Data Visualization and Shiny Apps with R
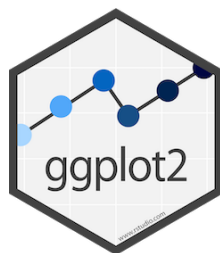
88th MORS Symposium
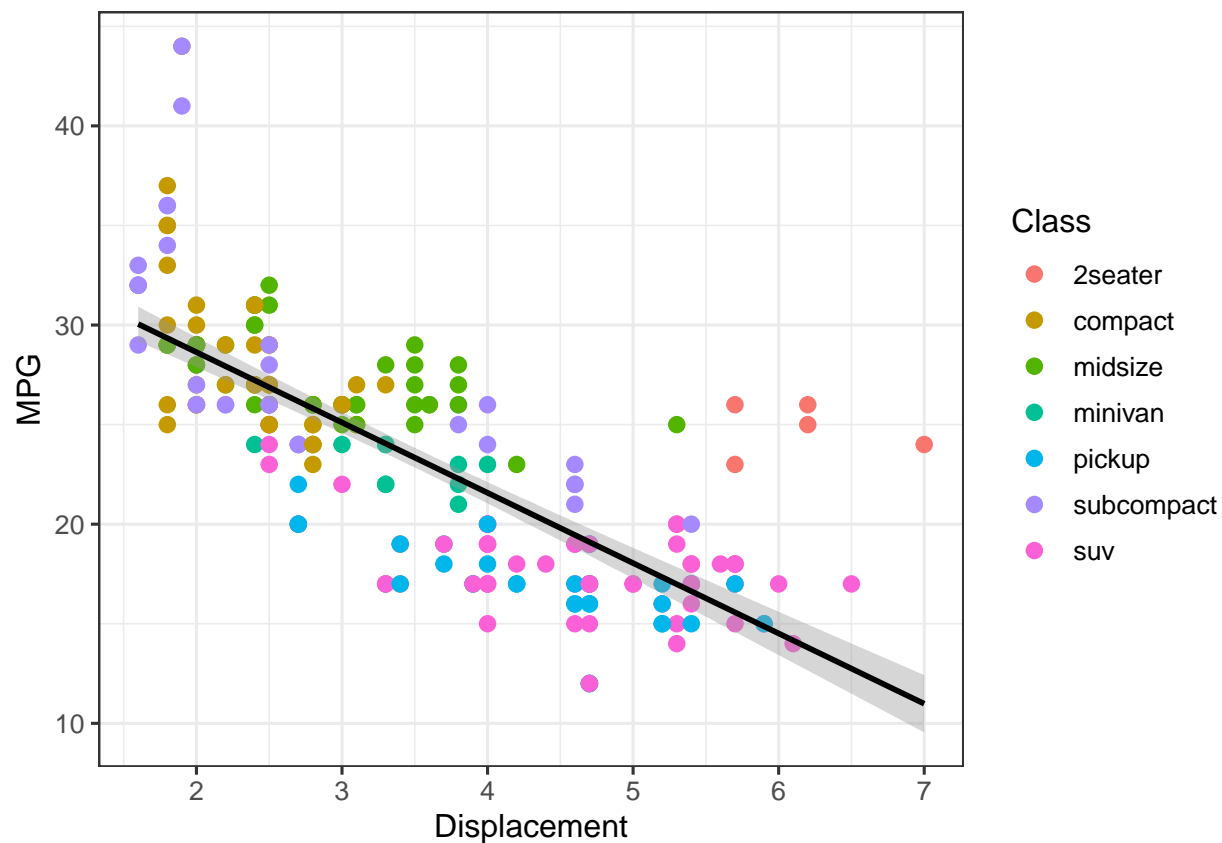
Contents

1 Data Visualization with ggplot2

1.1 About the tidyverse

The data visualization package, ggplot2, is a part of the tidyverse.

The tidyverse packages:

- Are a collection of R packages for designed for data science
- Were developed on the programming philosophy of statistician and avid R user Hadley Wickam
- Share the same underlying design, grammar, and data structures

1.2 ggplot2

ggplot2 can produce a large variety of two-dimensional plot types, including faceted plots and maps.
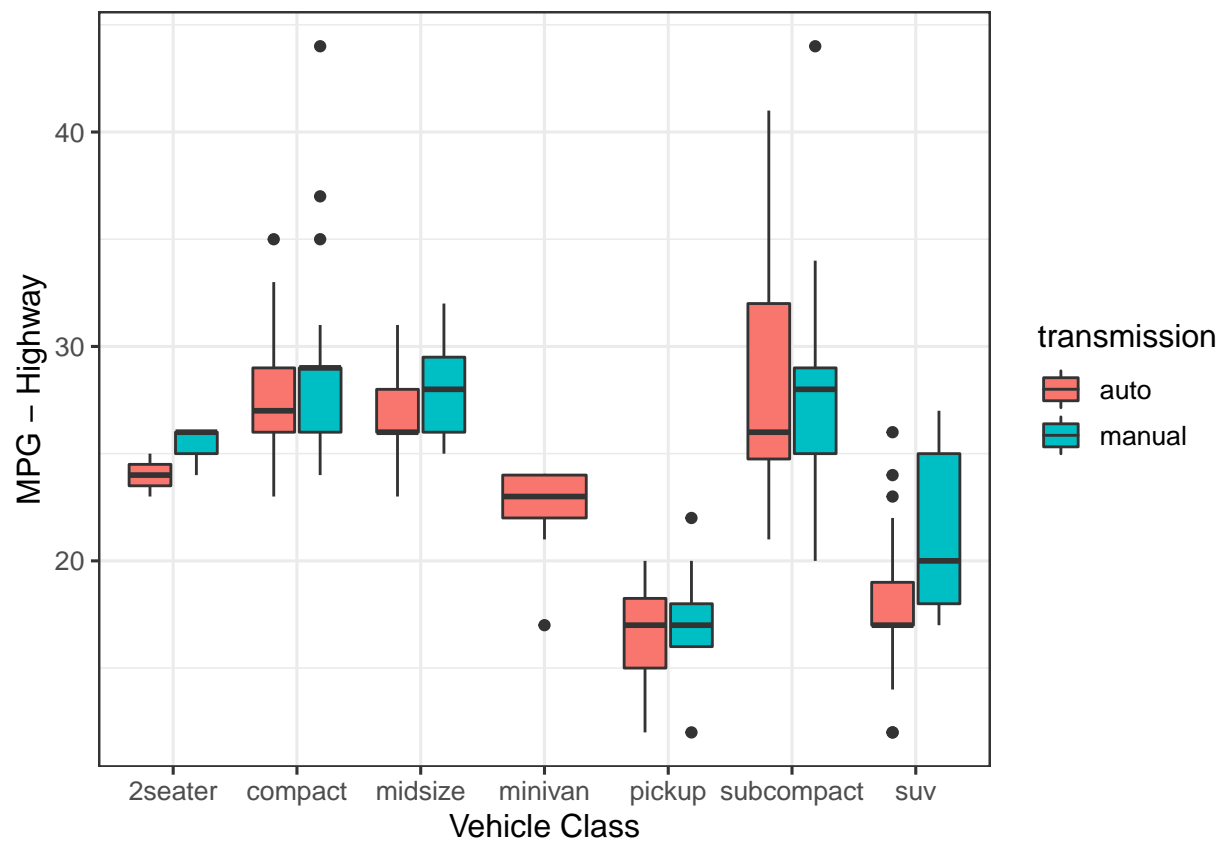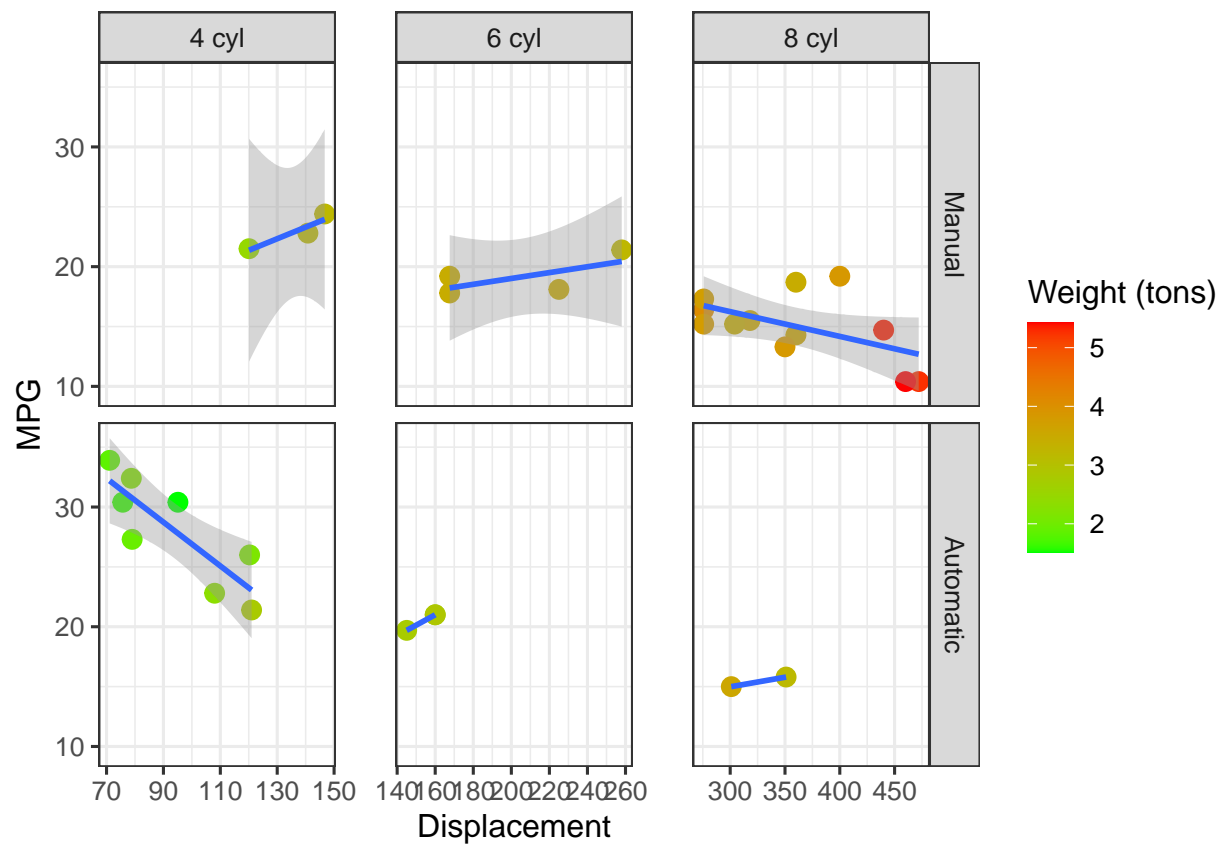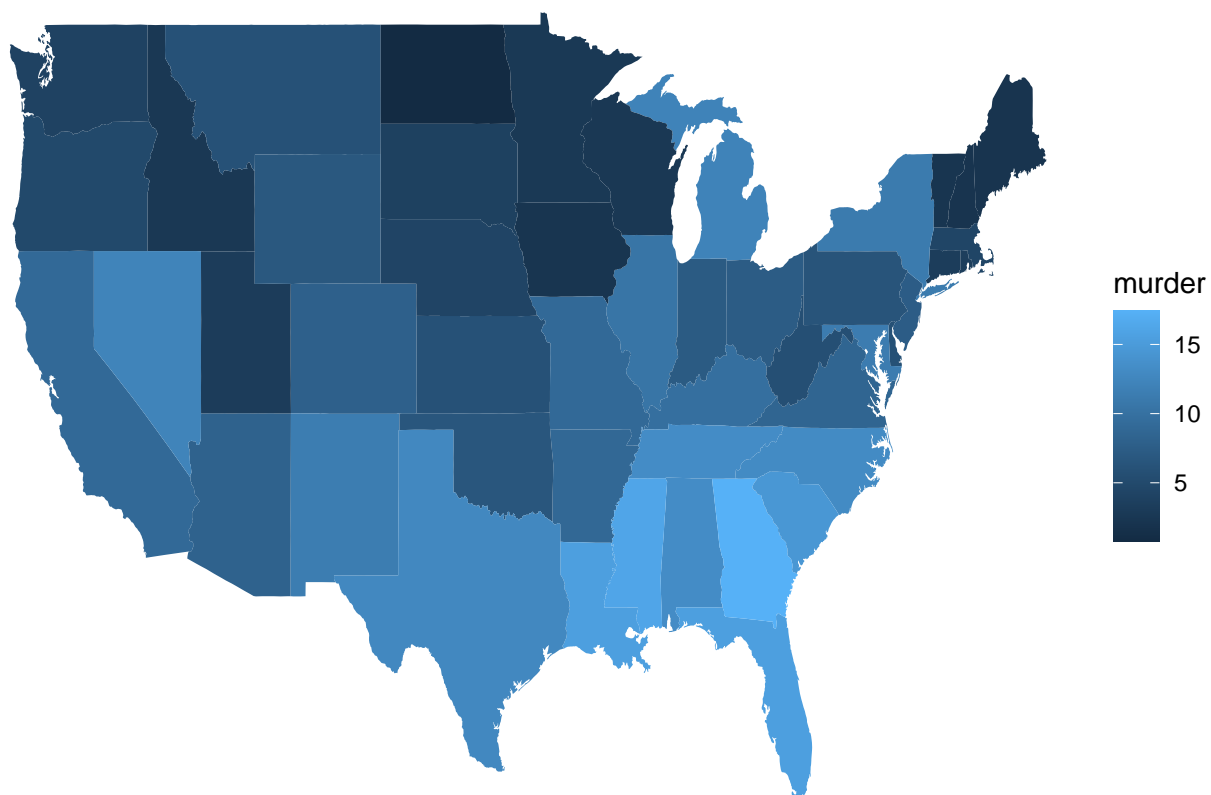
1.3 `qplot()`

The quickplot, or `qplot()` function, is an easy way to generate plots in a single function.

By default, `qplot()` generates a histogram if passed only one variable, or a scatterplot if passed two variables.

```
qplot(x=mpg$displ)
```
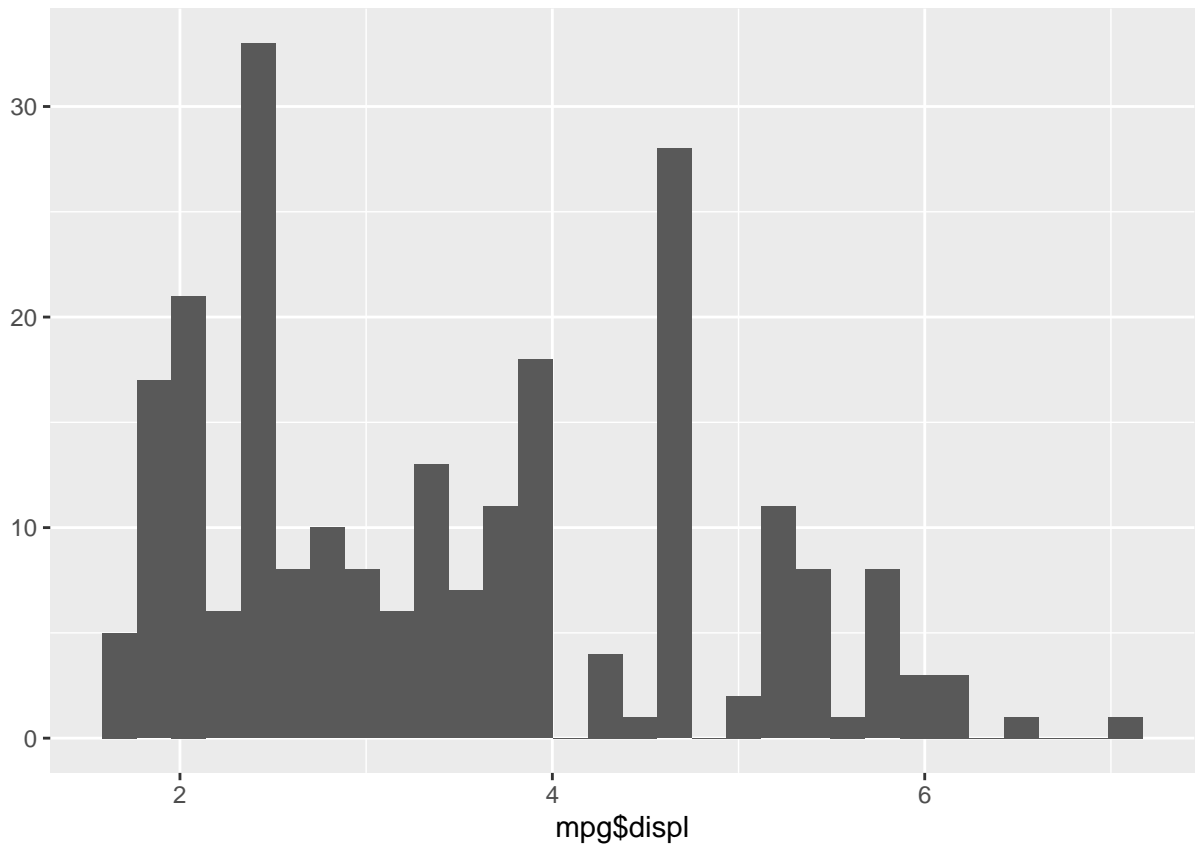
```
qplot(x=displ, y=cty, data=mpg)
```
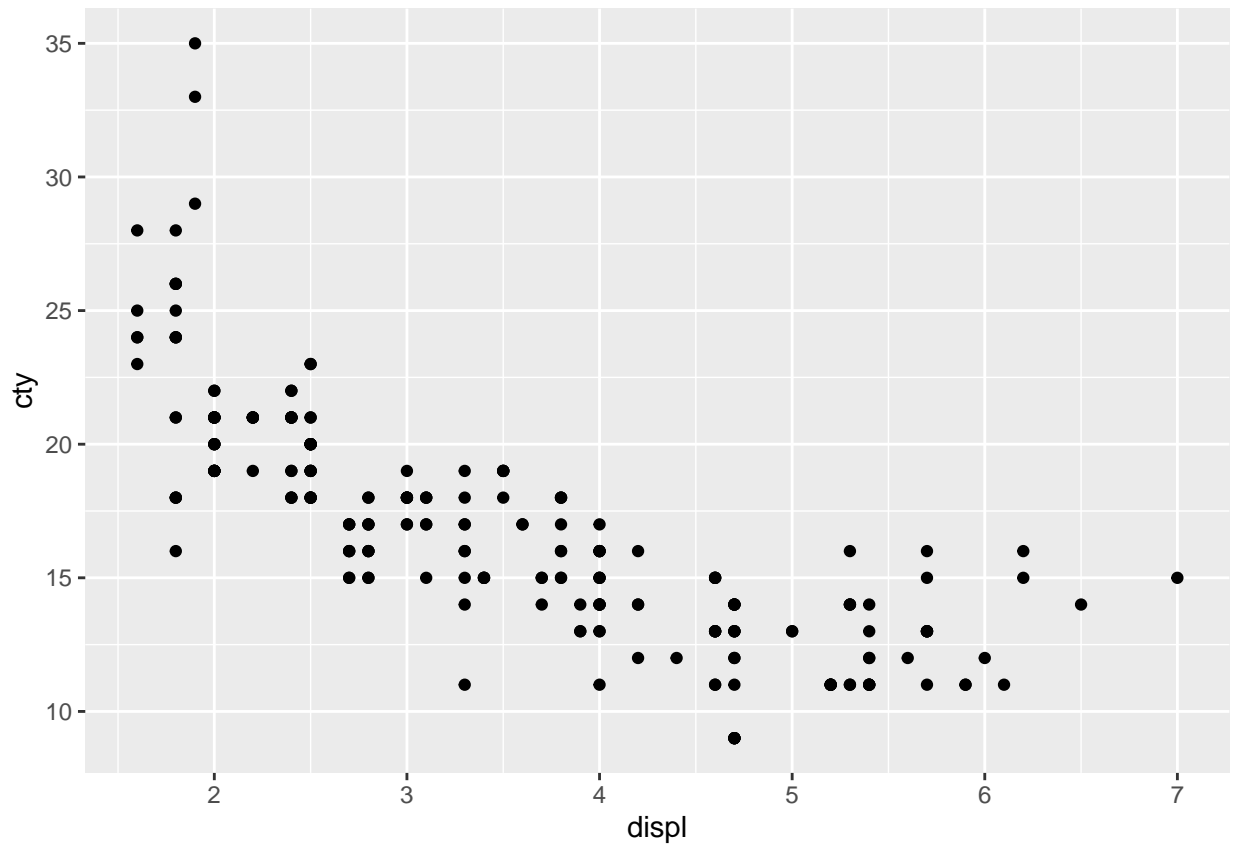
Other plot types can be generated by passing `geom=`. Other common `qplot()` types are:

- `"boxplot"`
- `"line"`
- `"area"`
- `"bar"`
- `"step"`
- `"density"`

```
qplot(x=class, data=mpg, geom="bar")
```
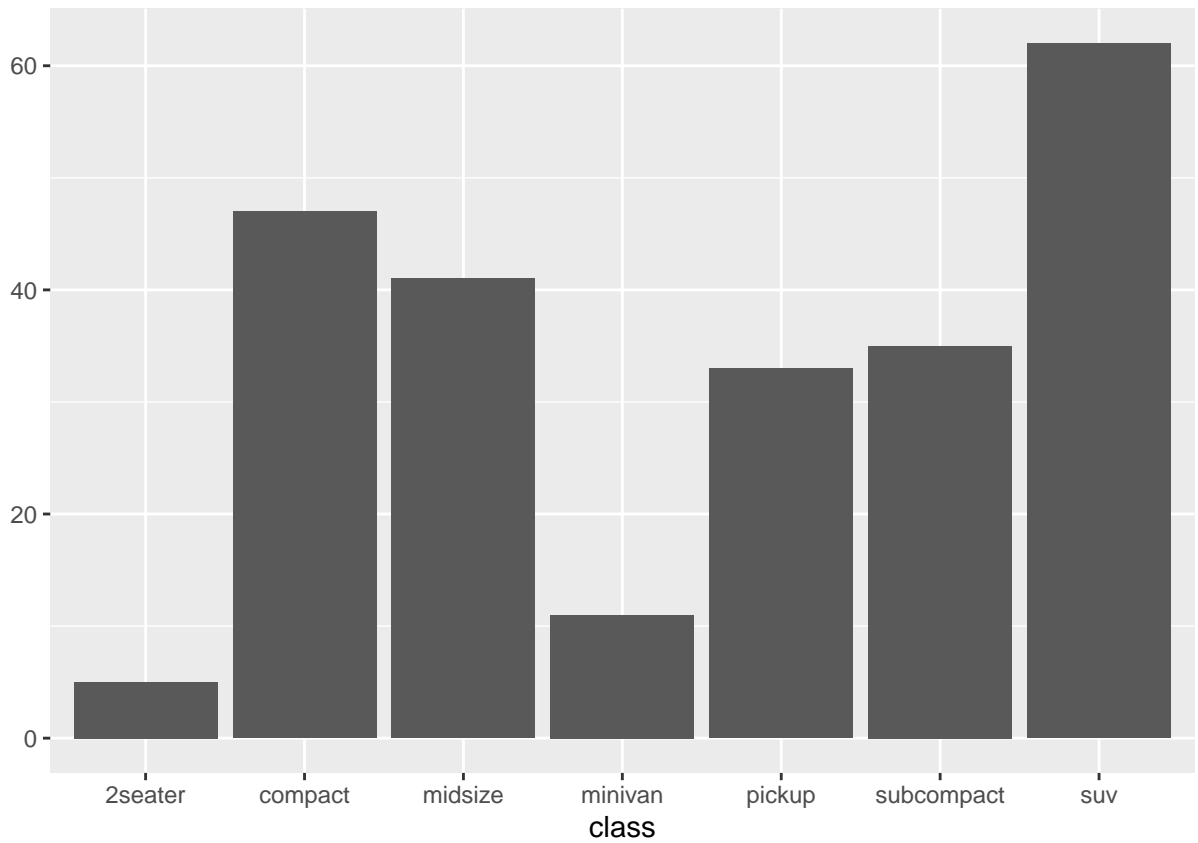
While it is possible to produce more complex plots using `qplot()`, it is generally better to use `ggplot()` for full control of all plot elements.

