Building Web Apps with R Shiny

Lisa DeBruine

2021-07-16

Contents

Overview

Shiny apps let you make web applications that do anything you can code in R. For example, you can share your data analysis in a dynamic way with people who don't use R, collect and visualize data, or even make data aRt.

While there is a wealth of material available on the internet to help you get started with Shiny, it can be difficult to see how everything fits together. This class will take a predominantly live coding approach, rather than a lecture-only approach, so you can code along with the instructor and deal with the inevitable bugs and roadblocks together.

This class will teach you the basics of Shiny app programming, giving you skills that will form the basis of almost any app you want to build. By the end of the class, you will have created a custom app that collects and saves data, allows users to dynamically visualize the data, and produces downloadable reports.

0.1   Example Apps

The following are some diverse examples of Shiny apps that the instructor has made.

- Plot Demo Simulate data from a 2×2 factorial design and visualize it with 6 different plot styles.
- Simulating for LMEM companion to Understanding mixed effects models through data simulation (DeBruine & Barr, AMPPS 2021)
- Scienceverse is an ambitious (but in-progress) app for creating machine-readable descriptions of studies and human-readable summaries.
- Word Cloud Create a word cloud from text and customize its appearance. Created during the live-coding event at Hack Your Data Beautiful.

0.2   Code Horizons Course (27-30 July 2021)

Starting July 27, we are offering this seminar as a 4-day synchronous*, remote workshop for the first time. Each day will consist of a 3-hour live lecture held via the free video-conferencing software Zoom. You are encouraged to join the lecture live, but will have the opportunity to view the recorded session later if you are unable to attend at the scheduled time.

Each lecture session will conclude with a hands-on exercise reviewing the content covered, to be completed on your own. An additional lab session will be held Tuesday and Thursday afternoons, where you can review the exercise results with the instructor and ask any questions.

0.2.1   Schedule

0.2.1.1   Day 1

- Your First Shiny App
    - Overview of the UI/server structure
    - Inputs, outputs, and action buttons
    - Reactive functions

- ShinyDashboard
    - Basic template for shinydashboard projects
    - Sidebar, menu navigation, and tabs
    - Row- and column-based layouts

0.2.1.2   Day 2

- Different input types
- Different output types
- Reading and saving data
- Reactive functions

0.2.1.3   Day 3

- Customizing Your Apps
    - CSS, HTML, and Javascript
    - Structuring a complex app

- Intermediate Patterns
    - Debugging and error handling
    - Displaying elements contingent on the state of other elements

0.2.1.4 Day 4

- Sharing Your Apps

  - shinyapps.io
  - Self-hosting a shiny server
  - GitHub
  - In an R package

- Advanced Patterns

  - Creating and downloading a customized report
  - Shiny modules for repeated structures

We understand that scheduling is difficult during this unpredictable time. If you prefer, you may take all or part of the course asynchronously. The video recordings will be made available within 24 hours of each session and will be accessible for two weeks after the seminar, meaning that you will get all of the class content and discussions even if you cannot participate synchronously.

Closed captioning is available for all live and recorded sessions.

0.3 Computing

To participate in the hands-on exercises, you are strongly encouraged to use a computer with the most recent version of R installed. Participants are also encouraged to download and install RStudio, a front-end for R that makes it easier to work with. This software is free and available for Windows, Mac, and Linux platforms.

0.4 Who Should Register?

You need to have basic familiarity with R, including data import, data processing, visualization, and functions and control structures (e.g., if/else). Instruction will be done using RStudio. Some familiarity with ggplot2 and dplyr would be useful. You definitely do not need to be an expert coder, but the following code should not be challenging to understand.

```
library(ggplot2)

pets <- read.csv("pets.csv")

dv <- sample(c("score", "weight"), 1)
```

```r
if (dv == "score") {
  g <- ggplot(pets, aes(pet, score, fill = country))
} else if (dv == "weight") {
  g <- ggplot(pets, aes(pet, weight, fill = country))
}

g + geom_violin(alpha = 0.5)
```

If you want to brush up on your R (especially tidyverse), and also gain familiarity with the instructor's teaching style, the first seven chapters of Data Skills for Reproducible Science provide a good overview.

0.5   Further Resources

There are a lot of great resources online to reinforce or continue your learning about Shiny.

- RStudio Shiny Tutorials

Chapter 1

Your First Shiny App

1.1 The Demo App

To start, let's walk through the basics of setting up a shiny app, starting with the example built into RStudio I won't explain yet how shiny apps are structured; the goal is to just get something up and running, and give you some familiarity with the layout of a fairly simple app.

1.1.1 Set Up the Demo App
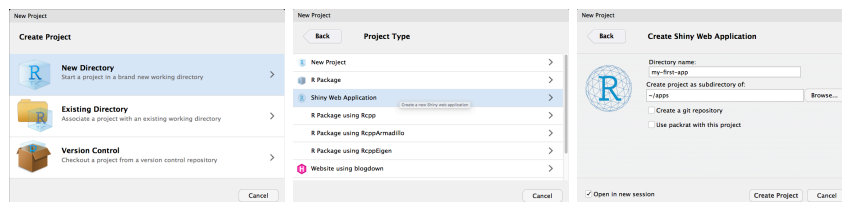
Figure 1.1: Creating a demo app.

1. Under the `File` menu, choose `New Project....` You will see a popup window like the one above. Choose `New Directory`.

2. Choose `Shiny Web Application` as the project type.

3. I like to put all of my apps in the same directory, but it doesn't matter where you save it.

4. Your RStudio interface should look like this now. You don't have to do anything else at this step.
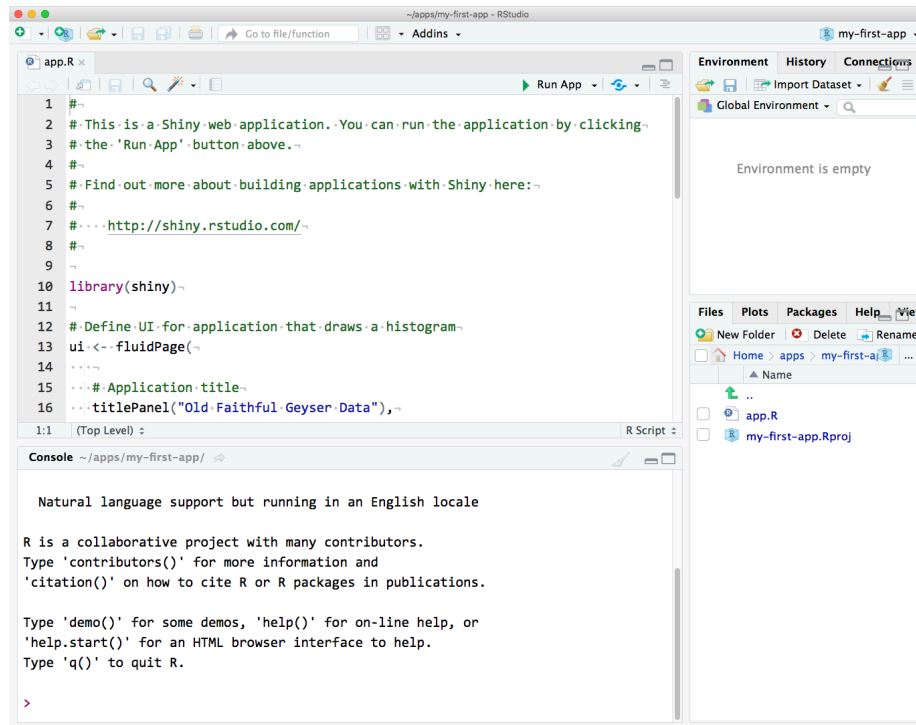
If RStudio has changed their demo app and your source code doesn't look like this, replace it with the code below:

View Code

```r
#
# This is a Shiny web application. You can run the application by clicking
# the 'Run App' button above.
#
# Find out more about building applications with Shiny here:
#
#    http://shiny.rstudio.com/
#

library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
```

```r
   sidebarLayout(
      sidebarPanel(
         sliderInput("bins",
                     "Number of bins:",
                     min = 1,
                     max = 50,
                     value = 30)
      ),

      # Show a plot of the generated distribution
      mainPanel(
         plotOutput("distPlot")
      )
   )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

   output$distPlot <- renderPlot({
      # generate bins based on input$bins from ui.R
      x    <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)

      # draw the histogram with the specified number of bins
      hist(x, breaks = bins, col = 'darkgray', border = 'white')
   })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Click on `Run App` in the top right corner of the source pane. The app will open up in a new window. Play with the slider and watch the histogram change.
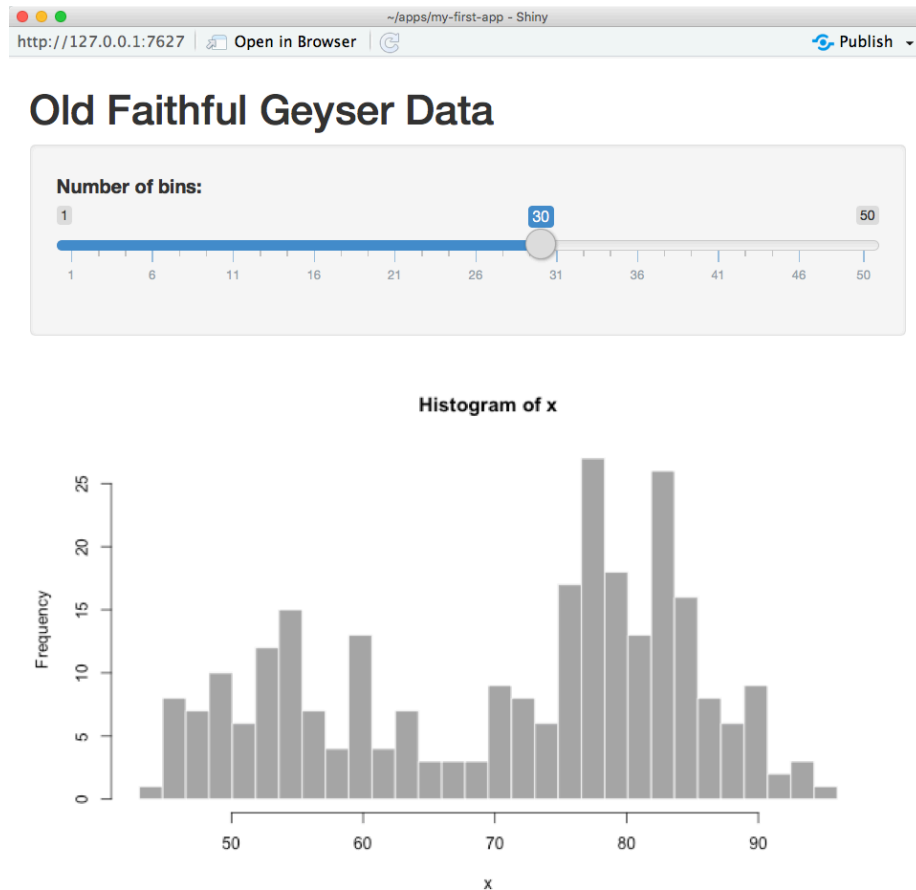
You can also open up the app in a web browser by clicking on `Open in Browser`.

1.1.2   Modify the Demo App

Now we're going to make a series of changes to the demo app until it's all your own.

You can close the app by closing the window or browser tab it's running in, or leave it running while you edit the code. If you have multiple screens, it's useful to have the app open on one screen and the code on another.

Find the application title. It is the first argument to the function `titlePanel`. Change the title to `"My First App"`. Make sure the title is inside quotes and the whole quoted string is inside the parentheses. Save the file (`cmd-S` or `File > Save`).
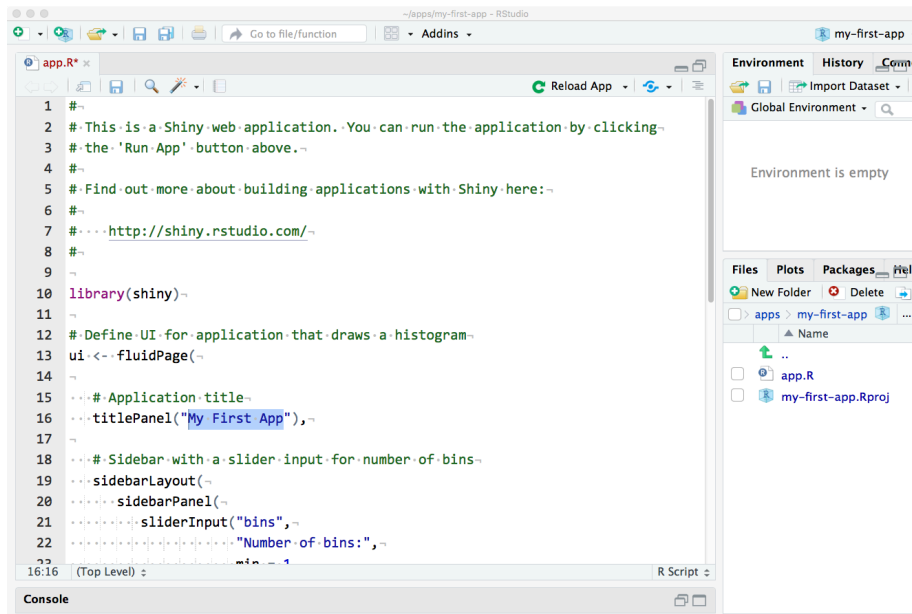
Click `Run App` (or `Reload App` if you haven't closed the app window) in the source pane. If you haven't saved your changes, it will prompt you to do so. Check that the app title has changed.

Now let's change the input. Find the function `sliderInput` (line 21). The first argument is the name you can use in the code to find the value of this input, so don't change it just yet. The second argument is the text that displays before the slider. Change this to something else and re-run the app.

```r
sliderInput("bins",
            "Number of bins:",
            min = 0,
            max = 50,
            value = 30)
```

See if you can figure out what the next three arguments to `sliderInput` do. Change them to different integers, then re-run the app to see what's changed.

The arguments to the function `sidebarPanel` are just a list of things you want to display in the sidebar. To add some explanatory text in a paragraph before the `sliderInput`, just use the paragraph function `p("My text")`

```r
sidebarPanel(
    p("I am explaining this perfectly"),
    sliderInput("bins",
                "Choose the best bin number:",
```

```r
                        min = 10,
                        max = 40,
                        value = 25)
    )
```
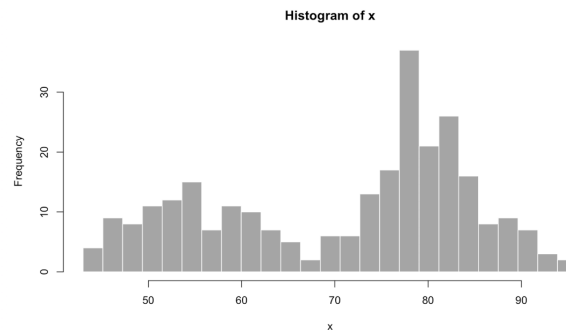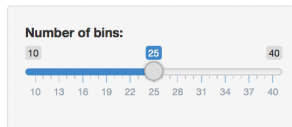

The sidebar shows up on the left if your window is wide enough, but moves to the top of the screen if it's too narrow.

I don't like it there, so we can move this text out of the sidebar and to the top of the page, just under the title. Try this and re-run the app.

```r
# Application title
titlePanel("My First App"),

p("I am explaining this perfectly"),

# Sidebar with a slider input for number of bins
sidebarLayout(...)
```

See where you can move the text in the layout of the page and where causes errors.

I'm also not keen on the grey plot. We can change the plot colour inside the `hist` function.

```r
# draw the histogram with the specified number of bins
hist(x, breaks = bins, col = 'steelblue3', border = 'grey30')
```

There are a lot of ways to represent colour in R. The easiest three are:

1. hexadecimal colours like `#0066CC`,

2. the `rgb` or `hsl` functions,
3. colour names (type `colours()` in the console)

I like `steelblue3`, as it's pretty close to the shiny interface default colour, but feel free to choose whatever you like.

I prefer `ggplot` graphs, so let's make the plot with `geom_histogram` instead of `hist` (which is a great function for really quick plots). Since we need several functions from the `ggplot2` package, we'll need to load that package at the top of the script, just under where the `shiny` package is loaded:

```
library(shiny)
library(ggplot2)
```

You can replace all of the code in the `renderPlot` function with the code below.

```
output$distPlot <- renderPlot({
  # create plot
  ggplot(faithful, aes(waiting)) +
    geom_histogram(bins = input$bins,
                   fill = "steelblue3",
                   colour = "grey30") +
    xlab("What are we even plotting here?") +
    theme_minimal()
})
```

You can set the `fill` and `colour` to whatever colours you like, and change `theme_minimal()` to one of the other built-in ggplot themes.

What *are* we even plotting here? Type `?faithful` into the console pane to see what the `waiting` column represents (`faithful` is a built-in demo dataset). Change the label on the x-axis to something more sensible.

1.1.3 Add New Things

The `faithful` dataset includes two columns:`eruptions` and `waiting`. We've been plotting the `waiting` variable, but what if you wanted to plot the `eruptions` variable instead?

Try plotting the eruption time (`eruptions`) instead of the waiting time. You just have to change one word in the `ggplot` function and update the x-axis label.

We can add another input widget to let the user switch between plotting eruption time and wait time. The RStudio Shiny tutorial has a great overview of the different input options. We need to toggle between two options, so we can

use either radio buttons or a select box. Radio buttons are probably best if you have only a few options and the user will want to see them all at the same time to decide.

Add the following code as the first argument to `sidebarPanel()`, which just takes a list of different widgets. `radioButtons` is the widget we're using. The first argument is `display_var`, which we will use later in the code to find the value of this widget. The second argument is the label to display to the user. The next argument is `choices`, which is a list of choices in the format `c("label1" = "value1", "label2 = "value2", ...)`. The label is what gets shown to the user and the value is what gets used by the code (these can be the same, but you often want the user label to be more descriptive). The last argument is `selected`, which is the value of the default choice. Save this and re-run the app.

