Shiny II - Excercises

Exercise1

Draw the reactive graph for the following server functions:

```r
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)
}
```

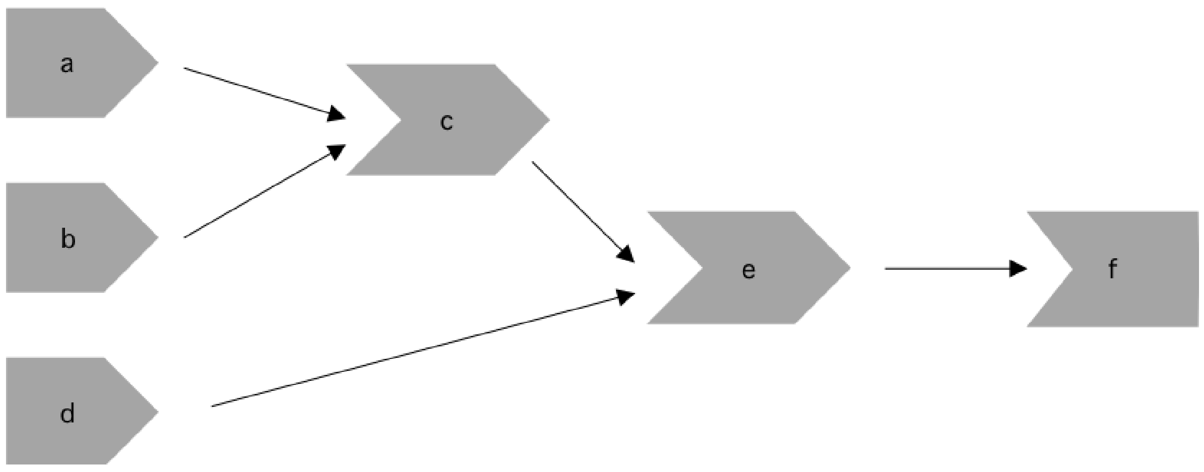
Solution

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`

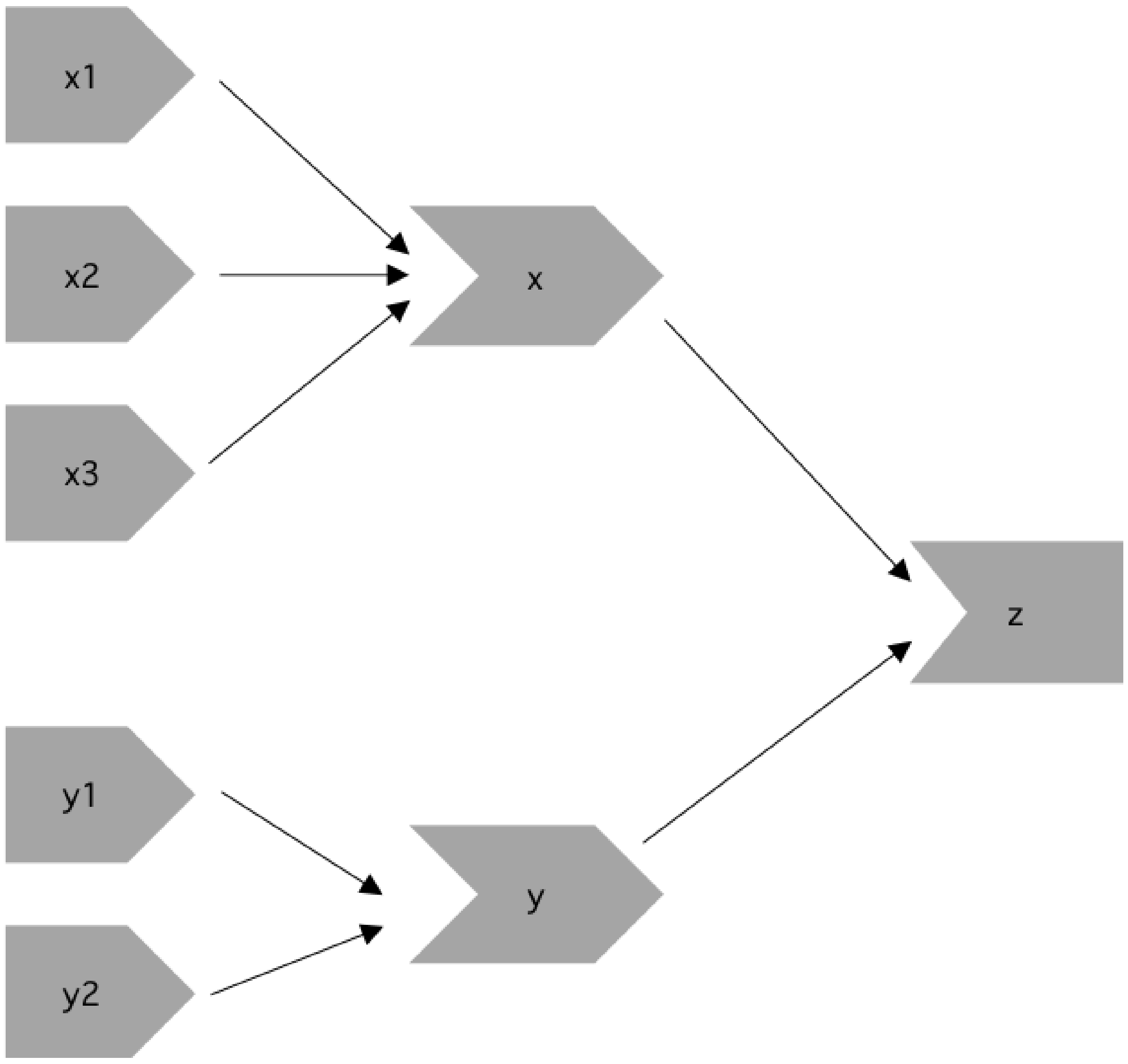
Inputs `input$a` and `input$b` are used to create `c()`, which is combined with `input$d` to create `e()`. The output depends only on `e()`.



