Mastering Shiny Solutions

The R4DS Online Learning Community

(updated on 2023-02-08)

Contents

Welcome

This book offers solutions to the exercises from Hadley Wickham's book Mastering Shiny. It is a work in progress and under active development. The code for this book can be found on GitHub. Your PRs and suggestions are very welcome.

Note Boxes

We use three different types of boxes throughout the book.

This "note" box denotes we were either unclear what the exercise was asking or have follow up questions.

This "TODO" box denotes that the exercises in the chapter have either partial solutions or have yet to be reviewed.

License

Acknowledgements

Chapter 1

Preface

There are no exercises in this chapter.

Chapter 2

Your First Shiny App

Exercise 2.8.1

Create an app that greets the user by name. You don't know all the functions you need to do this yet, so I've included some lines of code below. Figure out which lines you'll use and then copy and paste them into the right place in a Shiny app.

```
textInput("name", "What's your name?")
renderText({
  paste0("Hello ", input$name)
})
numericInput("age", "How old are you?")
textOutput("greeting")
tableOutput("mortgage")
renderPlot("histogram", {
  hist(rnorm(1000))
}, res = 96)
```

Solution.

Solution

In the UI, we will need a `textInput` for the user to input text, and a `textOutput` to output any custom text to the app. The corresponding server function to `textOutput` is `renderText`, which we can use to compose the output element we've named "greeting".

```
library(shiny)

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

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

shinyApp(ui, server)
```

Exercise 2.8.2

Suppose your friend wants to design an app that allows the user to set a number (`x`) between 1 and 50, and displays the result of multiplying this number by 5. This is their first attempt:

```
ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
  "then x times 5 is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    x * 5
  })
}
```

But unfortunately it has an error:

Can you help them find and correct the error?

Solution.

Solution

The error here arises because on the server side we need to write `input$x` rather than `x`. By writing `x`, we are looking for element `x` which doesn't exist in the Shiny environment; `x` only exists within the read-only object `input`.

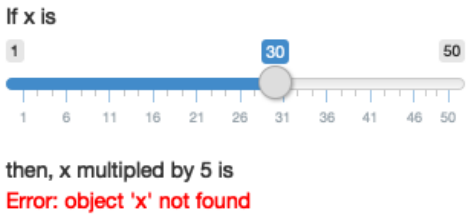
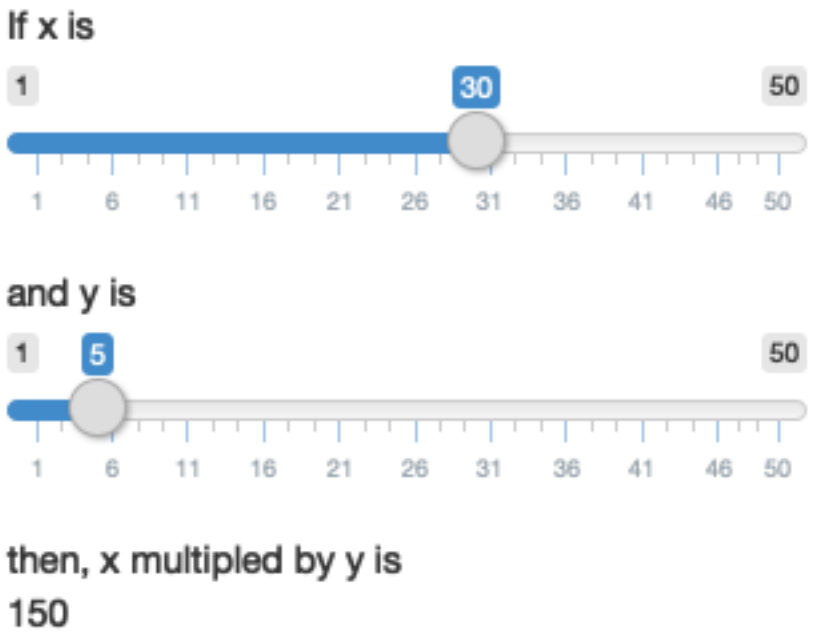


Figure 2.1: placeholder

```
library(shiny)

ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
  "then x times 5 is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    input$x * 5
  })
}

shinyApp(ui, server)
```

Exercise 2.8.3

Extend the app from the previous exercise to allow the user to set the value of the multiplier, `y`, so that the app yields the value of `x * y`. The final result should look like this:



Solution.

Solution

Let us add another `sliderInput` with ID `y`, and use both `input$x` and `input$y` to calculate `output$product`.

```
library(shiny)

ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
  sliderInput("y", label = "and y is", min = 1, max = 50, value = 30),
  "then x multiplied by y is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    input$x * input$y
  })
}

shinyApp(ui, server)
```

Exercise 2.8.4

Replace the UI and server components of your app from the previous exercise with the UI and server components below, run the app, and describe the app's functionality. Then reduce the duplication in the app by using a reactive expression.

```
ui <- fluidPage(
  sliderInput("x", "If x is", min = 1, max = 50, value = 30),
  sliderInput("y", "and y is", min = 1, max = 50, value = 5),
  "then, (x * y) is", textOutput("product"),
  "and, (x * y) + 5 is", textOutput("product_plus5"),
  "and (x * y) + 10 is", textOutput("product_plus10")
)

server <- function(input, output, session) {
  output$product <- renderText({
    product <- input$x * input$y
    product
  })
  output$product_plus5 <- renderText({
    product <- input$x * input$y
    product + 5
  })
  output$product_plus10 <- renderText({
    product <- input$x * input$y
    product + 10
  })
}
```

Solution.

Solution

The application above has two numeric inputs `input$x` and `input$y`. It computes three values: `x*y`, `x*y + 5`, and `x*y + 10`. We can reduce duplication by making the `product` variable a reactive value and using it within all three outputs.

```
library(shiny)

ui <- fluidPage(
  sliderInput("x", "If x is", min = 1, max = 50, value = 30),
  sliderInput("y", "and y is", min = 1, max = 50, value = 5),
  "then, (x * y) is", textOutput("product"),
```

```
  "and, (x * y) + 5 is", textOutput("product_plus5"),
  "and (x * y) + 10 is", textOutput("product_plus10")
)

server <- function(input, output, session) {

  product <- reactive(input$x * input$y)

  output$product <- renderText( product() )
  output$product_plus5 <- renderText( product() + 5 )
  output$product_plus10 <- renderText( product() + 10 )
}
shinyApp(ui, server)
```

Exercise 2.8.5

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

```
library(shiny)
library(ggplot2)

datasets <- c("economics", "faithfuld", "seals")

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

server <- function(input, output, session) {
  dataset <- reactive({
    get(input$dataset, "package:ggplot2")
  })
  output$summmry <- renderPrint({
    summary(dataset())
  })
  output$plot <- renderPlot({
    plot(dataset)
  }, res = 96)
```

```
}
```

Solution.

Solution

The app contains the following three bugs:

1. In the UI, the `tableOutput` object should really be a `plotOutput`.
2. In the server, the word "summry" in `output$summry` is misspelled.

- **NB: In the printed 1st edition (Exercise 1.5), the deliberate typo `summry` in the `server` was corrected and there are only 2 two bugs.**

3. In the server, the `plot` function in the `output$plot` should call `dataset()` rather than the reactive object.

The fixed app looks as follows:

```
library(shiny)
library(ggplot2)

datasets <- c("economics", "faithfuld", "seals")

ui <- fluidPage(
  selectInput("dataset", "Dataset", choices = datasets),
  verbatimTextOutput("summary"),
  # 1. Change tableOutput to plotOutput.
  plotOutput("plot")
)

server <- function(input, output, session) {
  dataset <- reactive({
    get(input$dataset, "package:ggplot2")
  })
  # 2. Change summry to summary.
  output$summary <- renderPrint({
    summary(dataset())
  })
  output$plot <- renderPlot({
    # 3. Change dataset to dataset().
    plot(dataset())
```

```
  })
}

shinyApp(ui, server)
```

Chapter 3

Basic UI

Exercise 3.2.8.1

When space is at a premium, it's useful to label text boxes using a placeholder that appears *inside* the text entry area. How do you call `textInput()` to generate the UI below?

Figure 3.1: placeholder

Solution.

Solution

Looking at the output of `?textInput`, we see the argument `placeholder` which takes:

> A character string giving the user a hint as to what can be entered into the control.

Therefore, we can use the `textInput` with arguments as shown below to generate the desired UI.

```
textInput("text", "", placeholder = "Your name")
```

Exercise 3.2.8.2

Carefully read the documentation for `sliderInput()` to figure out how to create a date slider, as shown below.

Figure 3.2: placeholder

Solution.

Solution

To create such slider, we need the following code.

```
sliderInput(
  "dates",
  "When should we deliver?",
  min = as.Date("2019-08-09"),
  max = as.Date("2019-08-16"),
  value = as.Date("2019-08-10")
)
```

Exercise 3.2.8.3

If you have a moderately long list, it's useful to create sub-headings that break the list up into pieces. Read the documentation for `selectInput()` to figure out how. (Hint: the underlying HTML is called `<optgroup>`.)

Solution.

Solution

We can make the `choices` argument a list of key-value pairs where the keys represent the sub-headings and the values are lists containing the categorized elements by keys. As an illustration, the following example separates animal breeds into two keys (categories): "dogs" and "cats".

```
selectInput(
  "breed",
  "Select your favorite animal breed:",
  choices =
    list(`dogs` = list('German Shepherd', 'Bulldog', 'Labrador Retriever'),
         `cats` = list('Persian cat', 'Bengal cat', 'Siamese Cat'))
)
```

If you run the snippet above in the console, you will see the HTML code needed to generate the input. You can also see the `<optgroup>` as hinted in the exercise.

Exercise 3.2.8.4

Create a slider input to select values between 0 and 100 where the interval between each selectable value on the slider is 5. Then, add animation to the input widget so when the user presses play the input widget scrolls through automatically.

Solution.

Solution

We can set the interval between each selectable value using the `step` argument. In addition, by setting `animate = TRUE`, the slider will automatically animate once the user presses play.

```
  sliderInput("number", "Select a number:",
              min = 0, max = 100, value = 0,
              step = 5, animate = TRUE)
```

Exercise 3.2.8.5

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

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

Solution.

Solution

The step argument is the amount by which the `numericInput` value is incremented (resp. decreased) when the user clicks the up (resp. down) arrow. In the

previous example, when the user clicks the up (resp. down) arrow the numeric value will increase (resp. decrease) by 50. Note: by using a `numericInput` the user still has the ability to type *any* number.

