )
```

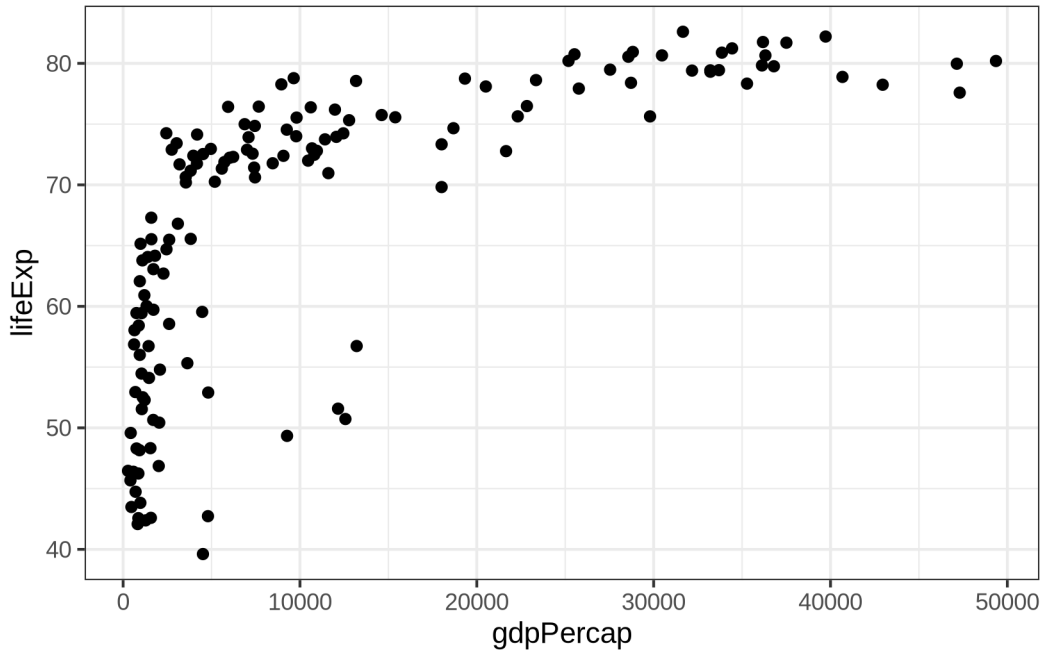


The title tells the reader the main subject. The subtitle adds context. Axis labels state the meaning of the plotted variables. The legend title clarifies the mapped aesthetic. The caption is a natural place for data source information.

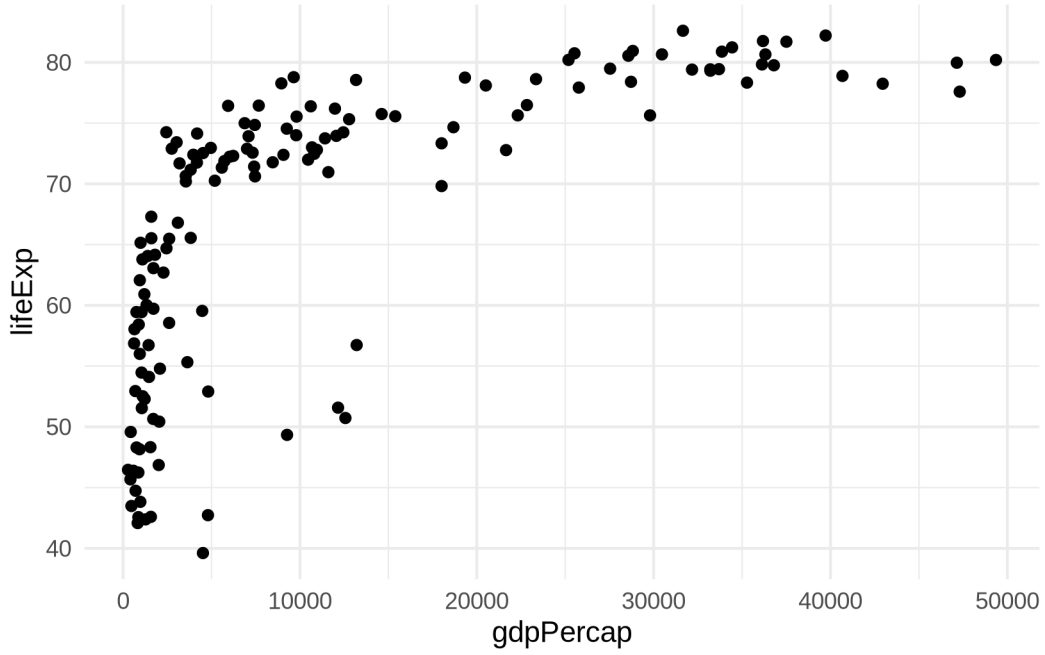
4.15 Themes

Themes control the non-data parts of the plot: background, grid lines, axis text, title placement, legend placement, and similar elements. A theme does not change the data being shown, but it can change how easy the plot is to read.

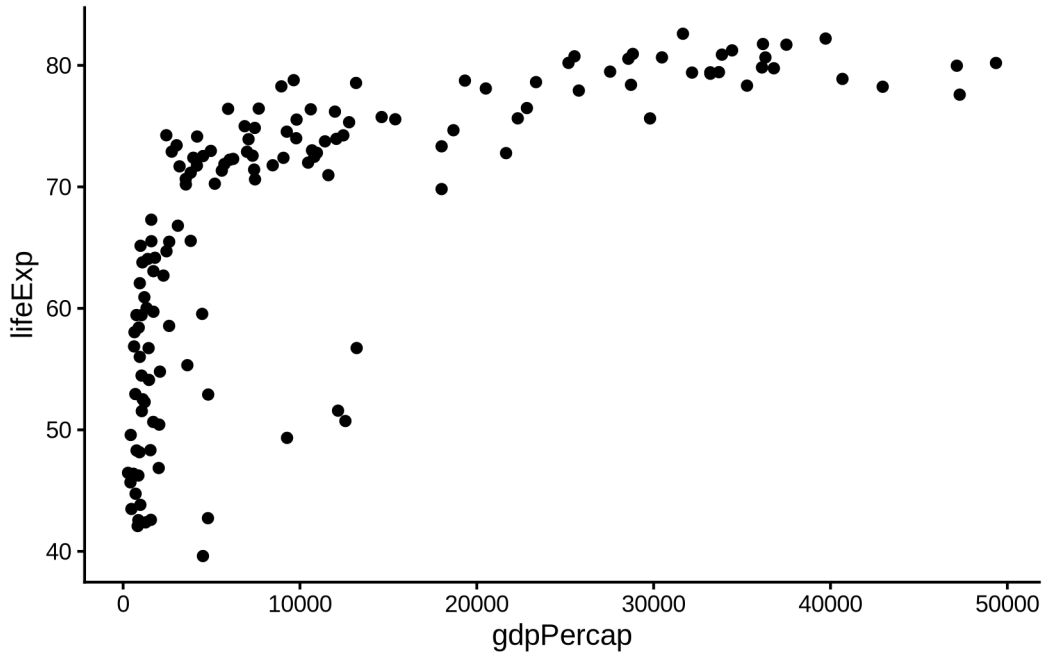
```
p_gap + geom_point() + theme_bw()
```




```
p_gap + geom_point() + theme_minimal()
```

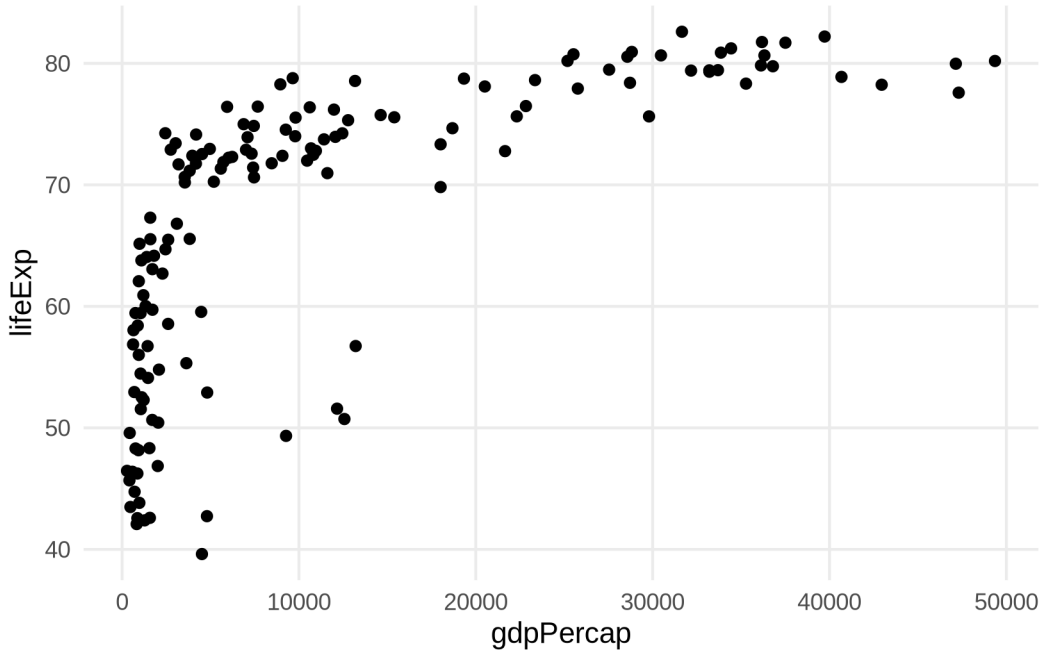


```
p_gap + geom_point() + theme_classic()
```



Built-in themes are good starting points. Individual elements can then be adjusted with `theme()`.

```
p_gap +
  geom_point() +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    plot.subtitle = element_text(size = 11),
    axis.title = element_text(size = 11),
    panel.grid.minor = element_blank()
  )
```

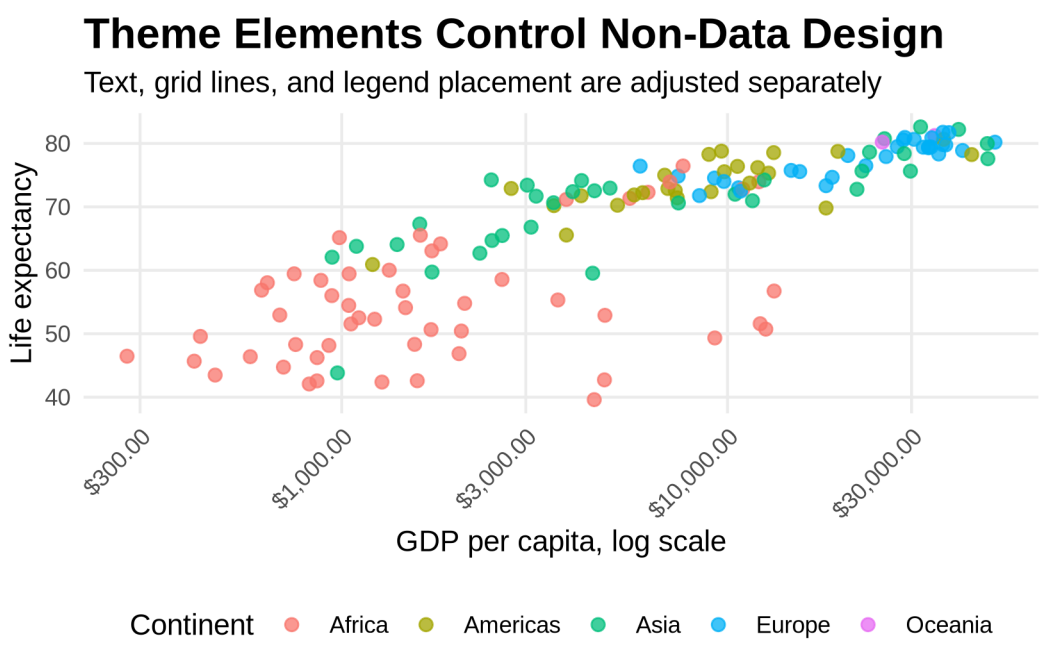


The `theme()` function changes individual non-data elements. The element functions describe what kind of thing is being changed:

- `element_text()` changes text, such as titles, axis labels, tick labels, and legend text.
- `element_blank()` removes an element completely.
- `element_line()` changes lines, such as grid lines and axis lines.
- `legend.position` controls where the legend appears.

```
p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "Theme Elements Control Non-Data Design",
    subtitle = "Text, grid lines, and legend placement are adjusted separately",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent"
  ) +
  theme_minimal() +
  theme(
    plot.title = element_text(face = "bold", size = 16),
    axis.text.x = element_text(angle = 45, hjust = 1),
    panel.grid.minor = element_blank(),
```

```
    legend.position = "bottom"
  )
```



Rotating axis text is useful when labels are long or crowded. Removing minor grid lines can make a figure less busy. Moving the legend to the bottom can help when a plot is wide and there is more horizontal than vertical room.

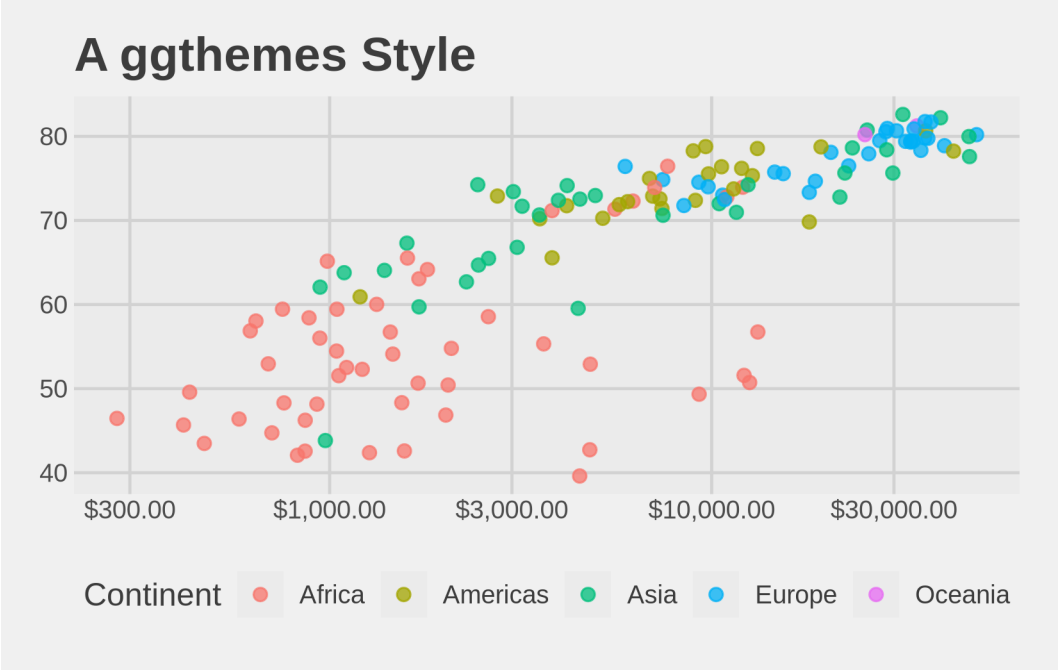
4.16 Theme Packages

Additional packages provide complete theme systems. These are useful when a plot needs to match a report style, publication style, or visual identity.

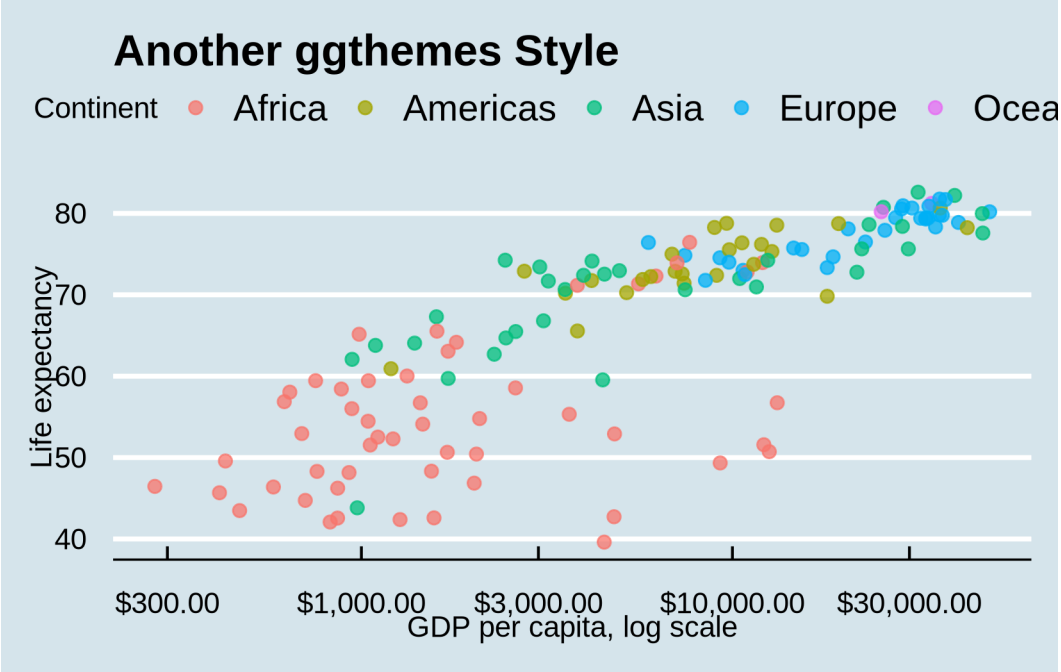
The `ggthemes` package includes themes that imitate familiar publication and media styles.

```
p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "A ggthemes Style",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent"
```

```
  ) +
  ggthemes::theme_fivethirtyeight()
```



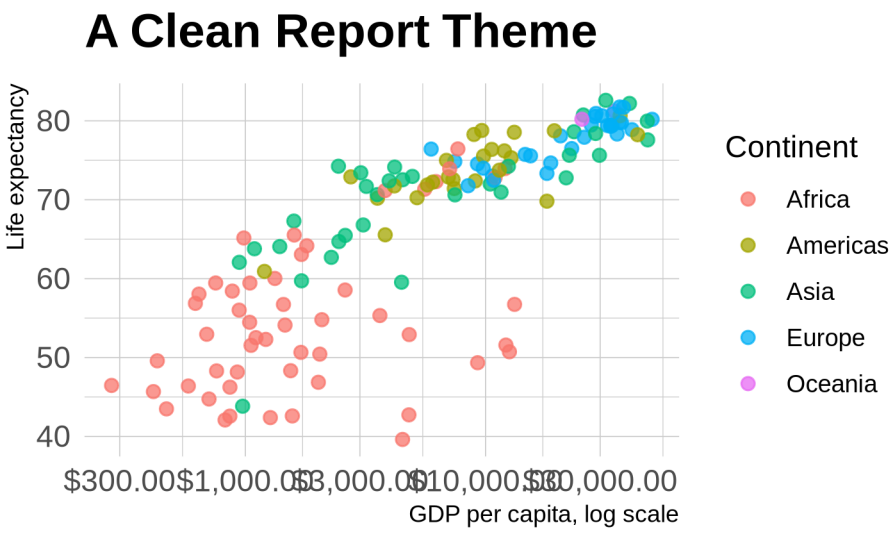

```
p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "Another ggthemes Style",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent"
  ) +
  ggthemes::theme_economist()
```



The hrbrthemes package focuses on restrained typography and clean spacing. Its themes tend to use lighter grid lines, larger readable text, and a report-like visual style. They are also simply aesthetically pleasing, which matters: an attractive figure is often more inviting, more memorable, and easier to take seriously. This is useful when the figure should feel more like an editorial or analytic graphic than a default software output.

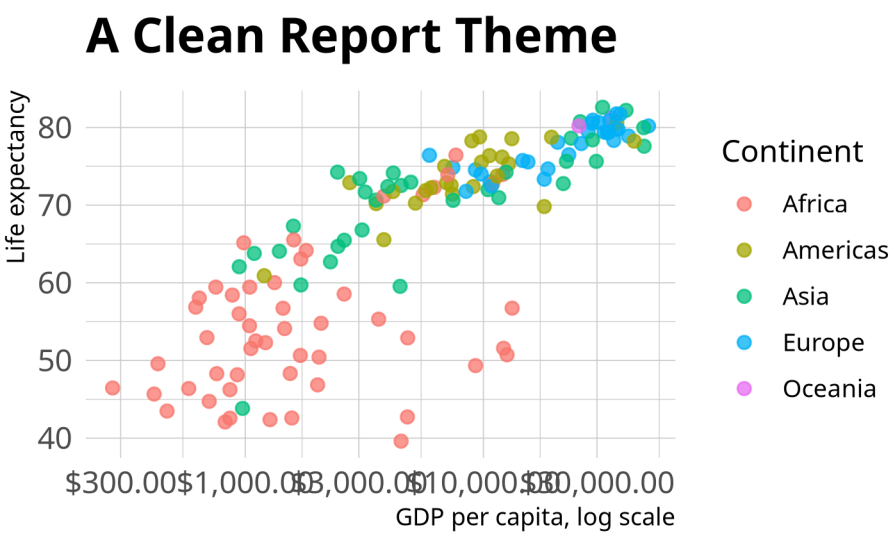
Typography matters because plots are read as documents. Titles, subtitles, axis labels, legends, and captions all guide interpretation. A theme with better spacing and text hierarchy can make the same data feel more deliberate and easier to scan.

```
p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "A Clean Report Theme",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent"
  ) +
  hrbrthemes::theme_ipsum(base_family = "sans")
```

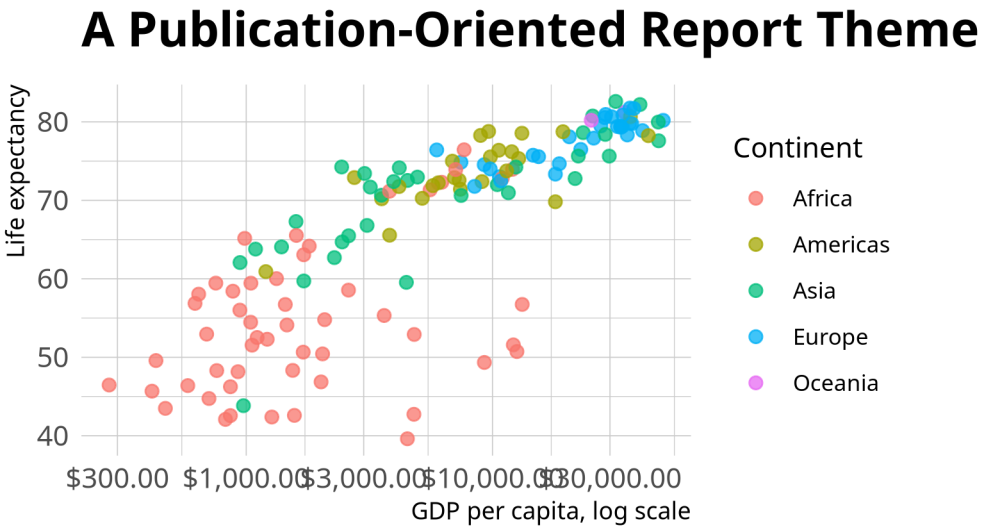


The same plot can be sent through related `hrbrthemes` themes. The point is not that one theme is objectively correct. The point is that theme choice changes the typography, spacing, grid emphasis, and overall feel of the figure.

```
p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "A Clean Report Theme",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent"
  ) +
  hrbrthemes::theme_ipsum_rc()
```




```
p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "A Publication-Oriented Report Theme",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent"
  ) +
  hrbrthemes::theme_ipsum_pub()
```



The `rc` names refer to Roboto Condensed, the font family these themes were designed around. The first variant changes the grid and axes to make the style visibly different. The second uses a publication-oriented theme with a different typographic feel.

These theme packages change the overall style, but the plot still needs clear mappings, labels, scales, and a readable comparison.

4.17 Color Palettes

Default colors are fine for early drafts. For more intentional figures, choose palettes that match the type of data.

- Qualitative palettes distinguish categories with no natural order.
- Sequential palettes show ordered values from low to high.
- Diverging palettes show distance from a meaningful midpoint.

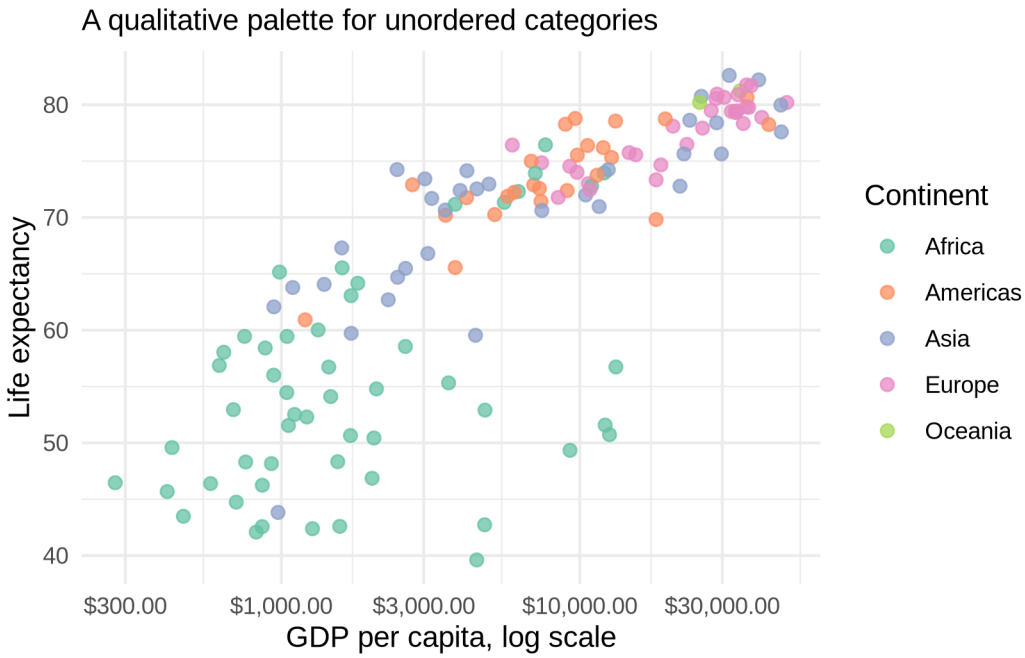
Start from a plot that maps a categorical variable to color:

```
p_gap_color <- p_gap +
  geom_point(aes(color = continent), alpha = 0.75, size = 2) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    x = "GDP per capita, log scale",
```

```
    y = "Life expectancy",
    color = "Continent"
  ) +
  theme_minimal()

p_gap_color
```

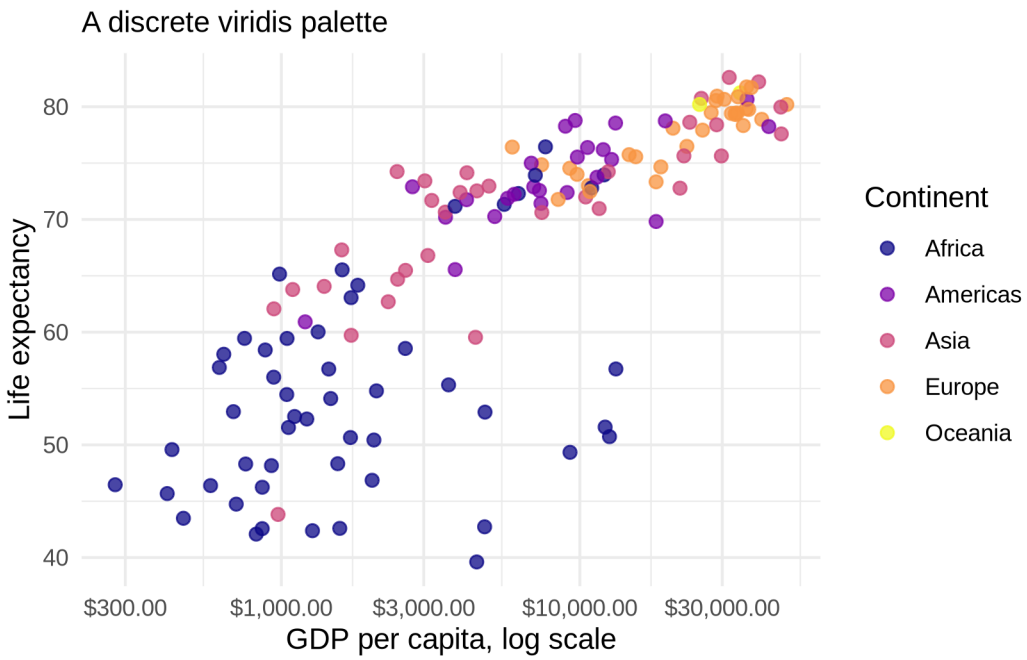
RColorBrewer provides many named palettes.

```
p_gap_color +
  scale_color_brewer(palette = "Set2") +
  labs(subtitle = "A qualitative palette for unordered categories")
```



The `viridis` scales are also useful because they are designed to be readable in color and when converted to grayscale.

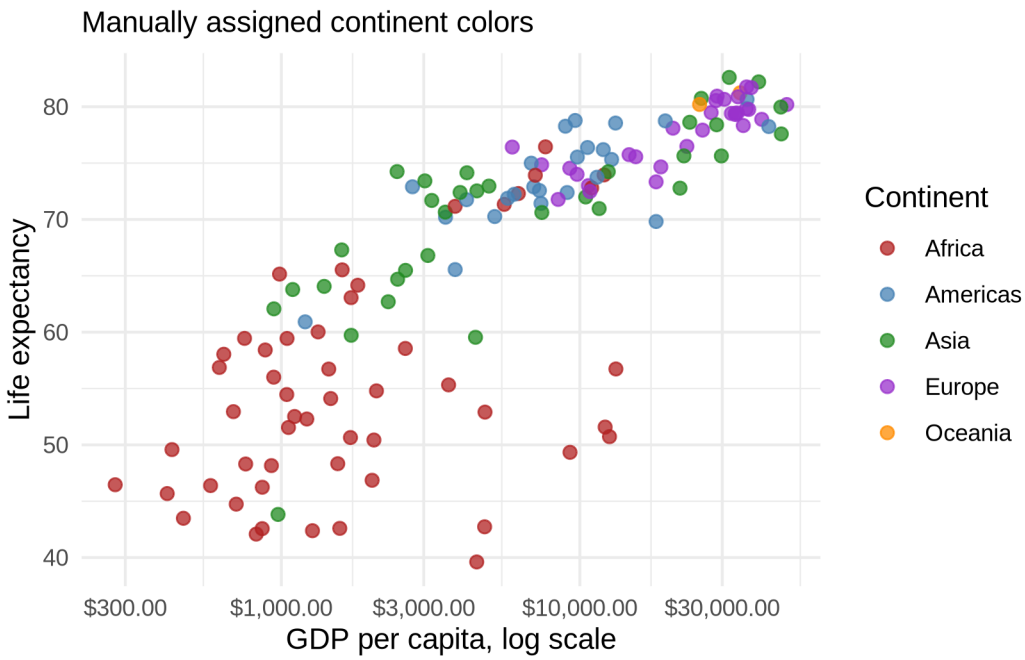
```
p_gap_color +
  scale_color_viridis_d(option = "plasma") +
  labs(subtitle = "A discrete viridis palette")
```



Diverging palettes are useful when a variable has a meaningful midpoint, such as zero, parity, or an average. They are especially natural in maps because the color immediately shows which places are above or below the midpoint. Chapter 12 returns to this idea with a choropleth map.

Manual color scales are useful when specific categories should keep specific colors across several plots.

```
continent_colors <- c(
  "Africa" = "firebrick",
  "Americas" = "steelblue",
  "Asia" = "forestgreen",
  "Europe" = "darkorchid",
  "Oceania" = "darkorange"
)

p_gap_color +
  scale_color_manual(values = continent_colors) +
  labs(subtitle = "Manually assigned continent colors")
```

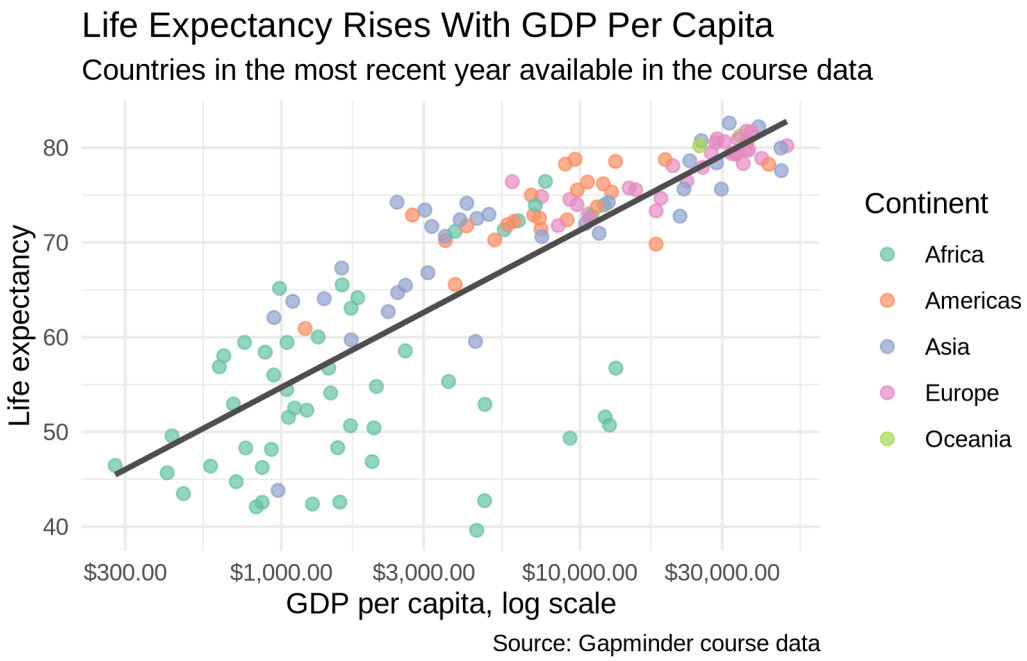


4.18 Putting It All Together

Each piece of this chapter has been introduced as one layer at a time. The full plot stacks them in order: base layer, point geom, smoother, scale, labels, palette, theme.

```
ggplot(
  data = gap_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)
) +
  geom_point(aes(color = continent), alpha = 0.7, size = 2) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "gray30") +
  scale_x_log10(labels = label_dollar()) +
  scale_y_continuous(breaks = seq(30, 90, by = 10)) +
  scale_color_brewer(palette = "Set2") +
  labs(
    title = "Life Expectancy Rises With GDP Per Capita",
    subtitle = "Countries in the most recent year available in the course data",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Continent",
    caption = "Source: Gapminder course data"
```

```
) +
theme_minimal()
```



Read the code from the top:

- `ggplot(...)` sets the base layer with data and mappings.
- `geom_point(...)` draws each country as a point and maps continent to color.
- `geom_smooth(...)` adds a single linear summary across all continents.
- `scale_x_log10(...)` transforms the x-axis so the lower values are easier to read.
- `scale_color_brewer(...)` swaps the default palette for a qualitative one.
- `labs(...)` adds the title, subtitle, axis labels, legend title, and caption.
- `theme_minimal()` controls the non-data styling.

If the plot needs to change, the line responsible for that change is usually easy to identify.

`scale_y_continuous(breaks = seq(30, 90, by = 10))` controls where tick marks appear on the y-axis. The `seq()` function generates a sequence from 30 to 90 in steps of 10. Without it, R chooses the breaks automatically, which is often fine, but explicit breaks can make the grid easier to read when the data spans a predictable range.

4.19 Saving Plots

`ggsave()` saves the most recently printed plot to a file. The two most useful arguments are `filename` and the dimensions `width` and `height` in inches.

```
ggsave(filename = "Plots/gdp_life_expectancy.png", width = 8, height = 5)
ggsave(filename = "Plots/gdp_life_expectancy.pdf", width = 8, height = 5)
```

By default, `ggsave()` saves the last plot displayed. To save a specific plot object, use the `plot` argument:

```
my_plot <- p_gap + geom_point()
ggsave(filename = "Plots/my_plot.png", plot = my_plot, width = 8, height = 5)
```

PNG is a good choice for slides, web pages, and Word documents. PDF is better for print and LaTeX documents because it is a vector format and stays sharp at any size.

4.20 Short Exercise

The built-in `mtcars` dataset is always available. Convert it to a tibble first so the car names become a column:

```
cars_tbl <- mtcars |>
  as_tibble(rownames = "car")
```

Using `cars_tbl`, build a scatterplot with:

1. `hp` on the x-axis
2. `mpg` on the y-axis
3. `cyl` mapped to color
4. a linear smoother with `se = FALSE`
5. readable labels with `labs()`

```
# Write your code here.
```

4.21 Text as Labels

The core scatterplot pipeline above uses titles and axis labels to make the figure readable. Sometimes the data points themselves also need labels. Points can be labeled with text using the `label` aesthetic. `geom_text()` draws text at the mapped x and y position for each row in the data.

Text labels are most useful when the units have meaningful names — country codes, state abbreviations, party names — that help the reader identify specific observations.

State-level data is a natural fit. The Correlates of State Policy Project tracks public opinion and policy outputs across all U.S. states over several decades. Two variables are especially useful as a starting point:

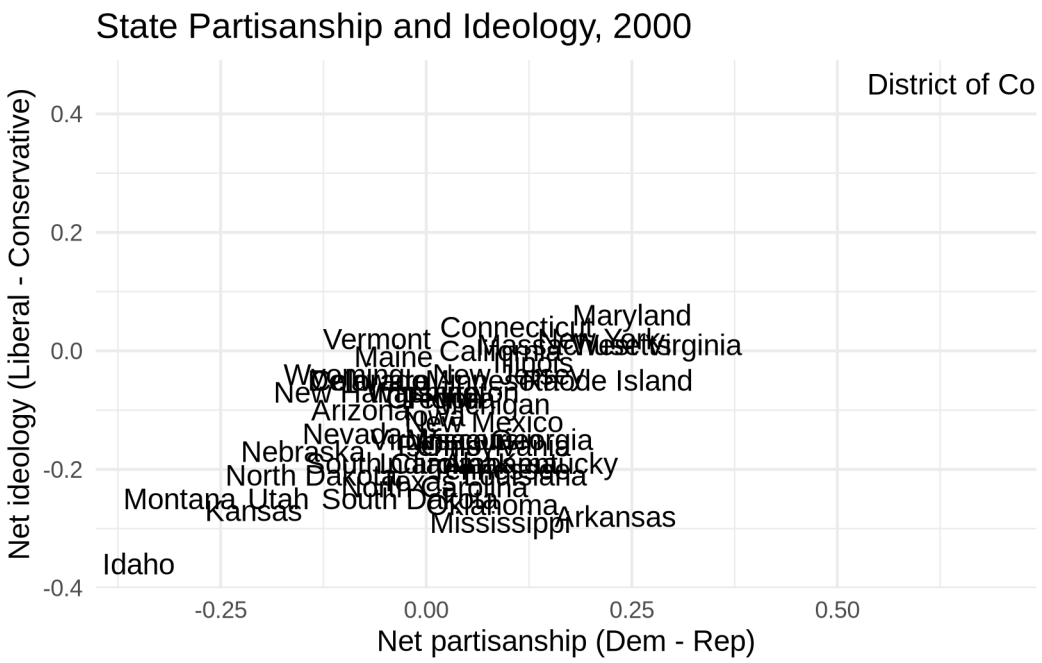
- `pid`: net partisanship — Democratic minus Republican party identification, as a share of the adult population
- `ideo`: net ideology — liberal minus conservative self-identification

We filter to a single year so each state appears once.

```
states <- read_csv("Data/state_policy/cspp_states.csv", show_col_types = FALSE)

states_2000 <- states |> filter(year == 2000)
```

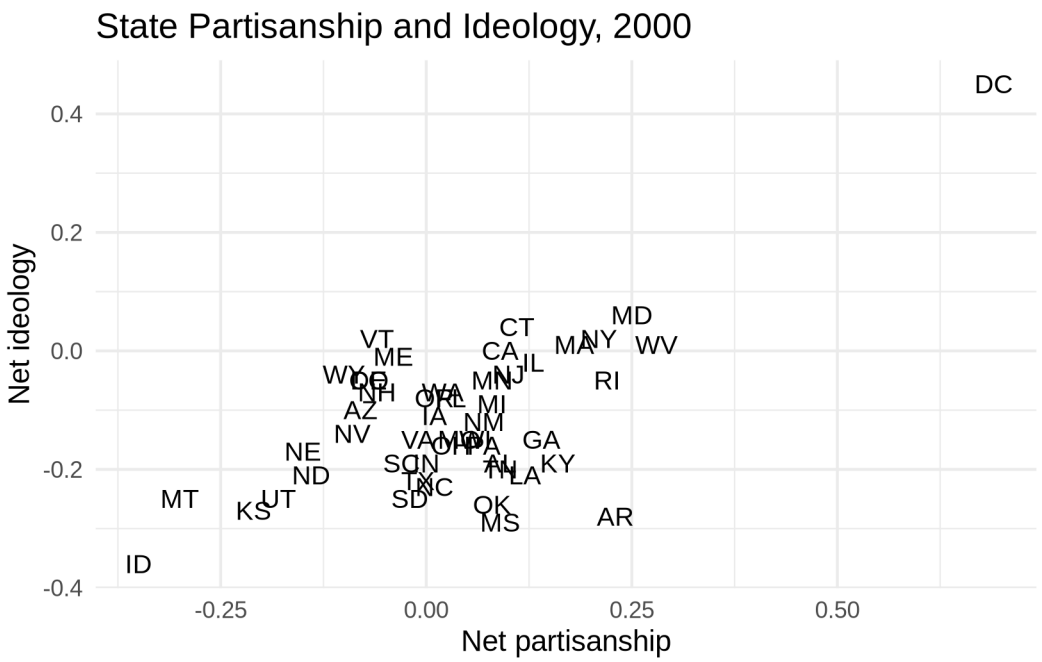
The first version maps full state names to `label`. Every point gets the state name written at its location.

```
ggplot(states_2000, aes(x = pid, y = ideo)) +
  geom_text(aes(label = state)) +
  labs(
    title = "State Partisanship and Ideology, 2000",
    x = "Net partisanship (Dem - Rep)",
    y = "Net ideology (Liberal - Conservative)"
  ) +
  theme_minimal()
```



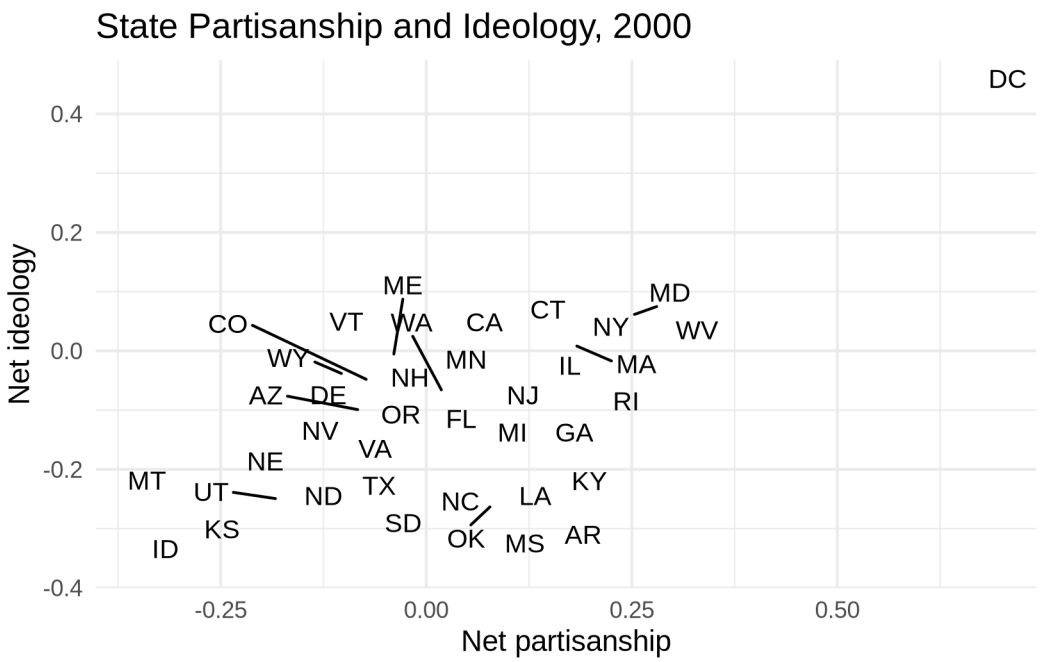
Full names crowd together badly. Switching to two-letter abbreviations is better.

```
ggplot(states_2000, aes(x = pid, y = ideo)) +
  geom_text(aes(label = st), size = 3.5) +
  labs(
    title = "State Partisanship and Ideology, 2000",
    x = "Net partisanship",
    y = "Net ideology"
  ) +
  theme_minimal()
```



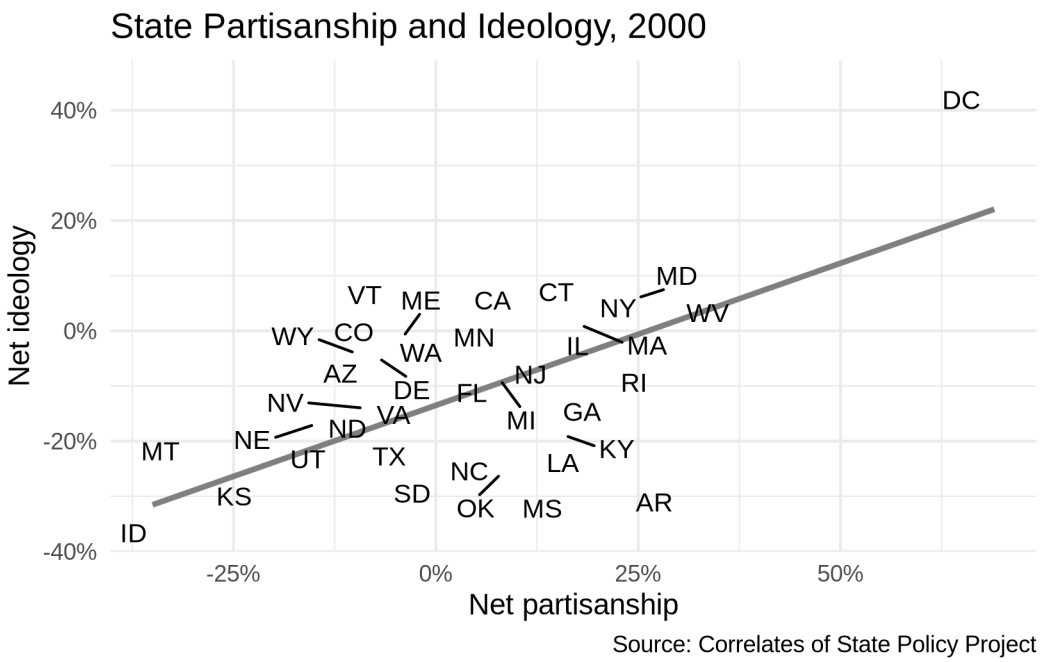
Some abbreviations still overlap where states cluster near each other. `geom_text_repel()` from the `ggrepel` package moves labels slightly to reduce overlap and draws a short line segment back to the point when it has to move far.

