HW7 Shiny Assignment

Taha Ababou

2024-11-15

Shiny Application

1. What is the difference between Hadley_1 and Hadley_2? Use the functions Katia showed last Wednesday to investigate the difference.

Hadley_1

```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices =
   ↪  ls("package:datasets")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)


server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })

  output$table <- renderTable({
    dataset <- get(input$dataset, "package:datasets")
    dataset
  })
}
```

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

Hadley_2

```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices =
   ↪  ls("package:datasets")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)

server <- function(input, output, session) {
  # Create a reactive expression
  dataset <- reactive({
    get(input$dataset, "package:datasets")
  })

  output$summary <- renderPrint({
    # Use a reactive expression by calling it like a function
    summary(dataset())
  })

  output$table <- renderTable({
    dataset()
  })
}
shinyApp(ui, server)
```

The key difference between `Hadley_1` and `Hadley_2` lies in how the dataset is retrieved. In `Hadley_1`, the dataset is retrieved separately in both `renderPrint` and `renderTable` using `get()`, resulting in repeated computations and inefficiency. In contrast, `Hadley_2` uses a reactive expression (`dataset <- reactive({...})`) to retrieve the dataset once and reuse it by calling `dataset()` in both outputs, making it more efficient and maintainable. `Hadley_2` aligns with Shiny's reactive programming best practices by centralizing the dataset retrieval, reducing redundancy, and improving performance, especially if multiple outputs depend on the same dataset.

2. Prepare Chapters 2-4 from Mastering Shiny. Complete in submit the homework in sections 2.3.5, 3.3.6, and 4.8.

2.3.5 Exercises

1. Which of `textOutput()` and `verbatimTextOutput()` should each of the following render functions be paired with?

   a. `renderPrint(summary(mtcars))` should be paired with `verbatimTextOutput()`

   b. `renderText("Good morning!")` should be paired with `textOutput()`

   c. `renderPrint(t.test(1:5, 2:6))` should be paired with `verbatimTextOutput()`

   d. `renderText(str(lm(mpg ~ wt, data = mtcars)))` should be paired with `verbatimTextOutput()`

2. Re-create the Shiny app from Section 2.3.3, this time setting height to 300px and width to 700px. Set the plot "alt" text so that a visually impaired user can tell that its a scatterplot of five random numbers.

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

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

shinyApp(ui, server)
```

3. Update the options in the call to `renderDataTable()` below so that the data is displayed, but all other controls are suppressed (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")
)
```

```
`shiny::dataTableOutput()` is deprecated as of shiny 1.8.1.
Please use `DT::DTOutput()` instead.
Since you have a suitable version of DT (>= v0.32.1), shiny::dataTableOutput() will auto
```

```
If this happens to break your app, set `options(shiny.legacy.datatable = TRUE)` to get t
See <https://rstudio.github.io/DT/shiny.html> for more information.
```

```
server <- function(input, output, session) {
  output$table <- renderDataTable(mtcars, options = list(pageLength = 5,
 ↪  ordering = FALSE, searching = FALSE))
}
```

4. Alternatively, read up on reactable, and convert the above app to use it instead.

```
ui <- fluidPage(
  reactableOutput("table")
)

server <- function(input, output) {
  output$table <- renderReactable({ reactable(mtcars) })
}

shinyApp(ui, server)
```

3.3.6 Exercises

1. Given this UI:

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

Fix the simple errors found in each of the three server functions below. First try spotting the problem just by reading the code; then run the code to make sure you've fixed it.

```
# fix: switch input and output
# fix: add `input$` to `name`

server1 <- function(input, output, server) {
  input$greeting <- renderText(paste0("Hello ", name))
}
```

```
# fix: make greeting a reactive and add `()`

server2 <- function(input, output, server) {
```

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

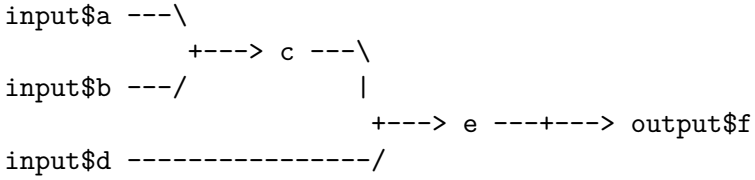
```
# fix: spelling error: "greting" -> "greeting"
# fix: no renderText()

server3 <- function(input, output, server) {
  output$greeting <- renderText(paste0("Hello ", input$name))
}
```