Exercise 3.3.5.1

Re-create the Shiny app from the plots section, this time setting height to 300px and width to 700px.

Solution.

Solution

The function `plotOutput` can take on static `width` and `height` arguments. Using the app from the plots section, we only need to add the height argument and modify the width.

```
library(shiny)

ui <- fluidPage(
  plotOutput("plot", width = "700px", height = "300px")
)

server <- function(input, output, session) {
  output$plot <- renderPlot(plot(1:5), res = 96)
}

shinyApp(ui, server)
```

Exercise 3.3.5.2

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

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

Solution.

Solution

We can achieve this by setting `ordering` and `searching` to `FALSE` within the `options` list.

```
library(shiny)

ui <- fluidPage(
  dataTableOutput("table")
)

server <- function(input, output, session) {
  output$table <- renderDataTable(
    mtcars, options = list(ordering = FALSE, searching = FALSE))
}

shinyApp(ui, server)
```

Exercise 3.4.6.1

Create an app that contains two plots, each of which takes up half the app (regardless of what size the whole app is)

Solution.

Solution

When creating the layout of a shiny app, you can use the `fluidRow` function to control the width of the objects it contains. This function can have columns and such columns can be set to have widths ranging from 1-12. Note that columns width within a `fluidRow` container should add up to 12.

For our exercise, we need two columns of 50% width each, i.e., we should set the width of each column to 6.

```
library(shiny)

ui <- fluidPage(
  fluidRow(
    column(width = 6, plotOutput("plot1")),
    column(width = 6, plotOutput("plot2"))
  )
)
server <- function(input, output, session) {
  output$plot1 <- renderPlot(plot(1:5))
```

```
  output$plot2 <- renderPlot(plot(1:5))
}

shinyApp(ui, server)
```

Exercise 3.4.6.2

Modify the Central Limit Theorem app so that the sidebar is on the right instead of the left.

Solution.

Solution

Looking at `?sidebarLayout` we can simply set the `position` argument to `right`. We only need to modify the UI of the app.

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

Exercise 3.4.6.3

Browse the themes available in the shinythemes package, pick an attractive theme, and apply it to the Central Limit Theorem app.

Solution.

Solution

We can browse the themes here and apply it by setting the `theme` argument within `fluidPage` to `shinythemes::shinytheme("theme_name")`

```
library(shinythemes)

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

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

shinyApp(ui, server)
```

Chapter 4

Basic Reactivity

Exercise 4.3.6.1

Draw the reactive graph for the following server functions:

```
server1 <- function(input, output, session) {
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)
  output$f <- renderText(e())
}

server2 <- function(input, output, session) {
  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$z <- renderText(x() / y())
}

server3 <- function(input, output, session) {
  d <- reactive(c() ^ input$d)
  a <- reactive(input$a * 10)
  c <- reactive(b() / input$c)
  b <- reactive(a() + input$b)
}
```

Solution.

Solution

To create the reactive graph we need to consider the inputs, reactive expressions, and outputs of the app.

For `server1` we have the following objects:

- inputs: `input$a`, `input$b`, and `input$d`
- reactives: `c()` and `e()`
- outputs: `output$f`

Inputs `input$a` and `input$b` are used to create `c()`, which is combined with `input$d` to create `e()`. The output depends only on `e()`.

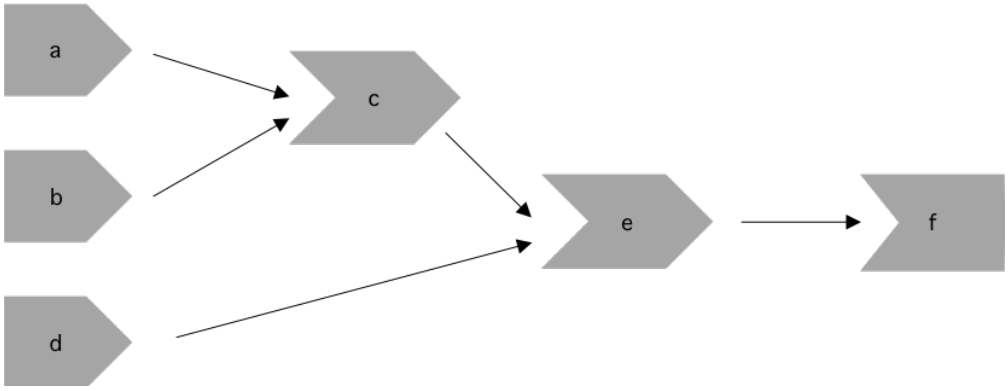


Figure 4.1: reactive graph - server 1

For `server2` we have the following objects:

- inputs: `input$y1`, `input$y2`, `input$x1`, `input$x2`, `input$x3`
- reactives: `y()` and `x()`
- outputs: `output$z`

Inputs `input$y1` and `input$y2` are needed to create the reactive `y()`. In addition, inputs `input$x1`, `input$x2`, and `input$x3` are required to create the reactive `x()`. The output depends on both `x()` and `y()`.

For `server3` we have the following objects:

- inputs: `input$a`, `input$b`, `input$c`, `input$d`
- reactives: `a()`, `b()`, `c()`, `d()`

As we can see below, `a()` relies on `input$a`, `b()` relies on both `a()` and `input$b`, and `c()` relies on both `b()` and `input$c`. The final output depends on both `c()` and `input$d`.

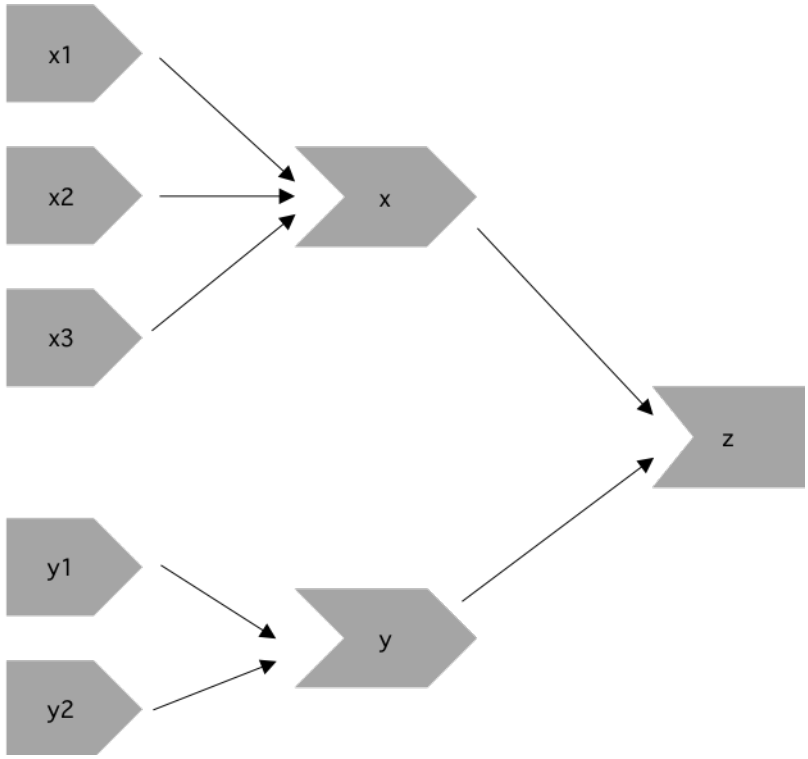


Figure 4.2: reactive graph - server 2

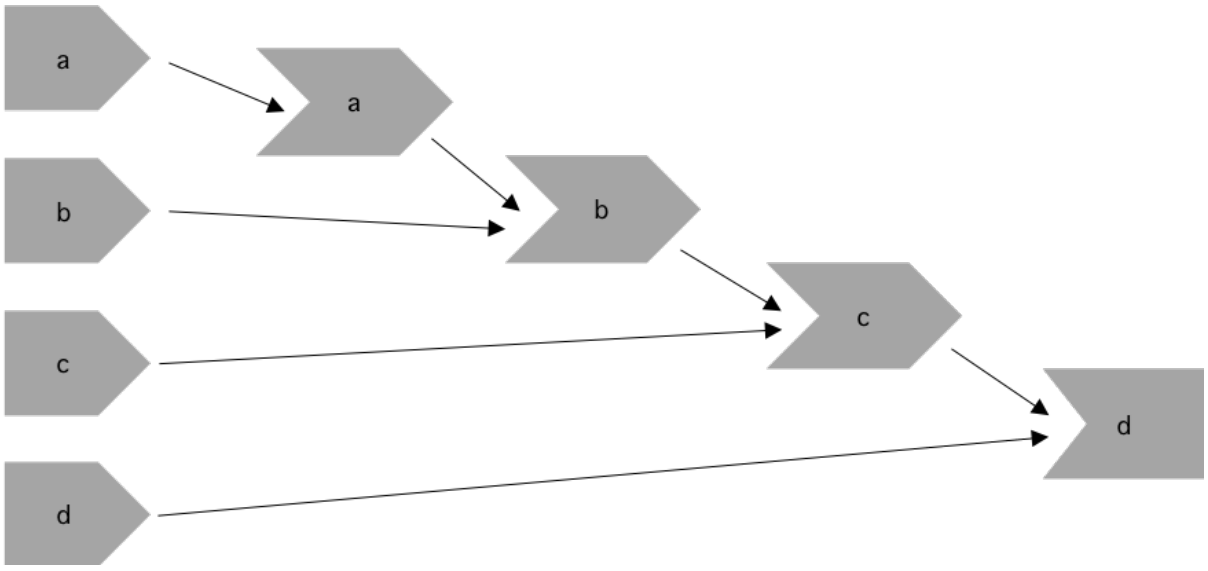


Figure 4.3: reactive graph - server 3

Exercise 4.3.6.2

Why will this code fail?

```
var <- reactive(df[input$var])
range <- reactive(range(var(), na.rm = TRUE))
```

Why is `var()` a bad name for a reactive?

Solution.