```
ggplot(states_2000, aes(x = pid, y = ideo)) +
  geom_text_repel(aes(label = st), size = 3.5) +
  labs(
    title = "State Partisanship and Ideology, 2000",
    x = "Net partisanship",
    y = "Net ideology"
  ) +
  theme_minimal()
```



A linear smoother added underneath shows the overall relationship. The `label` aesthetic is defined only in `geom_text_repel()`, so it does not apply to `geom_smooth()`.

```
ggplot(states_2000, aes(x = pid, y = ideo)) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "gray50") +
  geom_text_repel(aes(label = st), size = 3.5) +
  scale_x_continuous(labels = label_percent()) +
  scale_y_continuous(labels = label_percent()) +
  labs(
    title = "State Partisanship and Ideology, 2000",
    x = "Net partisanship",
    y = "Net ideology",
    caption = "Source: Correlates of State Policy Project"
  ) +
  theme_minimal()
```



Partisanship and ideology track closely together, but several states sit noticeably above or below the line. The labels make those outliers immediately identifiable.

4.22 Formatting Numbers

The `scales` package provides label functions that change how numbers print on axes and legends. These functions do not change the underlying data. They only change the displayed labels.

`label_percent()` formats proportions as percentages. Since `pid` and `ideo` are proportions — the difference between Democratic and Republican shares of the population — displaying them as percentages is more natural than as decimals.

`label_dollar()` formats numbers as dollars. That is why GDP per capita can appear as `$1,000` or `$10,000` instead of plain numbers.

`label_comma()` adds comma separators to large numbers without shortening them. This is useful when the exact number should remain visible.

Large numbers often need compact labels. `label_number(scale_cut = cut_short_scale())` prints values using short suffixes such as `K`, `M`, and `B`. This is useful for population, budgets, counts, and other variables where full numbers would make the axis or legend hard to scan.

The `accuracy` argument controls rounding. For example, `accuracy = 0.1` keeps one decimal place, while `accuracy = 1` rounds to whole numbers.

```
example_numbers <- c(0.1234, 1200, 5400000, 1250000000)

label_percent(accuracy = 1)(example_numbers[1])
```

```
[1] "12%"
```

```
label_dollar()(example_numbers[2:3])
```

```
[1] "$1,200"     "$5,400,000"
```

```
label_comma()(example_numbers[2:4])
```

```
[1] "1,200"         "5,400,000"     "1,250,000,000"
```

```
label_number(scale_cut = cut_short_scale())(example_numbers[2:4])
```

```
[1] "1K" "5M" "1B"
```

```
label_number(accuracy = 0.1)(c(12.345, 67.891))
```

```
[1] "12.3" "67.9"
```

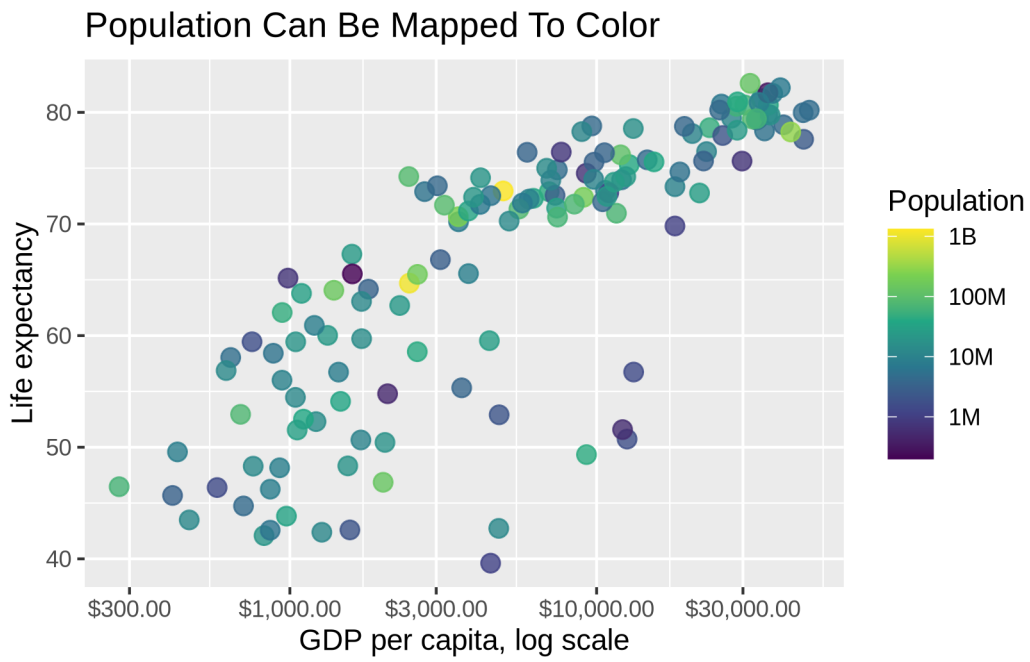
4.23 Continuous Aesthetics

Aesthetics can represent continuous variables as well as categories. A categorical aesthetic separates groups, such as continents. A continuous aesthetic represents a numeric variable, such as population or income.

4.23.1 Continuous Color

Color can represent a continuous variable. In the next plot, color represents population. Population is highly skewed: a few very large countries are much bigger than the rest. A log-transformed color scale spreads out the smaller and mid-sized countries so the colors are easier to distinguish. `scale_color_viridis_c()` uses a continuous `viridis` palette, and the `labels` argument applies the compact number labels introduced above.

```
p_gap +
  geom_point(aes(color = pop), size = 3, alpha = 0.8) +
  scale_x_log10(labels = label_dollar()) +
  scale_color_viridis_c(
    trans = "log10",
    labels = label_number(scale_cut = cut_short_scale())
  ) +
  labs(
    title = "Population Can Be Mapped To Color",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    color = "Population"
  )
```



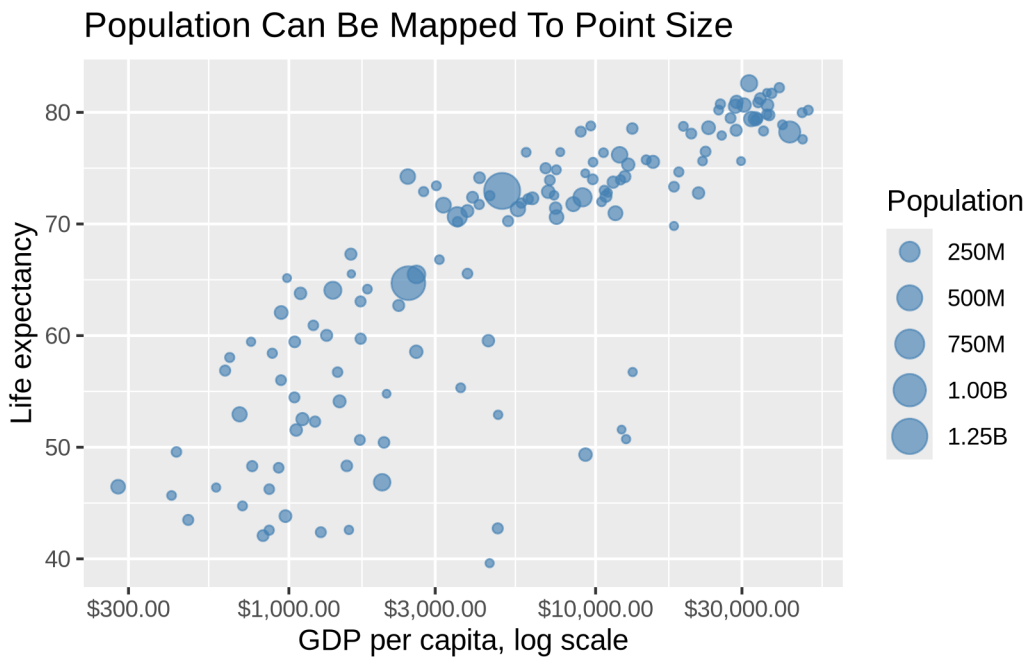
This is possible, but it is not always the best choice. Continuous color scales can be hard to

read precisely. They work best when the numeric variable adds useful context rather than simply making the plot look more complex.

4.23.2 Continuous Size

Size can also represent a continuous variable. In the next plot, population controls point size. `scale_size_continuous()` controls the size scale and the size legend.

```
p_gap +
  geom_point(aes(size = pop), color = "steelblue", alpha = 0.65) +
  scale_x_log10(labels = label_dollar()) +
  scale_size_continuous(labels = label_number(scale_cut = cut_short_scale())) +
  labs(
    title = "Population Can Be Mapped To Point Size",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    size = "Population"
  )
```

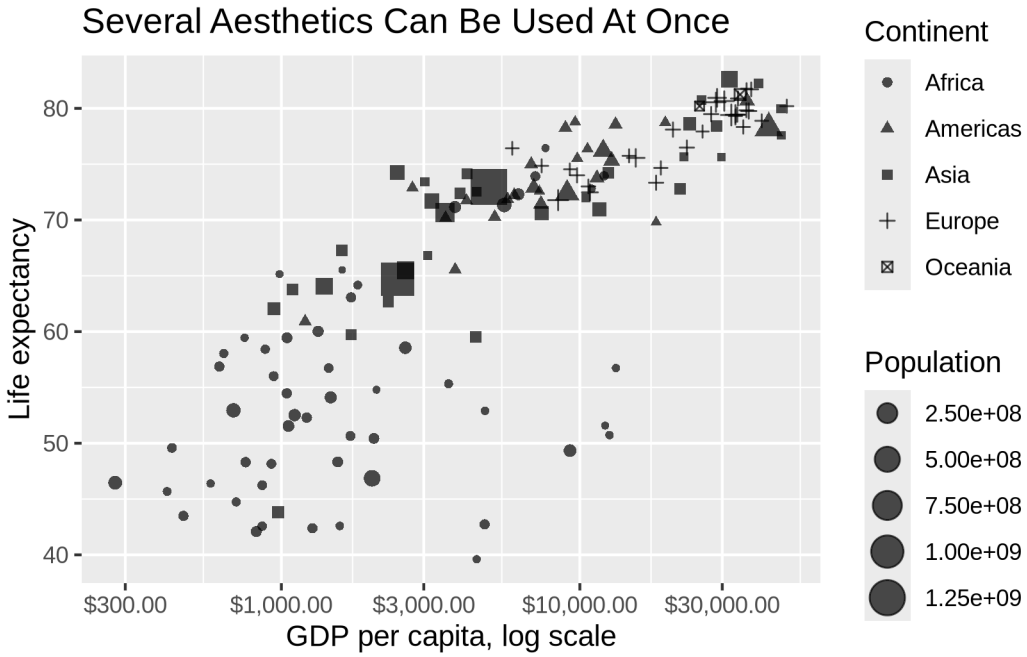


Size is often intuitive, but it is not very precise. Readers can usually see which points are much larger or much smaller, but they cannot easily compare exact values by area. That makes size useful for broad emphasis, not for precise measurement.

4.24 Extras: Shape And Size

Several aesthetics can be combined. Shape works best for a small number of categories. Size can work for quantities, but the plot gets harder to read as more encodings are added.

```
ggplot(gap_2007, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(shape = continent, size = pop), alpha = 0.7) +
  scale_x_log10(labels = label_dollar()) +
  labs(
    title = "Several Aesthetics Can Be Used At Once",
    x = "GDP per capita, log scale",
    y = "Life expectancy",
    shape = "Continent",
    size = "Population"
  )
```

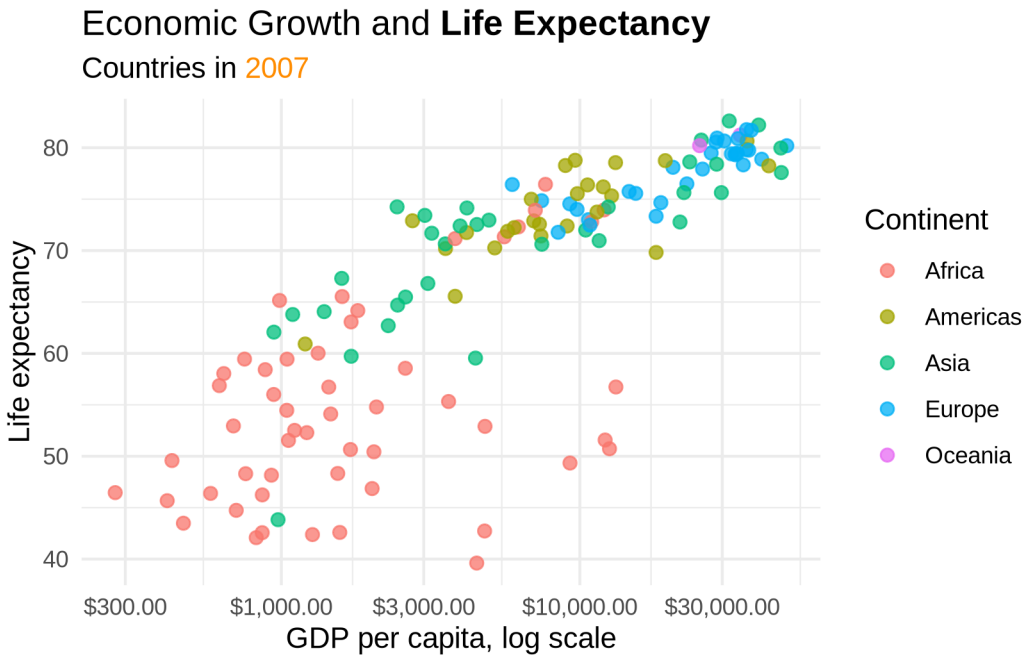


Adding more aesthetics is not automatically an improvement. Each additional encoding asks the reader to do more work. If it aids the goal of communication, then it's worth it.

4.25 Extras: Rich Text

`ggtext` allows limited Markdown and HTML-like formatting inside plot text. This is useful for emphasizing one phrase in a title or coloring part of a subtitle.

```
p_gap_color +
  labs(
    title = "Economic Growth and **Life Expectancy**",
    subtitle = "Countries in <span style='color:darkorange;'>2007</span>"
  ) +
  theme_minimal() +
  theme(
    plot.title = ggtext::element_markdown(),
    plot.subtitle = ggtext::element_markdown()
  )
```



4.26 What Comes Next

The next chapter moves from basic plot construction to visual comparison: separating groups with color, facets, small multiples, free scales, direct labels, and combining plots with `patchwork`. The focus shifts from making a plot work to making a comparison easier to see.

5 Separating and Comparing Data

5.1 Why separate data?

Many plots become more useful when we separate the data into meaningful groups. The relationship between two variables might look simple overall, but the pattern can differ across continents, years, regions, parties, income groups, or other subgroups.

There are two common `ggplot2` strategies:

- Map a grouping variable to an aesthetic such as `color`, `shape`, `fill`, or `linetype`.
- Use facets to create small multiples: repeated panels that show the same plot for different subsets of the data.

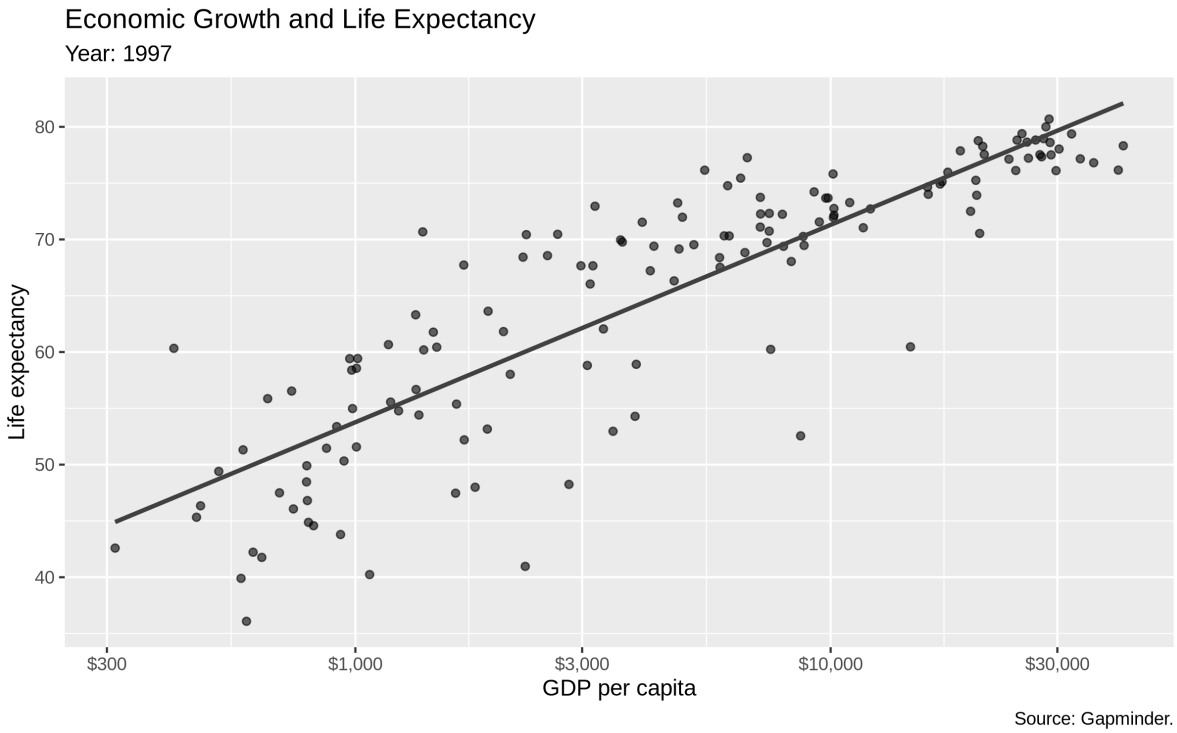
Both approaches are useful. The question is not which one is always better. The question is which one makes the comparison easier.

5.2 Start with one plot

We will use Gapminder data to compare GDP per capita and life expectancy. First, filter to one year so that each point is one country.

```
gap_1997 <- gapminder |>
  filter(year == 1997)

base_gap <- ggplot(
  data = gap_1997,
  mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "gray25") +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  scale_y_continuous(breaks = seq(20, 90, by = 10)) +
  labs(
    title = "Economic Growth and Life Expectancy",
    subtitle = "Year: 1997",
    x = "GDP per capita",
```

```
    y = "Life expectancy",
    caption = "Source: Gapminder."
  )

base_gap
```



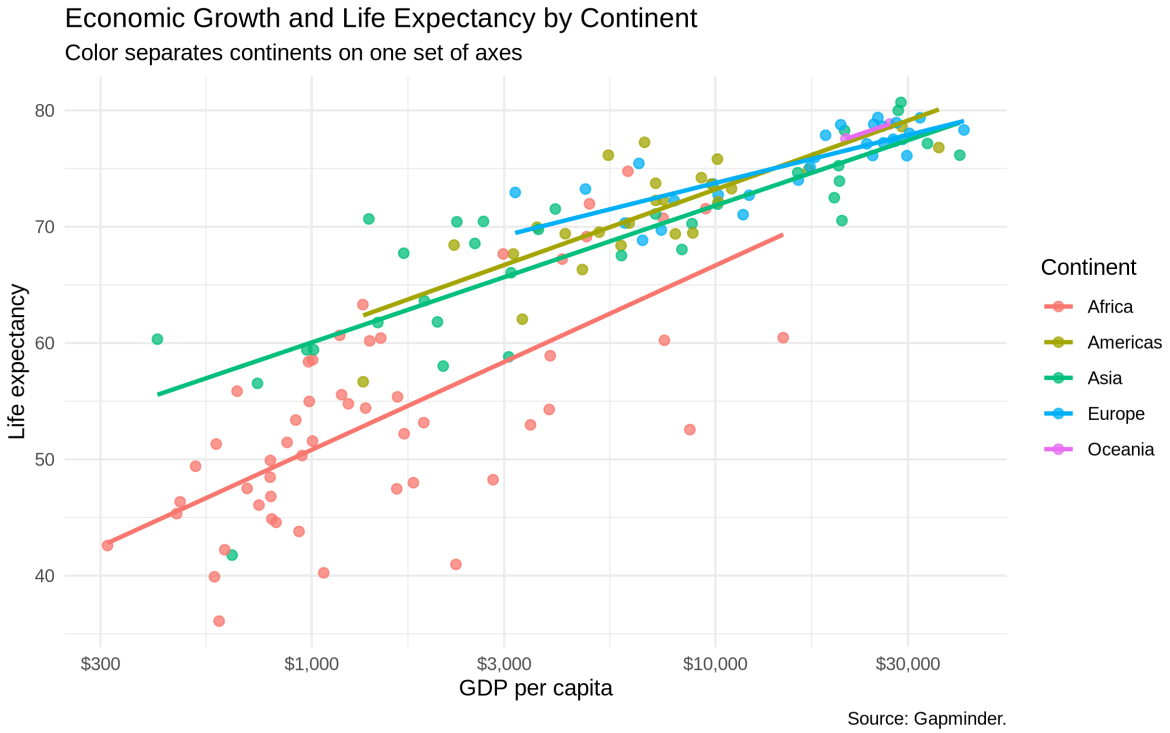
The log scale on the x-axis helps because GDP per capita is very skewed. Without it, the lower-income countries would be compressed into a small part of the plot.

5.3 Color as separation

One way to separate groups is to map a variable to color inside `aes()`.

```
gap_color <- ggplot(
  data = gap_1997,
  mapping = aes(x = gdpPercap, y = lifeExp, color = continent)
) +
  geom_point(alpha = 0.75, size = 2) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
```

```
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  labs(
    title = "Economic Growth and Life Expectancy by Continent",
    subtitle = "Color separates continents on one set of axes",
    x = "GDP per capita",
    y = "Life expectancy",
    color = "Continent",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()

gap_color
```

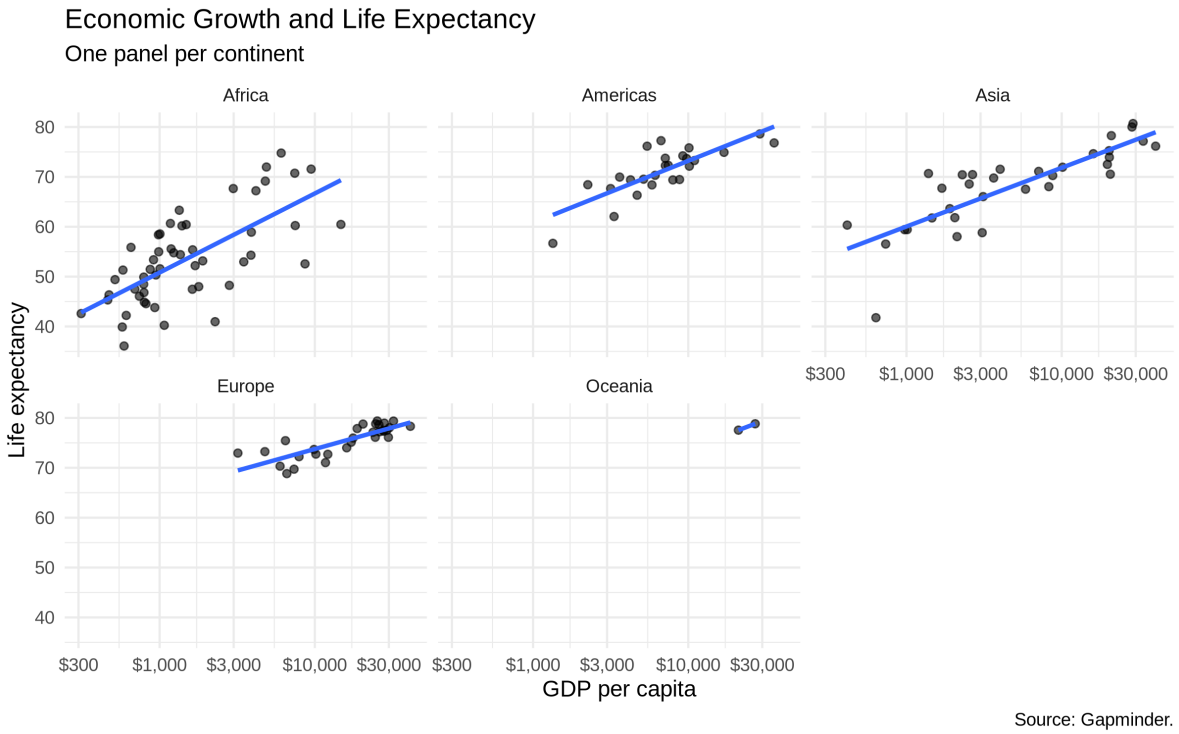


Because `continent` is mapped inside `aes()`, `ggplot2` treats color as part of the data mapping and creates a legend. If we wrote `geom_point(color = "steelblue")`, every point would be steel blue and no legend would be needed.

5.4 Facets as small multiples

Faceting creates one panel for each level of a variable. This gives each group its own space.

```r
ggplot(data = gap_1997, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  facet_wrap(~ continent) +
  labs(
    title = "Economic Growth and Life Expectancy",
    subtitle = "One panel per continent",
    x = "GDP per capita",
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```

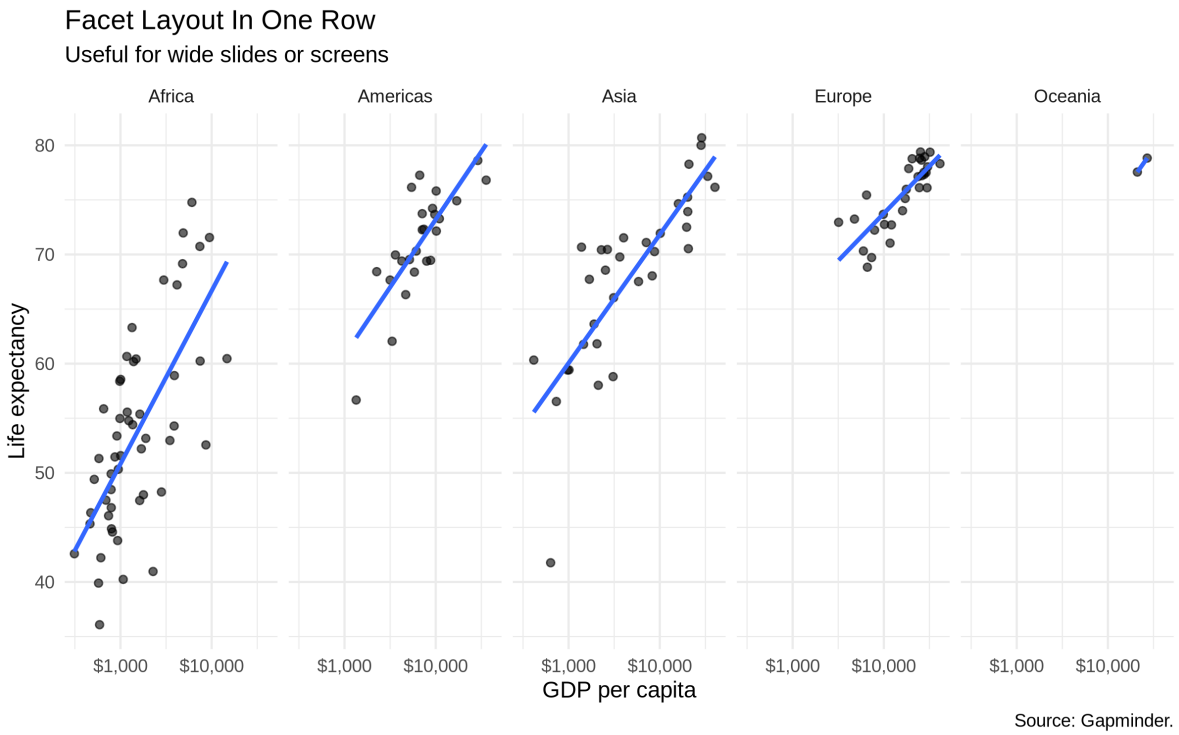


The default shared scales make comparisons across panels fair. Africa and Europe are plotted on the same x and y ranges, so differences in location and spread are visible.

5.5 Controlling Facet Layout

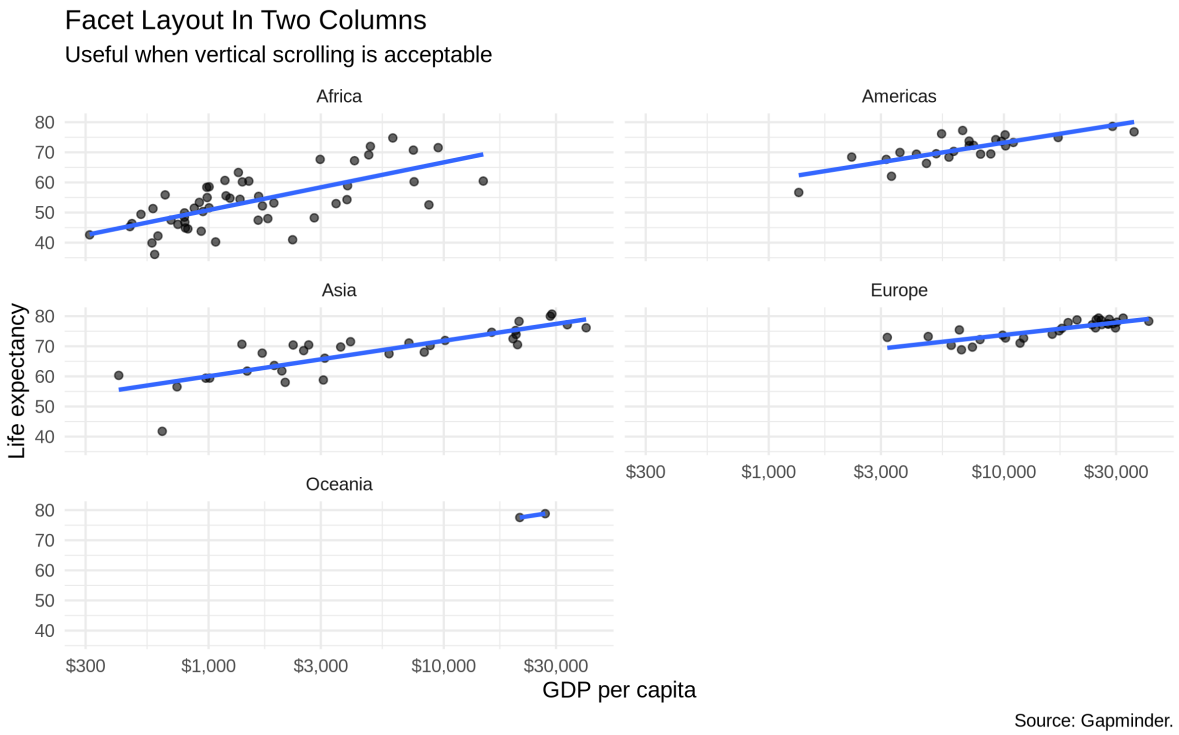
`facet_wrap()` chooses the panel layout automatically. The arguments `nrow` and `ncol` give direct control over the number of rows or columns. The data do not change; only the arrangement of panels changes.

```
ggplot(data = gap_1997, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  scale_x_log10(
    breaks = c(1000, 10000),
    labels = dollar_format(accuracy = 1)
  ) +
  facet_wrap(~ continent, nrow = 1) +
  labs(
    title = "Facet Layout In One Row",
    subtitle = "Useful for wide slides or screens",
    x = "GDP per capita",
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



The one-row layout leaves each panel narrow. Without fewer x-axis breaks, the dollar labels overlap. The `breaks = c(1000, 10000)` argument tells `ggplot2` to label only those two x-axis values, while the log scale and the data remain unchanged.

```
ggplot(data = gap_1997, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  facet_wrap(~ continent, ncol = 2) +
  labs(
    title = "Facet Layout In Two Columns",
    subtitle = "Useful when vertical scrolling is acceptable",
    x = "GDP per capita",
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



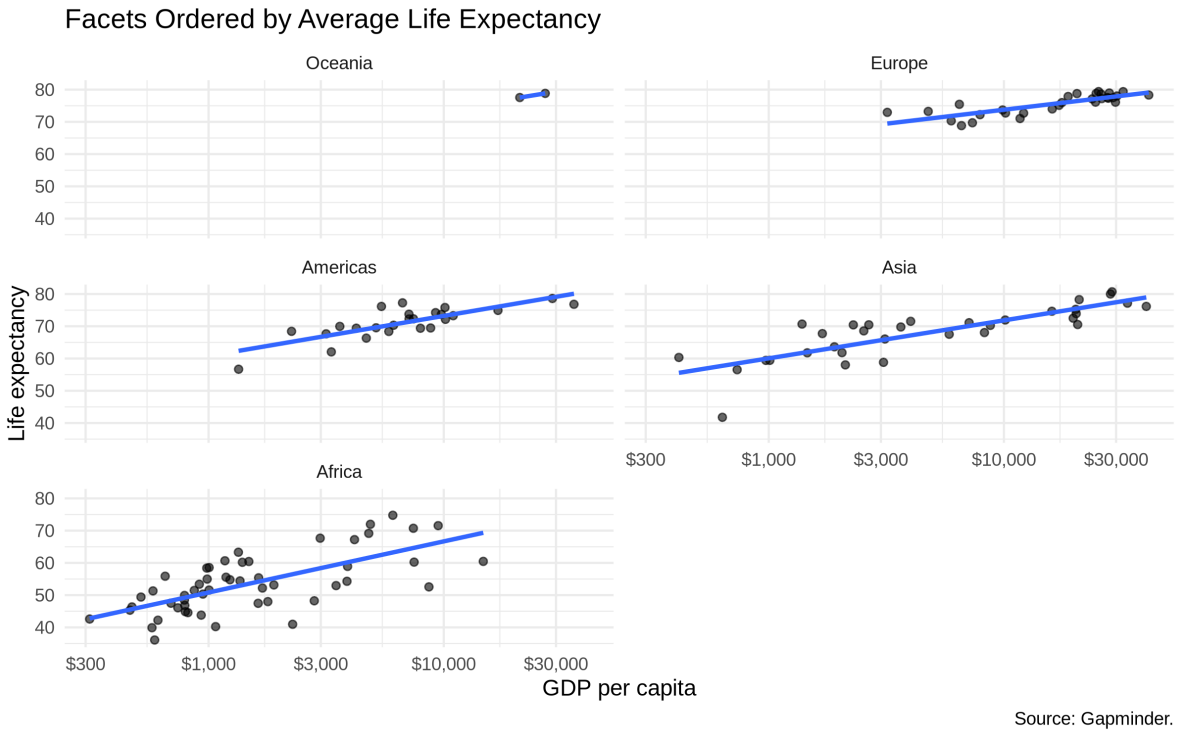
Use `nrow` or `ncol` when the default layout makes labels cramped or when the output format matters. A layout that works well in a wide slide may not work well in a narrow book column.

5.6 Reordering facets

Facet order is not always meaningful by default. We can reorder the continent variable by average life expectancy.

```
gap_1997_ordered <- gap_1997 |>
  mutate(
    continent = fct_reorder(
      .f = continent,
      .x = lifeExp,
      .fun = mean,
      .desc = TRUE
    )
  )

ggplot(data = gap_1997_ordered, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.6) +
```

```
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  facet_wrap(~ continent, ncol = 2) +
  labs(
    title = "Facets Ordered by Average Life Expectancy",
    x = "GDP per capita",
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```

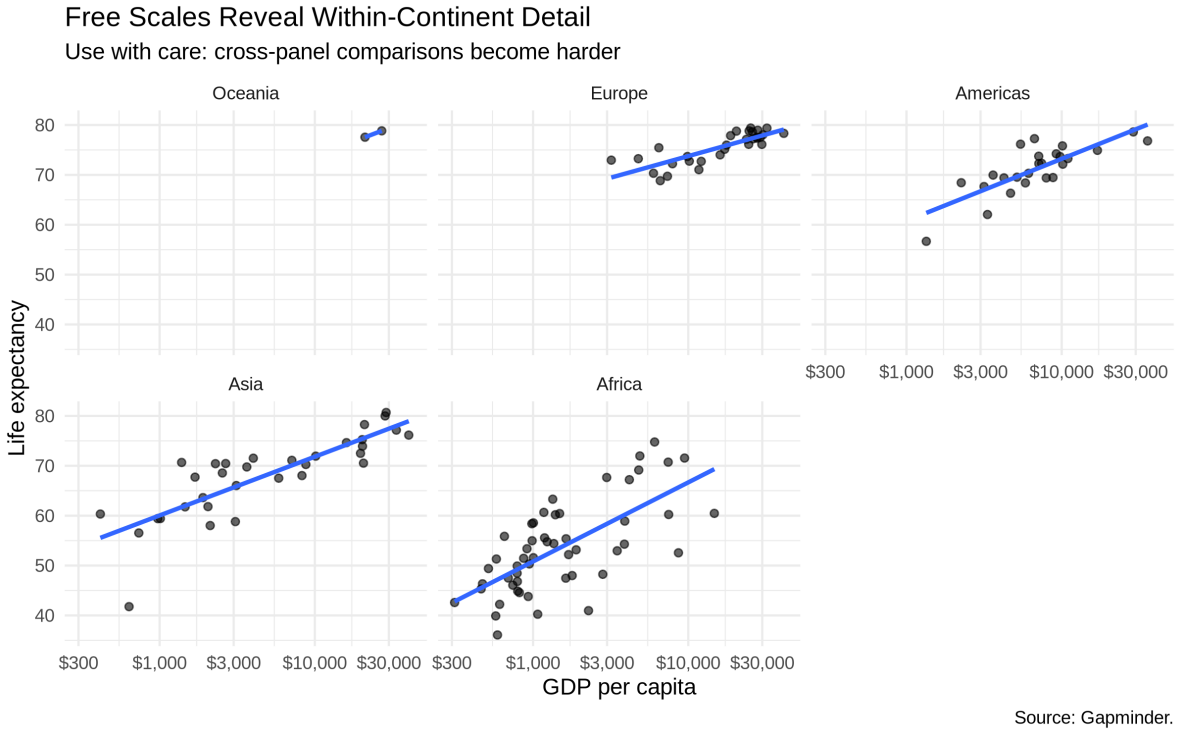


`fct_reorder()` changes the order of the factor levels. That order is then used by `facet_wrap()`.

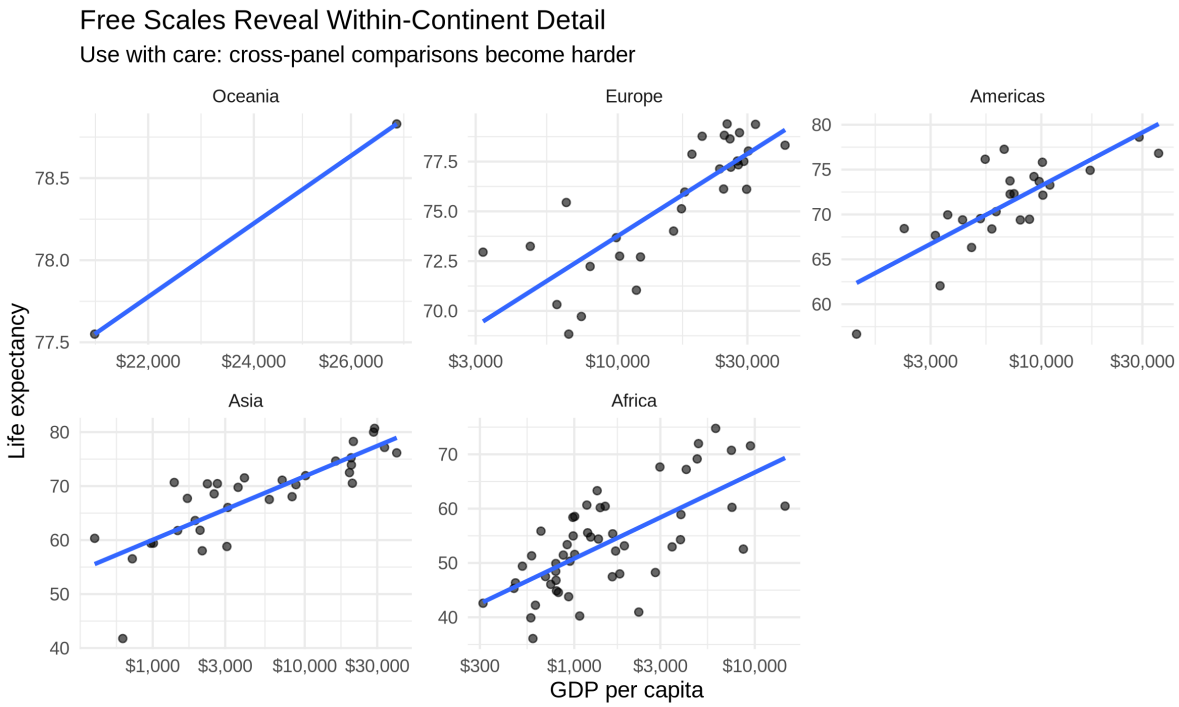
5.7 Free scales

Sometimes fixed scales make within-group patterns hard to see. Free scales let each panel use its own axis range.

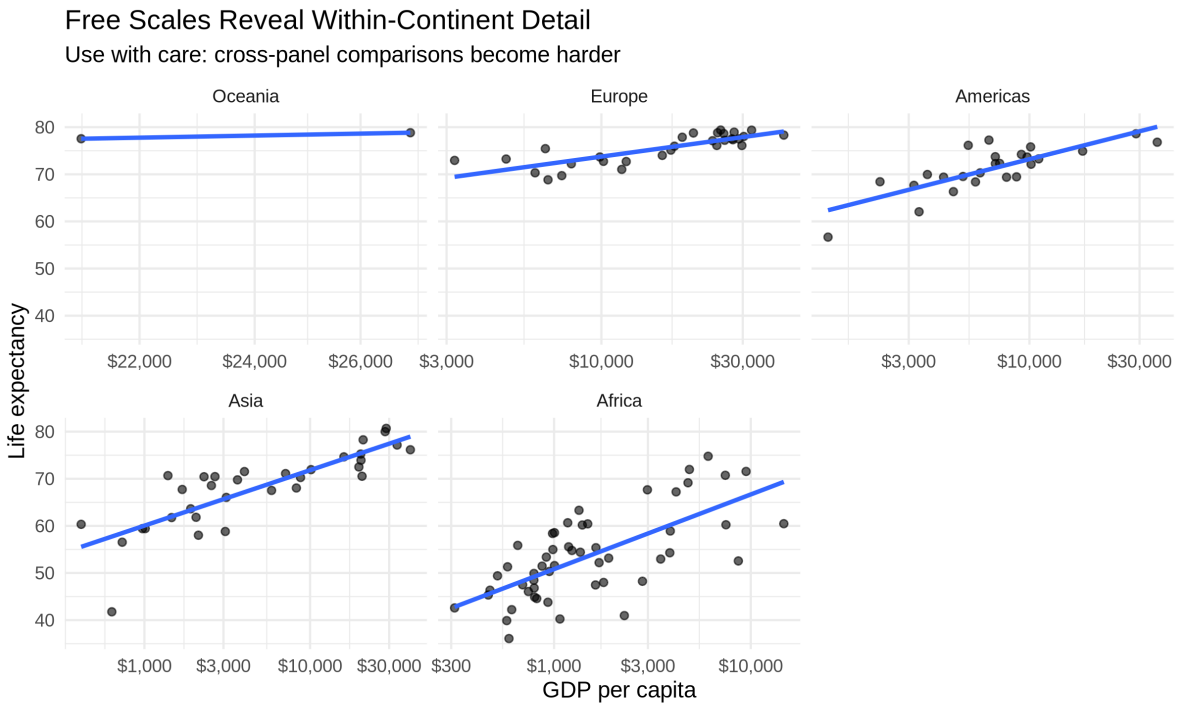
```
free97 <- ggplot(data = gap_1997_ordered, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  labs(
    title = "Free Scales Reveal Within-Continent Detail",
    subtitle = "Use with care: cross-panel comparisons become harder",
    x = "GDP per capita",
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()

# fixed scales
free97 +
  facet_wrap(~ continent)
```



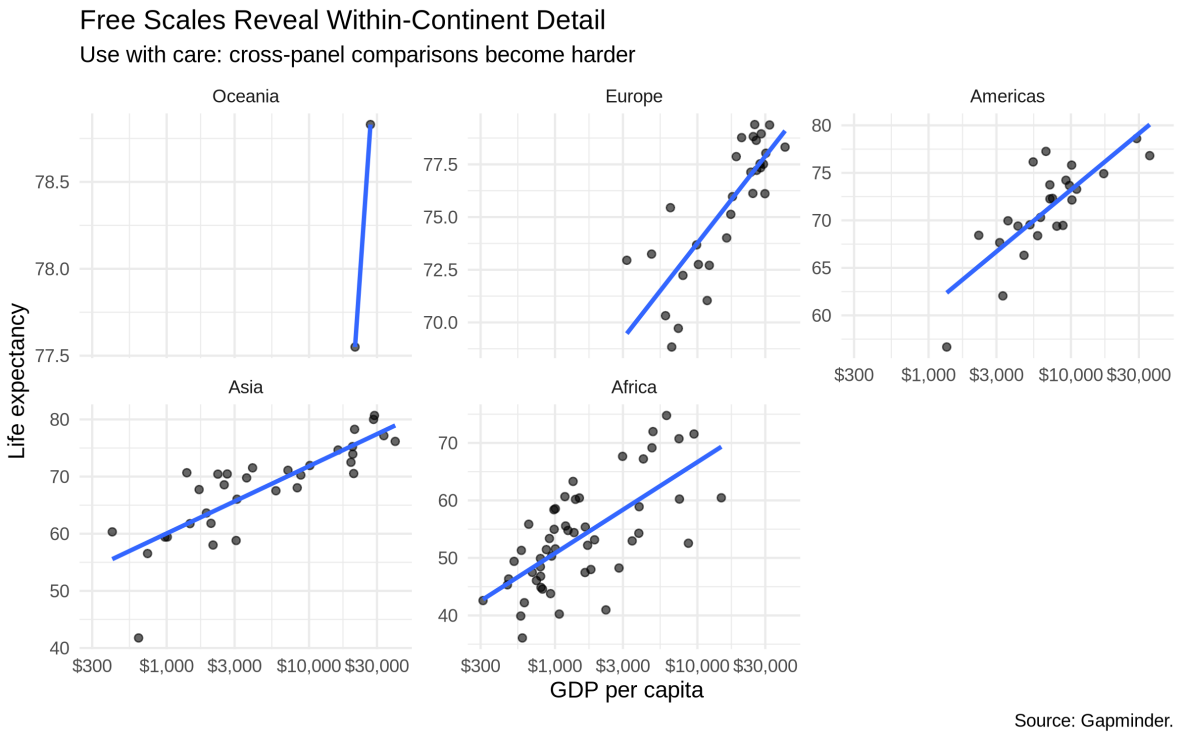

```
# free x and y scales
free97 +
  facet_wrap(~ continent, scales = "free")
```




```
# free x scale
free97 +
  facet_wrap(~ continent, scales = "free_x")
```




```
# free y scale
free97 +
  facet_wrap(~ continent, scales = "free_y")
```



Fixed scales emphasize comparison across groups. Free scales emphasize detail within each group.

5.8 Choosing Between Aesthetics And Faceting

There is no fixed rule here. Both approaches work, and the right choice usually reveals itself when you try one and find it unsatisfying.

A plot that maps groups to color keeps everything on shared axes, which can make comparisons direct and immediate. That works well when the groups are few enough that the colors stay distinct and the plot doesn't feel cluttered. When it starts to feel crowded — labels overlapping, colors hard to tell apart, too much happening in one panel — that is usually a sign to try faceting instead. Facets give each group its own space, and the consistent scales still allow comparison across panels.

You can also combine both: facet by one variable to create the panels, and use color within each panel to show a second grouping. That is what the state examples in this chapter do — facets separate years, region colors the text labels within each year.

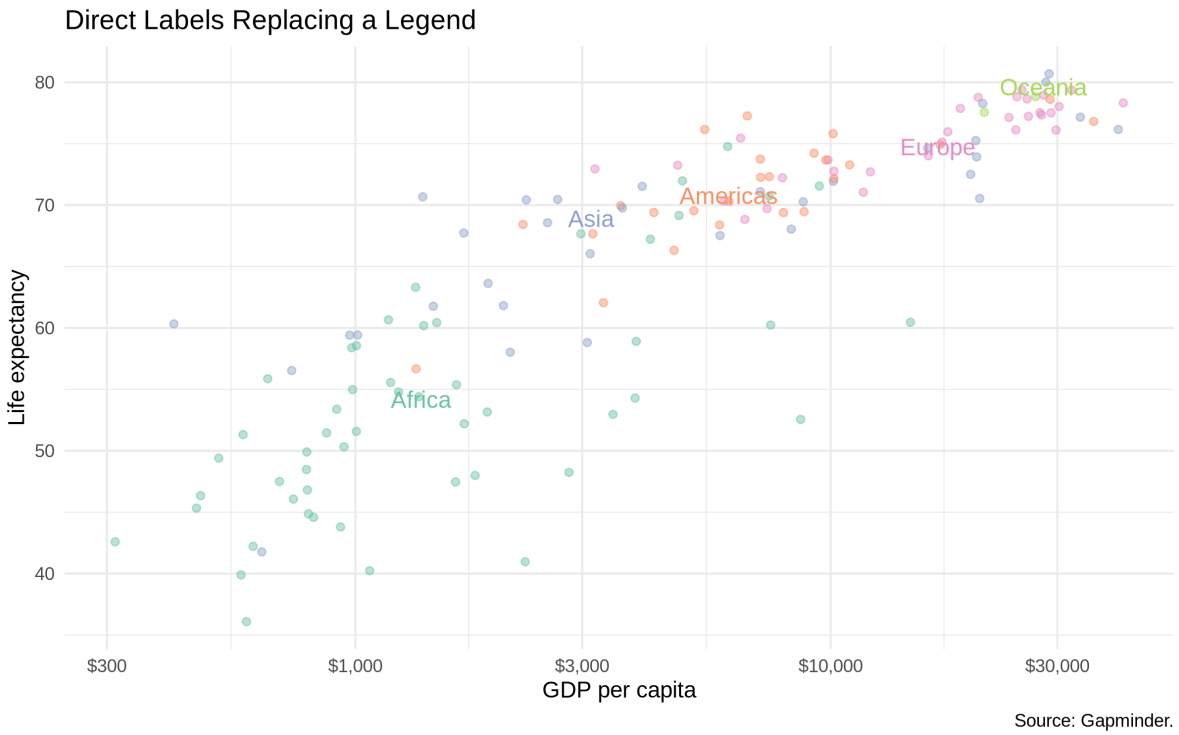
The practical test is simply whether the plot is easy to read. If the comparison you want to make is hard to see, try the other approach.

5.9 Direct labels

Legends require the reader to look back and forth between the data and the key. Direct labels can reduce that work.

The next plot uses `scale_color_brewer()` to apply a named `RColorBrewer` palette. The palette changes only the colors used for continents; it does not change the data, mapping, or labels. `Set2` is a qualitative palette, which means it is designed for unordered categories.