```r
radioButtons("display_var",
             "Which variable to display",
             choices = c("Waiting time to next eruption" = "waiting",
                         "Eruption time" = "eruptions"),
             selected = "waiting"
),
```
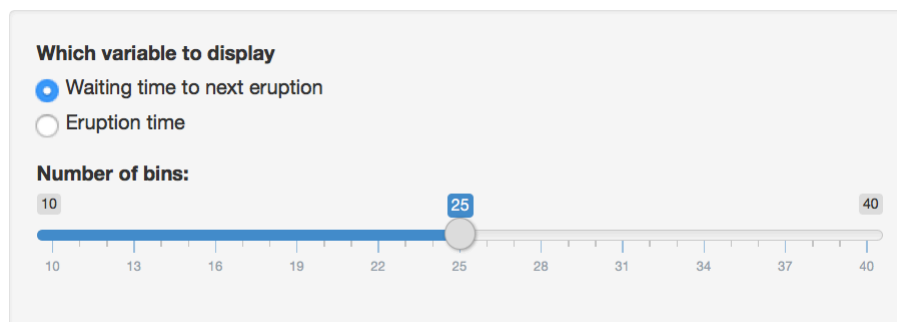

You should have a radio button interface now. You can click on the options to switch the button, but it won't do anything to your plot yet. We need to edit the plot-generating code to make that happen.

First, we need to change the x-axis label depending on what we're graphing. We use an if/else statement to set the variable `xlabel` to one thing if `input$display_var` is equivalent to `"eruptions"`, and to something else if it's equivalent to `"waiting"`. Put this code at the very beginning of the code block for the `renderPlot` function (after the line `output$distPlot <- renderPlot({`).

```r
# set x-axis label depending on the value of display_var
if (input$display_var == "eruptions") {
```

```
      xlabel <- "Eruption Time (in minutes)"
} else if (input$display_var == "waiting") {
      xlabel <- "Waiting Time to Next Eruption (in minutes)"
}
```

The double-equal-signs `==` means "equivalent to and is how you check if two things are the same; if you only use one equal sign, you set the variable on the left to the value on the right.

Then we have to edit the `ggplot` function to use the new label and to plot the correct column. The variable `input$display_var` gives you the user-input value of the widget called `display_var`.

```
# create plot
ggplot(faithful, aes(.data[[input$display_var]])) +
  geom_histogram(bins = input$bins,
                 fill = "steelblue3",
                 colour = "grey30") +
  xlab(xlabel) +
  theme_minimal()
```

Notice that the function `aes(waiting)` from before has changed to `aes(.data[[input$display_var]])`. Because `input$display_var` is a string, we have to select it from the `.data` placeholder (which refers to the `faithful` data table) using double brackets.

Re-run your app and see if you can change the data and x-axis label with your new widget. If not, check your code against the code.

\begin{figure}

My First App

I am explaining this perfectly

Which variable to display

◉ Waiting time to next eruption

○ Eruption time

Number of bins:

10   25   40

10   13   16   19   22   25   28   31   34   37   40

{
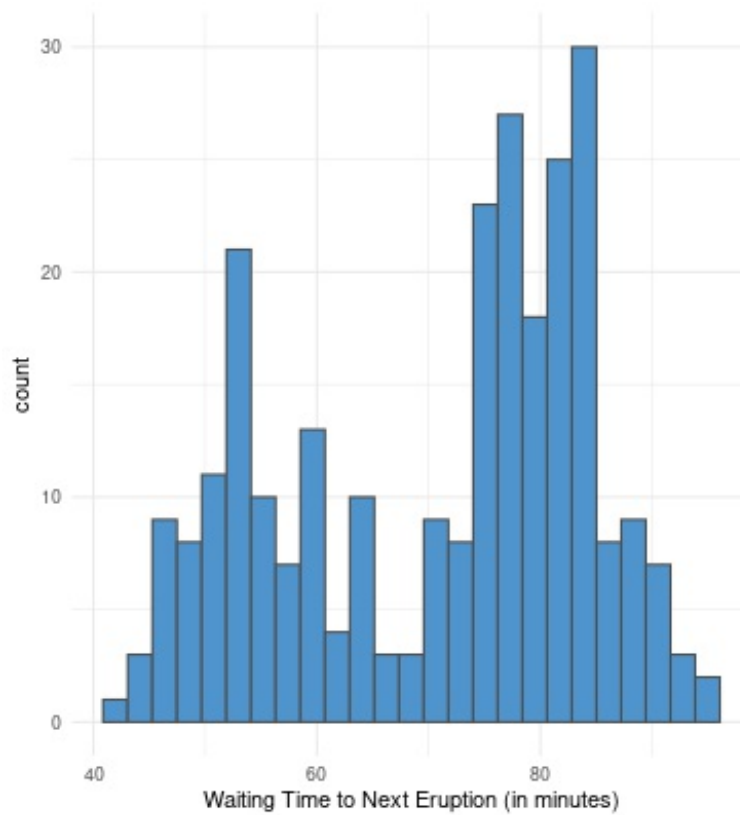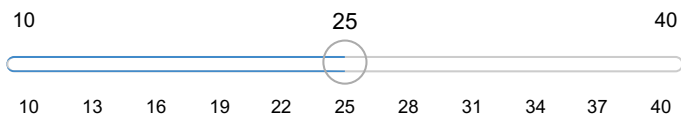
1.2   Overview of the UI/server structure

Now that we've made and modified our first working app, it's time to learn a bit about how a shiny app is structured.

A shiny app is made of two main parts, a UI, which defines what the user interface looks like, and a server function, which defines how the interface behaves. The function `shinyApp()` puts the two together to run the application in a web browser.

```r
# Load libraries ----
library(shiny)

# Define UI ----
ui <- fluidPage()

# Define server logic ----
server <- function(input, output) {
}

# Run the application ----
shinyApp(ui = ui, server = server)
```

Create a new app called "basic_demo" and replace all the text in app.R with the code above. You should be able to run the app and see just a blank page.

1.2.1   UI

The UI is created by one of the ui-building "*Page" functions from the shiny or shinydashboard package, such as `fluidPage()`, `fixedPage()`, `fillPage()` or `dashboardPage()`. The ui-building functions set up the parts of the webpage, which are created by more shiny functions that you list inside of the page function, separated by commas.

1.2.1.1   Tags

For example, the code below displays:

1. a title panel with the text "Basic Demo"
2. a level-two header with the text "My favourite things"
3. an unordered list (`tags$ul`) containing several list items (`tags$li`)
4. a paragraph with the text "This is a very basic demo."
5. an image of the shinyintro logo with a width and height of 100 pixels

```r
ui <- fluidPage(
   titlePanel("Basic Demo"),
   h2("My favourite things"),
   tags$ul(
      tags$li("Coding"),
      tags$li("Cycling"),
      tags$li("Cooking")
   ),
   p("This is a very basic demo."),
  tags$img(src = "https://debruine.github.io/shinyintro/images/shinyintro.png",
           width = "100px", height = "100px")
)
```

Many of the functions used to create parts of the website are the same as
HTML tags, which are ways to mark the beginning and end of different types
of text. Most HTML tags are available in shiny by using one of the `tags()`
sub-functions, but some of the more common tags, like `p()` or h1()–h6()`also
have a version where you can omit the`tags$` part. You can see a list of
all of the tags available in Shiny at the tag glossary

Add the code above to your basic_demo" app and replace my favourite things
with yours. Make the list an ordered list (instead of unordered) and change
the image size.

1.2.1.2 Page Layout

You usually want your apps to have a more complex layout than just each
element stacked below the previous one. The code below wraps the elements
after the title panel inside a `flowLayout()`.

```r
ui <- fluidPage(
  titlePanel("Basic Demo"),
  flowLayout(
    h2("My favourite things"),
    tags$ul(
      tags$li("Coding"),
      tags$li("Cycling"),
      tags$li("Cooking")
    ),
    p("This is a very basic demo."),
    tags$img(
      src = "https://debruine.github.io/shinyintro/images/shinyintro.png",
      width = "100px", height = "100px"
    )
```

```
  )
)
```

Replace the ui code in your basic_demo" app with the code above and run it in a web browser. What happens when you change the width of the web browser? Change `flowLayout` to `verticalLayout` or `splitLayout` and see what changes.

You can use a `sidebarLayout()` to arrange your elements into a `sidebarPanel()` and a `mainPanel()`. If the browser width is too narrow, the sidebar will display on top of the main panel.

```r
ui <- fluidPage(
   titlePanel("Basic Demo"),
   sidebarLayout(
      sidebarPanel(
         h2("My favourite things"),
         tags$ul(
            tags$li("Coding"),
            tags$li("Cycling"),
            tags$li("Cooking")
         )
      ),
      mainPanel(
         p("This is a very basic demo."),
         tags$img(
            src = "https://debruine.github.io/shinyintro/images/shinyintro.png",
            width = "100px", height = "100px"
         )
      )
   )
)
```

1.3 Inputs, outputs, and action buttons

So far, we've just put static elements into our UI. What makes Shiny apps work is dynamic elements like inputs, outputs, and action buttons.

1.3.1 Inputs

Inputs are ways for the users of your app to communicate with the app. They are things like drop-down menus or checkboxes. We'll go into the different

types of inputs in the Inputs chapter. Below we'll turn the list of favourite things into a checkbox list.

```r
checkboxGroupInput(inputId = "fav_things",
                   label = "What are your favourite things?",
                   choices = c("Coding", "Cycling", "Cooking")
)
```

Most inputs are structured like this, with an inputId, which needs to be a unique string not used as the ID for any other input or output in your app, a label that contains the question, and a list of choices or other parameters that determine what type of values the input will record.

1.3.2   Outputs

Outputs are placeholders for things that `server()` will create. There are different output functions for different types of outputs, like text, plots, and tables. We'll go into the different types of outputs in detail in the Outputs chapter. Below, we'll make a placeholder for some text that we'll display after counting the number of favourite things.

```r
textOutput(outputId = "n_fav_things")
```

Most outputs are structured like this, with just a unique outputId.

1.3.3   Action buttons

Action buttons are a special type of input that register button clicks. Below we'll make an action button that users can click once they've selected all of their favourite things.

```r
actionButton(inputId = "count_fav_things",
             label = "Count",
             icon = icon("calculator")
)
```

Action buttons require a unique inputId and a label for the button text. You can also add an icon. Choose a free icon from fontawesome.

Put the input, output, and action button into the ui and run it. You can see that the input checkboxes are selectable and the button is clickable, but nothing is displayed in the text output. We need some code in `server()` to handle that.

```
ui <- fluidPage(
   titlePanel("Basic Demo"),
   sidebarLayout(
      sidebarPanel(
         checkboxGroupInput(inputId = "fav_things",
                  label = "What are your favourite things?",
                  choices = c("Coding", "Cycling", "Cooking")
         ),
         actionButton(inputId = "count_fav_things",
             label = "Count",
             icon = icon("calculator")
         )
      ),
      mainPanel(
         textOutput(outputId = "n_fav_things")
      )
   )
)
```

1.4 Reactive functions

Reactive functions are functions that run when their inputs change. Inside the server function, the object `input` is a named list of the values of all of the inputs. For example, if you want to know which items in the select input named "fav_things" were selected, you would use `input$fav_things`.

Here, we just want to count how many items are checked. We want to do this whenever the button "count_fav_things" is clicked, so we can use the reactive function `observeEvent()` to do this. Every time the value of `input$count_fav_things` changes (which happens when it is clicked), it will run the code inside of the curly brackets `{}`. The code will only run when `input$count_fav_things` changes, not when any inputs inside the function change.

```
server <- function(input, output) {
   # count favourite things
   observeEvent(input$count_fav_things, {
      n <- length(input$fav_things)
      count_text <- sprintf("You have %d favourite things", n)
   })
}
```

Now we want to display this text in the output "n_fav_things". We need to use a render function that is paired with our output function. Since

"n_fav_things" was made with `textOutput()`, we fill it with `renderText()`.

```r
server <- function(input, output) {
   # count favourite things
   observeEvent(input$count_fav_things, {
      n <- length(input$fav_things)
      count_text <- sprintf("You have %d favourite things", n)
      output$n_fav_things <- renderText(count_text)
   })
}
```

As always in coding, there are many ways to accomplish the same thing. These methods have different pros and cons that we'll learn more about in the Reactive chapter. Here is another pattern that does that same as above.

This pattern uses `reactive()` to update the value of a new function called `count_text()` whenever any inputs inside the reactive function change. We use `isolate()` to prevent `count_text()` from changing when users click the checkboxes.

Whenever the returned value of `count_text()` changes, this triggers an update of the "n_fav_things" output.

```r
server <- function(input, output) {
   # update count_text on fav_things
   count_text <- reactive({
      input$count_fav_things # just here to trigger the reactive
      fav_things <- isolate(input$fav_things) # don't trigger on checks
      n <- length(fav_things)
      sprintf("You have %d favourite things", n)
   })

   # display count_text when it updates
   output$n_fav_things <- renderText(count_text())
}
```

Compare the app behaviour with the first pattern versus the second. How are they different? What happens if you remove `isolate()` from around `input$fav_things`?

1.5   Further Resources

- Application layout guide
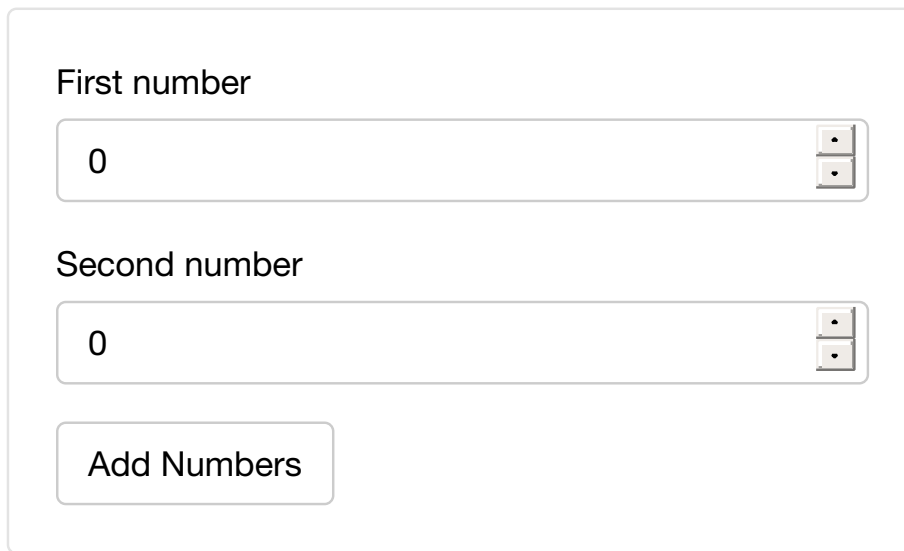
1.6   Exercises

1.6.1   Addition App - UI

Create the UI for following addition app. Use `numericInput()` to create the inputs.

\begin{figure}

Addition Demo

First number

0

Second number

0

Add Numbers

{$0 + 0 = 0$

Solution

```
ui <- fluidPage(
  titlePanel("Addition Demo"),
  sidebarLayout(
    sidebarPanel(
      numericInput("n1", "First number", 0),
      numericInput("n2", "Second number", 0),
      actionButton("add", "Add Numbers")
    ),
    mainPanel(
      textOutput(outputId = "n1_plus_n2")
```

```
    )
  )
)
```

1.6.2   observeEvent

Use `observeEvent()` to write a server function that displays "n1 + n2 = sum" when you click the action button.

Solution

```
server <- function(input, output) {
  # add numbers
  observeEvent(input$add, {
    sum <- input$n1 + input$n2
    add_text <- sprintf("%d + %d = %d", input$n1, input$n2, sum)
    output$n1_plus_n2 <- renderText(add_text)
  })
}
```

1.6.3   reactive

Use `reactive()` to accomplish the same behaviour.

Solution

```
server <- function(input, output) {
  add_text <- reactive({
    input$add # triggers reactive
    n1 <- isolate(input$n1)
    n2 <- isolate(input$n2)
    sprintf("%d + %d = %d", n1, n2, n1 + n2)
  })

  output$n1_plus_n2 <- renderText(add_text())
}
```

Chapter 2

ShinyDashboard

Shinydashboard is an R package that provides functions to upgrade your
Shiny apps into beautiful dashboards.

2.1   Basic template for shinydashboard projects

The shinyintro package provides a basic template for a shinydashboard project.

\begin{figure}

{

You can start a new app using the template with the code
`shinyintro::clone("basic_template", "myapp")`. This will create a new
directory called "myapp" in your working directory and open the `app.R` file in
RStudio.

This directory contains:

- `app.R`: the file where you define the ui and server
- `DESCRIPTION`: A file that contains some structured info about the app
- `README.md`: A file that can contain any information you want
- `scripts`: a directory that can contain external R code that you can source
  into the `app.R` file
- `www`: a directory that contains helper files like images, CSS, and JavaScript

2.2 Dashboard Structure

Notice that the ui is created with `dashboardPage()` now. This needs to be set
up a little differently than `fluidPage()`. The main parts of a dashboardPage
are the header, sidebar, and body.

You can also change the default skin colour. Possible skin colours are: red, yellow (looks orange to me), green, blue, purple, and black.

```
dashboardPage(
    skin = "purple",
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody()
)
```

2.2.1 dashboardHeader

You can add a title to the header, and change the title width.

```
dashboardHeader(
  title = "Basic Template",
  titleWidth = "calc(100% - 44px)" # puts sidebar toggle on right
)
```

You can also include message, notification, and task menus in the header. We won't be using those in this class, but you can learn more at the shinydashboard site.

If you don't want to use a header, include `dashboardHeader(disable = TRUE)` inside `dashboardPage()`.

2.2.2 dashboardSidebar

The sidebar usually contains a `sidebarMenu()`, which needs a unique ID and a list of menu items. Each `menuItem()` consists of the title, a tabName that will be used to refer to the tab later, and an icon. You can find a list of the available free icons at fontawesome.

You can also add in any other elements to the sidebar. The code below adds a link using `tags$a()` Text inside of the sidebar usually looks too close to the edges, so we use css to style the link and add padding.