1.4   ggplot2 Mechanics

The `ggplot()` function follows a layered grammar of graphics to build plots from:

Coordinate System
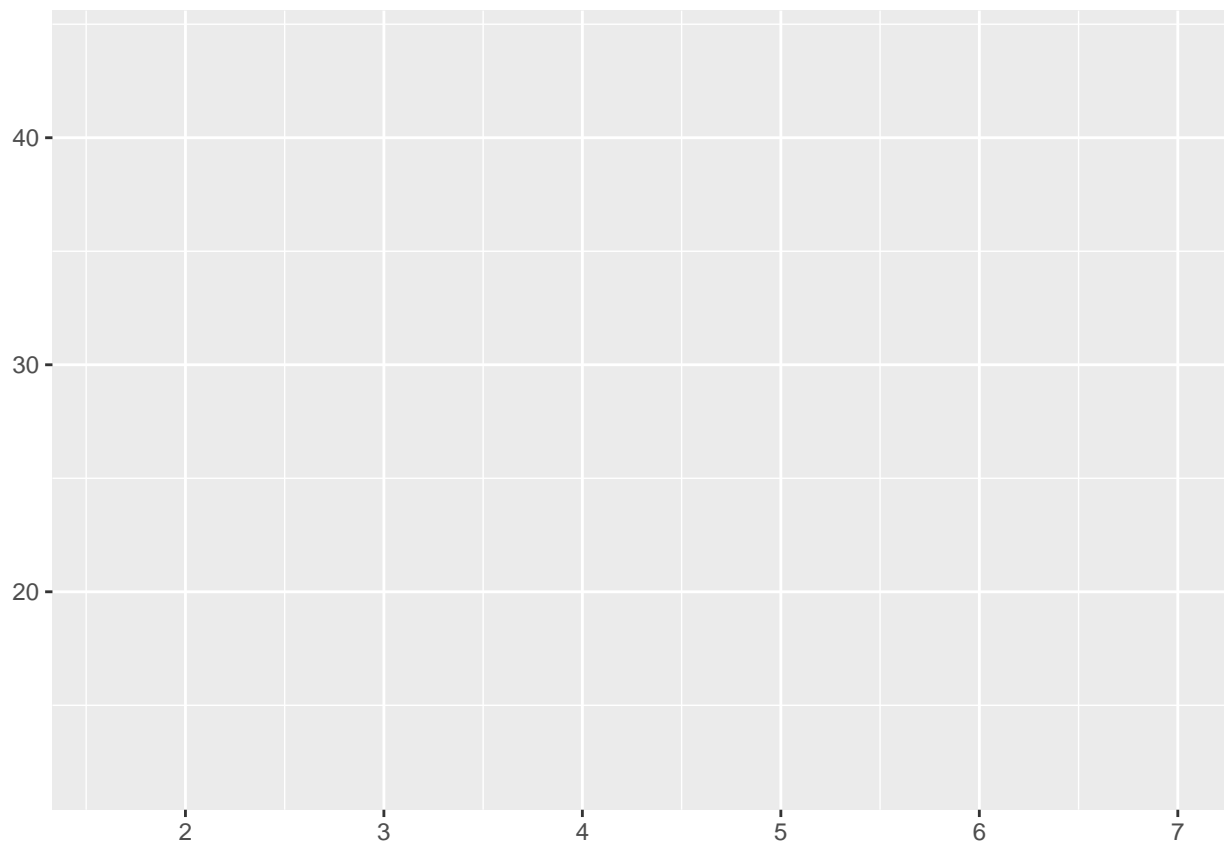
Geometric Objects

Plot Annotations
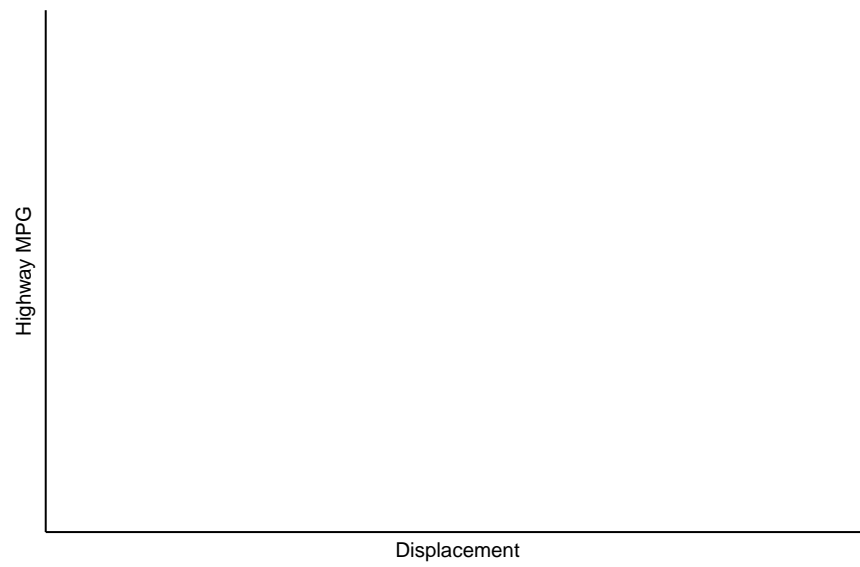
1.5 Aesthetic mapping

`ggplot()` sets up the coordinate system from the data it is passed

Single discrete variable:

```
ggplot(data=mpg, aes(x = class))
```
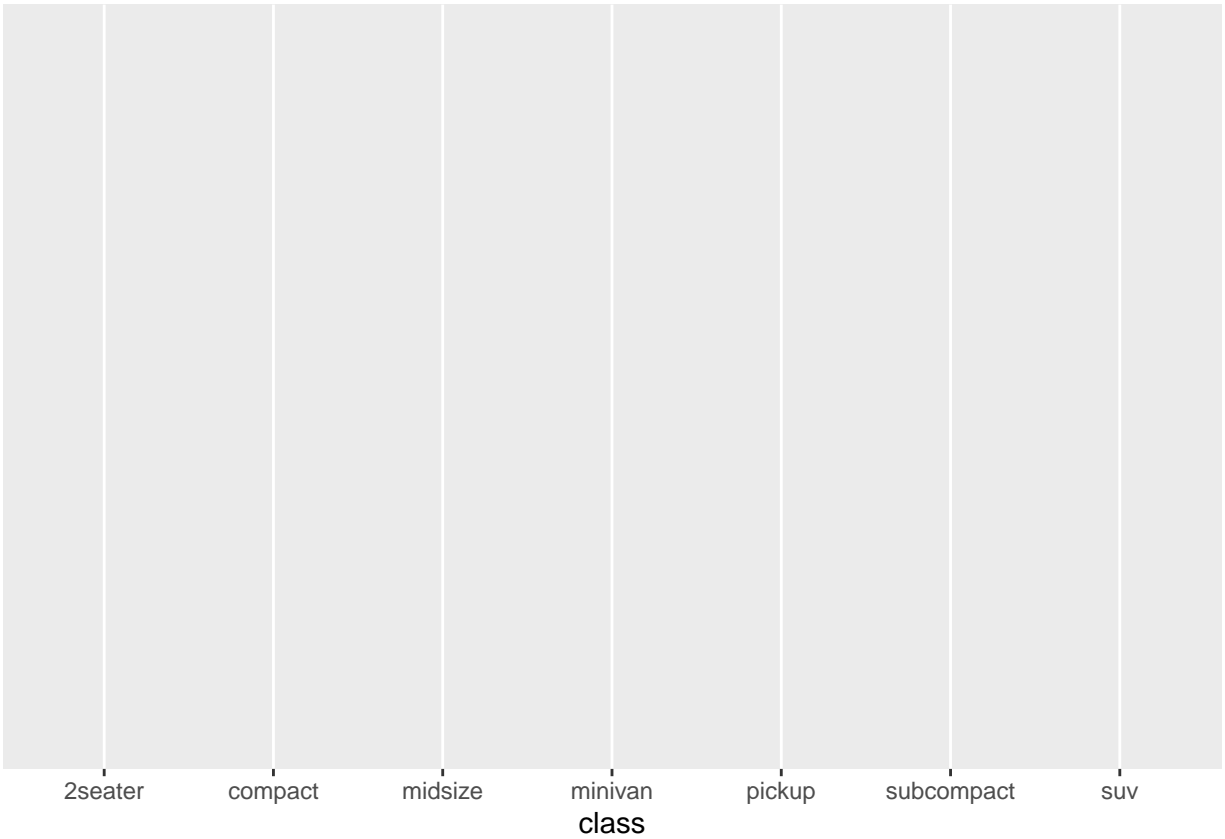
class

Discrete x variable, continuous y variable:

```
ggplot(data=mpg, aes(x = class, y = hwy))
```
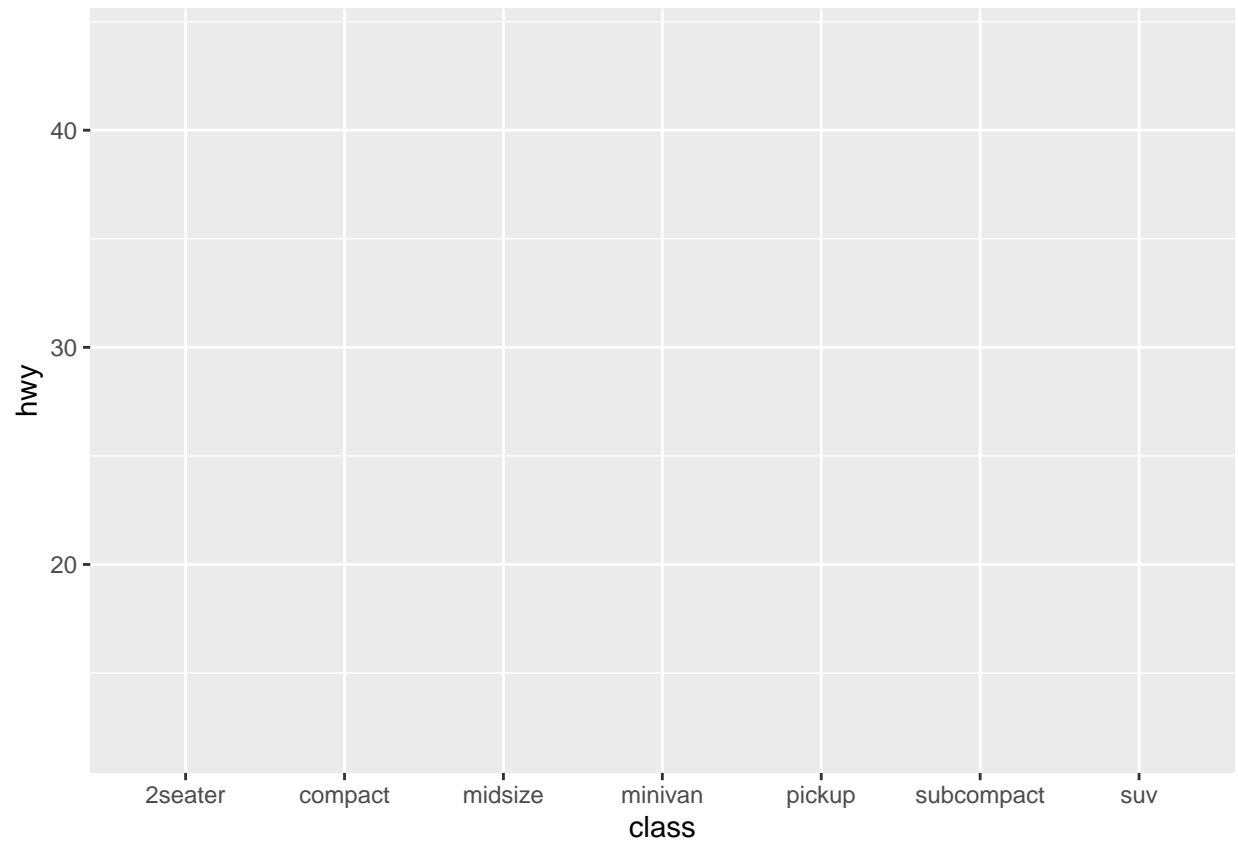
Continuous x and y variables:

```
ggplot(data=mpg, aes(x = displ, y = hwy))
```
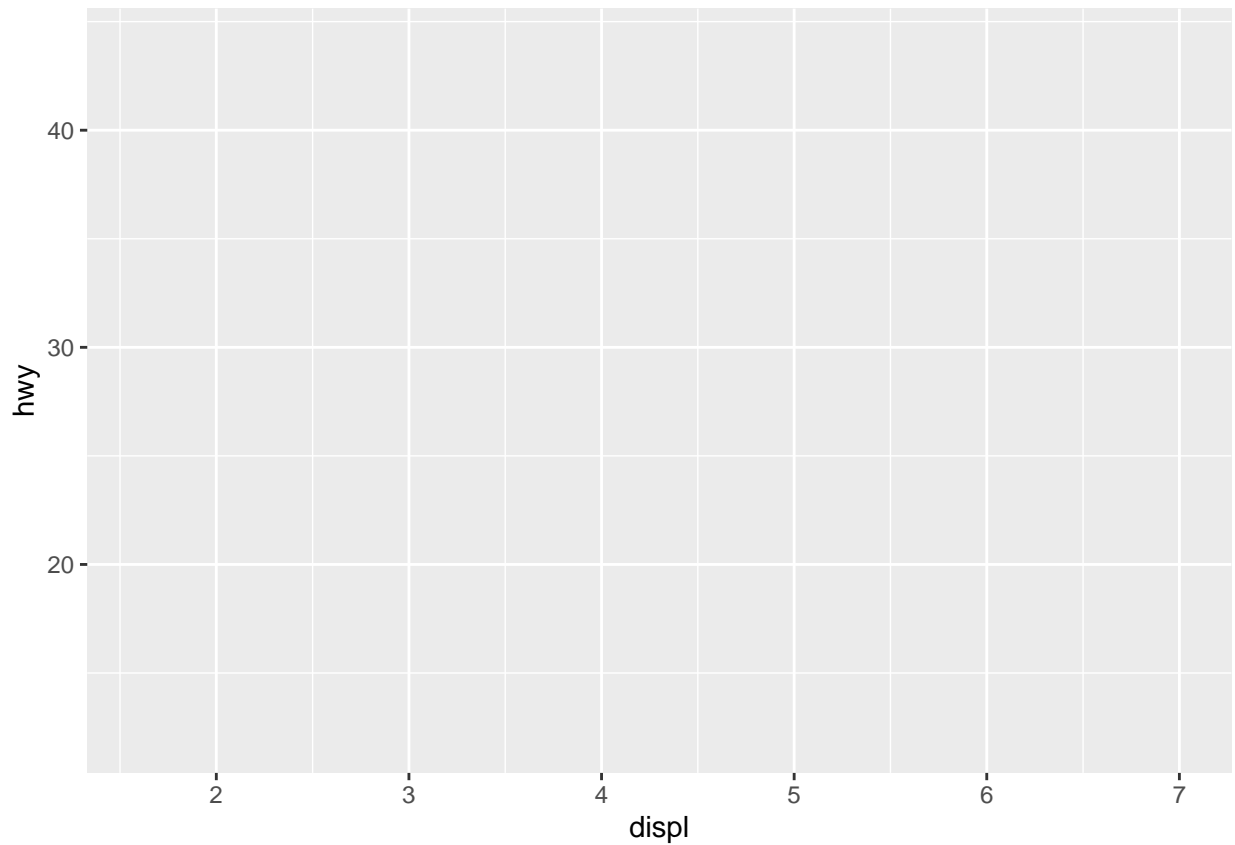
1.6 Adding geometric objects

Use `+ geom_` to add layers of geometric objects to your plot

Some common types:

- `geom_point`
- `geom_line`
- `geom_area`
- `geom_histogram`
- `geom_boxplot`

For a more comprehensive list, see the ggplot2 cheatsheet

Example using `geom_point`:

```
ggplot(data=mpg, aes(x = displ, y = hwy)) + geom_point()
```
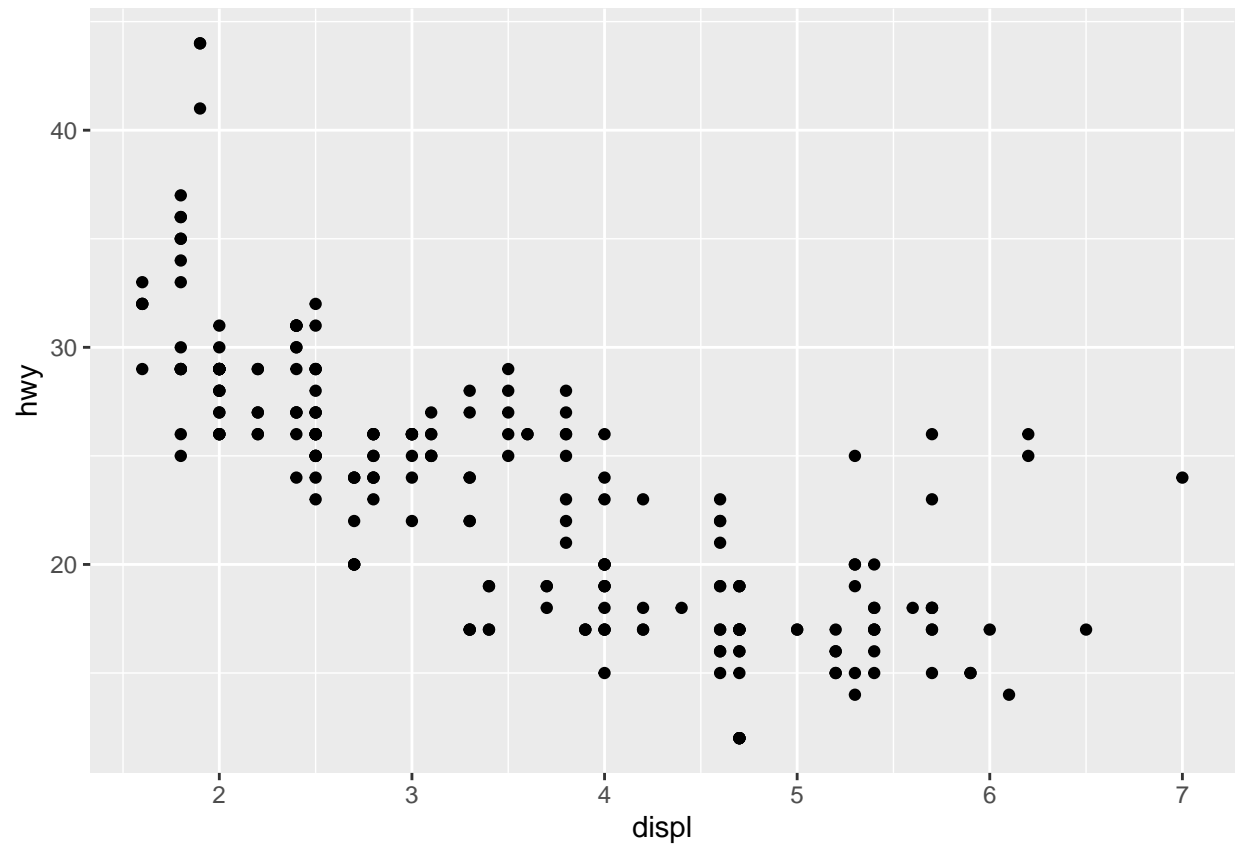
1.6.1 Adding multiple geometric markers

```
ggplot(data=mpg, aes(x = displ, y = hwy)) +
  geom_point() + geom_hline(yintercept = 30)
```
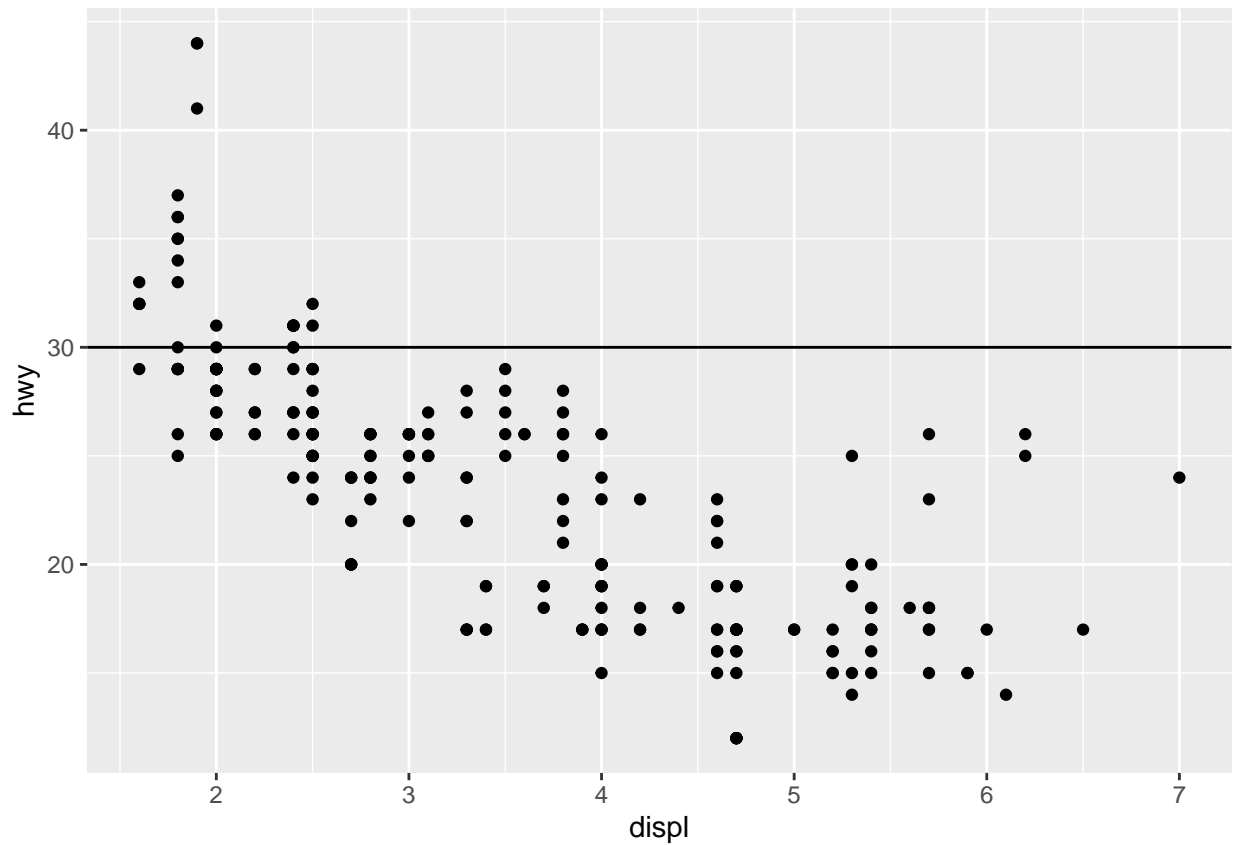
1.7   Adding aesthetic mappings