```
continent_summary <- gap_1997 |>
  group_by(continent) |>
  summarize(
    gdpPercap = median(gdpPercap),
    lifeExp = median(lifeExp)
  )

ggplot(data = gap_1997, mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.45, show.legend = FALSE) +
  geom_text_repel(
    data = continent_summary,
    mapping = aes(label = continent),
    size = 4,
    show.legend = FALSE
  ) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  scale_color_brewer(palette = "Set2") +
  labs(
    title = "Direct Labels Replacing a Legend",
    x = "GDP per capita",
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



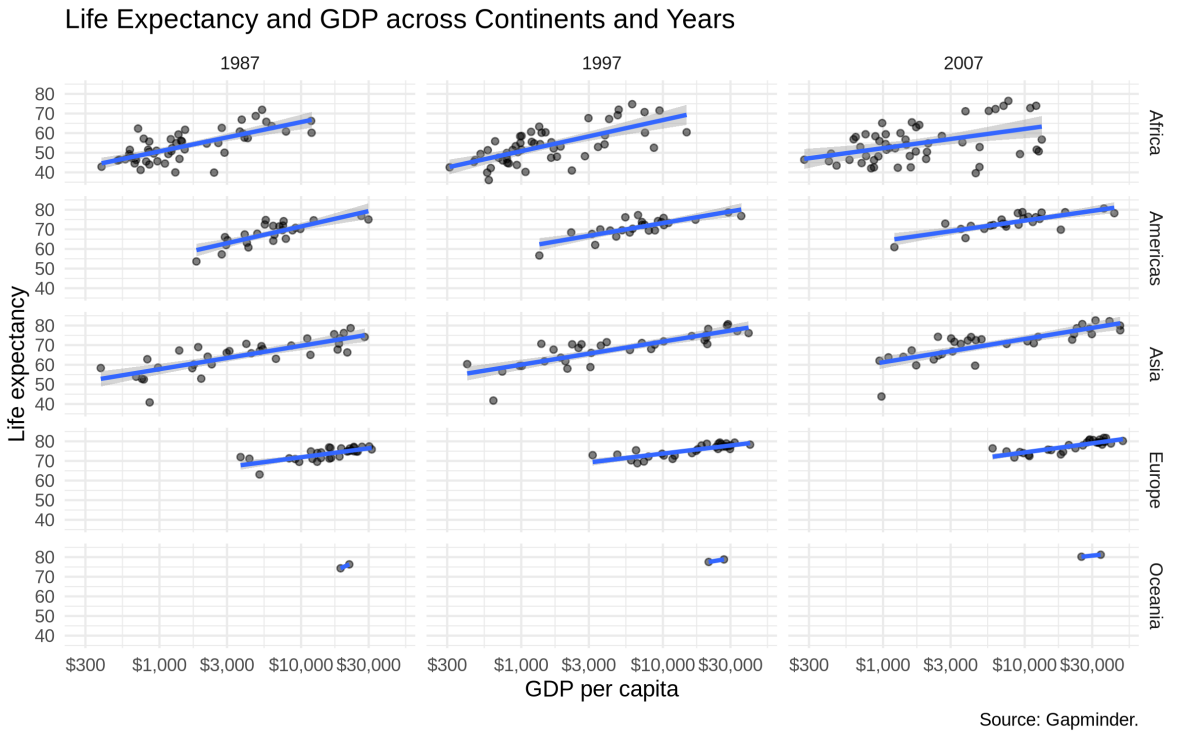
Direct labels work best when there are not too many groups and there is enough space for the labels. They are most useful when group identification matters and the legend forces too much back-and-forth reading.

5.10 `facet_grid()`

`facet_wrap()` is usually the easiest choice for one grouping variable. `facet_grid()` is useful when the panel layout has rows and columns with substantive meaning.

```
gap_recent <- gapminder |>
  filter(year %in% c(1987, 1997, 2007))

ggplot(data = gap_recent, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.5, size = 1.3) +
  geom_smooth(method = "lm") +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  facet_grid(continent ~ year) +
  labs(
    title = "Life Expectancy and GDP across Continents and Years",
    x = "GDP per capita",
```

```
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```

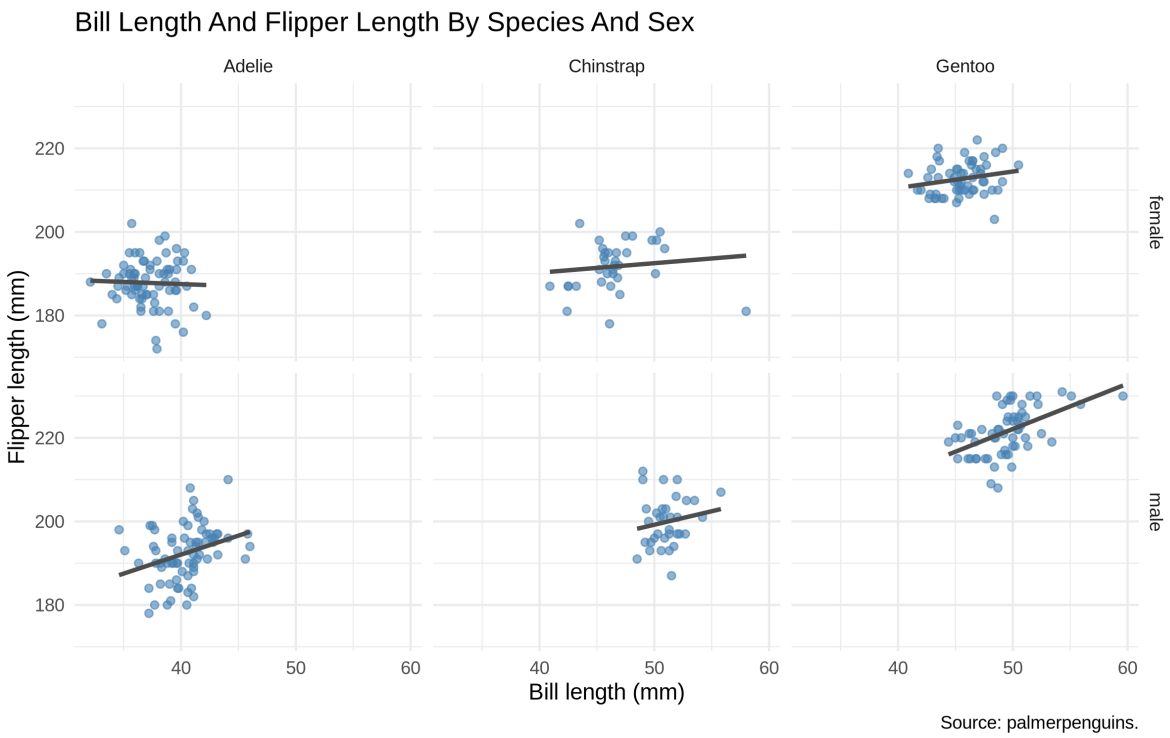


Read `facet_grid(continent ~ year)` as "continent defines the rows, year defines the columns."

`facet_grid()` is most compelling when both variables are genuinely categorical and unordered — not one categorical and one time dimension. Penguin data has two such variables: `species` (three unordered categories) and `sex` (two). A 3×2 grid of scatterplots lets the reader scan both dimensions at once.

```
penguins <- read_csv("Data/penguins/penguins.csv", show_col_types = FALSE) |>
  filter(!is.na(sex))

ggplot(penguins, aes(x = bill_length_mm, y = flipper_length_mm)) +
  geom_point(alpha = 0.6, color = "steelblue") +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "gray30") +
  facet_grid(sex ~ species) +
  labs(
```

```
    title = "Bill Length And Flipper Length By Species And Sex",
    x = "Bill length (mm)",
    y = "Flipper length (mm)",
    caption = "Source: palmerpenguins."
  ) +
  theme_minimal()
```



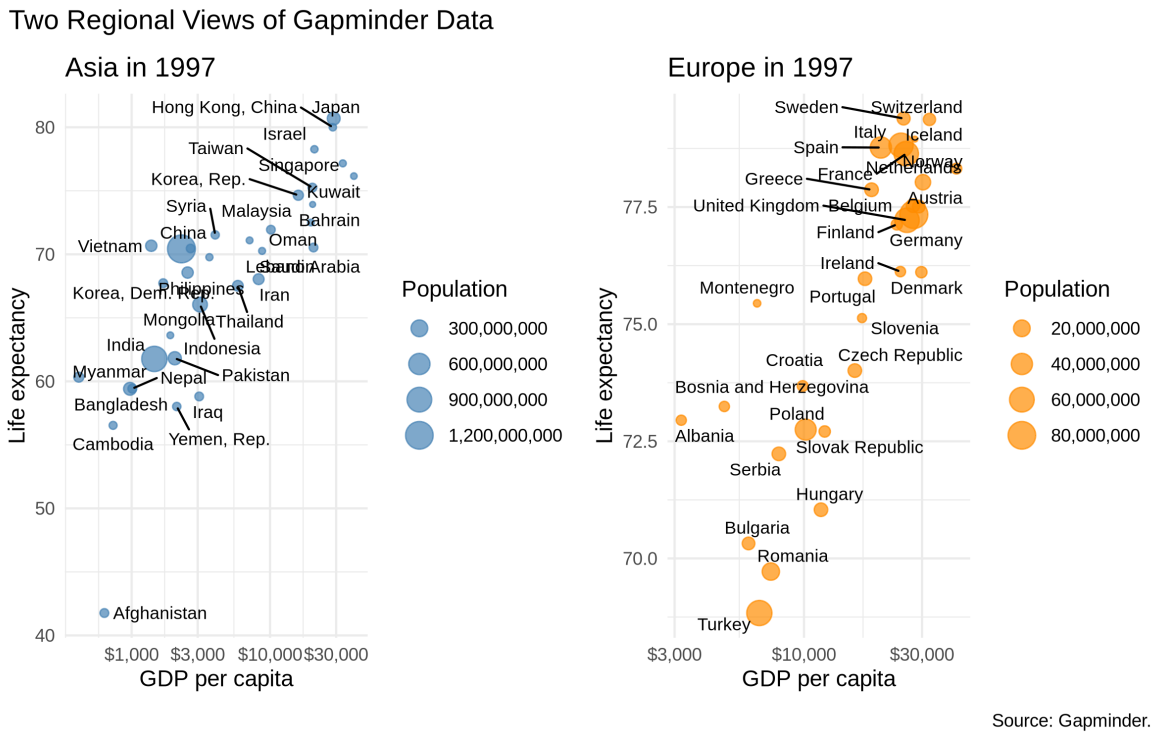
Each row is one sex, each column is one species. Scanning across a row shows species differences for a given sex. Scanning down a column shows sex differences within a species. Neither comparison requires the reader to look between panels that aren't aligned.

5.11 Patchwork

Facets repeat the same plot structure across groups. Sometimes you want to combine different plots into one figure. The `patchwork` package lets you assemble separate `ggplot` objects.

The next plots map population to point size with `aes(size = pop)`. As in the continuous-size example from Chapter 04, `scale_size_continuous()` adjusts the size scale and formats the size legend.

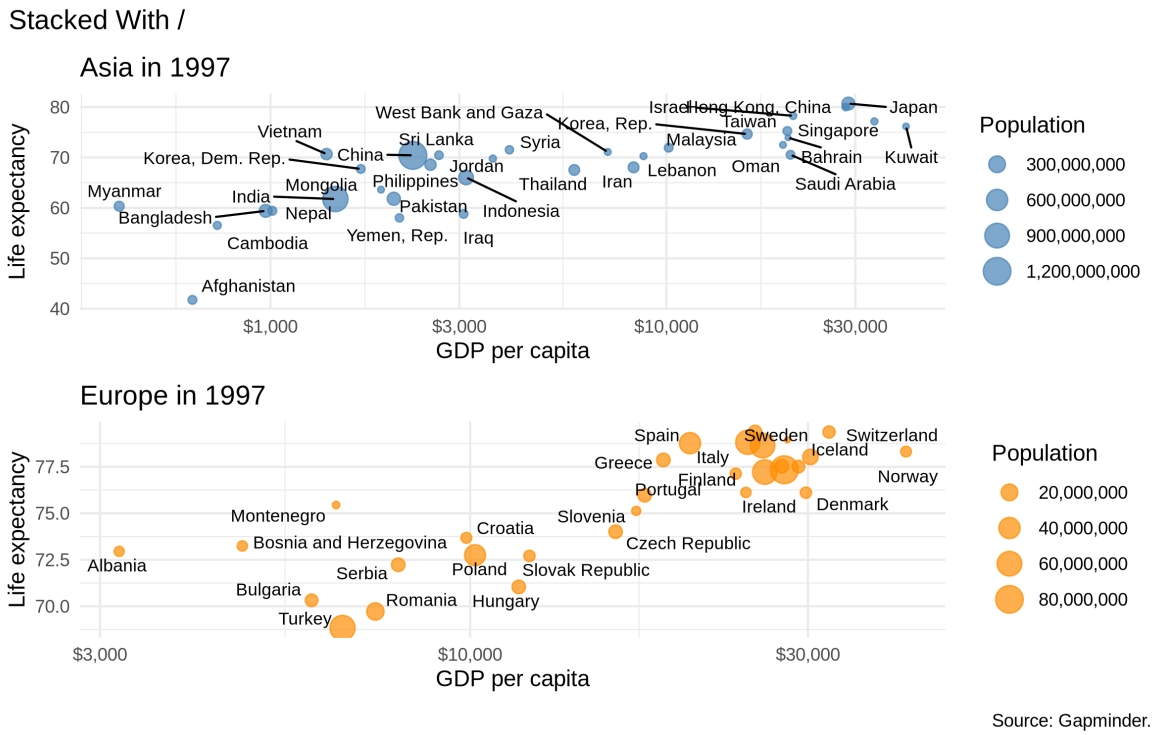
```r
asia_scatter <- gap_1997 |>
  filter(continent == "Asia") |>
  ggplot(mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(size = pop), color = "steelblue", alpha = 0.7) +
  geom_text_repel(aes(label = country), size = 3, max.overlaps = 12) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  scale_size_continuous(labels = comma) +
  labs(
    title = "Asia in 1997",
    x = "GDP per capita",
    y = "Life expectancy",
    size = "Population"
  ) +
  theme_minimal()

europe_scatter <- gap_1997 |>
  filter(continent == "Europe") |>
  ggplot(mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(size = pop), color = "darkorange", alpha = 0.7) +
  geom_text_repel(aes(label = country), size = 3, max.overlaps = 12) +
  scale_x_log10(labels = dollar_format(accuracy = 1)) +
  scale_size_continuous(labels = comma) +
  labs(
    title = "Europe in 1997",
    x = "GDP per capita",
    y = "Life expectancy",
    size = "Population"
  ) +
  theme_minimal()

asia_scatter + europe_scatter +
  plot_layout(widths = c(1, 1)) +
  plot_annotation(
    title = "Two Regional Views of Gapminder Data",
    caption = "Source: Gapminder."
  )
```



`patchwork` uses three operators:

- `+` or `|` places plots side by side
- `/` stacks plots vertically

```
asia_scatter / europe_scatter +
  plot_annotation(
    title = "Stacked With /",
    caption = "Source: Gapminder."
  )
```



5.12 Extra: State Policy Snapshot

State-level political and policy. This section uses a small local file from the course `Data/` folder.

The inspection function `glimpse()` comes from `dplyr`, which is loaded as part of the tidyverse.

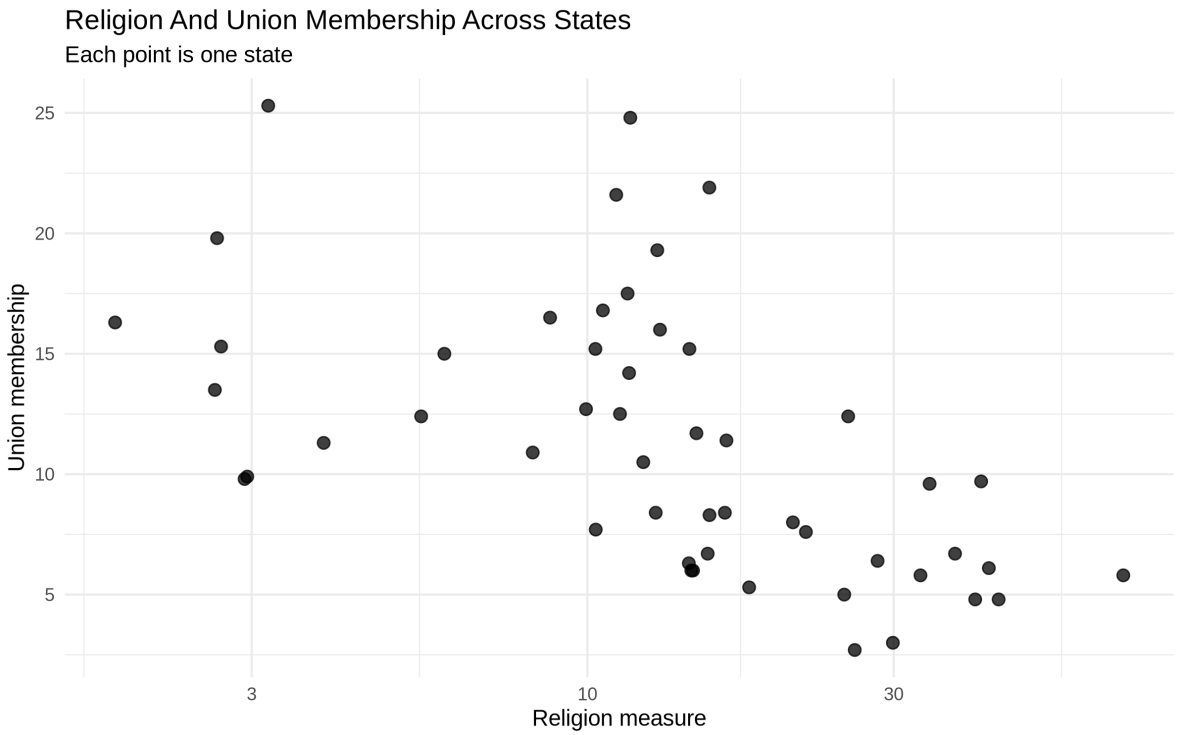
```
state_policy <- read_csv(
  "Data/state_policy/state_data_sorted.csv",
  show_col_types = FALSE
) |>
  mutate(
    union_pct = 100 * union,
    religion_pct = 100 * religion,
    region = str_to_title(cregion)
  )

glimpse(state_policy)
```

```
Rows: 51
```

```
Columns: 11
$ abb          <chr> "AK", "AL", "AR", "AZ", "CA", "CO", "CT", "DC", "DE", "FL~
$ union        <dbl> 0.219, 0.097, 0.048, 0.063, 0.165, 0.084, 0.153, 0.127, 0~
$ religion     <dbl> 0.1548637, 0.4104603, 0.4370625, 0.1438850, 0.0874711, 0.~
$ cst          <chr> "AK", "AL", "AR", "AZ", "CA", "CO", "CT", "DC", "DE", "FL~
$ cregion      <chr> "west", "south", "south", "west", "west", "west", "northe~
$ region_num   <dbl> 4, 3, 3, 4, 4, 4, 2, 2, 3, 3, 3, 4, 1, 4, 1, 1, 1, 3, 3, ~
$ state_num    <dbl> 50, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 51, 11, 12, 13, 14, 15~
$ pop          <dbl> 626932, 4447100, 2673400, 5130632, 33871648, 4301261, 340~
$ union_pct    <dbl> 21.9, 9.7, 4.8, 6.3, 16.5, 8.4, 15.3, 12.7, 12.4, 6.0, 6.~
$ religion_pct <dbl> 15.48637, 41.04603, 43.70625, 14.38850, 8.74711, 12.77672~
$ region       <chr> "West", "South", "South", "West", "West", "West", "Northe~
```

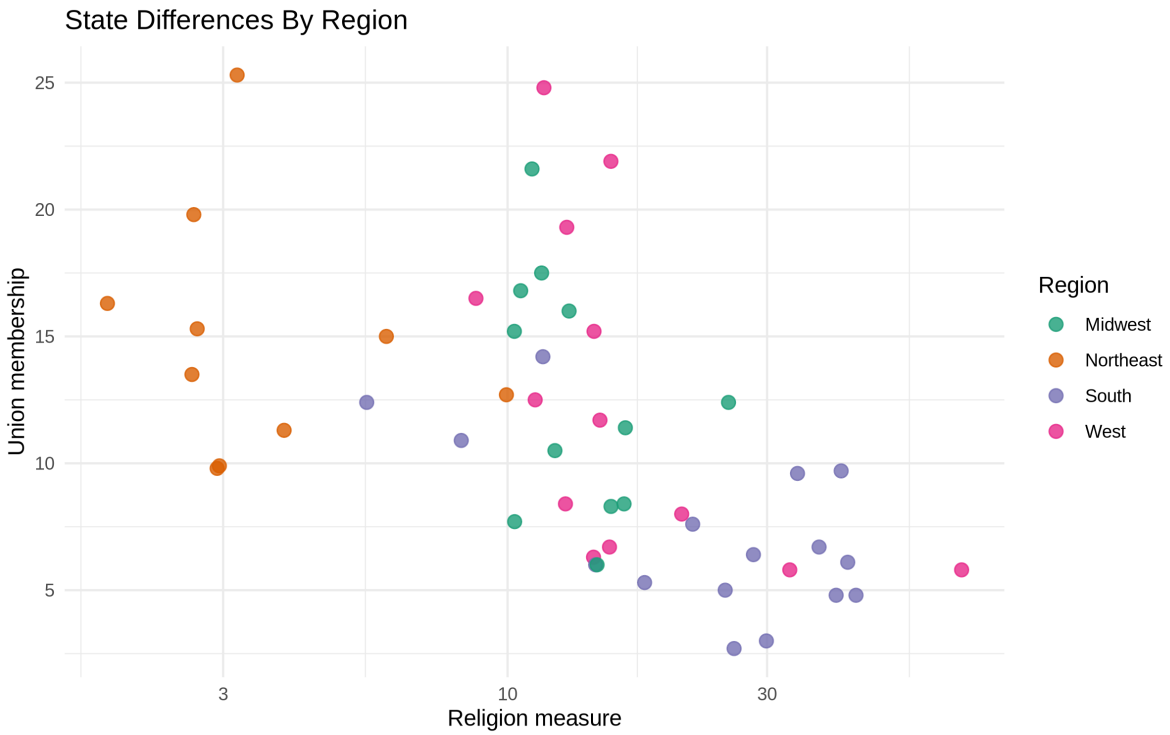
The two variables in the next plot are percentages.

```
ggplot(
  state_policy,
  aes(x = religion_pct, y = union_pct)
) +
  geom_point(size = 2.5, alpha = 0.75) +
  scale_x_log10() +
  labs(
    title = "Religion And Union Membership Across States",
    subtitle = "Each point is one state",
    x = "Religion measure",
    y = "Union membership"
  ) +
  theme_minimal()
```



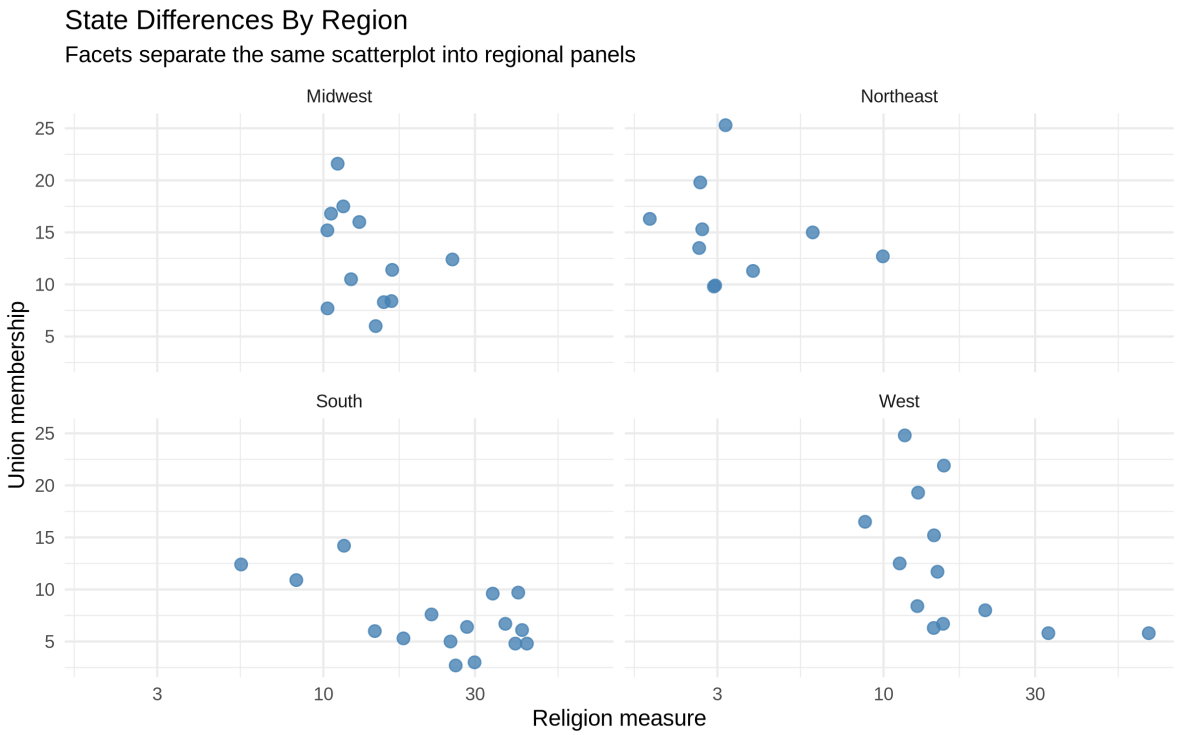
Color can separate regions without creating multiple panels. This is useful when the plot is still readable with all observations in one place.

```
ggplot(
  state_policy,
  aes(x = religion_pct, y = union_pct, color = region)
) +
  geom_point(size = 2.8, alpha = 0.8) +
  scale_x_log10() +
  scale_color_brewer(palette = "Dark2") +
  labs(
    title = "State Differences By Region",
    x = "Religion measure",
    y = "Union membership",
    color = "Region"
  ) +
  theme_minimal()
```



Facets split the same data into small multiples. This makes regional comparisons clearer when the color version starts to feel crowded.

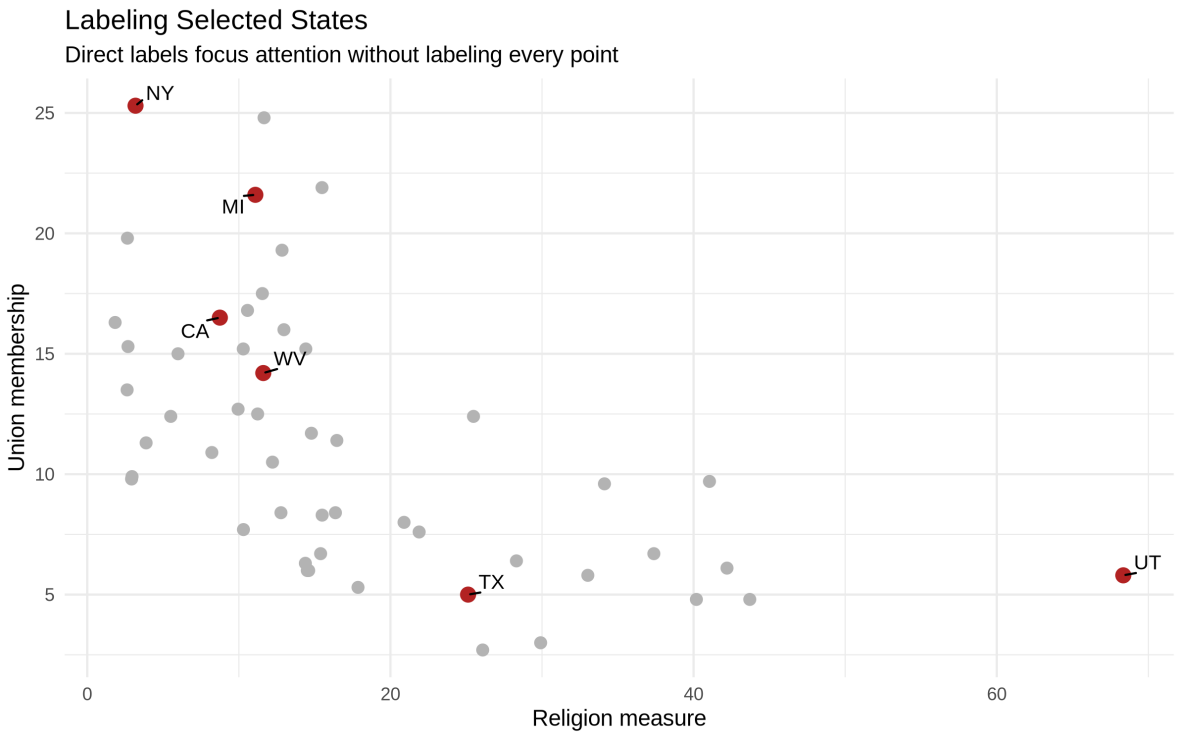
```
ggplot(
  state_policy,
  aes(x = religion_pct, y = union_pct)
) +
  geom_point(color = "steelblue", size = 2.5, alpha = 0.8) +
  scale_x_log10() +
  facet_wrap(~ region) +
  labs(
    title = "State Differences By Region",
    subtitle = "Facets separate the same scatterplot into regional panels",
    x = "Religion measure",
    y = "Union membership"
  ) +
  theme_minimal()
```



Direct labels are useful when a few cases matter. Labeling every state would crowd the figure, so the next example labels only selected states.

```
label_states <- c("CA", "MI", "NY", "TX", "UT", "WV")

ggplot(
  state_policy,
  aes(x = religion_pct, y = union_pct)
) +
  geom_point(color = "gray70", size = 2.3) +
  geom_point(
    data = filter(state_policy, abb %in% label_states),
    color = "firebrick",
    size = 3
  ) +
  geom_text_repel(
    data = filter(state_policy, abb %in% label_states),
    aes(label = abb),
    size = 3.5,
    min.segment.length = 0
  ) +
```

```
  labs(
    title = "Labeling Selected States",
    subtitle = "Direct labels focus attention without labeling every point",
    x = "Religion measure",
    y = "Union membership"
  ) +
  theme_minimal()
```



5.13 Extra: PISA international scores

The PISA dataset contains results from the 2006 OECD Programme for International Student Assessment. Each of the roughly 152,000 rows is one student. Variables include country and average scores in math, reading, and science.

```
pisa <- read_csv("Data/pisa/pisa_data_clean.csv", show_col_types = FALSE) |>
  janitor::clean_names()

pisa |>
```

```
  select(country_code, average_math_score, average_reading_score, average_science_score) |>
  slice_head(n = 6)
```

```
# A tibble: 6 x 4
  country_code average_math_score average_reading_score average_science_score
  <chr>                     <dbl>                 <dbl>                 <dbl>
1 Albania                    490.                  372.                  491.
2 Albania                    434.                  402.                  342.
3 Albania                    439.                  551.                  460.
4 Albania                    417.                  350.                  360.
5 Albania                    453.                  458.                  452.
6 Albania                    385.                  398.                  400.
```

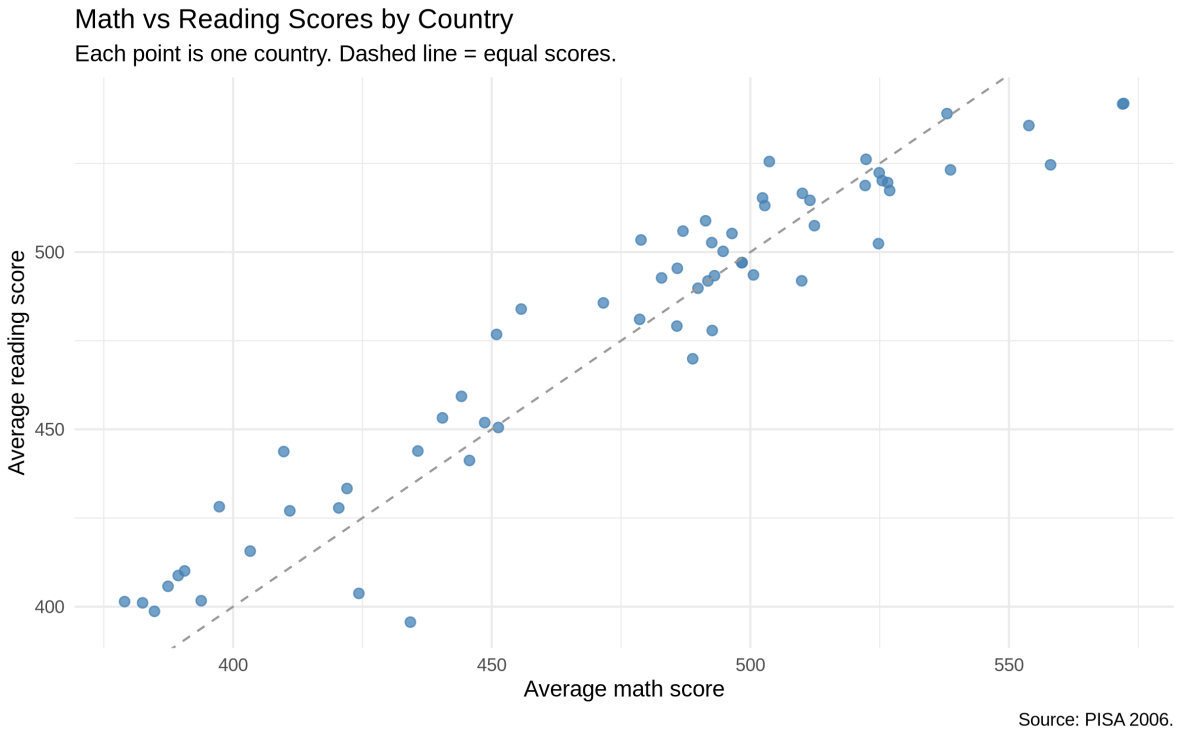
Summarize to country means to reduce 152k rows to a manageable comparison dataset.

```
pisa_country <- pisa |>
  group_by(country_code) |>
  summarize(
    math    = mean(average_math_score,    na.rm = TRUE),
    reading = mean(average_reading_score, na.rm = TRUE),
    science = mean(average_science_score, na.rm = TRUE)
  )

pisa_country |> slice_head(n = 6)
```

```
# A tibble: 6 x 4
  country_code  math reading science
  <chr>        <dbl>   <dbl>   <dbl>
1 Albania       394.    402.    402.
2 Argentina     403.    416.    419.
3 Australia     496.    505.    513.
4 Austria       510.    492.    511.
5 Belgium       527.    520.    517.
6 Brazil        389.    409.    405.
```

A scatterplot of math against reading scores shows how the two subjects correlate across countries. The dashed line marks equal scores in the two subjects.

```r
ggplot(pisa_country, aes(x = math, y = reading)) +
  geom_point(color = "steelblue", alpha = 0.75, size = 2) +
  geom_abline(linetype = "dashed", color = "gray60") +
  labs(
    title = "Math vs Reading Scores by Country",
    subtitle = "Each point is one country. Dashed line = equal scores.",
    x = "Average math score",
    y = "Average reading score",
    caption = "Source: PISA 2006."
  ) +
  theme_minimal()
```

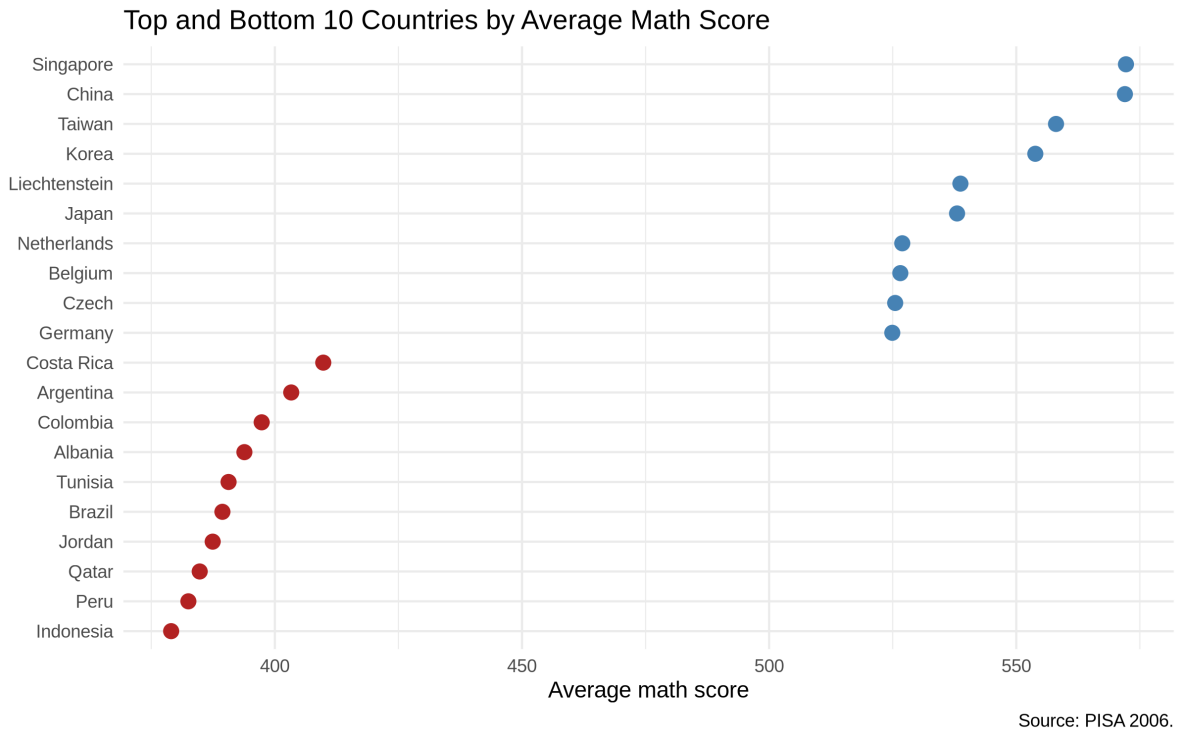


Most countries fall close to the dashed line, which means national math and reading averages tend to move together. Countries farther from the line are cases where one subject is notably stronger than the other.

We can also rank countries by math score and look at the top and bottom performers.

Here `scale_color_manual()` assigns exact colors to the two summary groups. Manual scales are useful when colors should carry consistent meaning across plots, such as top versus bottom groups or before versus after periods.