```
dashboardSidebar(
    sidebarMenu(
        id = "tabs",
        menuItem("Tab Title", tabName = "x_tab", icon = icon("dragon"))
    ),
    tags$a(href = "https://debruine.github.io/shinyintro/",
           "ShinyIntro book", style="padding: 1em;")
)
```

If you don't want to use a sidebar, include `dashboardSidebar(disable = TRUE)` inside `dashboardPage()`.

2.2.3   dashboardBody

The main part of the app goes inside `dashboardBody()` If you're going to use shinyjs functions (which the basic template does), you need to put `useShinyjs()` first. Then you include the header, linking to any custom CSS or JavaScript files.

The contents of the body go after that. The most common pattern is a multi-page tabbed pattern, which is set up with `tabItems()`, which contains a `tabItem()` for each tab. The `tabName` has to match the name you used in the sidebar `menuItem`, so that tab shows when the user clicks on the corresponding menu item.

```r
dashboardBody(
    shinyjs::useShinyjs(),
    tags$head(
        # links to files in www/
        tags$link(rel = "stylesheet", type = "text/css", href = "custom.css"),
        tags$script(src = "custom.js")
    ),
    tabItems(
        tabItem(tabName = "x_tab", imageOutput("logo"))
    )
)
```

Since each tab is usually a quite complex list of elements, I like to define each tab outside `dashboardPage()` and then just put a list of the tab objects inside `tabItems()`. This way, it's easy to move the whole tab definition to an external file if it gets too complex (see Chapter 8).

2.3   Body Structure

Tab items can be structured in several ways. At the simplest, you can just list each element after the tabName.

```r
tabItem(
    tabName = "demo_tab",
    textInput("given", "Given Name"),
    textInput("surname", "Surname"),
    selectInput("pet", "What is your favourite pet?", c("cats", "dogs", "ferrets")),
```

```
    textAreaInput("bio", NULL, height = "100px", placeholder = "brief bio")
)
```

2.3.1 Boxes

Most shinydashboard apps organise the parts inside boxes.

```
tabItem(
    tabName = "demo_tab",
    box(
      textInput("given", "Given Name"),
      textInput("surname", "Surname"),
      selectInput("pet", "What is your favourite pet?", c("cats", "dogs", "ferrets"))
    ),
    box(
      textAreaInput("bio", NULL, height = "100px", placeholder = "brief bio")
    )
)
```


You can add titles to the boxes, make them collapsible and/or give them solid headers.

```
tabItem(
    tabName = "demo_tab",
    box(title = "Personal Info",
      collapsible = TRUE,
      textInput("given", "Given Name"),
      textInput("surname", "Surname"),
      selectInput("pet", "What is your favourite pet?", c("cats", "dogs", "ferrets"))
    ),
    box(title = "Biography",
```

```
        solidHeader = TRUE,
        textAreaInput("bio", NULL, height = "100px", placeholder = "brief bio")
    )
)
```

In the normal shinydashboard style, solid headers only have a colour if the box also has the status argument set. In the basic template provided in this class, there is custom CSS to make solid headers the same colour as the theme skin, but you can also set the status.

```
tabItem(
    tabName = "x_tab",
    box(title = "No Status", solidHeader = TRUE),
    box(title = "Primary",   solidHeader = TRUE, status = "primary"),
    box(title = "Success",   solidHeader = TRUE, status = "success"),
    box(title = "Info",      solidHeader = TRUE, status = "info"),
    box(title = "Warning",   solidHeader = TRUE, status = "warning"),
    box(title = "Danger",    solidHeader = TRUE, status = "danger")
)
```


Basic Template
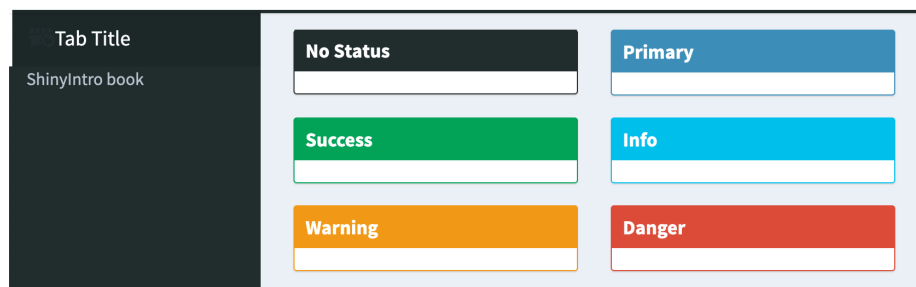
2.3.2   Info and Value Boxes

You can use an `infoBox()` or a `valueBox()` to highlight a small amount of information. The shinydashboard uses an aqua background as the default, but the basic template changes this to the skin colour. However, you can customise this by setting the color argument.

```
tabItem(
    tabName = "x_tab",
    infoBox("Default InfoBox", "Value", "Subtitle"),
    valueBox("Default ValueBox", "With subtitle"),
    valueBox("No subtitle", "")
)
```
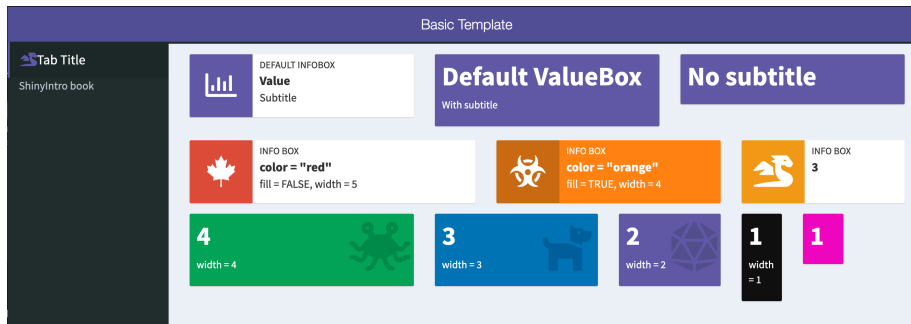
The default width of boxes is 6, and info and value boxes are 4.
Shinydashboard uses a grid system that is 12 units across

Try to write the code to create the second row of info boxes show above and
the third row of value boxes.

2.3.3   Tab Boxes

Create a box with multiple tabs using `tabBox()`, which contains `tabPanel()`.

```
tabItem(
    tabName = "demo_tab",
    tabBox(
        title = "Test Yourself 1",
        tabPanel("Question", "What function creates tabBox contents?"),
        tabPanel("Answer", "tabPanel()")
    ),
    tabBox(
        title = "Test Yourself 2",
        side = "right",
        selected = "Question",
        tabPanel("Answer", "selected"),
        tabPanel("Question", "What attribute changes the default tab?")
    )
)
```
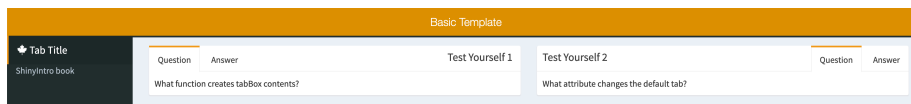

2.3.4   Row Layout

You can arrange the boxes inside a `fluidRow()`. Set the box height in pixels.
If the window gets too narrow, the boxes will move to stack instead of be in

rows.

```
tabItem(
    tabName = "demo_tab",
    fluidRow(
      box("A", title = "2x100", width = 2, height = 100),
      box("B", title = "1x100", width = 1, height = 100),
      box("C", title = "2x200", width = 2, height = 200),
      box("D", title = "3x300", width = 3, height = 300),
      box("E", title = "4x100", width = 4, height = 100),
      box("F", title = "5x100", width = 5, height = 100),
      box("G", title = "7x100", width = 7, height = 100)
    )
```
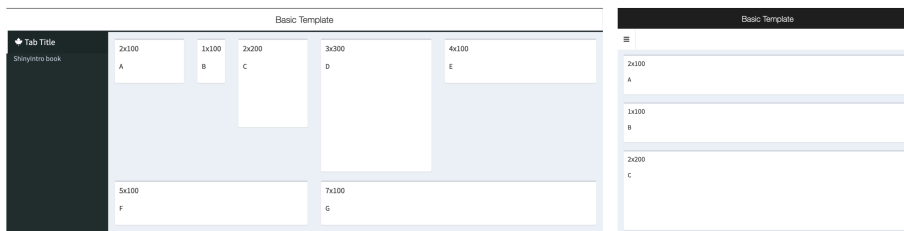

2.3.5   Column Layout

Alternatively, you can arrange boxes or other elements inside a column() with a specific width. Elements inside this column have a width relative to the column width, so no matter what value you set the column width to, an element inside with a width of 6 will be half the column width.

```
tabItem(
    tabName = "demo_tab",
    column(width = 6,
        box("A", title = "12x100", width = 12, height = 100),
        box("B", title = "6x100", width = 6, height = 100),
        box("C", title = "6x200", width = 6, height = 200)
    ),
    column(width = 4,
        box("D", title = "12x300", width = 12, height = 300),
        box("E", title = "12x100", width = 12, height = 100)
    ),
    column(width = 2,
        box("F", title = "12x100", width = 12, height = 100),
        box("G", title = "12x100", width = 12, height = 100)
```

```
    )
)
```
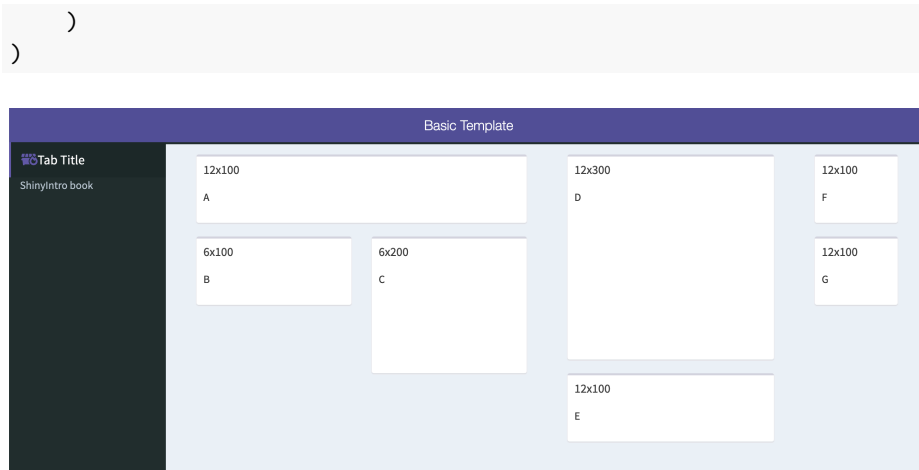

2.4   Further Resources

- ShinyDashboard

2.5   Exercises

2.5.1   Create an app from the basic template

- Create a demo app from the basic_template and run it.
- Close the app and run it again.
- Look at the code to see how the theme colour and sidebar icon change.
- Change the title and author in the `DESCRIPTION` file and set DisplayMode to "Showcase" instead of "Normal" to see what happens when you run it.

2.5.2   Customise the header

- Change the title to the title of the app you want to build.
- Choose a skin
- Experiment with different values for the titleWidth argument. Try "50%" or "200px" and see how the title behaves when you change the width of the browser window.

2.5.3   Customise the sidebar

- Add tabs to the sidebar menu for the pages of your custom app.

- Choose appropriate icons for each tab.
- Remove the link to the ShinyIntro book and add some information about your app, like the author or a use license.

2.5.4   Customise the body

- Create an empty tabItem for each tab you put in the sidebar menu and add them to the body.
- Start adding elements to each tab, such as a list of inputs with questions or a `plotOutput()` to display a feedback plot.
- Experiment with different ways to display the elements by grouping them into boxes
- Experiment with different layouts for your boxes and elements.
- Make sure you check how the app looks on different sized screens, such as phone screens.

Chapter 3

Inputs

3.1 Input functions

Inputs are ways that users can communicate information to the Shiny app.
Explore some different input types in the embedded app below before you read
about how to set up each type. You can run this app locally with
`shinyintro::app("input_demo")` or view it in a separate tab with the
showcase interface.

\begin{figure}

We're not recording anything here.

Name

Your Name

Biography

Tell us something interesting about you.

Do you like Shiny?

▼

Gender (select all that apply)

☐ Non-binary

☐ Male

☐ Female

☐ Agender

☐ Gender Fluid

Age

0   100

0   10   20   30   40   50   60   70   80   90   100

☑ I love R

Reset
{

3.1.1   textInput

`textInput()` creates a one-line box for short text input.

```r
demo_text <-
  textInput("demo_text",
            label = "Name",
            value = "",
            width = "100%",
            placeholder = "Your Name")
```

3.1.2   textAreaInput

`textAreaInput()` creates a multi-line box for longer text input.

```r
demo_textarea <-
  textAreaInput("demo_textarea",
                label = "Biography",
                value = "",
                width = "100%",
                rows = 5,
                placeholder = "Tell us something interesting about you.")
```

3.1.3   selectInput

`selectInput()` creates a drop-down menu. Set the first choice to `""` to default to `NA`. If your choices are a named list or vector, the names are what is shown and the values are what is recorded. If the choices aren't named, the displayed and recorded values are the same.

```r
demo_select <-
  selectInput("demo_select",
              label = "Do you like Shiny?",
              choices = list("",
                             "Yes, I do" = "y",
                             "No, I don't" = "n"),
              selected = NULL,
              width = "100%")
```

You can also make a select where users can choose multiple options.

```r
genders <- list( # no blank needed
  "Non-binary" = "nb",
  "Male" = "m",
  "Female" = "f",
  "Agender" = "a",
  "Gender Fluid" = "gf"
)

demo_select_multi <-
  selectInput("demo_select2",
              label = "Gender (select all that apply)",
              choices = genders,
              selected = NULL,
              multiple = TRUE,
              selectize = FALSE,
              size = 5)
```

3.1.4   checkboxGroupInput

However, this interface almost always looks better with a checkbox group.

```r
demo_cbgi <-
  checkboxGroupInput("demo_cbgi",
                     label = "Gender (select all that apply)",
                     choices = genders)
```

3.1.5   checkboxInput

You can also make a single checkbox. The value is `TRUE` when checked and `FALSE` when not.

```r
demo_cb <- checkboxInput("demo_cb",
                         label = "I love R",
                         value = TRUE)
```

Sliders allow you to choose numbers between a minimum and maximum.

```r
demo_slider <- sliderInput("demo_slider",
                           label = "Age",
                           min = 0,
                           max = 100,
                           value = 0,
```

```
                                    step = 1,
                                    width = "100%")
```

3.1.6   radioButtons

```
demo_radio <- radioButtons("demo_radio",
                           label = "Choose one",
                           choices = c("Cats", "Dogs"))
```

3.2   Setting inputs programatically

Sometimes you need to change the value of an input with code, such as when resetting a questionnaire or in response to an answer on another item. The following code resets all of the inputs above.

```
updateTextInput(session, "demo_text", value = "")
updateTextAreaInput(session, "demo_textarea", value = "")
updateSelectInput(session, "demo_select", selected = "")
updateCheckboxGroupInput(session, "demo_cbgi", selected = character(0))
updateCheckboxInput(session, "demo_cb", value = TRUE)
updateSliderInput(session, "demo_slider", value = 0)
```

Note that select inputs and checkbox groups use the argument `selected` and not `value`. If you want to set all the values in a checkbox group to unchecked, set `selected = character(0)`.

Chapter 4

Outputs

Output are ways that the Shiny app can dynamically display information to the user. In the user interface (UI), you create outputs with IDs that you reference in an associated rendering function inside the server function.

Explore some different output types in the embedded app below before you read about how to set up each type. You can run this app locally with `shinyintro::app("output_demo")` or view it in a separate tab with the [showcase interface](<https: //shiny.psy.gla.ac.uk/debruine/output_demo/({target="_blank"}.

\begin{figure}

Column to plot

Sepal.Length	▼

plotOutput	verbatimText	imageOutput
tableOutput	dataTableOutput	

{

4.0.1 textOutput

The function `textOutput()` defaults to text inside a `<span>` or `<div>` element, but you can use a different element with the `container` argument.

```
# in the UI
textOutput("demo_text", container = tags$h3)

# in the server function
output$demo_text <- renderText({
    sprintf("Plot of %s", input$y)
})
```

If you use `verbatimTextOutput()` in the UI (no change to the server function), it will show the output in a fixed-width font. This can be good for code or text you want the user to copy.

```
# in the UI
verbatimTextOutput("demo_verbatim")

# in the server function
output$demo_verbatim <- renderText({
        code <-
"ggplot(iris, aes(x = Species, y = %s, color = Species)) +
    geom_violin(show.legend = FALSE) +
    stat_summary(show.legend = FALSE)"

        sprintf(code, input$y)
    })
```

4.0.2 uiOutput

If you want to dynamically create parts of the UI, you can use `uiOutput()`. You can create the UI using the input functions

```
# in the UI
uiOutput("demo_ui")

# in the server function
output$demo_ui <- renderUI({
  cols <- names(iris)[1:4]
  selectInput("y", "Column to plot", cols, "Sepal.Length")
})
```

The function `htmlOutput()` is exactly the same as `uiOutput()`, so you might see that in some code examples, but I use `uiOutput()` to make the connection with `renderUI()` clearer, since there is no `renderHTML()`.

4.0.3 plotOutput

```
# in the UI
plotOutput("demo_plot")

# in the server function
output$demo_plot <- renderPlot({
  ggplot(iris, aes(x = Species, y = .data[[input$y]], color = Species)) +
    geom_violin(show.legend = FALSE) +
    stat_summary(show.legend = FALSE) +
    ylab(input$y)
})
```

If you want to create dynamic plots that change with input, note how you need to use `y = .data[[input$y]]` inside `aes()`, instead of just `y = input$y`.

4.0.4 imageOutput

```
# in the UI
imageOutput("demo_image")

# in the server function
output$demo_image <- renderImage({
    list(src = "images/flower.jpg",
         width = 100,
         height = 100,
         alt = "A flower")
}, deleteFile = FALSE)
```

You can dynamically display images of any type, but one of the most useful ways to use image outputs is to control the aspect ratio and relative size of plots. When you save a temporary file, you should set `deleteFile = TRUE` in `renderImage()` (this stops unneeded plots using memory).