You may also use `aes()` to map a variable to:

- `alpha` (opacity)
- `color` (for points)
- `fill` (for shapes)
- `size`
- `shape`

Point shape mapped to `cyl`:

```r
ggplot(data=mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(shape = factor(cyl))) + geom_hline(yintercept = 30)
```
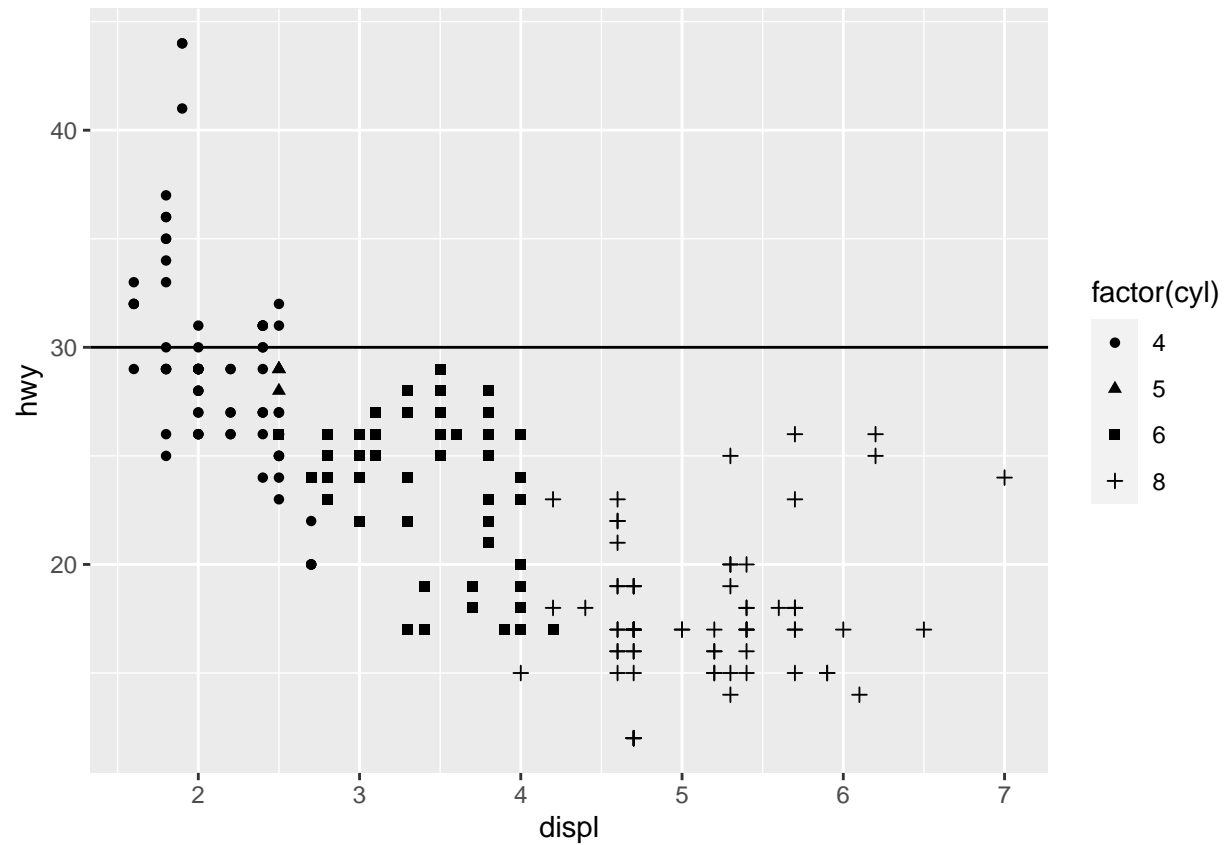
Point color mapped to `cyl`:

```
ggplot(data=mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = factor(cyl))) + geom_hline(yintercept = 30)
```
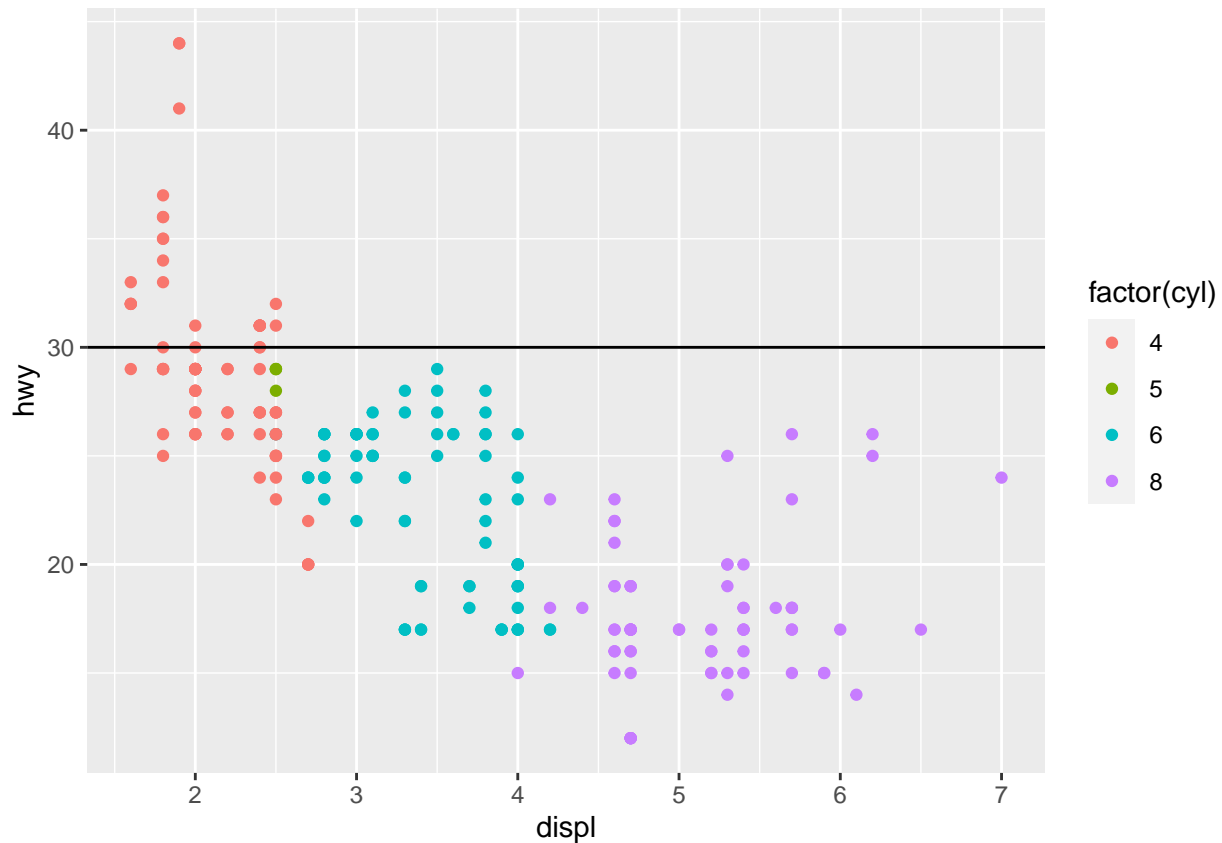
1.8 Changing Scales

Modify the scale of any of the previously mentioned mappings using the `scale_*_*`

Examples:

- `scale_color_continuous()`
- `scale_size_discrete()`
- `scale_alpha_identity()`- data values as visual values
- `scale_shape_manual()`

```
mpgscatter<- ggplot(data=mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = factor(cyl))) + geom_hline(yintercept = 30)
mpgscatter + scale_color_manual(values = c('red', 'blue', 'green', 'black'))
```
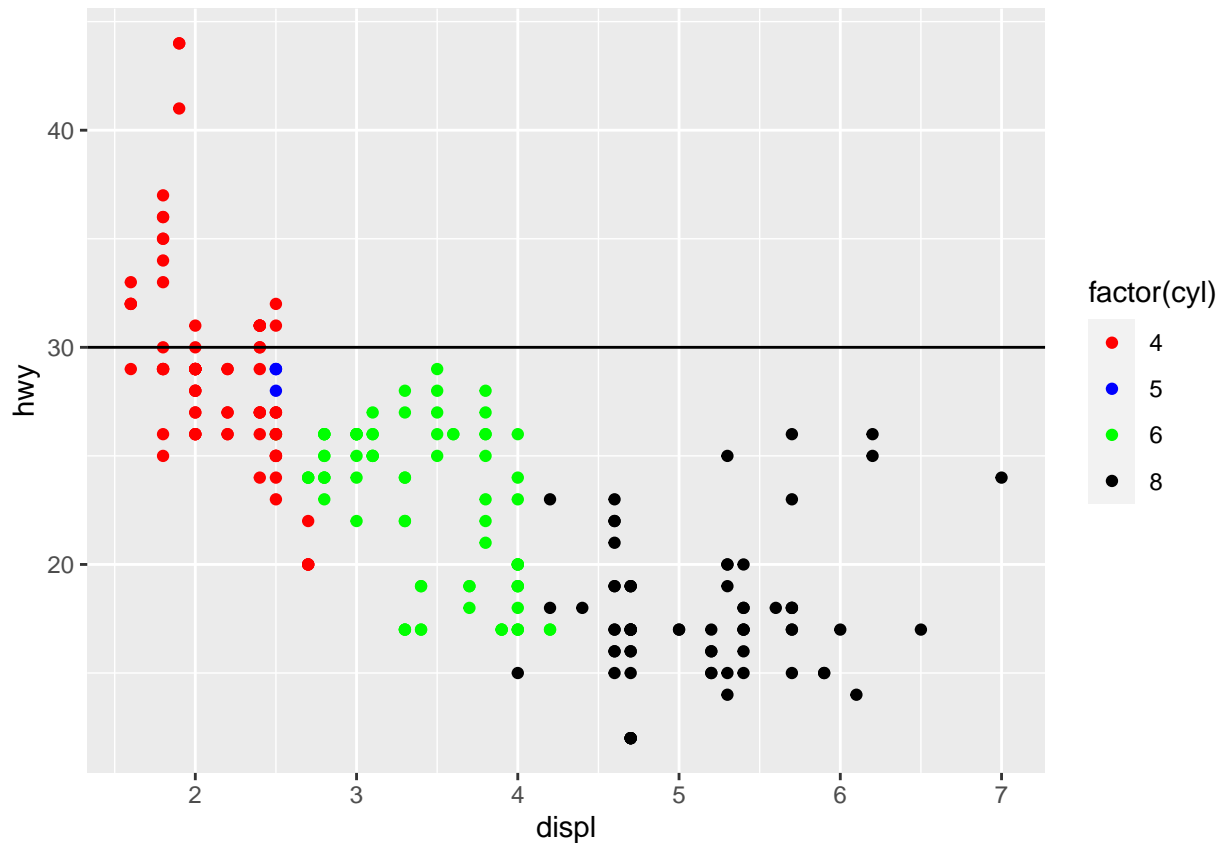
See the ggplot2 online documentation for more details.

See the ggplot2 colors for more color options

See the colorbrewer2 webapp for palletes and more.

1.9   Preset themes

Use the `theme_*` functions to change the complete theme of the plot.

Examples:

- `theme_grey()`
- `theme_classic()`
- `theme_bw()`
- `theme_minimal()`
- `theme_void()`

```
mpgscatter + scale_color_manual(values = c('red', 'blue', 'green', 'black')) +
  theme_bw()
```
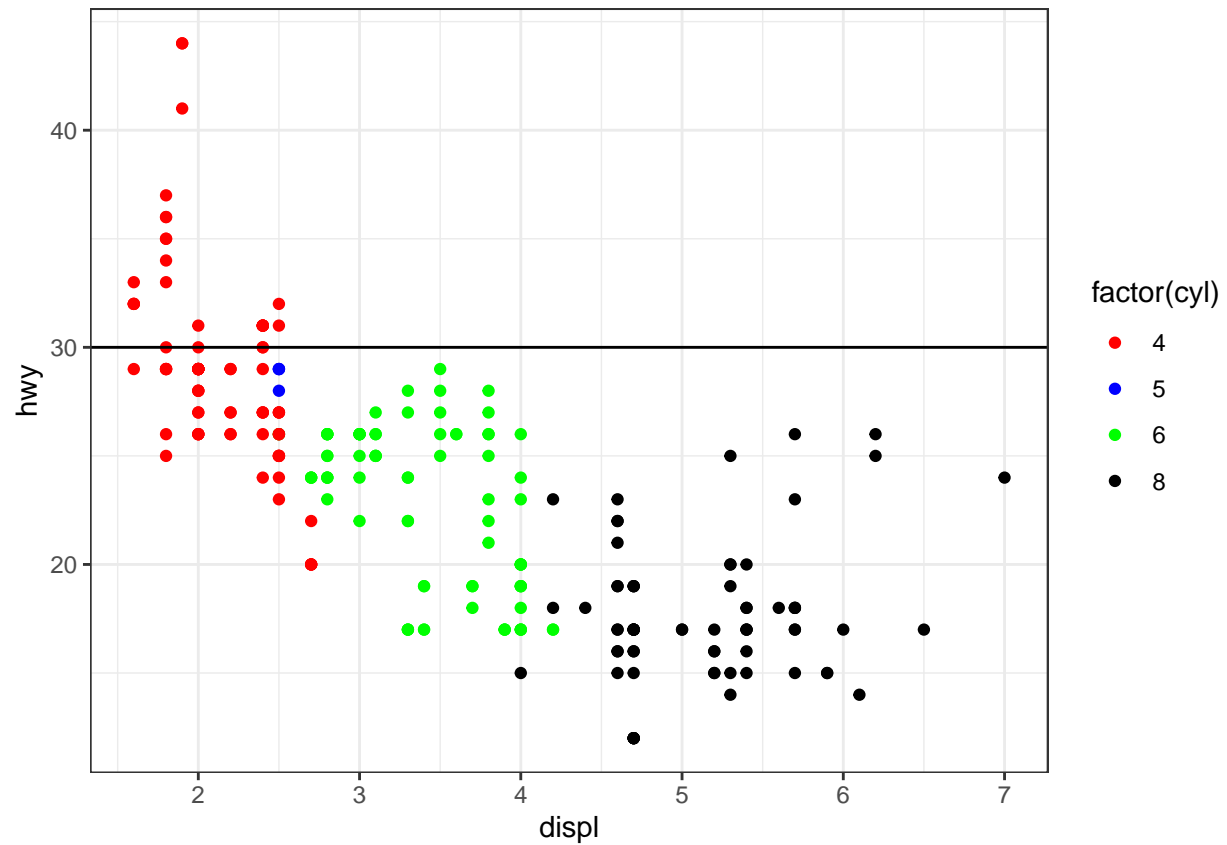
1.9.1   Base size

The base size of theme text elements can be controlled by `base_size`

```
mpgscatter + scale_color_manual(values = c('red', 'blue', 'green', 'black')) +
  theme_bw(base_size=18)
```
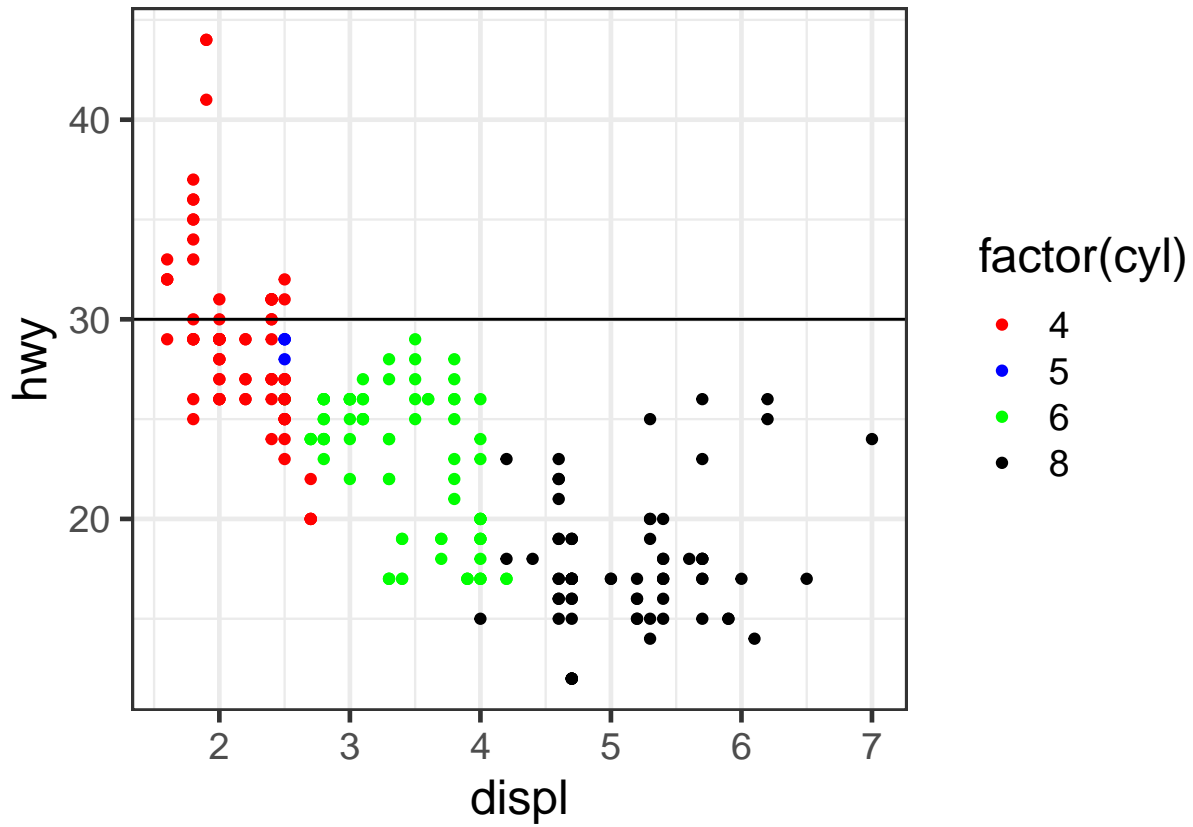
1.10 Labels

Use `labs()` to control the plot title, subtitle, x and y axis titles, and any legend titles.

```
mpgscatter +
  scale_color_manual(values = c('red', 'blue', 'green', 'black')) +
  theme_bw(base_size=18) +
  labs(title = "Fuel Economy",
       subtitle = "1998 and 2008 Vehicles",  x="Displacement",
       y = "MPG - Highway", color= "Cylinders")
```

Fuel Economy

1998 and 2008 Vehicles
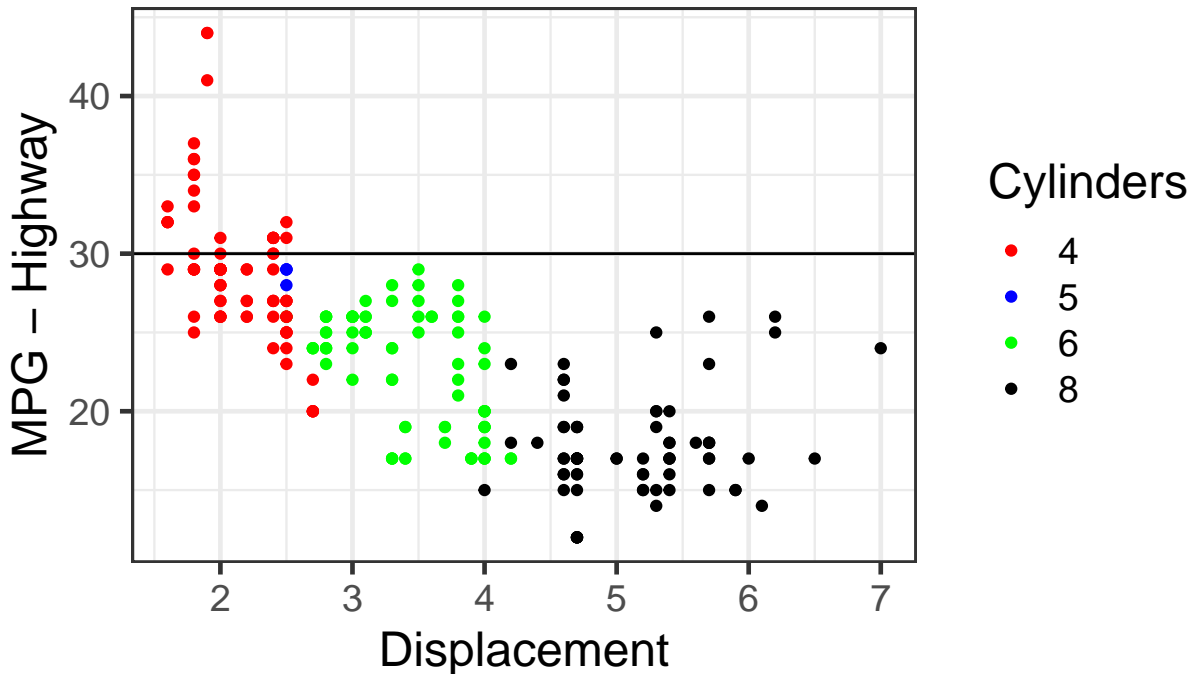
Note that the legend title is named by its mapping. It is possible to have multiple legends on a single plot.

1.11   Adjusting elements

Use `theme()` to change any individual element's size, position, color, etc.