```r
pisa_overall <- pisa |>
  group_by(country_code) |>
  summarize(
    math    = mean(average_math_score,    na.rm = TRUE),
    reading = mean(average_reading_score, na.rm = TRUE)
  )

top_bottom <- pisa_overall |>
  slice_max(math, n = 10) |>
  bind_rows(pisa_overall |> slice_min(math, n = 10)) |>
  mutate(group = if_else(math >= median(pisa_overall$math), "Top 10", "Bottom 10"))

ggplot(top_bottom, aes(x = math, y = reorder(country_code, math))) +
  geom_point(aes(color = group), size = 3, show.legend = FALSE) +
  scale_color_manual(values = c("Top 10" = "steelblue", "Bottom 10" = "firebrick")) +
  labs(
    title = "Top and Bottom 10 Countries by Average Math Score",
    x = "Average math score",
    y = NULL,
    caption = "Source: PISA 2006."
  ) +
  theme_minimal()
```



5.14 Extra: State Ideology And Partisanship

In chapter 4, `geom_text_repel()` labeled states in a single year. The separation techniques in this chapter give those labels more room and more meaning — faceting by year makes it possible to compare two cross-sections of the same states without the labels colliding.

The Correlates of State Policy data has two measures that capture different aspects of state politics:

- `pid`: net partisanship — Democratic minus Republican party identification
- `pollib_median`: policy liberalism — a composite index of policy outputs; higher values indicate more liberal policy on issues like minimum wage, abortion, gun control, and similar measures

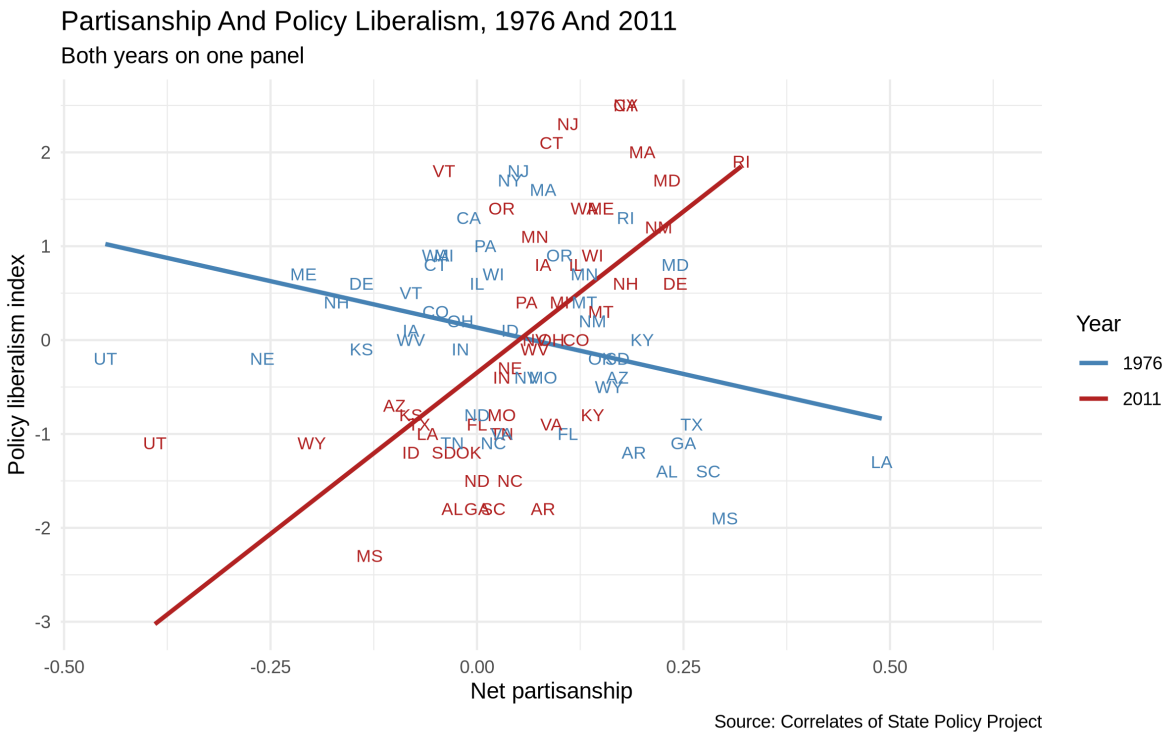
Loading the data:

```
states <- read_csv("Data/state_policy/cspp_states.csv", show_col_types = FALSE)

states_compare <- states |>
  filter(year %in% c(1976, 2011))
```

On a single panel with both years colored, we can see the overall shape of the relationship but individual states are hard to read.

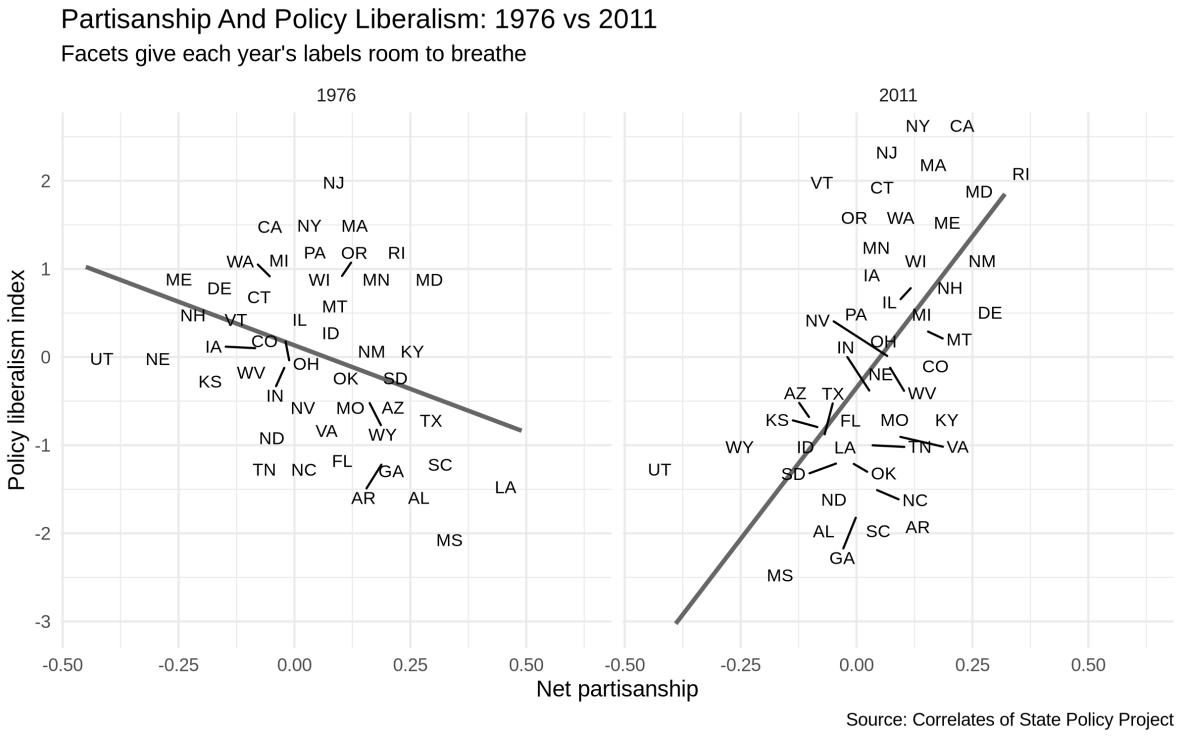
The next plot also uses a manual color scale so the two years have stable, named colors.

```
ggplot(states_compare, aes(x = pid, y = pollib_median, color = as.factor(year))) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  geom_text(aes(label = st), size = 3, show.legend = FALSE) +
  scale_color_manual(values = c("1976" = "steelblue", "2011" = "firebrick")) +
  labs(
    title = "Partisanship And Policy Liberalism, 1976 And 2011",
    subtitle = "Both years on one panel",
    x = "Net partisanship",
    y = "Policy liberalism index",
    color = "Year",
    caption = "Source: Correlates of State Policy Project"
  ) +
  theme_minimal()
```



The two regression lines suggest the relationship has shifted, but with both years overlaid the state labels are crowded and hard to read. Faceting by year gives each panel room.

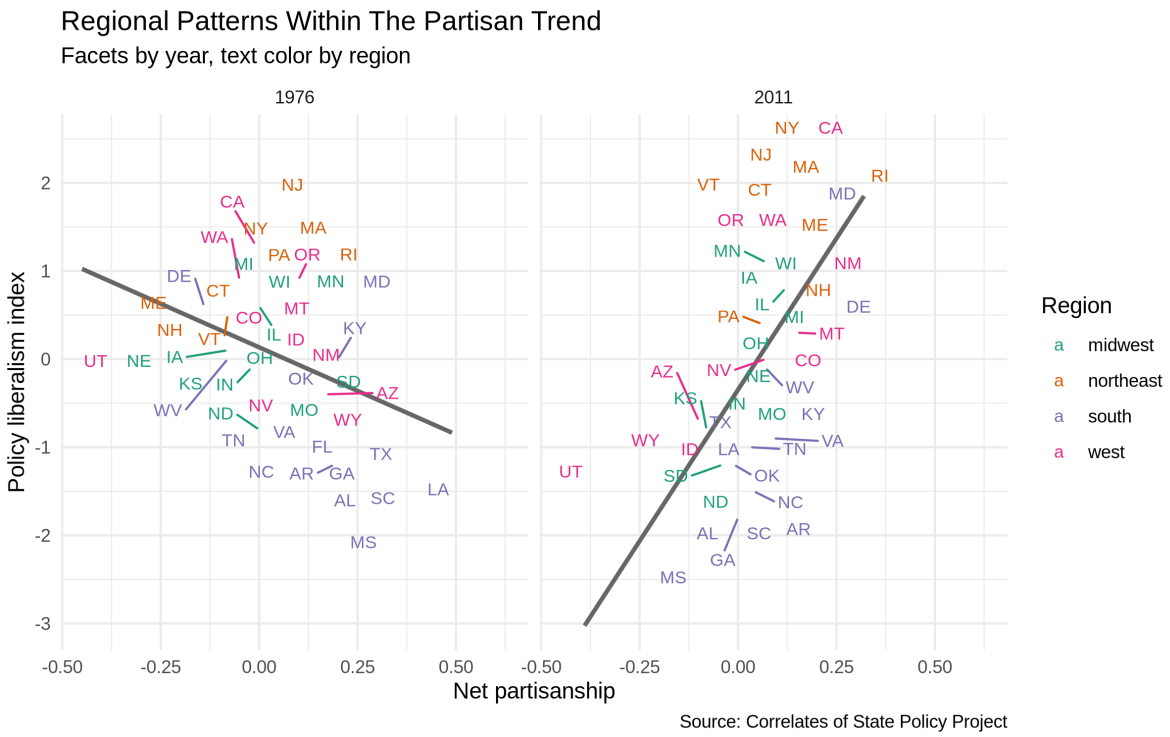
```
ggplot(states_compare, aes(x = pid, y = pollib_median)) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "gray40") +
  geom_text_repel(aes(label = st), size = 3) +
  facet_wrap(~ year) +
  labs(
    title = "Partisanship And Policy Liberalism: 1976 vs 2011",
    subtitle = "Facets give each year's labels room to breathe",
    x = "Net partisanship",
    y = "Policy liberalism index",
    caption = "Source: Correlates of State Policy Project"
  ) +
  theme_minimal()
```



The faceted version makes the comparison readable. The slope in 2011 is steeper: Democratic-leaning states had adopted more liberal policies relative to Republican-leaning states than was the case in 1976. The labels make it possible to identify which specific states drive that pattern.

Coloring the text by region adds another layer of separation without adding more panels. It shows whether states that sit above or below the trend line in a given year share a regional pattern.

```
ggplot(states_compare, aes(x = pid, y = pollib_median)) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "gray40") +
  geom_text_repel(aes(label = st, color = region.name), size = 3) +
  scale_color_brewer(palette = "Dark2") +
  facet_wrap(~ year) +
  labs(
    title = "Regional Patterns Within The Partisan Trend",
    subtitle = "Facets by year, text color by region",
    x = "Net partisanship",
    y = "Policy liberalism index",
    color = "Region",
    caption = "Source: Correlates of State Policy Project"
  ) +
  theme_minimal()
```



Southern states cluster on the right in 1976 but the regional clustering shifts by 2011 as partisan sorting hardened. This is a case where combining facets (separation across years) with color (separation across regions) reveals structure that neither technique could show on its own.

5.15 Exercise

Use `gapminder` to make a comparison plot for 2007.

1. Filter the data to `year == 2007`.
2. Make a scatterplot of GDP per capita and life expectancy.
3. Separate continents using color.
4. Make a second version using `facet_wrap(~ continent)`.
5. Decide which version makes the comparison clearer and write one sentence explaining why.

6 Distributions

6.1 Why distributions matter

A distribution shows how values of a variable are spread out. Before comparing means, fitting models, or making polished charts, it is often useful to ask simpler questions:

- What values are common?
- Are there extreme values?
- Is the variable symmetric or skewed?
- Do groups have similar or different spreads?
- Are there gaps, clusters, or unusual cases?

Distribution plots are especially useful during exploratory data analysis.

The previous chapters focused mostly on relationships between variables: income and life expectancy, partisanship and ideology, or groups separated by color and facets. This chapter asks a different kind of question. Instead of starting with a relationship, it starts with one variable and asks what its values look like.

6.2 Histograms

A histogram divides a continuous variable into bins and counts how many observations fall in each bin.

The `midwest` dataset is included with `ggplot2`. It has one row per county in five Midwestern states.

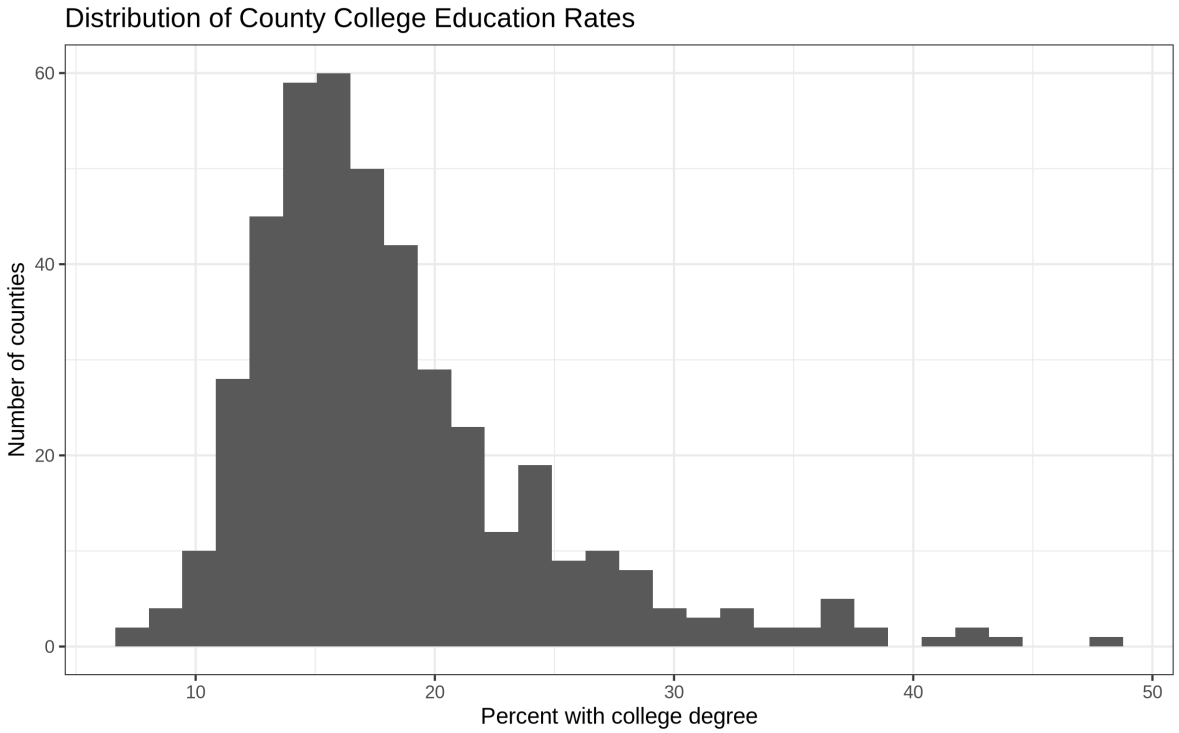
```
midwest |>
  select(state, county, percollege, poptotal, area) |>
  head(10)
```

```
# A tibble: 10 x 5
   state county    percollege poptotal  area
   <chr> <chr>          <dbl>    <int> <dbl>
 1 IL    ADAMS           19.6    66090 0.052
```

```
 2 IL    ALEXANDER       11.2    10626 0.014
 3 IL    BOND            17.0    14991 0.022
 4 IL    BOONE           17.3    30806 0.017
 5 IL    BROWN           14.5     5836 0.018
 6 IL    BUREAU          18.9    35688 0.05
 7 IL    CALHOUN         11.9     5322 0.017
 8 IL    CARROLL         16.2    16805 0.027
 9 IL    CASS            14.1    13437 0.024
10 IL    CHAMPAIGN       41.3   173025 0.058
```

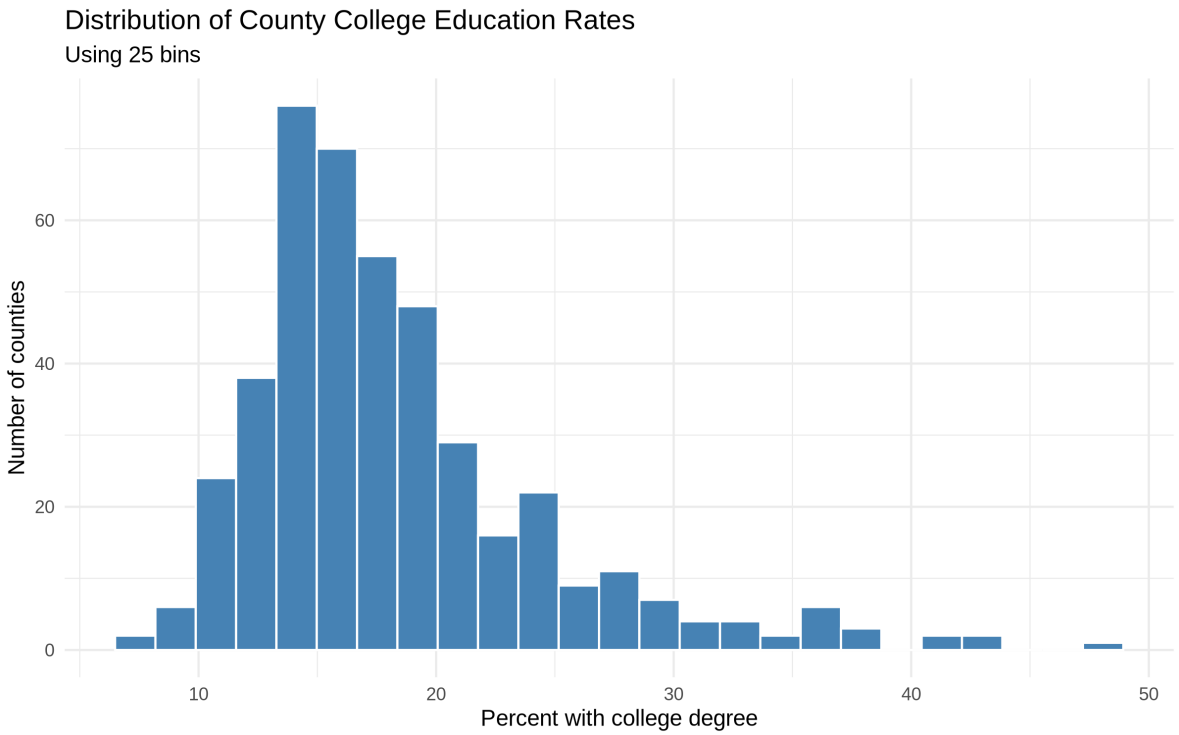
We will start with `percollege`, the percentage of adults in each county with a college degree.

```
ggplot(
  data = midwest |> filter(!is.na(percollege)),
  mapping = aes(x = percollege)
) +
  geom_histogram() +
  labs(
    title = "Distribution of County College Education Rates",
    x = "Percent with college degree",
    y = "Number of counties"
  ) +
  theme_bw()
```



When we do not set the number of bins, `ggplot2` chooses a default. That default is not always the best choice, so it is worth trying several values.

```
ggplot(
  data = midwest |> filter(!is.na(percollege)),
  mapping = aes(x = percollege)
) +
  geom_histogram(bins = 25, fill = "steelblue", color = "white") +
  labs(
    title = "Distribution of County College Education Rates",
    subtitle = "Using 25 bins",
    x = "Percent with college degree",
    y = "Number of counties"
  ) +
  theme_minimal()
```



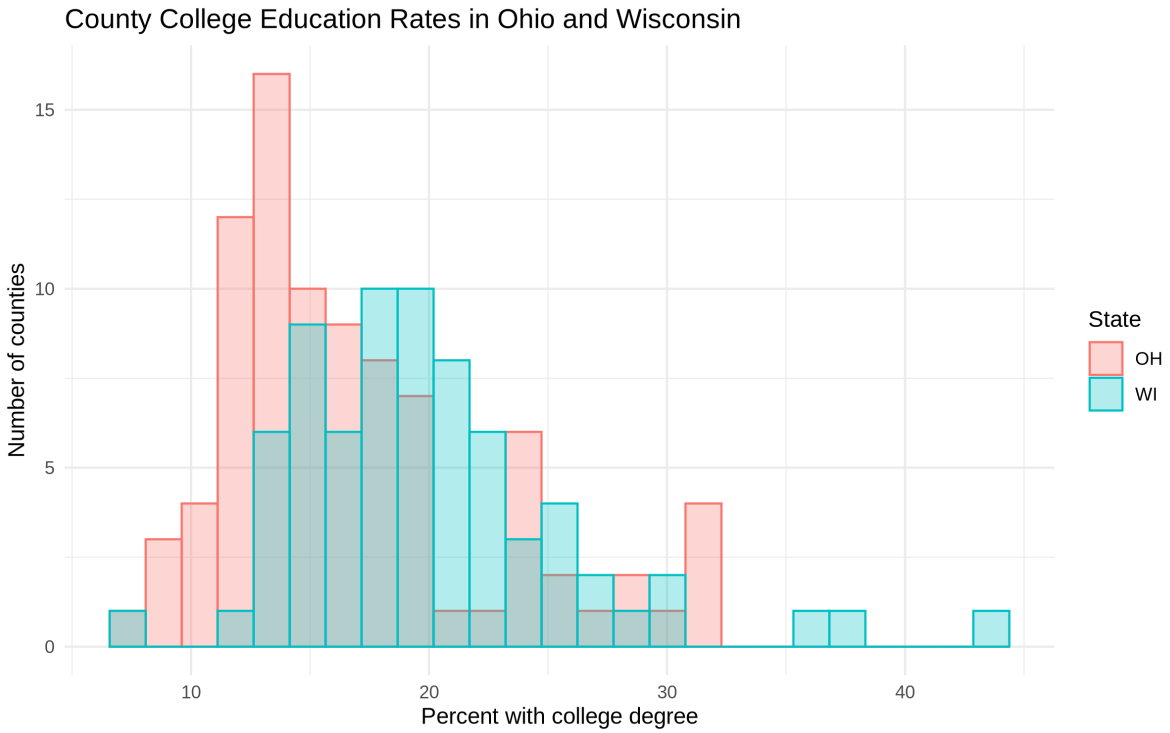
Changing the number of bins changes the visual summary. Very few bins can hide structure. Too many bins can make the plot look noisy. Histograms are useful for seeing skew, clustering, and unusual tails.

6.3 Comparing histograms

We can compare distributions across groups by mapping a group to `fill` and using transparency.

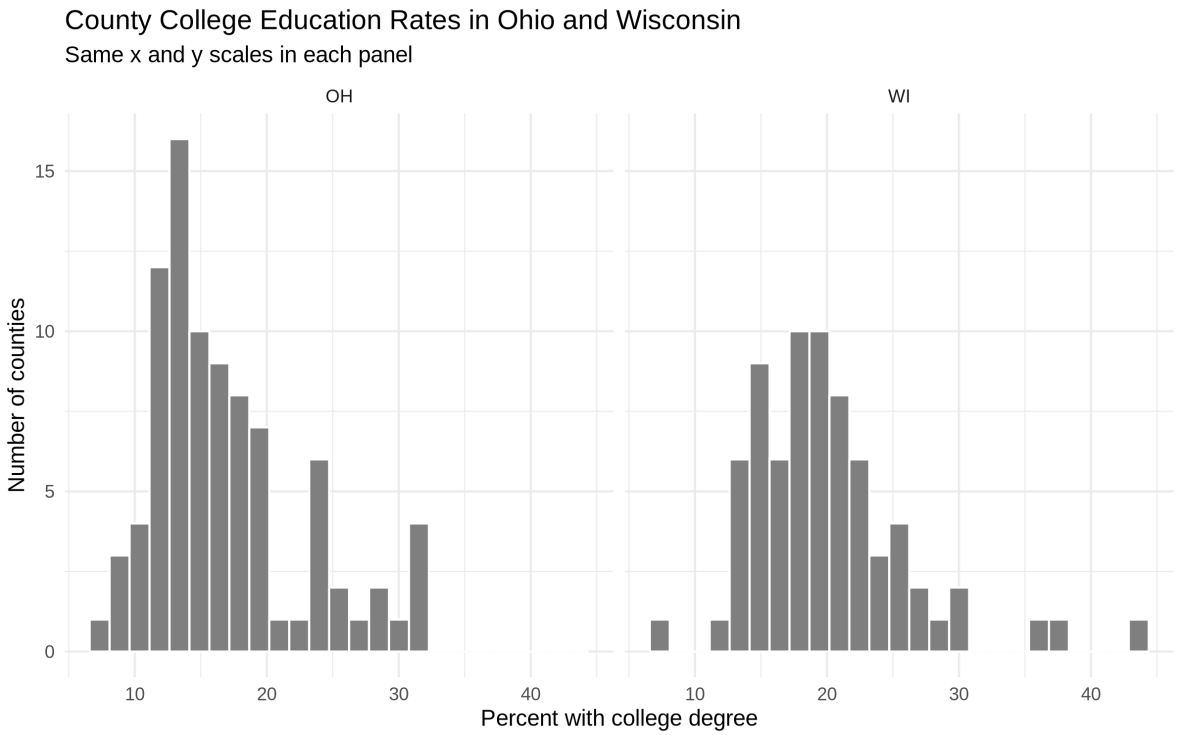
```
oh_wi <- midwest |>
  filter(state %in% c("OH", "WI"), !is.na(percollege))

ggplot(data = oh_wi, mapping = aes(x = percollege, fill = state, color = state)) +
  geom_histogram(alpha = 0.3, bins = 25, position = "identity") +
  labs(
    title = "County College Education Rates in Ohio and Wisconsin",
    x = "Percent with college degree",
    y = "Number of counties",
    fill = "State",
    color = "State"
```

```
  ) +
  theme_minimal()
```



Overlaid histograms are compact, but they can be difficult to read when the groups overlap. Small multiples are often clearer.

```
ggplot(data = oh_wi, mapping = aes(x = percollege)) +
  geom_histogram(bins = 25, fill = "gray50", color = "white") +
  facet_wrap(~ state) +
  labs(
    title = "County College Education Rates in Ohio and Wisconsin",
    subtitle = "Same x and y scales in each panel",
    x = "Percent with college degree",
    y = "Number of counties"
  ) +
  theme_minimal()
```

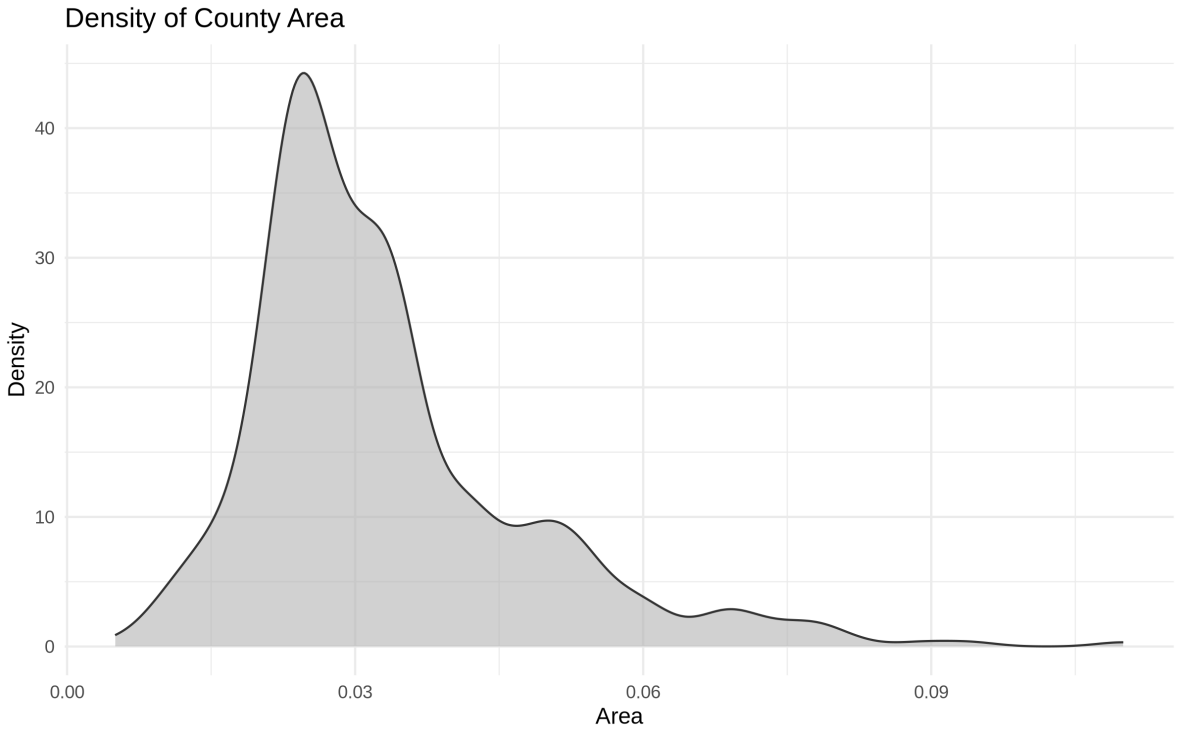


Because the panels share scales by default, the two histograms can be compared directly.

6.4 Density plots

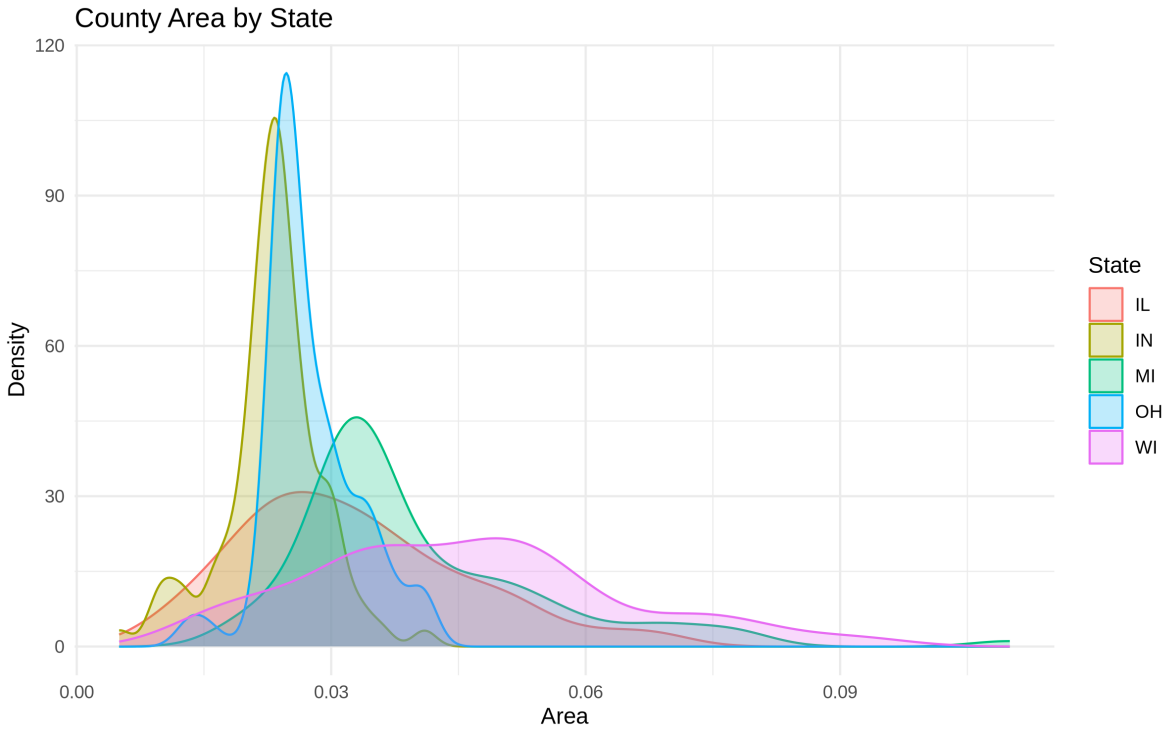
A density plot is a smoothed version of a distribution. Instead of showing counts in bins, it estimates the shape of the distribution.

```
ggplot(data = midwest, mapping = aes(x = area)) +
  geom_density(fill = "gray70", color = "gray20", alpha = 0.6) +
  labs(
    title = "Density of County Area",
    x = "Area",
    y = "Density"
  ) +
  theme_minimal()
```



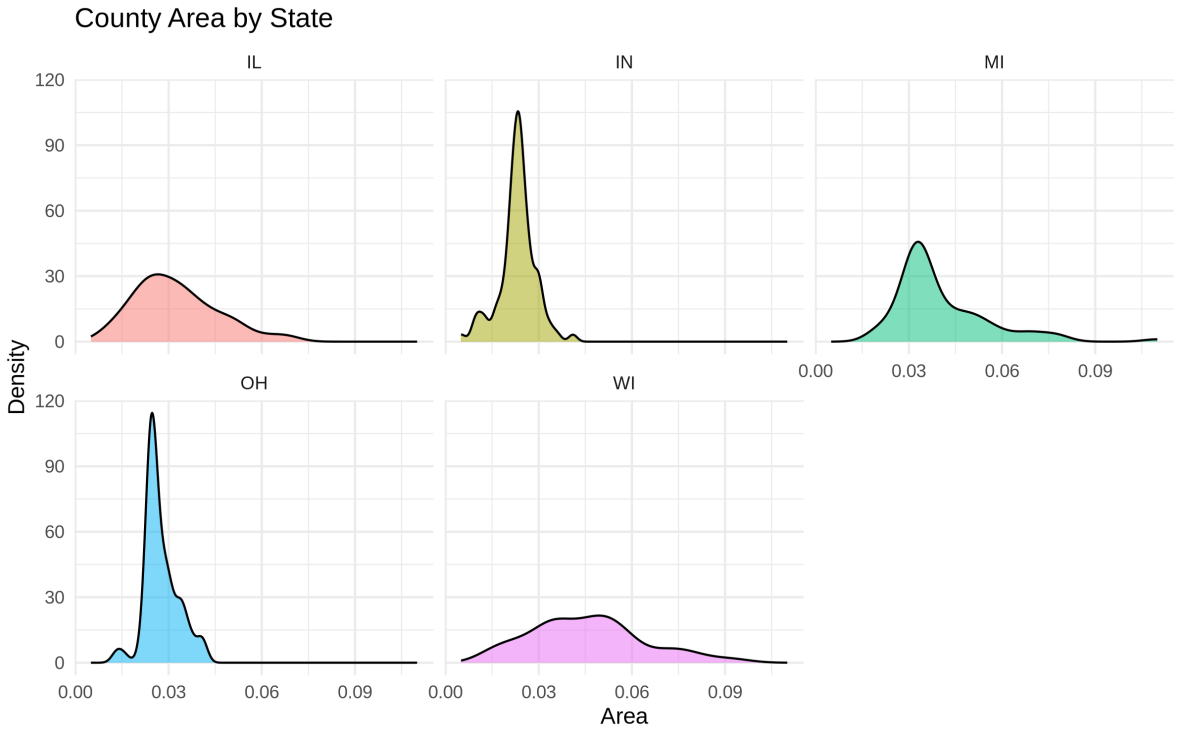
Density plots are useful for comparing shapes across groups.

```
ggplot(data = midwest, mapping = aes(x = area, fill = state, color = state)) +
  geom_density(alpha = 0.25) +
  labs(
    title = "County Area by State",
    x = "Area",
    y = "Density",
    fill = "State",
    color = "State"
  ) +
  theme_minimal()
```



When there are several groups, faceting can again reduce clutter.

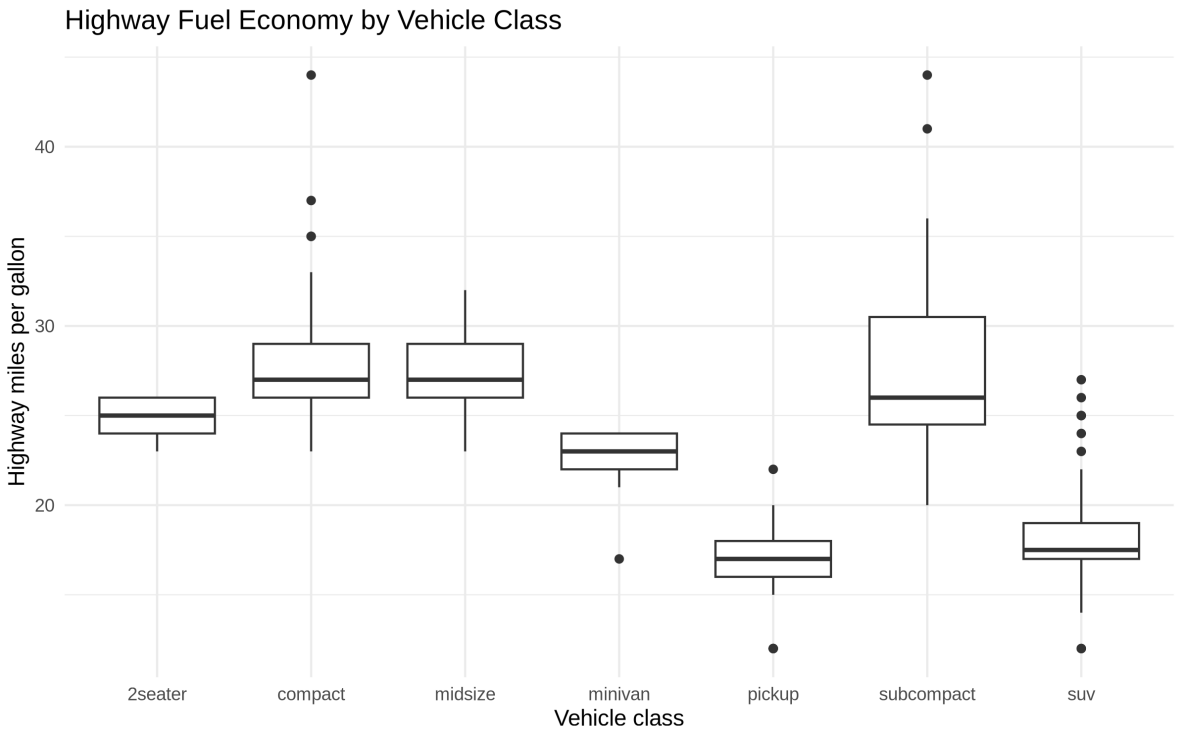
```
ggplot(data = midwest, mapping = aes(x = area, fill = state)) +
  geom_density(alpha = 0.5, show.legend = FALSE) +
  facet_wrap(~ state) +
  labs(
    title = "County Area by State",
    x = "Area",
    y = "Density"
  ) +
  theme_minimal()
```



6.5 Boxplots

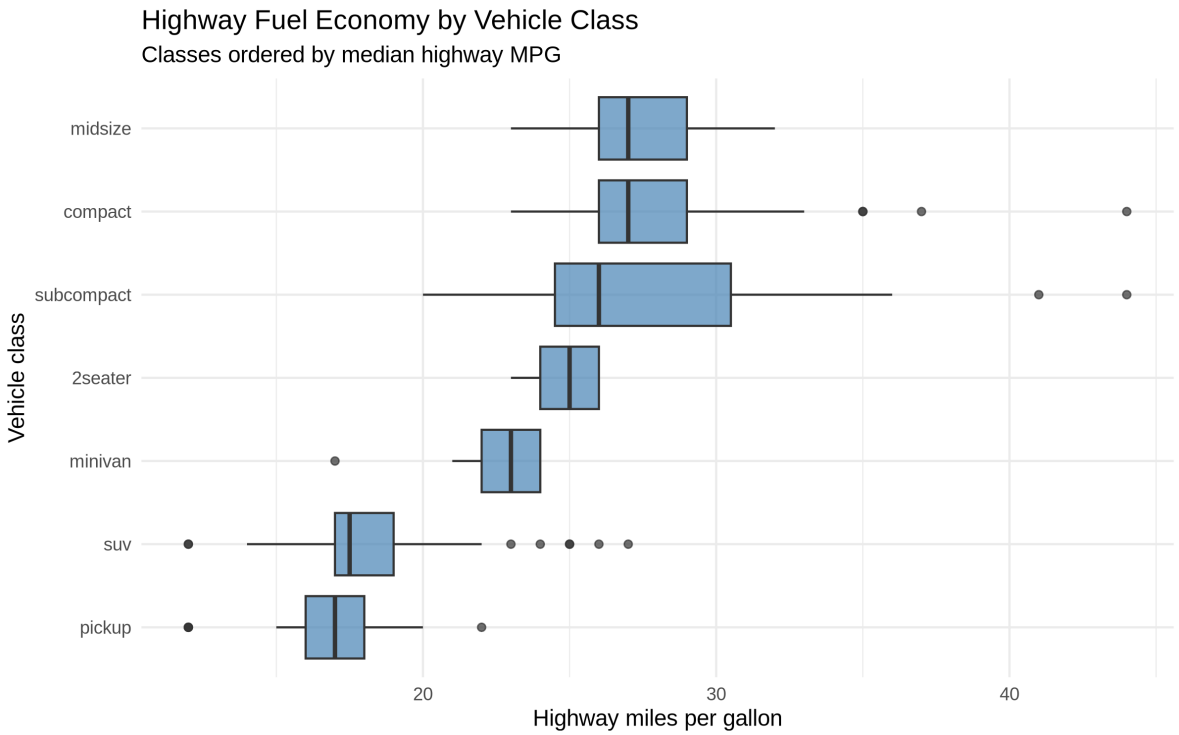
Boxplots summarize a distribution with a median, interquartile range, whiskers, and outliers. They are compact and good for comparing many groups.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  labs(
    title = "Highway Fuel Economy by Vehicle Class",
    x = "Vehicle class",
    y = "Highway miles per gallon"
  ) +
  theme_minimal()
```



The x-axis is crowded. We can reorder vehicle classes by median highway MPG and flip the coordinates.

```
ggplot(data = mpg, mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  geom_boxplot(fill = "steelblue", alpha = 0.7) +
  coord_flip() +
  labs(
    title = "Highway Fuel Economy by Vehicle Class",
    subtitle = "Classes ordered by median highway MPG",
    x = "Vehicle class",
    y = "Highway miles per gallon"
  ) +
  theme_minimal()
```



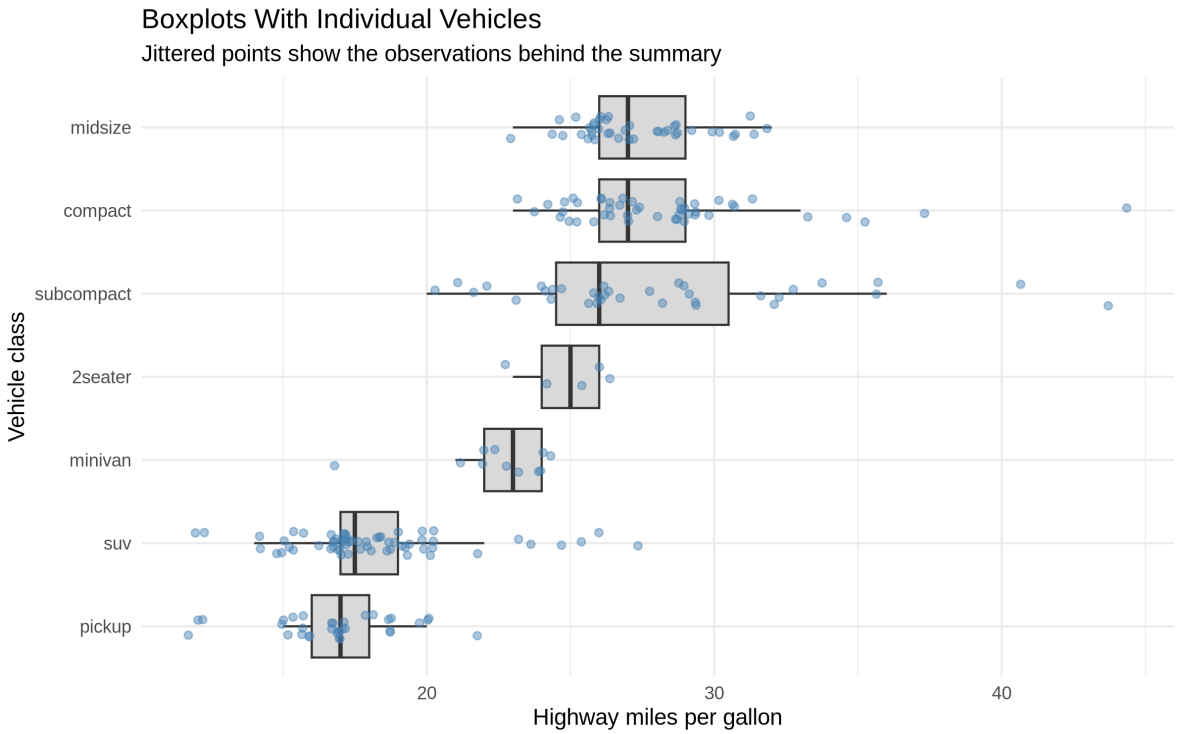
Boxplots hide individual observations, but they make the comparison of medians and spreads easy.

6.6 Adding Individual Points With Jitter

Boxplots are compact, but they can hide how many observations are behind each summary. `geom_jitter()` adds individual points with a small amount of random horizontal movement so points do not sit directly on top of each other.

```
mpg |>
  ggplot(aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  geom_boxplot(fill = "gray85", outlier.shape = NA) +
  geom_jitter(width = 0.15, alpha = 0.45, color = "steelblue") +
  coord_flip() +
  labs(
    title = "Boxplots With Individual Vehicles",
    subtitle = "Jittered points show the observations behind the summary",
    x = "Vehicle class",
    y = "Highway miles per gallon"
```

```
  ) +
  theme_minimal()
```

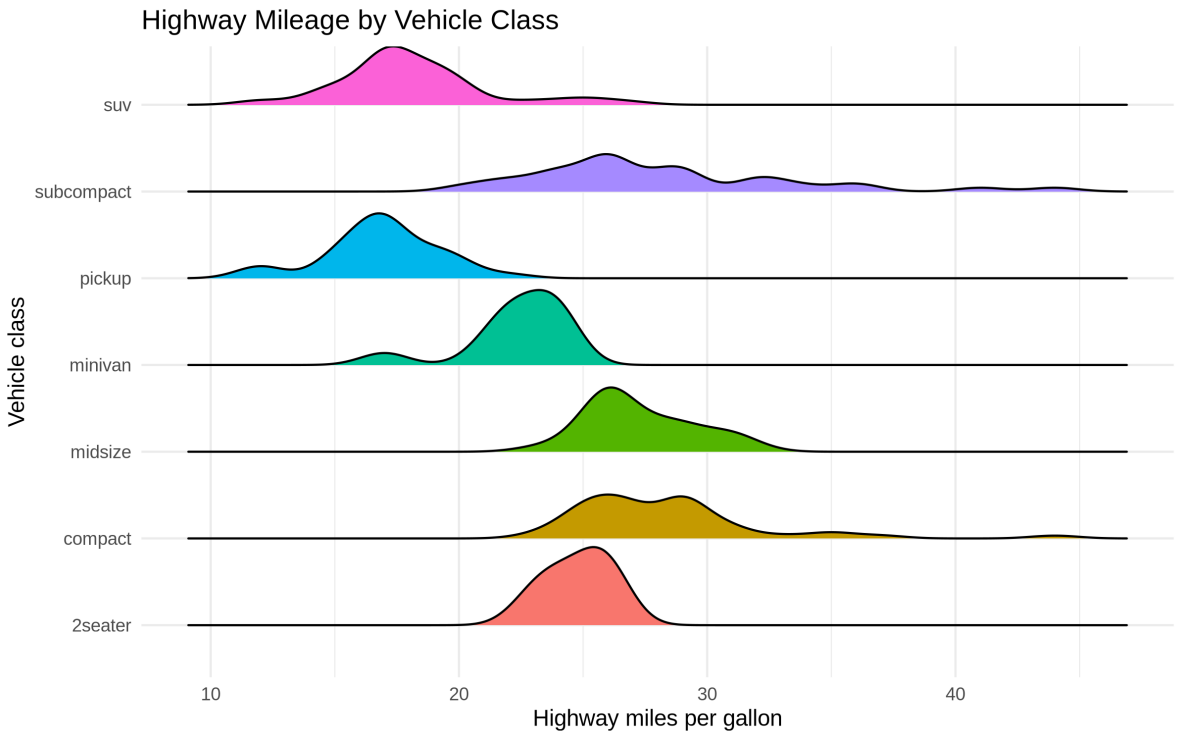


The `width` argument controls how far points can move side to side. Jittering is useful when many observations share similar values and would otherwise overlap.