```
# in the UI
imageOutput("demo_image")
```

```r
# in the server function
output$demo_image <- renderImage({
  # make the plot
  g <- ggplot(iris, aes(x = Species, y = .data[[input$y]], color = Species)) +
    geom_violin(show.legend = FALSE) +
    stat_summary(show.legend = FALSE) +
    ylab(input$y)

  # save to a temporary file
  plot_file <- tempfile(fileext = ".png")

  ggsave(plot_file, demo_plot(), units = "in",
         dpi = 72, width = 7, height = 5)

  # Return a list containing the filename
  list(src = plot_file,
       width = "100%", # don't set the height to keep aspect ratio
       alt = "The plot")
}, deleteFile = TRUE)
```

4.0.5   tableOutput

Display a table using `renderTable()`, which makes a table out of any data frame it returns.

```r
# in the UI
tableOutput("demo_table")

# in the server function
output$demo_table <- renderTable({
  iris %>%
    group_by(Species) %>%
    summarise(mean = mean(.data[[input$y]]),
              sd = sd(.data[[input$y]]))
})
```

Note how you need to use `.data[[input$y]]` inside `dplyr::summarise()`, instead of just `input$y` to dynamically choose which variable to summarise.

4.0.6   dataTableOutput

If you have a long table to show, or one that you want users to be able to sort or search, use `dataTableOutput()`. You can customise data tables in many ways, but we'll stick with a basic example here.

```r
# in the UI
DT::dataTableOutput("demo_datatable")

# in the server function
output$demo_datatable <- DT::renderDataTable({
    iris
}, options = list(pageLength = 10))
```

The basic `shiny` package has `dataTableOutput()` and `renderDataTable()` functions, but they can be buggy. The versions in the `DT` package are better and have some additional functions, so I use those.

Chapter 5

Reading and saving data

5.1 Local Data

You can read and write data from a Shiny app the same way you do from any R script. We will focus on reading data, since writing data locally can cause problems and is better done with Google Sheets.

The working directory for a Shiny app is the directory that app.R is in. I recommend keeping your data in a directory called "data" to keep things tidy.

```
# read local data
my_data <- readxl::read_xls("data/my_data.xls")

# read data on the web
countries <- readr::read_csv("https://datahub.io/core/country-list/r/data.csv")
languages <- jsonlite::read_json("https://datahub.io/core/language-codes/r/language-codes.json")
```

5.2 Google Sheets

One of the best ways to start collecting data with a Shiny app is with Google Sheets. This allows you to collect data to the same place from multiple servers, which might happen if you're running the app locally on more than one computer or through a service like shinyapps.io. The R package `googlesheets4` makes it easy to work with Google Sheets from R.

If you just want to read data from a public Google Sheet, you don't need any authorisation. Just start your code with `gs4_deauth()` after you load the `googlesheets4` library (otherwise you'll be prompted to log in). Then you can read data like this:

```
library(googlesheets4)
gs4_deauth()
sheet_id <- "https://docs.google.com/spreadsheets/d/1tQCYQrI4xITlPyxb9dQ-JpMDYeADovIei2
read_sheet(sheet_id)
```

```
## # A tibble: 3 x 2
##   number letter
##    <dbl> <chr>
## 1      1 A
## 2      2 B
## 3      3 C
```

However, even if a Google Sheet is publicly editable, you can't add data to it without authorising your account.

```
data <- data.frame(number = 4, letter = "D")
sheet_append(sheet_id, data)
```

```
## Error: Client error: (401) UNAUTHENTICATED
## * Request not authenticated due to missing, invalid, or expired OAuth token.
## * Request is missing required authentication credential. Expected OAuth 2 access tok
```

You can authorise interactively using the following code (and your own email), which will prompt you to authorise "Tidyverse API Packages" the first time you do this.

```
gs4_auth(email = "debruine@gmail.com")
```

However, this won't work if you want your Shiny apps to be able to access your Google Sheets.

5.2.1   Authorisation for Apps

First, you need to get a token and store it in a cache folder in your app directory. We're going to call that directory ".secrets". Run the following code in your console (NOT in an Rmd file). This will open up a web browser window and prompt you to choose your Google account and authorise "Tidyverse API Packages".

```
setwd(app_directory)
gs4_auth(email = "debruine@gmail.com", cache = ".secrets")

# optionally, authorise google drive to search your drive
# googledrive::drive_auth(email = "debruine@gmail.com", cache = ".secrets")
```
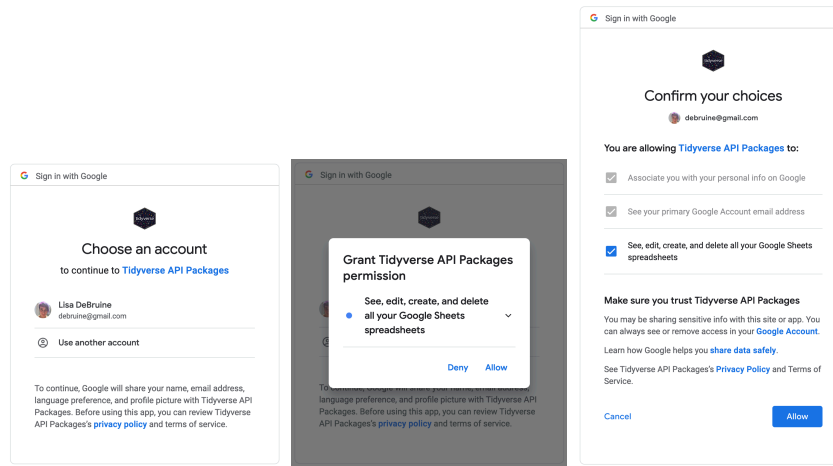
Figure 5.1: Prompts to choose an account, grant permissions, and confirm.

When you have finished, you will see a page that says something like, "Authentication complete. Please close this page and return to R." In the file pane in RStudio, you should now see a directory. called ".secrets" in the app directory.

If you are using GitHub, you don't want to save your secret info to a public repository, so run the following code to ignore any directories called ".secrets" (so they will only exist on your computer and not on GitHub).

```
usethis::use_git_ignore(".secrets")
usethis::use_git_ignore("*/.secrets")
```

Now, you can include the following code at the top of your app.R script to authorise the app to read from and write to your files.

```
gs4_auth(cache = ".secrets", email = "debruine@gmail.com")
```

5.2.2 Accessing an existing sheet

If you have an existing Google Sheet, you can access it by URL.

```
sheet_id <- "https://docs.google.com/spreadsheets/d/1tQCYQrI4xITlPyxb9dQ-JpMDYeADovIeiZZRNHkctGA/
data <- data.frame(number = 4, letter = "D")
sheet_append(sheet_id, data)
read_sheet(sheet_id)
```

```
## # A tibble: 4 x 2
##   number letter
##    <dbl> <chr>
## 1      1 A
## 2      2 B
## 3      3 C
## 4      4 D
```

5.2.3   Make a new sheet

You can set up a new Google Sheet with code. You only need to do this once for a sheet that you will use with a Shiny app, and you will need to save the sheet ID. If you don't specify the tab name(s), the sheet will be created with one tab called "Sheet1". I recommend making only one sheet per app and saving each table in a separate tab.

```r
id <- gs4_create("demo2", sheets = c("demographics", "questionnaire"))
id
```

```
## Spreadsheet name: demo2
##               ID: 1KcCQDaTS5ngeNgAaOSvV856jULlihfWV_yxjNSYVOT4
##           Locale: en_US
##        Time zone: Europe/London
##      # of sheets: 2
##
##  (Sheet name): (Nominal extent in rows x columns)
##   demographics: 1000 x 26
## questionnaire: 1000 x 26
```

Include the ID at the top of your app like this:

5.2.4   Add data

You can add an empty data structure to your sheet by specifying the data types of each column like this:

```r
data <- data.frame(
  name = character(0),
  birthyear = integer(0),
  parent = logical(0),
  score = double(0)
)
```

```
write_sheet(data, SHEET_ID, "demographics")
read_sheet(SHEET_ID, "demographics") %>% names()
```

```
## [1] "name"      "birthyear" "parent"    "score"
```

Or you can populate the table with starting data.

```
data <- data.frame(
  name = "Lisa",
  birthyear = 1976L,
  R_user = TRUE,
  score = 10.2
)

write_sheet(data, SHEET_ID, "demographics")
read_sheet(SHEET_ID, "demographics")
```

```
## # A tibble: 1 x 4
##   name  birthyear R_user score
##   <chr>     <dbl> <lgl>  <dbl>
## 1 Lisa       1976 TRUE    10.2
```

Notice that `birthyear` is a double, not an integer. Google Sheets only have one numeric type, so both doubles and integers are coerced to doubles.

5.2.5 Appending data

Then you can append new rows of data to the sheet.

```
data <- data.frame(
  name = "Robbie",
  birthyear = 2007,
  R_user = FALSE,
  score = 12.1
)

sheet_append(SHEET_ID, data, "demographics")
read_sheet(SHEET_ID, "demographics")
```

```
## # A tibble: 2 x 4
##   name   birthyear R_user score
##   <chr>      <dbl> <lgl>  <dbl>
## 1 Lisa        1976 TRUE    10.2
## 2 Robbie      2007 FALSE   12.1
```

If you try to append data of a different type, some weird things can happen.
Logical values added to a numeric column are cast as 0 (FALSE) and 1
(TRUE), while numeric values added to a logical column change the column to
numeric. If you mix character and numeric values in a column, the resulting
column is a column of one-item lists so that each list can have the appropriate
data type. (Data frames in R cannot mix data types in the same column.)

```r
data <- data.frame(
  name = 1,
  birthyear = FALSE,
  R_user = 0,
  score = "No"
)

sheet_append(SHEET_ID, data, "demographics")
read_sheet(SHEET_ID, "demographics")
```

```
## # A tibble: 3 x 4
##   name       birthyear R_user score
##   <list>         <dbl>  <dbl> <list>
## 1 <chr [1]>       1976      1 <dbl [1]>
## 2 <chr [1]>       2007      0 <dbl [1]>
## 3 <dbl [1]>          0      0 <chr [1]>
```

You must append data that has the same number and order of columns as the
Google Sheet. If you send columns out of order, they will be recorded in the
order you sent them, not in the order of the column names. If you send extra
columns, the append will fail.

The Shiny template we're working with has a safer version of `sheet_append()`
that you can access by uncommenting the line:

<div align="center">

`# source("R/sheet_append.R")`

</div>

This version gracefully handles data with new columns, missing columns,
columns in a different order, and columns with a different data type. However,
it reads the whole data sheet before deciding whether to append or overwrite
the data, which can slow down your app, so is best used only during
development when you're changing things a lot. Once you have the final
structure of your data, it's better to use the original
`googlesheets4::sheet_append()` function.

5.3   Saving Googlesheet data

If you mix data types in a column, the data frame returned by
`googlesheets4::read_sheet()` has list columns for any mixed columns.

Dates can also get written in different ways that look the same when you print to the console, but are a mix of characters and doubles, so you have to convert them to strings like this before you can save as CSV.

```
string_data <- lapply(data, sapply, toString) %>% as.data.frame()
readr::write_csv(string_data, "data.csv")
```

5.4 Exercises

5.4.1 Read others' data

Read in data from the public google sheet at https://docs.google.com/spreadsheets/d/1QjpRZPNOOL0pfRO6IVT5WiafnyNdahsch1A03iHdv7s/. Find the sheet ID and figure out which sheet has data on US states (assign this to the object `states`) and which has data on eutherian mammals (assign this to `mammals`).

Solution

```
library(googlesheets4)

gs4_deauth()

sheet_url <- "https://docs.google.com/spreadsheets/d/1QjpRZPNOOL0pfRO6IVT5WiafnyNdahsch1A03iHdv7s
sheet_id <- as_sheets_id(sheet_url)

states <- read_sheet(sheet_id, 1)
mammals <- read_sheet(sheet_id, 2)
```

5.4.2 Read your own data

Create a google sheet online and read its contents in R. You will need to either make it public first (click on the green Share icon in the upper right) or authorise googlesheets to access your account.

Solution

```
gs4_auth()
my_sheet_url <- ""
mydata <- read_sheet(my_sheet_url)
```

5.4.3 Write data

Append some data to your google sheet

5.4.4   Shiny app

In the app you're developing, determine what data need to be saved and set up a google sheet (or local data if you're having trouble with google). Write the server function to save data from your app when an action button is pressed.

Chapter 6

Reactive functions

There are a lot of great tutorials that explain the principles behind reactive functions, but that never made any sense to me when I first started out, so I'm just going to give you examples that you can extrapolate principles from.

Reactivity is how Shiny determines which code in `server()` gets to run when. Some types of objects, such as the `input` object or objects made by `reactiveValues()`, can trigger some types of functions to run whenever they change.

For our example, we will use the `reactive_demo` app. It shows three select inputs that allow the user to choose values from the cut, color, and clarity columns of the `diamonds` dataset from ggplot2, then draws a plot of the relationship between carat and price for the selected subset.

\begin{figure}

Diamonds

Cut

Fair   ▼

Color

D   ▼

Clarity

I1   ▼

Update Plot

{Plot

Here is the relevant code for the UI. There are four inputs: cut, color, clarity, and update. There are two outputs: title and plot.

```
box(
  title = "Diamonds",
  solidHeader = TRUE,
  selectInput("cut", "Cut", levels(diamonds$cut)),
  selectInput("color", "Color", levels(diamonds$color)),
  selectInput("clarity", "Clarity", levels(diamonds$clarity)),
  actionButton("update", "Update Plot")
),
box(
  title = "Plot",
  solidHeader = TRUE,
  textOutput("title"),
```

```
  plotOutput("plot")
)
```

Whenever an input changes, it will trigger some types of functions to run.

6.1 observeEvent()

We learned about `observeEvent()` in Chapter 1. This function runs the code whenever the value of the first argument changes. If there are reactives inside the function, they won't trigger the code to run when they change.

```
server <- function(input, output, session) {
  observeEvent(input$update, {
    data <- filter(diamonds,
                   cut == input$cut,
                   color == input$color,
                   clarity == input$clarity)

    title <- sprintf("Cut: %s, Color: %s, Clarity: %s",
                     input$cut,
                     input$color,
                     input$clarity)

    output$plot <- renderPlot({
      ggplot(data, aes(carat, price)) +
        geom_point(color = "#605CA8", alpha = 0.5) +
        geom_smooth(method = lm, color = "#605CA8")
    })

    output$title <- renderText(title)
  })
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut color clarity update cut, color & clarity all of the above

In the example above, which inputs will trigger `renderText()` to run and produce a new title?

cut color clarity update cut, color & clarity all of the above

6.2   Render functions

Functions that render an output, like `renderText()` or `renderPlot()` will run whenever an input in their code changes. You can trigger a render function just by putting a reactive alone on a line, even if you aren't using it in the rest of the code.

```r
server <- function(input, output, session) {
  output$plot <- renderPlot({
    data <- filter(diamonds,
                   cut == input$cut,
                   color == input$color,
                   clarity == input$clarity)

    ggplot(data, aes(carat, price)) +
      geom_point(color = "#605CA8", alpha = 0.5) +
      geom_smooth(method = lm, color = "#605CA8")
  })

  output$title <- renderText{
    input$update

    sprintf("Cut: %s, Color: %s, Clarity: %s",
                       input$cut,
                       input$color,
                       input$clarity)
  })
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut color clarity update cut, color & clarity all of the above

Which inputs will trigger `renderText()` to run and produce a new title?

cut color clarity update cut, color & clarity all of the above

6.3   reactive()

If you move the `data` filtering outside of `renderPlot()`, you'll get an error message like "Can't access reactive value 'cut' outside of reactive consumer." This means that the `input` values can only be read inside certain functions, like `reactive()`, `observeEvent()`, or a render function.

However, we can put the data filtering inside `reactive()`. This means that whenever an input inside that function changes, the code will run and update the value of `data()`. This can be useful if you need to recalculate the data table each time the inputs change, and then use it in more than one function.

```
server <- function(input, output, session) {
  data <- reactive({
    filter(diamonds,
           cut == input$cut,
           color == input$color,
           clarity == input$clarity)
  })

  title <- reactive({
    sprintf("Cut: %s, Color: %s, Clarity: %s, N: %d",
                    input$cut,
                    input$color,
                    input$clarity,
            nrow(data()))
  })

  output$plot <- renderPlot({
    ggplot(data(), aes(carat, price)) +
      geom_point(color = "#605CA8", alpha = 0.5) +
      geom_smooth(method = lm, color = "#605CA8")
  })

  output$text <- renderText(title())
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut   color   clarity   update   cut, color & clarity   all of the above

Which inputs will trigger `renderText()` to run and produce a new title?

cut   color   clarity   update   cut, color & clarity   all of the above

My most common error is trying to use `data` or `title` as an object instead of as a function. Notice how the first argument to ggplot is no longer `data`, but `data()` and you set the value of data with `data(newdata)`, not `data <- newdata`. For now, just remember this as a quirk of shiny.

6.4 eventReactive()

While `reactive()` is triggered whenever any input values inside it change, `eventReactive()` is only triggered when the value of the first argument changes, like `observeEvent()`, but returns a reactive function like `reactive()`.

```
server <- function(input, output, session) {
  data <- eventReactive(input$update, {
    filter(diamonds,
           cut == input$cut,
           color == input$color,
           clarity == input$clarity)
  })

  title <- eventReactive(input$update, {
    sprintf("Cut: %s, Color: %s, Clarity: %s",
                       input$cut,
                       input$color,
                       input$clarity)
  })

  output$plot <- renderPlot({
    ggplot(data(), aes(carat, price)) +
      geom_point(color = "#605CA8", alpha = 0.5) +
      geom_smooth(method = lm, color = "#605CA8")
  })

  output$text <- renderText(title())
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut color clarity update cut, color & clarity all of the above

Which inputs will trigger `renderText()` to run and produce a new title?

cut color clarity update cut, color & clarity all of the above

6.5 reactiveVal()

Another pattern that accomplishes a similar thing to `reactive()` is to create a reactive value using `reactiveVal()`. This allows you to update the value of `data()` not just using the code inside the `reactive()` function that created it,

but in any function. This is useful when you have multiple functions that need to update that value.

Here, we use `observeEvent()` to trigger the data filtering code only when the update button is pressed. This new data set is assigned to `data()` using the code `data(newdata)`.

Because `data()` is a reactive value, it will trigger `renderPlot()` whenever it changes.