Solution

This code doesn't work because we called our reactive `range`, so when we call the `range` function we're actually calling our new reactive. If we change the name of the reactive from `range` to `col_range` then the code will work. Similarly, `var()` is not a good name for a reactive because it's already a function to compute the variance of `x`! `?cor::var`

```
library(shiny)

df <- mtcars

ui <- fluidPage(
    selectInput("var", NULL, choices = colnames(df)),
    verbatimTextOutput("debug")
)

server <- function(input, output, session) {
    col_var <- reactive( df[input$var] )
    col_range <- reactive({ range(col_var(), na.rm = TRUE ) })
    output$debug <- renderPrint({ col_range() })

}

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

Exercise 4.4.6.1

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

Solution.

Solution

Unclear about the apps mentioned in the exercise.

Chapter 5

Case Study: ER Injuries

Exercise 5.8.1

Draw the reactive graph for each app.

Solution.

Solution

Prototype The prototype application has a single input, `input$code`, which is used to generate the `selected()` reactive. This reactive is used directly in 3 outputs, `output$diag`, `output$body_part`, and `output$location`, and it is also used indirectly in the `output$age_sex` plot via the `summary()` reactive.

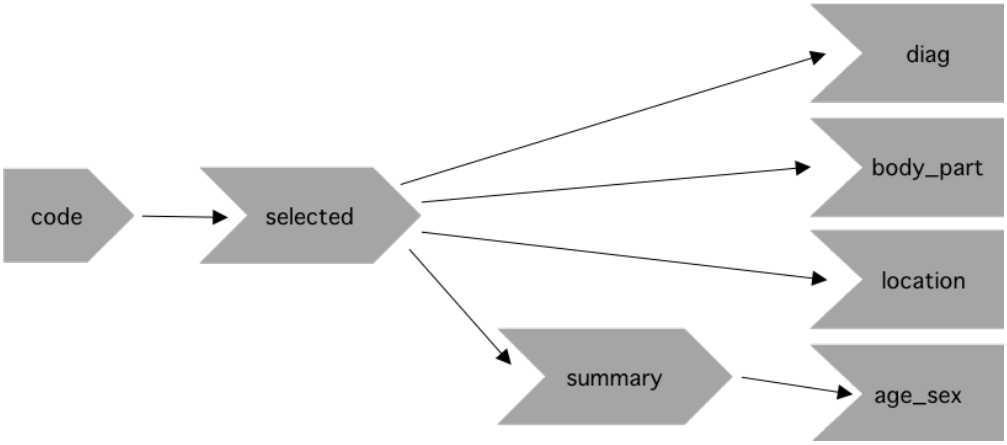


Figure 5.1: reactive graph - prototype app

Rate vs. Count Building on the prototype, we create a second input `input$y` which is used along with the `summary()` reactive to create the `output$age_sex` plot.

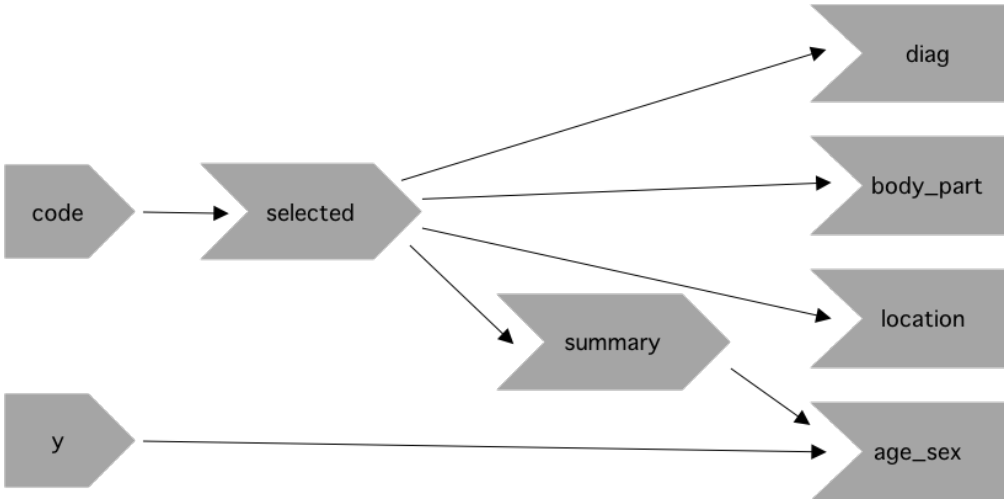


Figure 5.2: reactive graph - rate vs. count app

Narrative Building on the application once more, we create an `output$narrative` that depends on the `selected()` reactive and a new input, `input$story`.

Exercise 5.8.2

What happens if you flip `fct_infreq()` and `fct_lump()` in the code that reduces the summary tables?

Solution.

Solution

As in the book, we will use the datasets `injuries`, `products`, and `population` appearing here: https://github.com/hadley/mastering-shiny/blob/main/neiss/data.R.

Flipping the order of `fct_infreq()` and `fct_lump()` will only change the factor levels order. In particular, the function `fct_infreq()` orders the factor levels by frequency, and the function `fct_lump()` also orders the factor levels by frequency but it will only keep the top `n` factors and label the rest as `Other`.

Let us look at the top five levels in terms of count within the `diag` column in the `injuries` dataset:

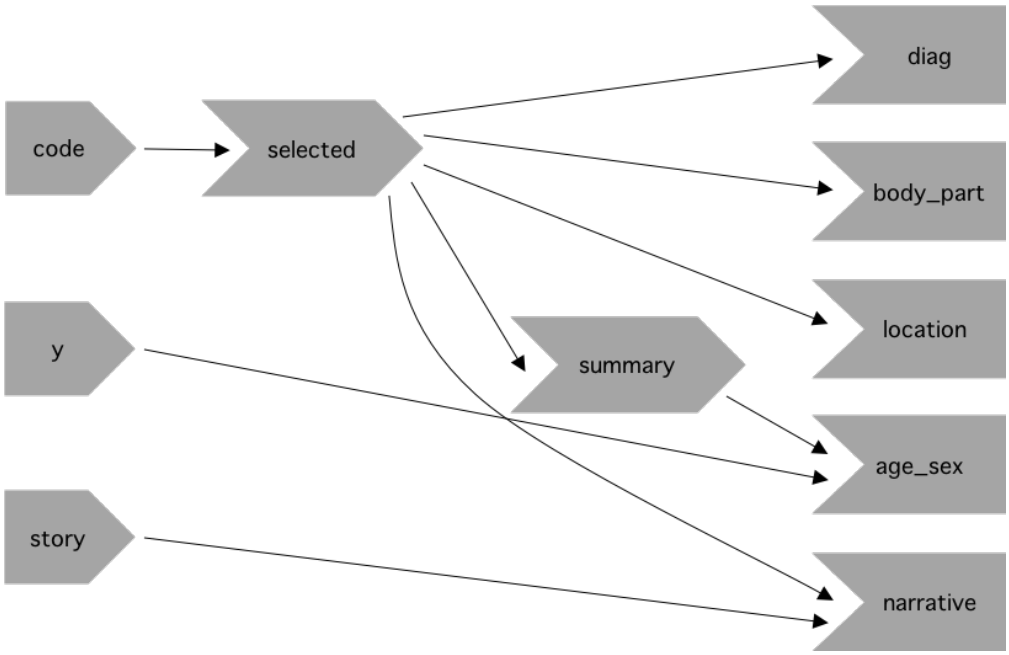


Figure 5.3: reactive graph - narrative app

```
injuries %>%
  group_by(diag) %>%
  count() %>%
  arrange(-n) %>%
  head(5)
```

```
## # A tibble: 5 x 2
## # Groups:   diag [5]
##   diag                      n
##   <chr>                 <int>
## 1 Other Or Not Stated   44937
## 2 Fracture              43093
## 3 Laceration            39230
## 4 Strain, Sprain        37002
## 5 Contusion Or Abrasion 35259
```

If we apply `fct_infreq()` first, then it will reorder the factor levels in descending order as seen in the previous output. If afterwards we apply `fct_lump()`, then it will lump together everything after the nth most commonly seen level.

```
diag <- injuries %>%
  mutate(diag = fct_lump(fct_infreq(diag), n = 5)) %>%
  pull(diag)

levels(diag)
```

```
## [1] "Other Or Not Stated"   "Fracture"              "Laceration"
## [4] "Strain, Sprain"        "Contusion Or Abrasion" "Other"
```

Conversely, if we apply `fct_lump()` first, then it will label the most frequently seen factor level as "Other". If afterwards we apply `fct_infreq()`, then it will label the first level as "Other" and not as "Other Or Not Stated", which was the case for the previous code.

```
diag <- injuries %>%
  mutate(diag = fct_infreq(fct_lump(diag, n = 5))) %>%
  pull(diag)

levels(diag)
```

```
## [1] "Other"                 "Other Or Not Stated"   "Fracture"
## [4] "Laceration"            "Strain, Sprain"        "Contusion Or Abrasion"
```

Exercise 5.8.3

Add an input control that lets the user decide how many rows to show in the summary tables.

Solution.

Solution

Our function `count_top` is responsible for grouping our variables into a set number of factors, lumping the rest of the values into "Other". The function has an argument `n` which is set to `5`. By creating a `numericInput` called `rows` we can let the user set the number of `fct_infreq` dynamically. However, because `fct_infreq` is the number of factors + `Other`, we need to subtract 1 from what the user selects in order to display the number of rows they input.

```
library(shiny)
library(forcats)
library(dplyr)
library(ggplot2)
```

```
# Note: these exercises use the datasets `injuries`, `products`, and
# `population` as created here:
# https://github.com/hadley/mastering-shiny/blob/main/neiss/data.R

count_top <- function(df, var, n = 5) {
  df %>%
    mutate({{ var }} := fct_lump(fct_infreq({{ var }}), n = n)) %>%
    group_by({{ var }}) %>%
    summarise(n = as.integer(sum(weight)))
}

ui <- fluidPage(
  fluidRow(
    column(8, selectInput("code", "Product",
                          choices = setNames(products$prod_code, products$title),
                          width = "100%")
    ),
    column(2, numericInput("rows", "Number of Rows",
                           min = 1, max = 10, value = 5)),
    column(2, selectInput("y", "Y Axis", c("rate", "count")))
  ),
  fluidRow(
    column(4, tableOutput("diag")),
    column(4, tableOutput("body_part")),
    column(4, tableOutput("location"))
  ),
  fluidRow(
    column(12, plotOutput("age_sex"))
  ),
  fluidRow(
    column(2, actionButton("story", "Tell me a story")),
    column(10, textOutput("narrative"))
  )
)

server <- function(input, output, session) {
  selected <- reactive(injuries %>% filter(prod_code == input$code))

  # Find the maximum possible of rows.
  max_no_rows <- reactive(
    max(length(unique(selected()$diag)),
        length(unique(selected()$body_part)),
        length(unique(selected()$location)))
  )
```

