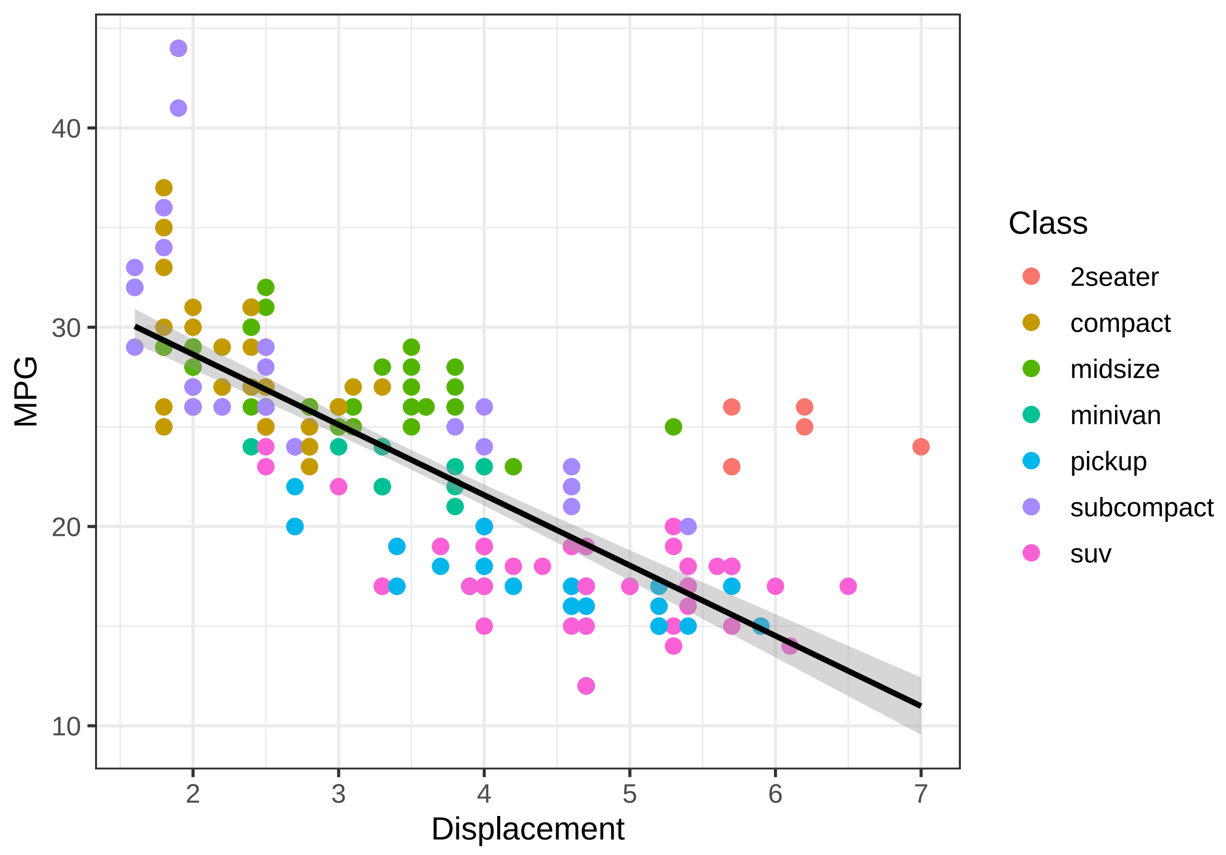
Data Visualization and Shiny Apps with R

88th MORS Symposium

Contents

1 Data Visualization with ggplot2



1.1 About the tidyverse

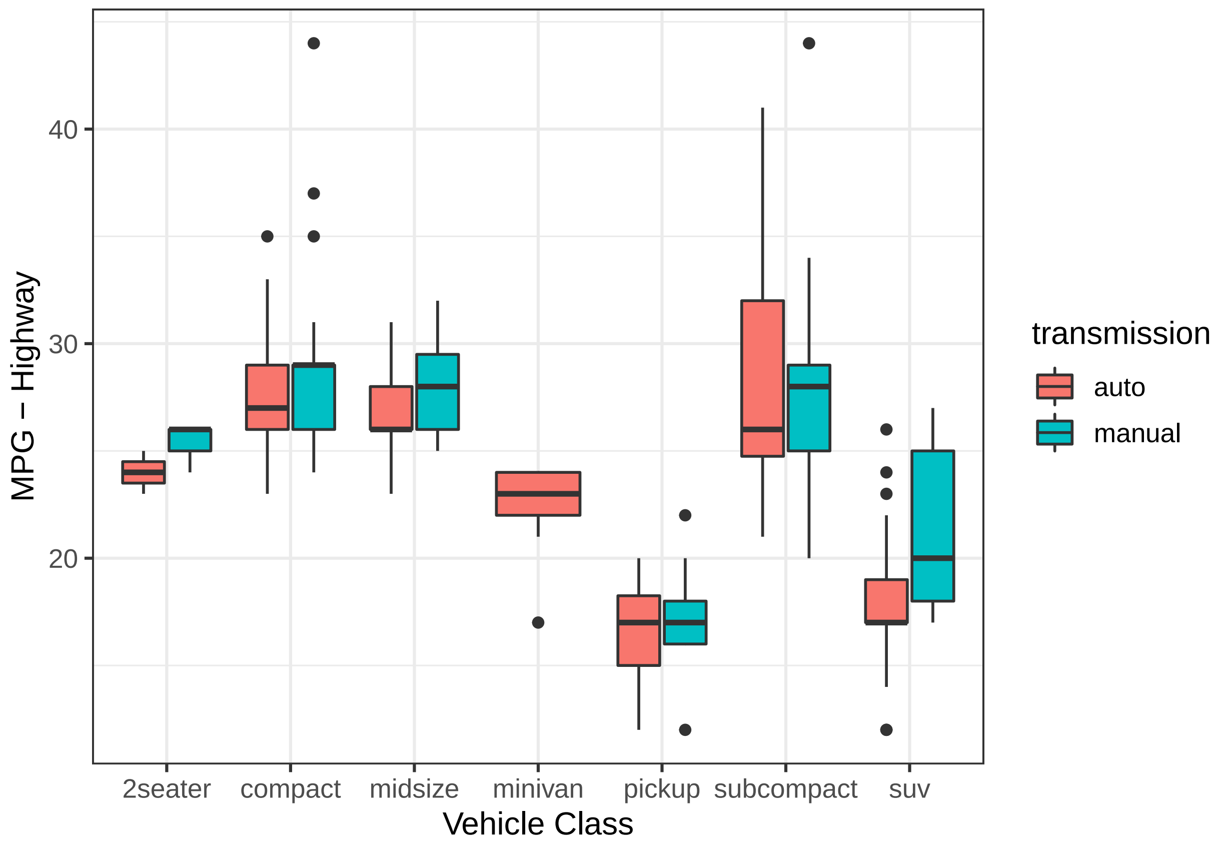
The data visualization package, ggplot2, is a part of the tidyverse.

The tidyverse packages:

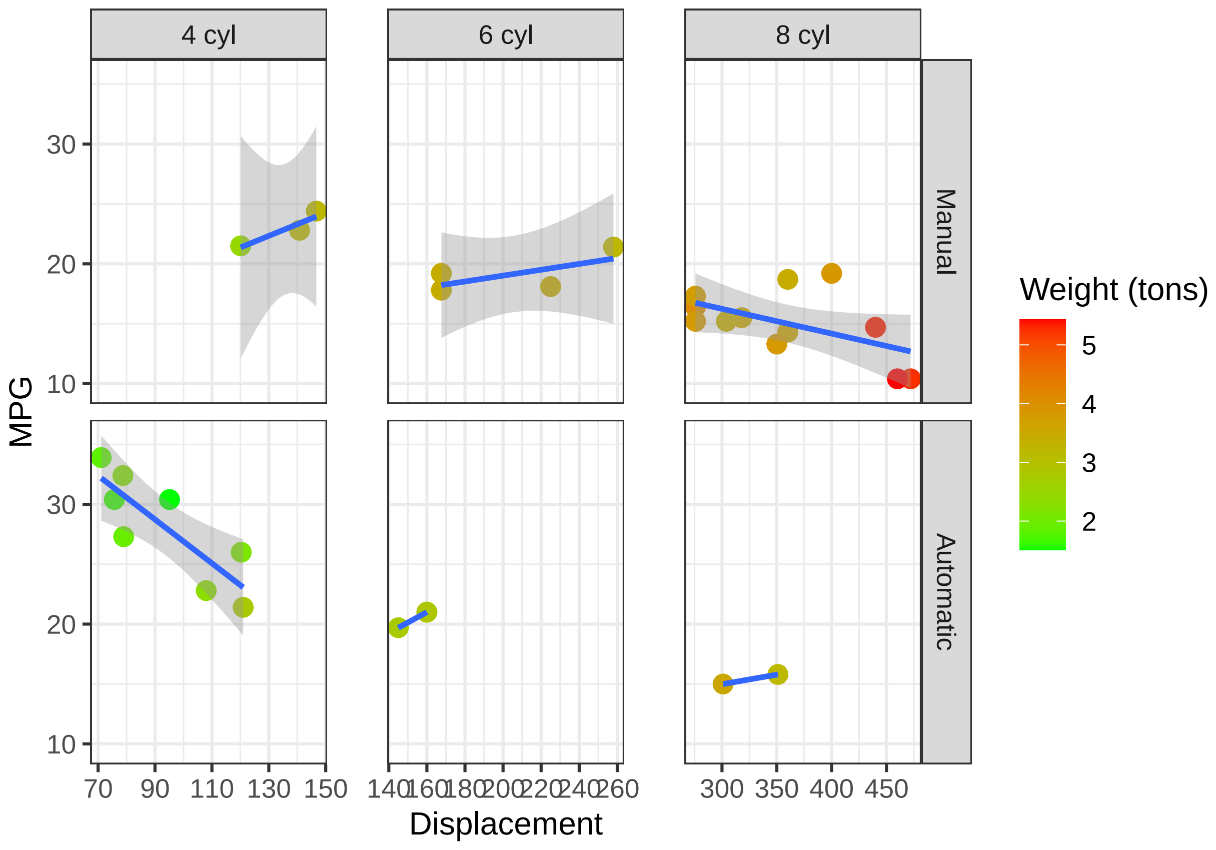
- Are a collection of R packages for designed for data science
- Were developed on the programming philosophy of statistician and avid R user Hadley Wickam
- Share the same underlying design, grammar, and data structures

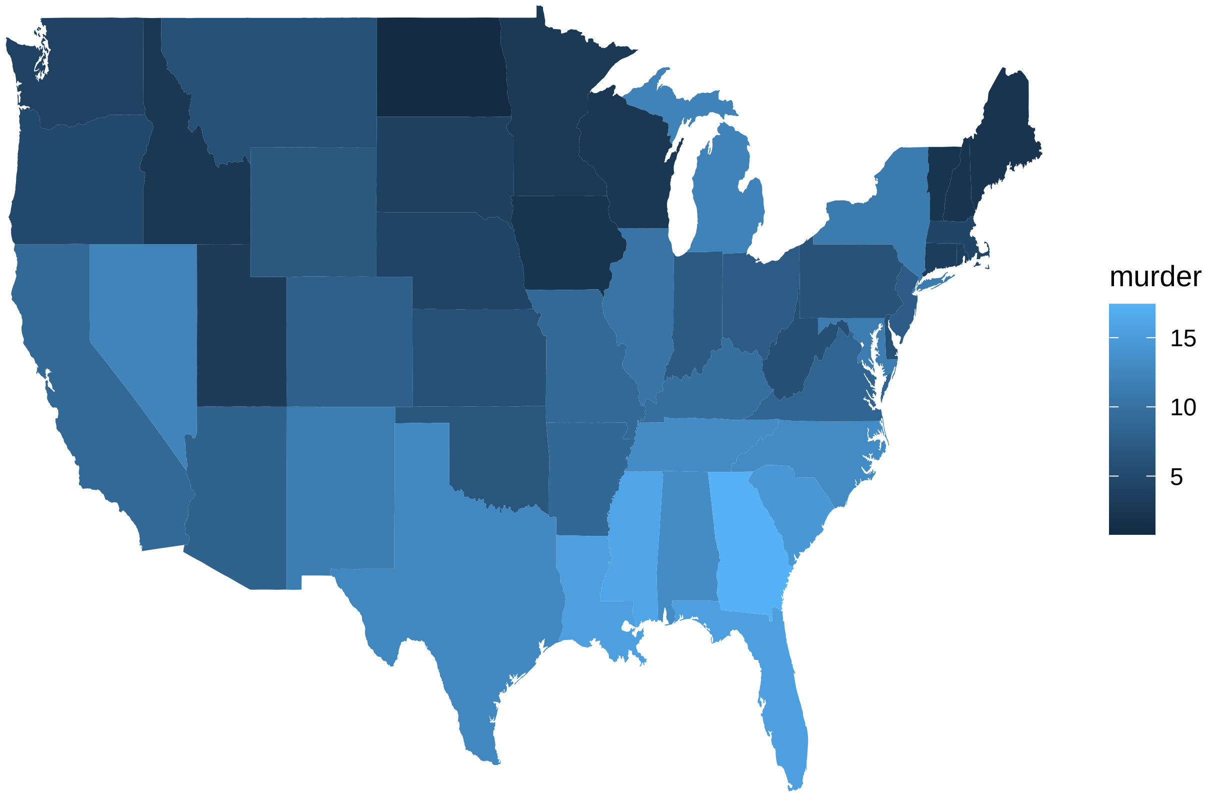
1.2 ggplot2

ggplot2 can produce a large variety of two-dimensional plot types, including faceted plots and maps.



MPG – Highway
40
30
20
2seater
compact
midsize
minivan
pickup
subcompact
suv
Vehicle Class
transmission
auto
manual


4 cyl
6 cyl
8 cyl
Manual
Automatic
MPG
30
20
10
30
20
10
70
90
110
130
150
140
160
180
200
220
240
260
300
350
400
450
Displacement
Weight (tons)
5
4
3
2