```r
server <- function(input, output, session) {
    data <- reactiveVal(diamonds)

    observeEvent(input$update, {
        newdata <- filter(diamonds,
                cut == input$cut,
                color == input$color,
                clarity == input$clarity)

        data(newdata) # updates data()
    })

    output$plot <- renderPlot({
        ggplot(data(), aes(carat, price)) +
            geom_point(color = "#605CA8", alpha = 0.5) +
            geom_smooth(method = lm, color = "#605CA8")
    })

    output$title <- renderText({
      sprintf("Cut: %s, Color: %s, Clarity: %s",
                    input$cut,
                    input$color,
                    input$clarity)
    })
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut   color   clarity   update   cut, color & clarity   all of the above

Which inputs will trigger `renderText()` to run and produce a new title?

cut   color   clarity   update   cut, color & clarity   all of the above

We used `data <- reactiveVal(diamonds)` in order for `data()` to have a value that didn't cause an error when `renderPlot()` runs for the first time.

6.6   reactiveValue()

You need to set up a new `reactiveVal()` for each value in an app that you want to make reactive. I prefer to use `reactiveValues()` because it can be used for any new reactive value you need and works just like `input`.

You can just set your new object to `reactiveValues()` or you can initialise it with starting values like below. The object `v` is a named list, just like `input`, and when it's values change, it triggers reactive functions exactly like `input` does.

```r
server <- function(input, output, session) {
    v <- reactiveValues(
        data = diamonds,
        title = "All Data"
    )

    observeEvent(input$update, {
        v$data <- filter(diamonds,
                         cut == input$cut,
                         color == input$color,
                         clarity == input$clarity)

        v$title <- sprintf("Cut: %s, Color: %s, Clarity: %s",
                            input$cut,
                            input$color,
                            input$clarity)
    })

    output$plot <- renderPlot({
        ggplot(v$data, aes(carat, price)) +
            geom_point(color = "#605CA8", alpha = 0.5) +
            geom_smooth(method = lm, color = "#605CA8")
    })

    output$title <- renderText(v$title)
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut color clarity update cut, color & clarity all of the above

Which inputs will trigger `renderText()` to run and produce a new title?

cut color clarity update cut, color & clarity all of the above

Note that you refer to reactive values set up this way as `v$data` and `v$title`, not `data()` and `title()`.

6.7 isolate()

If you want to use an input or reactive value inside a reactive function, but don't want to trigger that function, you can `isolate()` it. You can also use `isolate()` to use a reactive value outside a reactive function.

```
server <- function(input, output, session) {
  data <- reactive({
    filter(
      diamonds,
      cut == isolate(input$cut),
      color == isolate(input$color),
      clarity == input$clarity
    )
  })

  title <- reactive({
    sprintf(
      "Cut: %s, Color: %s, Clarity: %s",
      input$cut,
      isolate(input$color),
      isolate(input$clarity)
    )
  })

  # what is the title at initialisation?
  debug_msg(isolate(title()))

  output$plot <- renderPlot({
    ggplot(data(), aes(carat, price)) +
      geom_point(color = "#605CA8", alpha = 0.5) +
      geom_smooth(method = lm, color = "#605CA8")
  })

  output$title <- renderText(title())
}
```

In the example above, which inputs will trigger `renderPlot()` to run and produce a new plot?

cut color clarity update cut, color & clarity all of the above

Which inputs will trigger `renderText()` to run and produce a new title?

cut color clarity update cut, color & clarity all of the above

6.8 Further Resources {#resources-reactive)

- Reactivity - An overview
- Use reactive expressions

6.9 Exercises

For the following exercises, clone "reactive_demo" and replace the boxes in the ui with the code below. Delete all the code in `server()`. Make sure this runs before you go ahead.

```r
box(width = 4,
    selectInput("stat", "Statistic", c("mean", "sd")),
    selectInput("group", "Group By", c("vore", "order", "conservation")),
    actionButton("update", "Update Table")),
box(width = 8,
    solidHeader = TRUE,
    title = textOutput("caption"),
    tableOutput("table"))
```

You will grouping and summarising the `msleep` data table from `ggplot2` by calculating the mean or standard deviation for all (or some) of the numeric columns grouped by the categorical columns vore, order, or conservation. If you're not sure how to create such a summary table with dplyr, look at the following code for a concrete example.

Hint

```r
msleep %>%
  group_by(vore) %>%
  summarise_if(is.numeric, "mean", na.rm = TRUE)
```

```
## # A tibble: 5 x 7
##   vore    sleep_total sleep_rem sleep_cycle awake brainwt  bodywt
##   <chr>         <dbl>     <dbl>       <dbl> <dbl>   <dbl>   <dbl>
## 1 carni          10.4      2.29       0.373  13.6  0.0793    90.8
## 2 herbi           9.51     1.37       0.418  14.5  0.622    367.
## 3 insecti        14.9      3.52       0.161   9.06 0.0216    12.9
## 4 omni           10.9      1.96       0.592  13.1  0.146     12.7
## 5 <NA>           10.2      1.88       0.183  13.8  0.00763    0.858
```

6.9.1   observeEvent

Use `observeEvent()` to update the output table with the appropriate summary table and to update the caption with an appropriate caption only when the update button is clicked.

Solution

```
server <- function(input, output, session) {
    observeEvent(input$update, {
        data <- msleep %>%
            group_by(.data[[input$group]]) %>%
            summarise_if(is.numeric, input$stat, na.rm = TRUE)
        output$table <- renderTable(data)

        caption <- sprintf("%ss by %s", toupper(input$stat), input$group)
        output$caption <- renderText(caption)
    })
}
```

6.9.2   render

Use render functions to update the output table and caption whenever group or stat change.

Solution

```
server <- function(input, output, session) {
    output$table <- renderTable({
        msleep %>%
            group_by(.data[[input$group]]) %>%
            summarise_if(is.numeric, input$stat, na.rm = TRUE)
    })

    output$caption <- renderText({
        sprintf("%ss by %s", toupper(input$stat), input$group)
    })
}
```

6.9.3   reactive

Use reactive functions to update the output table and caption whenever group or stat change. Ignore the update button.

Solution

```r
server <- function(input, output, session) {
  data <- reactive({
    msleep %>%
      group_by(.data[[input$group]]) %>%
      summarise_if(is.numeric, input$stat, na.rm = TRUE)
  })
  output$table <- renderTable(data())

  caption <- reactive({
    sprintf("%ss by %s", toupper(input$stat), input$group)
  })
  output$caption <- renderText(caption())
}
```

6.9.4   eventReactive

Use `eventReactive()` to update the output table and caption only when the update button is clicked.

Solution

```r
server <- function(input, output, session) {
  data <- eventReactive(input$update, {
    msleep %>%
      group_by(.data[[input$group]]) %>%
      summarise_if(is.numeric, input$stat, na.rm = TRUE)
  })
  output$table <- renderTable(data())

  caption <- eventReactive(input$update, {
    sprintf("%ss by %s", toupper(input$stat), input$group)
  })
  output$caption <- renderText(caption())
}
```

6.9.5   reactiveVal

Use `reactiveVal()` to update the output table and caption only when the update button is clicked.

Solution

```r
server <- function(input, output, session) {
  data <- reactiveVal()
```

```
  caption <- reactiveVal()

  observeEvent(input$update, {
    newdata <- msleep %>%
      group_by(.data[[input$group]]) %>%
      summarise_if(is.numeric, input$stat, na.rm = TRUE)
    data(newdata)

    # this is an alternative way to set reactiveVal
    # by piping the value into the function
    sprintf("%ss by %s", toupper(input$stat), input$group) %>%
      caption()
  })

  output$table <- renderTable(data())
  output$caption <- renderText(caption())
}
```

6.9.6   reactiveValues

Use `reactiveValues()` to update the output table and caption only when the
update button is clicked.

Solution

```
server <- function(input, output, session) {
  v <- reactiveValues()

  observeEvent(input$update, {
    v$data <- msleep %>%
      group_by(.data[[input$group]]) %>%
      summarise_if(is.numeric, input$stat, na.rm = TRUE)

    v$caption <- sprintf("%ss by %s", toupper(input$stat), input$group)
  })

  output$table <- renderTable(v$data)
  output$caption <- renderText(v$caption)
}
```

Chapter 7

HTML, CSS, and JavaScript

You don't need to know anything about HTML, CSS and JavaScript to make basic Shiny apps, but a little knowledge can really help you customise your apps. This chapter will cover some of the basics so you have enough vocabulary to get started.

7.1 HTML

HTML stands for Hyper-Text Markup Language, a system for semantically tagging structure and information on web pages. The term "semantically" is important here; HTML should tell you **what** something is, not **how** to display it (that's handled by CSS). This separation helps your apps be accessible to people who use screen readers.

7.1.1 HTML Tags

We learned about the `tags()` function in Chapter 1 and how it is linked to HTML tags. For example, the R code `tags$h2("Methods")` creates the HTML `<h2>Methods</h2>`. It surrounds the text content with a starting tag `<h2>` and an ending tag `</h2>`.

Tags create elements, which can be thought of kind of like boxes. An element can contain one or more elements. Elements can have attributes that provide more information such as the element's ID or class, which can be used by CSS or JavaScript to refer to the element or a group of elements.

For example, the following code creates an unordered list (`<ul>`) with the class "animals". It contains three list item elements (`<li>`), each with its own id.

```
<ul class="animals">
    <li id="aardvark">Aardvarks</li>
    <li id="bee">Bees</li>
    <li id="capybara">Capybaras</li>
</ul>
```

You seldom *have to* write HTML directly in Shiny, but if you have experience with HTML, it can sometimes be easier to create something in HTML than with the relevant Shiny functions. For example, the code above in Shiny would be:

```
tags$ul(class = "animals",
        tags$li("aardvark", "Aardvark"),
        tags$li("bee", "Bee"),
        tags$li("capybara", "Capybara")
)
```

Alternatively, you can use `HTML()` to include raw HTML in a ui function.

```
HTML('<ul class="animals">
    <li id="aardvark">Aardvarks</li>
    <li id="bee">Bees</li>
    <li id="capybara">Capybaras</li>
</ul>')
```

7.1.2   Viewing HTML

But the main reason for you to learn a little about HTML is that it can help you to customise the appearance of your Shiny apps using CSS and their behaviour using JavaScript.

Often, you'll need to find out how to refer to a specific element or group of elements that were created by a shiny ui function. For example, it isn't obvious how you'd refer to the sidebar tab for `x_tab` in the following code:

```
sidebarMenu(
    id = "tabs",
    menuItem("Tab Title", tabName = "x_tab", icon = icon("dragon"))
)
```

If you right click on the resulting app and choose "View Page Source", you'll (eventually) find that the code above created the HTML below.

```
<ul class="sidebar-menu">
  <li>
    <a href="#shiny-tab-x_tab" data-toggle="tab" data-value="x_tab">
      <i class="fa fa-dragon" role="presentation" aria-label="dragon icon"></i>
      <span>Tab Title</span>
    </a>
  </li>
  <div id="tabs" class="sidebarMenuSelectedTabItem" data-value="null"></div>
</ul>
```

Now you know that the dragon icon is made by an italics tag (`<i>`) that's inside an anchor (`<a>`) with the id "#shiny-tab-x_tab". Therefore, you can change the colour of this icon using the following CSS:

```
#shiny-tab-x_tab i { color: red; }
```

It can be tricky to find what code you're looking for, but developer tools can help. I use FireFox Developer Edition when I'm developing web apps, but Chrome also has developer tools. In FireFox, go to `Tools > Browser Tools > Web Developer Tools` (opt-cmd-I). In Chrome, go to `View > Developer > Developer Tools` (opt-cmd-I). You can dock the tools to the bottom, right , or left of the window, or as a separate window.

Open the Inspector (FireFox) or Elements (Chrome) tab of the tools and click on the icon that looks like an arrow pointing into a box. When you hover over parts of the web page, not you will see boxes outlining each element. You can click on an element to highlight its HTML in the tools.

7.2   CSS

CSS stands for Cascading Style Sheets and is a way to control the visual presentation of HTML on web pages. You can use CSS to change the default appearance of anything on a web page.

7.2.1   CSS Basics

CSS is structured as follows:

```
selector { property: value; }
```

Selectors are how you refer to the parts of the HTML you want to style. There are dozens of selectors and they can get very complex. We'll focus on some basic examples below.
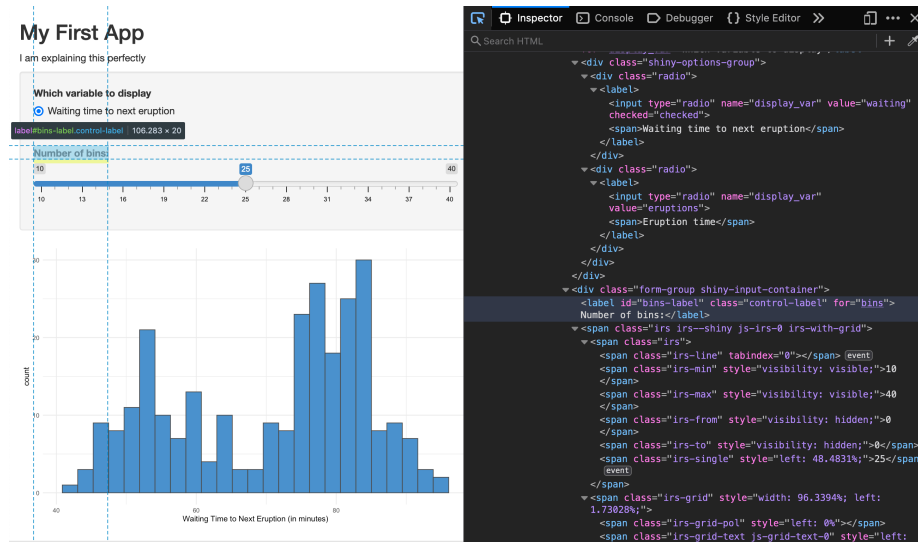
Figure 7.1: Viewing the HTML with the inspector.

Properties are aspects of the visual style that you want to change, such as "color" (text color), "background-color", or "border". Values have specific formats for each property, such as "1px" or "0.5em" to describe lengths, or "red" or "#FF0000" to describe colours.

There are hundreds of properties and you can't memorise them all. I usually Google something like "css font type" or "css underline text" and choose the first link from ww3schools.

For example, if you have HTML that looks like this:

```html
<ul class="animals">
    <li id="aardvark">Aardvarks</li>
    <li id="bee">Bees</li>
    <li id="capybara">Capybaras</li>
</ul>
```

You can refer to different parts of the list in many ways:

CSS	Meaning
.animals { border: 1px solid green; }	any element with the class "animals" will get a 1-pixel green border
ul.animals { border: 1px solid green; }	any unordered list with the class "animals" will get a 1-pixel green border

CSS	Meaning
`.animals li { color: blue; }`	any list item element inside an element with the class "animals" will have blue text
`#bee { font-style: italic; }`	the element with the id "bee" will be in italics
`.animals li + li { background-color: grey; }`	inside an element with the class "animals", any list item that follows another list item will have a grey background

7.2.2 Styling a single element

You can add styles to most elements that you make with shiny ui functions by adding an argument called `style`.

```
tags$ul(class = "animals", style="width: 10em;",
        tags$li("Aardvark"),
        tags$li("Bee", style="background-color: yellow;"),
        tags$li("Capybara")
)
```

7.2.3 Inline CSS

You can add a small amount of CSS to an app inside the header using the style tag. The code below makes the element with the class "animal" 10em wide (*em* is a unit of size that is proportional to text size). It also makes the element with the id "bee" italic and gives it a black background with yellow text. If the element is a list item, it makes the marker a bee emoji.

```
mystyle <- '
  .animal { width: 10em; }
  #bee {
    font-style: italic;
    color: yellow;
    background-color: black;
  }
  li#bee::marker { content: "  "; }
'

ui <- fluidPage(
  tags$head(
    tags$style(mystyle)
  ),
```

```
  tags$ul(class = "animals",
         tags$li("Aardvark", class = "aardvark"),
         tags$li("Bee", class = "bee"),
         tags$li("Capybara", class = "capybara")
  )
)
```

Aardvarks

Bees

Capybaras

7.2.4   External CSS

For anything longer than a few lines, you can see how this can get tedious. You can put all of your CSS in an external file and reference that in the header instead using `tags$link()`. The CSS file needs to be inside the `www` directory to let Shiny know that it's meant to be included like this. The template we're using in this class comes with a CSS file called `custom.css`.

```
tags$head(
  tags$link(rel = "stylesheet", type = "text/css", href = "custom.css")
)
```

Sometimes when you change external files, they don't seem to update when you test the app. This can be because of caching. You can usually solve this by reloading the app in your web browser, reloading in the web browser with the shift key pressed, stopping the app from running in RStudio with the stop sign icon and starting it up again, and, finally, restarting R.

7.3   JavaScript

JavaScript is a coding language that is very useful for adding dynamic behaviour to web pages. For simple apps, you don't need to understand any JavaScript, but a little bit can be really helpful for adding advanced behaviour.

7.3.1   shinyjs

The R package shinyjs provides several ways to work with JavaScript in a Shiny app. In orde to set it up so that your server function can use shinyjs function, you need to add `shinyjs::useShinyjs()` somewhere in your ui function.

Here is a list of the shinyjs functions that I find most useful (`id` refers to an element with the specified ID):

- `hide(id)`: hide the element
- `show(id)`: show the element
- `toggle(id)`: change the visibility of the element (hide it if it's visible, and show it if it's not)
- `alert(text)`: create an alert popup; this is useful for debugging
- `addClass(id, class)`: adds a CSS class to the element
- `removeClass(id, class)`: removes a CSS class from the element
- `click(id)`: simulates a click on the action button
- `disable(id)`: disable an input
- `enable(id)`: enables an input
- `reset(id)`: resets an input (much easier than the update* functions)

7.3.2 External JS

To do anything more complicated, it's best to put your JavaScript in an external file in the `www` directory. You can include a link to this script in the header. The template we're using in this class comes with a JavaScript file called "custom.js".