```
  # Update the maximum value for the numericInput based on max_no_rows().
  observeEvent(input$code, {
    updateNumericInput(session, "rows", max = max_no_rows())
  })

  table_rows <- reactive(input$rows - 1)

  output$diag <- renderTable(
    count_top(selected(), diag, n = table_rows()), width = "100%")

  output$body_part <- renderTable(
    count_top(selected(), body_part, n = table_rows()), width = "100%")

  output$location <- renderTable(
    count_top(selected(), location, n = table_rows()), width = "100%")

  summary <- reactive({
    selected() %>%
      count(age, sex, wt = weight) %>%
      left_join(population, by = c("age", "sex")) %>%
      mutate(rate = n / population * 1e4)
  })

  output$age_sex <- renderPlot({
    if (input$y == "count") {
      summary() %>%
        ggplot(aes(age, n, colour = sex)) +
        geom_line() +
        labs(y = "Estimated number of injuries") +
        theme_grey(15)
    } else {
      summary() %>%
        ggplot(aes(age, rate, colour = sex)) +
        geom_line(na.rm = TRUE) +
        labs(y = "Injuries per 10,000 people") +
        theme_grey(15)
    }
  })

  output$narrative <- renderText({
    input$story
    selected() %>% pull(narrative) %>% sample(1)
  })
}
```

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

Exercise 5.8.4

Provide a way to step through every narrative systematically with forward and backward buttons.

Advanced: Make the list of narratives "circular" so that advancing forward from the last narrative takes you to the first.

Solution.

Solution

We can add two action buttons `prev_story` and `next_story` to iterate through the narrative. We can leverage the fact that whenever you click an action button in Shiny, the button stores how many times that button has been clicked. To caculate the index of the current story, we can subtract the stored count of the `next_story` button from the `previous_story` button. Then, by using the modulus operator, we can increase the current position in the narrative while never go beyond the interval [1, length of the narrative].

```
library(shiny)
library(forcats)
library(dplyr)
library(ggplot2)

# Note: these exercises use the datasets `injuries`, `products`, and
# `population` as created here:
# https://github.com/hadley/mastering-shiny/blob/main/neiss/data.R

count_top <- function(df, var, n = 5) {
  df %>%
    mutate({{ var }} := fct_lump(fct_infreq({{ var }}), n = n)) %>%
    group_by({{ var }}) %>%
    summarise(n = as.integer(sum(weight)))
}

ui <- fluidPage(
  fluidRow(
    column(8, selectInput("code", "Product",
                          choices = setNames(products$prod_code, products$title),
                          width = "100%")
    ),
```

```
    column(2, numericInput("rows", "Number of Rows",
                           min = 1, max = 10, value = 5)),
    column(2, selectInput("y", "Y Axis", c("rate", "count")))
  ),
  fluidRow(
    column(4, tableOutput("diag")),
    column(4, tableOutput("body_part")),
    column(4, tableOutput("location"))
  ),
  fluidRow(
    column(12, plotOutput("age_sex"))
  ),
  fluidRow(
    column(2, actionButton("prev_story", "Previous story")),
    column(2, actionButton("next_story", "Next story")),
    column(8, textOutput("narrative"))
  )
)

server <- function(input, output, session) {
  selected <- reactive(injuries %>% filter(prod_code == input$code))

  # Find the maximum possible of rows.
  max_no_rows <- reactive(
    max(length(unique(selected()$diag)),
        length(unique(selected()$body_part)),
        length(unique(selected()$location)))
  )

  # Update the maximum value for the numericInput based on max_no_rows().
  observeEvent(input$code, {
    updateNumericInput(session, "rows", max = max_no_rows())
  })

  table_rows <- reactive(input$rows - 1)

  output$diag <- renderTable(
    count_top(selected(), diag, n = table_rows()), width = "100%")

  output$body_part <- renderTable(
    count_top(selected(), body_part, n = table_rows()), width = "100%")

  output$location <- renderTable(
    count_top(selected(), location, n = table_rows()), width = "100%")
```

```
summary <- reactive({
  selected() %>%
    count(age, sex, wt = weight) %>%
    left_join(population, by = c("age", "sex")) %>%
    mutate(rate = n / population * 1e4)
})

output$age_sex <- renderPlot({
  if (input$y == "count") {
    summary() %>%
      ggplot(aes(age, n, colour = sex)) +
      geom_line() +
      labs(y = "Estimated number of injuries") +
      theme_grey(15)
  } else {
    summary() %>%
      ggplot(aes(age, rate, colour = sex)) +
      geom_line(na.rm = TRUE) +
      labs(y = "Injuries per 10,000 people") +
      theme_grey(15)
  }
})

# Store the maximum posible number of stories.
max_no_stories <- reactive(length(selected()$narrative))

# Reactive used to save the current position in the narrative list.
story <- reactiveVal(1)

# Reset the story counter if the user changes the product code.
observeEvent(input$code, {
  story(1)
})

# When the user clicks "Next story", increase the current position in the
# narrative but never go beyond the interval [1, length of the narrative].
# Note that the mod function (%%) is keeping `current`` within this interval.
observeEvent(input$next_story, {
  story((story() %% max_no_stories()) + 1)
})

# When the user clicks "Previous story" decrease the current position in the
# narrative. Note that we also take advantage of the mod function.
observeEvent(input$prev_story, {
  story(((story() - 2) %% max_no_stories()) + 1)
```

```
  })

  output$narrative <- renderText({
    selected()$narrative[story()]
  })
}

shinyApp(ui, server)
```

Chapter 6

Workflow

There are no exercises in this chapter.

Chapter 7

Graphics

Exercise 7.6.1

Make a plot with click handle that shows all the data returned in the input.

Solution.

Solution

We can use the `allRows` argument in `nearPoints` to see the entire data and add a boolean column that will be true `TRUE` for the given point (i.e., row) that was clicked.

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    ggplot(mtcars, aes(wt, mpg)) + geom_point()
  }, res = 96)

  output$data <- renderTable({
    nearPoints(mtcars, input$plot_click, allRows = TRUE)
  })
}
```

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

Exercise 7.6.2

Make a plot with click, dblclick, hover, and brush output handlers and nicely display the current selection in the sidebar. Plot the plot in the main panel.

Solution.

Solution

We can use the `nearPoints` function to extract the data from `plot_click`, `plot_dbl`, and `plot_hover`. We need to use the function `brushedPoints` to extract the points within the `plot_brush` area.

To 'nicely' display the current selection, we will use `dataTableOutput`.

```
library(shiny)
library(ggplot2)

# Set options for rendering DataTables.
options <- list(
  autoWidth = FALSE,
  searching = FALSE,
  ordering = FALSE,
  lengthChange = FALSE,
  lengthMenu = FALSE,
  pageLength = 5, # Only show 5 rows per page.
  paging = TRUE, # Enable pagination. Must be set for pageLength to work.
  info = FALSE
)

ui <- fluidPage(

  sidebarLayout(
    sidebarPanel(
      width = 6,

      h4("Selected Points"),
      dataTableOutput("click"), br(),

      h4("Double Clicked Points"),
      dataTableOutput("dbl"), br(),
```

```
      h4("Hovered Points"),
      dataTableOutput("hover"), br(),

      h4("Brushed Points"),
      dataTableOutput("brush")
    ),

    mainPanel(width = 6,
              plotOutput("plot",
                         click = "plot_click",
                         dblclick = "plot_dbl",
                         hover = "plot_hover",
                         brush = "plot_brush")
    )
  )
)

server <- function(input, output, session) {

  output$plot <- renderPlot({
    ggplot(iris, aes(Sepal.Length, Sepal.Width)) + geom_point()
  }, res = 96)

  output$click <- renderDataTable(
    nearPoints(iris, input$plot_click),
    options = options)

  output$hover <- renderDataTable(
    nearPoints(iris, input$plot_hover),
    options = options)

  output$dbl <- renderDataTable(
    nearPoints(iris, input$plot_dbl),
    options = options)

  output$brush <- renderDataTable(
    brushedPoints(iris, input$plot_brush),
    options = options)
}

shinyApp(ui, server)
```

Exercise 7.6.3

Compute the limits of the distance scale using the size of the plot.

```
output_size <- function(id) {
  reactive(c(
    session$clientData[[paste0("output_", id, "_width")]],
    session$clientData[[paste0("output_", id, "_height")]]
  ))
}
```

Solution.

Solution

Let us use the plot's width and height to estimate the scale limits for our plot.

To verify that the recommended limits are correct, click around the plot and watch how the distance scale changes on the legend. These values should oscillate between the recommended limits.

Resize the browser's window to change the width and height reactives.

```
library(shiny)
library(ggplot2)

df <- data.frame(x = rnorm(100), y = rnorm(100))

ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  textOutput("width"),
  textOutput("height"),
  textOutput("scale")
)

server <- function(input, output, session) {

  # Save the plot's widht and height.
  width <- reactive(session$clientData[["output_plot_width"]])
  height <- reactive(session$clientData[["output_plot_height"]])

  # Print the plot's width, the plot's height, and the suggested scale limits.
  output$width <- renderText(paste0("Plot's width: ", width()))
  output$height <- renderText(paste0("Plot's height: ", height()))
  output$scale <- renderText({
    paste0("Recommended limits: (0, ", max(height(), width()), ")")
```

```
  })

  # Store the distance computed by the click event.
  dist <- reactiveVal(rep(1, nrow(df)))

  # Update the dist reactive as needed.
  observeEvent(input$plot_click, {
    req(input$plot_click)
    dist(nearPoints(df, input$plot_click, allRows = TRUE, addDist = TRUE)$dist_)
  })

  output$plot <- renderPlot({
    df$dist <- dist()
    ggplot(df, aes(x, y, size = dist)) +
      geom_point()
  })
}

shinyApp(ui, server)
```

Chapter 8

User Feedback

There are no exercises in this chapter.

Chapter 9

Uploads and Downloads

Exercise 9.4.1

Use the ambient package by Thomas Lin Pedersen to generate worley noise and download a PNG of it.

Solution.

Solution

A general method for saving a png file is to select the png driver using the function `png()`. The only argument the driver needs is a filename (this will be stored relative to your current working directory!). You will not see the plot when running the `plot` function because it is being saved to that file instead. When we're done plotting, we used the `dev.off()` command to close the connection to the driver.

```
library(ambient)
noise <- ambient::noise_worley(c(100, 100))

png("noise_plot.png")
plot(as.raster(normalise(noise)))
dev.off()
```

Exercise 9.4.2

Create an app that lets you upload a csv file, select a variable, and then perform a `t.test()` on that variable. After the user has uploaded the csv file, you'll need to use `updateSelectInput()` to fill in the available variables. See Section 10.1 for details.

Solution.