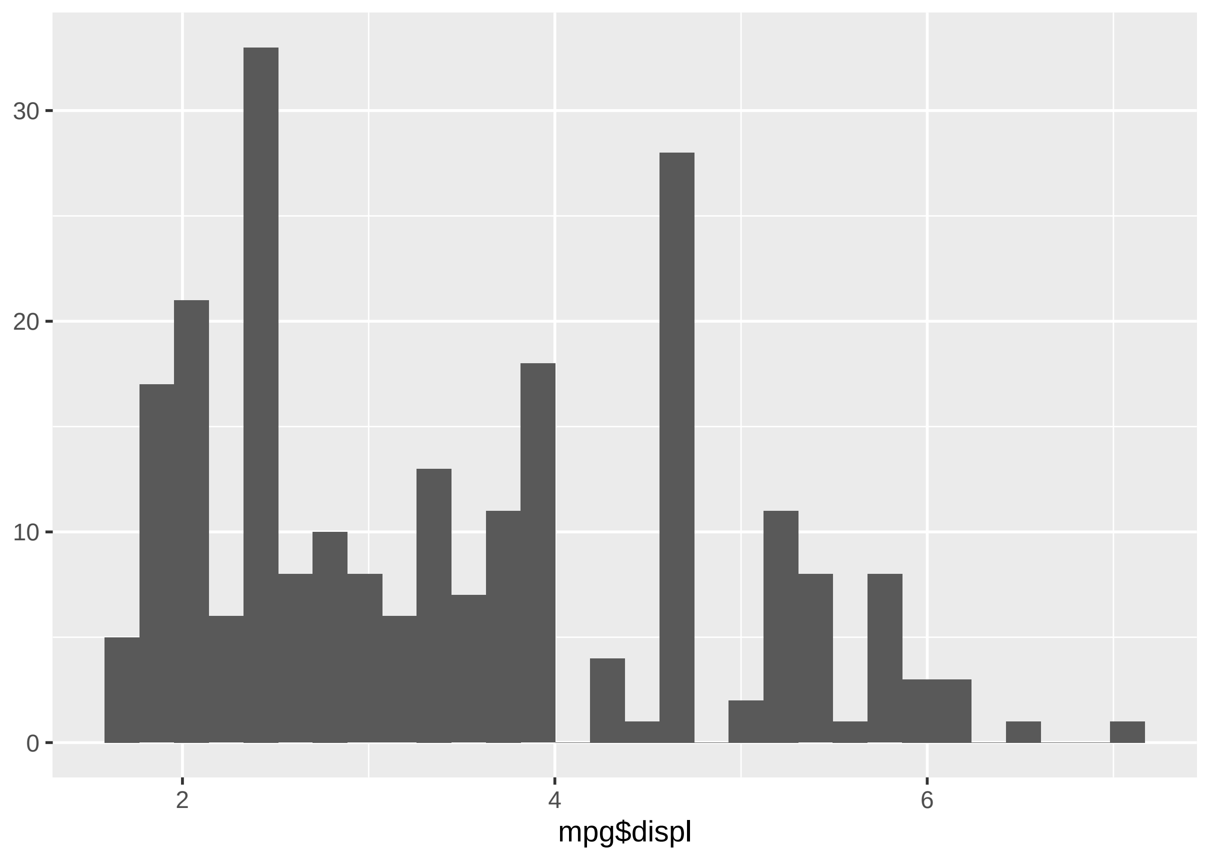


1.3 `qplot()`

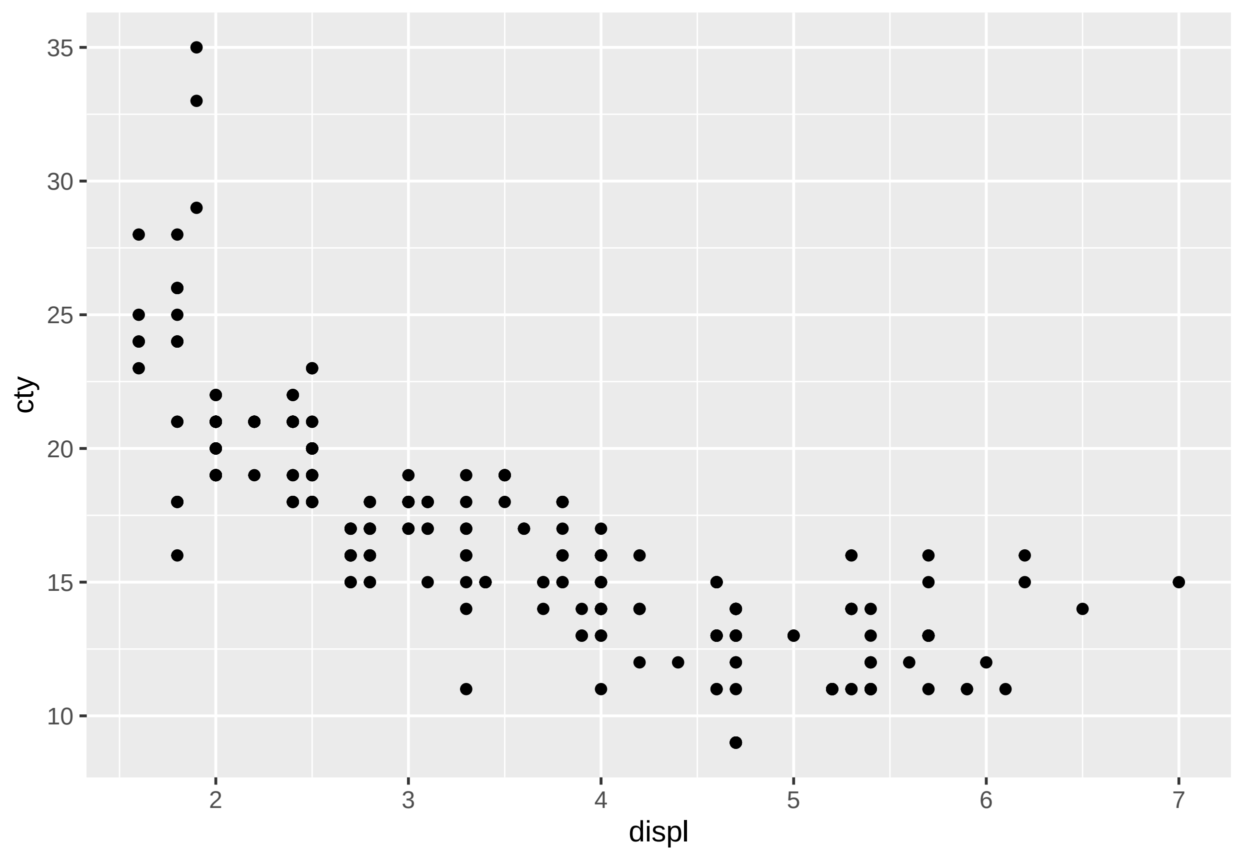
The quickplot, or`qplot()` function, is an easy way to generate plots in a single function.

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

```
qplot(x=mpg$displ)
```



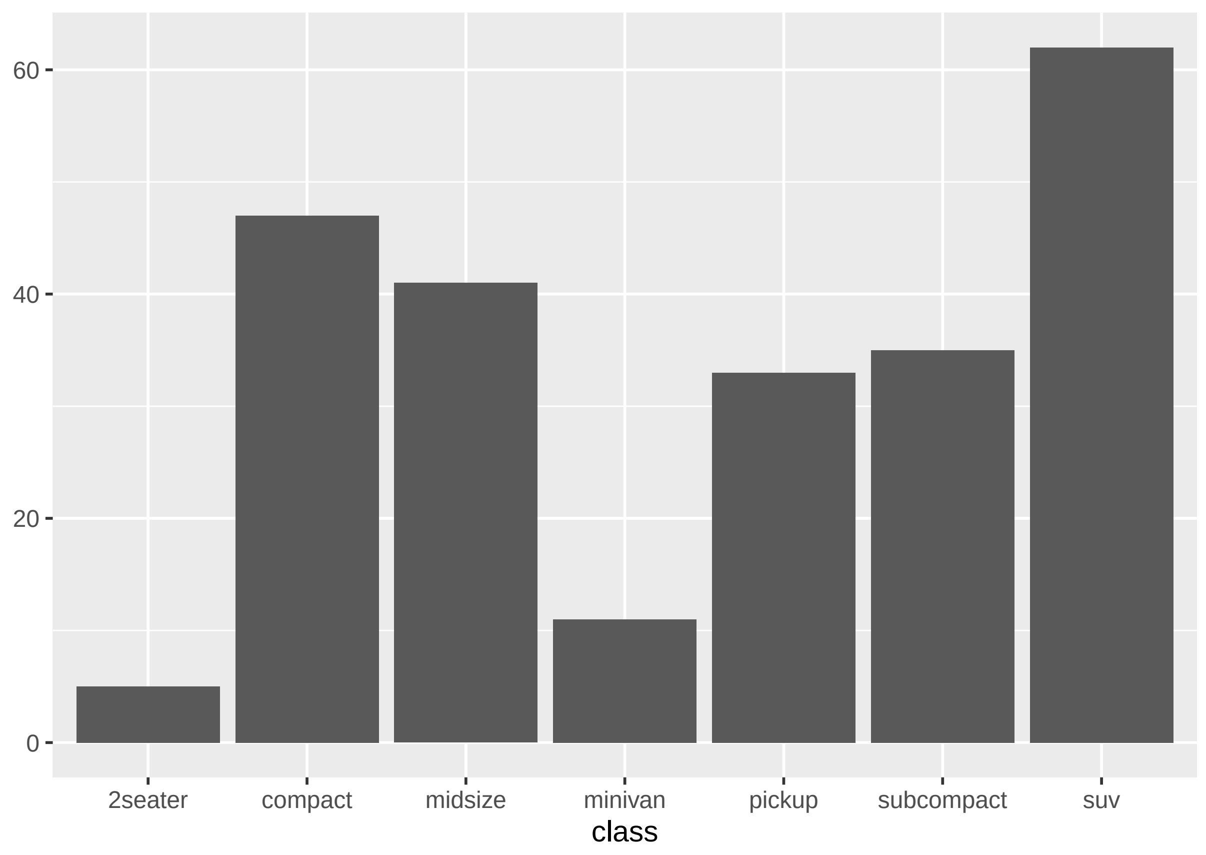
```r
qplot(x=displ, y=cty, data=mpg)
```



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")
```

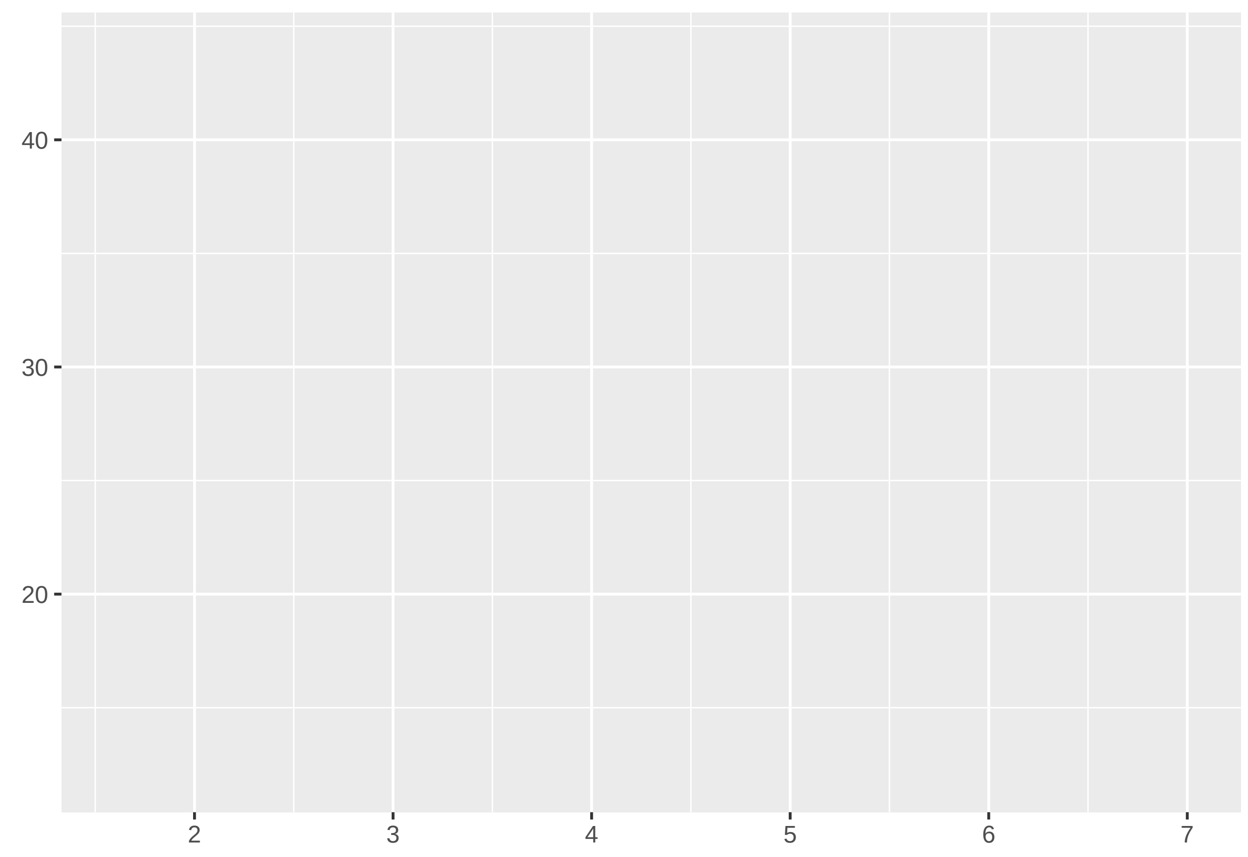


While it is possible to produce more complex plots using `qplot()`, it is generally better to use `ggplot()` for full control of all plot elements.

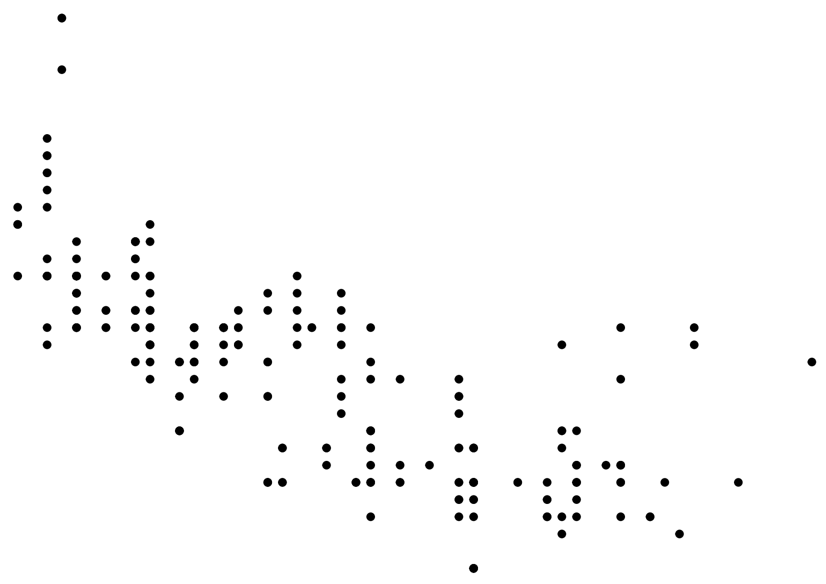
1.4 ggplot2 Mechanics

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

Coordinate System



Geometric Objects



Plot Annotations

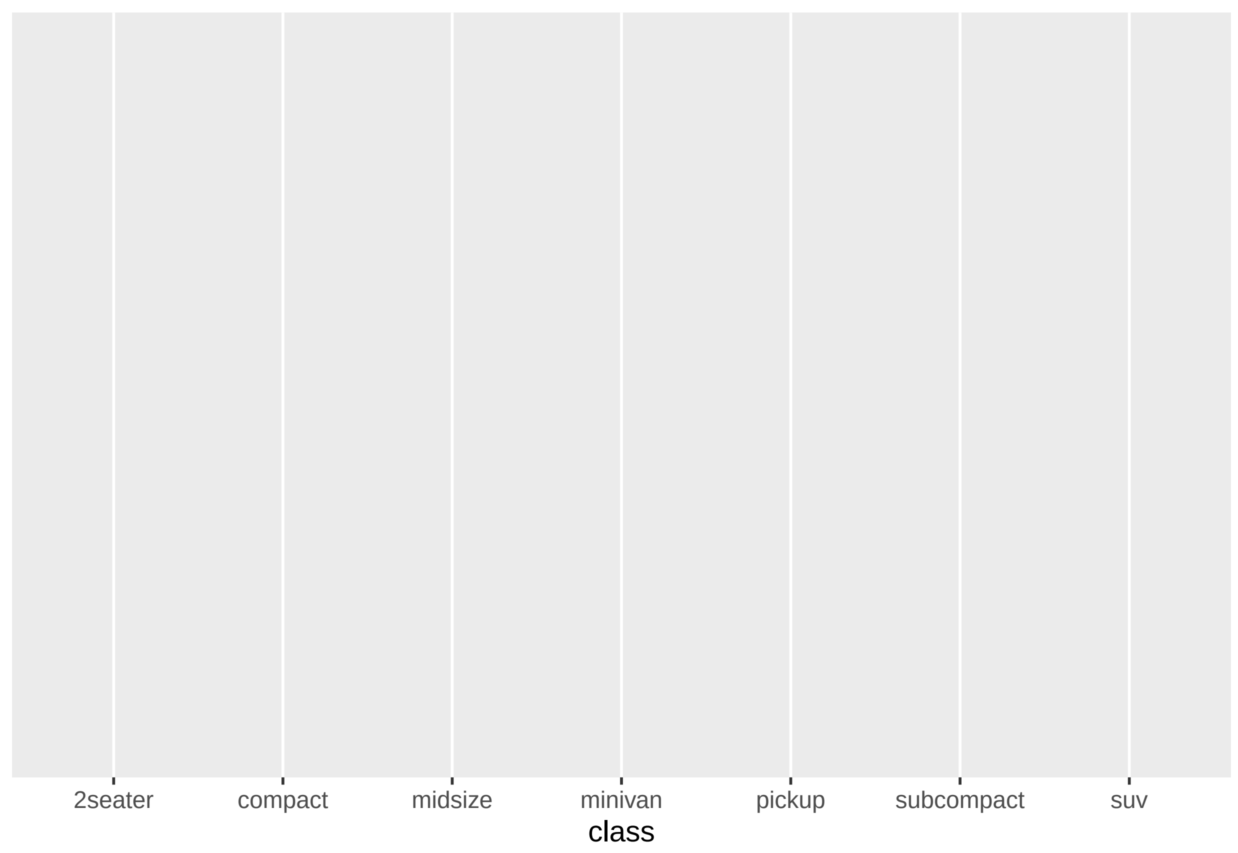
Highway MPG
Displacement

1.5 Aesthetic mapping

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

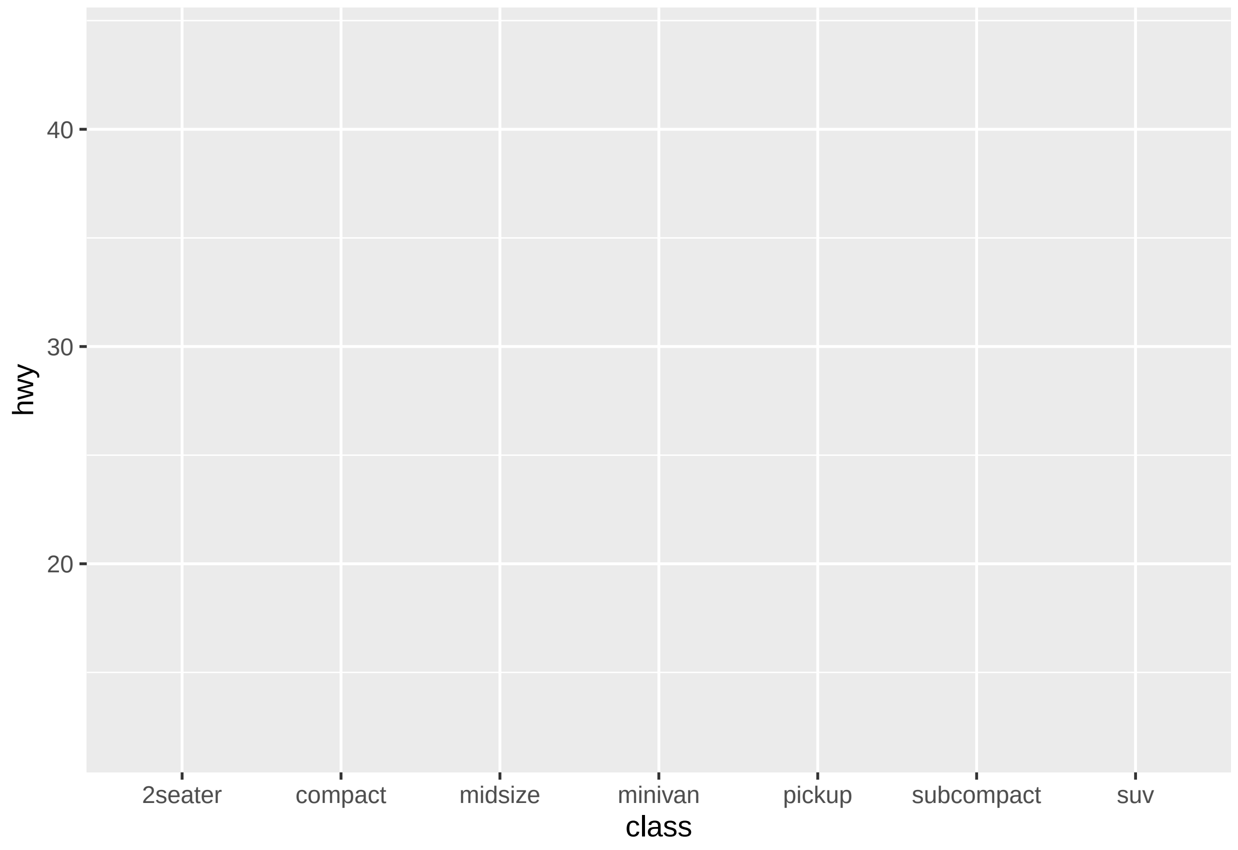
Single discrete variable:

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



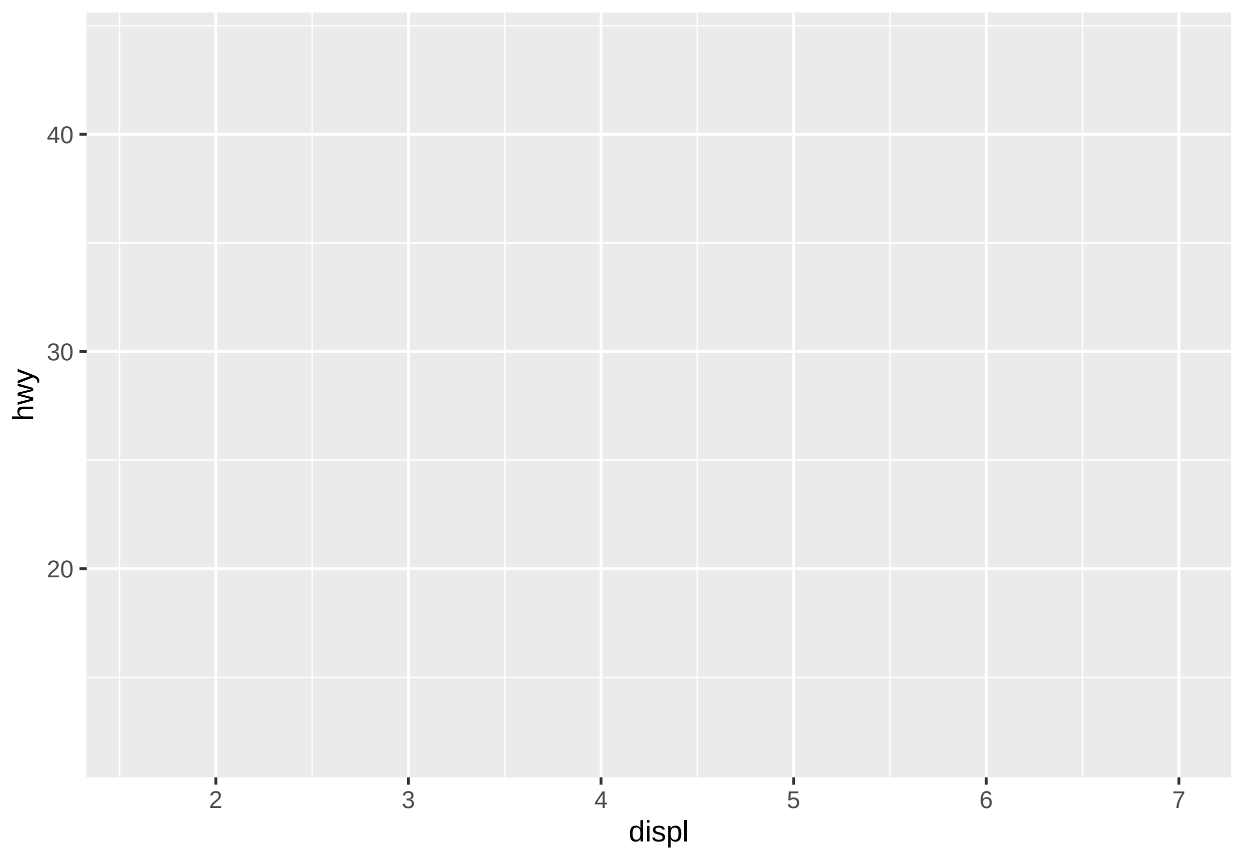
Discrete x variable, continuous y variable:

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



Continuous x and y variables:

```r
ggplot(data=mpg, aes(x = displ, y = hwy))
```



1.6 Adding geometric objects

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

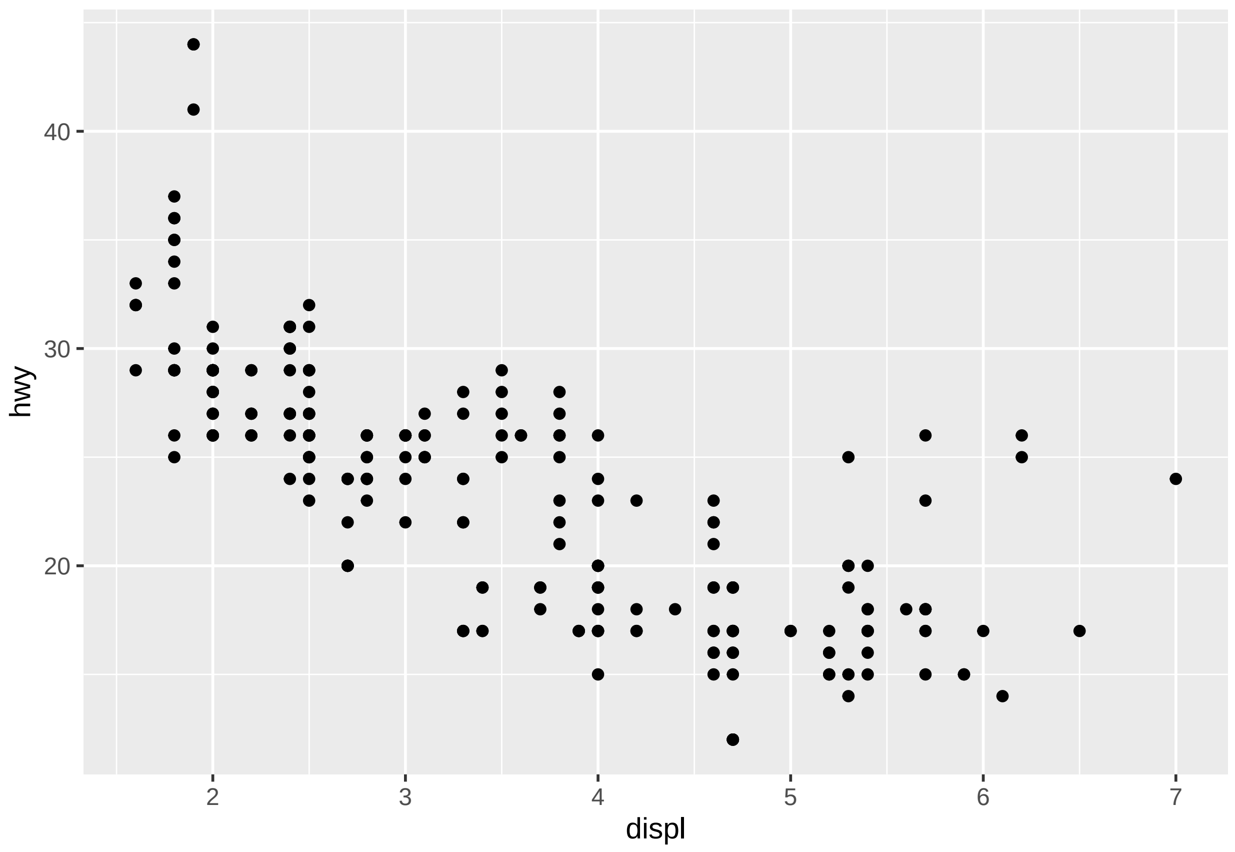
Some common types:

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

For a more comprehensive list, see the ggplot2 cheatsheet

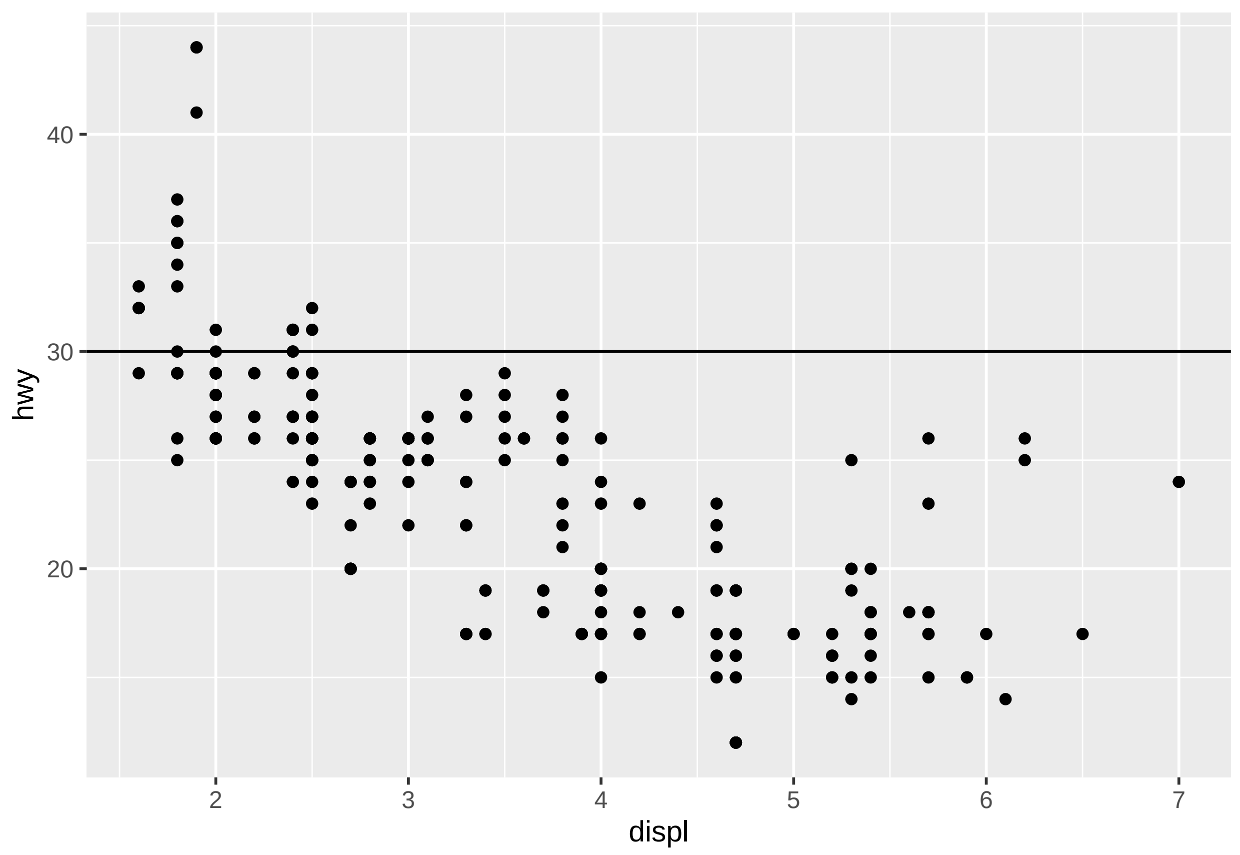
Example using `geom_point`:

```
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)
```



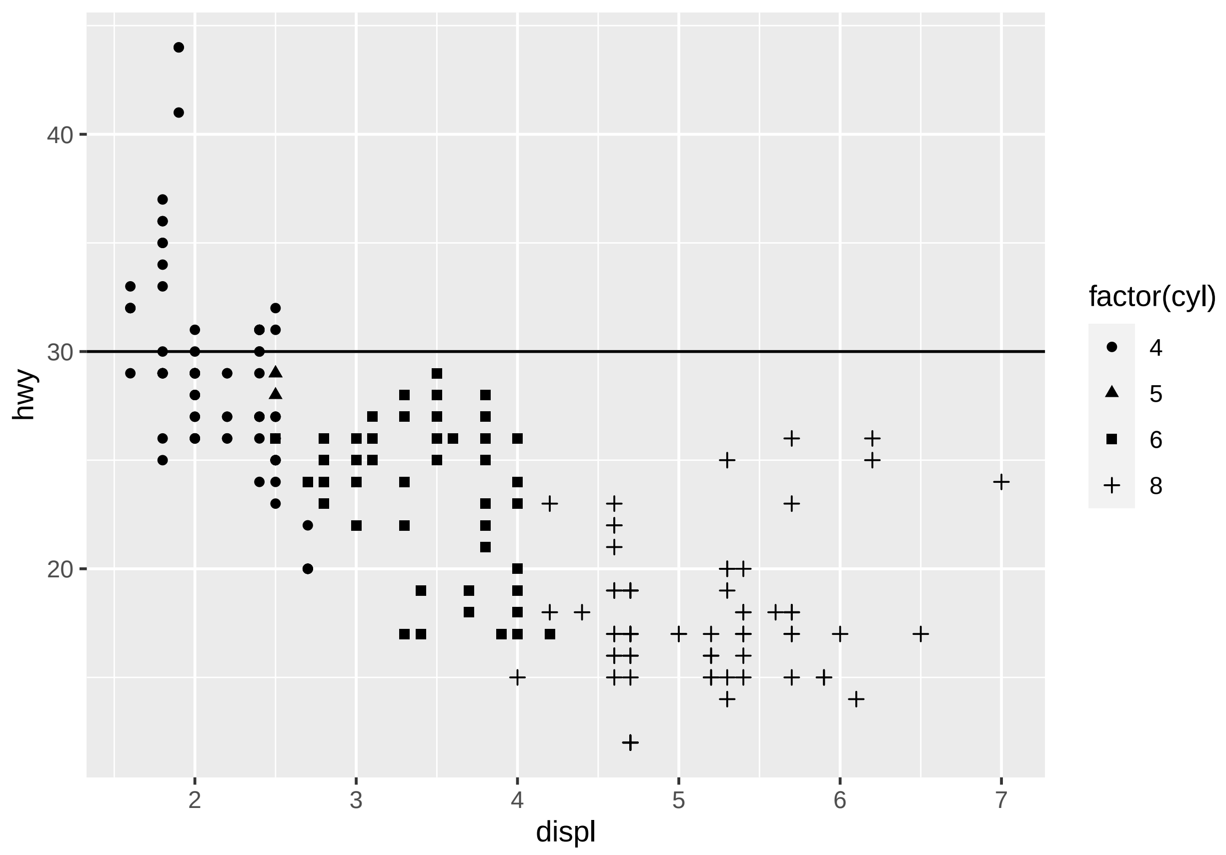
1.7 Adding aesthetic mappings

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

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

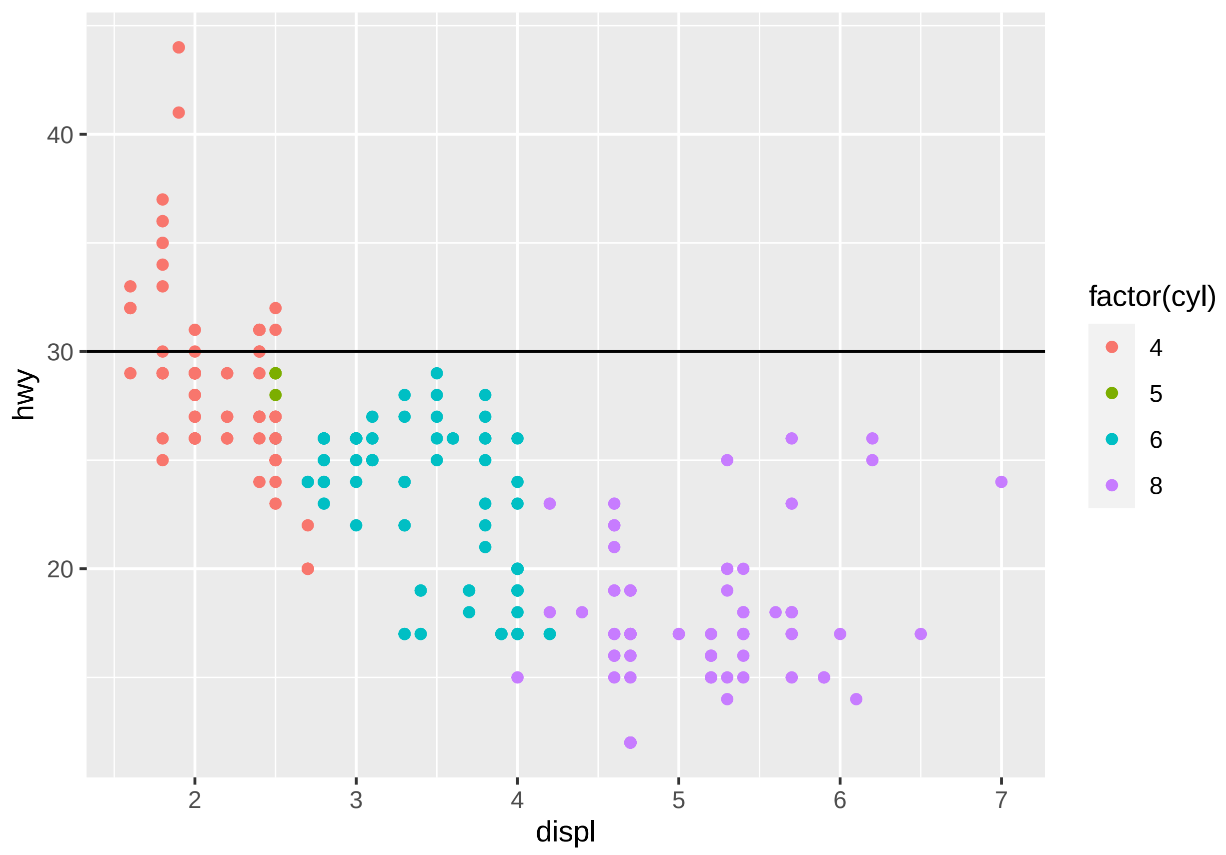
Point shape mapped to `cyl`:

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



Point color mapped to `cyl`:

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



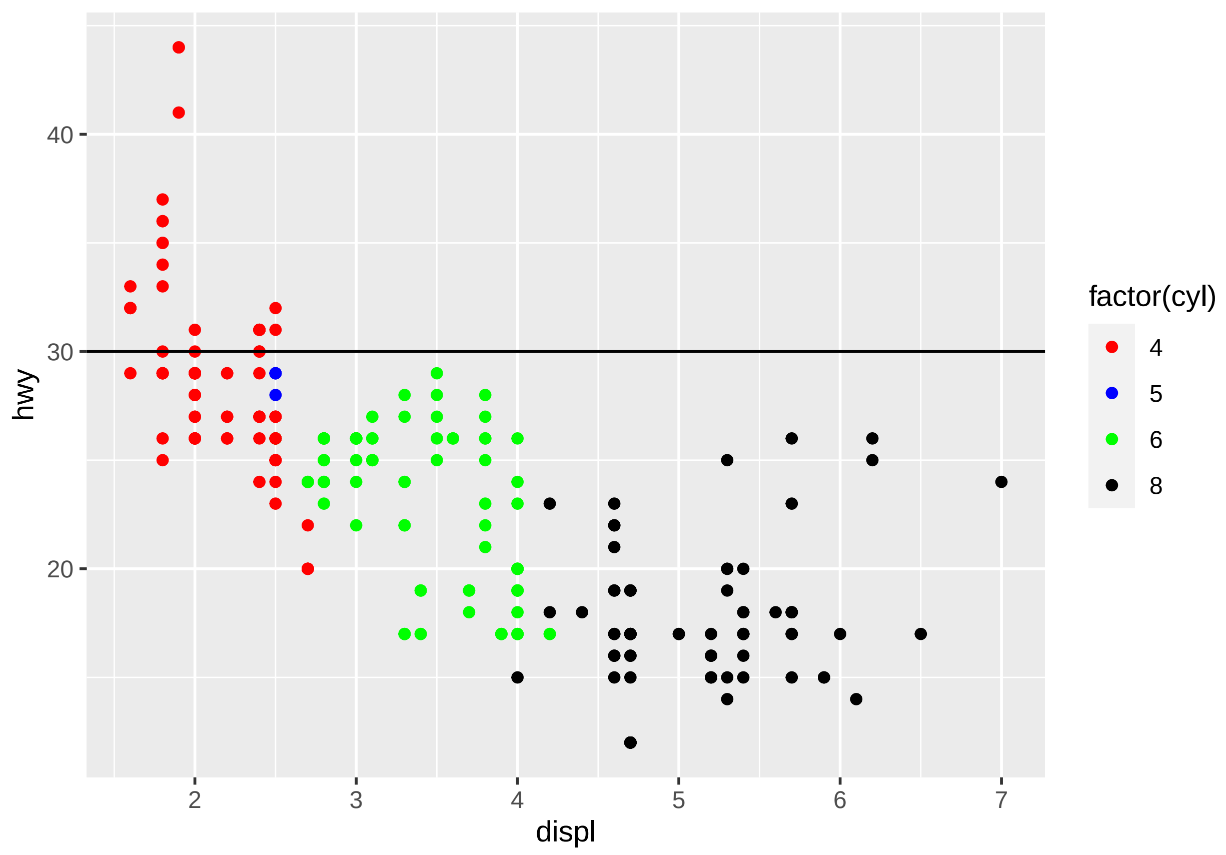
1.8 Changing Scales

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.

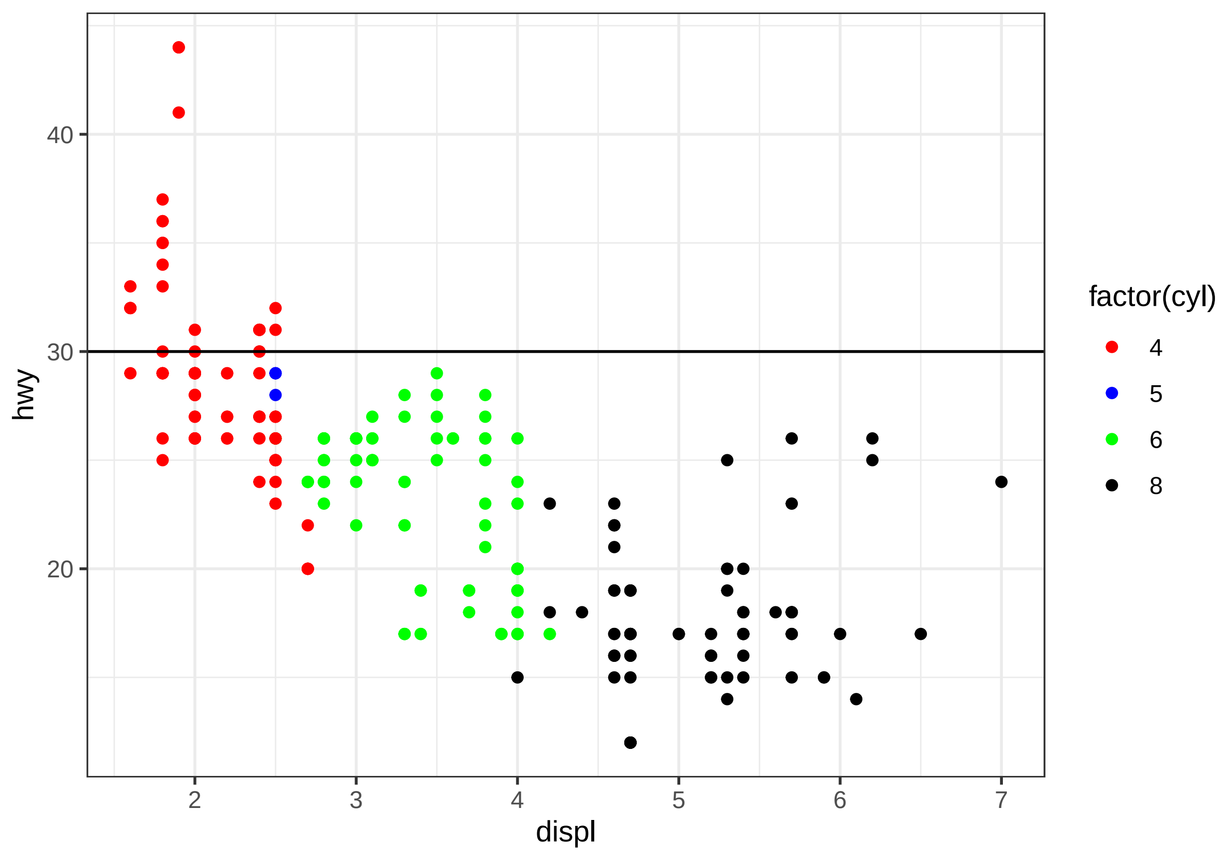
1.9 Preset themes

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

Examples:

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

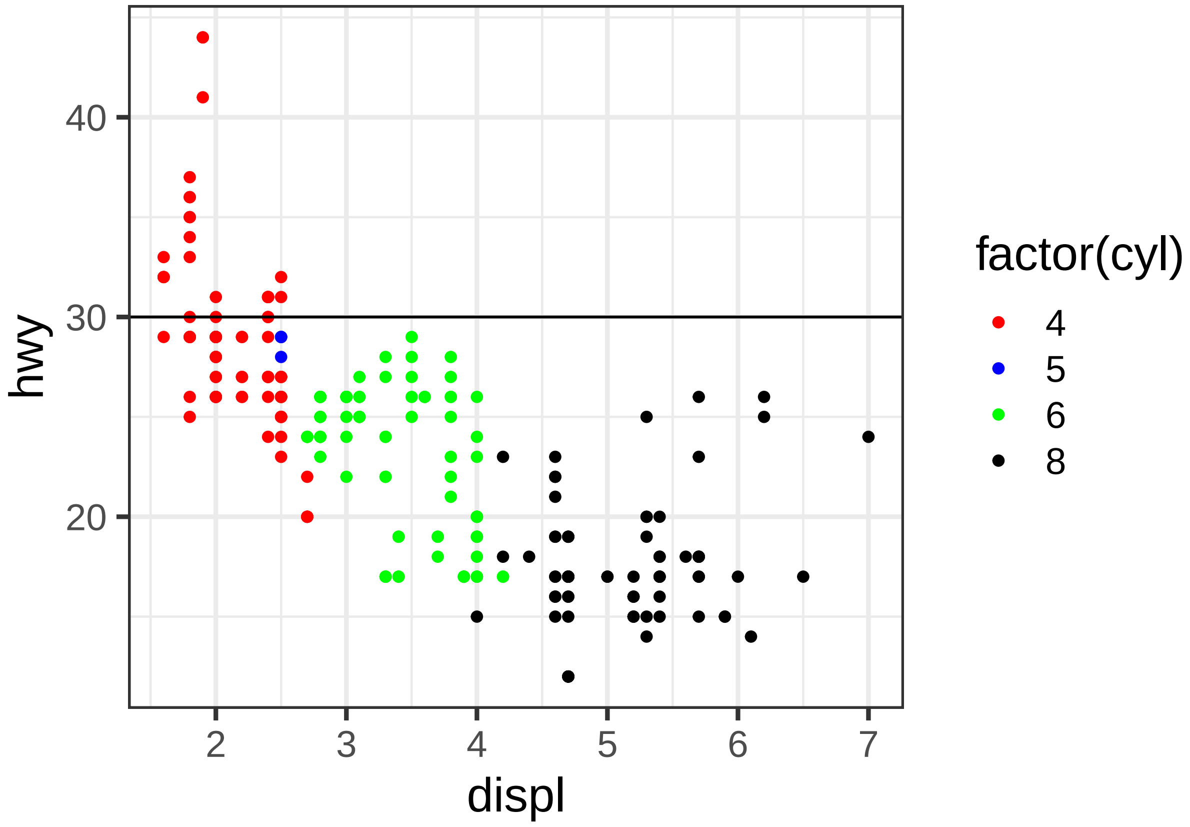
```
mpgscatter + scale_color_manual(values = c('red', 'blue', 'green', 'black')) +
  theme_bw()
```



1.9.1 Base size

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

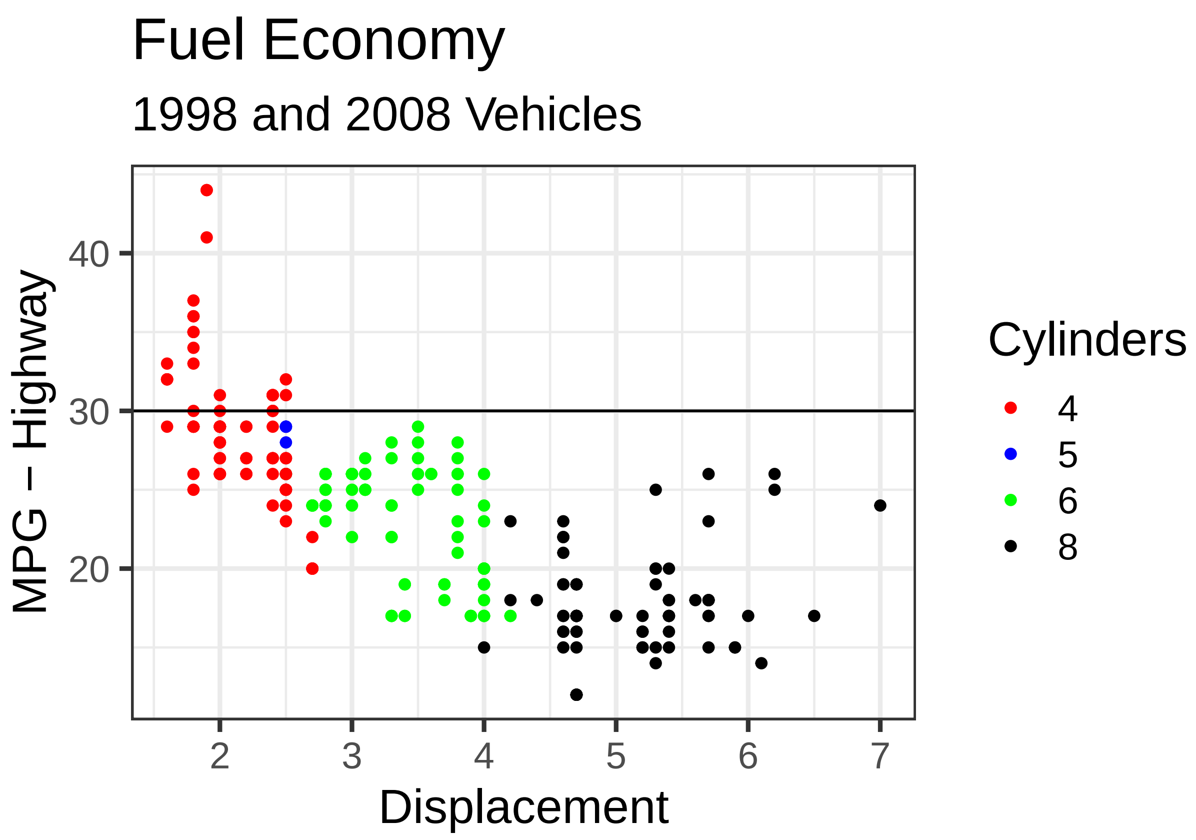
```
mpgscatter + scale_color_manual(values = c('red', 'blue', 'green', 'black')) +
  theme_bw(base_size=18)
```



1.10 Labels

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