```
tags$head(
  tags$script(src = "custom.js")
)
```

Shiny apps use jQuery, a framework for making JavaScript easier to write. It lets you refer to elements using their CSS selectors.

Here is some example code from the custom.js file in the template.

```
$(document).on("shiny:connected", function() {
  // send window width to shiny
  shiny_size = function() {
    Shiny.setInputValue("window_width", window.innerWidth);
    Shiny.setInputValue("window_height", window.innerHeight);
  }

  window.onresize = shiny_size;
  shiny_size(); // trigger once at start
})
```

JavaScript is similar to R in some ways, and maddeningly different in others. One big difference is that lines of code have to end with a semi-colon.

In the code above, the function `$(document).on("shiny:connected",`
`function() { ... })` is jQuery shorthand for making sure that the code
inside doesn't run until the whole webpage has been downloaded and the extra
javascript for shiny is available. Otherwise, you might try to run some code
that references an element that hasn't been created yet (HTML pages don't
always download all in one go) or uses a Shiny function that isn't available yet.

The we create a new function called `shiny_size`, which creates two new Shiny
input variables, "window_width" and "window_height", and sets then to the
values of the window dimensions (in pixels). The line `window.onresize =`
`shiny_size;` sets this function to run every time the window is resized and
the function is run once at the start to initialise those values.

The javaScript function `Shiny.setInputValue(input_id, value)` is a way
for you to communicate things that happen on the web page to the Shiny app
by changing or creating inputs. You can use this inside `server()` to, for
example, change a plot style if `input$window_width < 600`.

7.4   Further Resources

- W3 Schools HTML Tutorial
- Using Custom CSS in your App
- CSS Selectors Reference
- Packaging JavaScript Code for Shiny
- W3 Schools jQuery Tutorial
- Codeacademy interactive tutorials

7.5   Exercises

Clone the basic template for these exercises.

7.5.1   Add HTML

Add the following HTML before the image in `x_tab` using `HTML()`:

```
<p class="help">For more help, you can go to <a href="https://shiny.rstudio.com/article
```

Hint

Since the HTML has double quotes in it, you either need to escape them or
surround the string in single quotes instead.

Solution

```
x_tab <- tabItem(
    tabName = "x_tab",
    HTML('<p class="help">For more help, you can go to <a href="https://shiny.rstudio.com/article
    imageOutput("logo")
)
```

7.5.2   Class HTML

Add an unordered list under the paragraph that contains the following text
and links:

- Tags Glossary
- HTML Tutorial

Solution

```
x_tab <- tabItem(
    tabName = "x_tab",
    HTML('<p class="help">For more help, you can go to <a href="https://shiny.rstudio.com/article
    tags$ul(
        tags$li(tags$a(href = "https://shiny.rstudio.com/articles/tag-glossary.html", "Tags Gloss
        tags$li(tags$a(href = "https://www.w3schools.com/html/", "HTML Tutorial"))
    ),
    imageOutput("logo")
)
```

7.5.3   Style links

Change the style of all the links to make them hotpink. Use the inline CSS
method. As a bonus, change the colour of links when you hover over them, too.

Solution

```
tags$head(
    tags$style("a { color: hotpink; }
                a:hover { color: red; }")
)
```

7.5.4   Style a single link directly

Change the style of just the first link in the list to make it green.

Solution

```
tags$ul(
        tags$li(tags$a(href = "https://shiny.rstudio.com/articles/tag-glossary.html",
        tags$li(tags$a(href = "https://www.w3schools.com/html/", "HTML Tutorial"))
    )
```

7.5.5   External CSS

Change the style of just the links inside paragraphs with the class "help" to
have an underline. Use external CSS.

Solution

Add the following to `www/custom.css`:

```
.help a { text-decoration: underline; }
```

Chapter 8

Structuring a complex app

So far, we've been mostly structuring our app entirely in the `app.R` file, apart from some of the web helper files for CSS and JavaScript. However, once your apps start getting relatively complex, you might find it easier to move some of the code into external `.R` files and using `source()` to include them. There are a few things to watch out for when you do this.

8.1   External Server Functions

You can define functions you want to use in your app at the top of the `app.R` file, but that can make that file difficult to parse pretty quickly. The basic template includes external functions with the line:

```
source("scripts/func.R") # helper functions
```

This file contains definitions for the functions `debug_msg()` and `debug_sprintf()`. You can add your own custom functions to this file or to another file that you source in separately.

All `.R` files inside a directory called `R` will run before the app starts up, even if you don't source them into the app. You can use this to set up any functions your app needs without having to use `source()`, but I prefer to explicitly include external files, so I keep external functions in a directory called `scripts`.

8.1.1   Sourcing Locally

It can be tricky to use shiny functions to external files. For example, you can't just move the contents of `server()` to an external file called `scripts/logo.R`

and source the file in like this:

```r
server <- function(input, output, session) {
    source("scripts/logo.R")
}
```

You'll get an error like: "Error in output$logo <- renderImage({ : object 'output' not found". This is because the input and output objects only work like you'd expect when they are inside the `server()` function.

However, you can source in external code inside `server()` by setting the `local` argument to TRUE.

```r
server <- function(input, output, session) {
    source("scripts/logo.R", local = TRUE)
}
```

You might find it useful to break up parts of the server logic for a very big app into separate files like this, but it's more common to keep any code that uses reactive functions inside `server()` in the `app.R` file, and move large sections of code inside those functions to externally defined functions.

For example, you could define the function `logo_image()` in the external file `scripts/logo.R` like this:

```r
logo_image() <- function() {
  list(src = "www/img/shinyintro.png",
       width = "300px",
       height = "300px",
       alt = "ShinyIntro hex logo")
}
```

The following in the `app.R` file keeps the reactive function `renderImage()` inside `server()`, but lets you reduce the number of lines of code.

```r
source("scripts/logo.R")

server <- function(input, output, session) {
    source("scripts/logo.R", local = TRUE)
    output$logo <- renderImage(logo_image(), deleteFile = FALSE)
}
```

8.1.2   Inputs and Outputs

The objects `input` and `output` aren't available by default to externally defined functions. Let's add an action button to the ui for our app,

actionButton("change", "Change Image"), and change the `logo_image()` function so that it returns the ShinyIntro logo on odd-numbered clicks of the change button, and the psyTeachR logo on even-numbered clicks.

```
logo_image <- function() {
  odd_clicks <- input$change%%2 == 1
  src <- ifelse(odd_clicks,
                "www/img/shinyintro.png",
                "www/img/psyteachr.png")

  list(src = src,
       width = "300px",
       height = "300px",
       alt = "ShinyIntro hex logo")
}
```

If you try to run this, you'll get an error message like, "Error in logo_image: object 'input' not found". This is because the external function doesn't have access to reactive objects like `input`, `output`, `session`, or any `reactiveValues()`.

The best solution is to pass any variables to the function that you need. In some circumstances, you can pass the whole `input` object, but that's seldom necessary.

Here, we change `logo_image()` to take a single argument called `change` and replace `input$change` with this argument.

```
logo_image <- function(change) {
  odd_clicks <- change%%2 == 1
  src <- ifelse(odd_clicks,
                "www/img/shinyintro.png",
                "www/img/psyteachr.png")

  list(src = src,
       width = "300px",
       height = "300px",
       alt = "ShinyIntro hex logo")
}
```

Then we just have to pass the value of `input$change` to `logo_image()` inside `renderImage()`, where the `input` object is available.

```
server <- function(input, output, session) {
    output$logo <- renderImage({
        logo_image(input$change)
```

```
    }, deleteFile = FALSE)
}
```

Don't worry too much if this isn't making a lot of sense yet. The main thing I want you to take away from this section is that when you try to move some server code to external files, you might get errors (I frequently do). I hope that will remind you of this lesson and you'll have a better idea about where to start looking for the solution.

8.2   External UI Files

Defining a complex UI can be very challenging. The basic template uses a pattern that I find helpful with apps that have multiple tab items. I assign each tab to an object and then include the tabs of the app in `dashboardBody()` like this:

```
tabItems(
    intro_tab,
    questionnaire_tab,
    feedback_tab,
    info_tab
)
```

For a simple app, you can define the tabs in `app.R` just before you define the `ui`. You can do the same for any components of the `ui`, such as `dashboardHeader()` or `dashboardSidebar()`. When the sections start getting complex, you can move them into external files and source them in.

8.2.1   UI Lists

Select the Categories

name

Leia Organa

hair_color

brown

skin_color

fair

eye_color

brown

sex

female

gender

feminine

homeworld

Alderaan

species

Human

When parts of the UI repeat or can be created programmatically instead of manually, you can use `apply()` or `map()` functions to create a list of UI components. It can be a little tricky to figure out how to add a list of components into the UI, but this can be accomplished with `do.call()`.

Here's an example of how you would programatically create select inputs for each categorical column of the `starwars` dataset from `dplyr` and add them to a `box()`.

```r
# get the categorical columns
col_is_char <- sapply(starwars, is.character)
categorical_cols <- names(starwars)[col_is_char]

# set up the selectInputs
select_inputs <- lapply(categorical_cols, function(col) {
  unique_vals <- unique(starwars[[col]])

  selectInput(inputId = col, label = col, choices = unique_vals)
})

# add container arguments to select_inputs
select_inputs$title = "Select the Categories"
select_inputs$solidHeader = TRUE
select_inputs$width = 4

# add to container
select_box <- do.call(box, select_inputs)
```

8.3   Exercises

8.3.1   server functions

In the app you're developing, see if there are any long functions inside reactive functions in `server()` that can be moved to `scripts/func.R` or another external file.

8.3.2   UI files

In the app you're developing, move each tab into an external file and source it into `app.R`.

Chapter 9

Debugging and error handling

Bugs are a part of coding. A great coder isn't someone who writes bug-free code on the first try (this is an unachievable goal), but rather someone who knows how to efficiently catch bugs. This sections presents a few simple ways to debug your Shiny app. See the article Debugging Shiny applications for further debugging strategies, including breakpoints and reactlog.

9.1 RStudio Console Messages

Sending messages to the console is a simple way to debug your code.

I like to keep track of what functions are being called by starting every function inside the server function with a message. The template includes a custom message logging function that helps you use this with both development and deployed apps: `debug_msg()`.

```r
# display debugging messages in R if local,
# or in the console log if remote
debug_msg <- function(...) {
  is_local <- Sys.getenv('SHINY_PORT') == ""
  txt <- toString(list(...))
  if (is_local) {
    message(txt)
  } else {
    shinyjs::runjs(sprintf("console.debug(\"%s\")", txt))
  }
}
```

For example, the code below prints "questionnaire submitted" every time the action button `q_submit` is pressed. It prints to the RStudio console when you're developing and to the javascript console for deployed apps.

```
observeEvent(input$q_submit, {
  debug_msg("questionnaire submitted")
  # rest of code ...
})
```

9.2   JavaScript Console

I use FireFox Developer Edition when I'm developing web apps, but Chrome also has developer tools. In FireFox, go to `Tools > Browser Tools > Web Developer Tools` (opt-cmd-I). In Chrome, go to `View > Developer > Developer Tools` (opt-cmd-I). You can dock the tools to the bottom, right , or left of the window, or as a separate window.
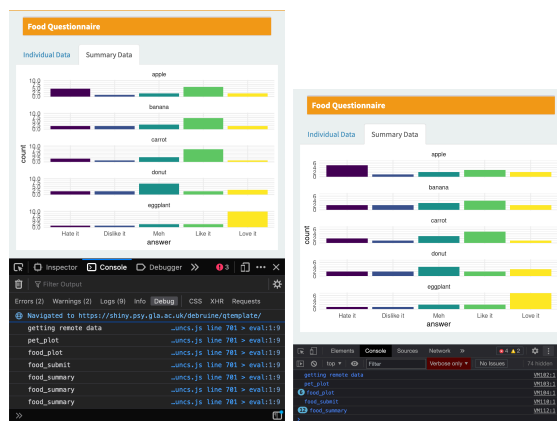
Figure 9.1: Javascript consoles in FireFox Developer Edition and Chrome.

Shiny puts a lot of info you won't care about into the logs, so `debug_msg()` writes messages to the debug console. You can filter just those messages by choosing only "Debug" in FireFox or "Verbose" in Chrome.

9.3   Showcase Mode

You can view an app in showcase mode by setting "DisplayMode" to "Showcase" (instead of "Normal") in the DESCRIPTION file in the app directory. When you're in this mode, you can see your app code, css files, and

javascript files. The functions in your server function will highlight in yellow each time they are run. However, this isn't much help if many of your functions are in external files or you are using modules. Also, if your script is very long, you won't be able to see the highlighting unless you've scrolled to the right section, so I find it more straightforward to use the message method described above.

```
Title: Questionnaire Template
Author: Lisa DeBruine
License: CC-BY-4.0
DisplayMode: Showcase
Type: Shiny
```
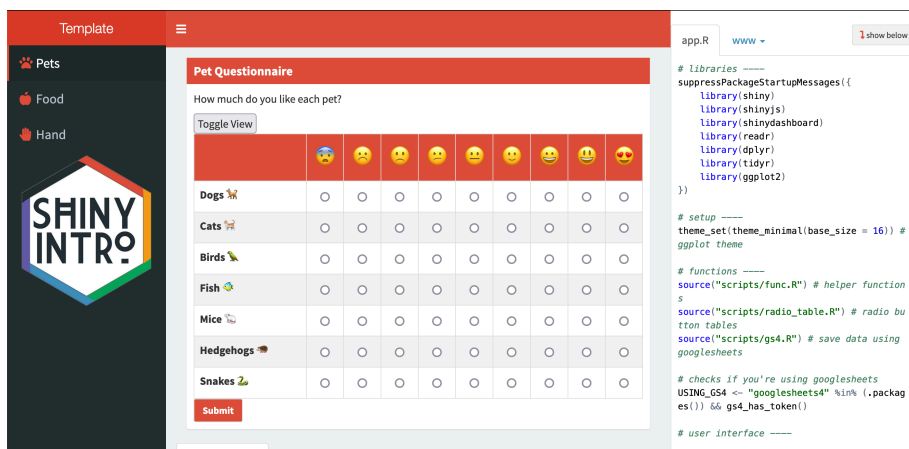

Figure 9.2: Showcase mode.

9.4 tryCatch

You've probably experienced the greyed out screen of a crashed app more than enough now. In development, the next step is to look at the console to see if you have a warning or error message. If you're lucky, you can figure out where in the code this is happening (this is easier if you start all your functions with a debug message).

However, sometimes there are errors that are difficult to prevent. For example, you can try to restrict inputs so the users only enter numeric values using `numericInput()`, but some browsers will let you enter text values anyways. To avoid crashing the whole app, you can wrap potentially error-triggering code in `tryCatch()`.

For example, the code below will cause an error because you can't add a number and a letter.

```r
input <- list(n1 = 10, n2 = "A")

sum <- input$n1 + input$n2
```

```
## Error in input$n1 + input$n2: non-numeric argument to binary operator
```

The following code tries to run the code inside the curly brackets (`{}`), but if it creates an error, the error function will run. The object `e` is the error object, and you can print the message from it using `debug_msg()` (this won't crash the app).

```r
sum <- tryCatch({
  input$n1 + input$n2
}, error = function(e) {
  debug_msg(e$message)
  return(0)
})
```

The return value from the error message is the value assigned to `sum` if there is an error. Sometimes it won't make sense to have a default value, or the code you're checking doesn't have a return value. In that case, you can just put all the code inside the brackets and not return anything from the error function.

```r
tryCatch({
  sum <- input$n1 + input$n2
  output$sum <- renderText(sum)
}, error = function(e) {
  debug_msg(e$message)
})
```

9.5 Input Checking

The user above might be frustrated if they've made a mistake that causes an error and don't know what it was. You can help prevent errors and make the experience of using your app nicer by doing input checking and sending your users useful messages.

```r
observeEvent(input$submit, {
  # check inputs
```

```r
  input_error <- dplyr::case_when(
    !is.numeric(input$n1) ~ "N1 needs to be a number",
    !is.numeric(input$n2) ~ "N2 needs to be a number",
    TRUE ~ ""
  )
  if (input_error != "") {
    shinyjs::alert(input_error)
    return() # exit the function here
  }

  # no input errors
  sum <- input$n1 + input$n2
  output$sum <- renderText(sum)
})
```

Chapter 10

Contingent Display

\begin{figure}

Questions

Have you ever had a pet?

<div style="border:1px solid #ccc;">   ▾</div>

Notice

Please pay attention to this text.

Notice Me	Ignore Me	Toggle Me

Data

Choose a dataset

<div style="border:1px solid #ccc;">mtcars   ▾</div>

Select the columns to show

☑ mpg ☑ cyl ☑ disp ☑ hp ☑ drat
☑ wt ☑ qsec ☑ vs ☑ am ☑ gear
☑ carb

mpg	cyl	disp	hp	drat	wt
21.00	6.00	160.00	110.00	3.90	2.62
21.00	6.00	160.00	110.00	3.90	2.88
22.80	4.00	108.00	93.00	3.85	2.32
21.40	6.00	258.00	110.00	3.08	3.21
18.70	8.00	360.00	175.00	3.15	3.44
18.10	6.00	225.00	105.00	2.76	3.46
14.30	8.00	360.00	245.00	3.21	3.57
24.40	4.00	146.70	62.00	3.69	3.19
22.80	4.00	140.80	95.00	3.92	3.15
19.20	6.00	167.60	123.00	3.92	3.44
17.80	6.00	167.60	123.00	3.92	3.44
16.40	8.00	275.80	180.00	3.07	4.07
17.30	8.00	275.80	180.00	3.07	3.73
15.20	8.00	275.80	180.00	3.07	3.78
10.40	8.00	472.00	205.00	2.93	5.25
10.40	8.00	460.00	215.00	3.00	5.42
14.70	8.00	440.00	230.00	3.23	5.34
32.40	4.00	78.70	66.00	4.08	2.20
30.40	4.00	75.70	52.00	4.93	1.61
33.90	4.00	71.10	65.00	4.22	1.83
21.50	4.00	120.10	97.00	3.70	2.46
15.50	8.00	318.00	150.00	2.76	3.52
15.20	8.00	304.00	150.00	3.15	3.44
13.30	8.00	350.00	245.00	3.73	3.84
19.20	8.00	400.00	175.00	3.08	3.85
27.30	4.00	79.00	66.00	4.08	1.94
26.00	4.00	120.30	91.00	4.43	2.14
30.40	4.00	95.10	113.00	3.77	1.51
15.80	8.00	351.00	264.00	4.22	3.17
19.70	6.00	145.00	175.00	3.62	2.77
15.00	8.00	301.00	335.00	3.54	3.57
21.40	4.00	121.00	109.00	4.11	2.78

10.1 Hide and Show

I frequently want to make some aspect of a shiny app contingent on the state of another aspect, such as only showing a text input of the value of a select input is "other". You can use the `hide()` and `show()` functions from shinyjs to accomplish this easily.

When you set up the UI, wrap any elements that should be hidden at the start in `hidden()`.