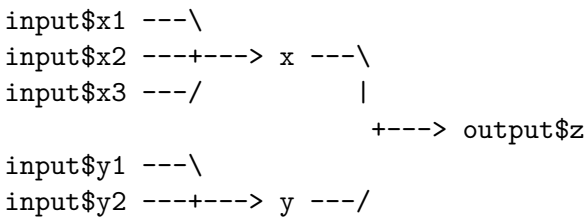
2. Draw the reactive graph for the following server functions:

```
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)
}
```

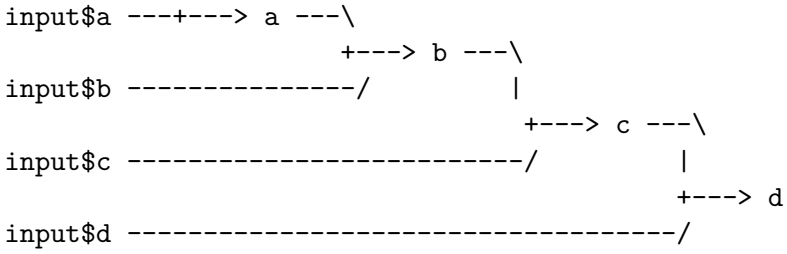
Server 1 Reactive Graph



Server 2 Reactive Graph



Server 3 Reactive Graph



3. Why will this code fail?

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

The code fails because the reactive object `range` conflicts with the base R function `range()`. When `range(var(), na.rm = TRUE)` is called, R tries to use the reactive `range` as a function, which is not allowed, causing an error. To fix this, we need to rename the reactive object to avoid the naming conflict, such as `range_val`, and explicitly use `base::range` to ensure the correct function is called, like so: `range_val <- reactive(base::range(var(), na.rm = TRUE))`.

Why are `range()` and `var()` bad names for reactive?

`range()` and `var()` are bad reactive names because they conflict with base R functions, causing confusion and errors when R tries to use the reactive objects as functions. Descriptive, unique names avoid this issue.

4.8 Exercises

1. Draw the reactive graph for each app.

Prototype App

```
input$code
    |
    v
selected() ---> output$diag
    |           output$body_part
    |           output$location
    v
summary() ---> output$age_sex
```

Rate vs. Count App

```
input$code
    |
    v
selected() ---> output$diag
    |           output$body_part
    |           output$location
    v
summary() ---> output$age_sex
    ^
    |
input$y
```

Narrative App

```
input$code
    |
    v
selected() ---> output$diag
    |           output$body_part
    |           output$location
    |           output$narrative
    v
summary() ---> output$age_sex
    ^
    |
input$y

input$story ---> output$narrative
```

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

```
# Code Attribution:
 ↪  https://mastering-shiny.org/basic-case-study.html#exercises-4

dir.create("neiss")
```

```
Warning in dir.create("neiss"): 'neiss' already exists
```

```
#> Warning in dir.create("neiss"): 'neiss' already exists
download <- function(name) {
  url <- "https://raw.github.com/hadley/mastering-shiny/main/neiss/"
  download.file(paste0(url, name), paste0("neiss/", name), quiet = TRUE)
}
download("injuries.tsv.gz")
download("population.tsv")
download("products.tsv")
```

```
injuries <- vroom::vroom("neiss/injuries.tsv.gz", show_col_types = FALSE)
products <- vroom::vroom("neiss/products.tsv", show_col_types = FALSE)
population <- vroom::vroom("neiss/population.tsv", show_col_types = FALSE)
```

```
# Original code (using fct_infreq)
injuries %>%
  mutate(diag = fct_lump(fct_infreq(diag), n = 5)) %>%
  group_by(diag) %>%
  summarise(n = as.integer(sum(weight)))
```

```
# A tibble: 6 x 2
  diag                      n
  <fct>                 <int>
1 Other Or Not Stated   1806436
2 Fracture              1558961
3 Laceration            1432407
4 Strain, Sprain        1432556
5 Contusion Or Abrasion 1451987
6 Other                 1929147
```

```
# Modified code (using fct_lump)
injuries %>%
```

```
  mutate(diag = fct_infreq(fct_lump(diag, n = 5))) %>%
  group_by(diag) %>%
  summarise(n = as.integer(sum(weight)))
```

```
# A tibble: 6 x 2
  diag                      n
  <fct>                 <int>
1 Other                 1929147
2 Other Or Not Stated   1806436
3 Fracture              1558961
4 Laceration            1432407
5 Strain, Sprain        1432556
6 Contusion Or Abrasion 1451987
```