```r
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")
```



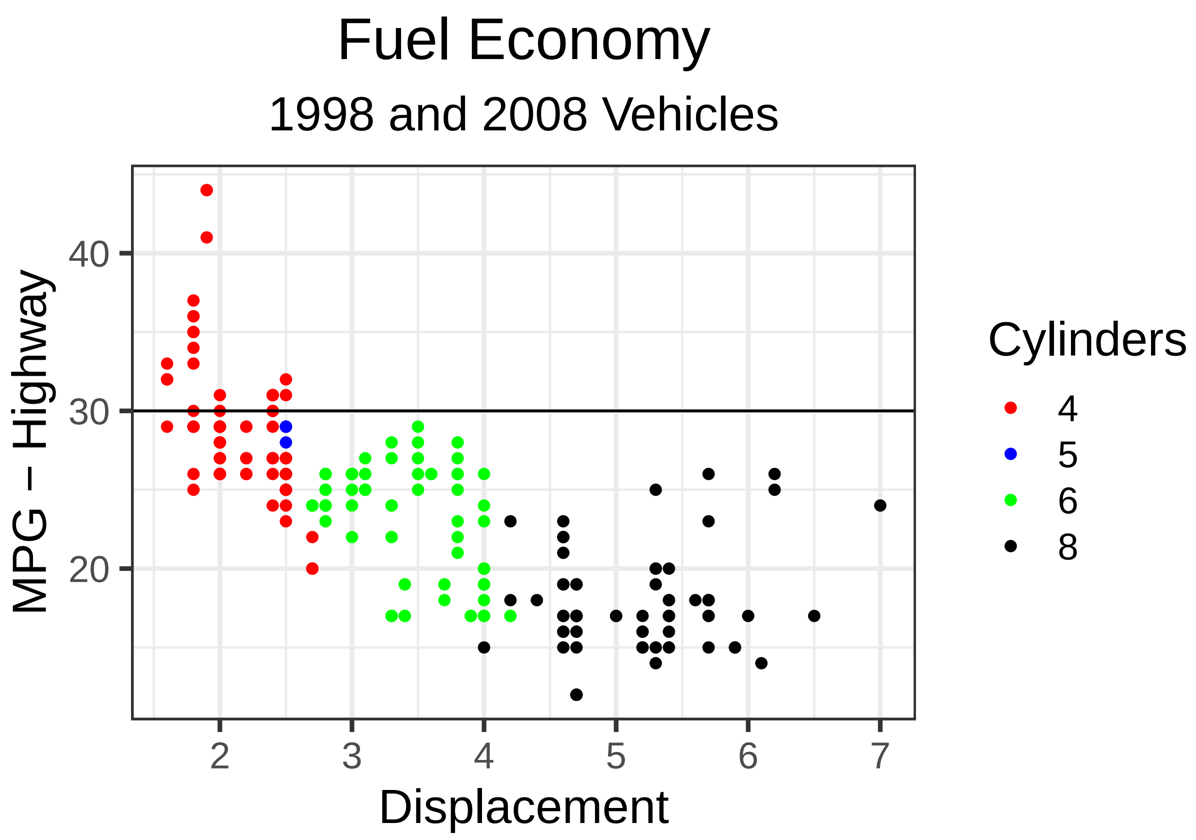
Note that the legend title is named by its mapping. It is possible to have multiple legends on a single plot.

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
```

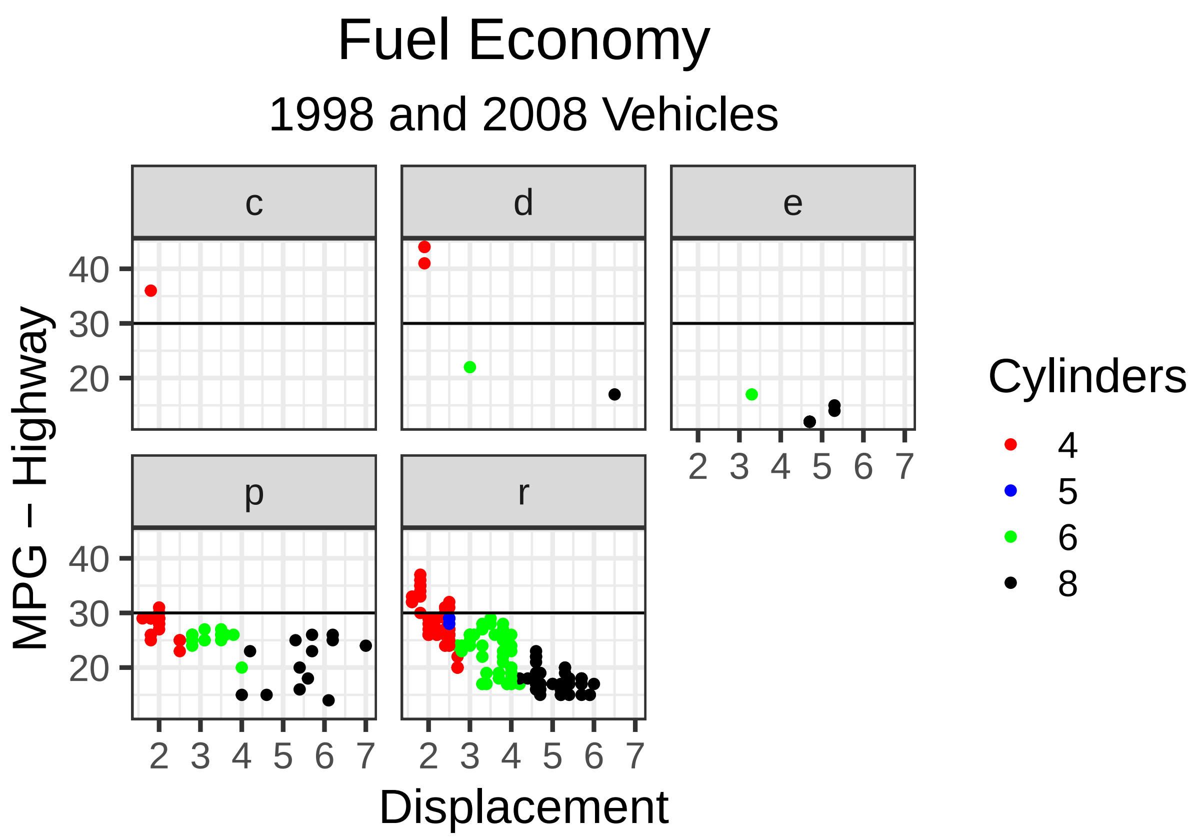


1.12 Facets

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

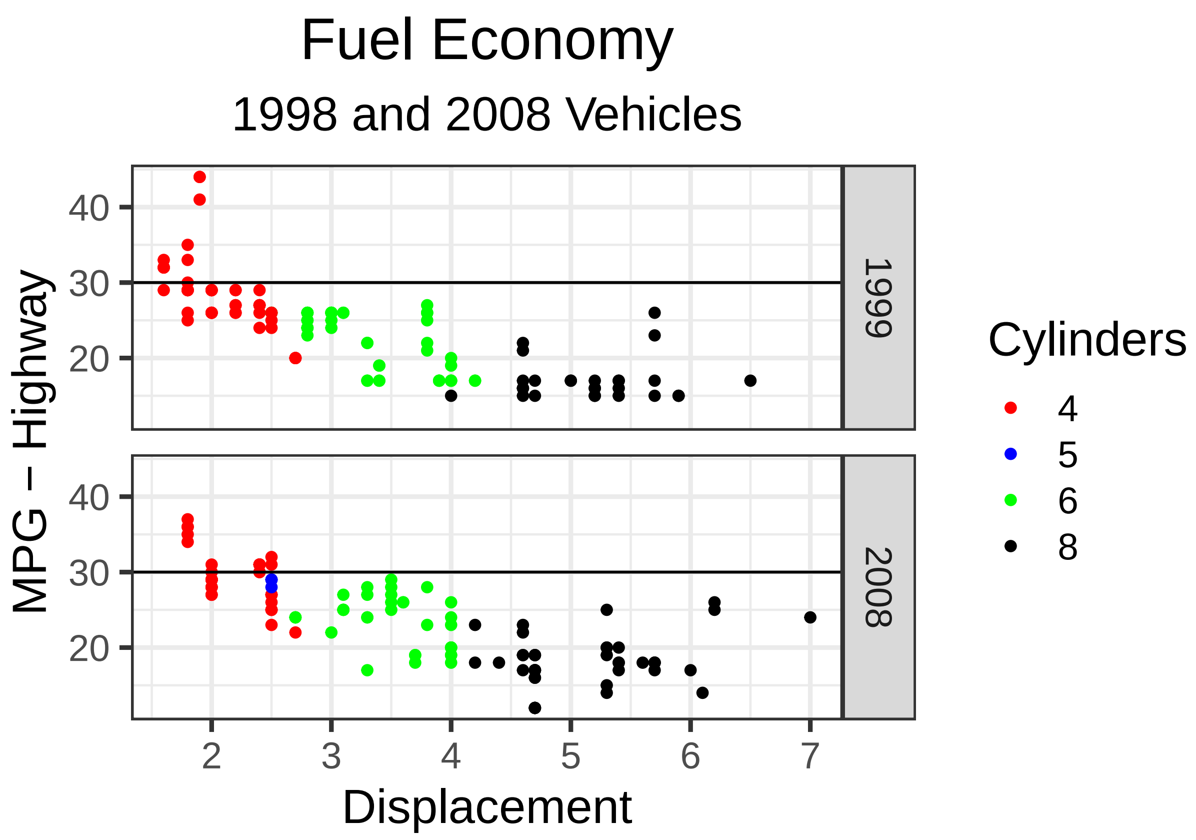
`facet_wrap()` creates rectanglar layout

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

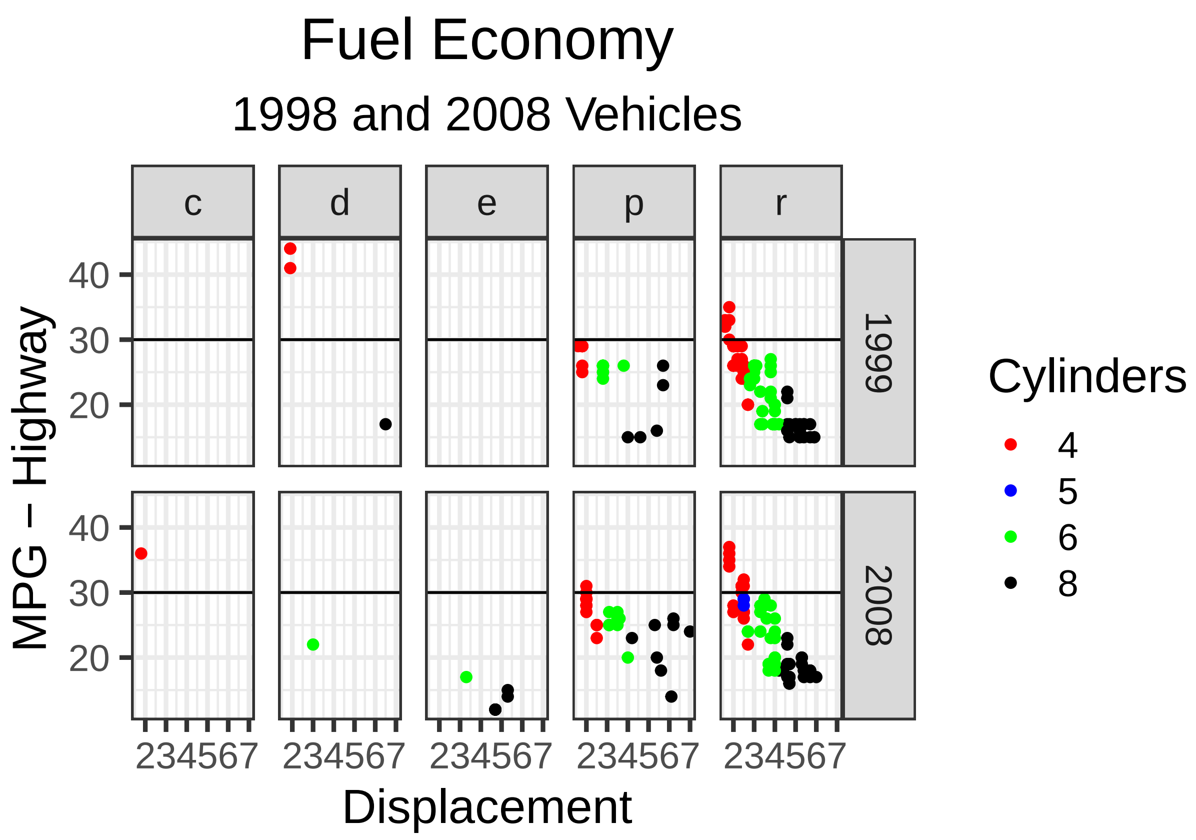


`facet_grid()` allows you to specify rows, columns, or both

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



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



1.13 References

Gitbooks:

R Graphics Cookbook R for Data Science

Article:

A Layered Grammar of Graphics

ggplot2:

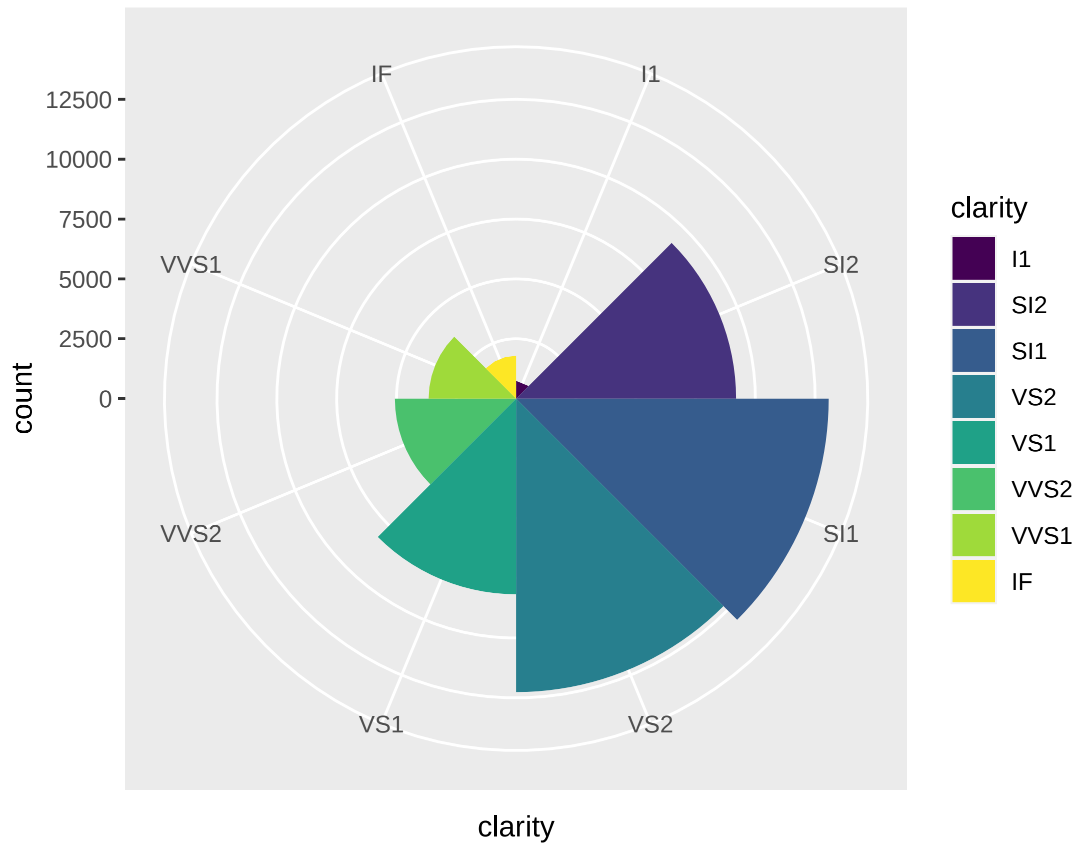
ggplot2 Cheatsheet ggplot2 Documentation

Websites:

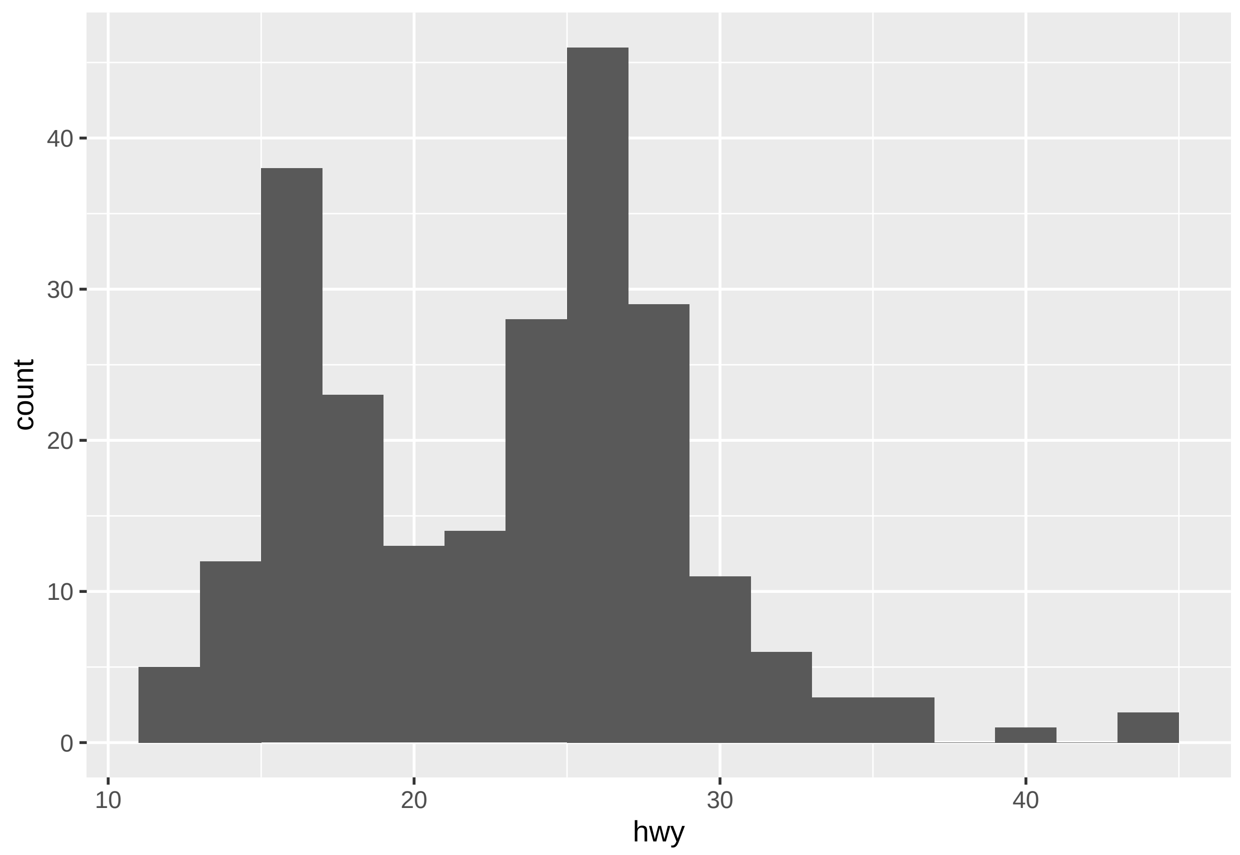
tidyverse Hadley Wickam

1.14 Additional plots

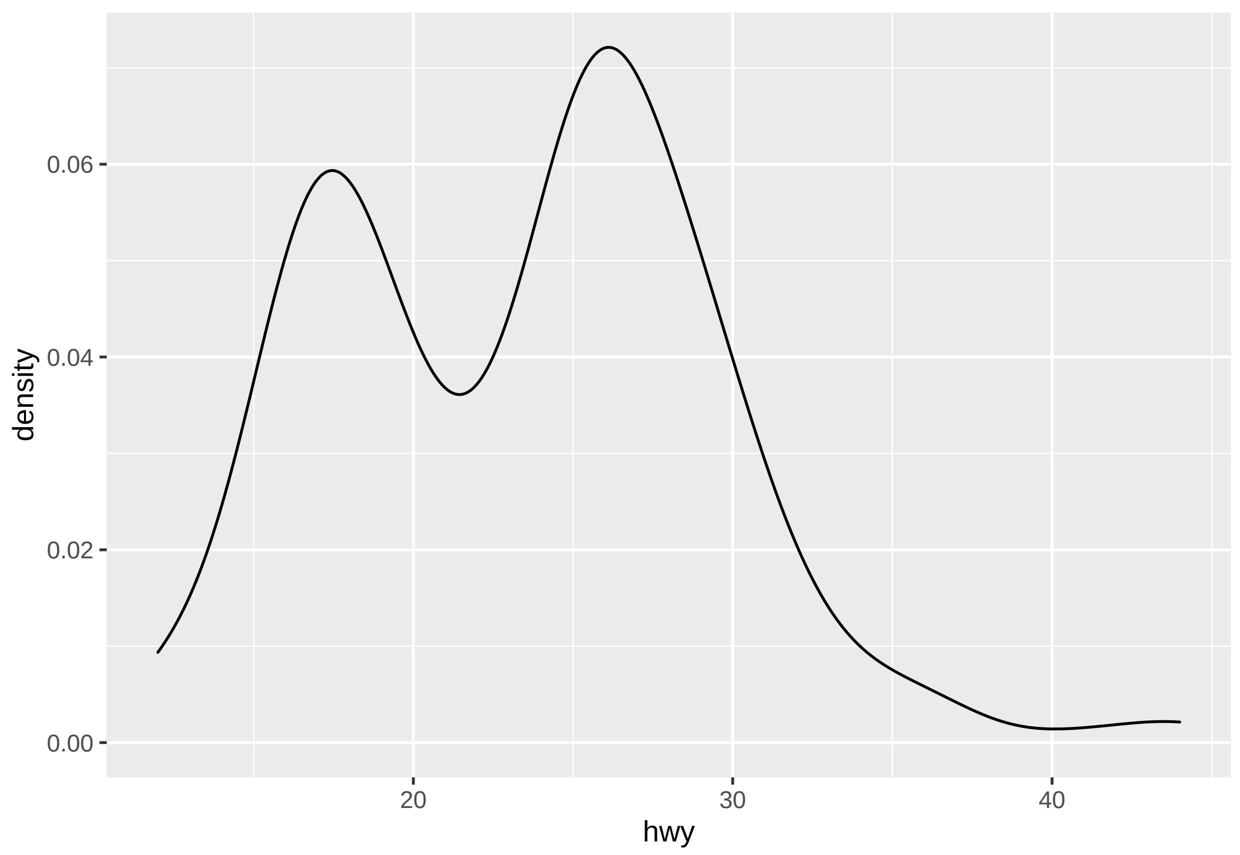
```
cxc <- ggplot(diamonds,aes(x = clarity, fill=clarity)) +
geom_bar(width = 1)
cxc + coord_polar()
```



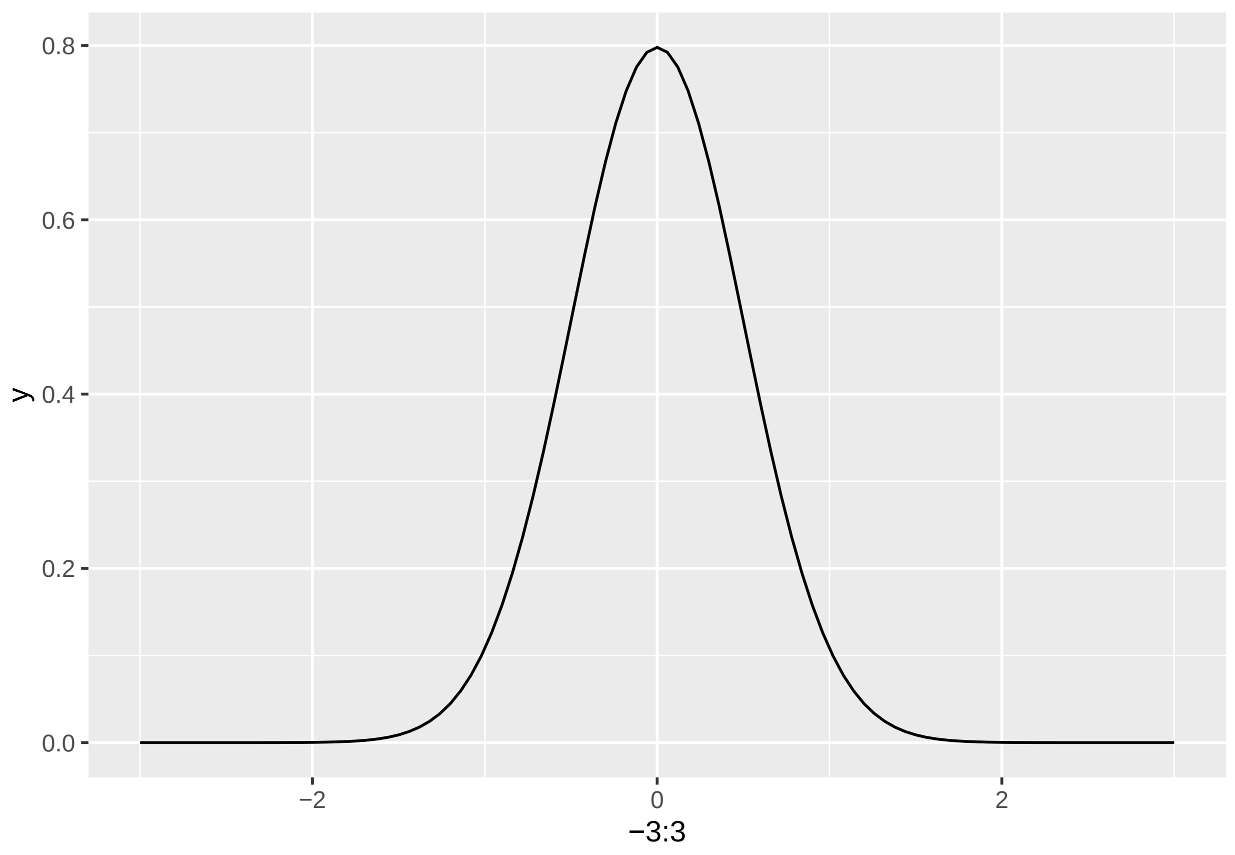
```
onevar <- ggplot(mpg, aes(x=hwy))
onevar + geom_histogram(binwidth = 2)
```



```
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.

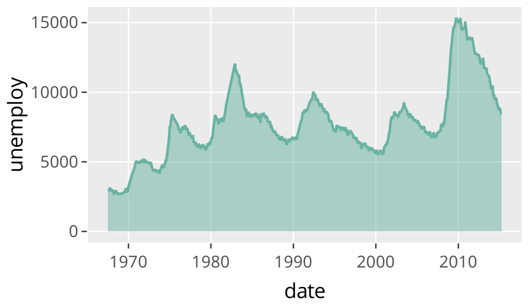
Plotly for R includes a handy wrapping function for `ggplot()` outputs called `ggplotly`.

plotly

2.1 `ggplotly`

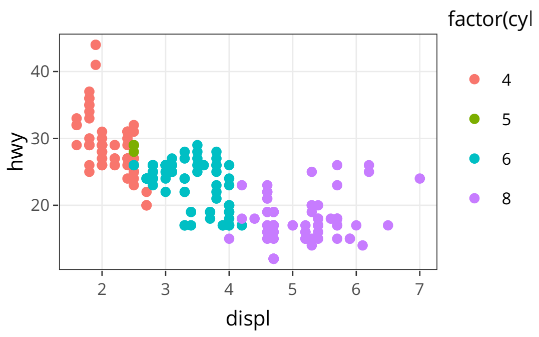
Simply save a `ggplot()` output to a variable. Then pass the variable to `ggplotly` for ineractive plots.

```
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.

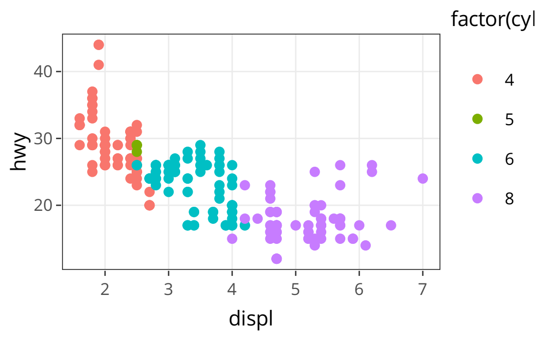
```
hwy <- ggplot(data=mpg, aes(x = displ, y = hwy, text = manufacturer)) + geom_point(aes(color = factor(cy

ggplotly(hwy)
```



## Custom Tooltips

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

```
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`

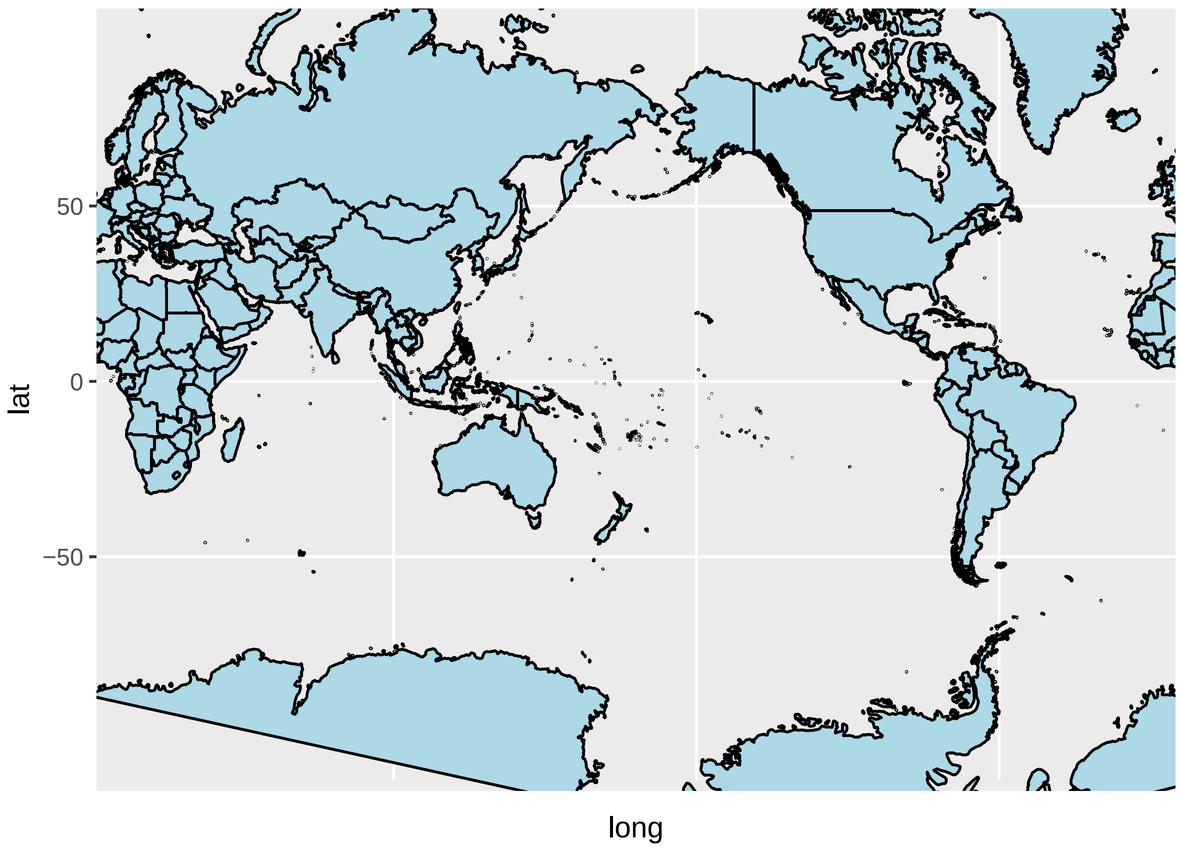
The `map_data()` function creates a data frame of map data.

```
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>
```

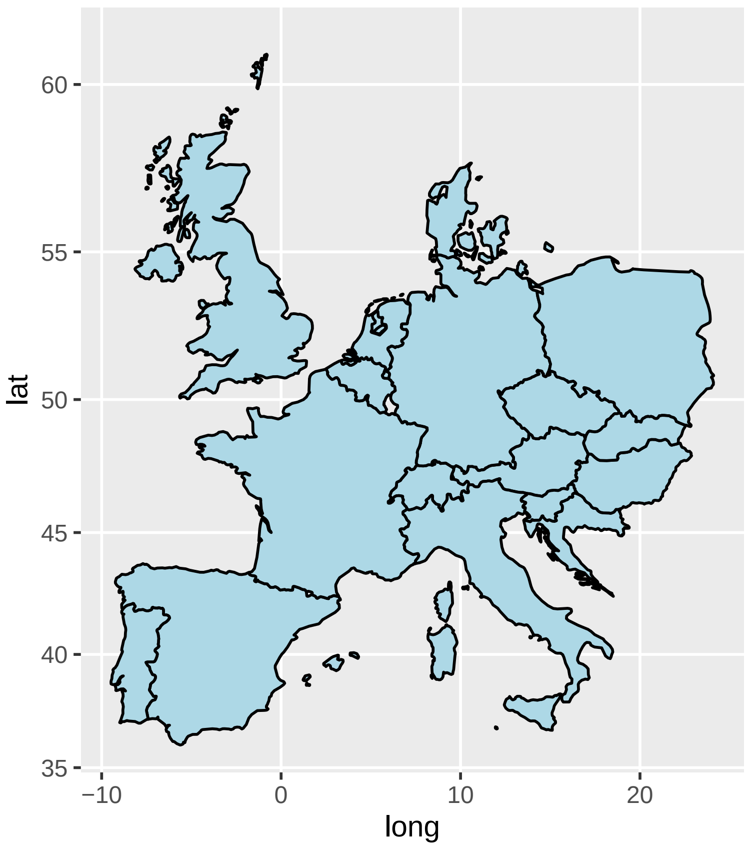
One way to create basic maps is to use `geom_polygon()`

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 = worl
```