```r
# in the ui
box(title = "Questions",
    solidHeader = TRUE,
    selectInput("first_pet", "What was your first pet?",
                c("", "dog", "cat", "ferret", "other")),
    hidden(textInput("first_pet_other", NULL,
                     placeholder = "Specify the other pet"))
)
```

Then set up the hide and show logic in `server()`.

```r
# in the server
observeEvent(input$first_pet, {
    if (input$first_pet == "other") {
        show("first_pet_other")
    } else {
        hide("first_pet_other")
    }
})
```

10.1.1 Groups

Sometimes you need to hide and show a group of elements, depending on something else. You can wrap the grouped elements in a div tag with an id and hide and show that id.

For example, it doesn't make sense to show the questions above to someone who has never had a pet. Add a `selectInput()` before the previous two questions, and then wrap those questions in `tags$div()` with an id of "first_pet_grp"

```r
# replace in ui
box(
  title = "Questions",
  solidHeader = TRUE,
```

```
  selectInput("had_pet", "Have you ever had a pet?", c("", "Yes", "No")),
  hidden(tags$div(
    id = "first_pet_grp",
    selectInput("first_pet", "What was your first pet?",
                c("", "dog", "cat", "ferret", "other")),
    textInput("first_pet_other", NULL,
              placeholder = "Specify the other pet")
  ))
)
```

Then add the following code to the server function to hide or show
first_pet_grp depending on the value of had_pet. The server code above
will take care of whether or not first_pet_other is visible.

```
# add to server
observeEvent(input$had_pet, {
  if (input$had_pet == "Yes") {
    show("first_pet_grp")
  } else {
    hide("first_pet_grp")
  }
})
```

Try to figure out what could go wrong if you didn't wrap "first_pet" and
"first_pet_other" in a group, and instead just hid or showed "first_pet" and
"first_pet_other" depending on the value of has_pet?

10.1.2 Toggle

Sometimes you need to change the visibility of an element when something
happens, rather than specifically hide or show it. You can use toggle() to
hide an element if it's visible and show it if it's hidden.

Add an actionButton() to the sidebar menu (not inside the box) and give
the box an id of "pet_box". Any element that you might want to refer to in
the code needs an id.

```
# add to ui
actionButton("toggle_pet_box", "Toggle Pet Questions")
```

Now, whenever you click the "toggle_pet_box" button, the contents of
"pet_box" will toggle their visibility.

```
# add to server
observeEvent(input$toggle_pet_box, {
  toggle("pet_box")
})
```

What would go wrong if you put the button inside the box?

10.2   Changing Styles

You can use `addClass()`, `removeClass()`, and `toggleClass()` to change element classes. You usually want to do this with classes you've defined yourself.

Add the following style to the `www/custom.css` file.

```
.notice-me {
  color: red;
  text-decoration: underline;
  font-weight: 800;
}
```

And add this box to the ui:

```
box(title = "Notice", solidHeader = TRUE,
    p(id = "notice_text", "Please pay attention to this text."),
    actionButton("add_notice", "Notice Me"),
    actionButton("remove_notice", "Ignore Me"),
    actionButton("toggle_notice", "Toggle Me")
)
```

This code adds the class `notice-me` to the paragraph element "notice_text" whenever the "add_notive" button is pressed.

```
observeEvent(input$add_notice, {
  addClass("notice_text", "notice-me")
})
```

Guess how you would use `removeClass()`, and `toggleClass()` with the buttons set up above.

10.2.1   Changing non-shiny elements

Unfortunately, not all elements on the web page have an ID that can be
altered by `addClass()` or `removeClass()`. For example, the skin of a
shinydashboard app is determined by the css class of the body element.
However, we can use `runjs()` to run any arbitrary JavaScript code.

Add the following action button into the sidebar menu.

```
actionButton("random_skin", "Random Skin")
```

The jQuery code below changes the skin of your app on a button press by
removing all possible skin-color classes and adding a random one.

```
observeEvent(input$random_skin, {
  skins <- c("red", "yellow", "green", "blue", "purple", "black")
  skin_color <- sample(skins, 1)

  js <- sprintf("$('body').removeClass('%s').addClass('skin-%s');",
                paste(paste0("skin-", skins), collapse = " "),
                skin_color)

  shinyjs::runjs(js)
})
```

Changing the skin color with a button press isn't something you'll easily find
documented in online materials. I figured it out through looking at how the
underlying html changed when I changed the skin color in the app code.
Hacks like this require lots of trial and error, but get easier the more you
understand about html, css and JavaScript.

10.3   Changing input options

The relevant options in a `selectInput()` or `radioButton()` may change
depending on the values of other inputs. Sometimes you can accommodate
this by creating multiple versions of a input and hiding or showing. Other
times you may wish to update the input directly.

Add the following box to the ui.

```
box(title = "Data", solidHeader = TRUE, width = 12,
  selectInput("dataset", "Choose a dataset", c("mtcars", "sleep")),
  checkboxGroupInput("columns", "Select the columns to show", inline = TRUE),
  tableOutput("data_table")
)
```

First, set up the code to display the correct data in the table.

```r
mydata <- reactive({
  if (input$dataset == "mtcars") { mtcars } else { sleep }
})

output$data_table <- renderTable(mydata())
```

Now we need to set the options for "columns" depending on which "dataset" is selected.

```r
observe({
  col_names <- names(data())
  debug_msg(col_names)
  updateCheckboxGroupInput(inputId = "columns",
                           choices = col_names,
                           selected = col_names)
})
```

Finally, we can add some code to select only the checked columns to display.

```r
observe({
  full_data <- if (input$dataset == "mtcars") { mtcars } else { sleep }
  col_names <- names(full_data)
  updateCheckboxGroupInput(
    inputId = "columns",
    choices = col_names,
    selected = col_names,
    inline = TRUE
  )
})
```

Why do we have to get the dataset again instead of using the data from `mydata()`?

Finally, alter the reactive function to only show the selected columns.

```r
mydata <- reactive({
  d <- if (input$dataset == "mtcars") { mtcars } else { sleep }
  d[input$columns]
})
```

What happens when you unselect all the columns? How can you fix this?

10.4 Exercises

Chapter 11

Sharing your Apps

## 11.1   shinyapps.io

1. Open **Tools > Global Options ...**
2. Go to the **Publishing** tab
3. Click the **Connect** button and choose ShinyApps.io
4. Click on the link to go to your account
5. Click the **Sign Up** button and **Sign up with GitHub**

6. You should now be in your shinyapps.io dashboard; click on your name in the upper right and choose **Tokens**
7. Add a token
8. Click **Show** next to the token and copy the text to the clipboard

The `shinyapps` package must be authorized to your account using a token and secret. To do this, click the copy button below and we'll copy the whole command you need to your clipboard. Just paste it into your console to authorize your account. Once you've entered the command successfully in R, that computer is now authorized to deploy applications to your shinyapps.io account.

```
rsconnect::setAccountInfo(name='debruine',
                          token='C4DB63458EB3B284968AA65623F8E8FB',
                          secret='<SECRET>')
```

Show secret

🖉 Copy to clipboard

OK

9. Go back to RStudio and paste the text in the box and click **Connect Account**
10. Make sure the box next to "Enable publishing…" is ticked, click **Apply**, and close the options window · You can test this by creating a simple app. If you have the shinyintro package, use the code below.

```
shinyintro::clone("input_demo", "mytestapp")
```


Open the app.R file and go to `File > Publish...` in the menu (or click on the blue icon in the upper right corner of the source pane). Make sure these

are the right files for your app, edit the title if you want, and click **Publish**. A web browser window will open after a few seconds showing your app online! You can now share it with your friends and colleagues.

If publishing fails, check the Console pane. If you already have too many apps on shinyapps.io, you'll see the message, "You have reached the maximum number of applications allowed for your account." You can archive some of your apps from the shinyapps.io dashboard if this is the problem.

11.2   Self-hosting a shiny server

Setting up a shiny server is beyond the scope of this class, but if you have access to one, you can ask the administrator how to access the correct directories and upload your app directories there.

This solution is good if you want to save data locally and do not want to use Google Sheets. You can't save data locally on shinyapps.io.

If you save data locally on a shiny server, you may need to change the owner or permissions of the directory you save data in so that the web user can write to it. Ask the administrator of the server for help if this doesn't make any sense to you.

11.3   GitHub

GitHub is a great place to organise and share your code using version control. You can also use it to host Shiny app code for others to download and run on their own computer.

See Appendix B for instructions on how to set up git and a GitHub account. Set up a github access token with `usethis::create_github_token()`. Your web browser will open and you'll be asked to log into your GitHub account and then asked to authorise a new token. Accept the defaults and click OK at the bottom of the page. In RStudio, run `gitcreds::gitcreds_set()` and follow the instructions to save the token.

Then, to share an app with people who use R, make a project that contains your `app.R` file and any related files. If you aren't already using version control for this project, make sure all of your files are saved and type `usethis::use_git()` into the console. Choose Yes to commit and Yes to restart R.

Make a new GitHub repository with `usethis::use_github(protocol="https")`; check that the suggested title and description are OK. If you choose the affirmative response (not always the

same number), you'll see some messages and your web browser will open the github repository page.

Now you can share your app with others by sending them the repository link. They can access your repository in RStudio by starting a New Project from version control, using the URL that is shown when you click on the green "Code" button on the repository page (something like "https://github.com/account/repository.git"). They can run your app the same way you do when developing it, by opening the `app.R` file and clicking the Run button.

To update your files on GitHub, you need to commit any changes you make using the Git tab in the upper right pane. Click on the checkbox of any files you want to update, click "Commit", and write a message to yourself explaining the changes (this will be publicly viewable on GitHub, so try to be professional ;).

Committing just creates a snapshot of the files on your computer so you can look at previous versions. To update the files on GitHub, you need to push the updates using the green up arrow button.

Git and GitHub can be tricky. Happy Git with R by Jenny Bryan is a fantastic in-depth book anbout how to work with git in R and RStudio.

11.4   In an R package

You can put your app in a custom R package to make it even easier for people to run the app. The usethis package is incredibly helpful for setting up packages.

```
usethis::create_package("mypackagename")
usethis::use_ccby_license()

# add packages your app uses
usethis::use_package("shiny")
usethis::use_package("shinydashboard")

# add the directory for your apps
dir.create("inst")
dir.create("inst/apps")
```

Copy any apps you want to include in this package into the inst/apps directory.

Now, create the app function by running `usethis::edit_file("R/app.R")` and copy the following text into the `app.R` file that just opened. Replace

"default_app" with the directory name of the app that you want to open if a user doesn't type any name in at all.

```r
#' Launch Shiny App
#'
#' @param name The name of the app to run
#' @param ... arguments to pass to shiny::runApp
#'
#' @export
#'
app <- function(name = "default_app", ...) {
  appDir <- system.file(paste0("apps/", name), package = "mypackagename")
  if (appDir == "") stop("The shiny app ", name, " does not exist")
  shiny::runApp(appDir, ...)
}
```

Next, open the `DESCRIPTION` file and edit the title, author and description.

```r
devtools::document()
devtools::install()
```

You should now be able to run your app with `mypackagename::app()`.

Set up git and save your package to GitHub to share it with others:

```r
usethis::use_git()
usethis::use_github(protocol="https")
```

Once it's uploaded to GitHub, other people can install it with the following code:

```r
devtools::install_github("myaccountname/mypackagename")
```

11.5 Exercises

11.5.1 Shinyapps.io

- Upload another demo app to shinyapps.io
- Check that you can access it online
- Archive the app in the shinyapps.io dashboard

11.5.2   GitHub

- Set up a GitHub account
- Create a New Project in RStudio from version control using https://github.com/debruine/demoapp
- Create your own project that contains an app you're working on and upload it to GitHub.
- Send a friend (or your instructor) directions to access and run it.

11.5.3   R Package

- Create an R package for your app and upload it to GitHub.
- Send a friend (or your instructor) directions to access and run it.

Chapter 12

Customized reports

While the best part of Shiny apps is their interactivity, sometimes your users need to download data, images, or a static report. This section will show you how.

\begin{figure}

Downloads

⬇ Pets Data ()   ⬇ Food Data ()

⬇ Pets Plot ()   ⬇ Food Plot ()

Plot Width (inches)

| 7 | |

Plot Height (inches)

| 5 | |

⬇ Pets Report ()   ⬇ Food Report ()

{

12.1   Download Data

First, we need to add the appropriate UI to our questionnaire app. Create a
new tab called "report_tab" with two `downloadButton()`s, one for the pets
data and one for the food data.

```
report_tab <- tabItem(
  tabName = "report_tab",
  box(
    id = "download_box",
    title = "Downloads",
    solidHeader = TRUE,
    downloadButton("pet_data_dl", "Pets Data"),
    downloadButton("food_data_dl", "Food Data")
  )
)
```

Remember to add this to `tabItems()` in `dashboardBody()` and also add a
corresponding `menuItem()` in `sidebarMenu()`. Run the app to make sure the
UI looks like you expect before you proceed.

Now we need to add code to `server()` to handle the downloads.
`downloadButton()` is a special type of output that is handled by
`downloadHandler()`. This function takes two arguments, a function to create
the filename and a function to create the content.

```
### pet_data_dl ----
output$pet_data_dl <- downloadHandler(
  filename = function() {
    debug_msg("pet_data_dl")
    paste0("pet-data_", Sys.Date(), ".csv")
  },
  content = function(file) {
    gs4_write_csv(v$pet_summary_data, file)
  }
)
```

Instead of `write.csv()` or `readr::write_csv()`, here we're using
`gs4_write_csv()` (defined in `scripts/gs4.R`) because googlesheets can
return list columns that cause errors when saving to CSV without some
preprocessing.

12.2 Download Images

Now that we need to use the summary plot in more than one place, it doesn't make sense to build it twice. Move all of the code from the `renderPlot()` for `output$pet_summary` into a `reactive()` called `pet_summary_plot`. Then we can build the plot whenever the inputs change and refer to it anywhere as `pet_summary_plot()`.

```r
### pet_summary_plot ----
pet_summary_plot <- reactive({ debug_msg("pet_summary_plot")
  # code from output$pet_summary ...
})


### pet_summary ----
output$pet_summary <- renderPlot({ debug_msg("pet_summary")
    pet_summary_plot()
})
```

The `downloadHandler()` works the same as for downloading a CSV file. You can use `ggsave()` to write the plot.

```r
# pet_plot_dl ----
output$pet_plot_dl <- downloadHandler(
  filename = function() {
    paste0("pet-plot_", Sys.Date(), ".png")
  },
  content = function(file) {
    ggsave(file,
           pet_summary_plot(),
           width = 7,
           height = 5)
  }
)
```

Add numeric inputs to the UI to let the user specify the downloaded plot width and height.

Solution

```r
# add to report_tab ui
numericInput("plot_width", "Plot Width (inches)", 7, min = 1, max = 10)
numericInput("plot_height", "Plot Height (inches)", 5, min = 1, max = 10)

### pet_plot_dl ----
output$pet_plot_dl <- downloadHandler(
```

```r
  filename = function() {
    debug_msg("pet_plot_dl")
    paste0("pet-plot_", Sys.Date(), ".png")
  },
  content = function(file) {
    ggsave(
      file,
      pet_summary_plot(),
      width = input$plot_width,
      height = input$plot_height
    )
  }
)
```

12.3   R Markdown

You can render an R Markdown report for users to download.

First, you need to save an R Markdown file (save the one below as
`reports/pet_report.Rmd`). The file will have access to any objects that are
available in `server()`, so you don't have to re-download the summary data
and re-create the plot.

```
---
title: "Pet Report"
date: "16 July, 2021"
output: html_document
---


```r
n_responses <- nrow(v$pet_summary_data)

n_sessions <- v$pet_summary_data$session_id %>%
  unique() %>%
  length()
```

We asked people to rate how much they like pets. We have obtained `r n_responses` resp

Summary Plot

```r
pet_summary_plot()
```
```

The `content` function uses `rmarkdown::render()` to create the output file
from Rmd. We are using `html_document` as the output type, but you can also
render as a PDF or Word document if you have pandoc or latex installed.