6.7 Ridge plots

Ridge plots stack density plots vertically. They are useful when you want to compare many distributions and still see the shape of each one. They use `geom_density_ridges()` from the `ggridges` package.

```
ggplot(mpg, aes(x = hwy, y = class, fill = class)) +
  ggridges::geom_density_ridges(scale = 0.9, show.legend = FALSE) +
  labs(
    title = "Highway Mileage by Vehicle Class",
    x = "Highway miles per gallon",
    y = "Vehicle class"
  ) +
  theme_minimal()
```



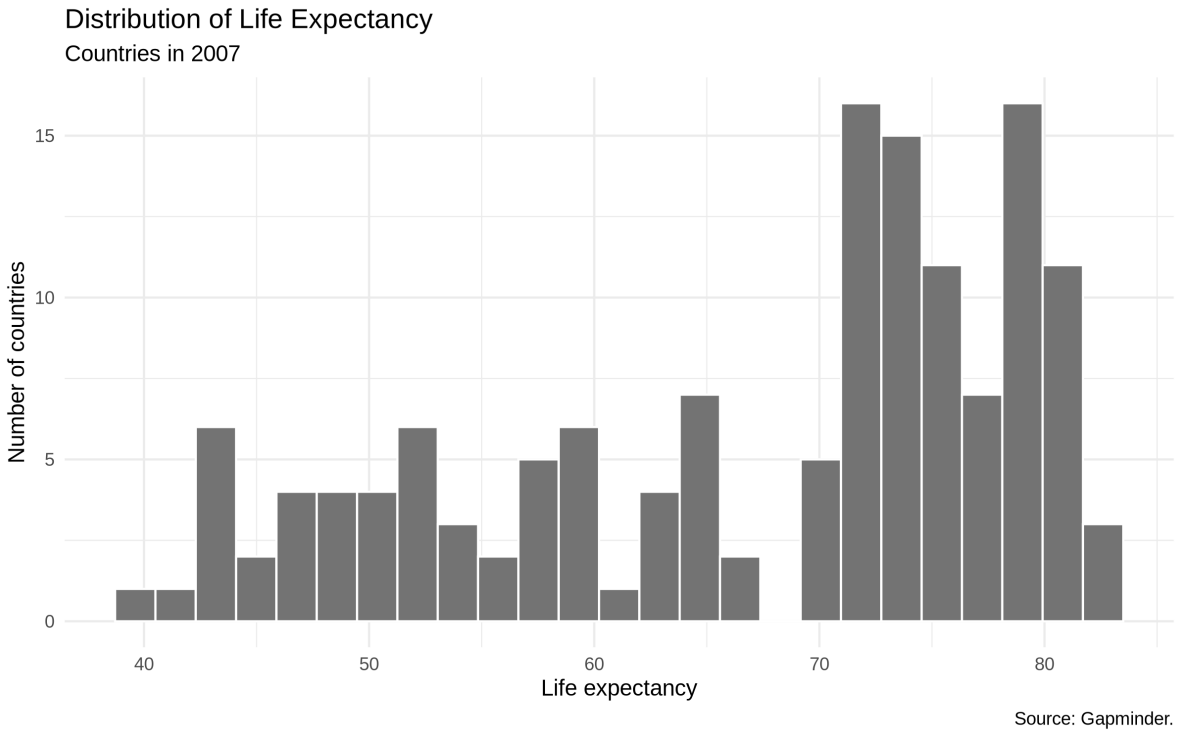
Ridge plots work best when there are enough categories to make a single boxplot crowded, but few enough that each density still has room to be readable.

6.8 Distributions in Gapminder

Distribution plots can also help before making cross-national comparisons.

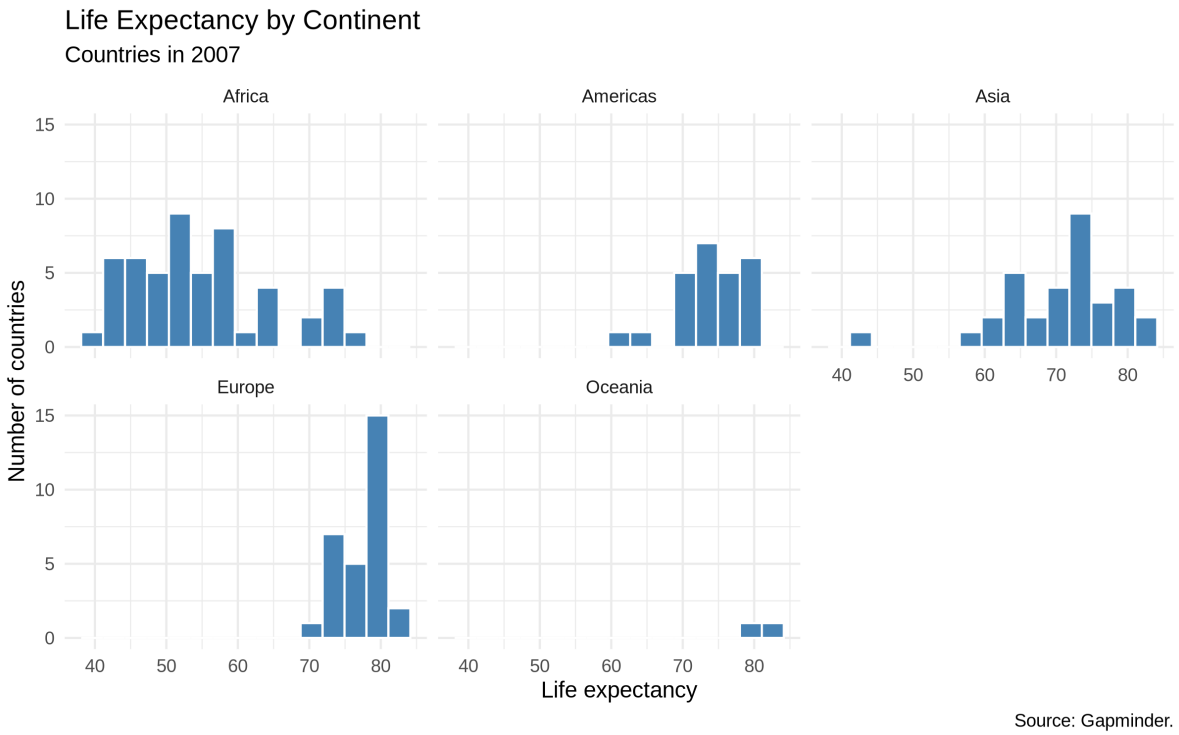
```
gap_2007 <- gapminder |>
  filter(year == 2007)

ggplot(data = gap_2007, mapping = aes(x = lifeExp)) +
  geom_histogram(bins = 25, fill = "gray45", color = "white") +
  labs(
    title = "Distribution of Life Expectancy",
    subtitle = "Countries in 2007",
    x = "Life expectancy",
    y = "Number of countries",
    caption = "Source: Gapminder."
```

```
  ) +
  theme_minimal()
```



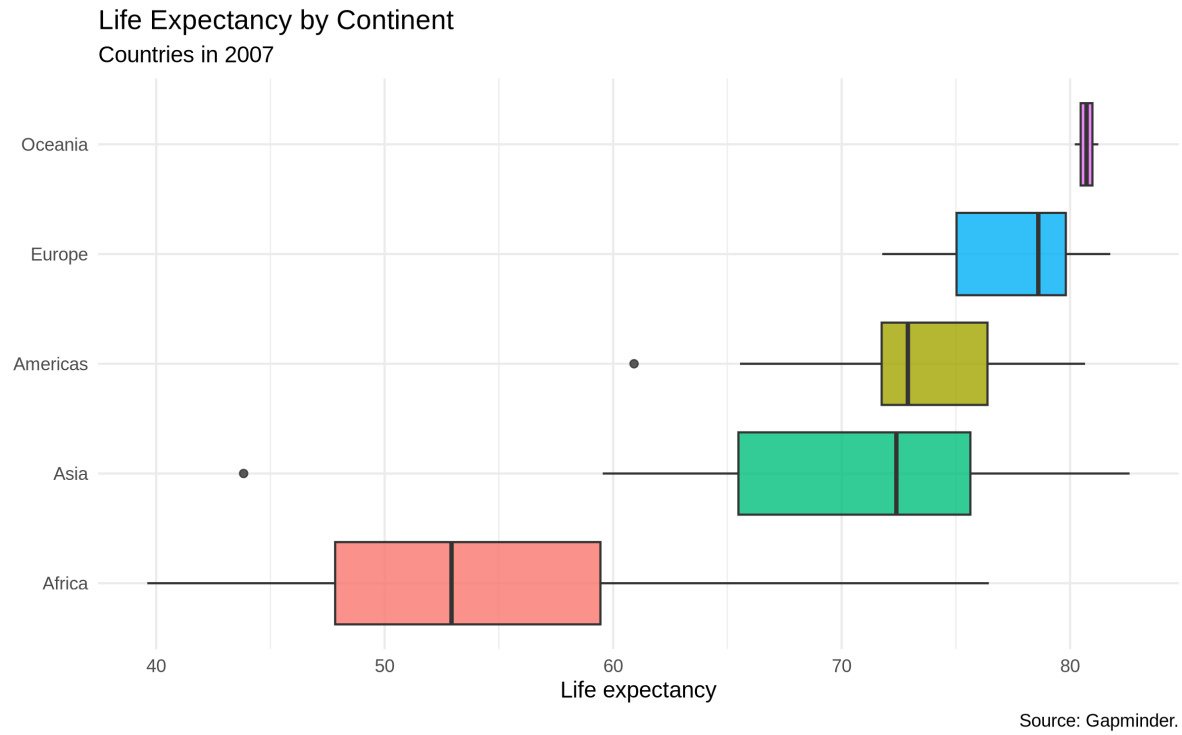
Now separate by continent. Faceted histograms keep the same geom but split the data into panels.

```
ggplot(data = gap_2007, mapping = aes(x = lifeExp)) +
  geom_histogram(bins = 15, fill = "steelblue", color = "white") +
  facet_wrap(~ continent) +
  labs(
    title = "Life Expectancy by Continent",
    subtitle = "Countries in 2007",
    x = "Life expectancy",
    y = "Number of countries",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



A boxplot gives a more compact comparison.

```
ggplot(data = gap_2007, mapping = aes(x = reorder(continent, lifeExp, FUN = median), y = life
  geom_boxplot(aes(fill = continent), show.legend = FALSE, alpha = 0.8) +
  coord_flip() +
  labs(
    title = "Life Expectancy by Continent",
    subtitle = "Countries in 2007",
    x = NULL,
    y = "Life expectancy",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



6.9 Choosing among distribution plots

Use a histogram when you want to show counts and binning is meaningful.

Use a density plot when you want to compare smooth distribution shapes.

Use a boxplot when you want compact comparisons across many groups.

No single distribution plot is always best. It is normal to make several versions while exploring the data.

6.10 Exercise 1: histogram

Use the built-in `iris` dataset.

1. Create a histogram of `Sepal.Length`.
2. Try `bins = 10`, `bins = 30`, and `bins = 50`.
3. Add a title and readable axis labels.
4. Write one sentence explaining which bin choice is easiest to interpret.

```r
data(iris)

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
```

```r
# Write your plot here.
```

6.11 Exercise 2: density plot

Use the built-in `diamonds` dataset.

1. Create a density plot of `carat`.
2. Map `cut` to `fill`.
3. Use `alpha` so the overlapping densities remain visible.
4. Try a faceted version with `facet_wrap(~ cut)`.

```r
diamonds |>
  select(carat, cut, price) |>
  head(10)
```

```
# A tibble: 10 x 3
   carat cut       price
   <dbl> <ord>     <int>
 1  0.23 Ideal       326
 2  0.21 Premium     326
 3  0.23 Good        327
 4  0.29 Premium     334
 5  0.31 Good        335
 6  0.24 Very Good   336
 7  0.24 Very Good   336
 8  0.26 Very Good   337
 9  0.22 Fair        337
10  0.23 Very Good   338
```

```
# Write your plot here.
```

6.12 Exercise 3: boxplot

Use the built-in `mpg` dataset.

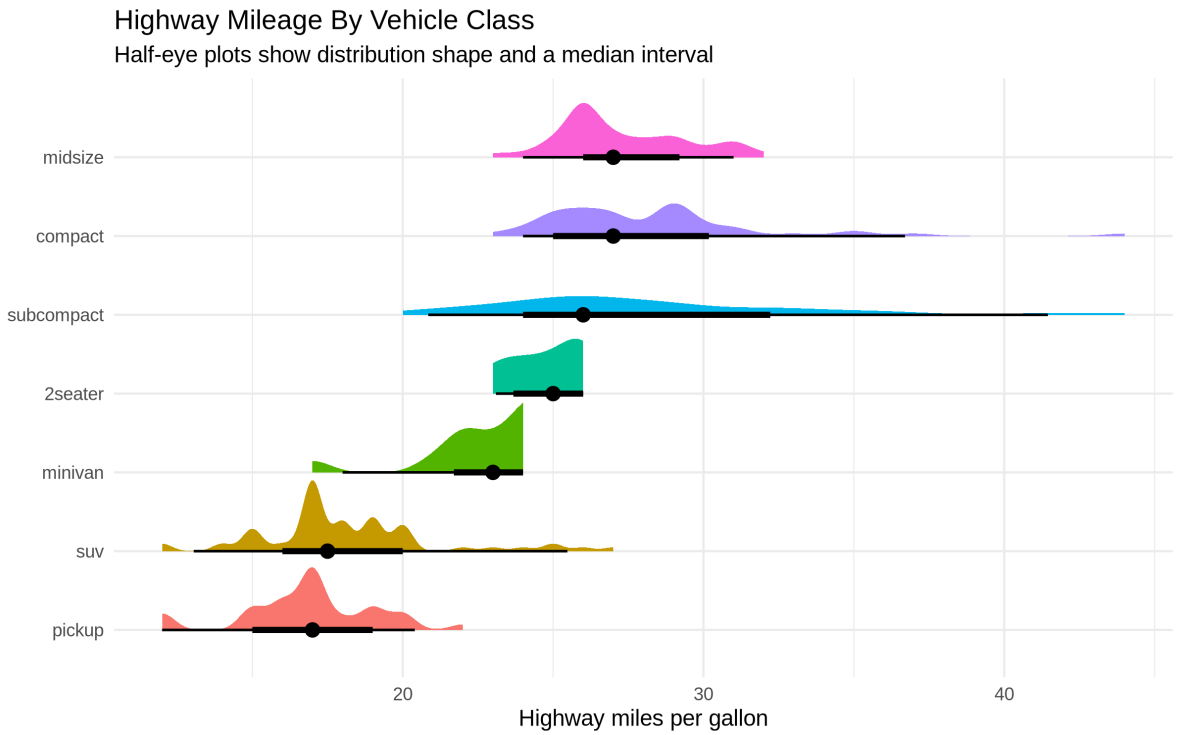
1. Compare `hwy` across vehicle `class`.
2. Reorder the classes by median `hwy`.
3. Flip the coordinates if the labels are hard to read.

```
# Write your plot here.
```

6.13 Extra: distribution intervals with `ggdist`

The `ggdist` package adds distribution geoms that combine several ideas at once. A half-eye plot shows the shape of a distribution while also marking a summary interval. It is useful when a boxplot feels too compressed but a full density plot takes too much space.

```
if (requireNamespace("ggdist", quietly = TRUE)) {
  mpg |>
    mutate(class = fct_reorder(class, hwy, .fun = median)) |>
    ggplot(aes(x = hwy, y = class, fill = class)) +
    ggdist::stat_halfeye(
      adjust = 0.7,
      width = 0.65,
      point_interval = ggdist::median_qi,
      show.legend = FALSE
    ) +
    labs(
      title = "Highway Mileage By Vehicle Class",
      subtitle = "Half-eye plots show distribution shape and a median interval",
      x = "Highway miles per gallon",
      y = NULL
    ) +
    theme_minimal()
}
```

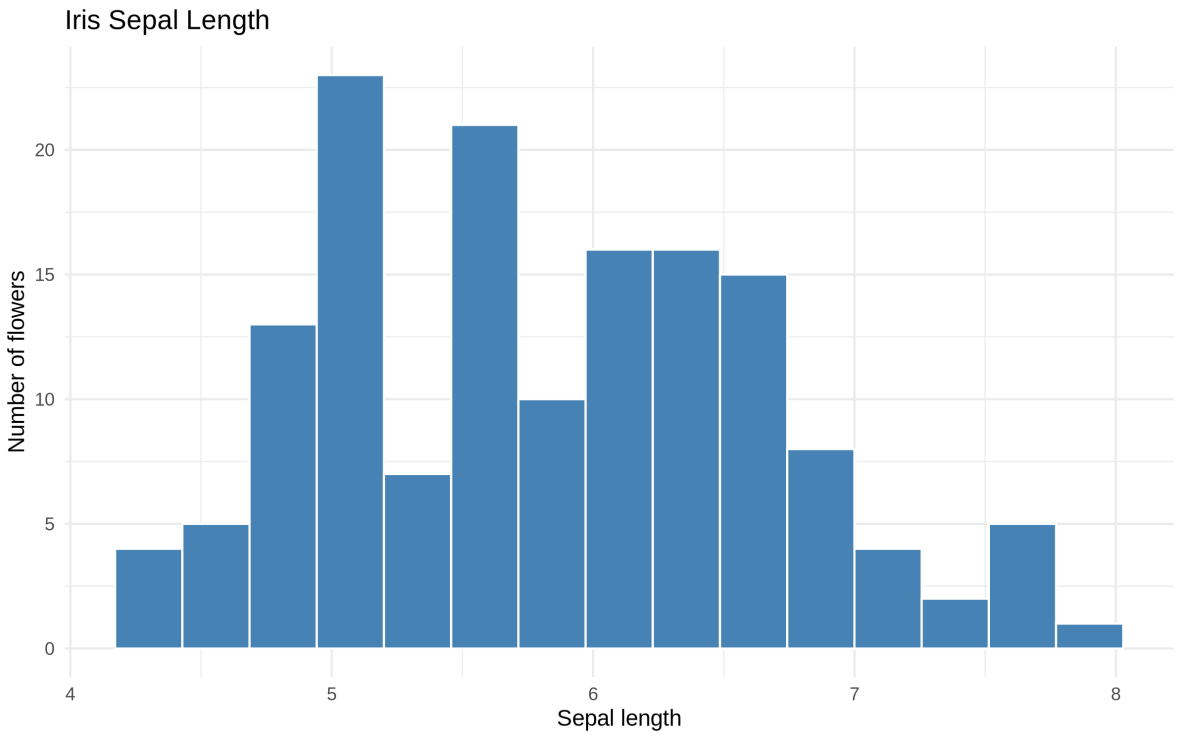


This is not a replacement for histograms, densities, or boxplots. It is another option when the goal is to show both distribution shape and uncertainty or spread in a compact display.

6.14 Extra: worked exercise variants

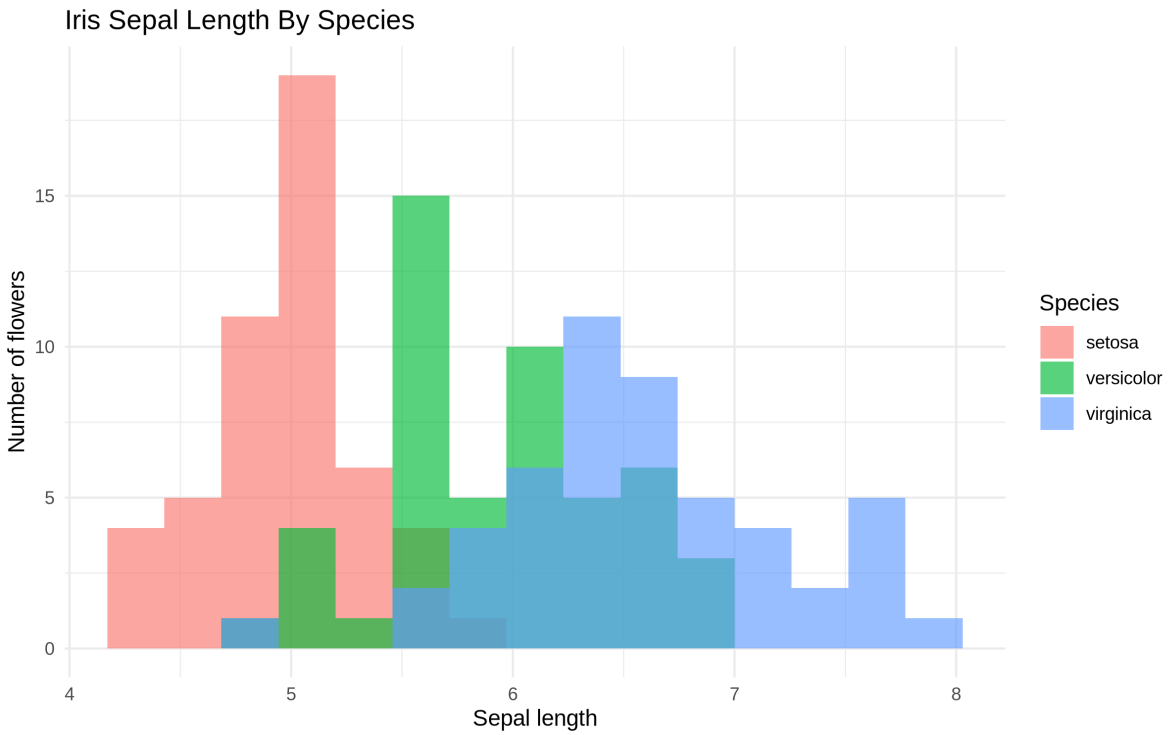
The next three chunks are short exercise-style examples. Each uses a built-in dataset and changes only one or two plotting decisions at a time.

```
iris |>
  ggplot(aes(x = Sepal.Length)) +
  geom_histogram(bins = 15, fill = "steelblue", color = "white") +
  labs(
    title = "Iris Sepal Length",
    x = "Sepal length",
    y = "Number of flowers"
  ) +
  theme_minimal()
```



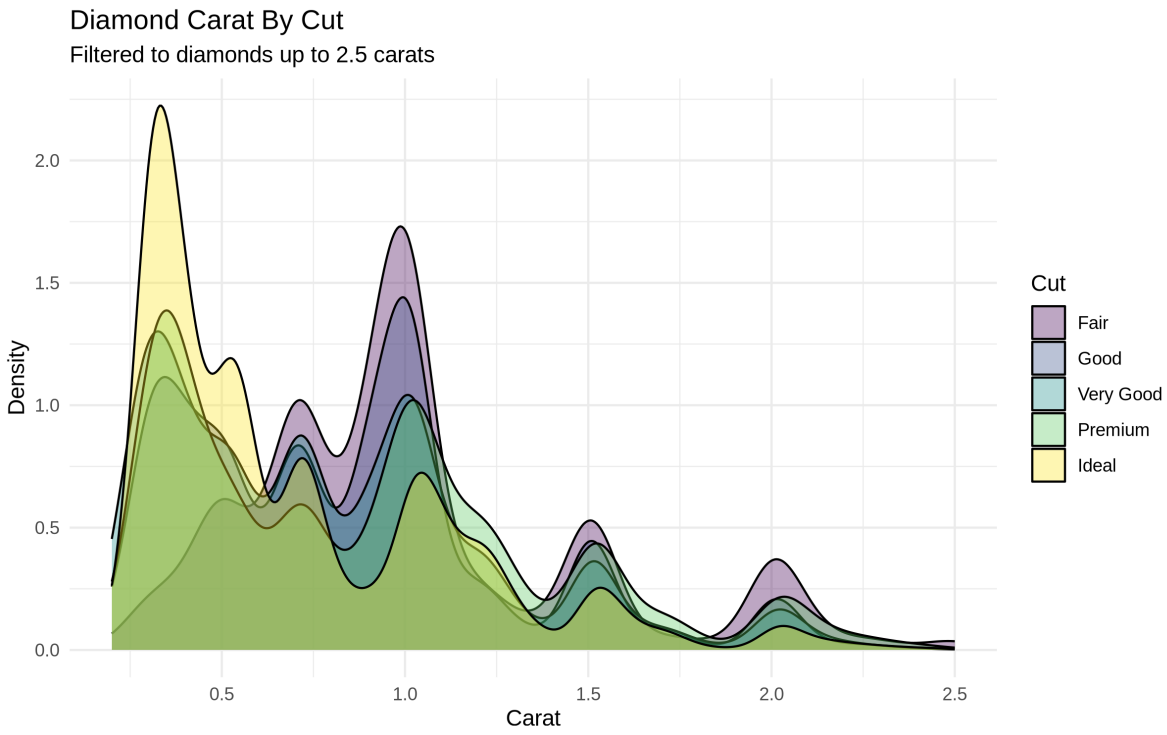
The same distribution can be separated by group. Here, `fill = Species` separates the histogram by species, while `alpha` makes overlapping bars partly transparent.

```
iris |>
  ggplot(aes(x = Sepal.Length, fill = Species)) +
  geom_histogram(bins = 15, alpha = 0.65, position = "identity") +
  labs(
    title = "Iris Sepal Length By Species",
    x = "Sepal length",
    y = "Number of flowers",
    fill = "Species"
  ) +
  theme_minimal()
```



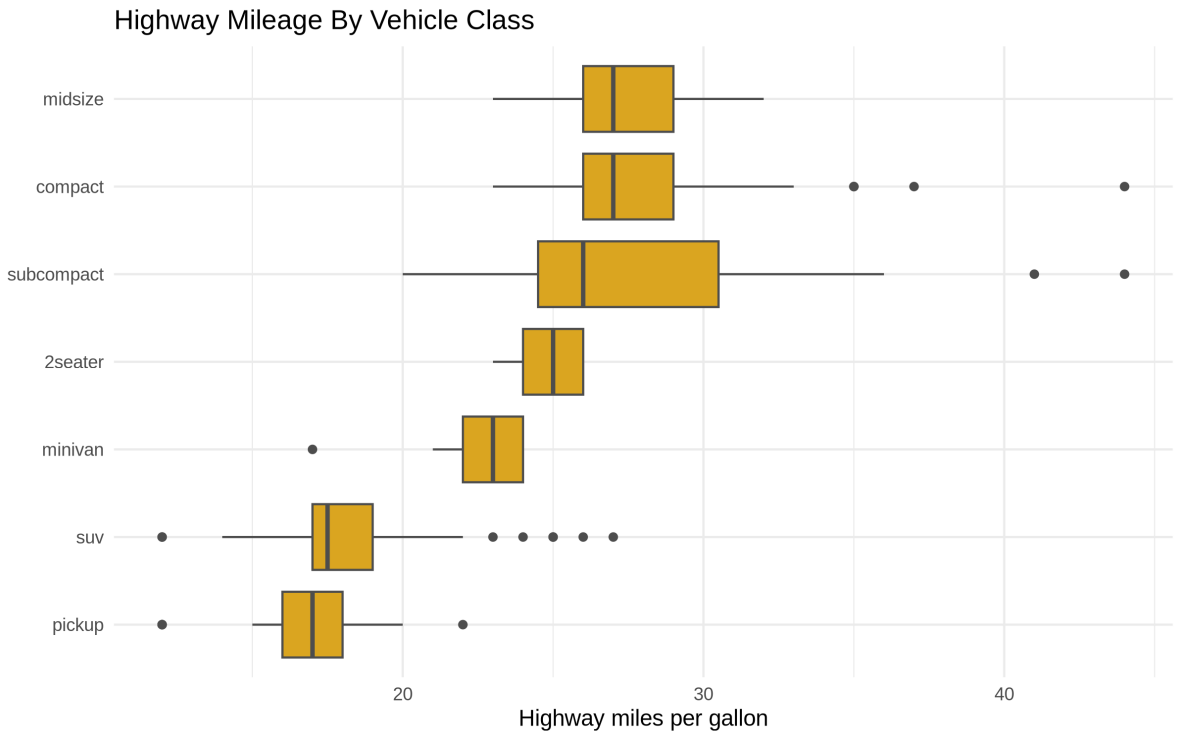
The `diamonds` dataset is larger, so density plots are often easier to compare than overlapping histograms.

```
diamonds |>
  filter(carat <= 2.5) |>
  ggplot(aes(x = carat, fill = cut)) +
  geom_density(alpha = 0.35) +
  labs(
    title = "Diamond Carat By Cut",
    subtitle = "Filtered to diamonds up to 2.5 carats",
    x = "Carat",
    y = "Density",
    fill = "Cut"
  ) +
  theme_minimal()
```



Boxplots are useful when the goal is a compact comparison of medians and spread across categories.

```
mpg |>
  ggplot(aes(x = fct_reorder(class, hwy, .fun = median), y = hwy)) +
  geom_boxplot(fill = "goldenrod", color = "gray30") +
  coord_flip() +
  labs(
    title = "Highway Mileage By Vehicle Class",
    x = NULL,
    y = "Highway miles per gallon"
  ) +
  theme_minimal()
```



6.15 Extra: PISA score distributions

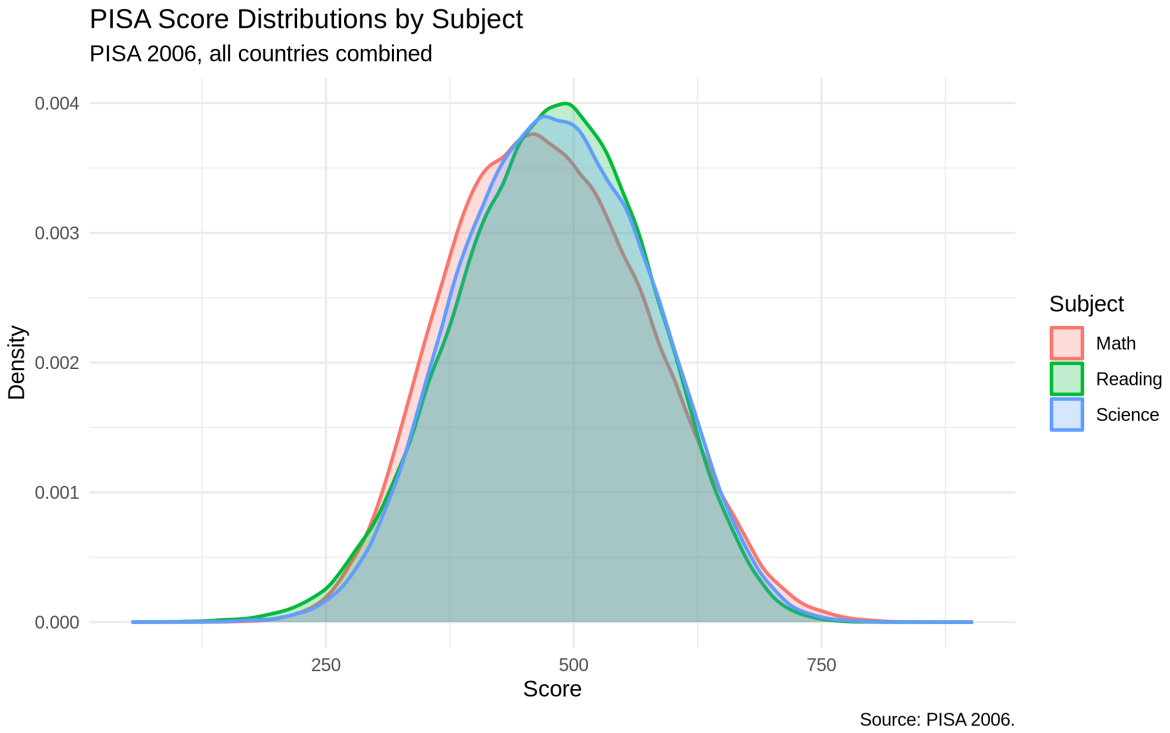
Individual-level data like PISA is well suited for distribution plots because 152,000 students provide enough data to make the shape of each distribution clearly visible.

```
pisa <- read_csv("Data/pisa/pisa_data_clean.csv", show_col_types = FALSE) |>
  janitor::clean_names()
```

A density plot comparing subject-score distributions shows where the three subjects have similar or different shapes.

```
pisa_subjects <- pisa |>
  select(
    Math    = average_math_score,
    Reading = average_reading_score,
    Science = average_science_score
  ) |>
  pivot_longer(everything(), names_to = "subject", values_to = "score")
```

```
ggplot(pisa_subjects, aes(x = score, fill = subject, color = subject)) +
  geom_density(alpha = 0.25, linewidth = 0.8) +
  labs(
    title = "PISA Score Distributions by Subject",
    subtitle = "PISA 2006, all countries combined",
    x = "Score",
    y = "Density",
    fill = "Subject",
    color = "Subject",
    caption = "Source: PISA 2006."
  ) +
  theme_minimal()
```

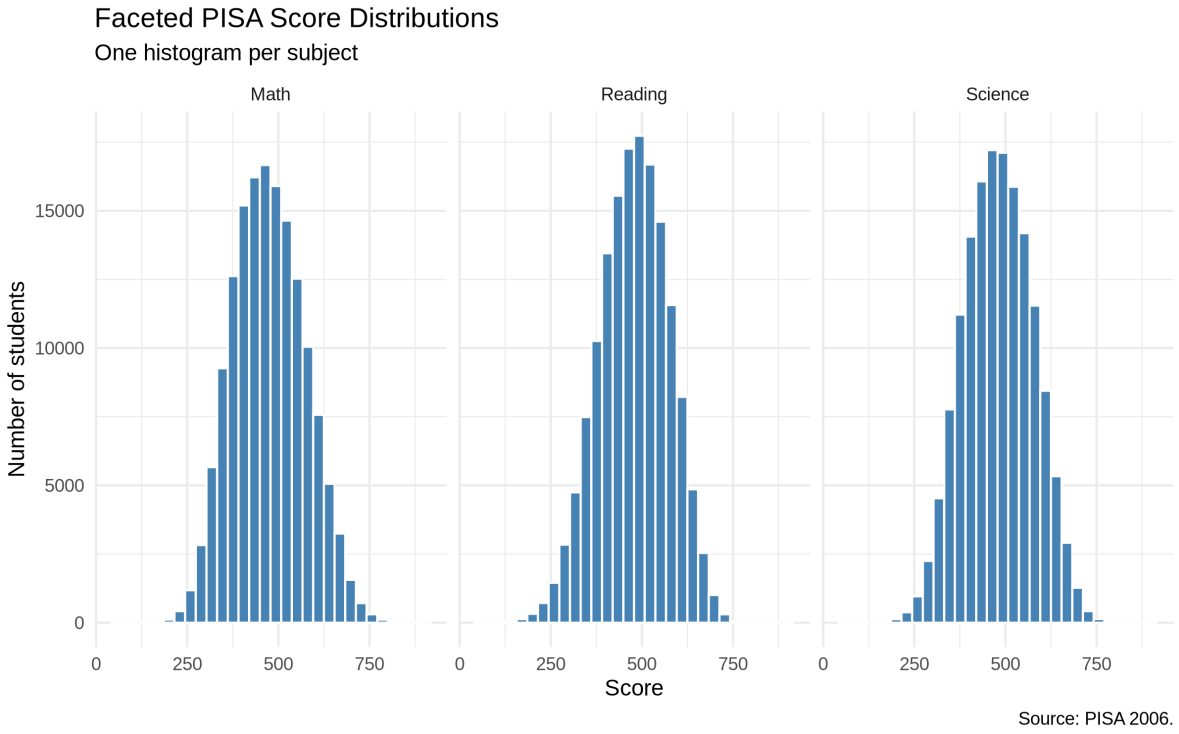


The distributions overlap substantially. The point of this plot is not to rank countries, but to compare the shape and spread of the three subject scores.

Faceting separates the subjects so each distribution has its own panel.

```
pisa_subjects |>
  ggplot(aes(x = score)) +
  geom_histogram(bins = 30, fill = "steelblue", color = "white") +
```

```
  facet_wrap(~ subject) +
  labs(
    title = "Faceted PISA Score Distributions",
    subtitle = "One histogram per subject",
    x = "Score",
    y = "Number of students",
    caption = "Source: PISA 2006."
  ) +
  theme_minimal()
```

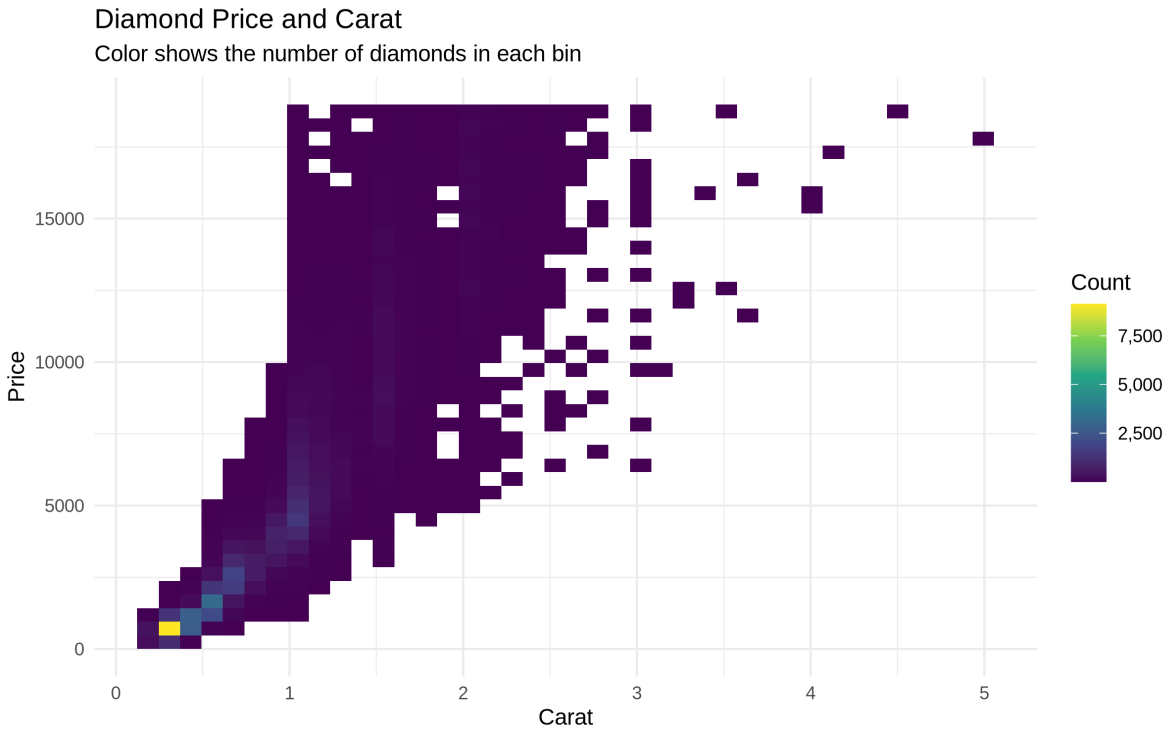


The faceted histograms make the subject comparison readable without placing every distribution on top of every other distribution.

6.16 Extra: heatmaps for dense two-variable distributions

When there are too many points, a scatterplot can turn into a cloud of overplotting. A two-dimensional bin plot counts observations inside rectangular bins.

```r
ggplot(data = diamonds, mapping = aes(x = carat, y = price)) +
  geom_bin2d(bins = 40) +
  scale_fill_viridis_c(labels = comma) +
  labs(
    title = "Diamond Price and Carat",
    subtitle = "Color shows the number of diamonds in each bin",
    x = "Carat",
    y = "Price",
    fill = "Count"
  ) +
  theme_minimal()
```



This is a distribution plot for two variables at once. It is useful when individual points would overlap too much.

7 6b: Misc Density

7.1 Overview

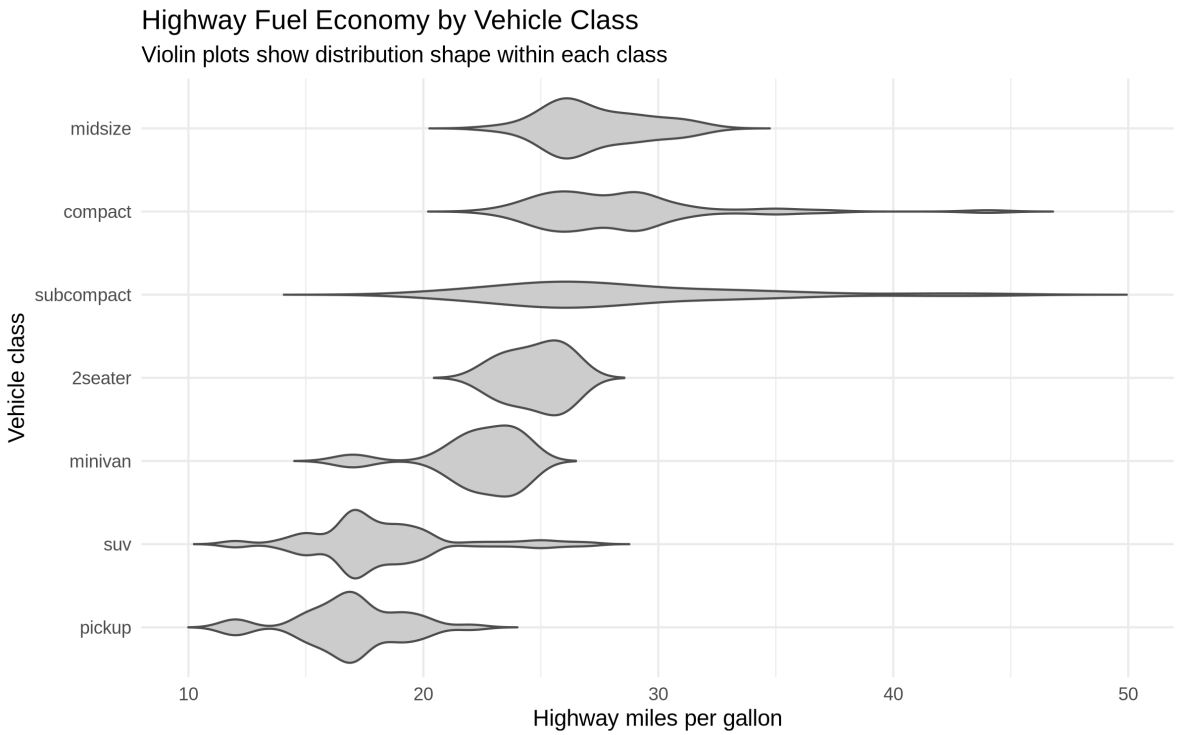
This lesson covers several additional distribution displays:

- violin plots, which show distribution shape by group
- sina plots, which combine density shape with individual observations
- dotplots, which work well when individual observations remain readable
- ECDF plots, which compare cumulative distributions
- population pyramids, which mirror age distributions for two groups

7.2 Violin Plots

Violin plots compare distributions across groups. A violin plot is like a density plot turned sideways and mirrored. The widest parts show where values are most common. The narrow parts show where values are less common.

```
ggplot(data = mpg, mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  geom_violin(fill = "gray80", color = "gray30", trim = FALSE) +
  coord_flip() +
  labs(
    title = "Highway Fuel Economy by Vehicle Class",
    subtitle = "Violin plots show distribution shape within each class",
    x = "Vehicle class",
    y = "Highway miles per gallon"
  ) +
  theme_minimal()
```

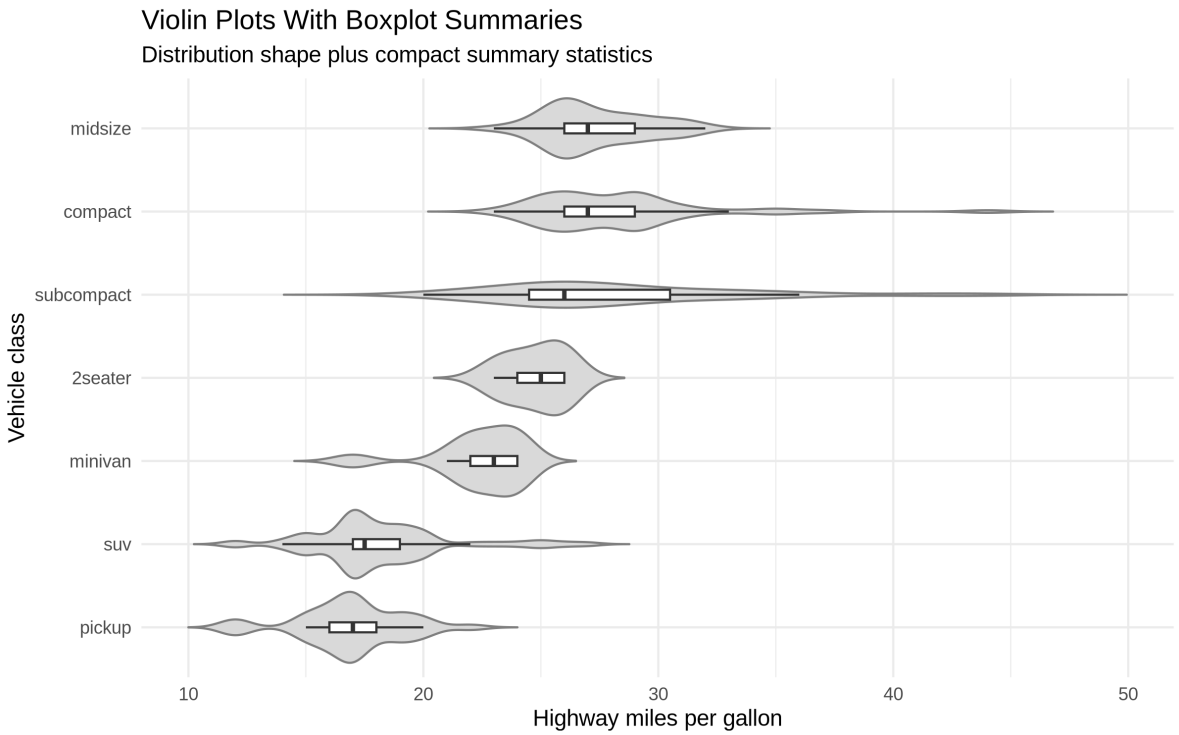


The violin plot shows more shape than a boxplot. A group can have a long tail, several clusters, or a narrow concentration of values. That information is mostly hidden in a boxplot.