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

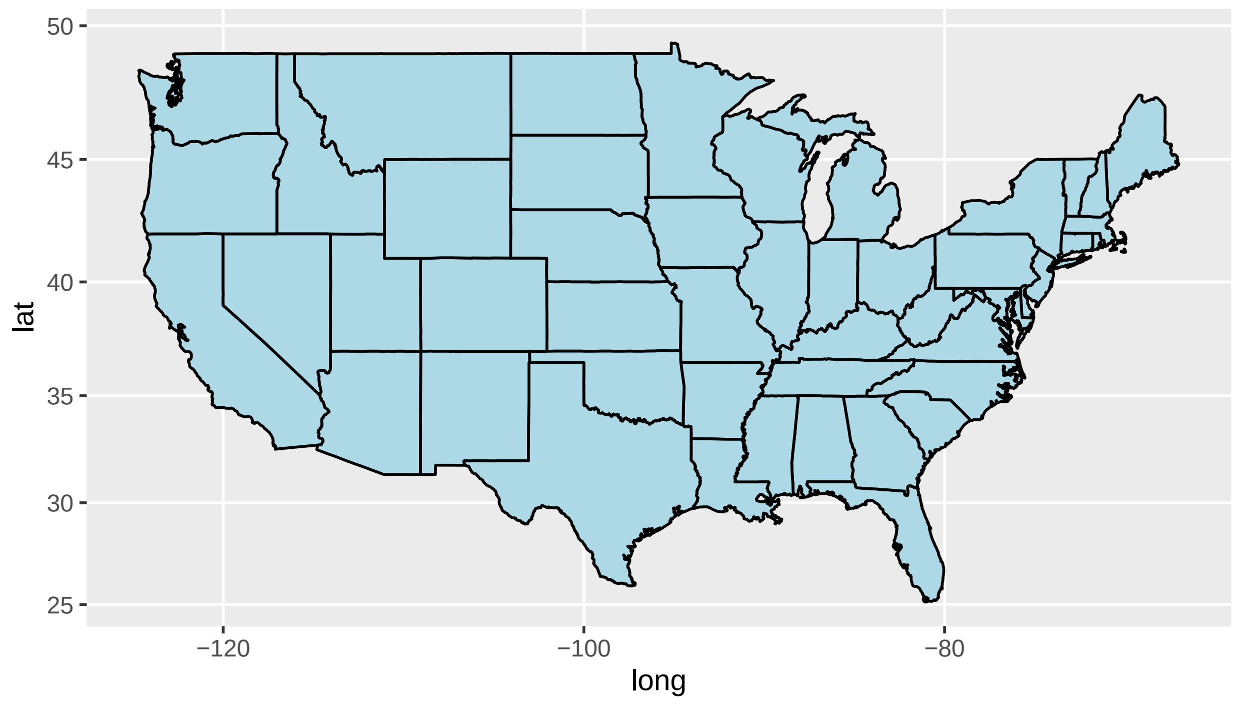


A full list of the available maps can be found in the `maps` documentation

3.2 Projections

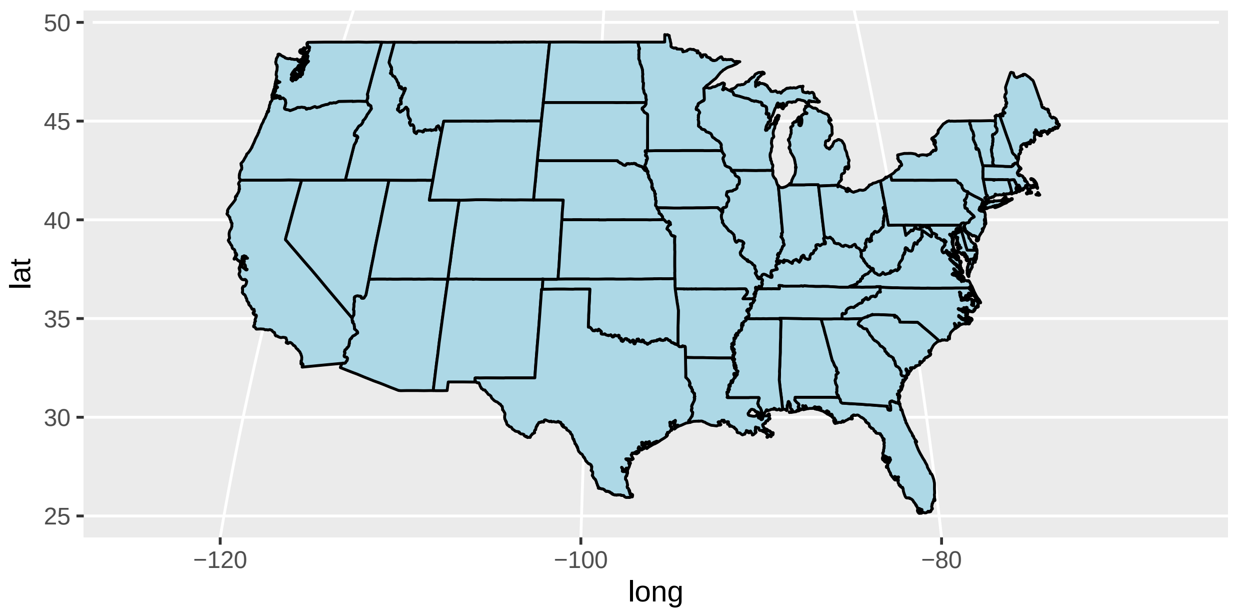
`ggplot2` uses the projections from the package `mapproj`.

The default projection is `"mercator"`

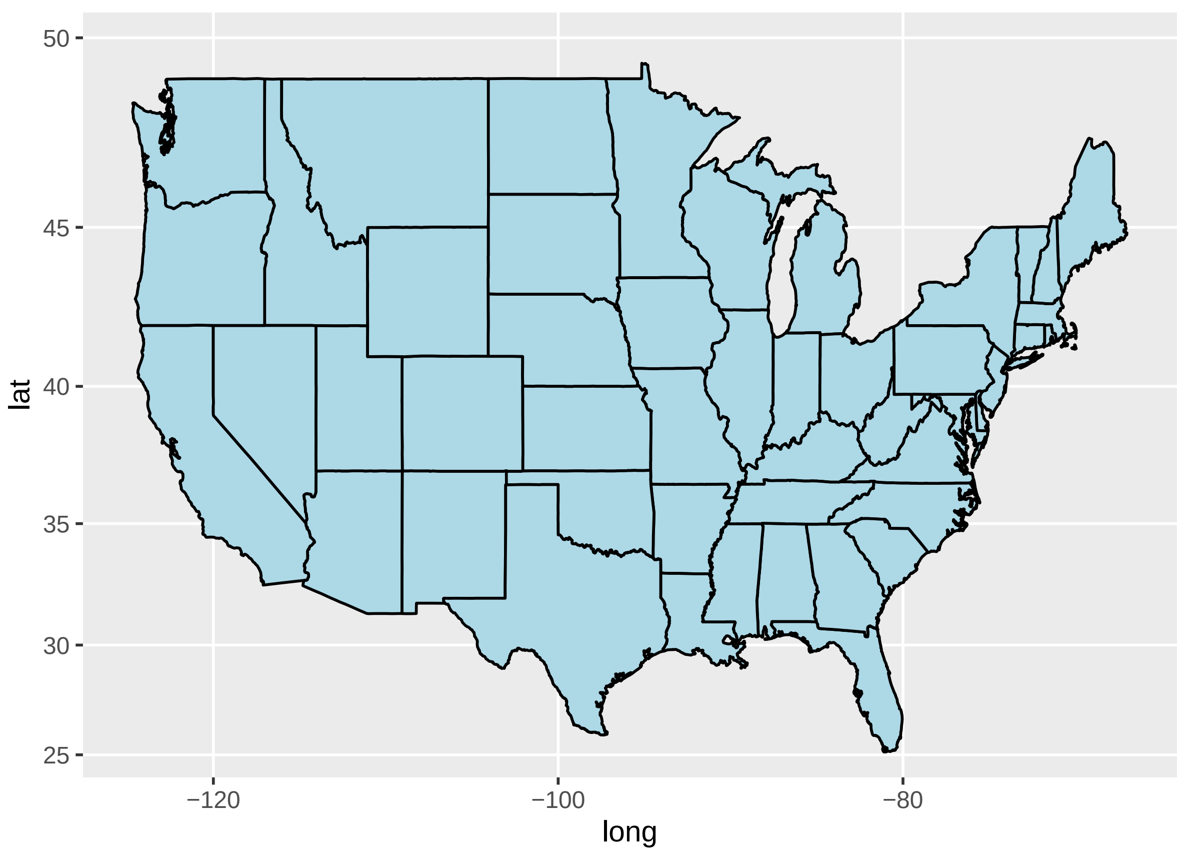


Projection is controlled by the `projection` argument to the `coord_map()` function.

```
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][mapproj]. [mapproj]: https://rdrr.io/cran/mapproj/man/mapproject.html "mapproj"

3.3 Choropleth maps

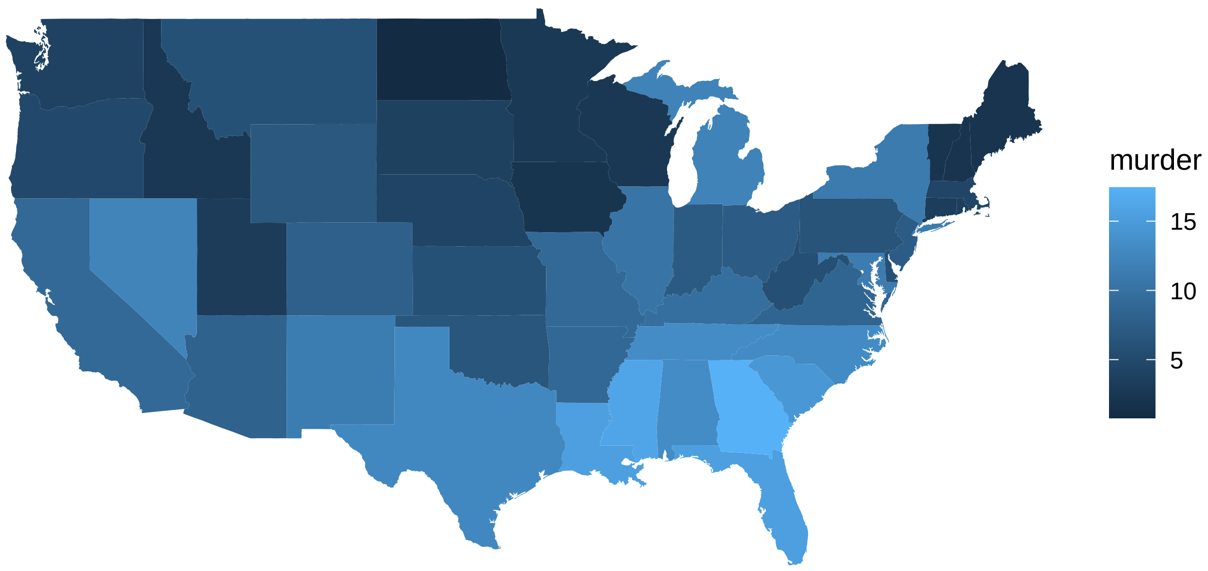
Another method for creating maps is `geom_map`.

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.

```r
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()
```



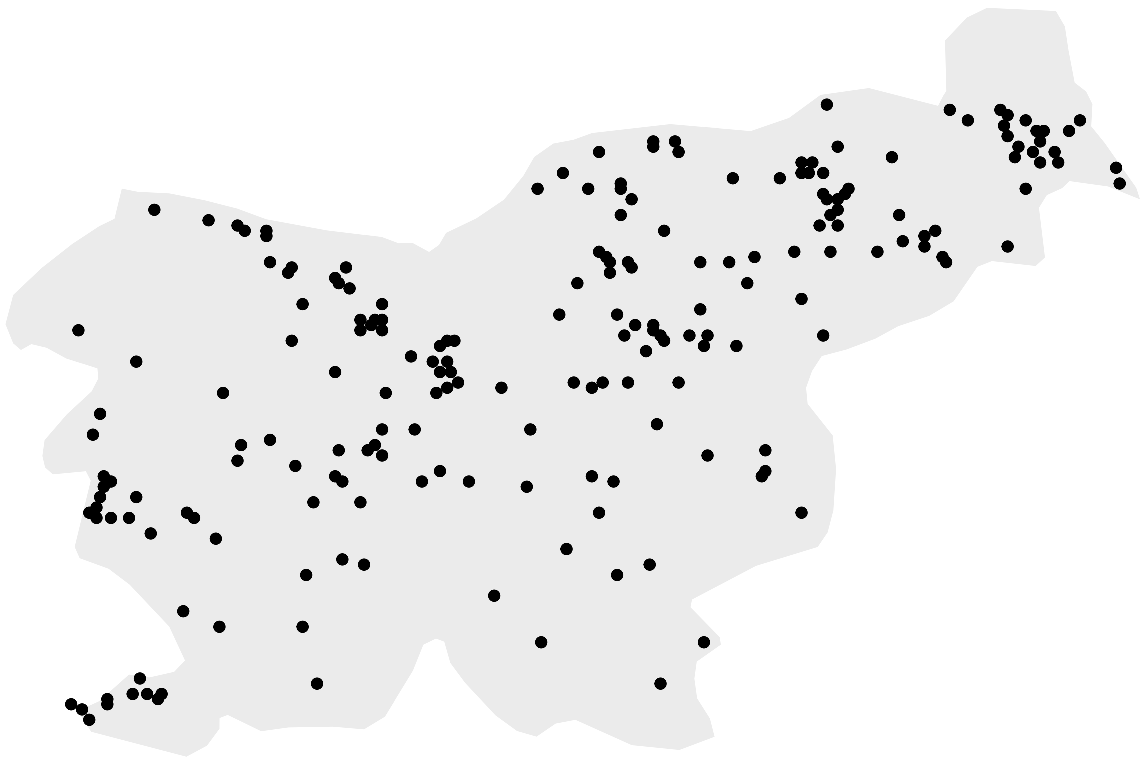
3.4 Bubble maps

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

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

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

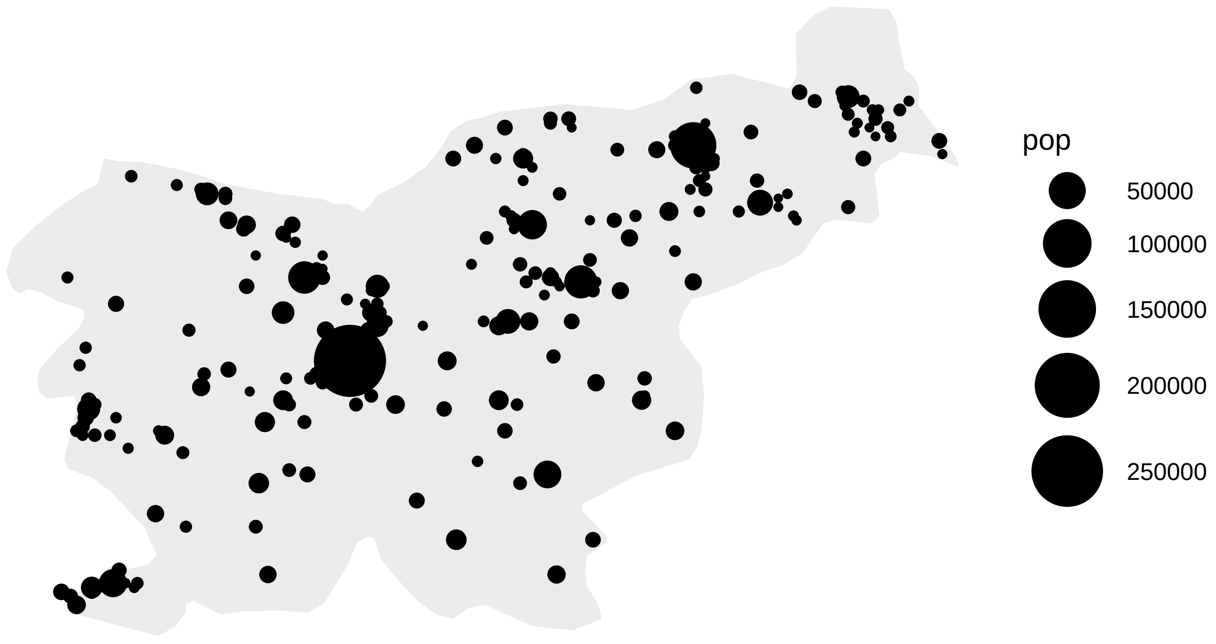
Use `geom_polygon` to draw the country, and `geom_point` for the 'bubbles' at the city coordinates.

```r
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.