The plot title and subtitle are left aligned, so we can use `element_text()` to center them

```
mpgscompl<- mpgscatter +
  scale_color_manual(values = c('red', 'blue', 'green', 'black')) +
  theme_bw(base_size=18) +
  labs(title = "Fuel Economy",subtitle = "1998 and 2008 Vehicles",
       x="Displacement", y = "MPG - Highway", color= "Cylinders") +
  theme(plot.title = element_text(hjust=0.5), plot.subtitle = element_text(hjust=0.5))
mpgscompl
```

Fuel Economy

1998 and 2008 Vehicles
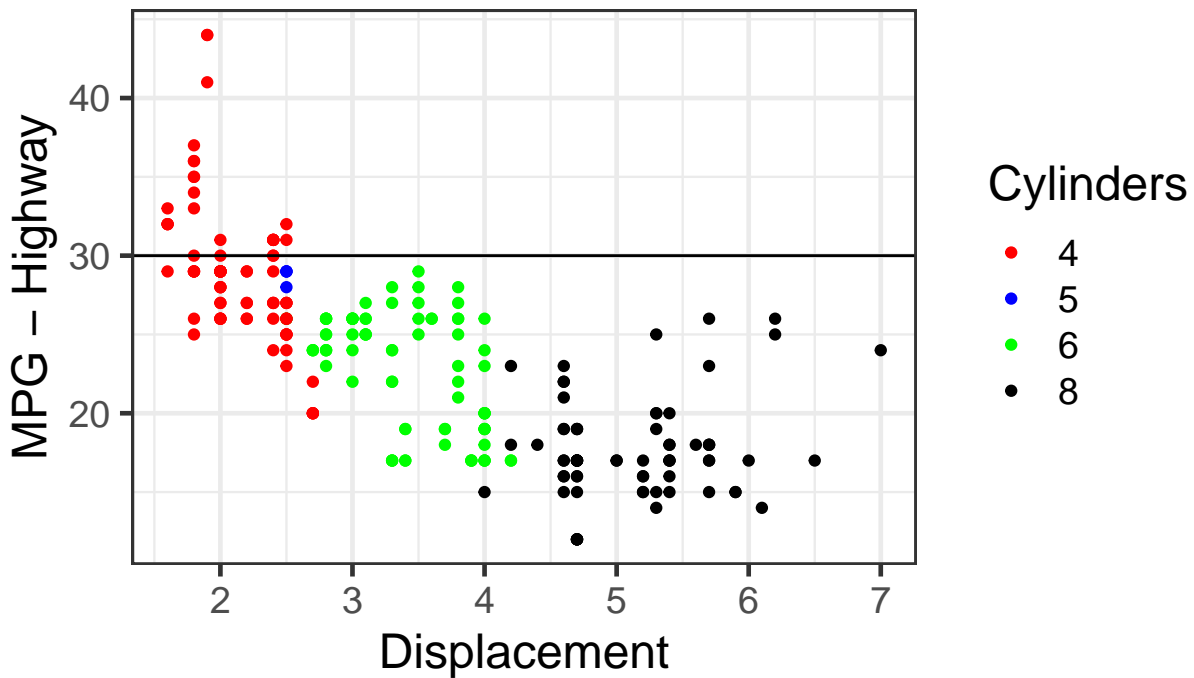
1.12 Facets

Use `facet_wrap()` or `facet_grid()` to facet plots.

`facet_wrap()` creates rectanglar layout

```
mpgscompl + facet_wrap(~fl)
```

Fuel Economy

1998 and 2008 Vehicles
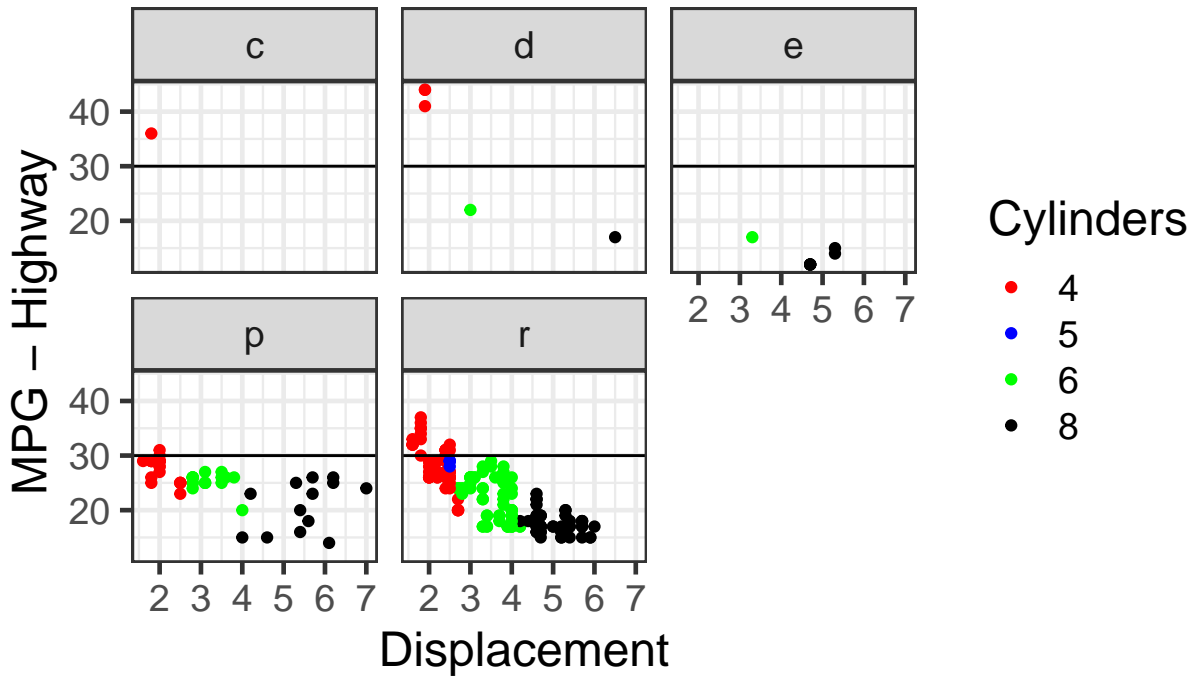
facet_grid() allows you to specify rows, columns, or both

```
mpgscompl + facet_grid(year~.)
```

Fuel Economy

1998 and 2008 Vehicles
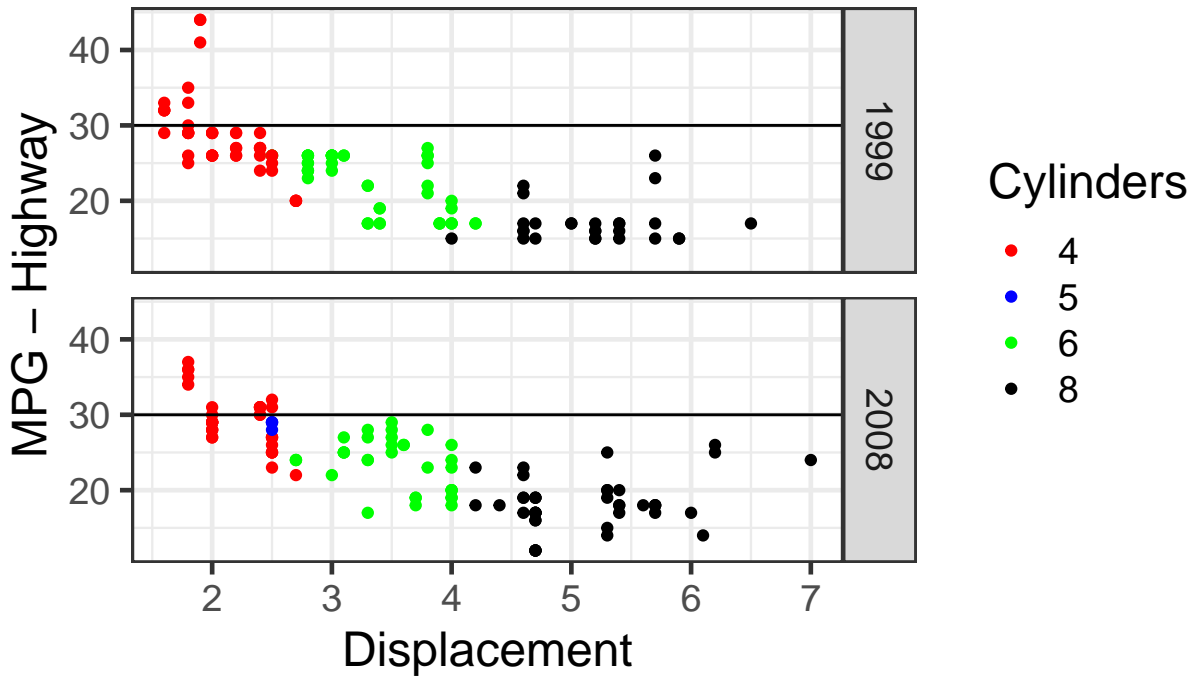
```
mpgscompl + facet_grid(year~fl)
```

Fuel Economy

1998 and 2008 Vehicles
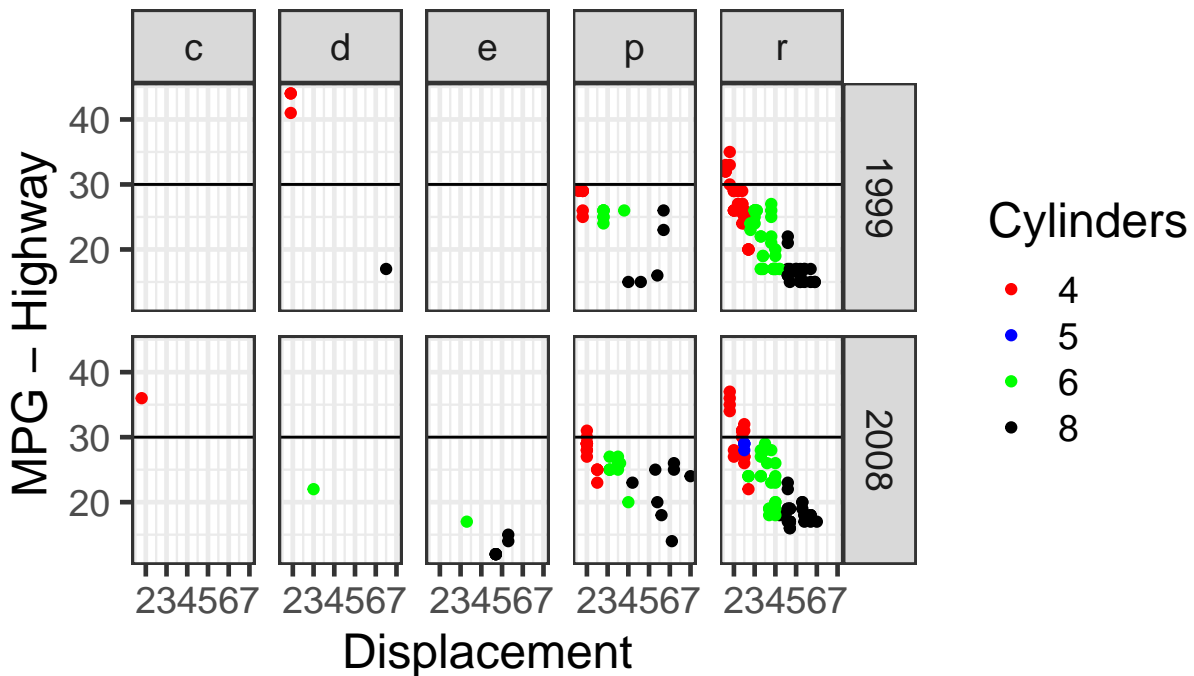
1.13   References

Gitbooks:

R Graphics Cookbook R for Data Science

Article:

A Layered Grammar of Graphics

ggplot2:

ggplot2 Cheatsheet ggplot2 Documentation

Websites:

tidyverse Hadley Wickam

1.14   Additional plots

```
cxc <- ggplot(diamonds,aes(x = clarity, fill=clarity)) +
geom_bar(width = 1)
cxc + coord_polar()
```
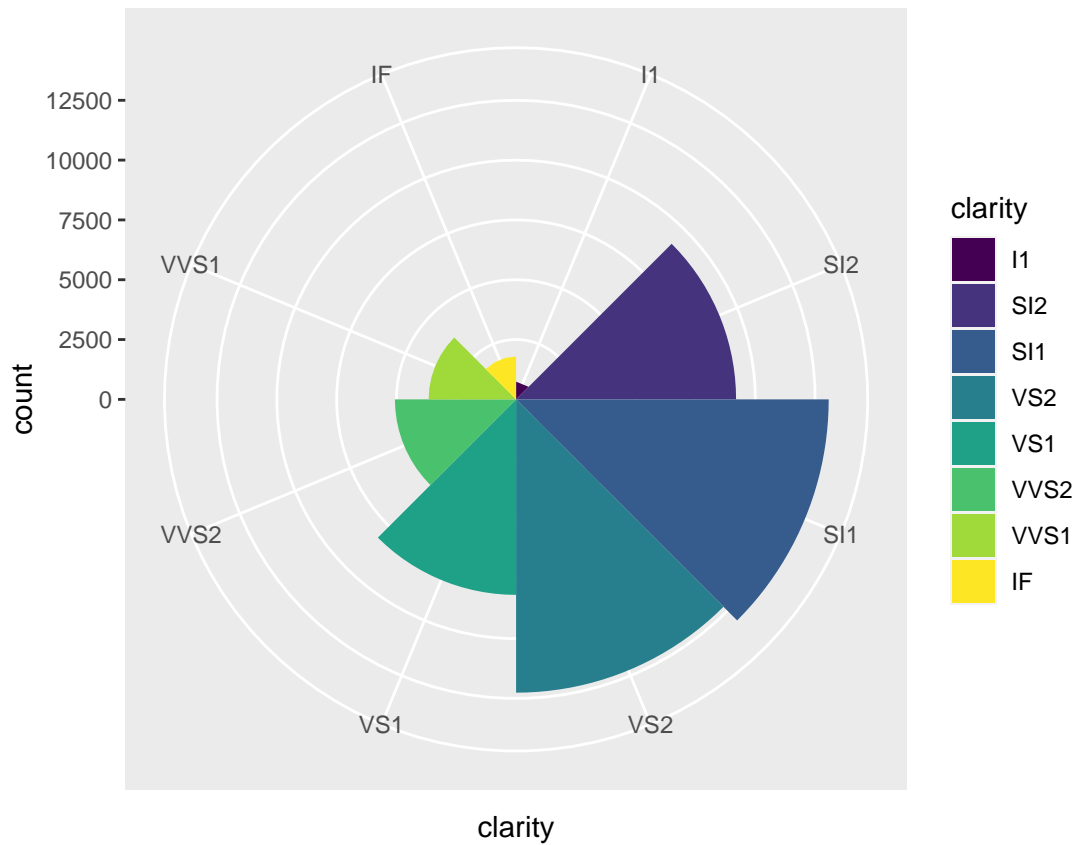
```
onevar <- ggplot(mpg, aes(x=hwy))
onevar + geom_histogram(binwidth = 2)
```
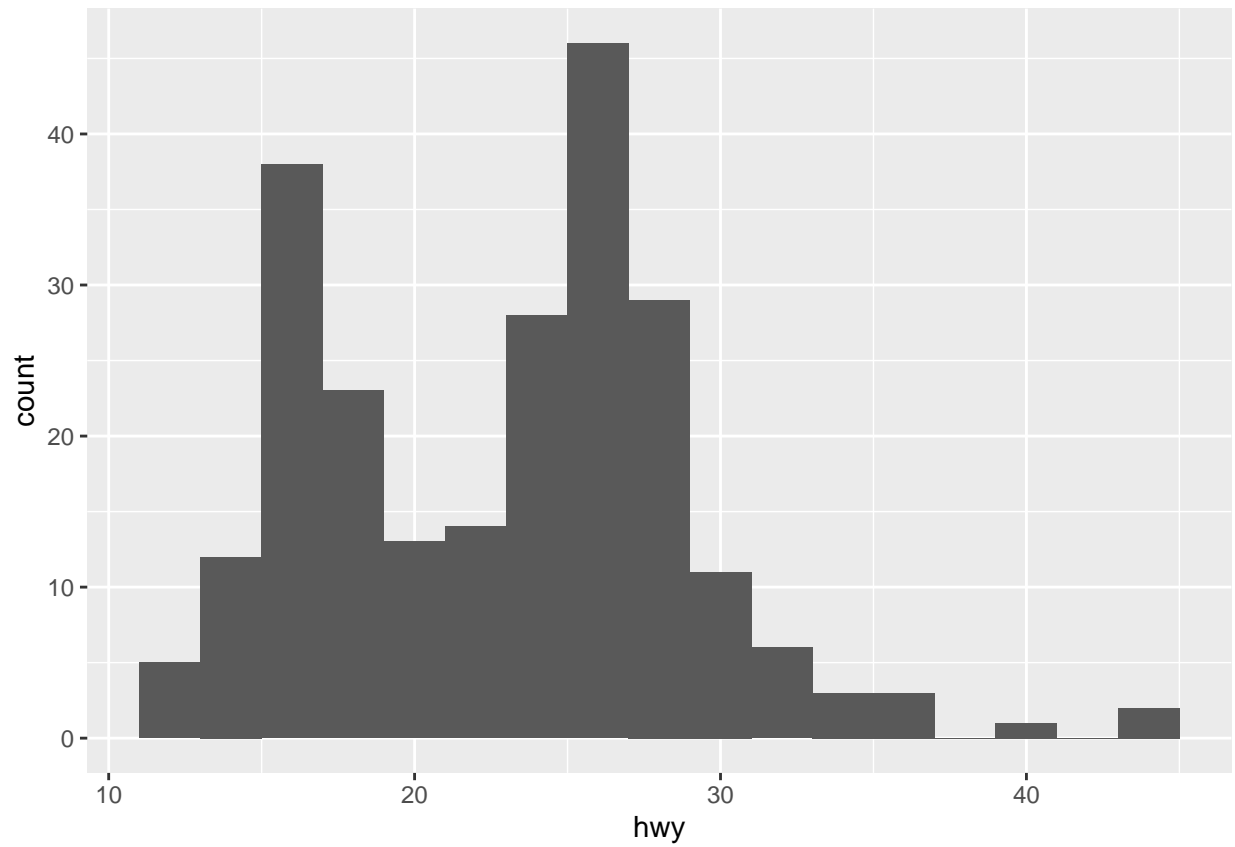
```
onevar + geom_density(kernel="gaussian")
```
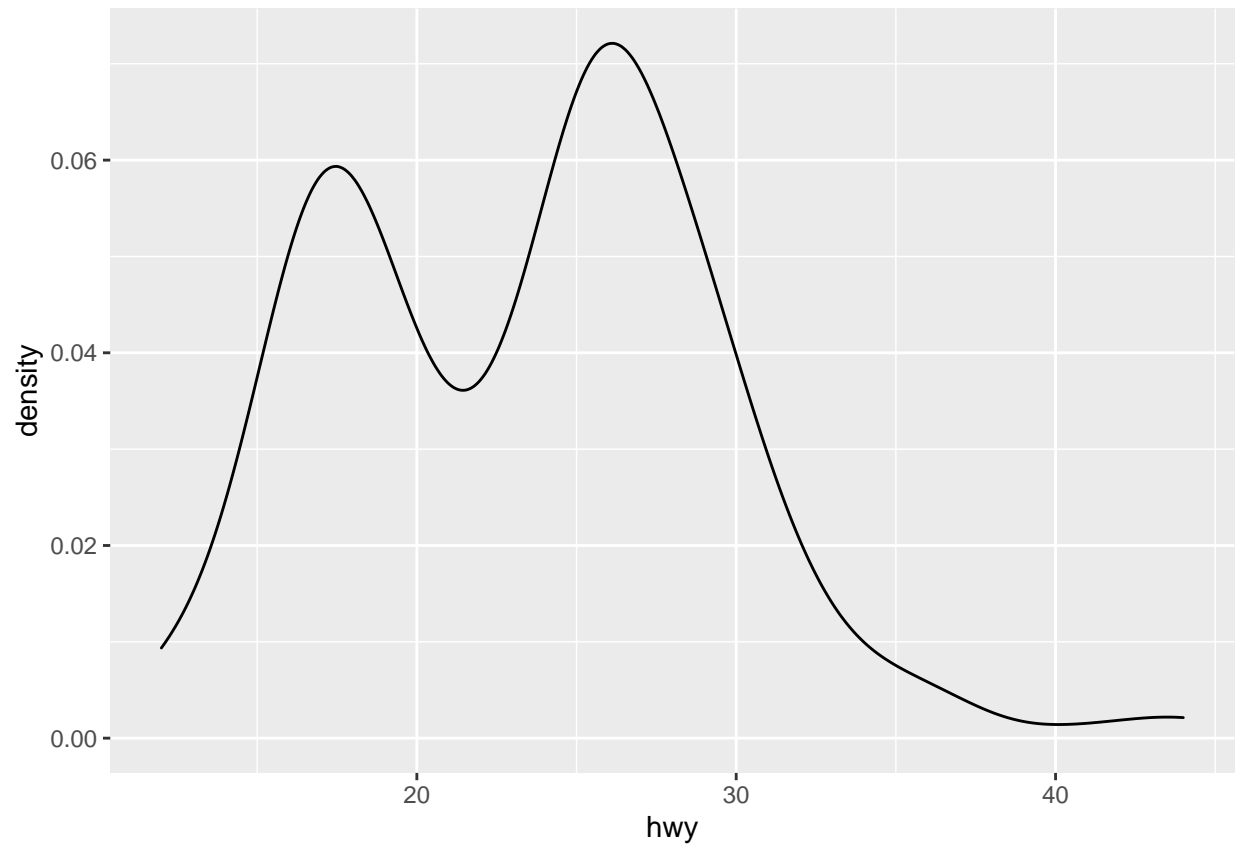
```
ggplot() + stat_function(aes(x = -3:3),
fun = dnorm, n = 101, args = list(sd=0.5))
```
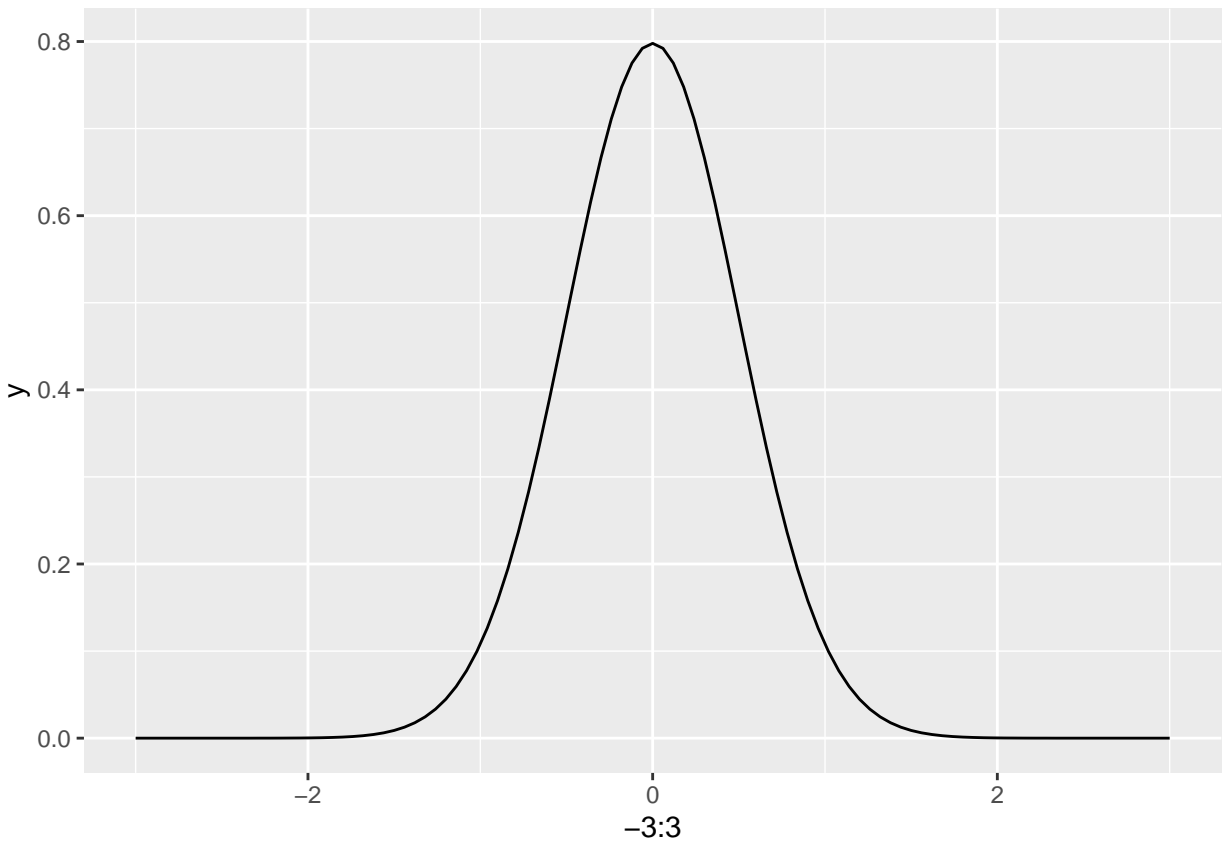
2 Interactive plots with plotly

Plotly is a collection of plotting libraries for various languages, based on the original javascript library. It includes many interactive features for charts.