7.3 Violin Plots With Boxplots

We can merge violin plots with boxplots. The violin shows the distribution shape, while the boxplot keeps the median and interquartile range visible.

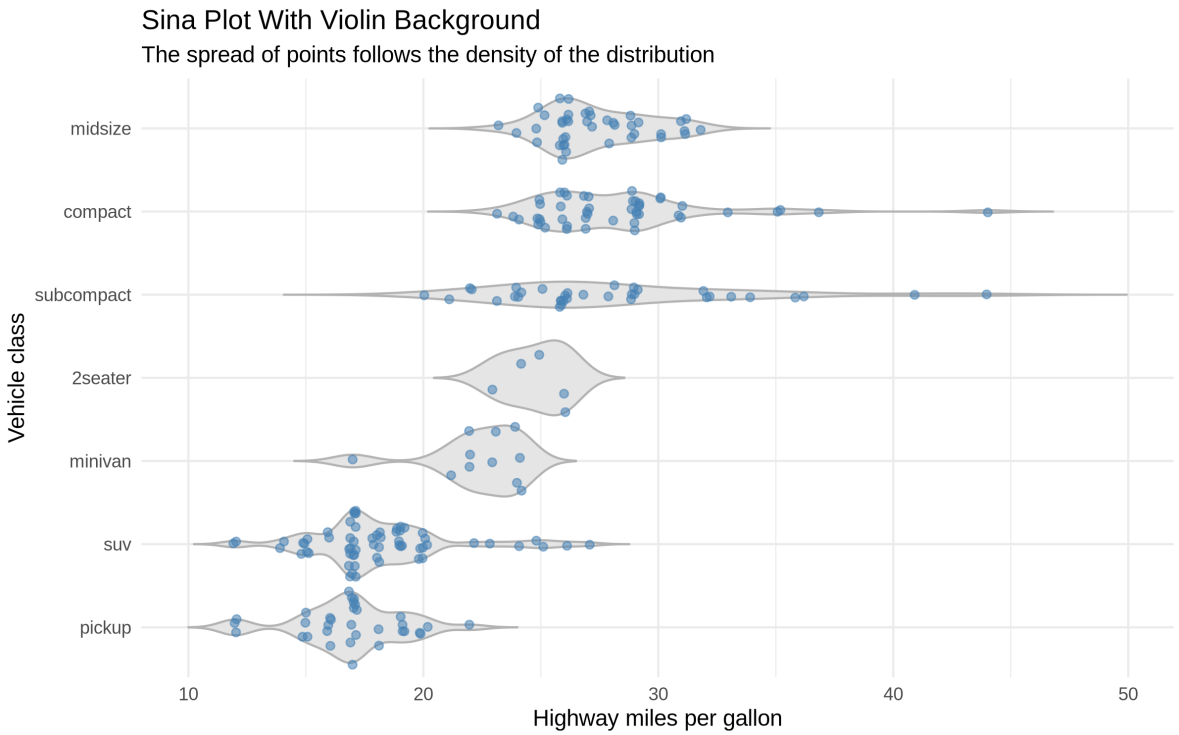
```
ggplot(data = mpg, mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  geom_violin(fill = "gray85", color = "gray50", trim = FALSE) +
  geom_boxplot(width = 0.12, outlier.shape = NA, fill = "white") +
  coord_flip() +
  labs(
    title = "Violin Plots With Boxplot Summaries",
    subtitle = "Distribution shape plus compact summary statistics",
    x = "Vehicle class",
    y = "Highway miles per gallon"
  ) +
  theme_minimal()
```



7.4 Sina Plots

A sina plot is another compromise between a violin plot and a jitter plot. The `geom_sina()` function from the `ggforce` package spreads points according to the density of the data. Dense parts of the distribution become wider; sparse parts stay narrow.

```
ggplot(data = mpg, mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  geom_violin(fill = "gray90", color = "gray70", trim = FALSE) +
  geom_sina(color = "steelblue", alpha = 0.55, size = 1.6) +
  coord_flip() +
  labs(
    title = "Sina Plot With Violin Background",
    subtitle = "The spread of points follows the density of the distribution",
    x = "Vehicle class",
    y = "Highway miles per gallon"
  ) +
  theme_minimal()
```



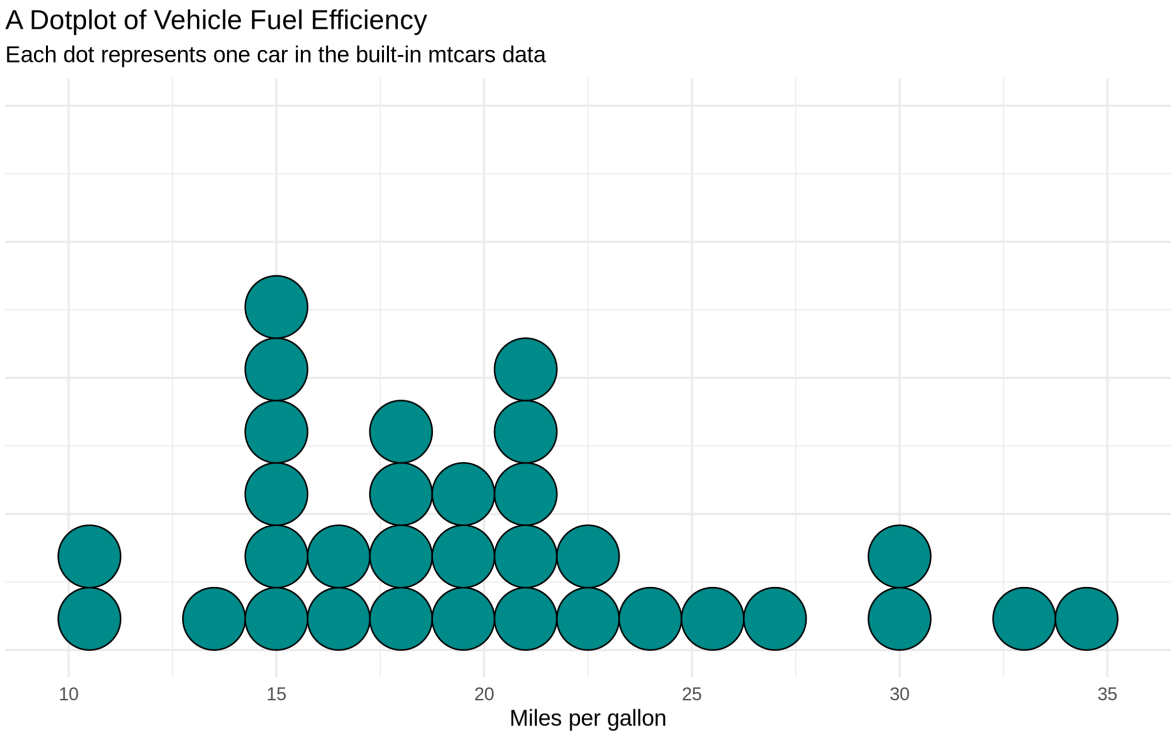
The advantage of a sina plot is that it preserves the individual observations while still communicating the shape of the distribution. The disadvantage is that it adds one more specialized geom, so it is most useful when the raw observations are substantively important.

7.5 Dotplots For Small Samples

A dotplot is useful when there are few enough observations that individual dots remain readable. It is a more literal display than a histogram or density plot because each dot represents an observation.

```
ggplot(data = mtcars, mapping = aes(x = mpg)) +
  geom_dotplot(binwidth = 1.5, method = "histodot", fill = "darkcyan") +
  labs(
    title = "A Dotplot of Vehicle Fuel Efficiency",
    subtitle = "Each dot represents one car in the built-in mtcars data",
    x = "Miles per gallon",
    y = NULL
  ) +
  theme_minimal() +
  theme(
```

```
    axis.text.y = element_blank(),
    axis.ticks.y = element_blank()
  )
```



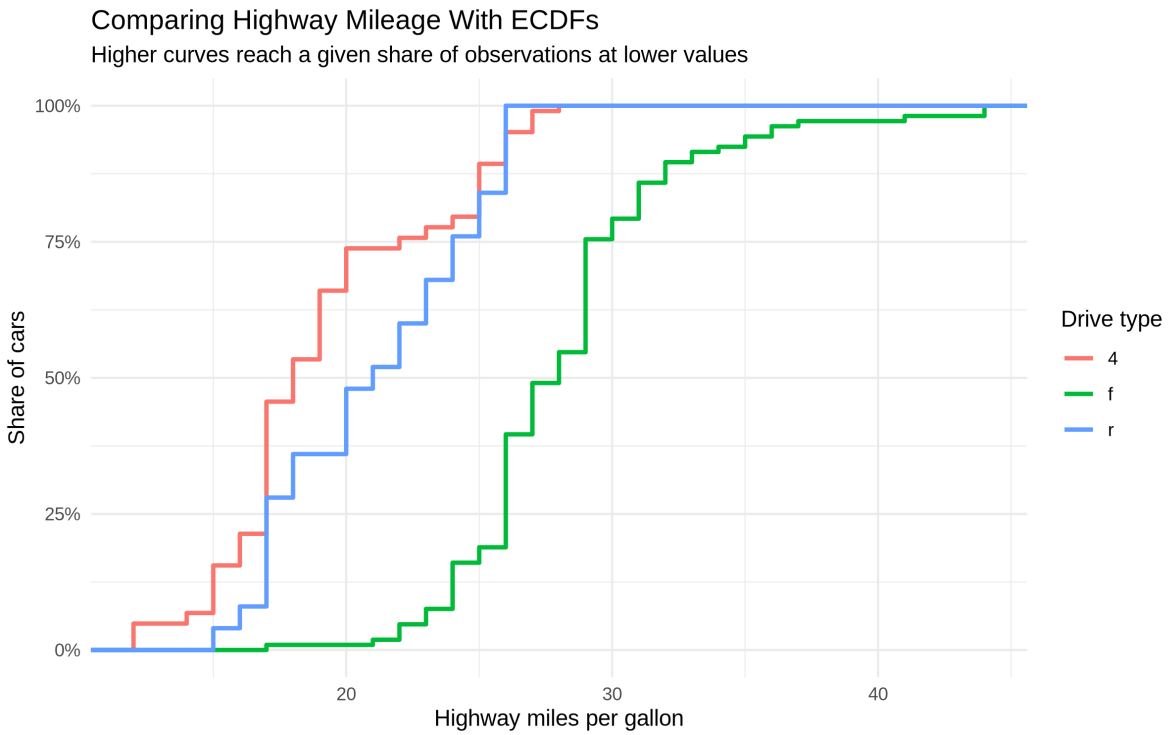
The y-axis is not meaningful in this display because the dots are stacked to avoid overlap. The important information is the horizontal position and clustering of the dots.

7.6 ECDF Plots

An empirical cumulative distribution function, or ECDF, shows the share of observations at or below each value on the x-axis. ECDF plots can be useful when several density curves would overlap too much.

```
ggplot(data = mpg, mapping = aes(x = hwy, color = drv)) +
  stat_ecdf(linewidth = 1) +
  scale_y_continuous(labels = percent) +
  labs(
    title = "Comparing Highway Mileage With ECDFs",
    subtitle = "Higher curves reach a given share of observations at lower values",
```

```
    x = "Highway miles per gallon",
    y = "Share of cars",
    color = "Drive type"
  ) +
  theme_minimal()
```



An ECDF makes medians and other percentiles easier to compare. For example, the point where a line crosses 50% is the median for that group.

There are two basic ways to read an ECDF:

- Pick an x-value and read upward to the curve. The y-value says what share of observations are at or below that x-value.
- Pick a y-value and read across to the curve. The x-value says the value at that percentile.

For example, the next chunk calculates two simple readings from the same `mpg` data. The first reading asks what share of front-wheel-drive cars have highway mileage of 30 MPG or less. The second reading asks for the median highway mileage among front-wheel-drive cars.

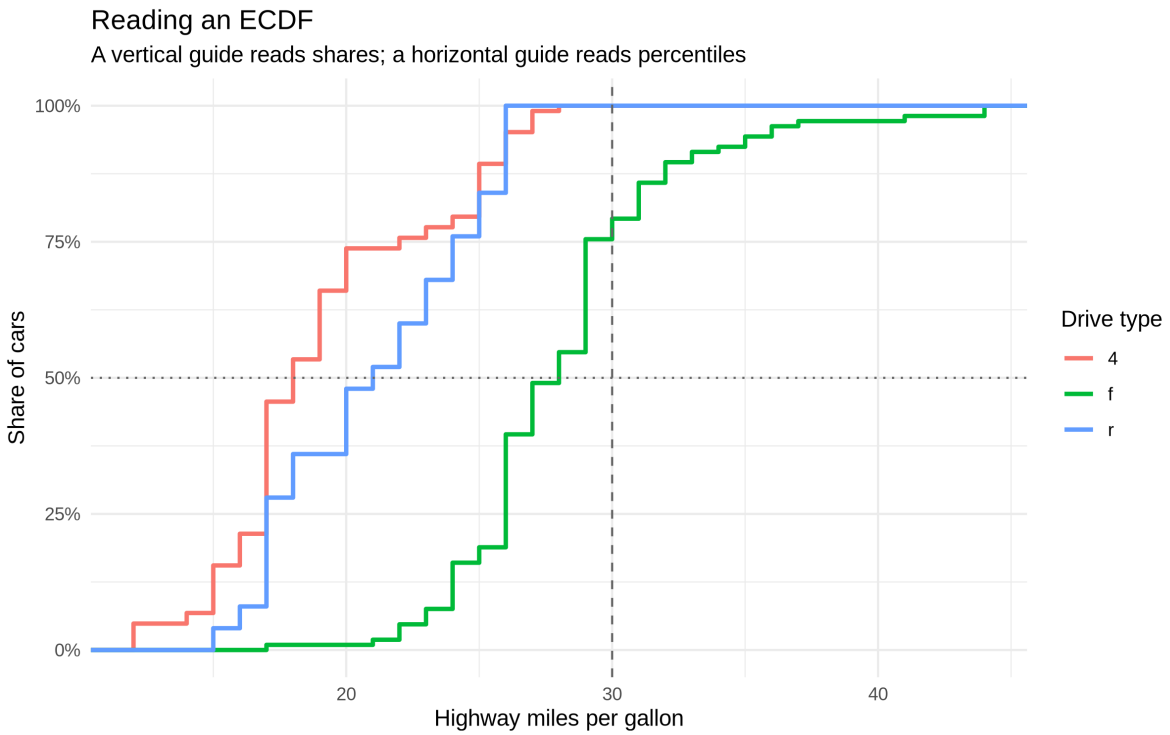
```
mpg |>
  filter(drv == "f") |>
  summarize(
```

```
    share_at_or_below_30 = mean(hwy <= 30),
    median_hwy = median(hwy)
  )
```

```
# A tibble: 1 x 2
  share_at_or_below_30 median_hwy
                 <dbl>      <dbl>
1                0.792         28
```

The ECDF plot shows the same information visually. At `hwy = 30`, the front-wheel-drive curve is near the share calculated above. At `y = 50%`, the x-position of the front-wheel-drive curve is the median.

```
ggplot(data = mpg, mapping = aes(x = hwy, color = drv)) +
  stat_ecdf(linewidth = 1) +
  geom_vline(xintercept = 30, linetype = "dashed", color = "gray40") +
  geom_hline(yintercept = 0.5, linetype = "dotted", color = "gray40") +
  scale_y_continuous(labels = percent) +
  labs(
    title = "Reading an ECDF",
    subtitle = "A vertical guide reads shares; a horizontal guide reads percentiles",
    x = "Highway miles per gallon",
    y = "Share of cars",
    color = "Drive type"
  ) +
  theme_minimal()
```



When one ECDF curve is farther to the right, that group tends to have larger values. When one curve rises very quickly, many observations are concentrated in a narrow range. When curves cross, the comparison depends on which part of the distribution is being discussed.

7.7 Population Pyramids

A population pyramid is a mirrored distribution. Age groups are placed on one axis, and two populations are placed on opposite sides of the other axis. The classic version puts men on one side and women on the other.

The `apyramid` package provides a direct population-pyramid function. `age_pyramid()` expects an age-group column, a splitting column such as sex, and, for pre-computed data, a count column.

Useful references for this package include the `apyramid` introduction vignette and the R4Epi sitrep site, which collects applied examples for public-health-oriented R workflows.

The `us_2018` data set contains population counts by age group and gender. The `levels()` line shows the order of the age groups, which matters because population pyramids should be arranged from youngest to oldest.

```
us_2018
```

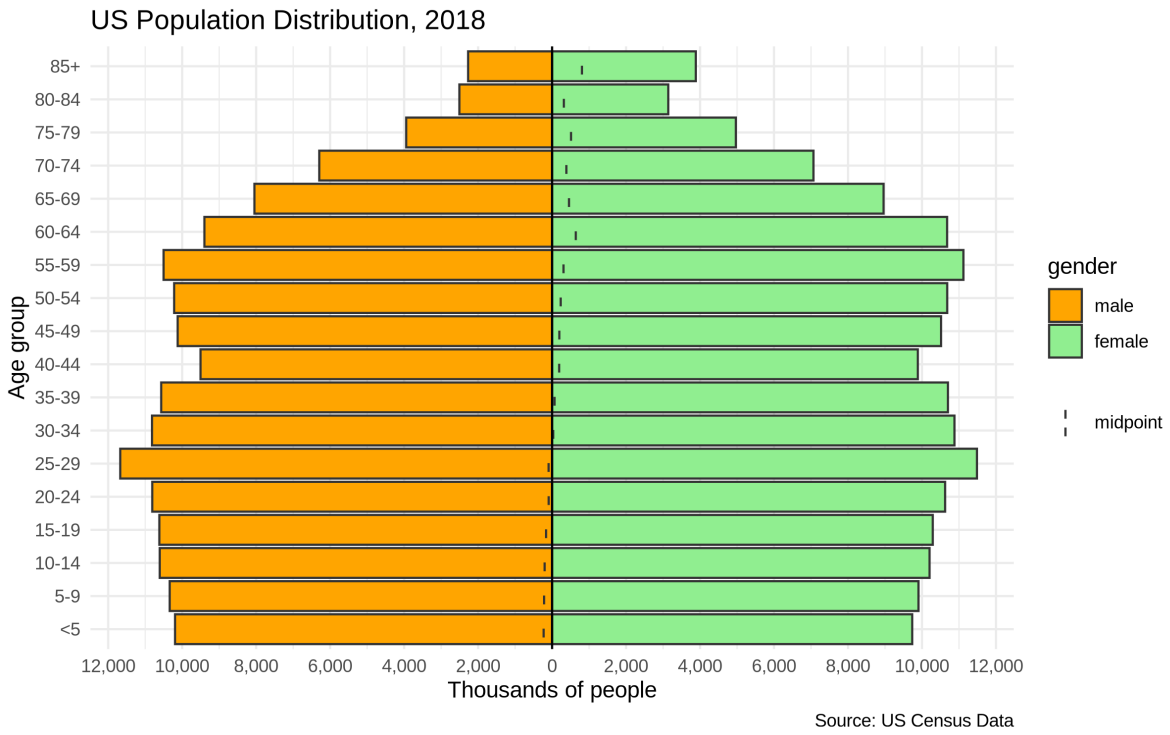
```
# A tibble: 36 x 4
   age   gender count percent
   <fct> <fct>  <int>   <dbl>
 1 <5    male   10193     6.4
 2 <5    female  9736     5.9
 3 5-9   male   10338     6.5
 4 5-9   female  9905     6
 5 10-14 male   10607     6.7
 6 10-14 female 10204     6.2
 7 15-19 male   10617     6.7
 8 15-19 female 10291     6.2
 9 20-24 male   10809     6.8
10 20-24 female 10625     6.4
# i 26 more rows
```

```
levels(us_2018$age)
```

```
 [1] "<5"    "5-9"   "10-14" "15-19" "20-24" "25-29" "30-34" "35-39" "40-44"
[10] "45-49" "50-54" "55-59" "60-64" "65-69" "70-74" "75-79" "80-84" "85+"
```

```
us_labels <- labs(
  x = "Age group",
  y = "Thousands of people",
  title = "US Population Distribution, 2018",
  caption = "Source: US Census Data"
)
```

```
age_pyramid(
  data = us_2018,
  age_group = age,
  split_by = gender,
  count = count,
  show_midpoint = TRUE,
  horizontal_lines = TRUE,
  pal = c("male" = "orange", "female" = "lightgreen")
) +
  us_labels +
  theme_minimal()
```



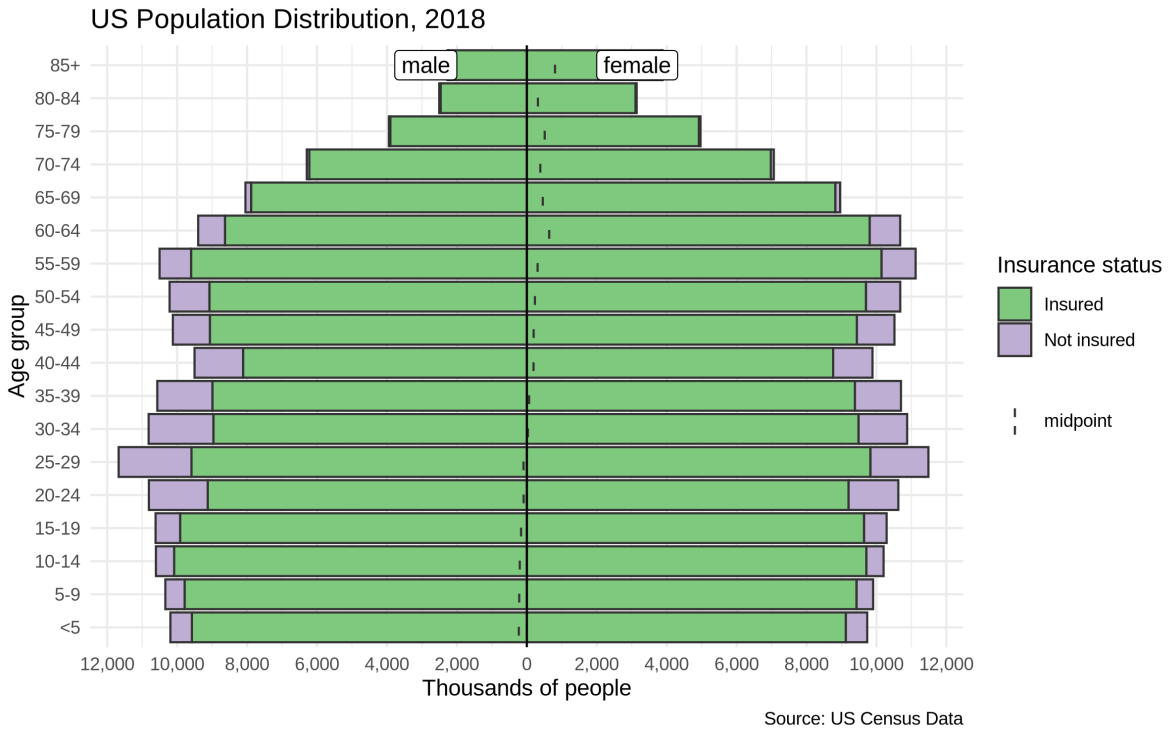
7.8 Adding A Stacked Split

The `apyramid` function can also split each side of the pyramid into stacked categories. The `split_by` argument still controls the two mirrored sides. The `stack_by` argument adds another grouping variable inside each side.

The `us_ins_2018` data set is stratified by gender and health insurance status. That makes it possible to show age, gender, and insurance status in one compact figure.

```
p_ins <- age_pyramid(
  data = us_ins_2018,
  age_group = age,
  split_by = gender,
  stack_by = insured,
  count = count
)

p_ins +
  us_labels +
  labs(
```

```
    fill = "Insurance status"
  ) +
  theme_minimal()
```



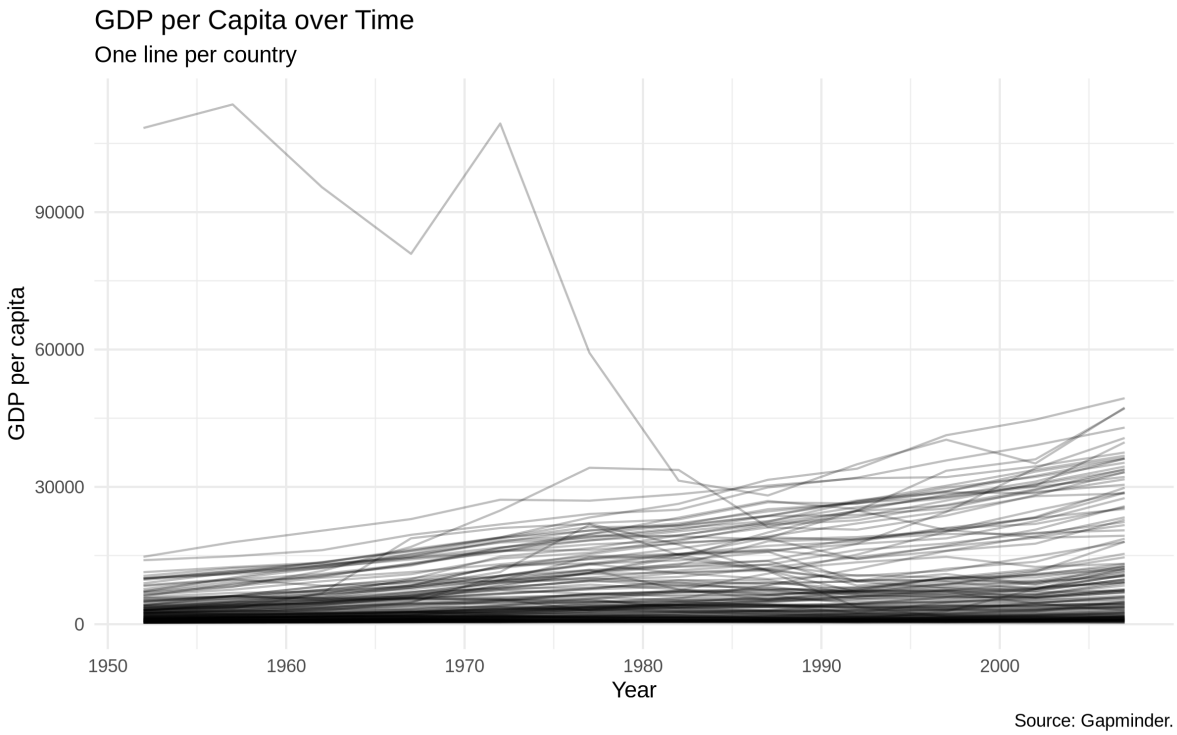
8 Lines, Bars, and Annotation

8.1 Line charts

Line charts are designed for ordered data, especially time. They work best when the x-axis has a meaningful sequence and the line connects observations that should be read as a path.

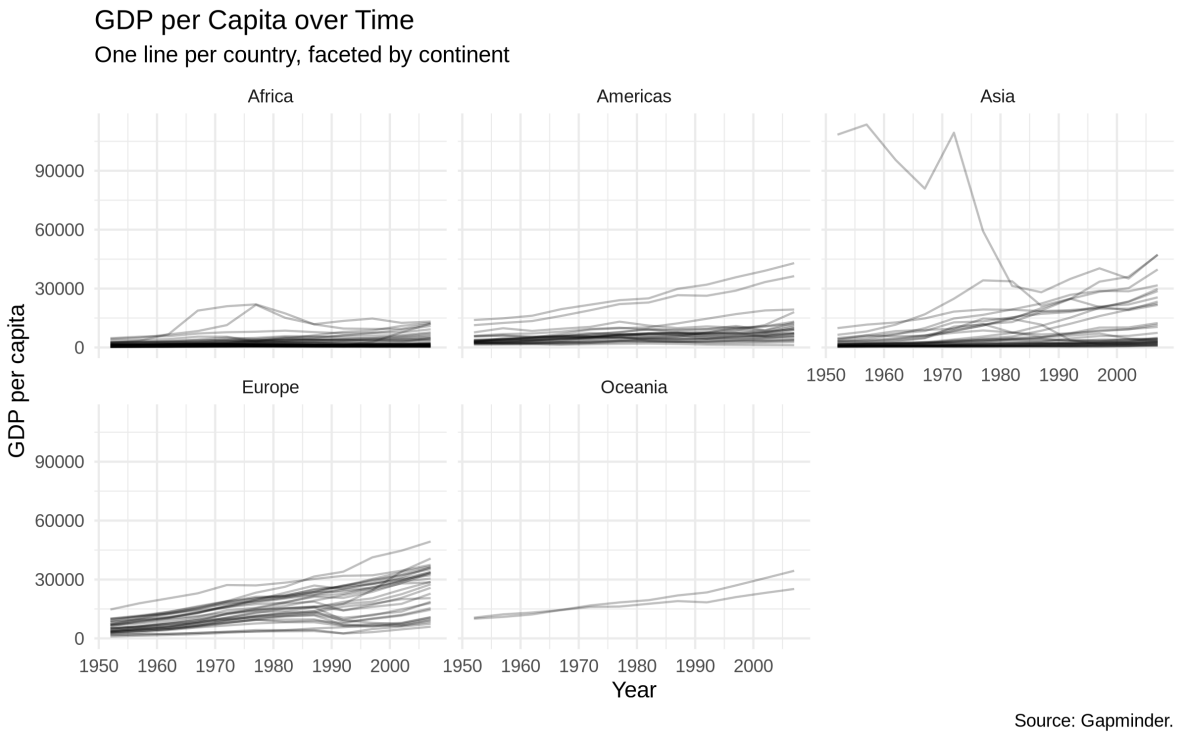
We will begin with GDP per capita over time in Gapminder.

```
ggplot(data = gapminder, mapping = aes(x = year, y = gdpPercap, group = country)) +
  geom_line(alpha = 0.25) +
  labs(
    title = "GDP per Capita over Time",
    subtitle = "One line per country",
    x = "Year",
    y = "GDP per capita",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



This plot has too many lines in one panel. We can separate continents with facets.

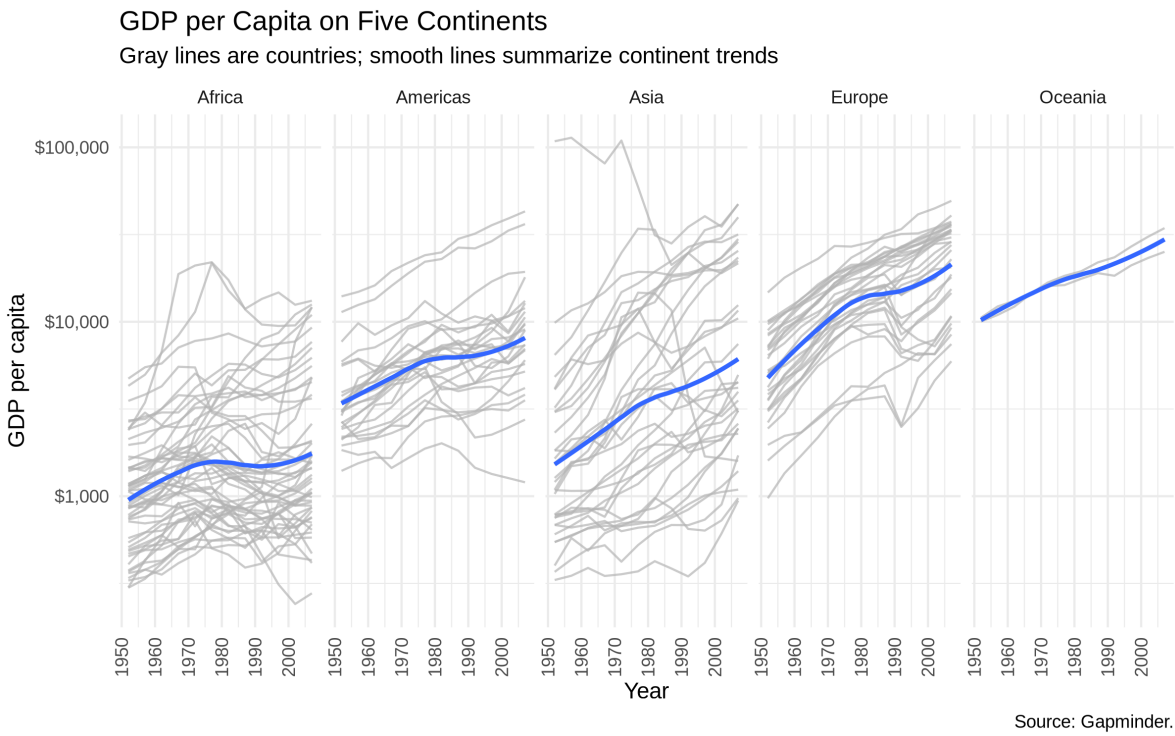
```
ggplot(data = gapminder, mapping = aes(x = year, y = gdpPercap, group = country)) +
  geom_line(alpha = 0.25) +
  facet_wrap(~ continent) +
  labs(
    title = "GDP per Capita over Time",
    subtitle = "One line per country, faceted by continent",
    x = "Year",
    y = "GDP per capita",
    caption = "Source: Gapminder."
  ) +
  theme_minimal()
```



Kuwait is an extreme case in this dataset. A log scale makes the vertical comparison easier.

```
ggplot(data = gapminder, mapping = aes(x = year, y = gdpPercap, group = country)) +
  geom_line(color = "gray70", alpha = 0.7) +
  geom_smooth(
    mapping = aes(group = continent),
    method = "loess",
    formula = y ~ x,
    se = FALSE,
    linewidth = 1
  ) +
  scale_y_log10(labels = dollar_format(accuracy = 1)) +
  facet_wrap(~ continent, ncol = 5) +
  labs(
    title = "GDP per Capita on Five Continents",
    subtitle = "Gray lines are countries; smooth lines summarize continent trends",
    x = "Year",
    y = "GDP per capita",
    caption = "Source: Gapminder."
  ) +
  theme_minimal() +
```

```
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5))
```



8.2 Working with dates

Some datasets store time as actual dates. Others store years as numbers. The `lubridate` package helps convert text or numeric values into date objects.

```
make_date(2026)
```

```
[1] "2026-01-01"
```

```
ymd(c("2009-01-02", "2009 01 03", "Created on 2009 1 6"))
```

```
[1] "2009-01-02" "2009-01-03" "2009-01-06"
```

The built-in `economics` dataset has monthly dates.

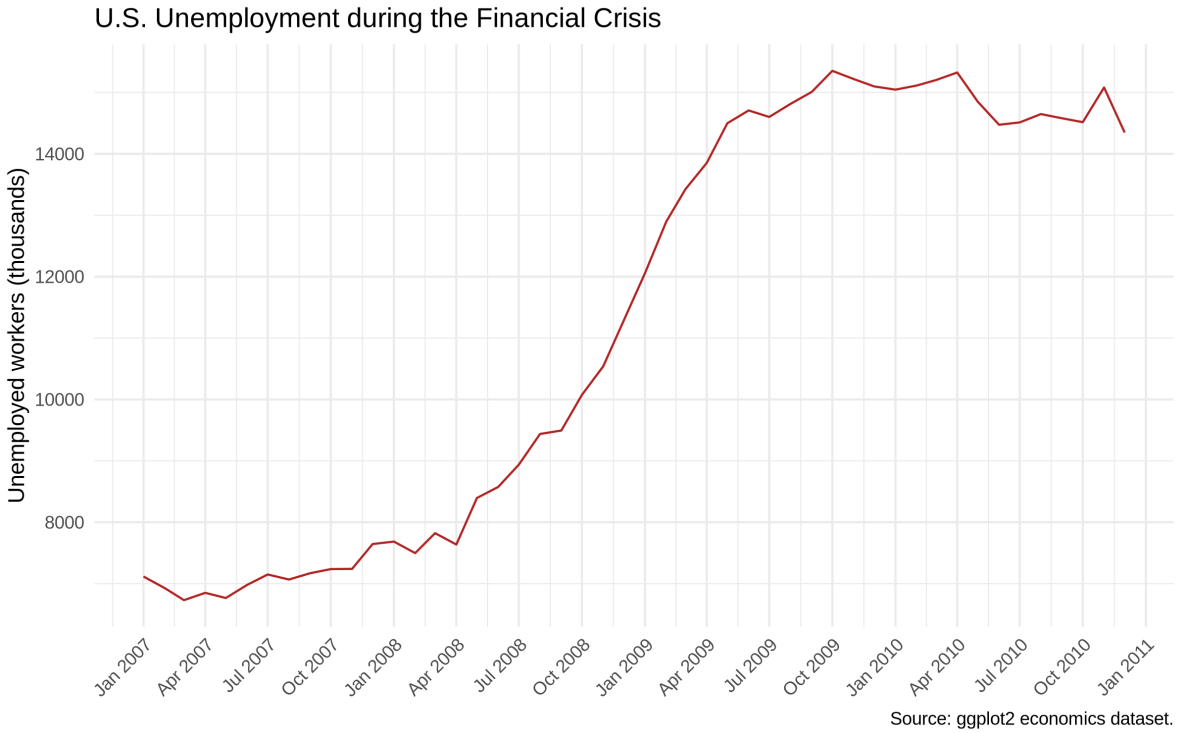
```r
economics |>
  select(date, unemploy, psavert) |>
  slice_head(n = 10)
```

```
# A tibble: 10 x 3
   date       unemploy psavert
   <date>        <dbl>   <dbl>
 1 1967-07-01     2944    12.6
 2 1967-08-01     2945    12.6
 3 1967-09-01     2958    11.9
 4 1967-10-01     3143    12.9
 5 1967-11-01     3066    12.8
 6 1967-12-01     3018    11.8
 7 1968-01-01     2878    11.7
 8 1968-02-01     3001    12.3
 9 1968-03-01     2877    11.7
10 1968-04-01     2709    12.3
```

```r
ggplot(data = economics, mapping = aes(x = date, y = unemploy)) +
  geom_line(color = "steelblue") +
  scale_x_date(date_breaks = "5 years", date_labels = "%Y") +
  labs(
    title = "U.S. Unemployment",
    subtitle = "Monthly observations, 1967-2015",
    x = NULL,
    y = "Unemployed workers (thousands)",
    caption = "Source: ggplot2 economics dataset."
  ) +
  theme_minimal()
```

For a shorter period, use denser date labels.

```
economics |>
  filter(date >= "2007-01-01", date <= "2010-12-31") |>
  ggplot(mapping = aes(x = date, y = unemploy)) +
  geom_line(color = "firebrick") +
  scale_x_date(date_breaks = "3 months", date_labels = "%b %Y") +
  labs(
    title = "U.S. Unemployment during the Financial Crisis",
    x = NULL,
    y = "Unemployed workers (thousands)",
    caption = "Source: ggplot2 economics dataset."
  ) +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



8.3 Exercise: Dates And Lines

Make a line chart from the built-in `economics` dataset.

1. Filter to dates from 2000 through 2015.
2. Put `date` on the x-axis and `psavert` on the y-axis.
3. Use `geom_line()`.
4. Use `scale_x_date()` to show labels every two years.
5. Add a title and axis labels with `labs()`.

```
# Write your plot here.
```

8.4 Highlighting trends

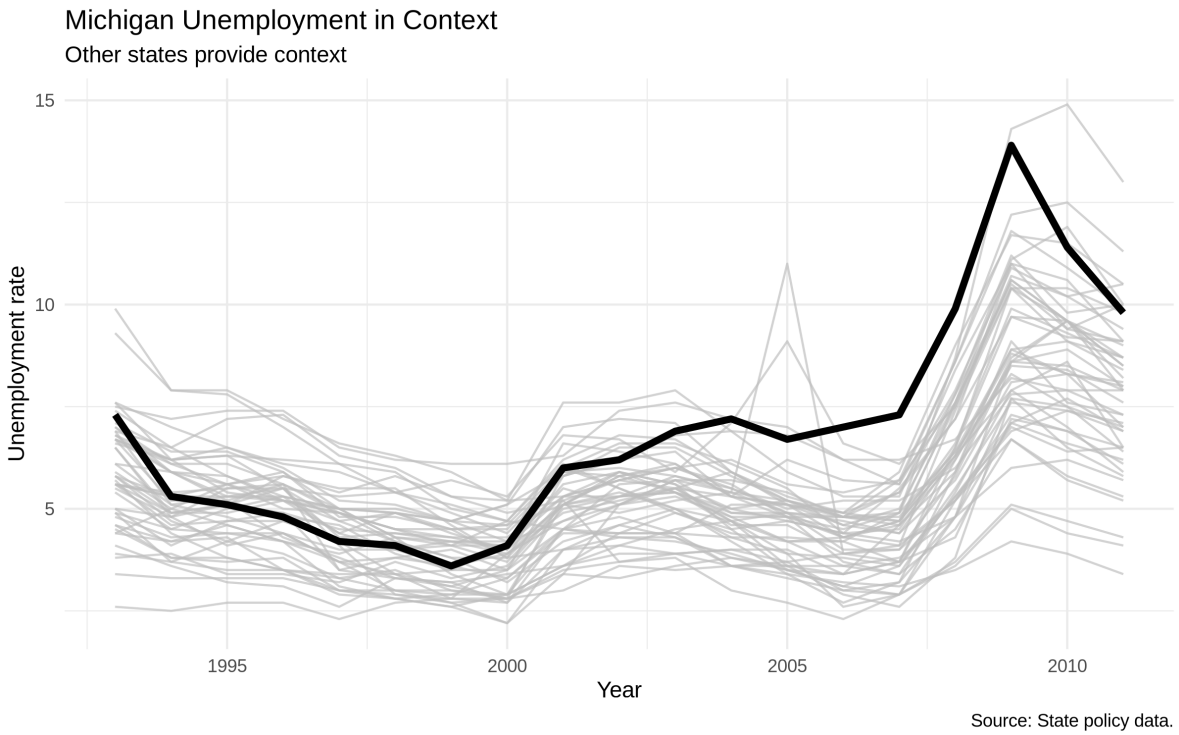
Line charts often need highlighting. A common strategy is to show all lines in gray and draw the important line in a stronger color.

The state policy dataset is useful for this because it has repeated state-year observations. The examples below use unemployment rates by state.

```r
state_policy_panel <- haven::read_dta("Data/state_policy/state_data_1993_2016.dta") |>
  mutate(st = as.character(st)) |>
  filter(!is.na(unemploy))

state_policy_panel |>
  select(st, year, unemploy) |>
  head(10)
```

```
# A tibble: 10 x 3
   st     year unemploy
   <chr> <dbl>    <dbl>
 1 AK     1993     7.5
 2 AK     1994     7.20
 3 AK     1995     7.40
 4 AK     1996     7.40
 5 AK     1997     6.5
 6 AK     1998     6.20
 7 AK     1999     6.10
 8 AK     2000     6.10
 9 AK     2001     6.30
10 AK     2002     7.40
```

```r
ggplot(data = state_policy_panel, mapping = aes(x = year, y = unemploy, group = st)) +
  geom_line(color = "gray75", alpha = 0.7) +
  geom_line(
    data = state_policy_panel |> filter(st == "MI"),
    color = "black",
    linewidth = 1.6
  ) +
  labs(
    title = "Michigan Unemployment in Context",
    subtitle = "Other states provide context",
    x = "Year",
    y = "Unemployment rate",
    caption = "Source: State policy data."
  ) +
  theme_minimal()
```

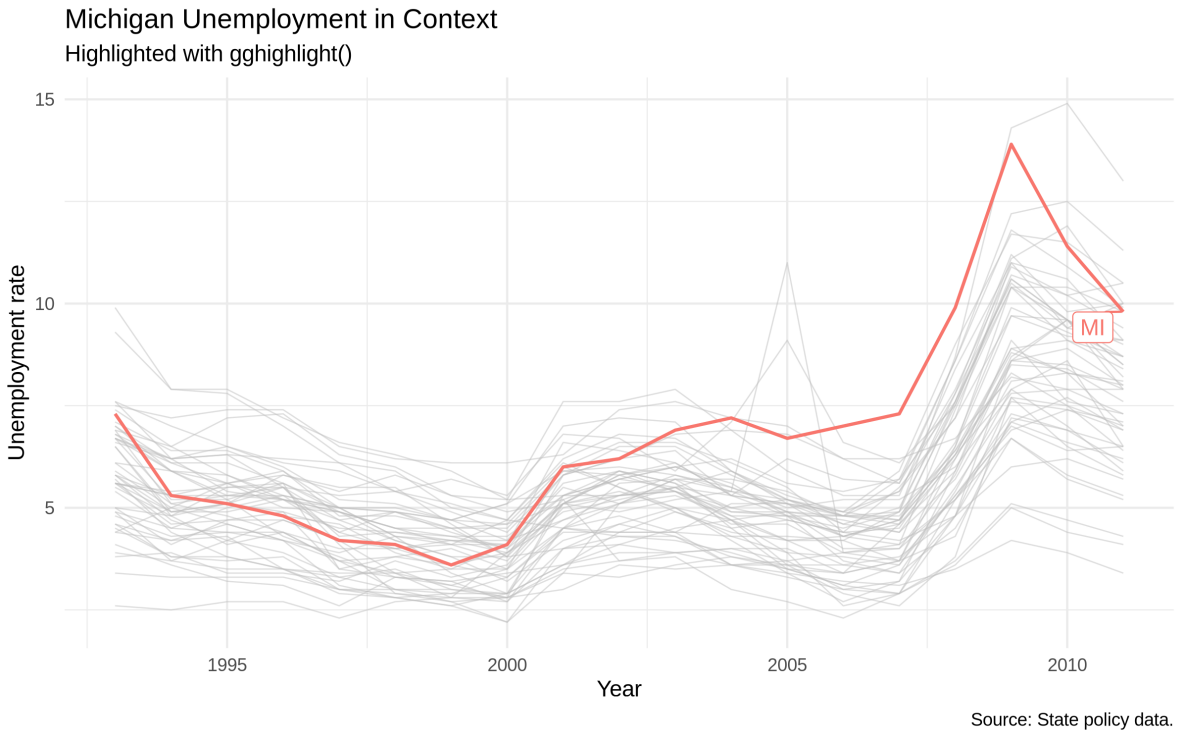


The plot above identifies Michigan visually but not by name. Chapter 12 returns to manual direct labels in a fuller workflow. Here, the shorter option is `gghighlight()`.