Solution

We can use the `fileInput` widget with the `accept` argument set to `.csv` to allow only the upload of csv files. In the `server` function we save the uploaded data to the the `data` reactive and use it to update `input$variable`, which displays variable (i.e. numeric data column) choices. Note that we put the `updateSelectInput` within an observe event because we need the `input$variable` to change if the user selects another file.

```
library(shiny)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      fileInput("file", "Upload CSV", accept = ".csv"), # file widget
      selectInput("variable", "Select Variable", choices = NULL) # select widget
    ),
    mainPanel(
      verbatimTextOutput("results") # t-test results
    )
  )
)

server <- function(input, output,session) {

  # get data from file
  data <- reactive({
    req(input$file)

    # as shown in the book, lets make sure the uploaded file is a csv
    ext <- tools::file_ext(input$file$name)
    validate(need(ext == "csv", "Invalid file. Please upload a .csv file"))

    dataset <- vroom::vroom(input$file$datapath, delim = ",")

    # let the user know if the data contains no numeric column
    validate(need(ncol(dplyr::select_if(dataset, is.numeric)) != 0,
                  "This dataset has no numeric columns."))
    dataset
  })

  # create the select input based on the numeric columns in the dataframe
  observeEvent(input$file, {
```

```
    req(data())
    num_cols <- dplyr::select_if(data(), is.numeric)
    updateSelectInput(session, "variable", choices = colnames(num_cols))
  })

  # print t-test results
  output$results <- renderPrint({
    if(!is.null(input$variable))
      t.test(data()[input$variable])
  })
}

shinyApp(ui, server)
```

Exercise 9.4.3

Create an app that lets the user upload a csv file, select one variable, draw a histogram, and then download the histogram. For an additional challenge, allow the user to select from .png, .pdf, and .svg output formats.

Solution.

Solution

Adapting the code from the example above, rather than print a t-test output, we save the plot in a reactive and use it to display the plot/download. We can use the `ggsave` function to switch between `input$extension` types.

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  tagList(
    br(), br(),
    column(4,
           wellPanel(
             fileInput("file", "Upload CSV", accept = ".csv"),
             selectInput("variable", "Select Variable", choices = NULL),
           ),
           wellPanel(
             radioButtons("extension", "Save As:",
                          choices = c("png", "pdf", "svg"), inline = TRUE),
             downloadButton("download", "Save Plot")
           )
```

```
    ),
    column(8, plotOutput("results"))
  )
)

server <- function(input, output,session) {

  # get data from file
  data <- reactive({
    req(input$file)

    # as shown in the book, lets make sure the uploaded file is a csv
    ext <- tools::file_ext(input$file$name)
    validate(need(ext == "csv", "Invalid file. Please upload a .csv file"))

    dataset <- vroom::vroom(input$file$datapath, delim = ",")

    # let the user know if the data contains no numeric column
    validate(need(ncol(dplyr::select_if(dataset, is.numeric)) != 0,
                  "This dataset has no numeric columns."))
    dataset
  })

  # create the select input based on the numeric columns in the dataframe
  observeEvent( input$file, {
    req(data())
    num_cols <- dplyr::select_if(data(), is.numeric)
    updateSelectInput(session, "variable", choices = colnames(num_cols))
  })

  # plot histogram
  plot_output <- reactive({
    req(!is.null(input$variable))

    ggplot(data()) +
      aes_string(x = input$variable) +
      geom_histogram()
  })

  output$results <- renderPlot(plot_output())

  # save histogram using downloadHandler and plot output type
  output$download <- downloadHandler(
    filename = function() {
      paste("histogram", input$extension, sep = ".")
```

```
    },
    content = function(file){
      ggsave(file, plot_output(), device = input$extension)
    }
  )
}

shinyApp(ui, server)
```

Exercise 9.4.4

Write an app that allows the user to create a Lego mosaic from any .png file using Ryan Timpe's brickr package. Once you've completed the basics, add controls to allow the user to select the size of the mosaic (in bricks), and choose whether to use "universal" or "generic" colour palettes.

Solution.

Solution

Instead of limiting our file selection to a csv as above, here we are going to limit our input to a png. We'll use the `png::readPNG` function to read in our file, and specify the size/color of our mosaic in `brickr`'s `image_to_mosaic` function. Read more about the package and examples here.

```
library(shiny)
library(brickr)
library(png)

# Function to provide user feedback (checkout Chapter 8 for more info).
notify <- function(msg, id = NULL) {
  showNotification(msg, id = id, duration = NULL, closeButton = FALSE)
}

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      fluidRow(
        fileInput("myFile", "Upload a PNG file", accept = c('image/png')),
        sliderInput("size", "Select size:", min = 1, max = 100, value = 35),
        radioButtons("color", "Select color palette:", choices = c("universal", "generic"))
      )
    ),
    mainPanel(
```

```
      plotOutput("result"))
  )
)

server <- function(input, output) {

  imageFile <- reactive({
    if(!is.null(input$myFile))
      png::readPNG(input$myFile$datapath)
  })

  output$result <- renderPlot({
    req(imageFile())

    id <- notify("Transforming image...")
    on.exit(removeNotification(id), add = TRUE)

    imageFile() %>%
      image_to_mosaic(img_size = input$size, color_palette = input$color) %>%
      build_mosaic()
  })
}

shinyApp(ui, server)
```

Chapter 10

Dynamic UI

Exercise 10.1.5.1

Complete the user interface below with a server function that updates `input$date` so that you can only select dates in `input$year`.

```
ui <- fluidPage(
  numericInput("year", "year", value = 2020),
  dateInput("date", "date")
)
```

Solution.

Solution

This solution was a little wonky because it required shinyjs for the dateInput to properly update. I opened up an issue here since I think this is not the most intuitive answer.

```
library(shiny)
library(shinyjs)

ui <- fluidPage(
  useShinyjs() ,
  numericInput("year", "year", value = 2020),
  dateInput("date", "date", value = Sys.Date())
)

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

```
  observeEvent(input$year, {

    req(input$year) # stop if year is blank
    daterange <- range(as.Date(paste0(input$year, "-01-01")),as.Date(paste0(input$year
    updateDateInput(session, "date", min = daterange[1], max = daterange[2] )
    delay(250,  # delay 250ms
          updateDateInput(session,"date",
                          value = daterange[1]
          ))
  })
}

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

Exercise 10.1.5.2

Complete the user interface below with a server function that updates `input$county` choices based on `input$state`. For an added challenge, also change the label from "County" to "Parrish" for Louisana and "Borrough" for "Alaska".

```
library(openintro)
states <- unique(county$state)

ui <- fluidPage(
  selectInput("state", "State", choices = states),
  selectInput("county", "County", choices = NULL)
)
```

Solution.

Solution

We can use `updateSelectInput` to filter the county choices based on the user selected state. By making the label of `input$county` a reactive, we can use `switch` to change the label when either Alaska or Louisiana is selected.

```
library(shiny)
library(tidyverse)
library(openintro)

states <- unique(county$state)
counties <- unique(county$state)
```

```
ui <- fluidPage(
  selectInput("state", "State", choices = states),
  selectInput("county", "County", choices = NULL)
)


server <- function(input, output, session) {

  label <- reactive({
    switch(input$state,
           "Alaska" = "Burrough",
           "Louisiana" = "Parish",
           "County")
  })

  observeEvent( input$state, {
    updateSelectInput(session, "county", label = label(),
                      choices = county %>%
                        filter(state == input$state) %>%
                        select(name) %>%
                        distinct())
  })

}


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

Exercise 10.1.5.3

Complete the user interface below with a server function that updates `input$country` choices based on the `input$continent`. Use `output$data` to display all matching rows.

```
library(gapminder)
continents <- unique(gapminder$continent)

ui <- fluidPage(
  selectInput("continent", "Continent", choices = continents),
  selectInput("country", "Country", choices = NULL),
  tableOutput("data")
)
```

Solution.

Solution

As the question above, we are filtering the country input based on the continent by using `updateSelectInput` in the server. By storing the selected data in a reactive, `selected_data()` we can use the same filtered dataset for our `selectInput` and the table, reducing code redundancy.

```
library(shiny)

library(gapminder)
continents <- unique(gapminder$continent)

ui <- fluidPage(
  selectInput("continent", "Continent", choices = c("", as.character(continents))),
  selectInput("country", "Country", choices = NULL),
  tableOutput("data")
)

server <- function(input, output, session) {

  selected_data <- reactive({
    if(input$continent %in% continents) {
      gapminder %>%
      filter(continent == input$continent)
    } else {
      gapminder
    }
  })

  observeEvent( input$continent, {
    updateSelectInput(session, "country", "Country",
                      choices = selected_data() %>%
                        select(country) %>%
                        distinct())
  })

  output$data <- renderTable({
    selected_data() %>%
      filter(country == input$country)
  })
}

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

Exercise 10.1.5.4

Extend the previous app so that you can also choose to select no continent, and hence see all countries. You'll need to add " " to the list of choices, and then handle that specially when filtering.

Solution.

Solution

Initially setting the choices to `c("", as.character(continents))` allows the user to see all the Country options prior to a continent being selected. That said, once a continent is selected this `""` option is no longer available.

```
library(shiny)

library(gapminder)
continents <- unique(gapminder$continent)

ui <- fluidPage(
  selectInput("continent", "Continent", choices = c("", as.character(continents))),
   # @tanho63:
  # selectInput("continent", "Continent", choices = c("All", as.character(continents))),
  selectInput("country", "Country", choices = NULL),
  tableOutput("data")
)


server <- function(input, output, session) {

  selected_data <- reactive({
    if(input$continent %in% continents) {
      gapminder %>%
      filter(continent == input$continent)
    } else {
      gapminder
    }
  })

  observeEvent( input$continent, {

    # @tanho63:
    updateSelectInput(session, "country",
                      choices = unique(selected_data()$country))
  })
```

```
  output$data <- renderTable({
    selected_data() %>%
      filter(country == input$country)
  })

}


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

Exercise 10.1.5.5

What is at the heart of the problem described at https://community.rstudio.com/t/29307?

Solution.

Solution

Updating all three sliders creates a circular reference!

Exercise 10.2.3.1

Use a hidden tabset to show additional controls only if the user checks an "advanced" check box.

Solution.