Flipping fct_infreq() and fct_lump() changes the order of operations, affecting how "Other" is treated and ordered. When fct_infreq() is applied first, levels are ordered by frequency before lumping, resulting in "Other" appearing last because it is the least frequent after reordering. When fct_lump() is applied first, less frequent levels are lumped into "Other" immediately, and fct_infreq() then reorders levels by their final frequencies, placing "Other" first if it becomes the most frequent. This flip shifts "Other" in the summary table.

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

```
# UI
ui <- fluidPage(
  titlePanel("Summary Table"),
  sidebarLayout(
    sidebarPanel(
      numericInput("n_rows", "Number of Rows:", value = 5, min = 1, step = 1)
       ↪  # Input for number of rows
    ),
    mainPanel(
      tableOutput("summaryTable") # Output the summary table
    )
  )
)

# Server
server <- function(input, output, session) {
  injuries <- vroom::vroom("neiss/injuries.tsv.gz", show_col_types = FALSE)
```

```
  output$summaryTable <- renderTable({
    injuries %>%
      mutate(diag = fct_lump(fct_infreq(diag), n = input$n_rows - 1)) %>% #
       ↪  Lump into n - 1 levels
      group_by(diag) %>%
      summarise(n = as.integer(sum(weight, na.rm = TRUE))) %>%
      arrange(desc(n)) %>%
      slice_head(n = input$n_rows) # Limit to exactly n_rows
  })
}

# Run the app
shinyApp(ui, server)
```

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

```
# Utility function to get top N categories
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(
  titlePanel("Injury Explorer"),
  sidebarLayout(
    sidebarPanel(
      selectInput("product", "Select Product:",
                  choices = c("All", setNames(products$prod_code,
                   ↪  products$title))),
      numericInput("age", "Filter by Age:", value = NA, min = 0),
      selectInput("sex", "Select Sex:", choices = c("All", "male",
       ↪  "female")),
      numericInput("rows", "Number of Rows", min = 1, max = 10, value = 5),
      selectInput("y_axis", "Y Axis", choices = c("rate", "count"))
    ),
    mainPanel(
      fluidRow(
        column(4, tableOutput("diag")),
```

```
        column(4, tableOutput("body_part")),
        column(4, tableOutput("location"))
      ),
      plotOutput("injury_rate_plot"),
      fluidRow(
        column(2, actionButton("prev_story", "Previous story")),
        column(2, actionButton("next_story", "Next story")),
        column(8, textOutput("narrative"))
      )
    )
  )
)

server <- function(input, output, session) {
  # Reactive data with joined products
  injuries_with_products <- reactive({
    injuries %>%
      left_join(products, by = "prod_code")
  })

  # Filter data based on inputs
  filtered_data <- reactive({
    data <- injuries_with_products()
    if (input$product != "All") {
      data <- data %>% filter(prod_code == input$product)
    }
    if (!is.na(input$age)) {
      data <- data %>% filter(age == input$age)
    }
    if (input$sex != "All") {
      data <- data %>% filter(sex == input$sex)
    }
    data
  })

  # Dynamic maximum rows for tables
  observe({
    max_rows <- max(
      length(unique(filtered_data()$diag)),
      length(unique(filtered_data()$body_part)),
      length(unique(filtered_data()$location))
    )
```

```
  updateNumericInput(session, "rows", max = max_rows)
})

# Render top N diagnoses, body parts, and locations
table_rows <- reactive(input$rows - 1)

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

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

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

# Summary for injury rate plot
summary <- reactive({
  filtered_data() %>%
    count(age, sex, wt = weight) %>%
    left_join(population, by = c("age", "sex")) %>%
    mutate(rate = n / population * 100000)
})

output$injury_rate_plot <- renderPlot({
  if (input$y_axis == "count") {
    summary() %>%
      ggplot(aes(x = age, y = n, color = sex)) +
      geom_line() +
      labs(title = "Estimated Number of Injuries", y = "Count")
  } else {
    summary() %>%
      ggplot(aes(x = age, y = rate, color = sex)) +
      geom_line() +
      labs(title = "Injuries per 100,000 People", y = "Rate")
  }
})

# Narrative navigation
max_no_stories <- reactive({
```

```
    nrow(filtered_data())
  })

  story <- reactiveVal(1)

  observeEvent(input$next_story, {
    story((story() %% max_no_stories()) + 1)
  })

  observeEvent(input$prev_story, {
    story(((story() - 2) %% max_no_stories()) + 1)
  })

  output$narrative <- renderText({
    selected_data <- filtered_data()
    if (nrow(selected_data) > 0) {
      selected_data$narrative[story()]
    } else {
      "No narrative available for the selected filters."
    }
  })
}

shinyApp(ui, server)
```