The `gghighlight` package automates highlighting and direct labeling when the highlighting rule can be stated as a logical condition.

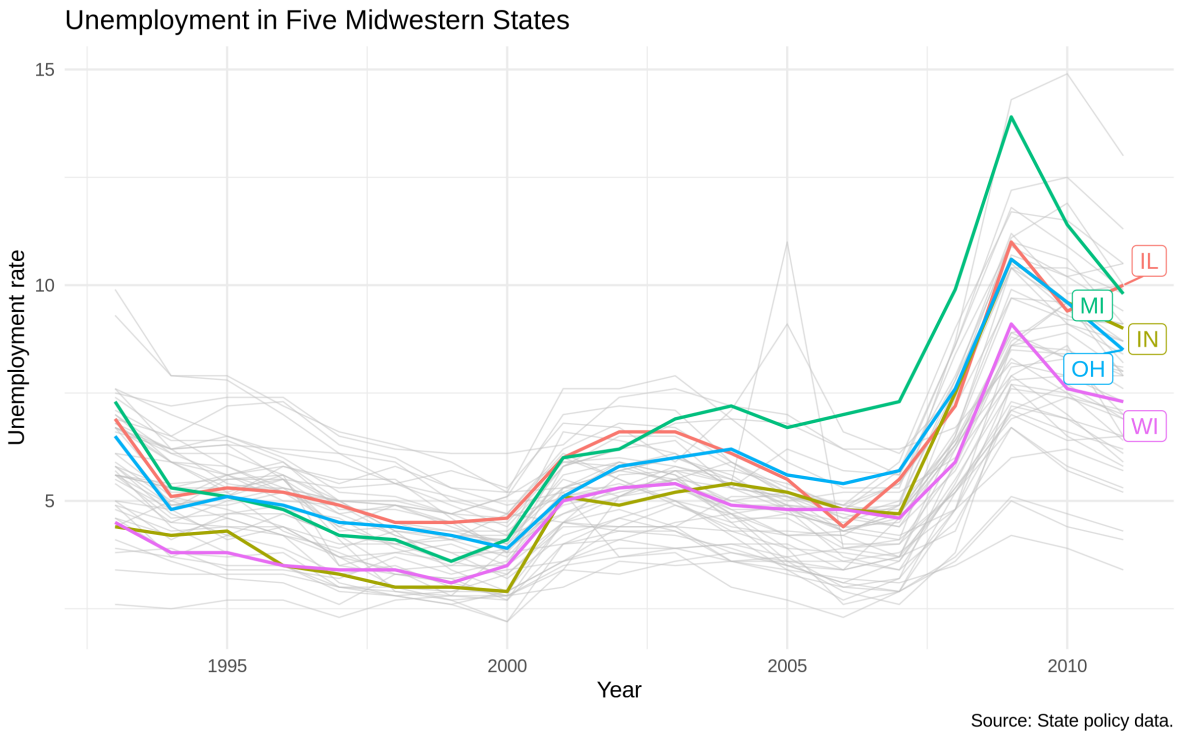
```
ggplot(data = state_policy_panel, mapping = aes(x = year, y = unemploy, color = st)) +
  geom_line(linewidth = 0.75) +
  gghighlight(
    st == "MI",
    use_group_by = FALSE,
    label_key = st,
    unhighlighted_params = list(linewidth = 0.3, alpha = 0.5)
  ) +
  labs(
    title = "Michigan Unemployment in Context",
    subtitle = "Highlighted with gghighlight()",
    x = "Year",
    y = "Unemployment rate",
    caption = "Source: State policy data."
  ) +
```

```
  theme_minimal() +
  theme(legend.position = "none")
```



We can also highlight a set of cases. The condition can name several states with `%in%`.

```
ggplot(data = state_policy_panel, mapping = aes(x = year, y = unemploy, color = st)) +
  geom_line(linewidth = 0.75) +
  gghighlight(
    st %in% c("IL", "IN", "MI", "OH", "WI"),
    use_group_by = FALSE,
    label_key = st,
    unhighlighted_params = list(linewidth = 0.3, alpha = 0.5)
  ) +
  labs(
    title = "Unemployment in Five Midwestern States",
    x = "Year",
    y = "Unemployment rate",
    caption = "Source: State policy data."
  ) +
  theme_minimal() +
  theme(legend.position = "none")
```



8.5 Bar charts

Bar charts are useful for comparing quantities across categories. When the data is already summarized, use `geom_col()`.

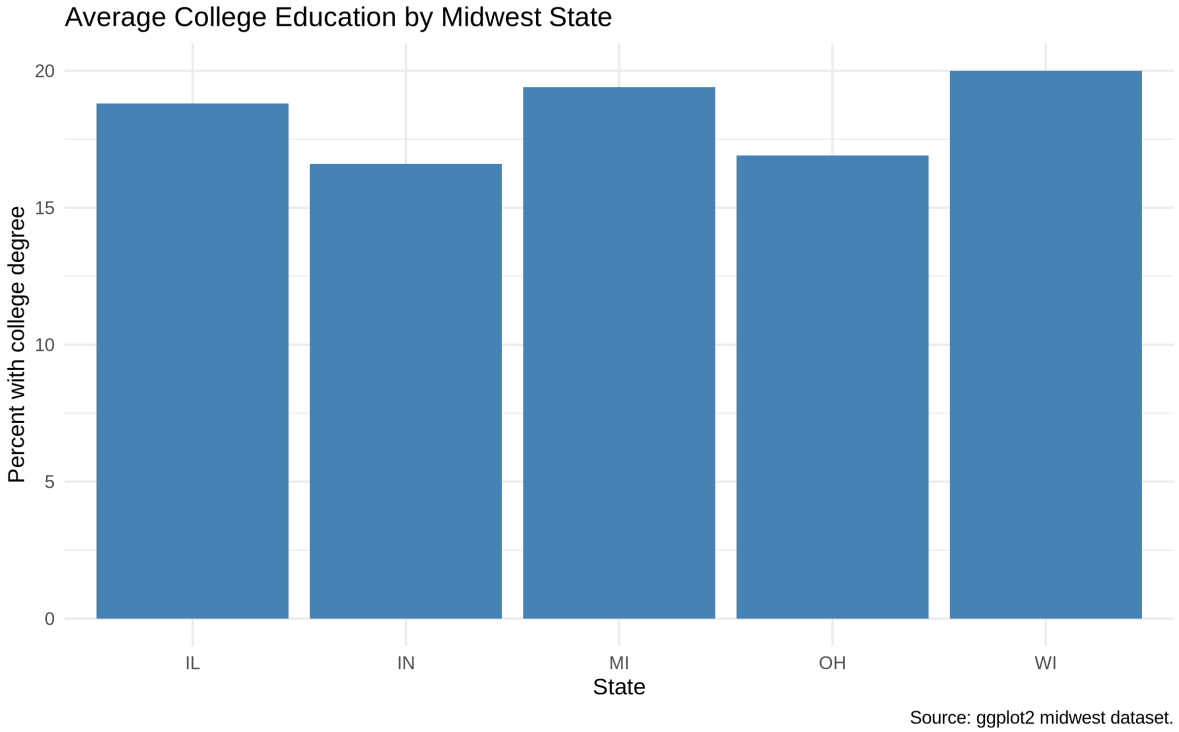
The `midwest` dataset has county-level observations. To compare average college education by state, we first summarize.

```
college_by_state <- midwest |>
  filter(!is.na(percollege)) |>
  group_by(state) |>
  summarize(
    avg_college = round(mean(percollege), 1)
  )

college_by_state
```

```
# A tibble: 5 x 2
  state avg_college
```

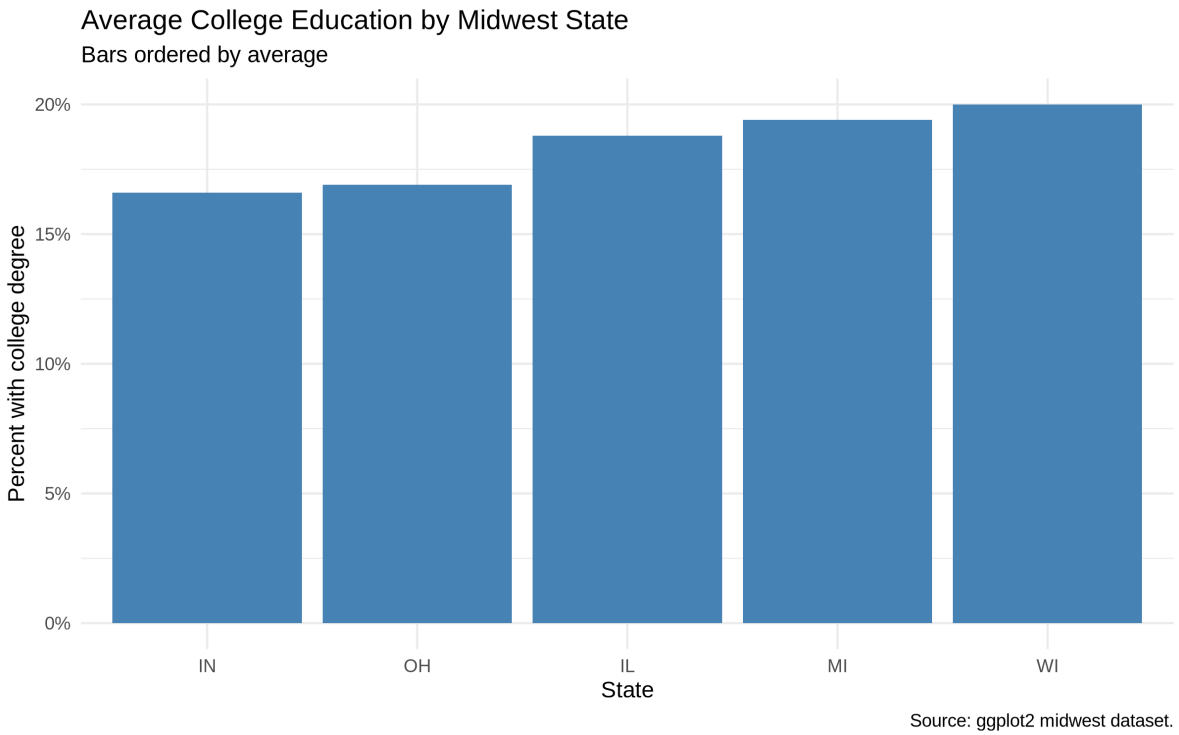
```
  <chr>       <dbl>
1 IL           18.8
2 IN           16.6
3 MI           19.4
4 OH           16.9
5 WI           20
```

```
ggplot(data = college_by_state, mapping = aes(x = state, y = avg_college)) +
  geom_col(fill = "steelblue") +
  labs(
    title = "Average College Education by Midwest State",
    x = "State",
    y = "Percent with college degree",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



Ordering the bars by value makes comparison easier.

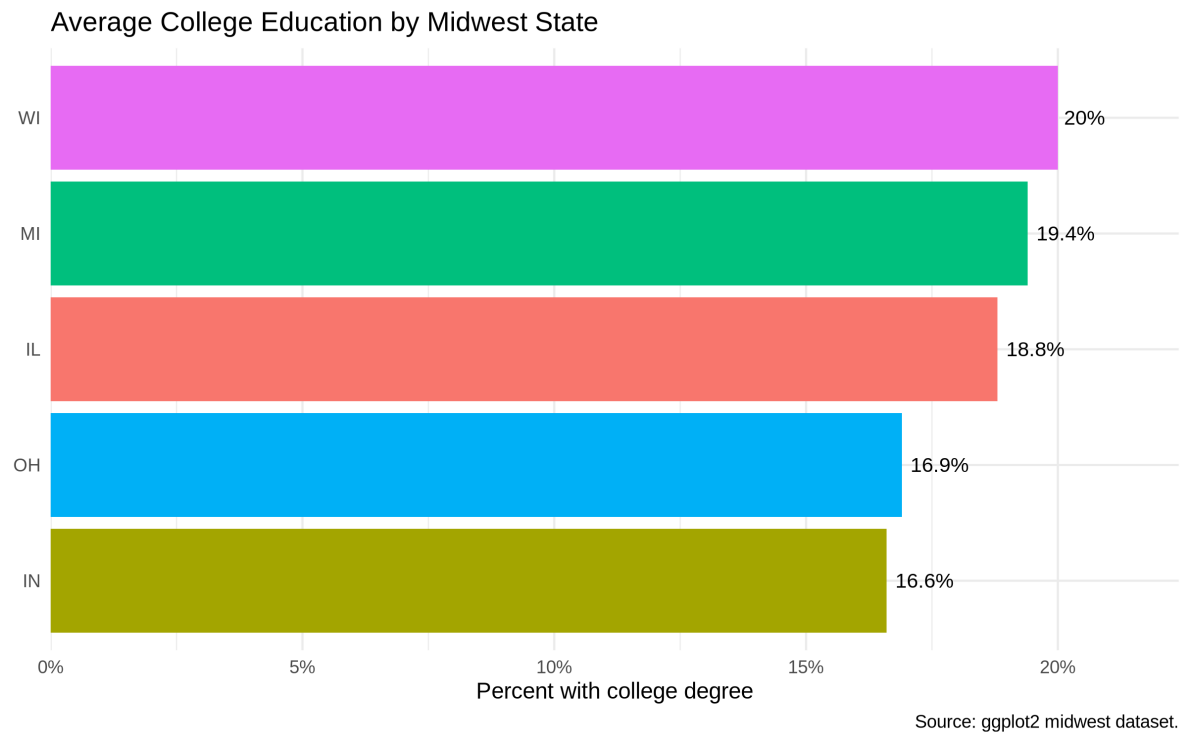
```r
ggplot(data = college_by_state, mapping = aes(x = reorder(state, avg_college), y = avg_colleg
  geom_col(fill = "steelblue") +
  scale_y_continuous(labels = percent_format(scale = 1)) +
  labs(
    title = "Average College Education by Midwest State",
    subtitle = "Bars ordered by average",
    x = "State",
    y = "Percent with college degree",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



If category labels are long, `coord_flip()` often helps.

```r
ggplot(data = college_by_state, mapping = aes(x = reorder(state, avg_college), y = avg_colleg
  geom_col(aes(fill = state), show.legend = FALSE) +
  geom_text(aes(label = str_c(avg_college, "%")), hjust = -0.15, size = 3.5) +
  coord_flip() +
  scale_y_continuous(labels = percent_format(scale = 1), expand = expansion(mult = c(0, 0.12
  labs(
    title = "Average College Education by Midwest State",
```

```
    x = NULL,
    y = "Percent with college degree",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



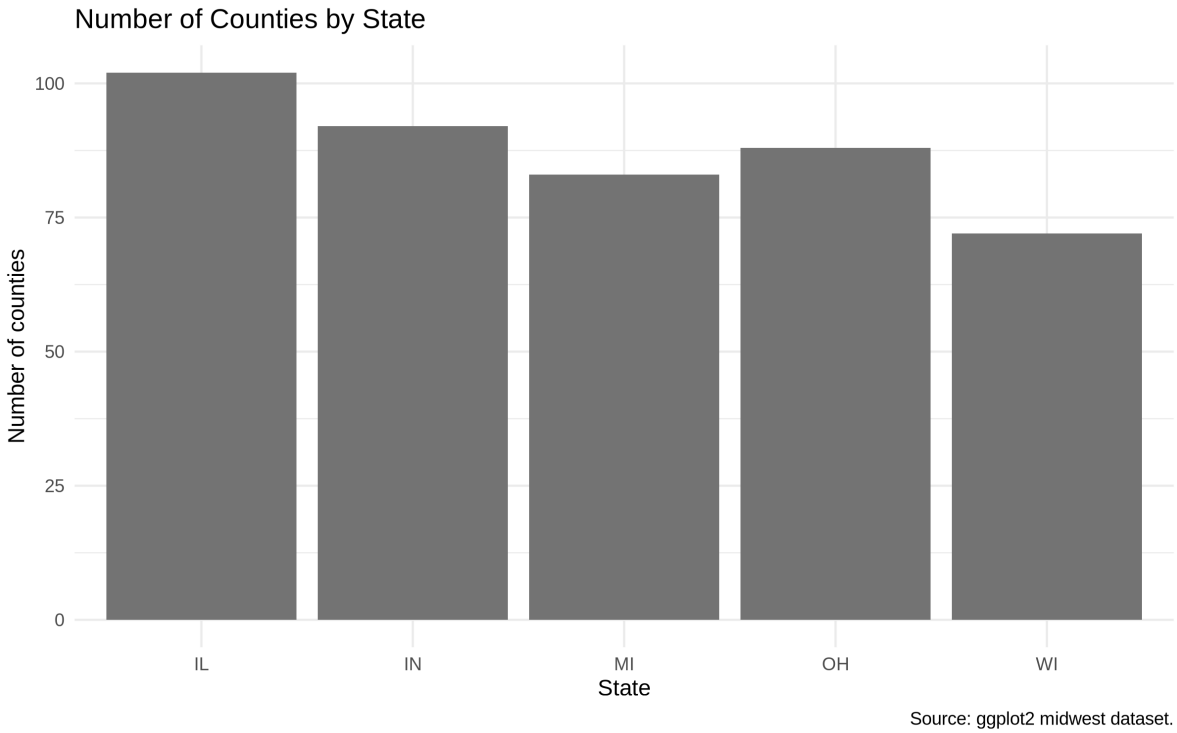
8.6 `geom_col()` versus `geom_bar()`

Use `geom_col()` when you already have the heights of the bars.

Use `geom_bar()` when you want `ggplot2` to count rows.

```
ggplot(data = midwest, mapping = aes(x = state)) +
  geom_bar(fill = "gray45") +
  labs(
    title = "Number of Counties by State",
    x = "State",
    y = "Number of counties",
    caption = "Source: ggplot2 midwest dataset."
```

```
  ) +
  theme_minimal()
```



Here, no y variable is mapped. `geom_bar()` counts the number of rows in each state.

8.7 Bar Label Placement

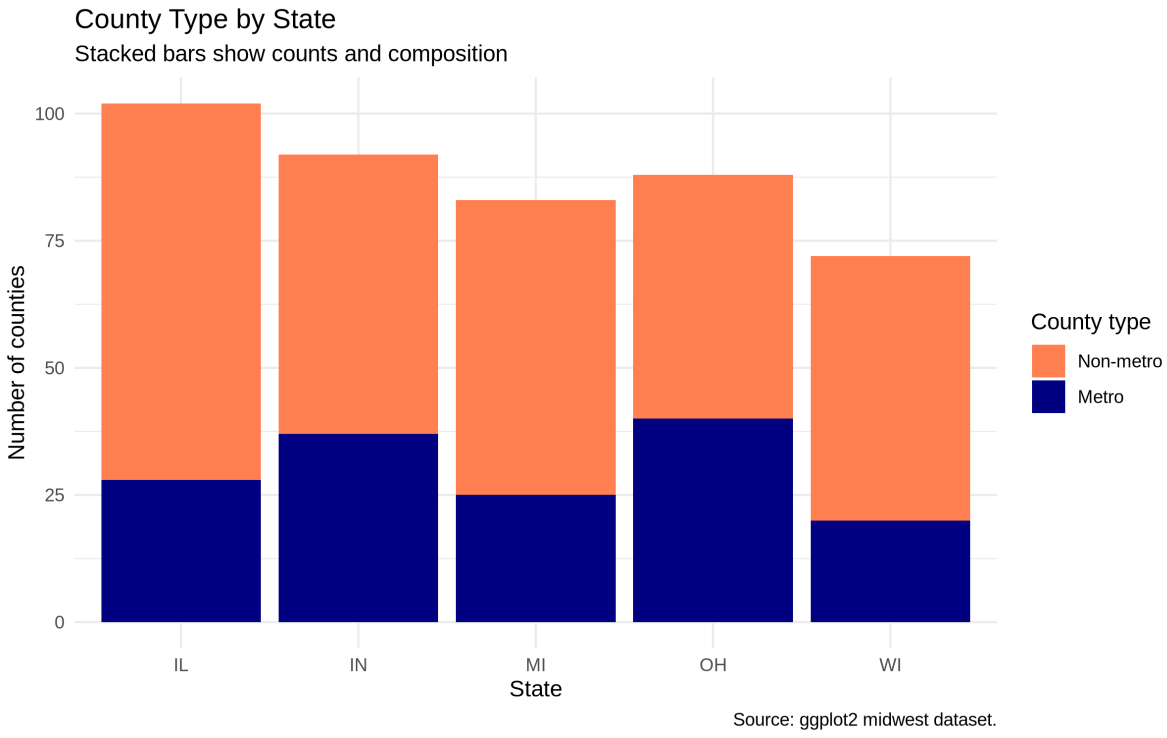
Text labels on bars need small position adjustments. For vertical bars, `vjust` moves labels up and down. For horizontal bars made with `coord_flip()`, `hjust` moves labels left and right after the coordinates are flipped.

In the ordered `geom_col()` chart above, `hjust = -0.15` places the label just outside the end of each bar after the coordinates are flipped. The `expand` argument in `scale_y_continuous()` adds extra room so the labels are not cut off. Use outside labels when exact values matter. If the bars are short or the panel is crowded, labels inside bars or a table may be cleaner.

8.8 Stacked And Proportional Bars

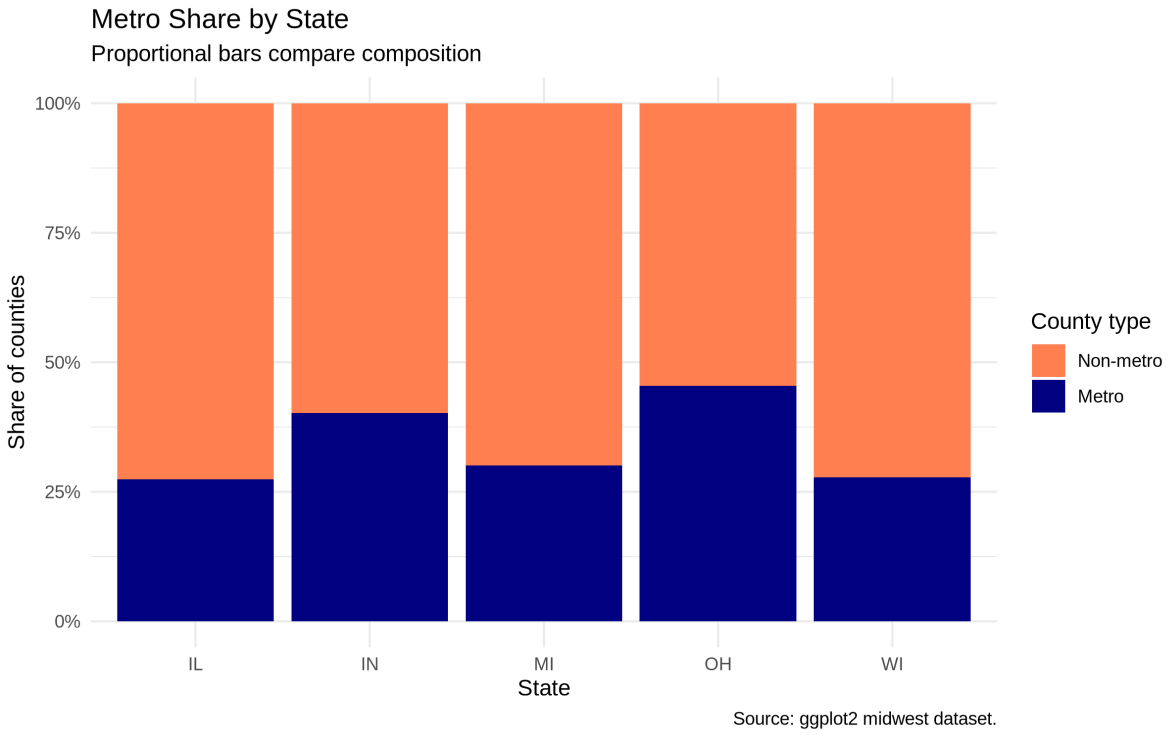
When a second categorical variable is mapped to `fill`, bars are stacked by default. Stacked bars show both the total height and the composition inside each category.

```
ggplot(data = midwest, mapping = aes(x = state, fill = factor(inmetro))) +
  geom_bar() +
  scale_fill_manual(
    values = c("0" = "coral", "1" = "navy"),
    labels = c("0" = "Non-metro", "1" = "Metro")
  ) +
  labs(
    title = "County Type by State",
    subtitle = "Stacked bars show counts and composition",
    x = "State",
    y = "Number of counties",
    fill = "County type",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



If the comparison is about composition rather than totals, `position = "fill"` makes each bar the same height and shows proportions.

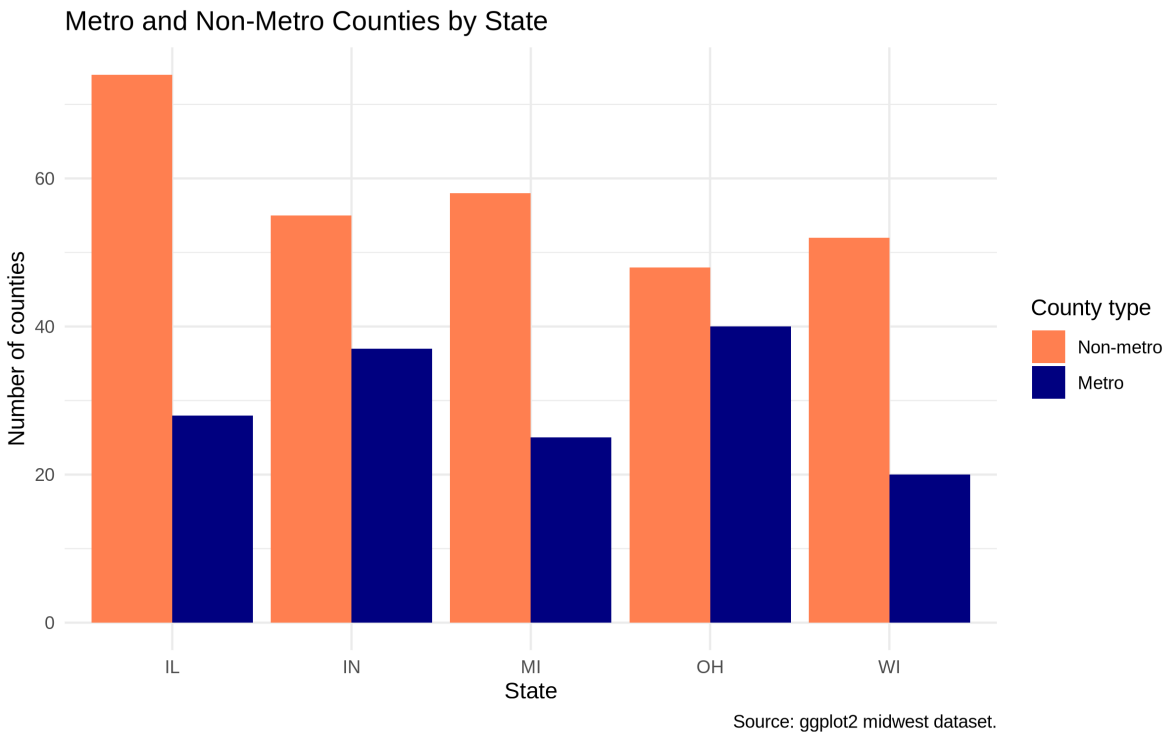
```
ggplot(data = midwest, mapping = aes(x = state, fill = factor(inmetro))) +
  geom_bar(position = "fill") +
  scale_y_continuous(labels = percent_format()) +
  scale_fill_manual(
    values = c("0" = "coral", "1" = "navy"),
    labels = c("0" = "Non-metro", "1" = "Metro")
  ) +
  labs(
    title = "Metro Share by State",
    subtitle = "Proportional bars compare composition",
    x = "State",
    y = "Share of counties",
    fill = "County type",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



8.9 Dodged bars

Dodged bars place bars side by side within each category. Here we compare metro and non-metro counties within states.

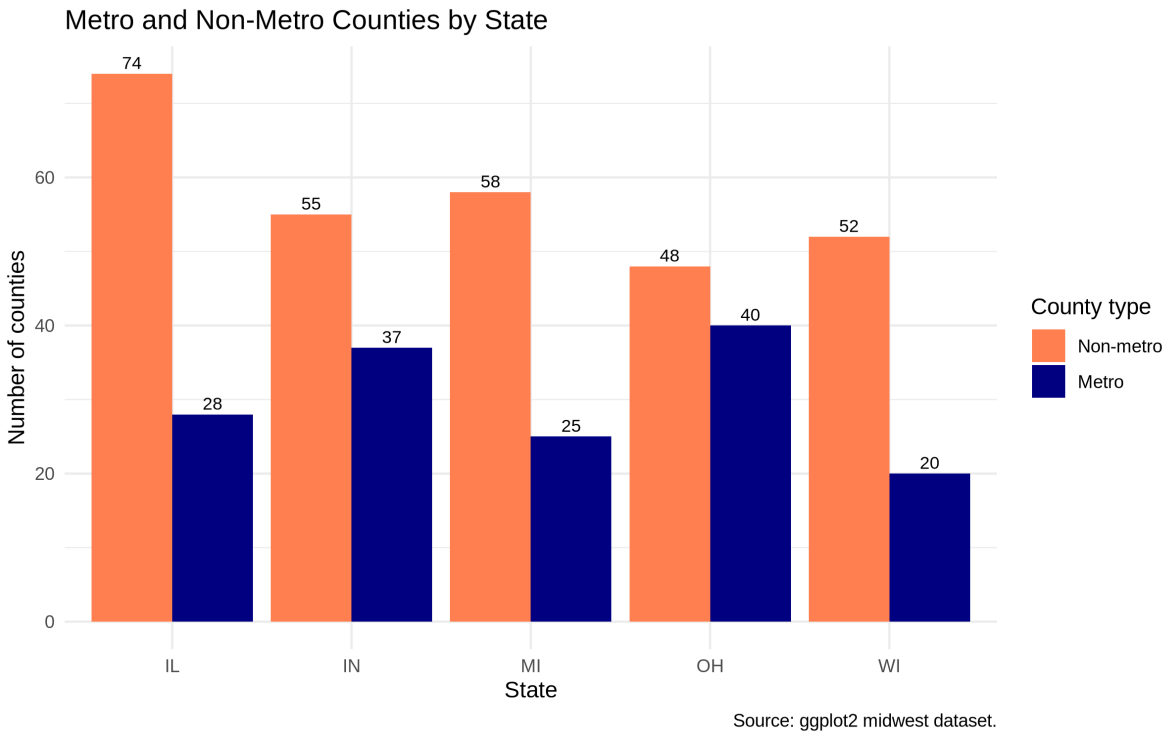
```
ggplot(data = midwest, mapping = aes(x = state, fill = factor(inmetro))) +
  geom_bar(position = "dodge") +
  scale_fill_manual(
    values = c("0" = "coral", "1" = "navy"),
    labels = c("0" = "Non-metro", "1" = "Metro")
  ) +
  labs(
    title = "Metro and Non-Metro Counties by State",
    x = "State",
    y = "Number of counties",
    fill = "County type",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



Adding labels to dodged bars requires the text to use the same dodging.

The important detail is that the bars and the text both use `position_dodge(width = 0.9)`. If the widths differ, the labels will not line up with the bars. The `group = factor(inmetro)` mapping tells the text layer which bars belong side by side.

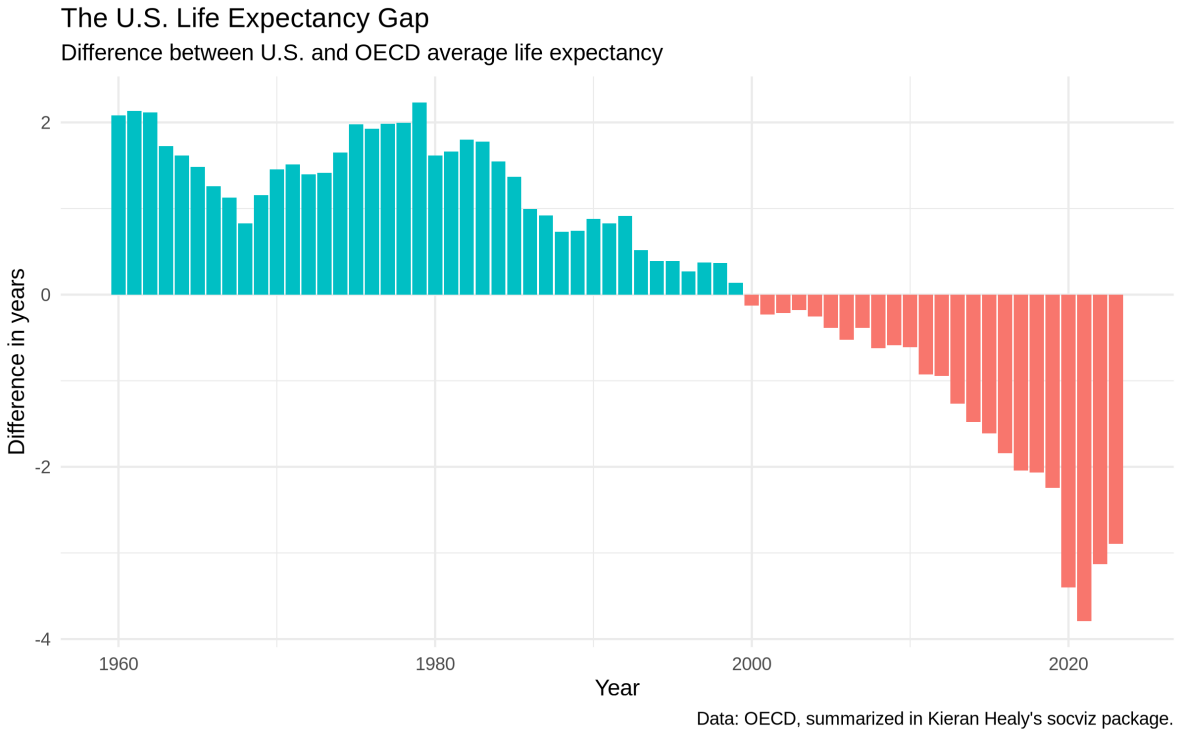
```
ggplot(data = midwest, mapping = aes(x = state, fill = factor(inmetro))) +
  geom_bar(position = position_dodge(width = 0.9)) +
  geom_text(
    mapping = aes(label = after_stat(count), group = factor(inmetro)),
    stat = "count",
    position = position_dodge(width = 0.9),
    vjust = -0.4,
    size = 3
  ) +
  scale_fill_manual(
    values = c("0" = "coral", "1" = "navy"),
    labels = c("0" = "Non-metro", "1" = "Metro")
  ) +
  labs(
    title = "Metro and Non-Metro Counties by State",
    x = "State",
    y = "Number of counties",
    fill = "County type",
    caption = "Source: ggplot2 midwest dataset."
  ) +
  theme_minimal()
```



8.10 Bar chart example: OECD life expectancy gap

A local copy of the `oecd_sum` summary lives at `Data/oecd/oecd_sum.csv`. Each row is one year, with columns for U.S. life expectancy, the OECD non-U.S. average, the difference, and a `hi_lo` indicator for whether the U.S. is above or below the average.

```
oecd_sum <- read_csv("Data/oecd/oecd_sum.csv", show_col_types = FALSE)

ggplot(data = oecd_sum, mapping = aes(x = year, y = diff, fill = hi_lo)) +
  geom_col() +
  guides(fill = "none") +
  labs(
    title = "The U.S. Life Expectancy Gap",
    subtitle = "Difference between U.S. and OECD average life expectancy",
    x = "Year",
    y = "Difference in years",
    caption = "Data: OECD, summarized in Kieran Healy's socviz package."
  ) +
  theme_minimal()
```



This is a bar chart because the main comparison is the size and direction of a yearly difference.

8.11 Exercise

Create an ordered bar chart from a summarized dataset.

1. Use `midwest`.
2. Group by `state`.
3. Calculate the median of `percollege`.
4. Make an ordered `geom_col()` chart.
5. Add value labels.
6. Remove any redundant legend.

```
# Write your plot here.
```

8.12 Extra: slopegraphs

Slopegraphs compare the same units at a small number of time points. They are especially useful when the start and end values matter more than the intermediate path. The form was

popularized by Edward Tufte in *The Visual Display of Quantitative Information*, but the design predates him by a century.

A generic Tufte-style slopegraph:

Ben Fry's slopegraph of baseball salary per win:

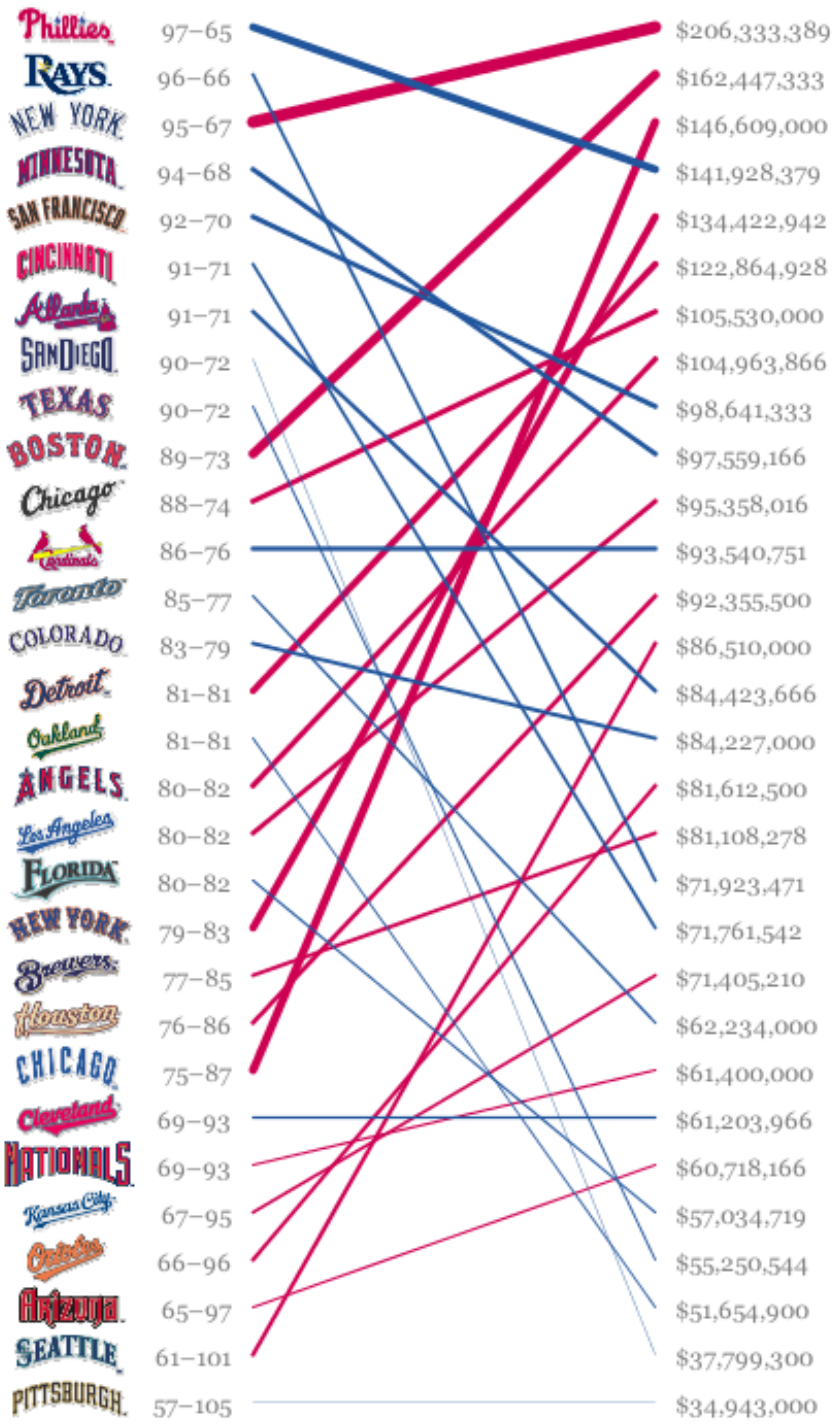


Figure 8.1: Baseball slopegraph by Ben Fry

And one from *Scribner's Statistical Atlas of the United States* in 1883, well before the modern term existed:

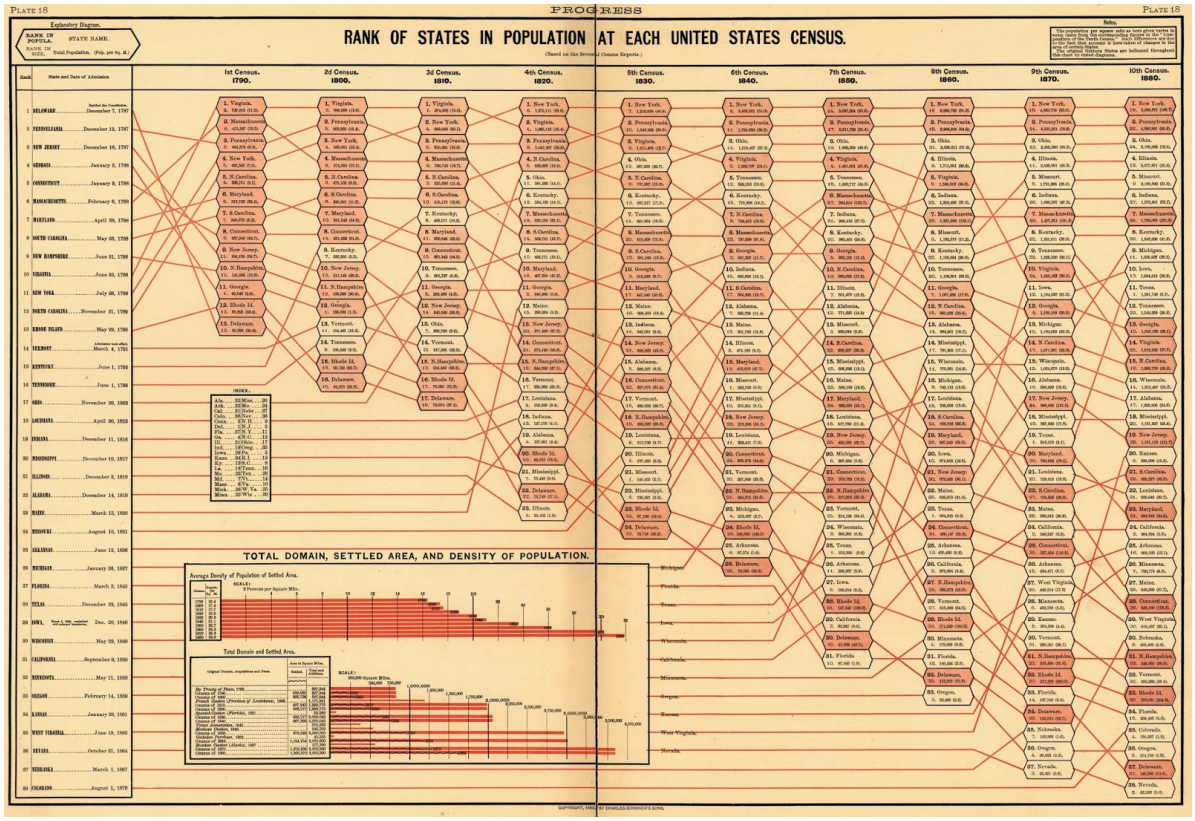


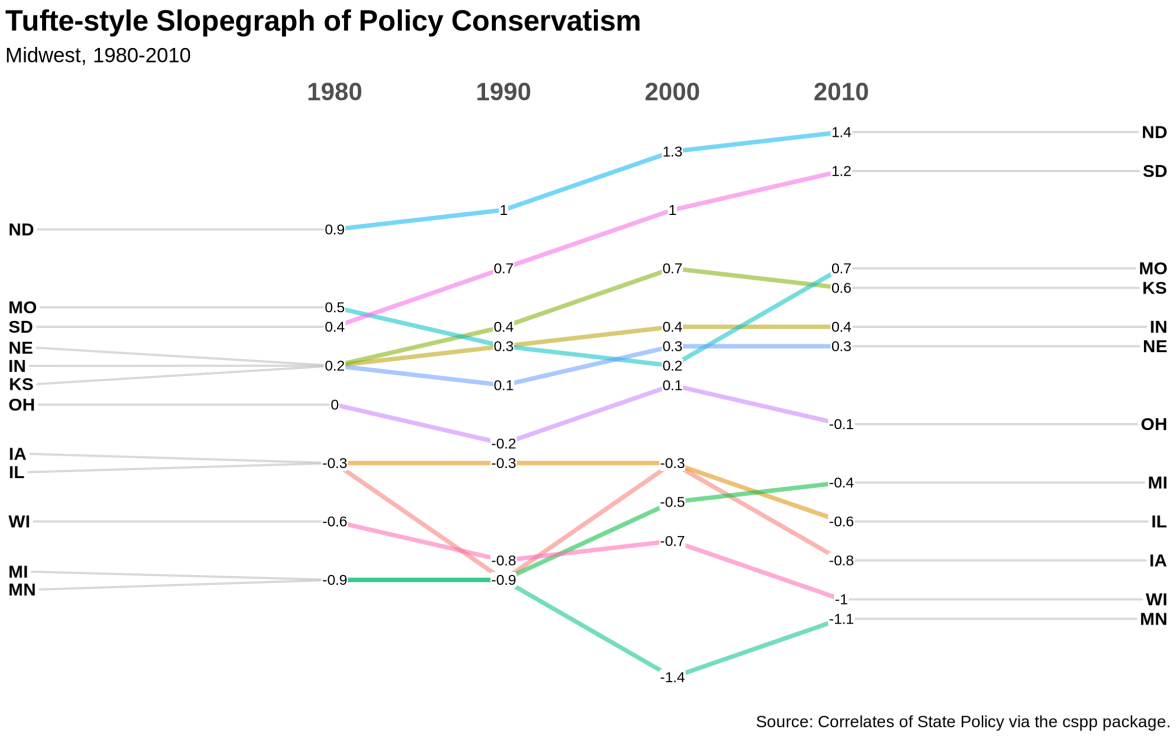
Figure 8.2: Scribner's Statistical Atlas, 1883

Unlike a traditional line chart, a slopegraph emphasizes the start and end values of each unit, so change is read as the slope of a line rather than as a path through many points.