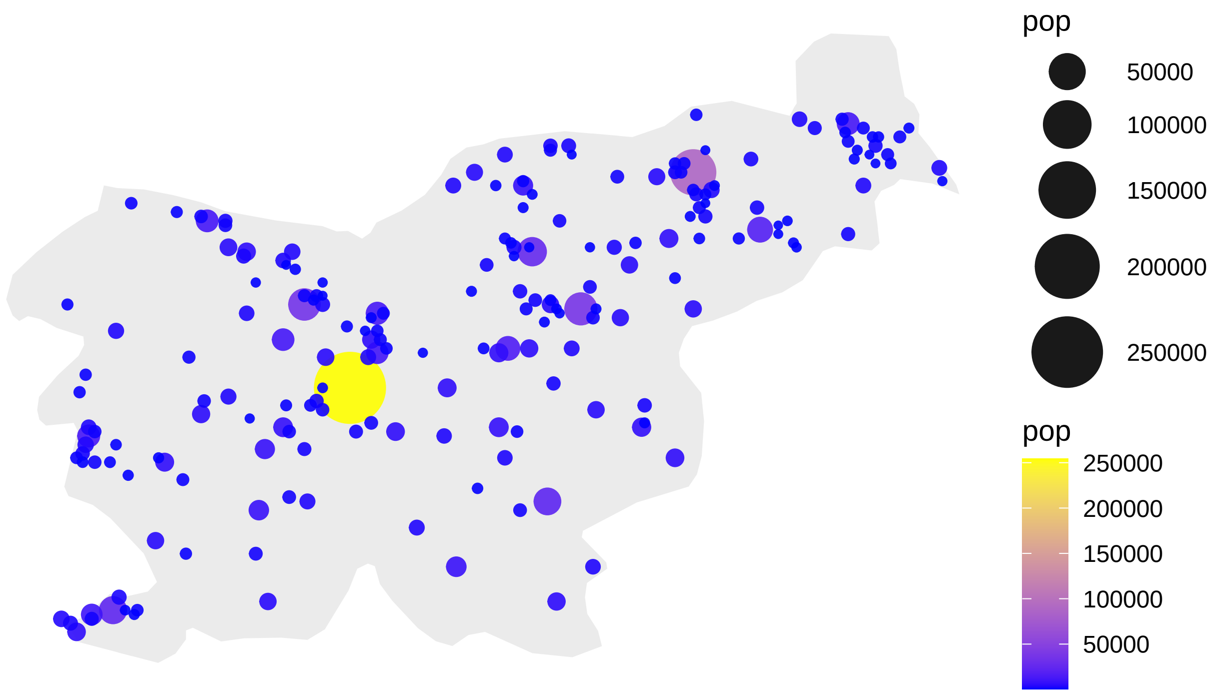
```
ggplot(data) +
   geom_polygon(data = sl, aes(x=long, y = lat, group = group), fill="grey", alpha=0.3) +
   geom_point( aes(x=long, y=lat, size=pop)) +
   scale_size_continuous(range=c(1,12)) +
   theme_void() + coord_map()
```



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.

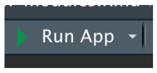
That's it! It won't *do* much, but running this code will run a shiny app. Notice a few things:

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

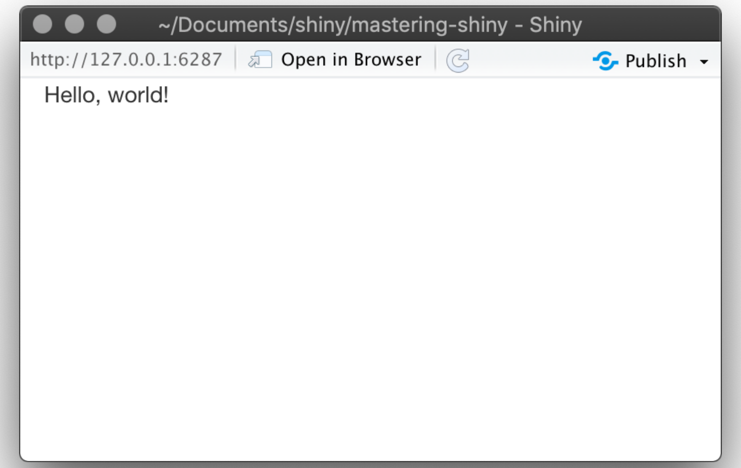
There are a few ways you can run this app:

- 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.



Hello, world!

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.

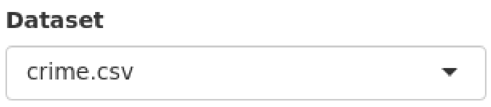


Figure 2: The datasets app with UI

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:

```r
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
  })
}
```

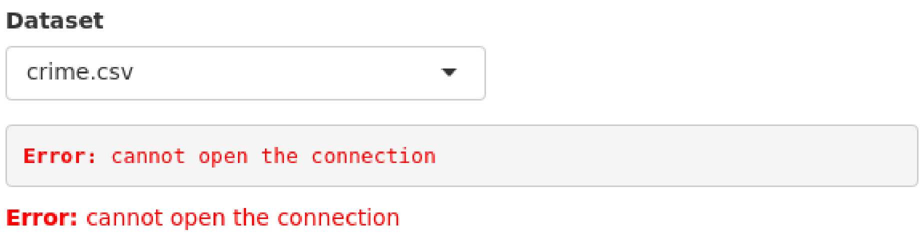
Almost every output you'll write in Shiny will follow this same pattern:

```r
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.



```
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.

```r
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**.

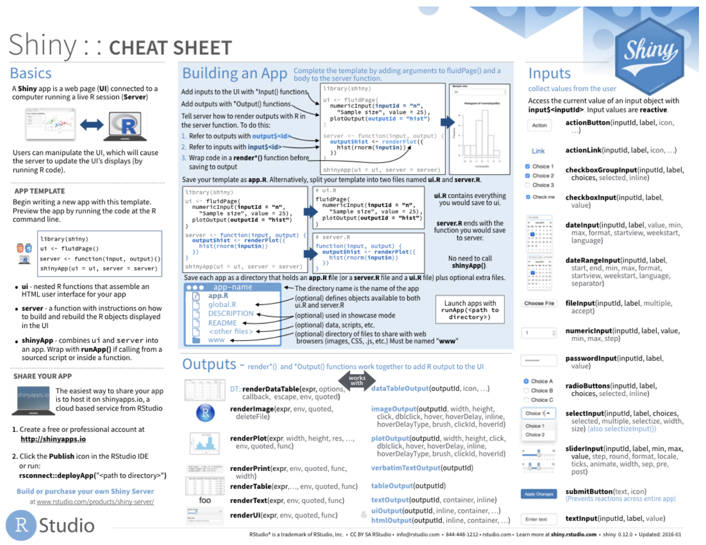
A simple way to create a reactive expression is by wrapping a block of code in `reactive({...})` and assigning it to a variable. You then use it by calling it like a function, with `()` at the end. While it looks like you're calling a function, a reactive expression only runs the first time it is called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

```r
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/.



Shiny : : CHEAT SHEET

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

```r
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE or run: **rsconnect::deployApp(**"<path to directory>"**)**

Build or purchase your own Shiny Server at www.rstudio.com/products/shiny-server/

Building an App Complete the template by adding arguments to fluidPage() and a body to the server function.

Add inputs to the UI with *Input() functions

Add outputs with *Output() functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with output$<id>
2. Refer to inputs with input$<id>
3. Wrap code in a **render*()** function before saving to output

```r
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui = ui, server = server)
```

Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```r
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```r
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

```r
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

app-name	The directory name is the name of the app
app.R	
global.R	(optional) defines objects available to both ui.R and server.R
DESCRIPTION	(optional) used in showcase mode
README	(optional) used in showcase mode
<other files>	(optional) data, scripts, etc.
www	(optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

Outputs - render*() and *Output() functions work together to add R output to the UI

	works with
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)	**dataTableOutput**(outputId, icon, ...)
renderImage(expr, env, quoted, deleteFile)	**imageOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
renderPlot(expr, width, height, res, ..., env, quoted, func)	**plotOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
renderPrint(expr, env, quoted, func, width)	**verbatimTextOutput**(outputId)
renderTable(expr,..., env, quoted, func)	**tableOutput**(outputId)
renderText(expr, env, quoted, func)	**textOutput**(outputId, container, inline)
renderUI(expr, env, quoted, func)	**uiOutput**(outputId, inline, container, ...) & **htmlOutput**(outputId, inline, container, ...)

Inputs

collect values from the user

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

Action	**actionButton**(inputId, label, icon, ...)
Link	**actionLink**(inputId, label, icon, ...)
Choice 1 Choice 2 Choice 3	**checkboxGroupInput**(inputId, label, choices, selected, inline)
Check me	**checkboxInput**(inputId, label, value)
	dateInput(inputId, label, value, min, max, format, startview, weekstart, language)
	dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)
Choose File	**fileInput**(inputId, label, multiple, accept)
1	**numericInput**(inputId, label, value, min, max, step)
••••••••	**passwordInput**(inputId, label, value)
Choice A Choice B Choice C	**radioButtons**(inputId, label, choices, selected, inline)
Choice 1 Choice 1 Choice 2	**selectInput**(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())
	sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)
Apply Changes	**submitButton**(text, icon) (Prevents reactions across entire app)
Enter text	**textInput**(inputId, label, value)

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **shiny.rstudio.com** • shiny 0.12.0 • Updated: 2016-01

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.

```r
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:

```r
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.

```r
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:

```r
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:

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

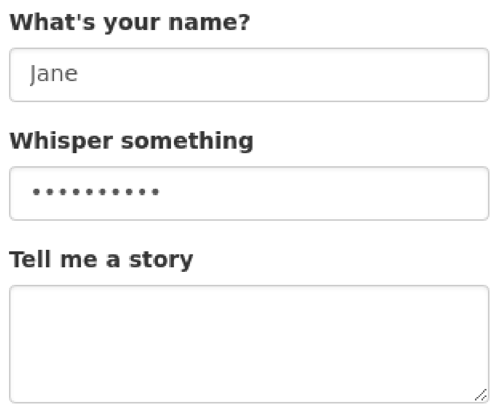
The inputs built into Shiny are described below. More complete information is available in the package documentation.

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)
)
```

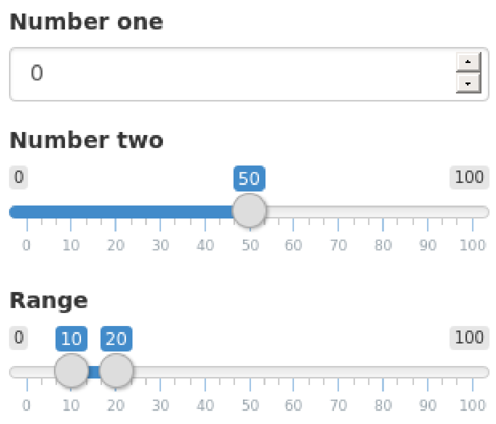


If you want to ensure that the text has certain properties you can use `validate()`.

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)
)
```

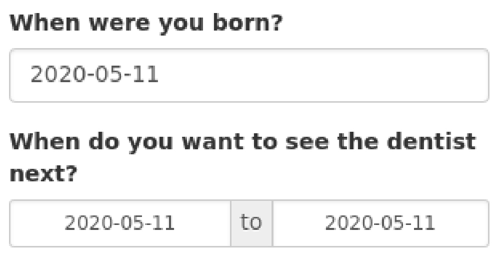


Use sliders for small ranges where the precise value is not so important, otherwise just use a numeric input. If you have had the experience before of setting a precise number on a small slider, it can be very frustrating!

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.

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

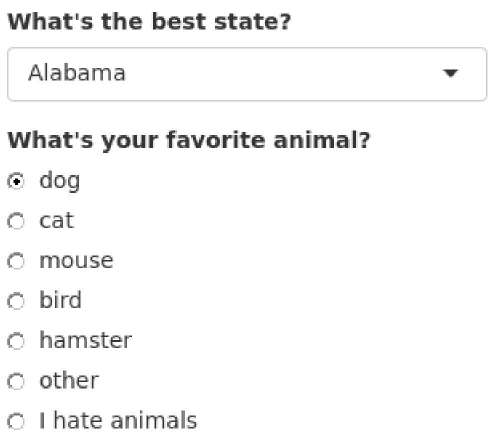


Date format, language, and the day on which the week starts defaults to US standards. If there are specific formats preferred in your organization, or you're creating an app with an international audience, try setting `format`, `language`, and `weekstart` appropriately, so that the dates are natural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

4.12.5 Limited set of choices

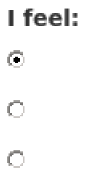
There are two different approaches to allow the user to choose from a prespecified set of options: `selectInput()` and `radioButtons()`.

```r
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)
)
```



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")
  )
)
```

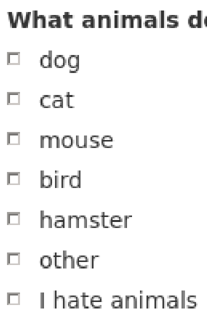


Dropdowns created with `selectInput()` are more suitable for longer take up the same amount of space, regardless of the number of options, making them more suitable for longer options. You can also set `multiple = TRUE` to allow the user to select multiple elements.

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

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

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



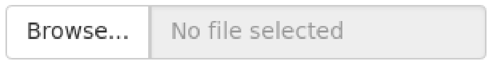
If you want a single checkbox for a single yes/no question, use `checkboxInput()`:

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

4.12.6 File uploads

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

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

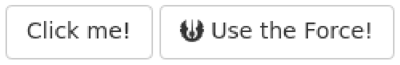


Note: `fileInput()` requires special handling on the server side.

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.

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



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?

```r
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({...})`.

```r
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.

```r
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.

```r
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.

```r
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()`:

```r
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/.

   ```
   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:

```
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:

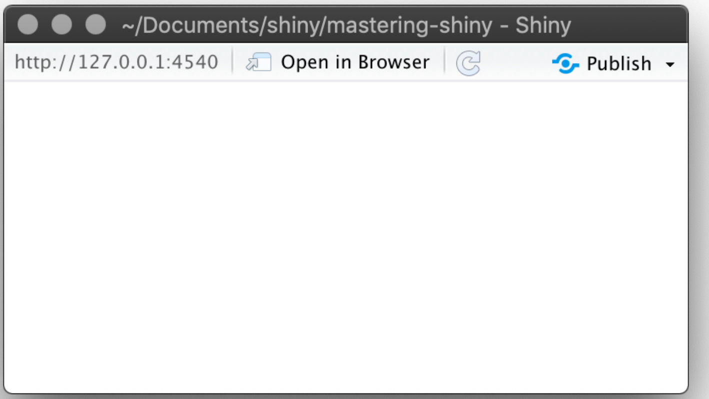
```
fluidPage(
  titlePanel(),
  sidebarLayout(
    sidebarPanel(
      sliderInput("obs")
    ),
```

```
    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.

```r
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.

```r
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()`.

```r
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.

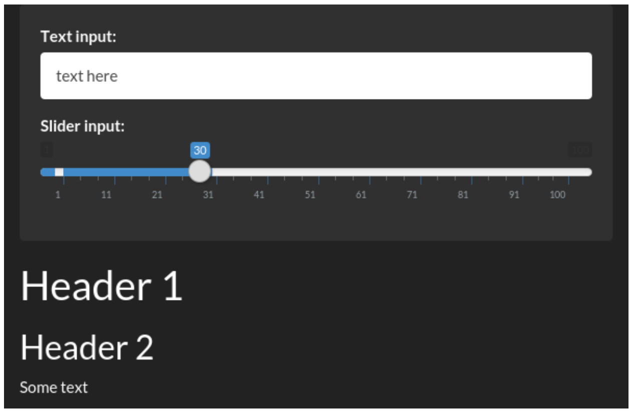
4.14.5 Themes

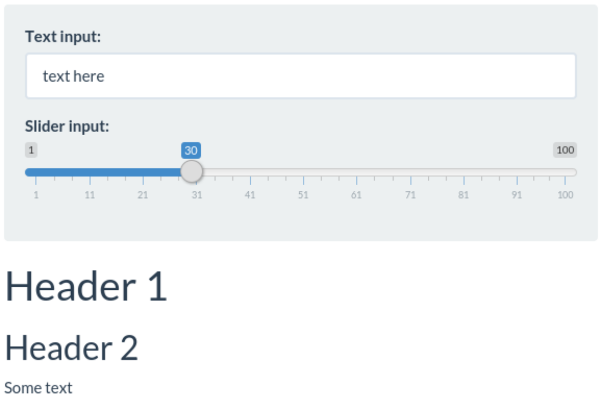
Creating a complete theme from scratch is a lot of work, but the easy wins with the `shinythemes` package. First make sure it is installed, it is not automatically installed with `shiny`:

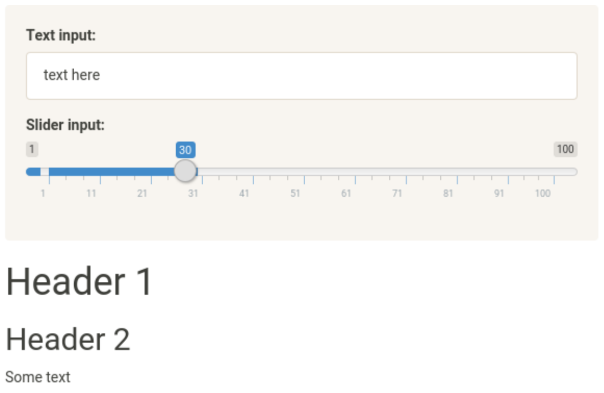
```
install.packages("shinythemes")
```

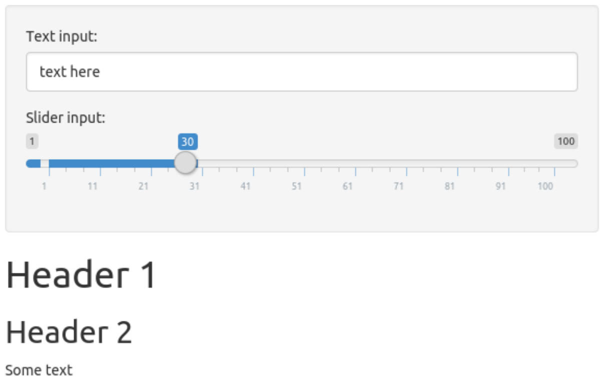
The following code shows four options:

```
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.