Solution

```
library(shiny)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      checkboxInput("moreControls",
        label = "Show advanced controls?",
        value = FALSE
      )
    ),
    mainPanel(
      tabsetPanel(
```

```
        id = "basic",
        type = "hidden",
        tabPanelBody("panel1",
          numericInput("basicControl", label = "Basic:", 0),
        )
      ),
      tabsetPanel(
        id = "advanced",
        type = "hidden",
        tabPanelBody("emptyPanel", style = "display: none"),
        tabPanelBody("panel2",
          numericInput("advancedCotrol", label = "Advanced:", 1)
        )
      )
    )
  )
)

server <- function(input, output, session) {
  observeEvent(input$moreControls, {
    if (input$moreControls) {
      updateTabsetPanel(session, "advanced", selected = "panel2")
    } else {
      updateTabsetPanel(session, "advanced", selected = "emptyPanel")
    }
  })
}

shinyApp(ui, server)
```

Exercise 10.2.3.2

Create an app that plots `ggplot(diamonds, aes(carat))` but allows the user to choose which geom to use: `geom_histogram()`, `geom_freqpoly()`, or `geom_density()`. Use a hidden tabset to allow the user to select different arguments depending on the geom: `geom_histogram()` and `geom_freqpoly()` have a `binwidth` argument; `geom_density()` has a `bw` argument.

Solution.

Solution

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("geom", "Geom function to use",
        choices = c("histogram", "freqpoly", "density")
      ),
      tabsetPanel(
        id = "params",
        type = "hidden",
        tabPanel("histogram",
          numericInput("hist_bw",
            label = "Binwidth", value = 0.1,
            min = 0.1, max = 5, step = 0.1
          )
        ),
        tabPanel("freqpoly",
          numericInput("freqpoly_bw",
            label = "Binwidth", value = 0.1,
            min = 0.1, max = 5, step = 0.1
          )
        ),
        tabPanel("density",
          numericInput("density_bw",
            label = "Standard deviation of smoothing kernel",
            value = 0.01, min = 0.01, max = 1, step = 0.01
          )
        )
      )
    ),
    mainPanel(
      plotOutput("gg")
    )
  )
)

server <- function(input, output, session) {
  observeEvent(input$geom, {
    updateTabsetPanel(inputId = "params", selected = input$geom)
  })

  gg_args <- reactive({
    switch(input$geom,
```

```
      histogram = geom_histogram(binwidth = input$hist_bw),
      freqpoly = geom_freqpoly(binwidth = input$freqpoly_bw),
      density = geom_density(bw = input$density_bw)
    )
  })

  output$gg <- renderPlot({
    ggplot(diamonds, aes(carat)) +
    gg_args()
  })
}

shinyApp(ui, server)
```

Exercise 10.2.3.3

Modify the app you created in the previous exercise to allow the user to choose whether each geom is shown or not (i.e. instead of always using one geom, they can picked 0, 1, 2, or 3). Make sure that you can control the binwidth of the histogram and frequency polygon independently.

Solution.

Solution

```
library(shiny)
library(ggplot2)

geom_choices <- c("histogram", "freqpoly", "density")

# ------------------------------------------- #
#    Generate the necessary code (as a string)    #
#    for ggplot, after having chosen one of       #
#    the available geom functions in the ui       #
# ------------------------------------------- #
# A list could be used, working only with the three
# provided geom choices, but a fuction will be used,
# to provide a template for a possible generalization
# when working with more geom choices and parameters.

geom_choice_code <- function(geom_choice) {
  # Using %in% could be a more general case,
  # considering more possible geom options.
```

```
  if (geom_choice == "density") {
    return(paste0(
      # Simple density
      # "geom_", geom_choice, "(bw = input$", geom_choice, "_bw)"

      # Match density to the histogram's y-scale
      "geom_density(
        color = 'blue',
        bw = input$density_bw,
        aes(y = ..density.. * (nrow(diamonds) * input$histogram_bw))
      )"
    ))
  }

  return(paste0(
    "geom_", geom_choice, "(",
    # Improve histograms' visibility
    ifelse(geom_choice == "histogram", "fill = 'transparent', color = 'red', ", ""),

    "binwidth = input$", geom_choice, "_bw)"
  ))
}


ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("geom",
        label = "Geom function to use", multiple = TRUE,
        choices = geom_choices, selected = "histogram"
      ),
      tabsetPanel(
        id = "histogram",
        type = "hidden",
        tabPanelBody("histogram_empty", style = "display: none"),
        tabPanelBody("histogram_params",
          # This parameter will also be used to scale the y-axis
          # when plotting density, so that all graphics
          # have a similar y-axis of "count"
          numericInput("histogram_bw",
            label = "Histogram's binwidth", value = 0.15,
            min = 0.1, max = 5, step = 0.1
          )
        )
        ),
```

```
      tabsetPanel(
        id = "freqpoly",
        type = "hidden",
        tabPanelBody("freqpoly_empty", style = "display: none"),
        tabPanelBody("freqpoly_params",
          numericInput("freqpoly_bw",
            label = "freqpoly's binwidth", value = 0.15,
            min = 0.1, max = 5, step = 0.1
          )
        )
      ),
      tabsetPanel(
        id = "density",
        type = "hidden",
        tabPanelBody("density_empty", style = "display: none"),
        tabPanelBody("density_params",
          numericInput("density_bw",
            label = "Standard deviation of smoothing kernel",
            value = 0.15, min = 0.01, max = 1, step = 0.01
          )
        )
      )
    ),
    mainPanel(
      plotOutput("gg")
    )
  )
)

server <- function(input, output, session) {
  # ------------------------------------------- #
  # Update inputs' visibility in sidebar panel #
  # ------------------------------------------- #
  observeEvent(input$geom, {
    # Get non selected geom functions, in order
    # to hide their respective parameters
    non_selected <- setdiff(geom_choices, input$geom)
    purrr::map(
      geom_choices,
      ~ updateTabsetPanel(
        inputId = .x,
        selected = paste0(
          .x,
          ifelse(.x %in% non_selected, "_empty", "_params")
        )
```

```
      )
    )
    # Run this code also when the select input is cleared
  }, ignoreNULL = FALSE)

  # ------------------------ #
  # Retrieve code for ggplot #
  # ------------------------ #
  gg_args <- reactive({
    req(input$geom)

    purrr::map_chr(input$geom, geom_choice_code) |>
      paste(collapse = " + ")
  })

  output$gg <- renderPlot({
    eval(parse(text = paste0(
      "ggplot(diamonds, aes(carat)) + ",
      gg_args(), " + ",
      "labs(y = 'Count')"
    )))
  })
}

shinyApp(ui, server)
```

Exercise 10.3.4.1

Take this very simple app based on the initial example in the chapter:

```
ui <- fluidPage(
  selectInput("type", "type", c("slider", "numeric")),
  uiOutput("numeric")
)
server <- function(input, output, session) {
  output$numeric <- renderUI({
    if (input$type == "slider") {
      sliderInput("n", "n", value = 0, min = 0, max = 100)
    } else {
      numericInput("n", "n", value = 0, min = 0, max = 100)
    }
  })
}
```

How could you instead implement it using dynamic visibility? If you implement dynamic visibility, how could you keep the values in sync when you change the controls?

Solution.

Solution

```
library(shiny)
parameter_tabs <- tagList(
  tags$style("#params { display:none; }"),
  tabsetPanel(id = "params",
              tabPanel("slider",
                       sliderInput("my_slider", "n", value = 0, min = 0, max = 100)
              ),
              tabPanel("numeric",
                       numericInput("my_numeric", "n", value = 0, min = 0, max = 100)
              )
  )
)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("my_selector", "Input Type",
                  choices = c("slider", "numeric")
      ),
      parameter_tabs,
    ),
    mainPanel()
  )
)

server <- function(input, output, session) {

  # if slider changes, update numeric
  observeEvent( input$my_slider, {
    updateNumericInput(session, "my_numeric", value = isolate(input$my_slider))
  })

  # if numeric changes update slider
  observeEvent( input$my_numeric, {
    updateSliderInput(session, "my_slider", value = isolate(input$my_numeric))
  })
```

```
  observeEvent(input$my_selector, {
    updateTabsetPanel(session, "params", selected = input$my_selector)
  })

}

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

Exercise 10.3.4.2

Explain how this app works. Why does the password disappear when you click the enter password button for the second time?

```
ui <- fluidPage(
  actionButton("go", "Enter password"),
  textOutput("text")
)
server <- function(input, output, session) {
  observeEvent(input$go, {
    showModal(modalDialog(
      passwordInput("password", NULL),
      title = "Please enter your password"
    ))
  })

  output$text <- renderText({
    if (!isTruthy(input$password)) {
      "No password"
    } else {
      "Password entered"
    }
  })
}
```

Exercise 10.3.4.3

Add support for date and date-time columns `make_ui()` and `filter_var()`.

Solution.

Solution

In order to complete this, I had to

1) make a new dummy dataframe I called `x` in order to test for dates
2) include checking for `is.Date` in the `make_ui` and `filter_var` functions
3) Change `tableOutput` and `renderTable` to `DT::renderTableOutput` and `DT::renderTableOutput` because `renderTable` was rendering the dates as numbers and I think this could be because it uses `xtable()` for HTML table rendering?

```
# 8.4.3.2
library(shiny)
library(purrr)
library(tidyverse)

make_ui <- function(x, var) {
  if (is.numeric(x)) {
    rng <- range(x, na.rm = TRUE)
    sliderInput(var, var, min = rng[1], max = rng[2], value = rng)
  } else if (is.factor(x)) {
    levs <- levels(x)
    selectInput(var, var, choices = levs, selected = levs, multiple = TRUE)
  } else if (lubridate::is.Date(x)) {
    rng <- range(x, na.rm = TRUE)
    dateInput(var, var, min = rng[1], max = rng[2], value = rng[1])
  } else {
    # No control, so don't filter
    NULL
  }
}


filter_var <- function(x, val) {
  if (is.numeric(x)) {
    !is.na(x) & x >= val[1] & x <= val[2]
  } else if (is.factor(x)) {
    x %in% val
  } else if (lubridate::is.Date(x)) {
    x %in% val
  } else {
    TRUE
  }
}

library(shiny)

dfs <- keep(ls("package:datasets"), ~ is.data.frame(get(.x, "package:datasets")))
```

```
# add a dataframe with dates in it since I cant find one in the datasets above
# rep 5 dates five times, each include 1 factor a-e
x <- data.frame(date = c(rep(as.Date("2020/1/1"), 5),
                         rep(as.Date("2020/2/2"), 5),
                         rep(as.Date("2020/3/3"), 5),
                         rep(as.Date("2020/4/4"), 5),
                         rep(as.Date("2020/5/5"), 5)),
                fac = as.factor(c("a", "b", "c", "d", "e")))

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      #selectInput("dataset", label = "Dataset", choices = c(dfs, "x")),
      uiOutput("filter")
    ),
    mainPanel(
      DT::dataTableOutput("data")
    )
  )
)

server <- function(input, output, session) {

  # data is either my dummy dataset or from datasets
  data <- reactive(x)

  vars <- reactive(names(data()))

  output$filter <- renderUI(
    # take eahc column name and make ui
    # data()[[.x]] is each column
    # and .x is each column name (vars())
    map(vars(), ~ make_ui(data()[[.x]], .x))
  )

  selected <- reactive({
    # take each column name and filer var
    # with the first argument the column in the data
    # and the second argument the input$vars()
    # so for date check that input[[date]] in data[[1]]
    each_var <- map(vars(), ~ filter_var(data()[[.x]], input[[.x]]))

    # notes from @mapaulacaldas
    # collapse list of TRUE and FALSE using `&`
    # conditions <- list(TRUE, TRUE, TRUE, FALSE)
```

```
    # purrr::reduce(conditions, `&`) ==
    # ((conditions[[1]] & conditions[[2]]) & conditions[[3]]) & conditions[[4]]
    reduce(each_var, `&`)
  })

  # subset the data by the vars that are true
  output$data <- DT::renderDataTable(data()[selected(), ])
}
# Run the application
shinyApp(ui = ui, server = server)
```

Exercise 10.3.4.4

(Advanced) If you know the S3 OOP system, consider how you could replace the if blocks in `make_ui()` and `filter_var()` with generic functions.