Plotly for R includes a handy wrapping function for `ggplot()` outputs called `ggplotly`.

2.1 `ggplotly`

Simply save a `ggplot()` output to a variable. Then pass the variable to `ggplotly` for ineractive plots.

```
unemp <- ggplot(economics, aes(x=date, y=unemploy)) +
  geom_area(fill="#69b3a2", alpha=0.5) +
```

```
  geom_line(color="#69b3a2")

ggplotly(unemp)
```
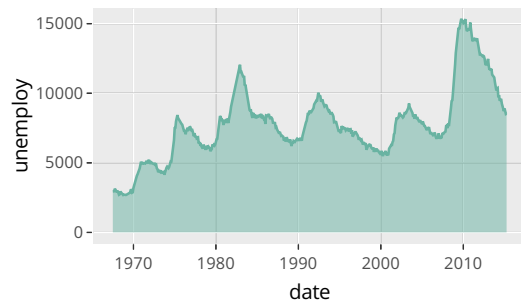

Use the `text` mapping to add a column as information in the tooltip.

```
hwy <- ggplot(data=mpg, aes(x = displ, y = hwy, text = manufacturer)) + geom_point(aes(color = factor(cy

ggplotly(hwy)
```
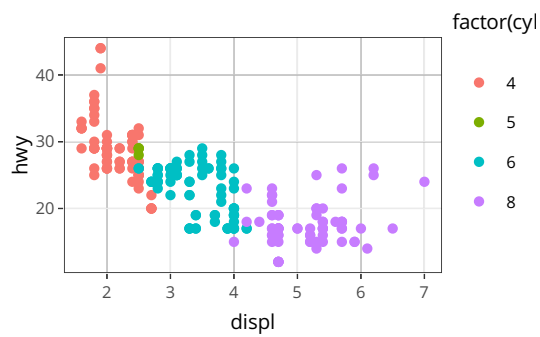
2.2 Custom Tooltips

Create a new column with all the information you want to view in the tooltip, seperated by a newline.

```r
modmpg <- mpg %>% mutate(tiptext = paste0(manufacturer, "\n", model, "\n", year, "\n", trans))

hwy <- ggplot(data=modmpg, aes(x = displ, y = hwy, text = tiptext)) + geom_point(aes(color = factor(cyl

ggplotly(hwy)
```
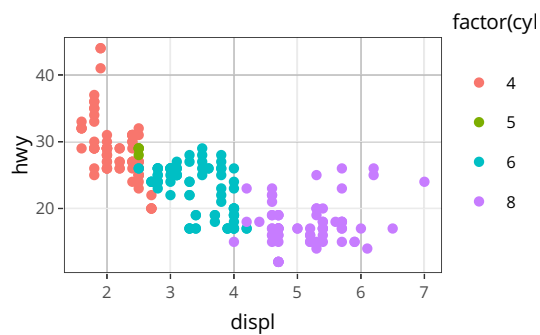
3 Maps with ggplot2

3.1 Maps available in `map_data`

Map data can come from many sources. Here we will use the package `maps`

The `map_data()` function creates a data frame of map data.

```
states <- map_data("world")
head(states)
##         long      lat group order region subregion
## 1 -69.89912 12.45200     1     1  Aruba      <NA>
## 2 -69.89571 12.42300     1     2  Aruba      <NA>
## 3 -69.94219 12.43853     1     3  Aruba      <NA>
## 4 -70.00415 12.50049     1     4  Aruba      <NA>
## 5 -70.06612 12.54697     1     5  Aruba      <NA>
## 6 -70.05088 12.59707     1     6  Aruba      <NA>
```

One way to create basic maps is to use `geom_polygon()`

World map (`"world2"`)

```
world <- map_data("world2")
ggplot() +
  geom_polygon(data=world, aes(x=long, y=lat, group=group),
                color="black", fill="lightblue" ) + coord_map() + expand_limits(x = world$long, y = wor
```
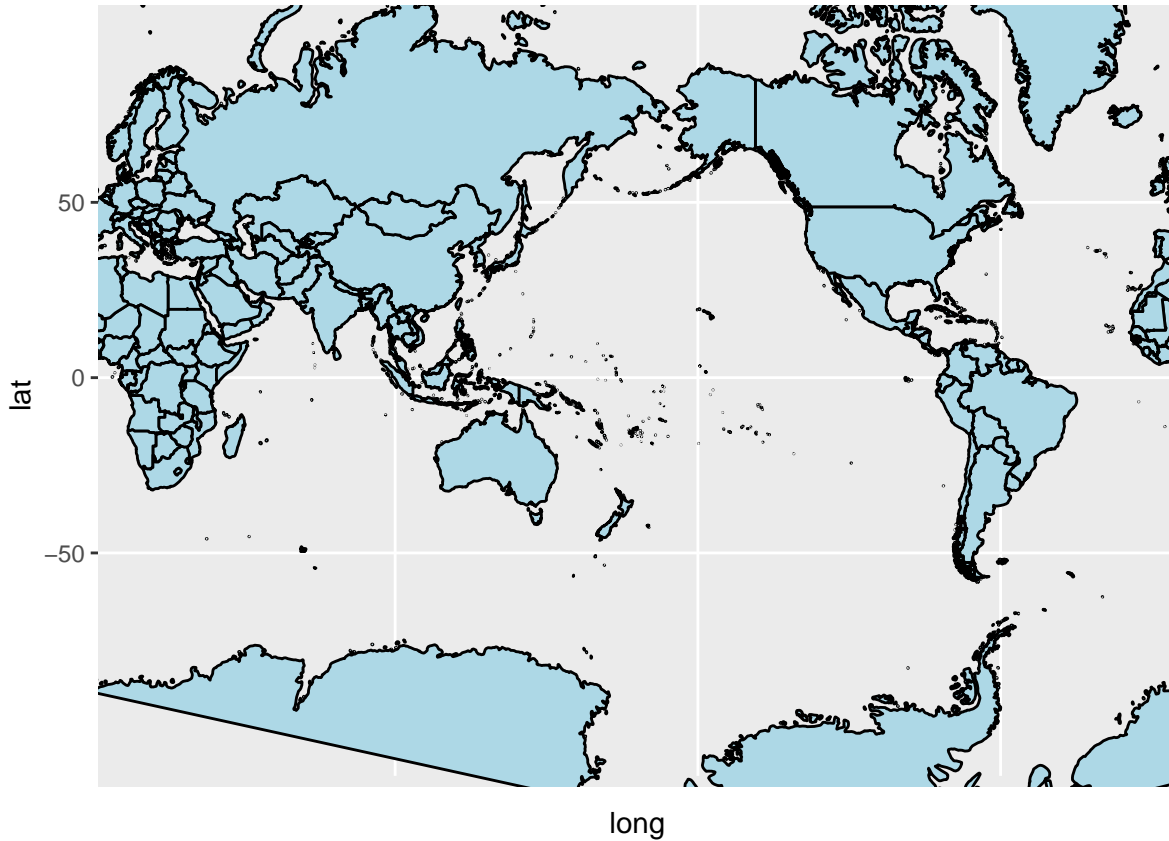

Use a list of select 'regions' to subset the world map to some european countries

```
some.eu.countries <- c(
  "Portugal", "Spain", "France", "Switzerland", "Germany",
  "Austria", "Belgium", "UK", "Netherlands",
  "Denmark", "Poland", "Italy",
  "Croatia", "Slovenia", "Hungary", "Slovakia",
  "Czech republic"
)

map <- map_data("world", region = some.eu.countries)

ggplot() +
  geom_polygon(data=map, aes(x=long, y=lat, group=group),
                color="black", fill="lightblue" ) + coord_map()
```
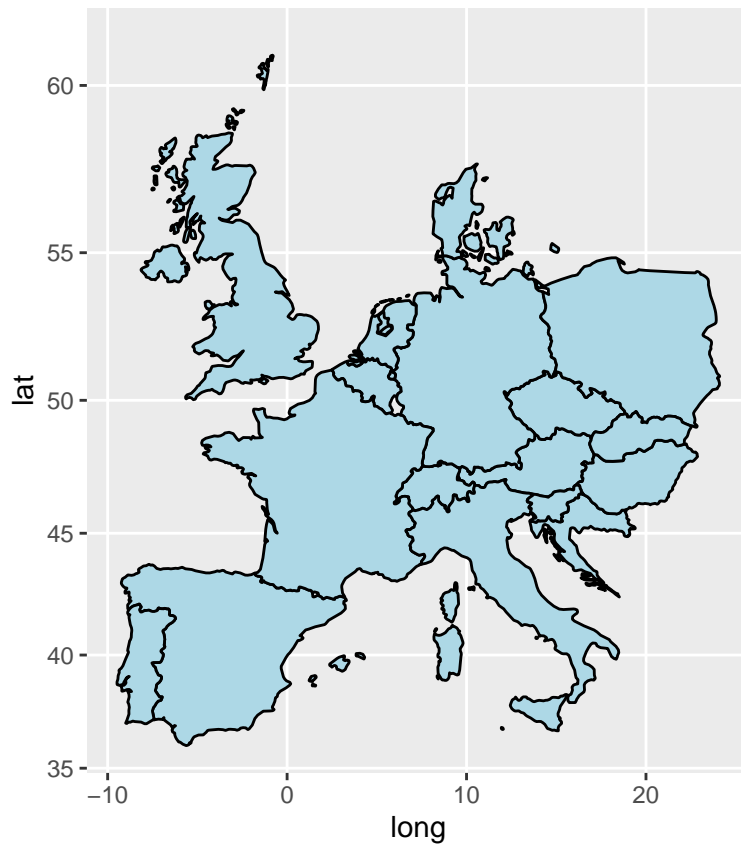
A full list of the available maps can be found in the `maps` documentation

3.2   Projections

`ggplot2` uses the projections from the package `mapproj`.

The default projection is `"mercator"`
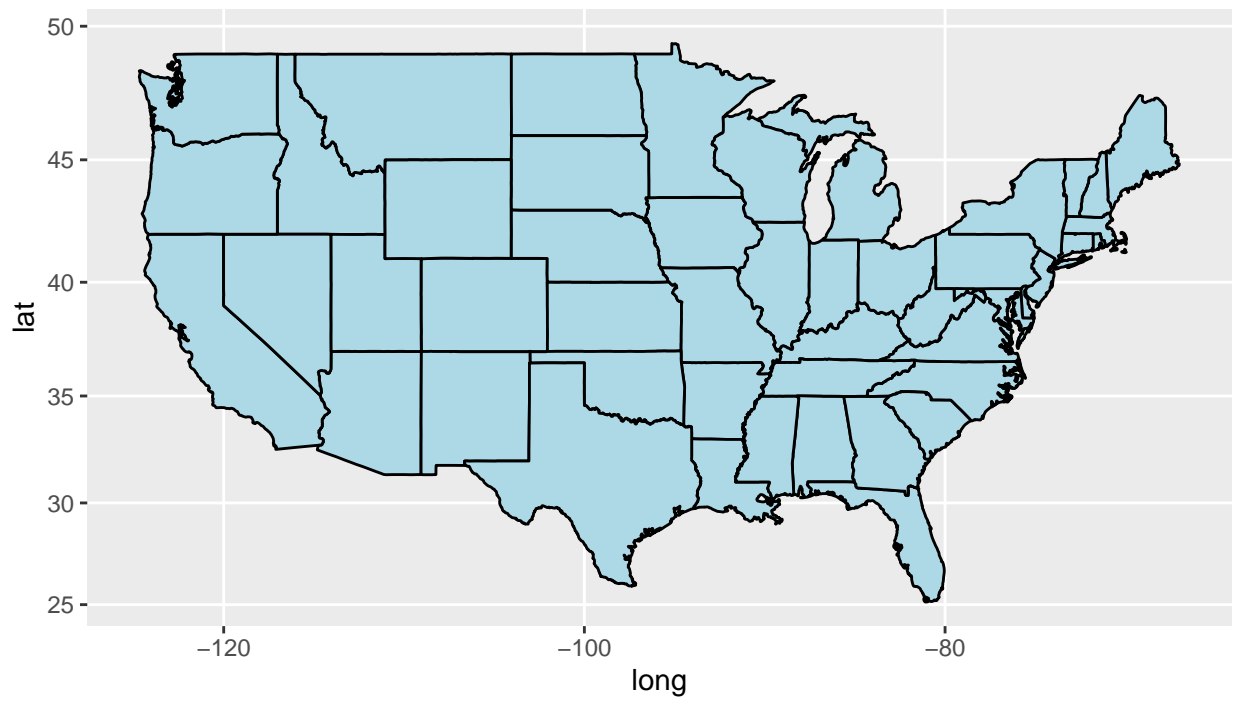
Projection is controlled by the `projection` argument to the `coord_map()` function.

```
states <- map_data("state")
ggplot() +
  geom_polygon(data=states, aes(x=long, y=lat, group=group),
               color="black", fill="lightblue" ) + coord_map(projection = "sinusoidal")
```
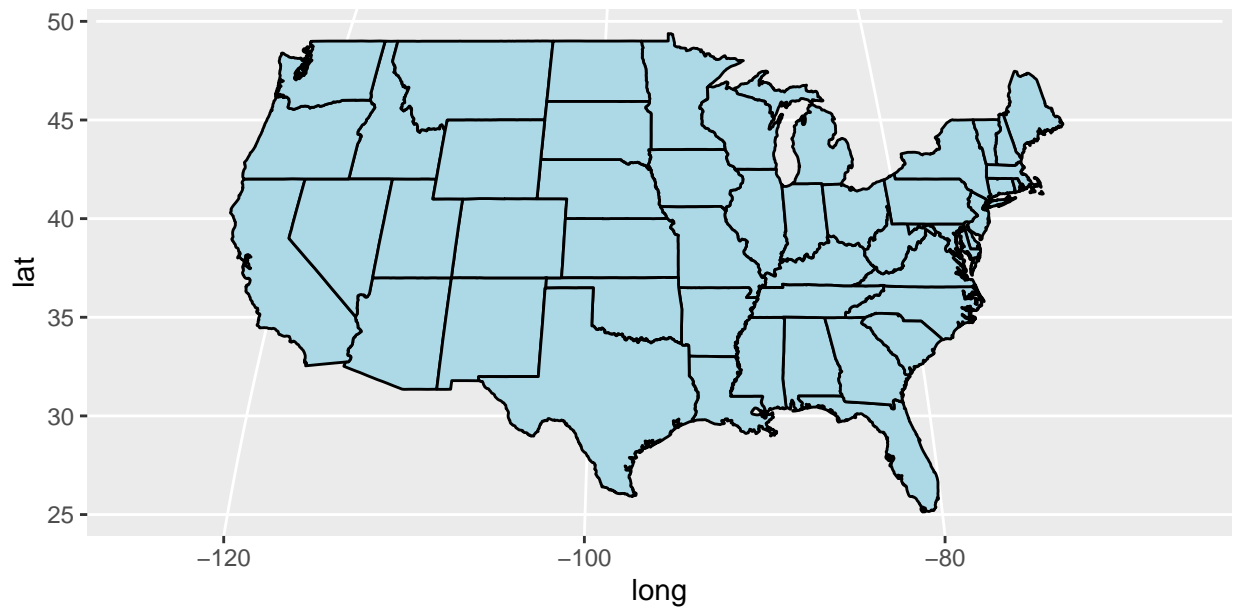
Cylindrical projection:
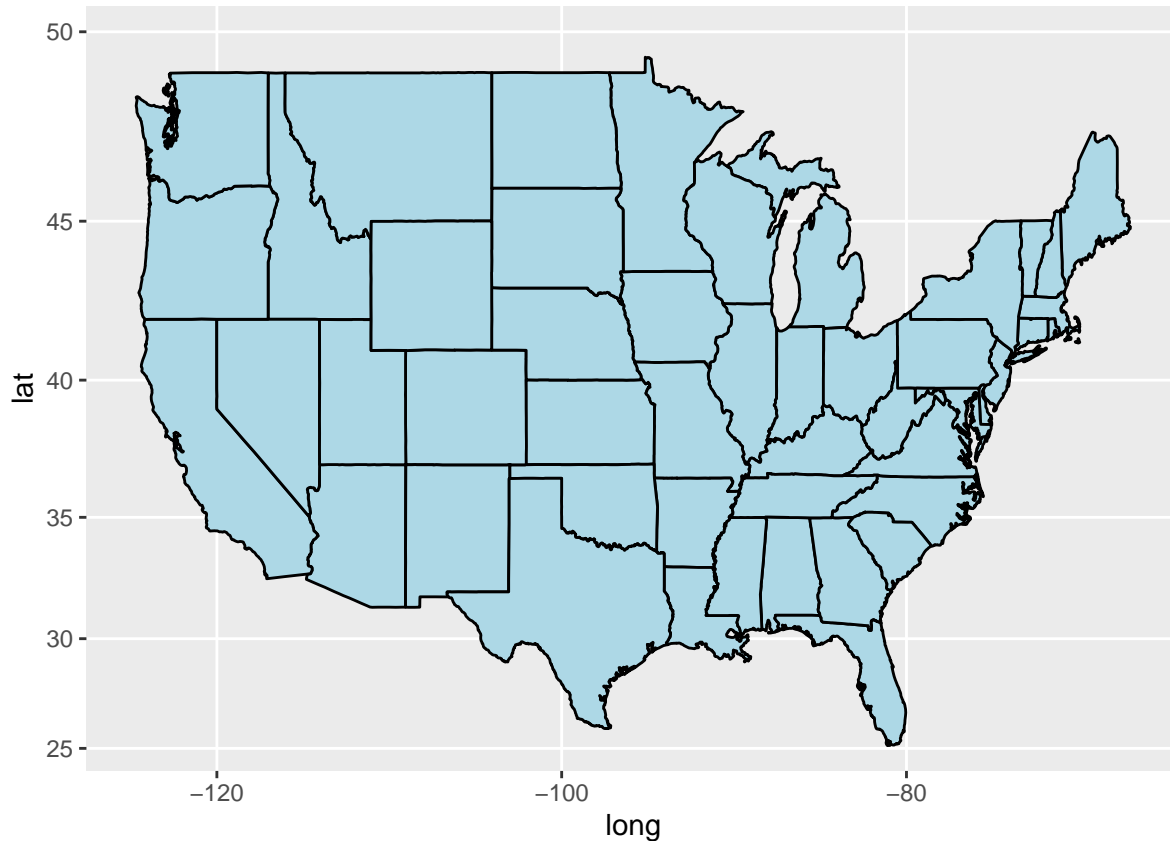
A full list and descriptions of the projections available can be found in the `mapproj` documentation.

3.3   Choropleth maps

Another method for creating maps is `geom_map`.

Starting with the data from the `USArrests` dataset, and the `"state"` map:

```
data <- data.frame(murder = USArrests$Murder, state = tolower(rownames(USArrests)))

head(data)
##   murder       state
## 1   13.2     alabama
## 2   10.0      alaska
## 3    8.1     arizona
## 4    8.8    arkansas
## 5    9.0  california
## 6    7.9    colorado

map <- map_data("state")
```

Use the `map_id=state` mapping to link the map to data by state name.

```
l <- ggplot(data, aes(fill = murder))
l + geom_map(aes(map_id = state), map = map) +
expand_limits(x = map$long, y = map$lat) + theme_void() + coord_map()
```
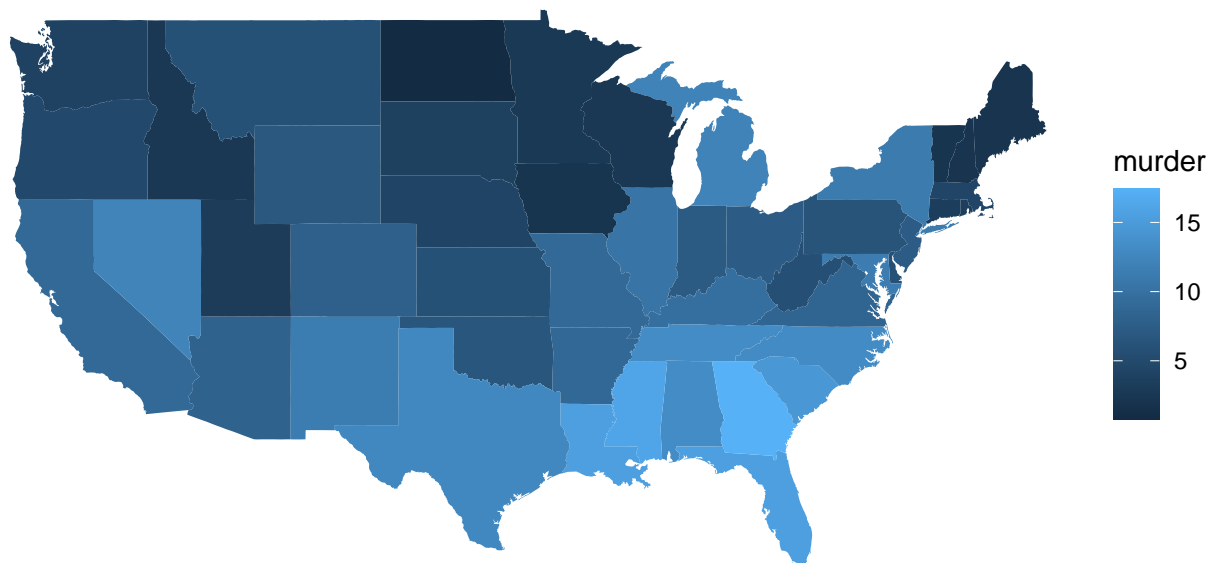
3.4 Bubble maps

Start by defining a map and the population data by city for the region of interest.

```
sl <- map_data("world") %>% filter(region=="Slovenia")

data <- world.cities %>% filter(country.etc=="Slovenia")
```

Use `geom_polygon` to draw the country, and `geom_point` for the 'bubbles' at the city coordinates.

```
ggplot() +
  geom_polygon(data = sl, aes(x=long, y = lat, group = group), fill="grey", alpha=0.3) +
  geom_point(data=data, aes(x=long, y=lat)) +
  theme_void() + coord_map()
```

Scale the size of the bubbles according to the city population.

```
ggplot(data) +
    geom_polygon(data = sl, aes(x=long, y = lat, group = group), fill="grey", alpha=0.3) +
    geom_point( aes(x=long, y=lat, size=pop)) +
    scale_size_continuous(range=c(1,12)) +
    theme_void() + coord_map()
```
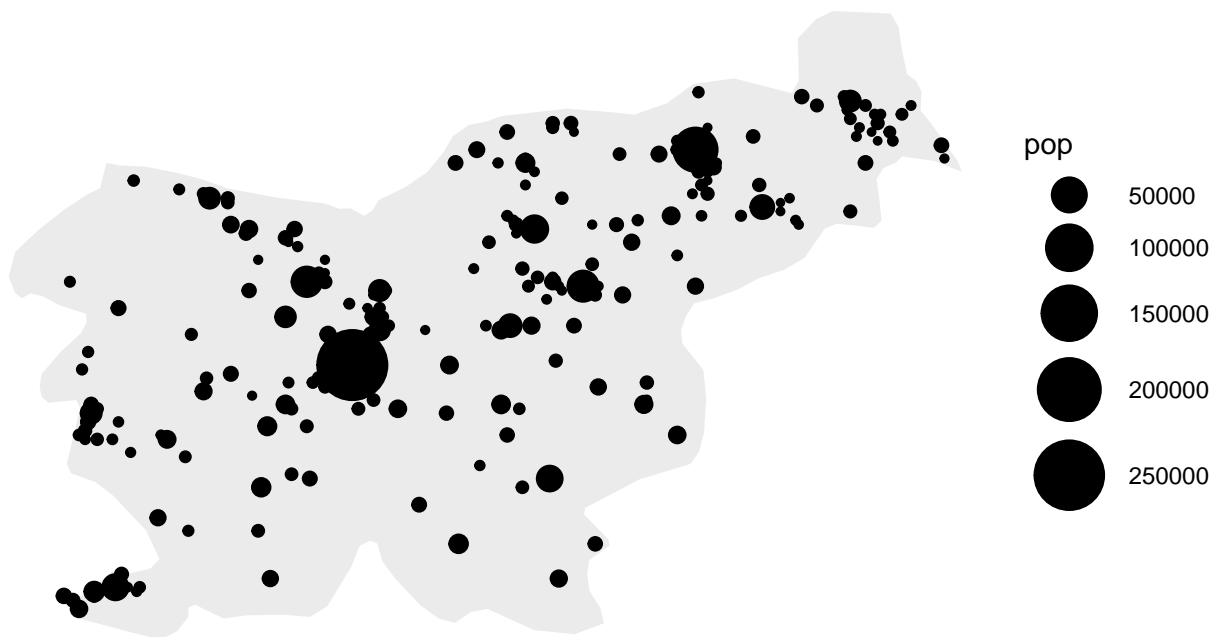
Add a color gradient scale.