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

```
server2 <- function(input, output, session) {
  # Reactive data with joined products
  injuries_with_products <- reactive({
    injuries %>%
      left_join(products, by = "prod_code")
  })

  # Filter data based on inputs
  filtered_data <- reactive({
    data <- injuries_with_products()
    if (input$product != "All") {
      data <- data %>% filter(prod_code == input$product)
    }
    if (!is.na(input$age)) {
```

```
    data <- data %>% filter(age == input$age)
  }
  if (input$sex != "All") {
    data <- data %>% filter(sex == input$sex)
  }
  data
})

# Dynamic maximum rows for tables
observe({
  max_rows <- max(
    length(unique(filtered_data()$diag)),
    length(unique(filtered_data()$body_part)),
    length(unique(filtered_data()$location))
  )
  updateNumericInput(session, "rows", max = max_rows)
})

# Render top N diagnoses, body parts, and locations
table_rows <- reactive(input$rows - 1)

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

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

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

# Summary for injury rate plot
summary <- reactive({
  filtered_data() %>%
    count(age, sex, wt = weight) %>%
    left_join(population, by = c("age", "sex")) %>%
    mutate(rate = n / population * 100000)
})

output$injury_rate_plot <- renderPlot({
```

```
    if (input$y_axis == "count") {
      summary() %>%
        ggplot(aes(x = age, y = n, color = sex)) +
        geom_line() +
        labs(title = "Estimated Number of Injuries", y = "Count")
    } else {
      summary() %>%
        ggplot(aes(x = age, y = rate, color = sex)) +
        geom_line() +
        labs(title = "Injuries per 100,000 People", y = "Rate")
    }
  })

  # Narrative navigation
  max_no_stories <- reactive({
    nrow(filtered_data())
  })

  story <- reactiveVal(1)

  observeEvent(input$next_story, {
    story(ifelse(max_no_stories() == 0, 1, ((story() %% max_no_stories()) +
     ↪  1)))
  })

  observeEvent(input$prev_story, {
    story(ifelse(max_no_stories() == 0, 1, ((story() - 2) %%
     ↪  max_no_stories()) + 1))
  })

  output$narrative <- renderText({
    selected_data <- filtered_data()
    if (nrow(selected_data) > 0) {
      selected_data$narrative[story()]
    } else {
      "No narrative available for the selected filters."
    }
  })
}

shinyApp(ui, server2)
```

I made the narratives circular using modular arithmetic so that navigating forward from the last narrative loops back to the first, and going backward from the first loops to the last. I used `((story() %% max_no_stories()) + 1)` for forward navigation and `((story() - 2) %% max_no_stories()) + 1` for backward navigation. To handle cases with no matching narratives, I defaulted the story index to `1` to avoid errors, ensuring smooth and seamless navigation through all available narratives.