Solution.

Solution

```
library(shiny)
library(purrr)

make_ui <- function(obj, var) { UseMethod("make_ui") }

make_ui.numeric <- function(x, var) {
  rng <- range(x, na.rm = TRUE)
  sliderInput(var, var, min = rng[1], max = rng[2], value = rng)
}

make_ui.factor <- function(x, var) {
  levs <- levels(x)
  selectInput(var, var, choices = levs, selected = levs, multiple = TRUE)
}

make_ui.default <- function(x, var) { NULL }

filter_var <- function(x, val) { UseMethod("filter_var") }
filter_var.numeric <- function(x, val) { !is.na(x) & x >= val[1] & x <= val[2] }
filter_var.factor <- function(x, val) { x %in% val }
filter_var.default <- function(x, val) { TRUE }

dfs <- keep(ls("package:datasets"), ~ is.data.frame(get(.x, "package:datasets")))
```

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("dataset", label = "Dataset", choices = dfs),
      uiOutput("filter")
    ),
    mainPanel(
      tableOutput("data")
    )
  )
)
server <- function(input, output, session) {
  data <- reactive({
    get(input$dataset, "package:datasets")
  })

  vars <- reactive(names(data()))

  output$filter <- renderUI(
    map(vars(), ~ make_ui(data()[[.x]], .x))
  )

  selected <- reactive({
    each_var <- map(vars(), ~ filter_var(data()[[.x]], input[[.x]]))
    reduce(each_var, `&`)
  })

  output$data <- renderTable(head(data()[selected(), ], 12))
}

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

Exercise 10.3.4.5

(Hard) Make a wizard that allows the user to upload their own dataset. The first page should handle the upload. The second should handle reading it, providing one drop down for each variable that lets the user select the column type. The third page should provide some way to get a summary of the dataset.

Solution.

Solution

I wasn't really sure what was meant by "some way to get a summary of the dataset" So I'm just using the summary function.

```
library(shiny)
library(readr)

make_dropdown <- function(name_of_vector) {
  selectInput(inputId = name_of_vector, label =  name_of_vector, choices =
                c("numeric", "character", "logical"))
}

ui <- fluidPage(
  tags$style("#wizard { display:none; }"),
  tabsetPanel(id = "wizard",
              tabPanel("page1",
                       fileInput("data_input", "input"),
                       actionButton("page12", "next")
              ),
              tabPanel("page2",
                         sidebarLayout(
                           sidebarPanel(
                             uiOutput("type_of")
                           ),
                           mainPanel(
                             tableOutput('type_table')
                           )),
                       actionButton("page21", "prev"),
                       actionButton("page23", "next")
              ),
              tabPanel("page3",
                       tableOutput("summary_table"),
                       actionButton("page32", "prev")
              )
  )
)

server <- function(input, output, session) {

  ################ WIZARD  ################################

  switch_tab <- function(page) {
    updateTabsetPanel(session, "wizard", selected = page)
  }
```

```
observeEvent(input$page12, switch_tab("page2"))
observeEvent(input$page21, switch_tab("page1"))
observeEvent(input$page23, switch_tab("page3"))
observeEvent(input$page32, switch_tab("page2"))

#################### FILE INPUT ########################

dat <- reactive({
  req(input$data_input)
  read.csv(input$data_input$datapath)
})

#################### TABLE TYPE ########################

# make a dropdown using the names of each column
output$type_of <- renderUI({ map(names(dat()), ~ make_dropdown(.x)) })


# switch the type of column based on the input
# name of vector == "Sepal.Length"
# vector == Sepal.Length
change_type <- function(vector, name_of_vector) {
  switch(input[[name_of_vector]],
         "numeric" = vector <- as.numeric(vector),
         "character" = vector <- as.character(vector),
         "logical" = vector <- as.complex(vector)
  )
}

# convert the supplied data to a list
# use imap because it is a condensed version o map
# with two arguments == x & name_of_x
# so we don't need to supply it arguments beyond the list!
df<- reactive({
  dat() %>%
    as.list() %>%
    imap(change_type) %>%
    as_tibble()
})

# create an output of the data's names
# and their types
output$type_table <- renderTable(data.frame(
  names = names(df()),
  type = map_chr(df(), function(x) typeof(x)))
```

```
  )

  ###################### TABLE OUTPUT ######################

  output$summary_table <- renderTable( summary(df()) )
}

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

Chapter 11

Bookmarking

Exercise 11.3.1

Generate app for visualising the results of noise::ambient_simplex(). Your app should allow the user to control the frequency, fractal, lacunarity, and gain, and be bookmarkable. How can you ensure the image looks exactly the same when reloaded from the bookmark? (Think about what the seed argument implies).

Solution.

Solution

For this example, we'll use the bookmarking by setting `enableBookmarking = "url"` within the `shinyApp` function. In order to ensure the simulation is the same each time we re-render the bookmark we'll create a grid to use for points then set the seed to `42` to ensure the same image is rendered when the bookmark is loaded.

```
library(shiny)
library(ambient)

ui <- function(request) {
    fluidPage(
    sidebarLayout(
        sidebarPanel(
            sliderInput("frequency", "Frequency", min = 0, max = 1, value = 0.01),
            selectInput("fractal", "Fractal", choices = c("none", "fbm", "billow", "rigid-multi")
            sliderInput("gain", "Gain", min = 0, max = 1, value = 0.5),
        ),
        mainPanel(
```

```
            plotOutput("result")
        )
    )
  )
}


server <- function(input, output, session) {

    grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))

    noise <- reactive({
        ambient::gen_simplex(
            x = grid$x,
            y = grid$y,
            seed = 42,
            frequency = input$frequency,
            fractal = input$fractal,
            gain = input$gain
        )
    })

    output$result <- renderPlot({
        plot(grid, noise())
    })

    observe({
        reactiveValuesToList(input)
        session$doBookmark()
    })
    onBookmarked(updateQueryString)
}

shinyApp(ui = ui, server = server, enableBookmarking = "url")
```

Exercise 11.3.2

Make a simple app that lets you upload a csv file and then bookmark it. Upload a few files and then look in shiny_bookmarks. How do the files correspond to the bookmarks? (Hint: you can use readRDS() to look inside the cache files that Shiny is generating).

Solution.

Solution

By setting the `state$values$data` equal to the `data` reactive, we can store the contents of the uploaded `csv`. Looking in the `shiny_bookmarks` folder we see an `input.rds` which has the same 4 arguments as `input$file`:

1. `name`
2. `size`
3. `type`
4. `datapath`

All of these except the `datapath` are the same as when we upload the file; rather than the temporary location the file is saved to within the shiny session, the datapath becomes `0.csv`, a csv file created within the same folder are our `input.RDS`.

```
library(shiny)

ui <- function(request){
    fluidPage(
        sidebarLayout(
            sidebarPanel(
                bookmarkButton(),
                fileInput("file", "Choose CSV File", multiple = TRUE,accept = ".csv")
                ),
            mainPanel(
                tableOutput("contents")
            )
        )
    )
}


server <- function(input, output) {

    # create reactive of input file
    data <- reactive({
        req(input$file)
        read.csv(input$file$datapath)
    })

    # display head
    output$contents <- renderTable( head(data()) )

    # set the state to the df reactive
```

```
    onBookmark(function(state){
        state$values$data <- data()
    })

    # on restore set df to the state
    onRestore(function(state){
        data <- reactive(state$values$data)
    })
    enableBookmarking(store="server")
}

shinyApp(ui, server)
```

Chapter 12

Tidy Evaluation

There are no exercises in this chapter.

Chapter 13

General Guidelines

There are no exercises in this chapter.

Chapter 14

Functions

Exercise 14.4.1

The following app plots user selected variables from the msleep dataset for three different types of mammals (carnivores, omnivores, and herbivores), with one tab for each type of mammal. Remove the redundancy in the `selectInput()` definitions with the use of functions.

```
library(tidyverse)

ui <- fluidPage(
  selectInput(inputId = "x",
              label = "X-axis:",
              choices = c("sleep_total", "sleep_rem", "sleep_cycle",
                          "awake", "brainwt", "bodywt"),
              selected = "sleep_rem"),
  selectInput(inputId = "y",
              label = "Y-axis:",
              choices = c("sleep_total", "sleep_rem", "sleep_cycle",
                          "awake", "brainwt", "bodywt"),
              selected = "sleep_total"),
  tabsetPanel(id = "vore",
              tabPanel("Carnivore",
                       plotOutput("plot_carni")),
              tabPanel("Omnivore",
                       plotOutput("plot_omni")),
              tabPanel("Herbivore",
                       plotOutput("plot_herbi")))
)
```

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

  # make subsets
  carni <- reactive( filter(msleep, vore == "carni") )
  omni  <- reactive( filter(msleep, vore == "omni")  )
  herbi <- reactive( filter(msleep, vore == "herbi") )

  # make plots
  output$plot_carni <- renderPlot({
    ggplot(data = carni(), aes_string(x = input$x, y = input$y)) +
      geom_point()
  }, res = 96)
  output$plot_omni <- renderPlot({
    ggplot(data = omni(), aes_string(x = input$x, y = input$y)) +
      geom_point()
  }, res = 96)
  output$plot_herbi <- renderPlot({
    ggplot(data = herbi(), aes_string(x = input$x, y = input$y)) +
      geom_point()
  }, res = 96)

}

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

Solution.

Solution

We can see a pattern here where we are creating the same type of plot for each tabset panel, with the only variable changing being the `vore` argument. We can reduce everything we see in triplicate to functions! We can use `map` to create a single `create_panels` function which will create a tab for each of our `species`. On the server side, the data is filtered three times, and the plots are created three times. We can create a single rendering function that given the correct string it will filter the data, create the correct plot, and assign it to the correct output.