Note that the data is arranged in descending order first, so that smaller bubbles are not obscured by larger ones.

```
data %>%
 arrange(desc(pop)) %>%
 mutate( name=factor(name, unique(name))) %>%
 ggplot() +
    geom_polygon(data = sl, aes(x=long, y = lat, group = group), fill="grey", alpha=0.3) +
    geom_point( aes(x=long, y=lat, size=pop, color=pop), alpha=0.9) +
    scale_size_continuous(range=c(1,12)) +
    scale_color_gradient(low="blue", high = "yellow") +
    theme_void()  + coord_map()
```
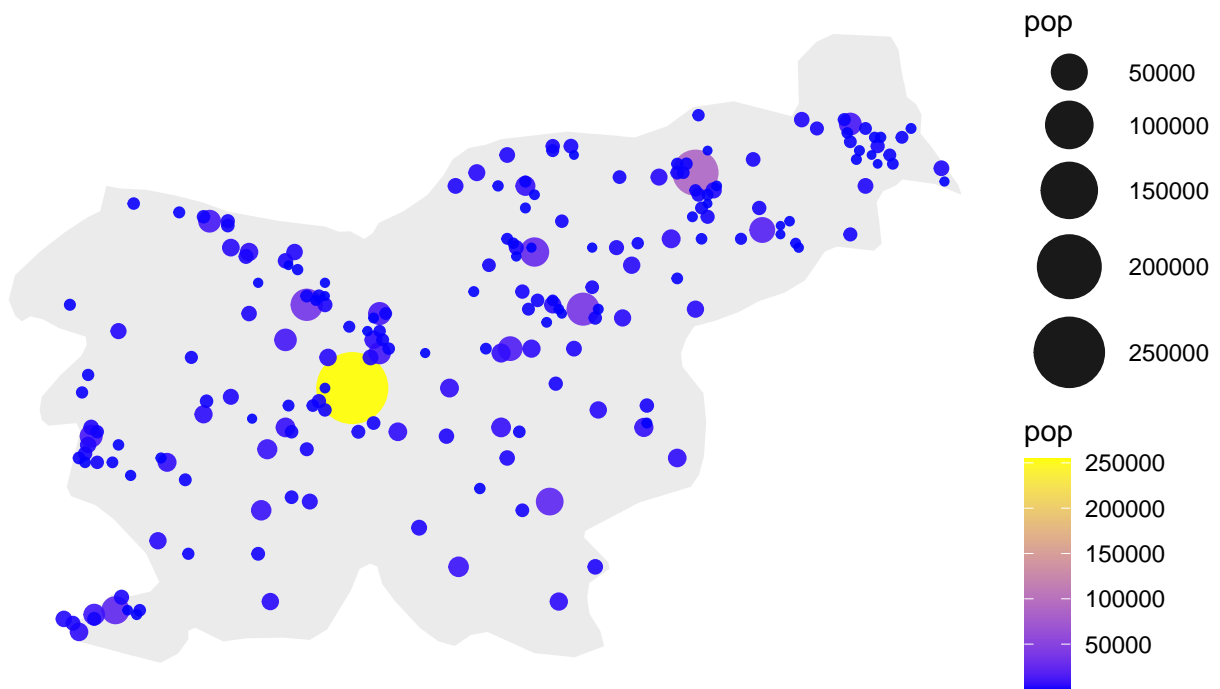
4 Shiny Apps

4.1 Why Shiny?

Say you have put together a comprehensive analysis of some data with R code. You probably have functions to processes your data, functions to create charts and tables, maybe even a report in R markdown that calls some of your code and creates a PDF. But what you'd really like is an app for that, maybe so others (your CO/boss/users/clients) can view and interact with the data themselves, without needing to know or even have R. That's what Shiny is for.

In the past, making an app was hard for R users because you needed knowledge of web technologies (HTML, CSS and Javascript), as well as a background in User Interface (UI) programming. Shiny reduces this burden by handling the specifics of the web technologies and providing a **reactive** programming framework that simplifies the complexity and quantity of code needed to define the rules of the app.

Here are just a few uses for shiny:

- Create dashboards to track indicators and drill down into data.
- Give users the ability to jump straight to the results they care about and avoid flipping through long report appendices of charts and tables.
- Allow users to apply an analysis to their own data without needing to have or know R.
- Teach concepts with interactive demos, allowing students to adjust parameters and view the down-stream effects.

4.2 A first Shiny app

4.3 Introduction

The goal of this session will be to create a simple Shiny app. We'll go through the minimum boilerplate code needed for an app, the distinction between UI and server components, and how to start and stop it.

Then we'll get into **reactive** expressions, the most important, and powerful, concept in Shiny.

Make sure Shiny is installed, it's a package like any other in R:

```r
install.packages("shiny")
```

and make sure it's loaded in your current R session:

```r
library(shiny)
```

4.4 Create app directory and file

The simplest way to create a shiny app is to create a folder and put a single file called `app.R` in it.

Try it out, and add the following code to the `app.R` file:

```r
library(shiny)
ui <- fluidPage(
  "Hello, world!"
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

RStudio Tip: You can do this automatically through the **File | New Project** menu, then select "New Directory" and "Shiny Web Application". The boilerplate code is added automatically.

That's it! It won't *do* much, but running this code will run a shiny app. Notice a few things:

1. `library(shiny)` loads the shiny package.

2. The code creates a `ui` object, which shiny will translate into an HTML webpage, one that just says "Hello, world!" for now.

3. It also creates a `server` object, specifically a function. In this case our server doesn't *do* anything (it's just a blank function), but in useful apps this will be where all the rules for the behavior of the app will go.

4. It calls `shinyApp(ui, server)` to start the app. A shiny app always requires a UI object and a server object.

4.5 Running and stopping

There are a few ways you can run this app:

- From RStudio, click the **Run App** button in the document toolbar.
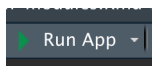
- From RStudio, use a keyboard shortcut: `Cmd/Ctrl` + `Shift` + `Enter`.

- From an R command prompt, you can `source()` the whole document, or call `shiny::runApp()` with the path to the directory containing `app.R`.

Choose whichever method you like and check that you see the same app as in Figure 1. Congratulations! You've made your first Shiny app.
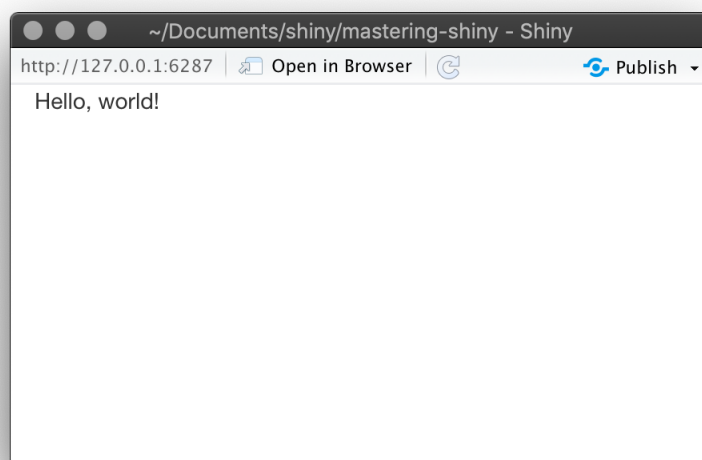
Figure 1: The very basic shiny app you'll see when you run the code above

You'll also notice in the R console that it says something like:

```
#> Listening on http://127.0.0.1:3694
```

This shows us a few things. Shiny has actually started a web server for us on our local computer. `127.0.0.1` is a standard address that means "this computer", and 3694 (you will likely see something different) is a randomly assigned port number. In fact, you can enter this URL into any reasonably modern web browser to open another copy of your app.

Notice that R stays busy, the R prompt isn't visible. A running Shiny app will "block" the R console, precisely because it needs to listen and respond to input from the UI.

There are a few ways to stop and return access to the console: clicking the stop sign icon, pressing `Esc` or `Ctrl` + 'C** in the console, or closing the Shiny app window.

The basic workflow for Shiny app development is to write some code, start the app, experiment, stop the app, and repeat.

4.6   Adding UI controls

Next, we'll add some inputs and outputs to the UI. We'll make a very simple app that shows you all the data files that accompany this course.

Replace your `ui` with this code:

```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices = list.files("./data")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)
```

This example uses four new functions:

- `fluidPage()` is a **layout function** that sets up the basic visual structure of the page.

- `selectInput()` is an **input control** that lets the user provide a value. In this case, it's a select box with the label "Dataset" and lets you choose one of the data files that accompany this course.

- `verbatimTextOutput()` and `tableOutput()` are **output controls** that tell Shiny *where* to put rendered output (we'll get into the *how* in a moment). `verbatimTextOutput()` displays code and `tableOutput` displays tables.

Layout functions, inputs, and are fundamentally the same under the covers: ways to generate HTML. If you call any of them outside of a Shiny app, you'll see HTML printed out at the console.

In fact, you can try that now, just to poke under the covers:

```
verbatimTextOutput("summary")
```

```
#> <pre id="summary" class="shiny-text-output noplaceholder"></pre>
```

Now run the app again. You'll see Figure 2, a page containing a select box. We only see the input, not the two outputs, because we haven't yet told Shiny how the input and outputs are related.

Dataset

crime.csv ▼

Figure 2: The datasets app with UI

If you don't see anything in the inputs, check that the path to the data folder, `"./data"` in the above code, is correct for your setup. The path to the data files is relative to the current working directory, `getcwd()`.

4.7   Adding behavior

Next, we'll bring the outputs to life by defining them in the server function.

Shiny uses reactive programming to make apps interactive, as do an increasing number web frameworks such as Elm, React (unsurprisingly), and Angular (which uses RxJS). Keep in mind that reactive programming means we tell Shiny *how* to perform a computation, but not to actually *do it*, which would be the more traditional *imperative* style of programming.

In this simple case, we're going to tell Shiny how to fill in the `summary` and `table` outputs—we're providing the "recipes" for those outputs. Replace your empty `server` function with this:

```
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- read.csv(file.path("./data", input$dataset))
    summary(dataset)
  })

  output$table <- renderTable({
    dataset <- read.csv(file.path("./data", input$dataset))
    dataset
  })
}
```

Almost every output you'll write in Shiny will follow this same pattern:

```
output$ID <- renderTYPE({
  # Expression that generates whatever kind of output
  # renderTYPE expects
})
```

The left-hand side of the assignment operator (`<-`), `output$ID`, indicates that you're providing the recipe for the Shiny output with the matching ID. The right-hand side of the assignment uses a specific **render function** to wrap some code that you provide; in the example above, we use `renderPrint()` and `renderTable()`.

Each `render*` function in the `server` works with a particular type of output that's passed to an `*Output` function in the `ui`. In this case, we're using `renderPrint()` to capture and display a statistical summary of the data with fixed-width (verbatim) text, and `renderTable()` to display the actual data frame in a table.

Run the app again and play around, watching what happens to the output when you change an input. Figure 3 shows what you'll see when you open the app.

Dataset

crime.csv ▾

Error: cannot open the connection

Error: cannot open the connection
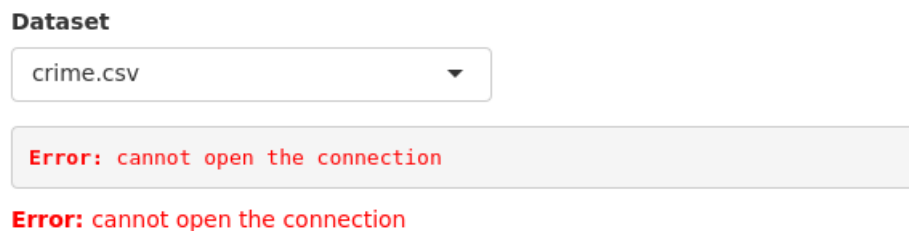
Figure 3: Now that we've provided a server function that connects and inputs, we have a fully functional app

Notice that we haven't written any explicit code that checks for changes to `input$dataset` and updates the two outputs. That's because outputs are **reactive**: *they automatically recalculate when their inputs change*. Because both of the rendering code blocks used `input$dataset`, whenever the value of `input$dataset` changes (i.e. the user changes their selection in the UI), both outputs recalculate and update in the browser.

4.8   Reducing duplication with reactive expressions

Even in this simple example, we have some code that's duplicated: the following line is present in both outputs.

```
read.csv(file.path("./data", input$dataset))
```

In every kind of programming, it's **poor practice to have duplicated code**; it can be computationally wasteful, and more importantly, it increases the difficulty of maintaining or debugging the code. It's not that important here, but this simple context illustrates the basic idea.

In traditional R scripting, we might deal with duplicated code by capturing the value using a variable, or capturing the computation with a function, but neither of these approaches work here. Shiny offers another way: **reactive expressions**.

A simple way to create a reactive expression is by wrapping a block of code in `reactive({...})` and assigning it to a variable. You then use it by calling it like a function, with `()` at the end. While it looks like you're calling a function, a reactive expression only runs the first time it is called and caches its result until it needs to be updated.

We can modify our `server()` to use reactive expressions, as shown below. The app behaves identically, but works more efficiently because it only needs to retrieve the dataset once, not twice.

```
server <- function(input, output, session) {
  dataset <- reactive({
    read.csv(file.path("./data", input$dataset))
  })

  output$summary <- renderPrint({
    summary(dataset())
  })

  output$table <- renderTable({
    dataset()
  })
}
```

4.9   Cheat sheet

A great resource to have with you as you develop your Shiny app is the Shiny cheatsheet https://www.rstudio.com/resources/cheatsheets/.

4.10 Exercises

1. Create an app that greets the user by name and rank. You may not know all the functions you need to do this yet, so some lines are included below. Figure out which lines to use and copy them into the ui and server components of a Shiny app, perhaps starting with the boilerplate code we started with.

```
textInput("name", "What's your name?")
renderText({
  paste0("Hello", input$rank, input$name, collapse = " ")
})
renderPlot("histogram", {
  hist(rnorm(1000))
})
numericInput("age", "How old are you?")
textInput("rank", "What's your rank?")
textOutput("greeting")
tableOutput("mortgage")
```

2. Suppose your meteorologist friend wants to design an app that lets the user set a slider to a temperature in degrees Fahrenheit (`fahr`) between -40 and 140, and displays the corresponding temperature in Celsius. This is their first attempt:

```
ui <- fluidPage(
  sliderInput("fahr", label = "If the temperature in ° Fahrenheit is",
              min = -58, max = 122, value = 77),
  "then the temperature in ° Celsius is ",
  textOutput("celsius")
)

server <- function(input, output, session) {
  output$celsius <- renderText({
    (fahr - 32) * 5 / 9
  })
}
```

But unfortunately it has an error:

Can you help them find and correct the error? Your chemist friend wants it to also display in Kelvin (freezing is 273.15 K), can you add another output for kelvin?

3. Extend the app from the previous exercise to allow the user to set another slider for the relative humidity (between 0 and 100) and display the heat index in degrees Fahrenheit and Celsius. The `heat_index_fahr` function is included below for computing the heat index from degrees fahrenheit and relative humidity. The code for the ui section is already implemented, you should only need to replace the lines with comments. Try to reduce duplication in the app by using a reactive expression.

```
ui <- fluidPage(
  sliderInput("fahr", label = "If the temperature in ° Fahrenheit is",
              min = -58, max = 122, value = 77),
  sliderInput("r", label = "and the relative humidity is",
              min = 0, max = 100, value = 0),
  "then the heat index in ° Fahrenheit is ",
  textOutput("fahr_hi_out"),
  "and the heat index in ° Celsius is",
```

```
  textOutput("celsius_hi")
)

server <- function(input, output, session) {
  fahr_hi <- reactive({
    ## maybe define something here to use twice below
  })

  output$fahr_hi_out <- renderText({
    # put some code here
  })

  output$celsius_hi_out <- renderText({
    # put some code here
  })
}

heat_index_fahr <- function (t, r) {
  ## https://en.wikipedia.org/wiki/Heat_index#Formula
  cf <- c(
    -8.78469475556, 1.61139411, 2.33854883889, -0.14611605,
    -0.012308094, -0.0164248277778, 0.002211732, 0.00072546,
    -0.000003582)
  cf[1] +
    (cf[2] * t) + (cf[3] * r) +
    (cf[4] * r * t) + (cf[5] * t * t) + (cf[6] * r * r) +
    (cf[7] * t * t * r) + (cf[8] * t * r * r) +
    (cf[9] * t * t * r * r)
}
```

4. The following app is very similar to one you've seen earlier in the chapter: you select a dataset (this time we're using data from the **ggplot2** package) and the app prints out a summary and plot of the data. It also follows good practice and makes use of reactive expressions to avoid redundancy of code. However there are three bugs in the code provided below. Can you find and fix them?

```
library(ggplot2)
datasets <- data(package = "ggplot2")$results[, "Item"]

ui <- fluidPage(
  selectInput("dataset", "Dataset", choices = datasets),
  verbatimTextOutput("summary"),
  tableOutput("plot")
)

server <- function(input, output, session) {
  dataset <- reactive({
    get(input$dataset, "package:ggplot2")
  })
  output$summmry <- renderPrint({
    summary(dataset())
  })
  output$plot <- renderPlot({
    plot(dataset)
```

```
  })
}
```

4.11   Basic UI

4.11.1   Introduction

Shiny encourages separation of the code that generates the user interface (the front end) from the code that
drives your app's behaviour (the backend). Here we'll dive deeper into the front end and explore the HTML
inputs, outputs, and layouts provided by Shiny.

As usual, begin by loading the shiny package:

```
library(shiny)
```

4.12   Inputs

As we saw before, you use `*Input()`functions like `sliderInput()`, `selectInput()`, `textInput()`, and
`numericInput()` to insert input controls into your UI specification. Now we'll discuss the common structure
that underlies all input functions and give a quick overview of the inputs built into Shiny.

4.12.1   Common structure

All input functions have the same first argument: `inputId`. This is the identifier used to connect the front
end with the back end: if your UI has an input with ID `"name"`, the server function will access it with
`input$name`.

The `inputId` has two constraints:

- It must be a simple string that contains only letters, numbers, and underscores. Name it like you
  would name a variable in R.

- It must be unique.

Most input functions have a second parameter called `label`. This is what users (i.e. humans) see. There are
no restrictions, but you should carefully think about how you label your input for humans!

The third parameter is typically `value`, which, where possible, lets you set the default value. Any remaining
parameters are unique to the control.

Good form in creating an input is to supply the `inputId` and `label` arguments by position, and all other
arguments by name:

```
sliderInput("fahr", "Temperature in Fahrenheit", value = 77, min = -58, max = 137)
```

The inputs built into Shiny are described below. More complete information is available in the package
documentation.

4.12.2 Free text

Collect *small* amounts of text with `textInput()`, passwords with `passwordInput()`[^password], and paragraphs of text with `textAreaInput()`.

Note: `passwordInput()` only hides what the user is typing, if you're using this in a way that needs to be secure, make sure you either have secure programming training, or consult someone who does.

```r
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "Whisper something"),
  textAreaInput("story", "Tell me a story", rows = 3)
)
```
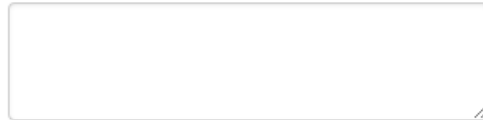
If you want to ensure that the text has certain properties you can use `validate()`.
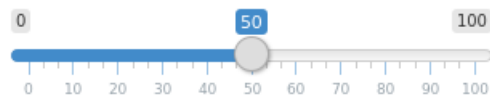
4.12.3 Numeric inputs

To collect numeric values, create a slider with `sliderInput()` or a numeric textbox with `numericInput()`. Supplying a length-2 numeric vector for the default value of `sliderInput()` provides a "range" slider with two ends.
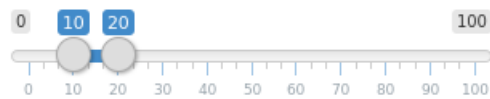
```r
ui <- fluidPage(
  numericInput("num", "First number", value = 0, min = 0, max = 100),
  sliderInput("num2", "Second number", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```

Number one

0

Number two

0   50   100

0 10 20 30 40 50 60 70 80 90 100

Range

0 10 20   100

0 10 20 30 40 50 60 70 80 90 100

Use sliders for small ranges where the precise value is not so important, otherwise just use a numeric input. If you have had the experience before of setting a precise number on a small slider, it can be very frustrating!

4.12.4 Dates

Collect a single day with `dateInput()` or a range of two days with `dateRangeInput()`. These provide a convenient calendar picker, and additional arguments like `datesdisabled` and `daysofweekdisabled` allow you to restrict the set of valid inputs.

```
ui <- fluidPage(
  dateInput("dob", "When were you born?"),
  dateRangeInput("dentist", "When do you want to see the dentist next?")
)
```

When were you born?