```
### pet_report_dl ----
output$pet_report_dl <- downloadHandler(
  filename = function() {
    debug_msg("pet_report_dl")
    paste0("pet-report_", Sys.Date(), ".html")
  },
  content = function(file) {
    rmarkdown::render("reports/pet_report.Rmd",
                      output_file = file,
                      intermediates_dir = tempdir())
  }
)
```

On your own computer, you don't need to set `intermediates_dir =`
`tempdir()`, but you will need to do this if you want to deploy your app on a
shiny server. When rmarkdown renders an Rmd file, it creates several
intermediate files in the working directory and then deletes them. you have
permission to write to the app's working directory on your own computer, but
might not on a shiny server. `tempdir()` is almost always a safe place to write
temporary files to.

12.4 Exercises

12.4.1 Food Data

Create a button that downloads the food data.

Solution

```
# add to report_tab ui ----
downloadButton("food_data_dl", "Food Data")

### food_data_dl ----
output$food_data_dl <- downloadHandler(
  filename = function() {
    debug_msg("food_data_dl")
    paste0("food-data_", Sys.Date(), ".csv")
  },
  content = function(file) {
    gs4_write_csv(v$food_summary_data, file)
```

```
  }
)
```

12.4.2   Food Image

Create a button that downloads the food summary plot

Solution

```
# add to report_tab ui ----
downloadButton("food_plot_dl", "Food Plot")

### food_summary_plot ----
food_summary_plot <- reactive({ debug_msg("food_summary_plot")
  # code from output$food_summary ...
})

### food_summary ----
output$food_summary <- renderPlot({ debug_msg("food_summary")
  food_summary_plot()
})

### food_plot_dl ----
output$food_plot_dl <- downloadHandler(
  filename = function() {
    debug_msg("food_plot_dl")
    paste0("food-plot_", Sys.Date(), ".png")
  },
  content = function(file) {
    ggsave(
      file,
      food_summary_plot(),
      width = input$plot_width,
      height = input$plot_height
    )
  }
)
```

12.4.3   Food Report

Create a downloadable report for the food data.

Solution

Don't forget to create an Rmd file with the contents of your food report.

```r
# add to report_tab ui ----
downloadButton("food_report_dl", "Food Report")

### food_report_dl ----
output$food_report_dl <- downloadHandler(
  filename = function() {
    debug_msg("food_report_dl")
    paste0("food-report_", Sys.Date(), ".html")
  },
  content = function(file) {
    rmarkdown::render("reports/food_report.Rmd",
                      output_file = file,
                      intermediates_dir = tempdir())
  }
)
```

Chapter 13

Shiny modules for repeated structures

If you find yourself making nearly identical UIs or server functions over and over in the same app, you might benefit from modules. This is a way to define a pattern to use repeatedly.

\begin{figure}

| iris | mtcars |

DV

| Sepal.Length | ▼ |

Show 10 ▼ entries   Search: [   ]

Sepal.Length   Processing**Sepal.Width**   **Petal.Length**

{
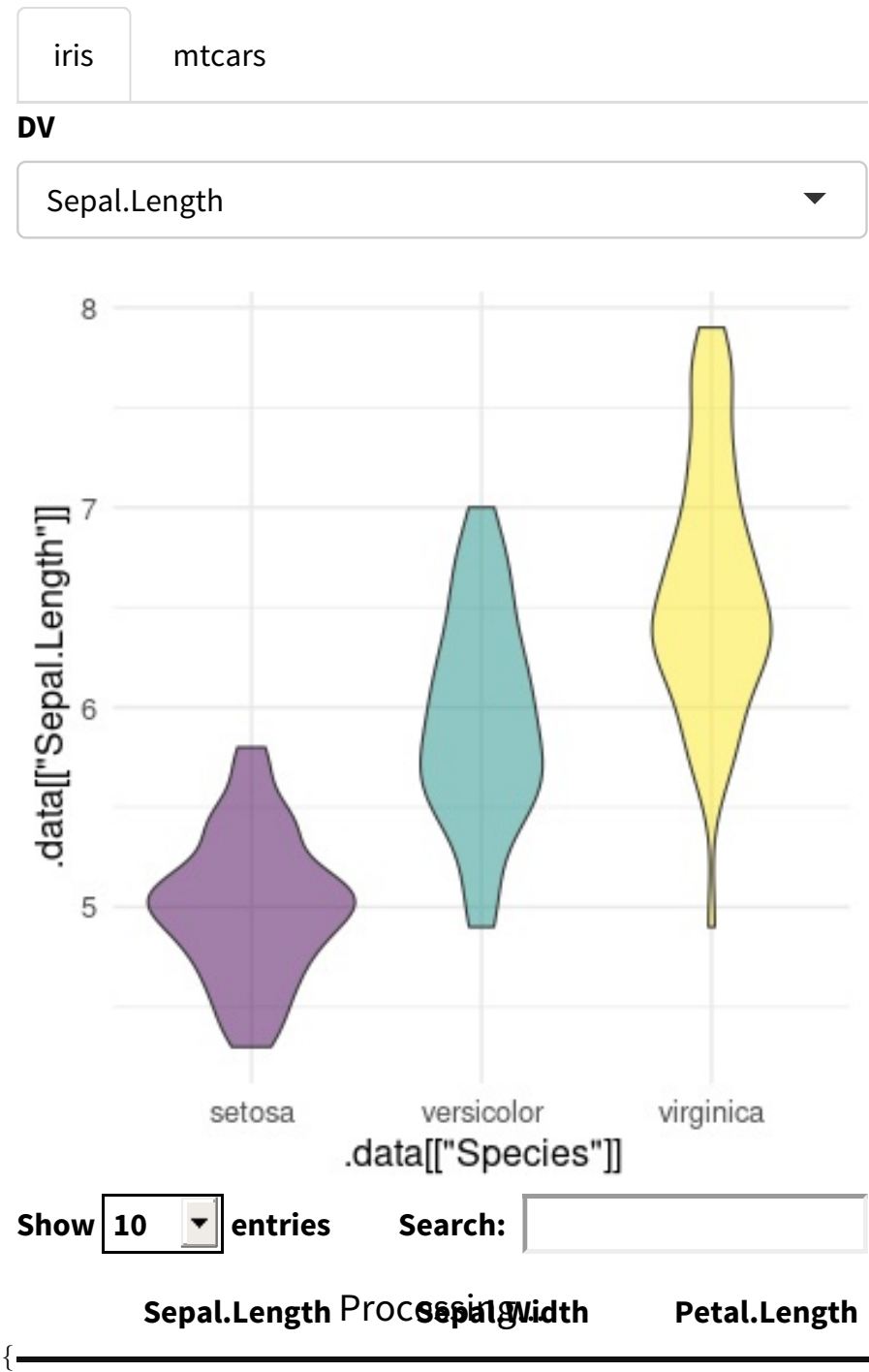
Run locally with `shinyintro::app("modules_demo")` or view it in a separate tab with the showcase interface.

13.1   Modularizing the UI

The two tabPanels below follow nearly identical patterns. You can often identify a place where modules might be useful when you use a naming convention like {base}_{type} for the ids.

```
iris_tab <- tabPanel(
  "iris",
  selectInput("iris_dv", "DV", choices = names(iris)[1:4]),
  plotOutput("iris_plot"),
  DT::dataTableOutput("iris_table")
)

mtcars_tab <- tabPanel(
  "mtcars",
  selectInput("mtcars_dv", "DV", choices = c("mpg", "disp", "hp", "drat")),
  plotOutput("mtcars_plot"),
  DT::dataTableOutput("mtcars_table")
)
```

The first step in modularising your code is to make a function that creates the UIs above from the base ID and any other changing aspects. In the example above, the choices are different for each selectInput, so we'll make a function that has the arguments `id` and `choices`.

The first line of a UI module function is always `ns <- NS(id)`, which creates a shorthand way to add the base id to the id type. So instead of the selectInput's name being "iris_dv" or "mtcars_dv", we set it as `ns("dv")`. All ids need to use `ns()` to add the namespace to thier ID.

```
tabPanelUI <- function(id, choices) {
    ns <- NS(id)

    tabPanel(
        id,
        selectInput(ns("dv"), "DV", choices = choices),
        plotOutput(ns("plot")),
        DT::dataTableOutput(ns("table"))
    )
}
```

Now, you can replace two tabPanel definitions with just the following code.

```r
iris_tab <- tabPanelUI("iris", names(iris)[1:4])
mtcars_tab <- tabPanelUI("mtcars", c("mpg", "disp", "hp", "drat"))
```

13.2   Modularizing server functions

In our original code, we have four functions that create the two output tables and two output plots, but these are also largely redundant.

```r
output$iris_table <- DT::renderDataTable({
    iris
})

output$iris_plot <- renderPlot({
    ggplot(iris, aes(x = Species,
                     y = .data[[input$iris_dv]],
                     fill = Species)) +
        geom_violin(alpha = 0.5, show.legend = FALSE) +
        scale_fill_viridis_d()
})

output$mtcars_table <- DT::renderDataTable({
    mtcars
})

output$mtcars_plot <- renderPlot({
    # handle non-string grouping
    mtcars$vs <- factor(mtcars$vs)
    ggplot(mtcars, aes(x = vs,
                       y = .data[[input$mtcars_dv]],
                       fill = vs)) +
        geom_violin(alpha = 0.5, show.legend = FALSE) +
        scale_fill_viridis_d()
})
```

The second step to modularising code is creating a server function. You can put all the functions the relate to the inputs and outputs in the UI function here, so we will include one to make the output table and one to make the output plot.

The server function takes the base id as the first argument, and then any arguments you need to specify things that change between base implementations. Above, the tables show different data and the plots use different groupings for the x axis and fill, so we'll add arguments for `data` and `group_by`.

A server function **always** contains `moduleServer()` set up like below.

```
tabPanelServer <- function(id, data, group_by) {
    moduleServer(id, function(input, output, session) {
      # code ...
    })
}
```

No you can copy in one set of server functions above, remove the base name (e.g., "iris_" or "mtcars_") from and inputs or outputs, and replace specific instances of the data or grouping columns with `data` and `group_by`.

```
tabPanelServer <- function(id, data, group_by) {
    moduleServer(id, function(input, output, session) {
        output$table <- DT::renderDataTable({
            data
        })

        output$plot <- renderPlot({
            # handle non-string groupings
            data[[group_by]] <- factor(data[[group_by]])
            ggplot(data, aes(x = .data[[group_by]],
                             y = .data[[input$dv]],
                             fill = .data[[group_by]])) +
                geom_violin(alpha = 0.5, show.legend = FALSE) +
                scale_fill_viridis_d()
        })
    })
}
```

In the original code, the grouping variables were unquoted, but it's tricky to pass unquoted variable names to custom functions, and we already know how to refer to columns by a character object using `.data[[char_obj]]`.

The grouping column `Species` in `iris` is already a factor, but recasting it as a factor won't hurt, and is required for the `mtcars` grouping column `vs`.

Now, you can replace the four functions inside the server function with these two lines of code.

```
tabPanelServer("iris", data = iris, group_by = "Species")
tabPanelServer("mtcars", data = mtcars, group_by = "vs")
```

Our example only reduced our code by 4 lines, but it can save a lot of time, effort, and debugging on projects with many similar modules. For example, if you want to change the plots in your app to use a different geom, now you only have to change one function instead of two.

13.3   Exercises

13.3.1   1. Repeat Example

Try to implement the code above on your own.

- Clone "no_modules_demo" `shinyintro::clone("no_modules_demo")`
- Run the app and see how it works
- Create the UI module function and use it to replace `iris_tab` and `mtcars_tab`
- Create the server function and use it to replace the server functions

13.3.2   2. New Instance

Add a new tab called "diamonds" that visualises the `diamonds` dataset. Choose the columns you want as choices in the `selectInput()` and the grouping column.

UI Solution

You can choose any of the numeric columns for the choices.

```
diamonds_tab <- tabPanelUI("diamonds", c("carat", "depth", "table", "price"))
```

Server Solution

You can group by any of the categorical columns: cut, color, or clarity.

```
tabPanelServer("diamonds", data = diamonds, group_by = "cut")
```

13.3.3   3. Altering modules

- Add another `selectInput()` to the UI that allows the user to select the grouping variable. (`iris` only has one possibility, but `mtcars` and `diamonds` should have several)

UI Solution

You need to add a new selectInput() to the tabPanel. Remember to use `ns()` for the id. The choices for this select will also differ by data set, so you need to add `group_choices` to the arguments of this function.

```r
tabPanelUI <- function(id, choices, group_choices) {
   ns <- NS(id)

   tabPanel(
       id,
       selectInput(ns("dv"), "DV", choices = choices),
       selectInput(ns("group_by"), "Group By", choices = group_choices),
       plotOutput(ns("plot")),
       DT::dataTableOutput(ns("table"))
   )
}
```

- Update the plot function to use the value of this new input instead of "Species", "vs", and whatever you chose for `diamonds`.

Server Solution

You no longer need `group_by` in the arguments for this function because you are getting that info from an input.

Instead of changing `group_by` to `input$group_by` in three places in the code below, I just added the line `group_by <- input$group_by` at the top of `moduleServer()`.

```r
tabPanelServer <- function(id, data) {
    moduleServer(id, function(input, output, session) {
        group_by <- input$group_by

        # rest of the code is the same ...
    })
}
```

13.3.4   4. New module

There is a fluidRow() before the tabsetPanel() in the ui that contains three `infoBoxOutput()` and three renderInfoBoxOutput() functions in the server function.

Modularise the info boxes and their associated server functions.

UI Function

```r
infoBoxUI <- function(id, width = 4) {
    ns <- NS(id)
```

```
    infoBoxOutput(ns("box"), width)
}
```

Server Function

```r
infoBoxServer <- function(id, title, fmt, icon, color = "purple") {
    moduleServer(id, function(input, output, session) {
        output$box <- renderInfoBox({
            infoBox(title = title,
                    value = format(Sys.Date(), fmt),
                    icon = icon(icon),
                    color = color)
        })
    })
}
```

UI Code

In the `ui`, replace the `fluidRow()` with this:

```r
fluidRow(
    infoBoxUI("day"),
    infoBoxUI("month"),
    infoBoxUI("year")
)
```

Server Code

In `server()`, replace `renderInfoBox()` with this:

```r
infoBoxServer("year", "Year", "%Y", "calendar")
infoBoxServer("month", "Month", "%m", "calendar-alt")
infoBoxServer("day", "Day", "%d", "calendar-day"))
```

Appendix A

Installing `R`

Installing R and RStudio is usually straightforward. The sections below explain how and there is a helpful YouTube video here.

A.1 Installing Base R

Install base R from https://cran.rstudio.com/. Choose the download link for your operating system (Linux, Mac OS X, or Windows).

If you have a Mac, install the latest release from the newest `R-x.x.x.pkg` link (or a legacy version if you have an older operating system). After you install R, you should also install XQuartz to be able to use some visualisation packages.

If you are installing the Windows version, choose the "base" subdirectory and click on the download link at the top of the page. After you install R, you should also install RTools; use the "recommended" version highlighted near the top of the list.

If you are using Linux, choose your specific operating system and follow the installation instructions.

A.2 Installing RStudio

Go to rstudio.com and download the RStudio Desktop (Open Source License) version for your operating system under the list titled **Installers for Supported Platforms**.

A.3   Installing shinyintro

To install the class package, which will provide you with a copy of all of the
shiny apps we'll use for demos and the basic template, paste the following code
into the console in RStudio.

```r
# you may have to install devtools first with
# install.packages("devtools")

devtools::install_github("debruine/shinyintro")
```

A.4   Installing LaTeX

You can install the LaTeX typesetting system to produce PDF reports from
RStudio. Without this additional installation, you will be able to produce
reports in HTML but not PDF. This course will not require you to make
PDFs. To generate PDF reports, you will additionally need:

1. pandoc, and
2. LaTeX, a typesetting language, available for

   - WINDOWS: MikTeX
   - Mac OS: MacTex (3.2GB download) or BasicTeX (78MB download,
     but should work fine)
   - Linux: TeX Live

Appendix B

Symbols

Symbol	Book Term	Also Known As
()	(round) brackets	parentheses
[]	square brackets	brackets
{}	curly brackets	squiggly brackets
<>	chevrons	angled brackets / guillemets
<	less than	
>	greater than	
&	ampersand	"and" symbol
#	hash	pound / octothorpe
/	slash	forward slash
\	backslash	
-	dash	hyphen / minus
_	underscore	
*	asterisk	star
^	caret	power symbol
~	tilde	twiddle / squiggle
=	equal sign	
==	double equal sign	
.	full stop	period / point
!	exclamation mark	bang / not
?	question mark	
'	single quote	quote / apostrophe
"	double quote	quote
%>%	pipe	magrittr pipe
|	vertical bar	pipe
,	comma	
;	semi-colon	
:	colon	

Symbol	Book Term	Also Known As
@	"at" symbol	various hilarious regional terms

Appendix C

Glossary

See the psyTeachR Glossary for more definitions of R jargon.

term	definition
argument	A variable that provides input to a function.
attribute-html	Extra information about an HTML element
cache-web	In a web browser, external files like CSS, JavaScript, and images are usually cach
commit	The action of storing a new snapshot of a project's state in the git history.
console	The pane in RStudio where you can type in commands and view output messages
css	Cascading Style Sheet: A system for controlling the visual presentation of HTML
data-wrangling	The process of preparing data for visualisation and statistical analysis.
directory	A collection or "folder" of files on a computer.
dynamic	Something that can change in response to user actions
element-html	A unit of HTML, such as a header, paragraph, or image.
function	A named section of code that can be reused.
github	A cloud-based service for storing and sharing your version controlled files.
html	Hyper-Text Markup Language: A system for semantically tagging structure and i
integer	A data type representing whole numbers.
javascript	An object-oriented computer programming language commonly used to create int
jquery	A library that makes it easier to write JavaScript.
latex	A typesetting program needed to create PDF files from R Markdown documents.
list	A container data type that allows items with different data types to be grouped t
object	A word that identifies and stores the value of some data for later use.
package	A group of R functions.
pandoc	A universal document convertor, used by R to make PDF or Word documents fro
panes	RStudio is arranged with four window "panes."
push	Updating the remote git project from the local project.
r-markdown	The R-specific version of markdown: a way to specify formatting, such as headers
rstudio	An integrated development environment (IDE) that helps you process R code.
server	This is the part of a Shiny app that works with logic.
shiny	An R package that builds interactive web apps
static	Something that does not change in response to user actions
string	A piece of text inside of quotes.
tag	A way to mark the start and end of HTML elements.
tidyverse	A set of R packages that help you create and work with tidy data
ui	The User Interface. This usually refers to a Shiny App as the user will see it.
vector	A type of data structure that is basically a list of things like T/F values, numbers
version-control	A way to save a record of changes to your files.
widget	A interactive web element, like a dropdown menu or a slider.
working-directory	The filepath where R is currently reading and writing files.