```
library(tidyverse)

# use a vector for function inputs
species <- c("Carnivore", "Omnivore", "Herbivore")

# educe to a single UI function
# Marly: this didn't work!
```

```
create_panels <- function(id) {
    tabPanel(id, plotOutput(paste0("plot_", id)))
}

ui <- fluidPage(
    selectInput(inputId = "x",
                label = "X-axis:",
                choices = c("sleep_total", "sleep_rem", "sleep_cycle",
                            "awake", "brainwt", "bodywt"),
                selected = "sleep_rem"),
    selectInput(inputId = "y",
                label = "Y-axis:",
                choices = c("sleep_total", "sleep_rem", "sleep_cycle",
                            "awake", "brainwt", "bodywt"),
                selected = "sleep_total"),
    tabsetPanel(
        tabPanel("Carnivore", plotOutput("plot_Carnivore")),
        tabPanel("Omnivore", plotOutput("plot_Omnivore")),
        tabPanel("Herbivore", plotOutput("plot_Herbivore"))
    )
    # this works without the tabsetPanel function - why?!
    # purrr::map(species, create_panels)
)

server <- function(input, output, session) {

    # rendering plot function for each panel
    render_outputs <- function(id) {
        output[[paste0("plot_", id)]] <- renderPlot({
            msleep %>%
                filter(vore == tolower(stringr::str_remove(id, "vore")) %>%
                ggplot() +
                aes_string(x = input$x, y = input$y) +
                geom_point()
            )
        })
    }

    # apply to the species vector using map
    purrr::map(species, render_outputs)
}

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

Exercise 14.4.2

Continue working with the same app from the previous exercise, and further remove redundancy in the code by modularizing how subsets and plots are created.

Solution.

Solution

TODO: I'm unsure what to do with this one since we haven't yet introduced modules?

Exercise 14.4.3

Suppose you have an app that is slow to launch when a user visits it. Can modularizing your app code help solve this problem? Explain your reasoning.

Solution.

Solution

No, we're just packaging our code into neater functions - this doesn't change or optimize what is loaded when the app is launched. In fact, modularizing might even make your application slower in some cases.

Chapter 15

Modules

Exercise 15.6.1

Example passing input$foo to reactive and it not working.

Solution.

Solution

I don't really know what this question is asking, but I think the point is to remember:

> The main challenge with this sort of code is remembering when you use the reactive (e.g. x$value) vs. when you use its value (e.g. x$value()). Just remember that when passing an argument to a module, you want the module to react to the value changing which means that you have to pass the reactive, not it's current value.

Where in this scenario, `input$foo` is analogous to `x$value`.

Exercise 15.6.2

Rewrite `selectVarServer()` so that both data and filter are reactive. Pair it with a app function that lets the user pick the dataset with the dataset module, a function with an `inputSelect()` that lets the user filter for numeric, character, or factor variables.

Solution.

Solution

The modules `datasetInput`, `datasetServer`, and `selectVarInput` are the same, as well as the `find_vars` function.

We can start by creating `selectFilterInput` which has the filtering options as choices, and `selectFilterServer` which returns the filtering function given the selected choice string.

```
# create a filter selection input
selectFilterInput <- function(id) {
    selectInput(NS(id, "filter"), "Filter",
                choices = c("Numeric", "Character", "Factor"),
                selected = "Numeric")
}

# switch the function to be applied within the server
selectFilterServer <- function(id) {
    moduleServer(id, function(input, output, session) {
        eventReactive(input$filter, {
            switch(input$filter,
               "Numeric" = is.numeric,
               "Character" = is.character,
               "Factor" = is.factor
               )
        })
    })
}
```

Now we can update the `selectVarServer` to take on an additional `filter` argument, and change the update function to not only observe when the `data` reactive changes but also our new `filter` widget changes. Lastly we pass in the filter reactive to the `find_vars` function.

```
selectVarServer <- function(id, data, filter) { # filter argument
    moduleServer(id, function(input, output, session) {
        observeEvent({
            data()
            filter() #observe changes in filter reactive
            }, {
            updateSelectInput(session, "var", choices = find_vars(data(), filter())) #
        })
        reactive(data()[[input$var]])
    })
}
```

Putting it together, we add our new module to the UI and server, and by saving the result of the `selectFilterServer` to `filt` we can pass that to the `selectVarServer`

```
selectVarApp <- function() {
    ui <- fluidPage(
        datasetInput("data", is.data.frame),
        # call the new filter UI
        selectFilterInput("filter"),
        selectVarInput("var"),
        verbatimTextOutput("out")
    )
    server <- function(input, output, session) {
        data <- datasetServer("data")
        # store the filtering function as a reactive
        filt <- selectFilterServer("filter")
        # pass the reactive to the select module
        var <- selectVarServer("var", data, filter = filt)
        output$out <- renderPrint(var())
    }

    shinyApp(ui, server)
}
```

Exercise 15.6.3

The following code defines output and server components of a module that takes a numeric input and produces a bulleted list of three summary statistics. Create an app function that allows you to experiment with it. The app function should take a data frame as input, and use `numericVarSelectInput()` to pick the variable to summarise.

```
summaryOuput <- function(id) {
  tags$ul(
    tags$li("Min: ", textOutput(NS(id, "min"), inline = TRUE)),
    tags$li("Max: ", textOutput(NS(id, "max"), inline = TRUE)),
    tags$li("Missing: ", textOutput(NS(id, "n_na"), inline = TRUE))
  )
}

summaryServer <- function(id, var) {
  moduleServer(id, function(input, output, session) {
    rng <- reactive({
      req(var())
```

```
      range(var(), na.rm = TRUE)
    })

    output$min <- renderText(rng()[[1]])
    output$max <- renderText(rng()[[2]])
    output$n_na <- renderText(sum(is.na(var())))
  })
}
```

Solution.

Solution

We only need to add the code above to the `selectVarApp()` example in the book, and adapt the app code to include the `summaryOutput` instead of the `verbatimTextOutput`, and on the server side pass `var` to the `summaryServer` function instead of to the text output.

```
selectVarApp <- function(filter = is.numeric) {
    ui <- fluidPage(
        datasetInput("data", is.data.frame),
        selectVarInput("var"),
        summaryOutput("summary")
    )
    server <- function(input, output, session) {
        data <- datasetServer("data")
        var <- selectVarServer("var", data, filter = filter)
        summaryServer("summary", var)
    }

    shinyApp(ui, server)
}

selectVarApp()
```

Exercise 15.6.4

The following module input provides a text control that lets you type a date in ISO8601 format (yyyy-mm-dd). Complete the module by providing a server function that uses output$error to display a message if the entered value is not a valid date. The module should return a Date object for valid dates. (Hint: use strptime(x, “%Y-%m-%d”) to parse the string; it will return NA if the value isn’t a valid date.)

```
ymdDateUI <- function(id, label) {
  label <- paste0(label, " (yyyy-mm-dd)")

  fluidRow(
    textInput(NS(id, "date"), label),
    textOutput(NS(id, "error"))
  )
}
```

Solution.

Solution

We create a `ymdDateServer` function that renders the error if `strptime(input$date, "%Y-%m-%d")` is `NA`.

```
ymdDateServer <- function(id, label) {
    moduleServer(id, function(input, output, session) {
        output$error <- renderText({
            print(input$date)
            print(strptime(input$date, "%Y-%m-%d"))
            if (!is.na(strptime(input$date, "%Y-%m-%d")) | input$date == "") {
                NULL
            } else {
               "Entered value is not a proper date"
            }

        })
    })
}
```

We put the `UI` and `Server` code together in the `ymdApp` function below:

```
ymdApp <- function(filter = is.numeric) {
    ui <- fluidPage(
        ymdDateUI("ymd", "Time")
    )
    server <- function(input, output, session) {
        ymdDateServer("ymd")
    }
    shinyApp(ui, server)
}
ymdApp()
```

Exercise 15.6.5

In `radioExtraServer()`, return a list that contains both the value and whether or not it came from other.

Solution.

Solution

We can adapt the reactive we return from `radioExtraServer` to return both the reactive and whether it came from the primary button choices or not as a list.

```
radioExtraServer <- function(id) {
    moduleServer(id, function(input, output, session) {
        observeEvent(input$other, ignoreInit = TRUE, {
            updateRadioButtons(session, "primary", selected = "other")
        })

        selected <- reactive({
            if (input$primary == "other") {
                input$other
            } else {
                input$primary
            }
        })

        # return the selected reactive inside a list
        # adding whether it came from primary or not
        list(selected =
                 reactive({
                     if (input$primary == "other") {
                         input$other
                     } else {
                        input$primary
                     }
                 }),
             primary =
                 reactive(input$primary != "other")
             )
    })
}
```

In doing so, we need to adapt the `radioExtraApp` code to return `extra$selected()` rather than `extra`.

```
radioExtraApp <- function(...) {
    ui <- fluidPage(
        radioExtraUI("extra", NULL, ...),
        textOutput("value")
    )
    server <- function(input, output, server) {
        extra <- radioExtraServer("extra")
        output$value <- renderText({
            paste0("Selected: ", extra$selected())
        })
    }

    shinyApp(ui, server)
}
radioExtraApp(c("a", "b", "c"))
```

Exercise 15.6.6

In `wizardServer()` verify that the namespacing has been set up correctly by using two or more wizards in a single add, and checking that you can navigate through each wizard independently.

Solution.

Solution

Chapter 16

Packages

There are no exercises in this chapter.

Chapter 17

Testing

There are no exercises in this chapter.

Chapter 18

Safety

There are no exercises in this chapter.

Chapter 19

Performance

There are no exercises in this chapter.

Chapter 20

Why reactivity?

There are no exercises in this chapter.

Chapter 21

Dependency Tracking

There are no exercises in this chapter.

Chapter 22

Scoping

There are no exercises in this chapter.

Chapter 23

Reactive Components

There are no exercises in this chapter.

Chapter 24

Advanced UI

There are no exercises in this chapter.