2020-05-11

When do you want to see the dentist next?

2020-05-11 to 2020-05-11

Date format, language, and the day on which the week starts defaults to US standards. If there are specific formats preferred in your organization, or you're creating an app with an international audience, try setting `format`, `language`, and `weekstart` appropriately, so that the dates are natural to your users.

4.12.5 Limited set of choices

There are two different approaches to allow the user to choose from a prespecified set of options: `selectInput()` and `radioButtons()`.

```
animals <- c("dog", "cat", "mouse", "bird", "hamster", "other",
             "I hate animals")

ui <- fluidPage(
```

```
  selectInput("state", "What's the best state?", state.name),
  radioButtons("animal", "What's your favorite animal?", animals)
)
```

What's the best state?

Alabama   ▼

What's your favorite animal?

- ⦿ dog
- ○ cat
- ○ mouse
- ○ bird
- ○ hamster
- ○ other
- ○ I hate animals

Radio buttons show all possible options, making them suitable for short lists. They also can display options other than plain text, via the `choiceNames` and `choiceValues` arguments.

```
ui <- fluidPage(
  radioButtons("rb", "I feel:",
               choiceNames = list(
                 icon("angry"),
                 icon("smile"),
                 icon("sad-tear")
               ),
               choiceValues = list("angry", "happy", "sad")
  )
)
```

I feel:

- ⦿
- ○
- ○

Dropdowns created with `selectInput()` are more suitable for longer take up the same amount of space, regardless of the number of options, making them more suitable for longer options. You can also set `multiple = TRUE` to allow the user to select multiple elements.

```
ui <- fluidPage(
  selectInput(
    "state", "What states border Tennessee?", state.name,
    multiple = TRUE
  )
)
```

`checkboxGroupInput()` is an alternative to radio buttons for selecting multiple values.

```
ui <- fluidPage(
  checkboxGroupInput("animal", "What animals do you like?", animals)
)
```

What animals do you like?

- ☐ dog
- ☐ cat
- ☐ mouse
- ☐ bird
- ☐ hamster
- ☐ other
- ☐ I hate animals

If you want a single checkbox for a single yes/no question, use `checkboxInput()`:

```
ui <- fluidPage(
  checkboxInput("shutdown", "Shutdown?")
)
```

☐ Shutdown?

4.12.6   File uploads

Allow the user to upload a file with `fileInput()`:

```
ui <- fluidPage(
  fileInput("upload", NULL)
)
```

Browse...   No file selected

Note: `fileInput()` requires special handling on the server side.

4.12.7   Action buttons

Let the user trigger an action through a button or link with `actionButton()` or `actionLink()`. These are naturally paired with `observeEvent()` or `eventReactive()` in the server function.

```
ui <- fluidPage(
  actionButton("click", "Click me!"),
  actionButton("food", "Use the Force!", icon = icon("jedi"))
)
```

Click me!   ☷ Use the Force!

4.12.8 Exercises

1. Create a slider input to select values between 0 and 100 where the interval between each selectable value on the slider is 5. Then, using the documentation for `sliderInput`, try to add animation so when the user presses play the input widget scrolls through automatically. Can you also loop the animation? (Hint: see https://shiny.rstudio.com/articles/sliders.html).

2. Using the following numeric input box the user can enter any value between 0 and 1000. What is the purpose of the step argument in this widget?

```
numericInput("number", "Select a value", value = 150, min = 0, max = 1000, step = 50)
```

4.13 Outputs

Outputs in the UI create placeholders that the server fills later. Like inputs, outputs take a unique ID as their first argument: if your UI specification creates an output with ID `"plot"`, you'll access it in the server function with `output$plot`.

Each `output` function in the UI pairs with a `render` function in the server. There are three main types of output corresponding to the three things you usually include in a report: *text*, *tables*, and *plots*.

4.13.1 Text

Output regular text with `textOutput()` and fixed code and console output with `verbatimTextOutput()`. These are paired in the server with `renderText({...})` and `renderPrint({...})`.

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("code")
)
server <- function(input, output, session) {
  output$text <- renderText({
    "Hello from the backend!"
  })
  output$code <- renderPrint({
    summary(rnorm(100))
  })
}
```

Note that the `{}` is only required in render functions if you need to run multiple lines of code. You could also write the server function more compactly, which is generally considered better style.

```
server <- function(input, output, session) {
  output$text <- renderText("Hello from the backend!")
  output$code <- renderPrint(summary(rnorm(100)))
}
```

Note: there are two render functions that can be used with either of the text output functions:

- `renderText()` displays text *returned* by the code.
- `renderPrint()` displays text *printed* by the code.

To understand the difference, examine the following function. It prints `a` and `b`, and returns `"c"`. A function can print multiple things, but can only return a single value.

```
print_and_return <- function() {
  print("a")
  print("b")
  "c"
}
x <- print_and_return()
#> [1] "a"
#> [1] "b"
x
#> [1] "c"
```

4.13.2 Tables

There are two options for displaying data frames in tables:

- `tableOutput()` and `renderTable()` render a static table of data, showing all the data at once.

- `dataTableOutput()` and `renderDataTable()` render a dynamic table, showing a fixed number of rows along with controls to change which rows are visible. This uses the DataTables Javascript library.

`tableOutput()` is useful for small, fixed summaries; `dataTableOutput()` is more appropriate if you want to expose a complete data frame to the user.

```
ui <- fluidPage(
  tableOutput("static"),
  dataTableOutput("dynamic")
)
server <- function(input, output, session) {
  output$static <- renderTable(head(mtcars))
  output$dynamic <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

4.13.3 Plots

You can display any type of R graphic (base, ggplot2, or anything else) with `plotOutput()` and `renderPlot()`:

```
ui <- fluidPage(
  plotOutput("plot", width = "400px")
)
server <- function(input, output, session) {
  output$plot <- renderPlot(plot(rnorm(100)))
}
```

Plots are very cool because they are outputs that can also act as inputs, which can allow for all types of interactivity when the user clicks or even hovers on the plot. `plotOutput()` has a number of arguments like `click`, `dblclick`, and `hover`. If you pass these a string, like `click = "plot_click"`, they'll create a reactive input (`input$plot_click`) that you can use to handle user interaction on the plot.

4.13.4 Downloads

You can let the user download a file with `downloadButton()` or `downloadLink()`. These require some handling in the server function.

4.13.5 Exercises

1. Update the options for `renderDataTable()` below so that the table is displayed, but nothing else, i.e. remove the search, ordering, and filtering commands. You'll need to read `?renderDataTable` and review the options at https://datatables.net/reference/option/.

```r
ui <- fluidPage(
  dataTableOutput("table")
)
server <- function(input, output, session) {
  output$table <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

4.14 Layouts

Now that you can create a full range of inputs and outputs, it would be nice to arrange them pleasingly on the page. That's the purpose of the layout functions.

`fluidPage()` provides the layout style used by most apps, which we'll focus on here, but there are also other layout families available like dashboards and dialog boxes.

4.14.1 Overview

Layouts, being fundamentally statements that create HTML, are just a hierarchy of function calls. When you see a complex layout like this:

```r
fluidPage(
  titlePanel("Hello Shiny!"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs", "Observations:", min = 0, max = 1000, value = 500)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

You can skim it by focusing on the hierarchy of the function calls:

```r
fluidPage(
  titlePanel(),
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs")
    ),
```

```
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

Without knowing anything about the layout functions themselves you can guess that this code will generate a classic app design: a title bar at top, with a sidebar to the left (containing a slider), and a main panel containing a plot.

4.14.2   Page functions

The most important, but least interesting, layout function is `fluidPage()`. You've seen it in every example above, because we use it to put multiple inputs or outputs into a single app. What happens if you use `fluidPage()` by itself?
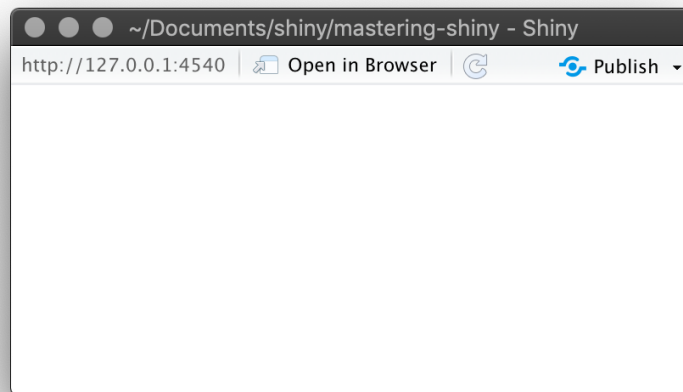
There's no content!, but behind the scenes `fluidPage()` does a lot of work. It sets up the HTML, CSS, and JS that Shiny needs, using a layout system called **Bootstrap**, https://getbootstrap.com, that provides attractive defaults. With a little knowledge of Bootstrap, you can really control the visual appearance of your app to make it more polished or match your organizational style.

While technically `fluidPage()` is all you need, dumping all the inputs and outputs in one place doesn't look very good.

There are two other common structures: a page with a sidebar, and a multi-row app.

4.14.3   Page with sidebar

`sidebarLayout()`, along with `titlePanel()`, `sidebarPanel()`, and `mainPanel()`, creates a two-column layout with inputs on the left and outputs on the right.

```
fluidPage(
  titlePanel(
    # app title/description
  ),
  sidebarLayout(
    sidebarPanel(
      # inputs
    ),
    mainPanel(
      # outputs
    )
  )
)
```

The following example, a classic Shiny app, demonstrates the Central Limit Theorem from basic statistics:
increasing the number of samples makes a distribution closer to a normal distribution.

```
ui <- fluidPage(
  headerPanel("Central limit theorem"),
  sidebarLayout(
    sidebarPanel(
      numericInput("m", "Number of samples:", 2, min = 1, max = 100)
    ),
    mainPanel(
      plotOutput("hist")
    )
  )
)

server <- function(input, output, session) {
  output$hist <- renderPlot({
    means <- replicate(1e4, mean(runif(input$m)))
    hist(means, breaks = 20)
  })
}
```

4.14.4   Multi-row

Under the hood, `sidebarLayout()` is built on top of a flexible multi-row layout. To use it directly, you still
start with `fluidPage()`, but create rows with `fluidRow()` and columns within the rows with `column()`.

```
fluidPage(
  fluidRow(
    column(4,
      ...
    ),
    column(8,
      ...
    )
  ),
  fluidRow(
    column(6,
```

```
        ...
    ),
    column(6,
        ...
    )
  )
)
```

The first argument to `column()` is the width. The width of each row must add up to 12, which offers flexibility because 12 columns divide easily into 2-, 3-, or 4-column layouts.

4.14.5   Themes

Creating a complete theme from scratch is a lot of work, but the easy wins with the `shinythemes` package. First make sure it is installed, it is not automatically installed with `shiny`:

```
install.packages("shinythemes")
```

The following code shows four options:

```
theme_demo <- function(theme) {
  fluidPage(
    theme = shinythemes::shinytheme(theme),
    sidebarLayout(
      sidebarPanel(
        textInput("txt", "Text input:", "text here"),
        sliderInput("slider", "Slider input:", 1, 100, 30)
      ),
      mainPanel(
        h1("Header 1"),
        h2("Header 2"),
        p("Some text")
      )
    )
  )
}
theme_demo("darkly")
theme_demo("flatly")
theme_demo("sandstone")
theme_demo("united")
```
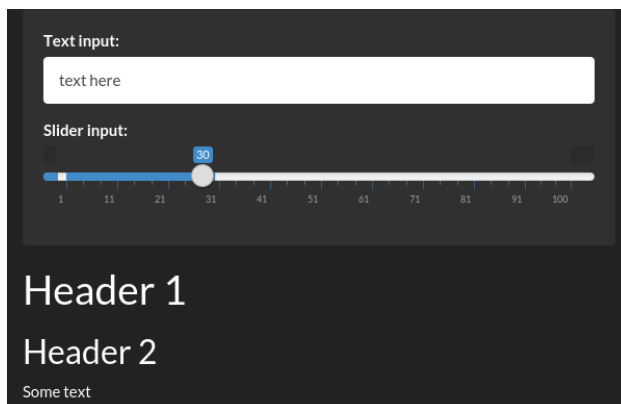
Theming is quite straightforward: you just need to use the `theme` argument to `fluidPage()`. Available themes are demo'd in the Shiny theme selector app at https://shiny.rstudio.com/gallery/shiny-theme-selector.html.

5   Reactivity

5.1   Introduction

The previous chapter discussed user interfaces. Now we'll see how the server side of Shiny can bring your user interface to life.

Writing server logic in shiny involves something called "reactive programming". It is an elegant and powerful programming paradigm, but it can be disorienting at first because it's a very different from writing a script. The key idea of reactive programming is to specify a graph of dependencies so that when an input changes, all related outputs are automatically updated. This makes the flow of an app considerably simpler, but it takes a while to get your head around how it all fits together.

```
library(shiny)
```

5.2   The server function

To review, the basic structure of any shiny app looks like this.

```
library(shiny)

ui <- fluidPage(
  # front end interface
)

server <- function(input, output, session) {
  # back end logic
}

shinyApp(ui, server)
```

The previous section covered the basics of the front end, the `ui` object that contains the HTML presented to every user of your app. The `ui` is simple because every user gets the same HTML. The `server` is more

complicated, because every user needs to get an independent version of the app; when user A moves a slider, user B shouldn't see their outputs change.

To achieve this independence, Shiny invokes your `server()` function each time a new session[1] starts. Just like any other R function, when the server function is called it creates a new local environment that is independent of every other invocation of the function. This allows each session to have a unique state, as well as isolating the variables created *inside* the function.

Server functions take three parameters: `input`, `output`, and `session`[2]. Because you never call the server function yourself, you'll never create these objects yourself. Instead, they're created by Shiny when the session begins, connecting back to a specific session. For this course, we'll focus on the `input` and `output` arguments'.

5.2.1   Input

The `input` argument is a list-like object that contains all the input data sent from the browser, named according to the input ID. For example, if your UI contains a numeric input control with an input ID of `volume`, like so:

```
ui <- fluidPage(
  numericInput("", label = "Number of values", value = 100)
)
```

you can access the value of that input with `input$count`. It will initially contain the value 100, and it will be automatically updated as the user changes the value in the browser.

Unlike a typical list, `input` objects are read-only. If you attempt to modify an input inside the server function, you'll get an error:

```
server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
#> Error: Attempted to assign value to a read-only reactivevalues object
```

This error occurs because `input` reflects what's happening in the browser, and the browser is Shiny's "single source of truth". If you could modify the value in R, you could introduce inconsistencies, where the input slider said one thing in the browser, and `input$count` said something different in R. That would make programming challenging! Later, in Chapter **??**, you'll learn how to use functions like `updateNumericInput()` to modify the value in the browser, and then `input$count` will update accordingly.

One more important thing about `input`: it's selective about who is allowed to read it. To read from an `input`, you must be in a **reactive context** created by a function like `renderText()` or `reactive()`. We'll come back to that idea very shortly, but it's an important constraint that allows outputs to automatically update when an input changes. This code illustrates the error you'll see if you make this mistake:

```
server <- function(input, output, session) {
  message("The value of input$count is ", input$count)
}
```

---

[1]Each connection to a Shiny app starts a new session whether it's connections from different people, or with multiple tabs from the same person.
[2]For legacy reasons, `session` is optional, but you should always include it.

```
shinyApp(ui, server)
#> Error: Operation not allowed without an active reactive context.
#> (You tried to do something that can only be done from inside
#> a reactive expression or observer.)
```

5.2.2 Output

`output` is very similar to `input`: it's also a list-like object named according to the output ID. The main difference is that you use it for sending output instead of receiving input. You always use the `output` object in concert with a `render` function, as in the following simple example:

```
ui <- fluidPage(
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText("Hello human!")
}
```

(Note that the ID is quoted in the UI, but not in the server.)

The render function does two things:

- It sets up a special reactive context that automatically tracks what inputs the output uses.

- It converts the output of your R code into HTML suitable for display on a web page.

Like the `input`, the `output` is picky about how you use it. You'll get an error if:

- You forget the `render` function.

  ```
  server <- function(input, output, session) {
    output$greeting <- "Hello human"
  }
  shinyApp(ui, server)
  #> Error: Unexpected character output for greeting
  ```

- You attempt to read from an output.

  ```
  server <- function(input, output, session) {
    message("The greeting is ", output$greeting)
  }
  shinyApp(ui, server)
  #> Error: Reading objects from shinyoutput object not allowed.
  ```

5.3   Reactive programming

An app is going to be pretty boring if it only has inputs or only has outputs. The real magic of Shiny happens when you have an app with both. Let's look at a simple example:

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
```

It's hard to show exactly how this works in a book, but if you run the app, and type in the name box, you'll notice that the greeting updates automatically as you type

If you're running the live app, notice that you have to type fairly slowly for the output to update one letter at a time. That's because Shiny uses a technique called **debouncing**, which means that it waits for a few ms before sending an update. That considerably reduces the amount of work that Shiny needs to do, without appreciably reducing the response time of the app.

This is the big idea in Shiny: you don't need to tell an output when to update, because Shiny automatically figures it out for you. How does it work? What exactly is going on in the body of the function? Let's think about the code inside the server function more precisely:

```
output$greeting <- renderText({
  paste0("Hello ", input$name, "!")
})
```

It's easy to read this as "paste together 'hello' and the user's name, then send it to `output$greeting`". But this mental model is wrong in a subtle, but important, way. Think about it: with this model, you only issue the instruction once. But Shiny performs the action every time we update `input$name`, so there must be something more going on.

The app works because the code doesn't *tell* Shiny to create the string and send it to the browser, but instead, it informs Shiny *how it could* create the string if it needs to. It's up to Shiny when (and even if!) the code should be run. It might be run as soon as the app launches, it might be quite a bit later; it might be run many times, or it might never be run! This isn't to imply that Shiny is capricious, only that it's Shiny's responsibility to decide when code is executed, not yours. Think of your app as providing Shiny with recipes, not giving it commands.

5.3.1   Imperative vs declarative programming

This difference between commands and recipes is one of the key differences between two important styles of programming:

- In **imperative** programming, you issue a specific command and it's carried out immediately. This is the style of programming you're used to in your analysis scripts: you command R to load your data, transform it, visualise it, and save the results to disk.

- In **declarative** programming, you express higher-level goals or describe important constraints, and rely on someone else to decide how and/or when to translate that into action. This is the style of programming you use in Shiny.

With imperative code you say "Make me a sandwich". With declarative code you say "Ensure there is a sandwich in the refrigerator whenever I look inside of it". Imperative code is assertive; declarative code is passive-aggressive.

Most of the time, declarative programming is tremendously freeing: you describe your overall goals, and the software figures out how to achieve them without further intervention. The downside is the occasional time where you know exactly what you want, but you can't figure out how to frame it in a way that the declarative system understands[3]. The goal of this book is to help you develop your understanding of the underlying theory so that happens as infrequently as possible.

5.3.2 Laziness

One of the strengths of declarative programming in Shiny is that it allows apps to be extremely lazy. A Shiny app will only ever do the minimal amount of work needed to update the output controls that you can currently see[4]. This laziness, however, comes with an important downside that you should be aware of. Can you spot what's wrong with the server function below?

```
server <- function(input, output, session) {
  output$greetnig <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
```

If you look closely, you might notice that I've written `greetnig` instead of `greeting`. This won't generate an error in Shiny, but it won't do what you want. The `greetnig` output doesn't exist, the code inside `renderText()` will never be run.

If you're working on a Shiny app and you just can't figure out why your code never gets run, double check that your UI and server functions are using the same identifiers.

5.3.3 The reactive graph

Shiny's laziness has another important property. In most R code, you can understand the order of execution by reading the code from top to bottom. That doesn't work in Shiny, because code is only run when needed. To understand the order of execution you need to instead look at the **reactive graph**, which describes how inputs and outputs are connected. The reactive graph for the app above is very simple:
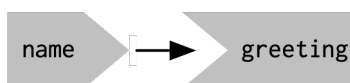
The reactive graph contains one symbol for every input and output, and we connect an input to an output whenever the output accesses the input. This graph tells you that `greeting` will need to be recomputed whenever `name` is changed. We'll often describe this relationship as `greeting` has a **reactive dependency** on `name`.

Note the graphical conventions we used for the inputs and outputs: the `name` input naturally fits into the `greeting` output. We could draw them closely packed together, as below, to emphasise the way that they fit together; we won't normally do that because it only works for the simplest of apps.

---

[3]If you've ever struggled to get a ggplot2 legend to look exactly the way you want, you've encountered this problem!

[4]Yes, Shiny doesn't update the output if you can't see it in your browser! Shiny is so lazy that it doesn't do the work unless you can actually see the results.

The reactive graph is a powerful tool for understanding how your app works. As your app gets more complicated, it's often useful to make a quick high-level sketch of the reactive graph to remind you how all the pieces fit together. Throughout this book we'll show you the reactive graph to help understand how the examples work, and later on, in Chapter XYZ, you'll learn how to use reactlog which will draw the graph for you.

5.3.4 Reactive expressions

There's one more important component that you'll see in the reactive graph: the reactive expression. We'll come back to reactive expressions in detail very shortly; for now think of them as a tool that reduces duplication in your reactive code by introducing additional nodes into the reactive graph.

We don't need a reactive expression in our very simple app, but I'll add one anyway so you can see how it affects the graph:

```r
server <- function(input, output, session) {
  text <- reactive(paste0("Hello ", input$name, "!"))
  output$greeting <- renderText(text())
}
```


Reactive expressions take inputs and produce outputs so they have a shape that combines features of both inputs and outputs. Hopefully, the shapes will help you remember how the components fit together.

5.3.5 Execution order

It's important to understand that the order in which your code is run is determined solely by the reactive graph. This is different from most R code where the execution order is determined by the order of lines. For example, we could flip the order of the two lines in our simple server function:
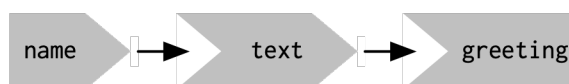
```r
server <- function(input, output, session) {
  output$greeting <- renderText(text())
  text <- reactive(paste0("Hello ", input$name, "!"))
}
```

You might think that this would yield an error because `output$greeting` refers to a reactive expression, `text`, that hasn't been created yet. But remember Shiny is lazy, so that code is only run when the session starts, after `text` has been created.

Instead, this code yields the same reactive graph as above, so the order in which the code is run is exactly the same. Organising your code like this is confusing for humans, and best avoided. Instead, make sure that reactive expressions and outputs only refer to things defined above, not below[5]. This will make your code easier to understand.

This concept is very important and different to most other R code, so I'll say it again: the order in which reactive code is run is determined only by the reactive graph, not by its layout in the server function.

---

[5]The technical term for this ordering is a "topological sort".

5.3.6 Exercises

1. Draw the reactive graph for the following server functions:

```
server1 <- function(input, output, session) {
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)
  output$f <- renderText(e())
}
server2 <- function(input, output, session) {
  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$z <- renderText(x() / y())
}
server3 <- function(input, output, session) {
  d <- reactive(c() ^ input$d)
  a <- reactive(input$a * 10)
  c <- reactive(b() / input$c)
  b <- reactive(a() + input$b)
}
```

2. Can the reactive graph contain a cycle? Why/why not?

5.4 Reactive expressions

We've quickly skimmed over reactive expressions a couple of times, so you're hopefully getting a sense for what they might do. Now we'll dive into more of the details, and show why they are so important when constructing real apps.