The `states` table is a local snapshot of Correlates of State Policy data, originally fetched with `cspp::get_cspp_data()`. `polconserv` is policy conservatism (the negation of `pollib_median`), so larger values mean a more conservative policy environment.

```
states <- read_csv("Data/state_policy/cspp_states.csv", show_col_types = FALSE)

library(ggslopegraph)

ggslopegraph(
  dataframe = states |>
    mutate(year = as.character(year)) |>
```

```
    filter(year %in% c("1980", "1990", "2000", "2010"), region.name == "midwest"),
  Times = year,
  Measurement = polconserv,
  Grouping = st,
  Title = "Tufte-style Slopegraph of Policy Conservatism",
  SubTitle = "Midwest, 1980-2010",
  Caption = "Source: Correlates of State Policy via the cspp package."
)
```



`ggslopegraph()` requires the `Times` column to be a character or factor, which is why `year` is converted before filtering. If `ggslopegraph` is not available, the same plot can be built with plain `geom_line()` and `geom_text()`.

9 Tables

Counts, summaries, publication tables, and export

Tables are part of a visualization workflow. A plot is often the fastest way to show a pattern, but a table is often the clearest way to show exact values, counts, summaries, and compact descriptions of a dataset.

Table or plot? Use a plot when the reader needs to grasp a shape — a trend, a distribution, a cluster. Use a table when the reader needs to look up a specific number, compare exact values across categories, or read a ranked list. The two formats are complements, not competitors.

This chapter starts with quick exploratory tables and then moves toward tables that belong in reports, slides, papers, and appendices.

9.1 Which tool for which table

R has several strong table packages. A quick orientation before diving in:

- **Base R** `table()` — quick Console frequency counts; no dependencies.
- `janitor::tabyl()` — counts, percentages, totals, and crosstabs in tidy pipelines.
- `tinytable` — compact publication-style display; works across HTML, PDF, and Word.
- `modelsummary` — descriptive summaries and formatted crosstabs.
- `gt` — highly designed HTML tables with fine-grained cell control.
- `flextable` — Word output as the primary target.
- `DT` — interactive search and pagination in HTML output.

You do not need all of them. The choice depends on what kind of table you need: quick exploration, descriptive summary, publication display, or interactive browsing.

```
library(tidyverse)
library(scales)
library(janitor)
library(tinytable)
```

```
library(modelsummary)
library(corrr)
```

9.2 A first frequency table

Base R has a built-in function called `table()`. It counts how often each value appears.

```
table(mtcars$cyl)
```

```
 4  6  8
11  7 14
```

This table counts the number of cars in `mtcars` with 4, 6, and 8 cylinders.

A two-way table counts combinations of two variables.

```
table(mtcars$cyl, mtcars$gear)
```

```
   3  4  5
4  1  8  2
6  2  4  1
8 12  0  2
```

The output is useful, but the labels are not very descriptive. We can improve the row and column labels before printing the table.

```
cylinders_gear <- table(mtcars$cyl, mtcars$gear)

dimnames(cylinders_gear) <- list(
  "Cylinders" = paste(rownames(cylinders_gear), "cyl"),
  "Gears" = paste(colnames(cylinders_gear), "gears")
)

cylinders_gear
```

```
         Gears
Cylinders 3 gears 4 gears 5 gears
    4 cyl       1       8       2
    6 cyl       2       4       1
    8 cyl      12       0       2
```

Base R tables are dependable and fast. They are especially useful when you are exploring data in the Console. For tables that will appear in a report, other packages give us cleaner formatting and easier percentages.

9.3 Cleaner crosstabs with `janitor`

The `janitor` package has a function called `tabyl()`. It behaves like a modern version of `table()`: it counts values, keeps the output as a data frame, and works well in tidyverse pipelines.

We will use a local copy of the Gapminder data.

The function `glimpse()` comes from `dplyr`, which is loaded as part of the tidyverse.

```
gapminder <- readr::read_csv("Data/gapminder/gapminder.csv")

glimpse(gapminder)
```

```
Rows: 1,704
Columns: 6
$ country   <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan", ~
$ continent <chr> "Asia", "Asia", "Asia", "Asia", "Asia", "Asia", "Asia", "Asi~
$ year      <dbl> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, ~
$ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40.8~
$ pop       <dbl> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, 12~
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134, ~
```

A one-variable `tabyl()` gives counts and proportions.

```
gapminder |>
  tabyl(continent)
```

```
 continent   n    percent
    Africa 624 0.36619718
  Americas 300 0.17605634
      Asia 396 0.23239437
    Europe 360 0.21126761
   Oceania  24 0.01408451
```

A two-variable `tabyl()` gives a crosstab.

```
gapminder |>
  tabyl(continent, year)
```

```
 continent 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
    Africa   52   52   52   52   52   52   52   52   52   52   52   52
  Americas   25   25   25   25   25   25   25   25   25   25   25   25
      Asia   33   33   33   33   33   33   33   33   33   33   33   33
    Europe   30   30   30   30   30   30   30   30   30   30   30   30
   Oceania    2    2    2    2    2    2    2    2    2    2    2    2
```

This table counts country-year observations by continent and year. A different kind of crosstab uses a category we create ourselves. Here we classify countries by whether life expectancy in 2007 was at least 70 years.

```
gapminder_2007 <- gapminder |>
  filter(year == 2007) |>
  mutate(
    life_expectancy_group = if_else(lifeExp >= 70, "70 years or more", "Under 70 years")
  )

gapminder_2007 |>
  tabyl(continent, life_expectancy_group)
```

```
 continent 70 years or more Under 70 years
    Africa                7             45
  Americas               22              3
      Asia               22             11
    Europe               30              0
   Oceania                2              0
```

The `adorn_*()` functions add totals, percentages, and count labels.

```
gapminder_2007 |>
  tabyl(continent, life_expectancy_group) |>
  adorn_totals(c("row", "col")) |>
  adorn_percentages("row") |>
  adorn_pct_formatting(digits = 1) |>
  adorn_ns()
```

```
 continent 70 years or more Under 70 years        Total
    Africa       13.5%  (7)     86.5% (45) 100.0%  (52)
  Americas       88.0% (22)     12.0%  (3) 100.0%  (25)
      Asia       66.7% (22)     33.3% (11) 100.0%  (33)
    Europe      100.0% (30)      0.0%  (0) 100.0%  (30)
   Oceania      100.0%  (2)      0.0%  (0) 100.0%   (2)
     Total       58.5% (83)     41.5% (59) 100.0                      (142)
```

Use row percentages when the row categories are your comparison groups. In this case, the question is: within each continent, what share of countries had life expectancy of at least 70 years?

Column percentages answer a different question.

```
gapminder_2007 |>
  tabyl(continent, life_expectancy_group) |>
  adorn_totals(c("row", "col")) |>
  adorn_percentages("col") |>
  adorn_pct_formatting(digits = 1) |>
  adorn_ns()
```

```
 continent 70 years or more Under 70 years        Total
    Africa        8.4%  (7)     76.3% (45)  36.6%  (52)
  Americas       26.5% (22)      5.1%  (3)  17.6%  (25)
      Asia       26.5% (22)     18.6% (11)  23.2%  (33)
    Europe       36.1% (30)      0.0%  (0)  21.1%  (30)
   Oceania        2.4%  (2)      0.0%  (0)   1.4%   (2)
     Total      100.0% (83)    100.0% (59) 100.0                      (142)
```

Here the question is: within each life expectancy group, what share of countries came from each continent?

`tabyl()` can extend to a third variable. The result is a list of two-way tables, one for each value of the third variable — useful for showing how a crosstab changes across time or groups.

```r
gapminder |>
  filter(year %in% c(1952, 2007)) |>
  mutate(
    life_expectancy_group = if_else(lifeExp >= 70, "70 years or more", "Under 70 years")
  ) |>
  tabyl(continent, life_expectancy_group, year)
```

```
$`1952`
 continent 70 years or more Under 70 years
    Africa                0             52
  Americas                0             25
      Asia                0             33
    Europe                5             25
   Oceania                0              2

$`2007`
 continent 70 years or more Under 70 years
    Africa                7             45
  Americas               22              3
      Asia               22             11
    Europe               30              0
   Oceania                2              0
```

If the three-way result feels dense, split it into separate tables or make a faceted plot instead.

9.4 From summarize to table

A common workflow is to compute a summary with `group_by()` and `summarize()` and then format the result as a display table. The two steps are separate: the first produces a data frame, the second formats it.

```r
continent_summary <- gapminder |>
  filter(year == 2007) |>
  group_by(continent) |>
  summarize(
    countries      = n(),
    median_life    = median(lifeExp),
    median_gdp     = median(gdpPercap)
  )
```

Table 9.1: Summary statistics by continent, 2007

Continent	Countries	Median life expectancy	Median GDP per capita
Africa	52	53	1452
Americas	25	73	8948
Asia	33	72	4471
Europe	30	79	28054
Oceania	2	81	29810

```
continent_summary
```

```
# A tibble: 5 x 4
  continent countries median_life median_gdp
  <chr>         <int>       <dbl>      <dbl>
1 Africa           52        52.9      1452.
2 Americas         25        72.9      8948.
3 Asia             33        72.4      4471.
4 Europe           30        78.6     28054.
5 Oceania           2        80.7     29810.
```

Pass the result directly to `tt()` for a formatted table. Rename the columns before calling `tt()` so the display names are readable.

```
continent_summary |>
  rename(
    "Continent"              = continent,
    "Countries"              = countries,
    "Median life expectancy" = median_life,
    "Median GDP per capita"  = median_gdp
  ) |>
  tt(
    caption = "Summary statistics by continent, 2007",
    digits  = 1
  )
```

Renaming columns in the data frame before calling `tt()` is the most reliable way to control display names across output formats.

9.5 Reshaping for tables

Long-format data is easy to compute on but often hard to read as a table. A summary grouped by two variables — say, continent and year — has one row per combination, which makes for a long, narrow table.

```
life_by_year <- gapminder |>
  filter(year %in% c(1957, 1977, 1997, 2007)) |>
  group_by(continent, year) |>
  summarize(median_life = median(lifeExp), .groups = "drop")

life_by_year
```

```
# A tibble: 20 x 3
   continent  year median_life
   <chr>     <dbl>       <dbl>
 1 Africa     1957        40.6
 2 Africa     1977        49.3
 3 Africa     1997        52.8
 4 Africa     2007        52.9
 5 Americas   1957        56.1
 6 Americas   1977        66.4
 7 Americas   1997        72.1
 8 Americas   2007        72.9
 9 Asia       1957        48.3
10 Asia       1977        60.8
11 Asia       1997        70.3
12 Asia       2007        72.4
13 Europe     1957        67.6
14 Europe     1977        72.3
15 Europe     1997        76.1
16 Europe     2007        78.6
17 Oceania    1957        70.3
18 Oceania    1977        72.9
19 Oceania    1997        78.2
20 Oceania    2007        80.7
```

The same continent appears in four rows. Pivot `year` into columns to put a continent's trajectory in one row.

Table 9.2: Median life expectancy by continent, selected years

Continent	1957	1977	1997	2007
Africa	41	49	53	53
Americas	56	66	72	73
Asia	48	61	70	72
Europe	68	72	76	79
Oceania	70	73	78	81

```
life_by_year |>
  pivot_wider(names_from = year, values_from = median_life) |>
  rename("Continent" = continent) |>
  tt(
    caption = "Median life expectancy by continent, selected years",
    digits  = 1
  )
```

Long format is right for computation and ggplot; wide format is often right for display.

9.6 Formatting numbers

Raw R numbers — `1234.5678`, `0.043`, `1.23e+09` — are often not the right form for a published table. The `scales` package provides helpers that turn numbers into currency, percentages, comma-separated thousands, and other common formats. Apply them with `mutate()` before passing the data frame to a display function.

```
continent_summary |>
  mutate(
    countries   = comma(countries),
    median_life = number(median_life, accuracy = 0.1),
    median_gdp  = comma(median_gdp, accuracy = 1)
  ) |>
  rename(
    "Continent"              = continent,
    "Countries"              = countries,
    "Median life expectancy" = median_life,
    "Median GDP per capita"  = median_gdp
```

Table 9.3: Summary statistics by continent, 2007

Continent	Countries	Median life expectancy	Median GDP per capita
Africa	52	52.9	1,452
Americas	25	72.9	8,948
Asia	33	72.4	4,471
Europe	30	78.6	28,054
Oceania	2	80.7	29,810

```
  ) |>
  tt(caption = "Summary statistics by continent, 2007")
```

A few common formatters from `scales`:

- `dollar()` — currency with `$` and commas
- `percent()` — multiplies by 100 and appends `%`
- `comma()` — comma-separated thousands
- `number(accuracy = 0.1)` — controls decimal places

Formatting in the data frame works across any table package. As an alternative, `tinytable::format_tt()` formats inside the table object while keeping the underlying column numeric.

9.7 Publication-style data tables with `tinytable`

`tinytable` is useful when the table needs to look good in a document. It starts with a data frame and returns a formatted table.

```
vehicle_sample <- mtcars |>
  rownames_to_column("car") |>
  slice_sample(n = 6) |>
  select(car, mpg, cyl, disp, hp, wt)

tt(vehicle_sample)
```

Add a caption and format the numbers.

car	mpg	cyl	disp	hp	wt
Pontiac Firebird	19.2	8	400.0	175	3.845
Hornet Sportabout	18.7	8	360.0	175	3.440
Volvo 142E	21.4	4	121.0	109	2.780
Lotus Europa	30.4	4	95.1	113	1.513
Cadillac Fleetwood	10.4	8	472.0	205	5.250
Lincoln Continental	10.4	8	460.0	215	5.424

Table 9.4: A Small Sample of Vehicle Characteristics

car	mpg	cyl	disp	hp	wt
Pontiac Firebird	19	8	400	175	4
Hornet Sportabout	19	8	360	175	3
Volvo 142E	21	4	121	109	3
Lotus Europa	30	4	95	113	2
Cadillac Fleetwood	10	8	472	205	5
Lincoln Continental	10	8	460	215	5

```
tt(
  vehicle_sample,
  caption = "A Small Sample of Vehicle Characteristics",
  digits = 1
)
```

You can add notes to explain the data source or a measurement choice.

```
tt(
  vehicle_sample,
  caption = "A Small Sample of Vehicle Characteristics",
  digits = 1,
  notes = "Data are from the built-in mtcars dataset."
)
```

You can also style headers and selected cells.

```
tt(vehicle_sample, digits = 1) |>
  style_tt(i = 0, color = "white", background = "steelblue", fontweight = "bold") |>
  style_tt(i = 2, j = 4, background = "lightyellow")
```

Table 9.5: A Small Sample of Vehicle Characteristics

car	mpg	cyl	disp	hp	wt
Pontiac Firebird	19	8	400	175	4
Hornet Sportabout	19	8	360	175	3
Volvo 142E	21	4	121	109	3
Lotus Europa	30	4	95	113	2
Cadillac Fleetwood	10	8	472	205	5
Lincoln Continental	10	8	460	215	5

Data are from the built-in mtcars dataset.

car	mpg	cyl	disp	hp	wt
Pontiac Firebird	19	8	400	175	4
Hornet Sportabout	19	8	360	175	3
Volvo 142E	21	4	121	109	3
Lotus Europa	30	4	95	113	2
Cadillac Fleetwood	10	8	472	205	5
Lincoln Continental	10	8	460	215	5

Column groups help when several variables belong together.

```
tt(vehicle_sample, digits = 1) |>
  group_tt(
    j = list(
      "Identity" = 1,
      "Engine and Performance" = 2:6
    )
  )
```

9.8 Conditional formatting

A table reads faster when extreme values stand out. `tinytable::style_tt()` accepts row and column indices to apply background colors, font weights, and other styles. Combine it with `which.max()` and `which.min()` to color cells based on the data itself.

Identity	Engine and Performance				
car	mpg	cyl	disp	hp	wt
Pontiac Firebird	19	8	400	175	4
Hornet Sportabout	19	8	360	175	3
Volvo 142E	21	4	121	109	3
Lotus Europa	30	4	95	113	2
Cadillac Fleetwood	10	8	472	205	5
Lincoln Continental	10	8	460	215	5

Table 9.6: Highest and lowest continent in 2007 highlighted

Continent	1957	1977	1997	2007
Africa	41	49	53	53
Americas	56	66	72	73
Asia	48	61	70	72
Europe	68	72	76	79
Oceania	70	73	78	81

```
wide_life <- life_by_year |>
  pivot_wider(names_from = year, values_from = median_life) |>
  rename("Continent" = continent)

highest_2007 <- which.max(wide_life[["2007"]])
lowest_2007  <- which.min(wide_life[["2007"]])

tt(
  wide_life,
  digits  = 1,
  caption = "Highest and lowest continent in 2007 highlighted"
) |>
  style_tt(i = highest_2007, background = "#cde7c2") |>
  style_tt(i = lowest_2007, background = "#f7c8c2")
```

Highlighting works best when one or two values per table need attention. Past three or four, the colors compete with each other and the reader loses the cue.

	Unique	Missing Pct.	Mean	SD	Min	Median	Max	Histogram
lifeExp	142	0	67.0	12.1	39.6	71.9	82.6	
gdpPercap	142	0	11 680.1	12 859.9	277.6	6124.4	49 357.2	
continent	N	%						
Africa	52	36.6						
Americas	25	17.6						
Asia	33	23.2						
Europe	30	21.1						
Oceania	2	1.4						

		N	%
continent	Africa	52	36.6
	Americas	25	17.6
	Asia	33	23.2
	Europe	30	21.1
	Oceania	2	1.4
life_expectancy_group	70 years or more	83	58.5
	Under 70 years	59	41.5

9.9 Summary tables with `modelsummary`

The `modelsummary` package includes useful functions for descriptive statistics. The function `datasummary_skim()` comes from `modelsummary`.

```
gapminder_2007 |>
  select(continent, lifeExp, gdpPercap) |>
  datasummary_skim()
```

The default skim table is good for numeric variables. Categorical variables can be summarized separately.

```
gapminder_2007 |>
  select(continent, life_expectancy_group) |>
  datasummary_skim(type = "categorical")
```

`datasummary_crosstab()` creates crosstabs using a formula. The left side of the formula becomes the rows. The right side becomes the columns.

continent		70 years or more	Under 70 years	All
Africa	N	7	45	52
	% row	13.5	86.5	100.0
Americas	N	22	3	25
	% row	88.0	12.0	100.0
Asia	N	22	11	33
	% row	66.7	33.3	100.0
Europe	N	30	0	30
	% row	100.0	0.0	100.0
Oceania	N	2	0	2
	% row	100.0	0.0	100.0
All	N	83	59	142
	% row	58.5	41.5	100.0

```
datasummary_crosstab(
  continent ~ life_expectancy_group,
  data = gapminder_2007
)
```

9.10 Correlation tables with `corrr`

A correlation table inspects linear relationships among numeric variables. The `corrr` package is built for correlation workflows: `correlate()` computes the matrix, and helpers like `shave()` and `fashion()` format it for display.

```
correlation_data <- gapminder_2007 |>
  transmute(
    `Life expectancy`    = lifeExp,
    `Log GDP per capita` = log10(gdpPercap),
    `Log population`     = log10(pop)
  )

correlation_data |>
  correlate(quiet = TRUE) |>
  fashion(decimals = 2)
```

Table 9.7: Correlations among Gapminder variables, 2007

term	Life expectancy	Log GDP per capita	Log population
Life expectancy	1	0.809	0.065
Log GDP per capita	0.809	1	-0.046
Log population	0.065	-0.046	1

```
                term Life.expectancy Log.GDP.per.capita Log.population
1    Life expectancy                               .81             .07
2 Log GDP per capita             .81                             -.05
3     Log population             .07               -.05
```

```
correlation_data |>
  correlate(quiet = TRUE) |>
  shave() |>
  fashion(decimals = 2)
```

```
                term Life.expectancy Log.GDP.per.capita Log.population
1    Life expectancy
2 Log GDP per capita             .81
3     Log population             .07               -.05
```

`shave()` hides the redundant upper triangle and `fashion()` rounds and right-pads the numbers so the columns align cleanly.

For a publication version, pass the result of `correlate()` to `tt()`.

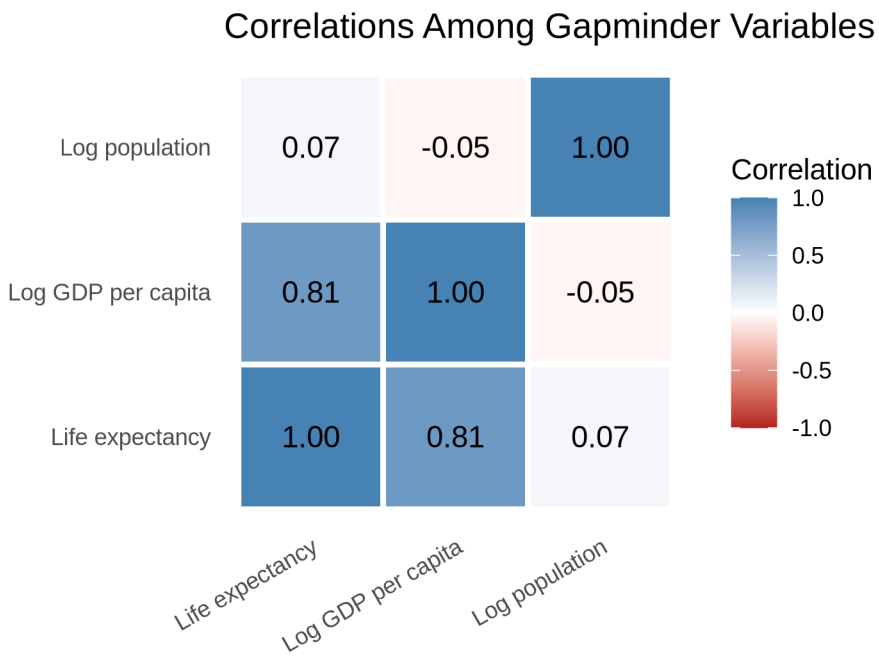
```
correlation_data |>
  correlate(quiet = TRUE, diagonal = 1) |>
  tt(
    caption = "Correlations among Gapminder variables, 2007",
    digits  = 2
  )
```

9.10.1 As a heatmap

The same correlations can be shown as a heatmap. Color makes direction and strength easier to scan than rows of numbers.

```r
correlation_table <- correlation_data |>
  correlate(quiet = TRUE, diagonal = 1) |>
  stretch()

ggplot(correlation_table, aes(x = x, y = y, fill = r)) +
  geom_tile(color = "white", linewidth = 1) +
  geom_text(aes(label = sprintf("%.2f", r)), size = 4) +
  scale_fill_gradient2(
    low    = "firebrick",
    mid    = "white",
    high   = "steelblue",
    limits = c(-1, 1),
    name   = "Correlation"
  ) +
  labs(
    title = "Correlations Among Gapminder Variables",
    x = NULL,
    y = NULL
  ) +
  coord_equal() +
  theme_minimal() +
  theme(
    panel.grid  = element_blank(),
    axis.text.x = element_text(angle = 30, hjust = 1)
  )
```



A small number of variables fits cleanly in a table; once there are eight or ten, the heatmap usually reads faster.

9.11 Cross-referencing tables in text

Quarto can number tables automatically and let you refer to them by name. Add a `label:` and `tbl-cap:` to the chunk that produces the table, then use `@tbl-...` in prose.

```
::: {#tbl-continents .cell tbl-cap='Summary statistics by continent, 2007'}

```{.r .cell-code}
continent_summary |>
  tt(digits = 1)
```

::: {.cell-output-display}
\begin{table}
\centering
\begin{tblr}[%% tabularray outer open
]   %% tabularray outer close
{   %% tabularray inner open
```

```
colspec={Q[]Q[]Q[]Q[]},
hline{2}={1-4}{solid, black, 0.05em},
hline{1}={1-4}{solid, black, 0.08em},
hline{7}={1-4}{solid, black, 0.08em},
}                     %% tabularray inner close
continent & countries & median_life & median_gdp \\
Africa & 52 & 53 & 1452 \\
Americas & 25 & 73 & 8948 \\
Asia & 33 & 72 & 4471 \\
Europe & 30 & 79 & 28054 \\
Oceania & 2 & 81 & 29810 \\
\end{tblr}
\end{table}
:::
:::


The summary appears in @tbl-continents.
```

Quarto renders `@tbl-continents` as `Table 1`, `Table 2`, and so on, and the reference is a clickable link in HTML output. The label must start with `tbl-` for the cross-reference to register. The same pattern applies to figures with `fig-` and equations with `eq-`.

9.12 Exporting tables

When a table belongs in another file, save it deliberately. The examples below are not run during rendering because they write files. If you use them, make sure the output folder exists.

```
vehicle_table <- tt(
  vehicle_sample,
  caption = "A Small Sample of Vehicle Characteristics",
  digits = 1
)

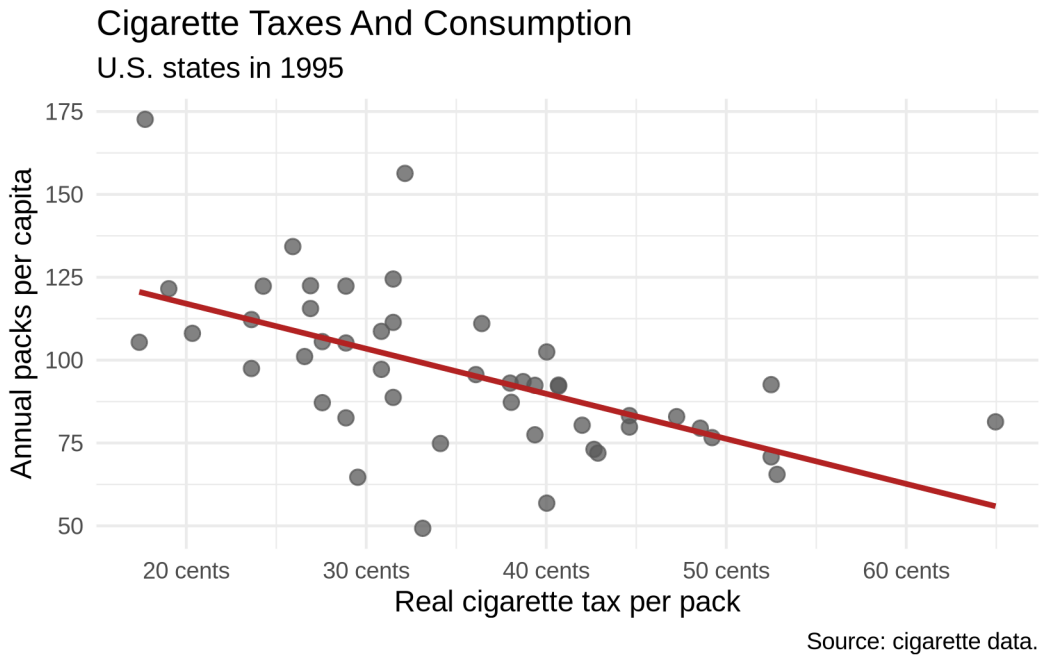
vehicle_table |> save_tt("Tables/vehicle_characteristics.html", overwrite = TRUE)
vehicle_table |> save_tt("Tables/vehicle_characteristics.docx", overwrite = TRUE)
vehicle_table |> save_tt("Tables/vehicle_characteristics.png", overwrite = TRUE)
```

In a Quarto project, keep exported tables in a clear output folder such as `Tables/` or `Output/`. Avoid saving tables into the `Data/` folder, which should hold source data.

9.13 Short exercise

Use `gapminder_2007`.

1. Create a new variable that classifies countries as above or below the median GDP per capita.
2. Make a `janitor::tabyl()` table of `continent` by your new GDP group.
3. Add row percentages and counts.
4. Make a `tinytable` version of a six-row sample containing `country`, `continent`, `lifeExp`, and `gdpPercap`.

9.14 Extra: highly designed tables with `gt`

`gt` is built for HTML tables with fine-grained control over every cell, header, and footer. It is less concise than `tinytable` but offers more design surface for HTML-first publications.

```
library(gt)

continent_summary |>
  gt() |>
  tab_header(
    title    = "Life expectancy and GDP by continent",
    subtitle = "Gapminder, 2007"
  ) |>
  fmt_number(columns = median_life, decimals = 1) |>
  fmt_currency(columns = median_gdp, decimals = 0) |>
  cols_label(
    continent   = "Continent",
    countries   = "Countries",
    median_life = "Median life expectancy",
    median_gdp  = "Median GDP per capita"
  )
```

`gt` has many more options for color, grouping, and conditional formatting. For a Word-first workflow, `flextable` is a similar choice with stronger Word output.

9.15 Extra: interactive tables with `DT`

Interactive tables are useful when readers need to search, sort, or inspect more rows than would fit comfortably on a page. The `DT` package creates HTML tables with built-in search

Life expectancy and GDP by continent

Gapminder, 2007

Continent	Countries	Median life expectancy	Median GDP per capita
Africa	52	52.9	$1,452
Americas	25	72.9	$8,948
Asia	33	72.4	$4,471
Europe	30	78.6	$28,054
Oceania	2	80.7	$29,810

and pagination.

```
table_for_browsing <- gapminder_2007 |>
  select(country, continent, lifeExp, gdpPercap) |>
  mutate(gdpPercap = round(gdpPercap)) |>
  arrange(continent, desc(lifeExp))

if (knitr::is_html_output() && requireNamespace("DT", quietly = TRUE)) {
  DT::datatable(
    table_for_browsing,
    rownames = FALSE,
    options = list(
      pageLength = 8,
      autoWidth = TRUE
    )
  )
} else {
  tt(table_for_browsing |> slice_head(n = 12), digits = 1)
}
```

Interactive tables work best for exploration or web appendices. For a printed report, a smaller static table is usually better.

country	continent	lifeExp	gdpPercap
Reunion	Africa	76	7670
Libya	Africa	74	12057
Tunisia	Africa	74	7093
Mauritius	Africa	73	10957
Algeria	Africa	72	6223
Egypt	Africa	71	5581
Morocco	Africa	71	3820
Sao Tome and Principe	Africa	66	1598
Comoros	Africa	65	986
Mauritania	Africa	64	1803
Senegal	Africa	63	1712
Ghana	Africa	60	1328

10 Interactivity And Dashboards

10.1 Learning Goals

By the end of this chapter, the main goals are:

- convert a static `ggplot2` figure to an interactive HTML figure with `ggplotly()`
- control hover text so interactive output shows useful information
- build a simple animated plot with a time slider
- recognize when a dashboard is a good output format
- understand the basic structure of a Quarto dashboard file

10.2 Where This Fits

The earlier chapters focused on data import, data wrangling, static plots, labels, scales, annotations, and complete figure revision. This chapter keeps those skills in place and changes the output form.

Interactivity is most useful when readers need to inspect details, zoom into dense areas, or compare individual cases. A dashboard is most useful when several related outputs belong together.

The underlying workflow is still the same:

1. load the data
2. inspect the variables
3. wrangle the data into the shape needed for the plot
4. build a clear static version
5. add interactivity or package the output only after the basic figure works

10.3 Setup

This chapter uses the local cigarette data copied into the 2026 course project.

```
library(tidyverse)
library(plotly)
library(scales)
library(DT)

cigarettes <- read_csv(
  "Data/cigarettes/cigarette.csv",
  show_col_types = FALSE
) |>
  select(-any_of("rownames")) |>
  rename(
    packpc = any_of("packs"),
    avgprs = any_of("price"),
    pop = any_of("population")
  ) |>
  mutate(
    year_date = make_date(year),
    real_tax = tax / cpi,
    real_price = avgprs / cpi,
    income_per_capita = income / pop
  )
```

10.4 From Static To Interactive

The easiest route into interactivity is to begin with a normal `ggplot2` plot. The plot below uses the final year in the cigarette data and compares state cigarette taxes with annual packs per capita.

```
cigarettes_1995 <- cigarettes |>
  filter(year == max(year))

tax_plot <- ggplot(
  cigarettes_1995,
  aes(x = real_tax, y = packpc)
) +
  geom_point(color = "gray35", alpha = 0.75, size = 2.4) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "firebrick") +
  scale_x_continuous(labels = label_number(suffix = " cents")) +
  labs(
    title = "Cigarette Taxes And Consumption",
```

```
    subtitle = "U.S. states in 1995",
    x = "Real cigarette tax per pack",
    y = "Annual packs per capita",
    caption = "Source: cigarette data."
  ) +
  theme_minimal()

tax_plot
```



The static plot should come first because it is easier to check. The next step wraps the completed plot in `ggplotly()`.

```
ggplotly(tax_plot)
```

That one call adds hover, zoom, and pan behavior in HTML output.

10.5 HTML Widgets And Output Formats

`ggplotly()` creates an HTML widget. HTML widgets are R objects that rely on JavaScript in the rendered page. This same general idea applies to `plotly`, `highcharter`, `DT`, `leaflet`, and many dashboard components.

That output-format detail matters. HTML widgets are interactive in HTML documents, websites, slides, and dashboards. They are not interactive in PDF output because PDF is a static document format. This chapter is therefore mainly about HTML output.

10.6 Custom Hover Text

The default hover text is often too mechanical. It usually reports mapped variables, but it may not show the case name or format numbers cleanly.

Add a `text` aesthetic when building the plot. Then tell `ggplotly()` to use only that text for the tooltip.

```
cigarettes_1995_hover <- cigarettes_1995 |>
  mutate(
    hover_text = paste0(
      state,
      "<br>Real tax: ", number(real_tax, accuracy = 0.1), " cents",
      "<br>Packs per capita: ", number(packpc, accuracy = 0.1),
      "<br>Real price: ", number(real_price, accuracy = 0.1), " cents"
    )
  )

tax_hover_plot <- ggplot(
  cigarettes_1995_hover,
  aes(x = real_tax, y = packpc, text = hover_text)
) +
  geom_point(color = "steelblue", alpha = 0.8, size = 2.5) +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE, color = "firebrick") +
  scale_x_continuous(labels = label_number(suffix = " cents")) +
  labs(
    title = "Cigarette Taxes And Consumption",
    subtitle = "Hover to inspect each state",
    x = "Real cigarette tax per pack",
    y = "Annual packs per capita"
  ) +
  theme_minimal()

ggplotly(tax_hover_plot, tooltip = "text")
```

The `text = hover_text` part is a special bridge between `ggplot2` and `plotly`. A normal static `ggplot2` figure does not draw the `text` aesthetic on the plot. After conversion,

`ggplotly(tax_hover_plot, tooltip = "text")` tells `plotly` to use that text as the hover label.

The `<br>` tags create line breaks inside the HTML tooltip. This is one place where HTML syntax appears inside R code because the output is an HTML widget.

10.7 Interactive Time Series

Interactivity is also useful for line charts. Hover text can identify the state, year, and exact value without putting every label directly on the static figure.

```
focus_states <- c("CA", "KY", "NY", "TX", "VA")

focus_trends <- cigarettes |>
  filter(state %in% focus_states) |>
  mutate(
    hover_text = paste0(
      state,
      "<br>Year: ", year,
      "<br>Packs per capita: ", number(packpc, accuracy = 0.1),
      "<br>Real tax: ", number(real_tax, accuracy = 0.1), " cents"
    )
  )

trend_plot <- ggplot(
  focus_trends,
  aes(x = year, y = packpc, color = state, group = state, text = hover_text)
) +
  geom_line(linewidth = 0.9) +
  geom_point(size = 1.7) +
  scale_x_continuous(breaks = seq(1985, 1995, by = 2)) +
  labs(
    title = "Per-Capita Cigarette Consumption Over Time",
    subtitle = "Five selected states",
    x = NULL,
    y = "Annual packs per capita",
    color = "State"
  ) +
  theme_minimal()

ggplotly(trend_plot, tooltip = "text")
```

The `group = state` mapping tells `ggplot2` and `plotly` which observations belong to the same line. This matters because the hover text is different for every row. Without an explicit group, the conversion to `plotly` can treat points as separate observations rather than connecting them into one line per state.

10.8 Linked Highlighting

`plotly` can also dim unselected lines when the pointer moves across a figure. The `highlight_key()` function attaches an interactive key to the data. Here the key is the state abbreviation.

```
trend_key <- highlight_key(focus_trends, ~state)

highlight_plot <- ggplot(
  trend_key,
  aes(x = year, y = packpc, color = state, group = state)
) +
  geom_line(linewidth = 1) +
  geom_point(aes(text = hover_text), size = 1.7) +
  scale_x_continuous(breaks = seq(1985, 1995, by = 2)) +
  labs(
    title = "Hover Highlighting By State",
    x = NULL,
    y = "Annual packs per capita",
    color = "State"
  ) +
  theme_minimal()

ggplotly(highlight_plot, tooltip = "text") |>
  highlight(
    on = "plotly_hover",
    dynamic = TRUE,
    opacityDim = 0.18,
    selected = attrs_selected(line = list(width = 4))
  )
```

The key tells `plotly` which rows should be treated as the same interactive object. Here all rows for California share the key `CA`, all rows for Kentucky share the key `KY`, and so on. That lets the whole state line respond together.

The `highlight()` call controls the interaction. `on = "plotly_hover"` means the highlighting happens when the pointer moves over a line or point. `dynamic = TRUE` lets the highlighted

object change as the pointer moves. `opacityDim = 0.18` fades the non-highlighted lines. `attrs_selected()` controls the appearance of the selected line.

This is useful when a line chart contains several series. It is less useful when a plot is already simple enough to read without interaction.

10.9 Animated Plots With Sliders

Some interactive plots add a time slider. A slider is useful when the same relationship changes across many time points and showing every year at once would be too crowded.

The example below uses the local `gapminder` data. The plot compares GDP per capita and life expectancy, then lets the year change through the slider. The `frame = year` argument tells `plotly` which variable should define the animation frames.

```
gapminder <- read_csv("Data/gapminder/gapminder.csv", show_col_types = FALSE)

plot_ly(
  data = gapminder,
  x = ~gdpPercap,
  y = ~lifeExp,
  color = ~continent,
  frame = ~year,
  type = "scatter",
  mode = "markers"
) |>
  layout(
    title = "Life Expectancy And GDP Per Capita Over Time",
    xaxis = list(type = "log", title = "GDP per capita"),
    yaxis = list(title = "Life expectancy")
  )
```

This example uses `plot_ly()` directly rather than converting a `ggplot2` object with `ggplotly()`. Direct `plot_ly()` syntax is sometimes useful when the interactive behavior is the main point, especially for animation.

10.10 Toolbar Control

By default, plotly shows a toolbar when the pointer enters the figure. `config(displayModeBar = FALSE)` hides it entirely, which produces a cleaner embed in a report. Set it to `TRUE` to keep the toolbar always visible when zoom or download buttons are useful.

```
ggplotly(tax_hover_plot, tooltip = "text") |>
  config(displayModeBar = FALSE)
```

Plotly's `layout()` function exposes many more options — tooltip styling, margins, legend placement, axis formatting — but most of that work is better done in `ggplot2` before the conversion, where the tools are more familiar.

`config()` is different from `layout()`. `layout()` changes the plot's visual structure, such as axes and titles. `config()` changes widget behavior, such as whether the toolbar appears.

10.11 Highcharter

`highcharter` is an R interface to the Highcharts JavaScript library. It is another route to interactive HTML figures. The syntax is different from `ggplot2`: instead of building a static plot and converting it with `ggplotly()`, `highcharter` builds the interactive widget directly.

The examples in this section use local or built-in data. They render as interactive widgets in HTML.

```
penguins <- read_csv("Data/penguins/penguins.csv", show_col_types = FALSE) |>
  filter(!is.na(flipper_length_mm), !is.na(bill_length_mm), !is.na(species))

highcharter::hchart(
  penguins,
  "scatter",
  highcharter::hcaes(
    x = flipper_length_mm,
    y = bill_length_mm,
    group = species
  )
) |>
  highcharter::hc_title(text = "Penguin Bill Length And Flipper Length") |>
  highcharter::hc_xAxis(title = list(text = "Flipper length")) |>
  highcharter::hc_yAxis(title = list(text = "Bill length"))
```

This example restores a second path into interactive scatterplots. It also reinforces a broader pattern: interactive charts still need clear variables, readable labels, and data cleaning before the widget is created.

10.12 Highcharter Line Charts

Line charts are a natural place to use interactivity because hover text can identify the exact date and value without crowding the static figure.

```
economics_long2 <- economics_long |>
  filter(variable %in% c("pop", "uempmed", "unemploy"))

highcharter::hchart(
  economics_long2,
  "line",
  highcharter::hcaes(x = date, y = value01, group = variable)
) |>
  highcharter::hc_title(text = "Selected U.S. Economic Indicators") |>
  highcharter::hc_xAxis(title = list(text = "Date")) |>
  highcharter::hc_yAxis(title = list(text = "Indexed value")) |>
  highcharter::hc_tooltip(valueDecimals = 2)
```

10.13 Highcharter Range Selector

Some Highcharts outputs include interface controls that go beyond ordinary hover text. A range selector lets the reader zoom to common time windows.

```
highcharter::hchart(
  economics,
  "line",
  highcharter::hcaes(x = date, y = uempmed)
) |>
  highcharter::hc_title(
    text = "Median Duration Of Unemployment Over Time"
  ) |>
  highcharter::hc_xAxis(type = "datetime", title = list(text = "Date")) |>
  highcharter::hc_yAxis(title = list(text = "Median duration in weeks")) |>
  highcharter::hc_rangeSelector(
    enabled = TRUE,
    buttons = list(
      list(type = "year", count = 1, text = "1y"),
      list(type = "year", count = 5, text = "5y"),
      list(type = "year", count = 10, text = "10y"),
      list(type = "all", text = "All")
```

```
    ),
    inputEnabled = TRUE
  )
```

The range selector is a good example of why HTML output can matter. The same underlying data becomes an exploratory widget in HTML.

10.14 Highcharter Column Chart

Interactive bar and column charts are useful when exact values should appear on hover but do not need to be printed on every bar.

```
economics_decade <- economics |>
  mutate(decade = floor(year(date) / 10) * 10) |>
  group_by(decade) |>
  summarize(avg_psavert = mean(psavert, na.rm = TRUE))

highcharter::hchart(
  economics_decade,
  "column",
  highcharter::hcaes(x = decade, y = avg_psavert)
) |>
  highcharter::hc_title(text = "Average Personal Savings Rate By Decade") |>
  highcharter::hc_xAxis(title = list(text = "Decade")) |>
  highcharter::hc_yAxis(title = list(text = "Average savings rate")) |>
  highcharter::hc_tooltip(
    pointFormat = "Decade: {point.x}<br>Average savings rate: {point.y:.1f}%"
  )
```

10.15 Highcharter Heatmap

Heatmaps are another common interactive form. The hover text can show the exact value inside each cell.

```
diamonds_heatmap <- diamonds |>
  group_by(cut, clarity) |>
  summarize(median_price = median(price))

highcharter::hchart(
```

```
  diamonds_heatmap,
  "heatmap",
  highcharter::hcaes(x = cut, y = clarity, value = median_price),
  name = "Median price"
) |>
  highcharter::hc_title(text = "Median Diamond Price By Cut And Clarity") |>
  highcharter::hc_tooltip(
    pointFormat = "Cut: {point.x}<br>Clarity: {point.y}<br>Median price: ${point.value:.0f}"
  )
```

The practical choice is not `plotly` versus `highcharter` in the abstract. The choice is which tool produces the clearest result for a specific data task and output format.

10.16 Interactive Tables With DT

Interactivity is not limited to charts. Sometimes the best interactive object is a table that can be searched, sorted, and filtered. The `datatable()` function from the `DT` package creates an HTML table widget.

This is useful when individual cases matter. A static plot can show the pattern, while a table can preserve the lookup details.

```
cigarette_table <- cigarettes_1995 |>
  transmute(
    state,
    tax_cents = round(real_tax, 1),
    price_cents = round(real_price, 1),
    packs_per_capita = round(packpc, 1),
    income_per_capita = dollar(round(income_per_capita, 0))
  ) |>
  arrange(desc(tax_cents))

datatable(
  cigarette_table,
  rownames = FALSE,
  filter = "top",
  options = list(
    pageLength = 10,
    autoWidth = TRUE
  )
)
```

The search box and column filters make the table useful as a small lookup tool. The same object can also be embedded inside a dashboard card.

10.17 Interactivity Checklist

Interactivity is useful when it adds a specific capability:

- hover reveals case-level details that should not be printed on the static plot
- zooming helps inspect a dense region
- clicking or highlighting helps compare many series
- search and sorting help readers find individual rows
- a dashboard keeps related plots, tables, and summary quantities together

Interactivity is less useful when it only makes a simple chart more elaborate. If the static chart already communicates the main point clearly, interaction may add novelty rather than information.

10.18 Dashboards As Output

A dashboard is a layout for several related outputs. It is not a substitute for choosing good measures or building readable plots.

A useful dashboard usually contains a small set of components:

- one or two headline quantities
- one main relationship or trend
- one supporting distribution or ranking
- one table or lookup view when individual cases matter

The course includes a standalone dashboard demo for this chapter. `Demos/babynames-dashboard-demo.qmd` uses U.S. Social Security name data from 1880 to 2017. It includes overview cards, a trend chart for selected names, a top-15 ranking for the most recent year, a sex-split trend for a single name, and an all-time table.

The dashboard format is set in the YAML header:

```
---
title: "Dashboard Demo"
format:
  dashboard:
    orientation: columns
---
```

Inside the file, headings create sections, columns, and cards. The plots and tables are ordinary R code chunks. The source files contain comments explaining the data preparation and plotting choices.

Dashboard files can also use value boxes or cards for headline quantities. A minimal example looks like this:

````
```{.r}
valueBox(
  value = "48",
  title = "States in the data",
  icon = "map"
)
```
````

The surrounding dashboard layout determines where that card appears. The R code inside the card still follows the same rule as the rest of the course: calculate the quantity first, then display it clearly.

10.19 Exercise

Take one static plot from an earlier chapter and make it interactive.

1. Save the static plot as a named object.
2. Add a `text` aesthetic with at least three lines of hover text.
3. Convert the plot with `ggplotly(plot_name, tooltip = "text")`.
4. Decide whether the interactive version adds useful detail or only adds novelty.

10.20 Closing

Interactivity and dashboards work best after the static figure is already understandable. The durable sequence is to build the plot, check the plot, then choose whether interaction or dashboard packaging helps the audience use it.