Reactive expressions are important for two reasons:

- They give Shiny more information so that it can do less recomputation when inputs change, making apps more efficient.

- They make it easier for humans to understand the app by simplifying the reactive graph.

Reactive expressions have a flavour of both inputs and outputs:

- Like inputs, you can use the results of a reactive expression in an output.

- Like outputs, reactive expressions depend on inputs and automatically know when they need updating.

Because of this duality, some functions work with either reactive inputs or expressions, and some functions work with either reactive expressions or reactive outputs. We'll use **producers** to refer to either reactive inputs or expressions, and **consumers** to refer to either reactive expressions or outputs. Figure 4 shows this relationship with a Venn diagram.

We're going to need a more complex app to see the benefits of using reactive expressions. First, we'll set the stage by defining some regular R functions that we'll use to power our app.
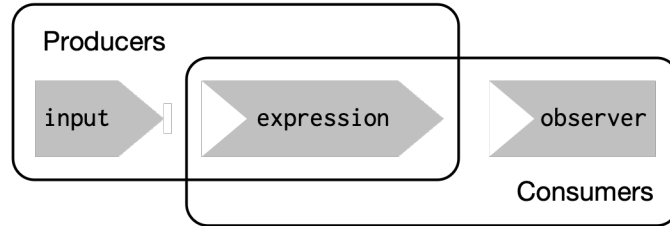
Figure 4: Inputs and expressions are reactive producers; expressions and outputs are reactive consumers

5.4.1 The motivation

Imagine I want to compare two simulated datasets with a plot and a hypothesis test. I've done a little experimentation and come up with the functions below: `histogram()` visualises the two distributions with a histogram, and `t_test()` uses a t-test to compare means and summarises the results with a string:

```r
library(ggplot2)

histogram <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {
  df <- data.frame(
    x = c(x1, x2),
    g = c(rep("x1", length(x1)), rep("x2", length(x2)))
  )

  ggplot(df, aes(x, fill = g)) +
    geom_histogram(binwidth = binwidth) +
    coord_cartesian(xlim = xlim)
}

t_test <- function(x1, x2) {
  test <- t.test(x1, x2)

  sprintf(
    "p value: %0.3f\n[%0.2f, %0.2f]",
    test$p.value, test$conf.int[1], test$conf.int[2]
  )
}
```

If I have some simulated data, I can use these functions to compare two variables:

```r
x1 <- rnorm(100, mean = 0, sd = 0.5)
x2 <- rnorm(200, mean = 0.15, sd = 0.9)

histogram(x1, x2)
cat(t_test(x1, x2))
#> p value: 0.001
#> [-0.40, -0.11]
```

In a real analysis, you probably would've done a bunch of exploration before you ended up with these functions. I've skipped that exploration here so we can get to the app as quickly as possible. But extracting imperative code out into regular functions is an important technique for all Shiny apps: the more code you can extract out of your app, the easier it will be to understand. This is good software engineering because it helps isolate concerns: the functions outside of the app focus on the computation so that the code inside of the app can focus on responding to user actions.
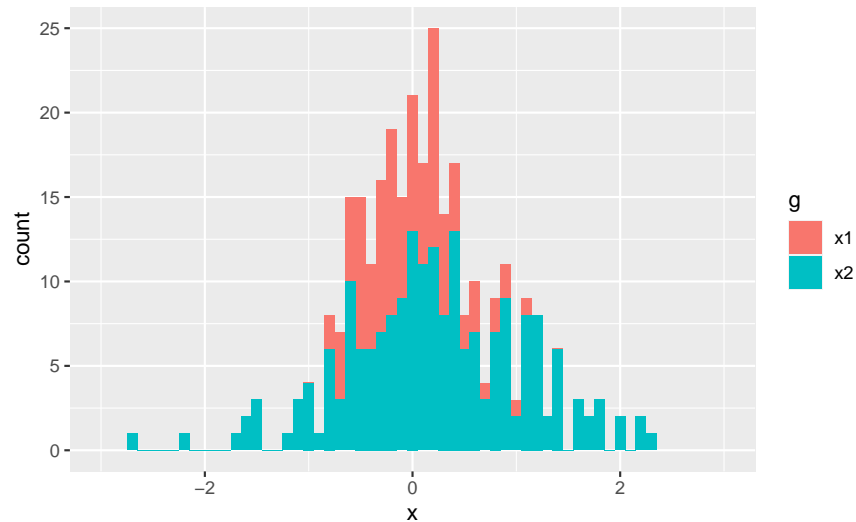
5.4.2 The app

I'd like to use these two tools to quickly explore a bunch of simulations. A Shiny app is a great way to do this because it lets you avoid tediously modifying and re-running R code. Below I wrap the pieces into a Shiny app where I can interactively tweak the inputs.

Let's start with the UI. The first row has three columns for input controls (distribution 1, distribution 2, and plot controls). The second row has a wide column for the plot, and a narrow column for the hypothesis test.

```
ui <- fluidPage(
  fluidRow(
    column(4,
      "Distribution 1",
      numericInput("n1", label = "n", value = 1000, min = 1)
      # numericInput("mean1", label = "µ", value = 0, step = 0.1)
    ),
    column(4,
      "Distribution 2",
      numericInput("n2", label = "n", value = 1000, min = 1)
     # numericInput("mean2", label = "µ", value = 0, step = 0.1),
    ),
    column(4,
      "Histogram",
      numericInput("binwidth", label = "Bin width", value = 0.1, step = 0.1),
      sliderInput("range", label = "range", value = c(-3, 3), min = -5, max = 5)
    )
  ),
```

```
  fluidRow(
    column(9, plotOutput("hist")),
    column(3, verbatimTextOutput("ttest"))
  )
)
```

The server function combines calls to `histogram()` and `t_test()` functions after drawing from the specified distributions:

```
server <- function(input, output, session) {
  output$hist <- renderPlot({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    histogram(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    t_test(x1, x2)
  })
}
```

This definition of `server` and `ui` yields Figure **??**. You can find a live version at https://hadley.shinyapps. io/basic-reactivity-cs/; I recommend opening the app and having a quick play to make sure you understand its basic operation before you continue reading.

5.4.3   The reactive graph

Let's start by drawing the reactive graph of this app. Shiny is smart enough to update an output only when the inputs it refers to change; it's not smart enough to only selectively run pieces of code inside an output. In other words, outputs are atomic: they're either executed or not as a whole.

For example, take this snippet from the server:

```
x1 <- rnorm(input$n1, input$mean1, input$sd1)
x2 <- rnorm(input$n2, input$mean2, input$sd2)
t_test(x1, x2)
```

As a human reading this code you can tell that we only need to update `x1` when `n1`, `mean1`, or `sd1` changes, and we only need to update `x2` when `n2`, `mean2`, or `sd2` changes. Shiny, however, only looks at the output as a whole, so it will update both `x1` and `x2` every time one of `n1`, `mean1`, `sd1`, `n2`, `mean2`, or `sd2` changes. This leads to the reactive graph shown in Figure 5:

You'll notice that the graph is very dense: almost every input is connected directly to every output. This creates two problems:

- The app is hard to understand because there are so many connections. There are no pieces of the app that you can pull out and analyse in isolation.

- The app is inefficient because it does more work than necessary. For example, if you change the breaks of the plot, the data is recalculated; if you change the value of `n1`, `x2` is updated (in two places!).
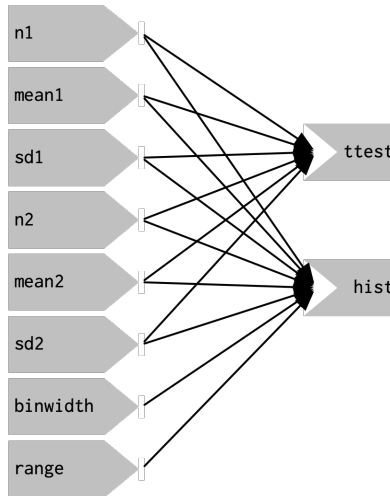
70

Figure 5: The reactive graph shows that every output depends on every input

There's one other major flaw in the app: the histogram and t-test use separate random draws. This is rather misleading, as you'd expect them to be working on the same underlying data.

Fortunately, we can fix all these problems by using reactive expressions to pull out repeated computation.

5.4.4 Simplifying the graph

In the server function below we refactor the existing code to pull out the repeated code into two new reactive expressions, `x1` and `x2`, which simulate the data from the two distributions. To create a reactive expression, we call `reactive()` and assign the results to a variable. To later use the expression, we call the variable like it's a function.

```r
server <- function(input, output, session) {
  x1 <- reactive(rnorm(input$n1, input$mean1, input$sd1))
  x2 <- reactive(rnorm(input$n2, input$mean2, input$sd2))

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

This transformation yields the substantially simpler graph shown in Figure 6. This simpler graph makes it easier to understand the app because you can understand connected components in isolation; the values of the distribution parameters only affect the output via `x1` and `x2`. This rewrite also makes the app much more efficient since it does much less computation. Now, when you change the `binwidth` or `range`, only the plot changes, not the underlying data.

To emphasise this modularity the Figure 7 draws boxes around the independent components. We'll come back to this idea in Chapter **??**, when we discuss modules. Modules allow you to extract out repeated code for reuse, while guaranteeing that it's isolated from everything else in the app. Modules are an extremely useful and powerful technique for more complex apps.
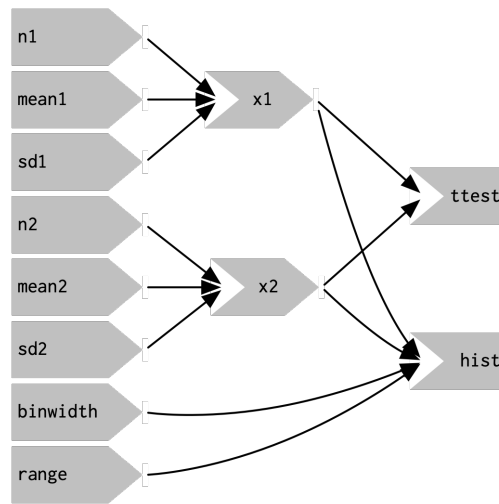
Figure 6: Using reactive expressions considerably simplifies the graph, making it much easier to understand
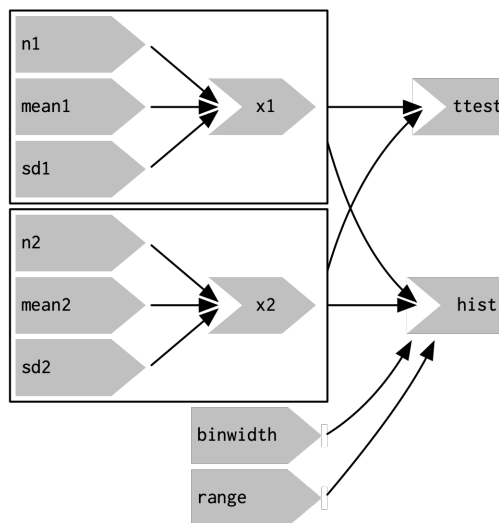

Figure 7: Modules enforce isolation between parts of an app

You might be familiar with the "rule of three" of programming: whenever you copy and paste something three times, you should figure out how to reduce the duplication (typically by writing a function). This is important because it reduces the amount of duplication in your code, which makes it easier to understand, and easier to update as your requirements change.

In Shiny, however, I think you should consider the rule of one: whenever you copy and paste something *once*, you should consider extracting the repeated code out into a reactive expression. The rule is stricter for Shiny because reactive expressions don't just make it easier for humans to understand the code, they also improve Shiny's ability to efficiently rerun code.

5.4.5   Why do we need reactive expressions?

When you first start working with reactive code, you might wonder why we need reactive expression. Why can't you use your existing tools for reducing duplication in code: creating new variables and writing functions? Unfortunately neither of these techniques work in a reactive environment.

If you try to use a variable to reduce duplication, you might write something like this:

```
server <- function(input, output, session) {
  x1 <- rnorm(input$n1, input$mean1, input$sd1)
  x2 <- rnorm(input$n2, input$mean2, input$sd2)

  output$hist <- renderPlot({
    histogram(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1, x2)
  })
}
```

If you run this code, you'll get an error because you're attempting to access input values outside of a reactive context. Even if you didn't get that error, you'd still have a problem: `x1` and `x2` would only be computed once, when the session begins, not every time one of the inputs was updated.

If you try to use a function, the app will work:

```
server <- function(input, output, session) {
  x1 <- function() rnorm(input$n1, input$mean1, input$sd1)
  x2 <- function() rnorm(input$n2, input$mean2, input$sd2)

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

But it has the same problem as the original code: any input will cause all outputs to be recomputed, and the t-test and the histogram will be run on separate samples. Reactive expressions automatically cache their results, and only update when their inputs change[6].

---

[6]If you're familiar with memoisation, this is a similar idea.

While variables calculate the value only once (the porridge is too cold), and functions calculate the value every time they're called (the porridge is too hot), reactive expressions calculate the value only when it might have changed (the porridge is just right!).

5.4.6   Exercise

1. Use reactive expressions to reduce the duplicated code in the following simple apps.

5.5   Controlling timing of evaluation

Now that you're familiar with the basic ideas of reactivity, we'll discuss two more advanced techniques that allow you to either increase or decrease how often a reactive expression is executed. Here I'll show how to use the basic techniques; in Chapter XYZ, we'll come back to their underlying implementations.

To explore the basic ideas, I'm going to simplify my simulation app. I'll use a distribution with only one parameter, and force both samples to share the same `n`. I'll also remove the plot controls. This yields a smaller UI object and server function:

```
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 3),
      numericInput("n", label = "n", value = 1e4, min = 0)
    ),
    column(9, plotOutput("hist"))
  )
)
server <- function(input, output, session) {
  x1 <- reactive(rpois(input$n, input$lambda1))
  x2 <- reactive(rpois(input$n, input$lambda2))
  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

This generates the app shown in Figure **??** and reactive graph shown in Figure 8.
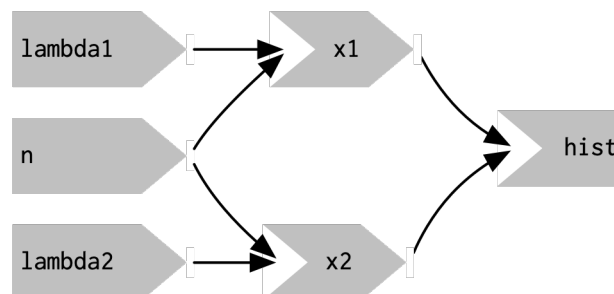

Figure 8: The reactive graph

5.5.1 Timed invalidation

Imagine you wanted to reinforce the fact that this is for simulated data by constantly resimulating the data, so that you see an animation rather than a static plot[7]. We can increase the frequency of updates with a new function: `reactiveTimer()`.

`reactiveTimer()` is a reactive expression that has a dependency on a hidden input: the current time. You can use a `reactiveTimer()` when you want a reactive expression to invalidate itself more often than it otherwise would. For example, the following code uses an interval of 500 ms so that the plot will update twice a second. This is fast enough to remind you that you're looking at a simulation, without dizzying you with rapid changes. This change yields the reactive graph shown in Figure 9

```
server <- function(input, output, session) {
  timer <- reactiveTimer(500)

  x1 <- reactive({
    timer()
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    timer()
    rpois(input$n, input$lambda2)
  })

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```
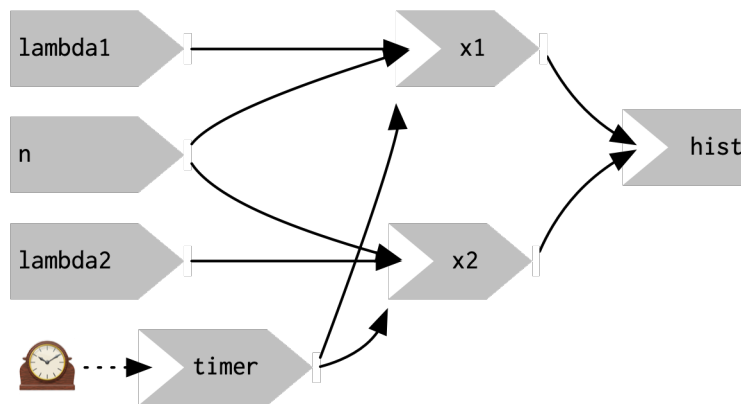

Figure 9: 'reactiveTimer(500)' introduces a new reactive input that automatically invalidates every half a second

Note how we use `timer()` in the reactive expressions that compute `x1()` and `x2()`: we call it, but don't use the value. This lets `x1` and `x2` take a reactive dependency on `timer`, without worrying about exactly what value it returns.

---

[7]The New York Times used this technique particularly effectively in their article discussing how to interpret the jobs report: https://www.nytimes.com/2014/05/02/upshot/how-not-to-be-misled-by-the-jobs-report.html

5.5.2 On click

In the above scenario, think about what would happen if the simulation code took 1 second to run. We perform the simulation every 0.5s, so Shiny would have more and more to do, and would never be able to catch up. The same problem can happen if someone is rapidly clicking buttons in your app and the computation you are doing is relatively expensive. It's possible to create a big backlog of work for Shiny, and while it's working on the backlog, it can't respond to any new events. This leads to a poor user experience.

If this situation arises in your app, you might want to require the user to opt-in to performing the expensive calculation by requiring them to click a button. This is a great use case for an `actionButton()`:

```r
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 3),
      numericInput("n", label = "n", value = 1e4, min = 0),
      actionButton("simulate", "Simulate!")
    ),
    column(9, plotOutput("hist"))
  )
)
```

To use the action button we need to learn a new tool. To see why, let's first tackle the problem using the same approach as above. As above, we refer to `simulate` without using its value to take a reactive dependency on it.

```r
server <- function(input, output, session) {
  x1 <- reactive({
    input$simulate
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    input$simulate
    rpois(input$n, input$lambda2)
  })
  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

This yields the app in Figure **??** and reactive graph in Figure 10. This doesn't achieve our goal because it just introduces a new dependency: `x1()` and `x2()` will update when we click the simulate button, but they'll also continue to update when `lambda1`, `lambda2`, or `n` change. We want to *replace* the existing dependencies, not add to them.

To solve this problem we need a new tool: a way to use input values without taking a reactive dependency on them. We need `eventReactive()`, which has two arguments: the first argument specifies what to take a dependency on, and the second argument specifies what to compute. That allows this app to only compute `x1()` and `x2()` when `simulate` is clicked:

```r
server <- function(input, output, session) {
  x1 <- eventReactive(input$simulate, {
    rpois(input$n, input$lambda1)
```
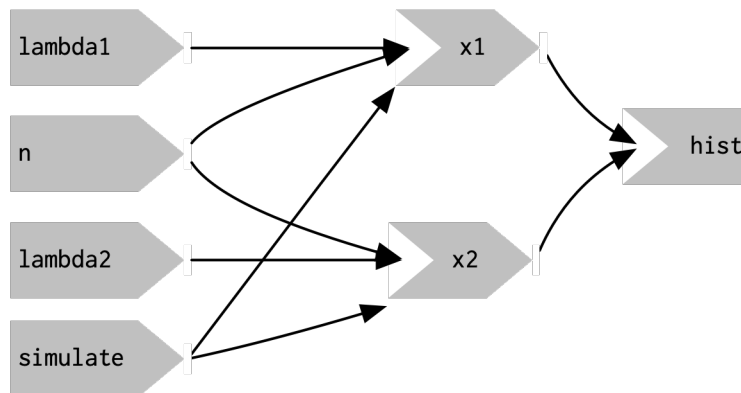
Figure 10: This reactive graph doesn't accomplish our goal; we've added a dependency instead of replacing the existing dependencies.

```
  })
  x2 <- eventReactive(input$simulate, {
    rpois(input$n, input$lambda2)
  })

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

Figure 11 shows the new reactive graph. Note that, as desired, x1 and x2 no longer have a reactive dependency on lambda1, lambda2, and n: changing their values will not trigger computation. I left the arrows in very pale grey just to remind you that x1 and x2 continue to use the values, but no longer take a reactive dependency on them.
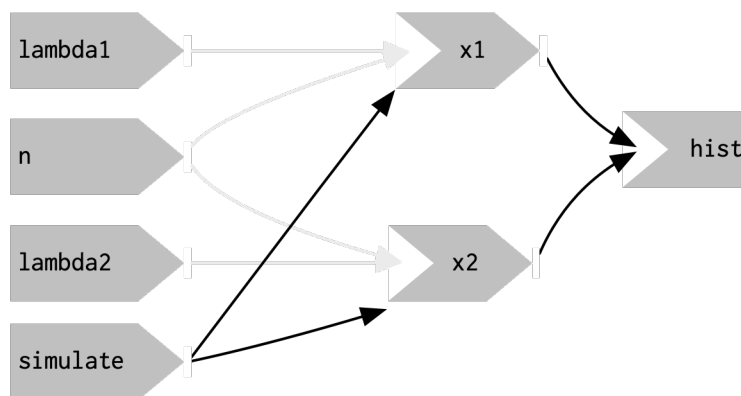

Figure 11: 'eventReactive()' makes it possible to separate the dependencies (black arrows) from the values used to compute the result (pale gray arrows).

5.6 Observers

So far, we've focused on what's happening inside the app. But sometimes you need to reach outside of the app and cause side-effects to happen elsewhere in the world. This might be saving a file to a shared network drive, sending data to a web API, updating a database, or (most commonly) printing a debugging message to the console. These actions don't affect how your app looks, so you can't use an output and a `render` function. Instead you need to use an **observer**.

There are multiple ways to create an observer, and we'll come back to them later in Chapter XYZ. For now, I wanted to show you how to use `observeEvent()`, because it gives you an important debugging tool when you're first learning Shiny.

`observeEvent()` is very similar to `eventReactive()`. It has two important arguments: `eventExpr` and `handlerExpr`. The first argument is the input or expression to take a dependency on; the second argument is the code that will be run. For example, the following modification to `server()` means that every time that `name` is updated, a message will be sent to the console:

```
server <- function(input, output, session) {
  text <- reactive(paste0("Hello ", input$name, "!"))

  output$greeting <- renderText(text())
  observeEvent(input$name, {
    message("Greeting performed")
  })
}
```

There are two important differences between `observeEvent()` and `eventReactive()`:

- You don't assign the result of `observeEvent()` to a variable, so
- You can't refer to it from other reactive consumers.

Observers and outputs are closely related. You can think of outputs as having a special side-effect: updating the HTML in the user's browser. To emphasise this closeness, we'll draw them the same way in the reactive graph. This yields the following reactive graph shown in Figure 12.
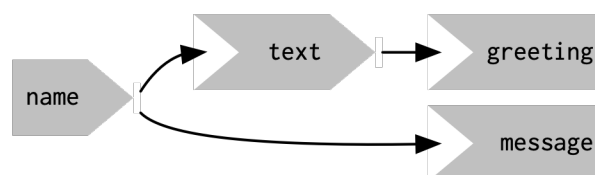

Figure 12: In the reactive graph, an observer looks the same as an output