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`

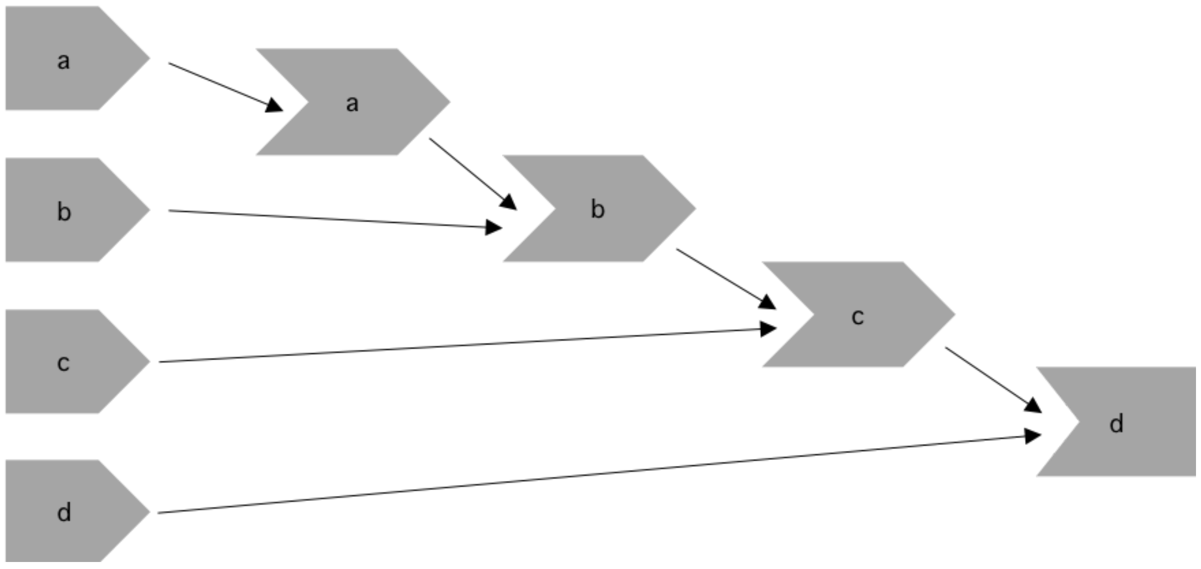
Inputs `input$y1` and `input$y2` are needed to create the reactive `y()`. In addition, inputs `input$x1`, `input$x2`, and `input$x3` are required to create the reactive `x()`. The output depends on both `x()` and `y()`.



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`.



Exercise 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

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 3

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

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

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

Solution

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

```r
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 4

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

```r
ui <- fluidPage(
  titlePanel("Central limit theorem"),
  sidebarLayout(
    sidebarPanel(
      numericInput("m", "Number of samples:", 2, min = 1, max = 100)
    ),
    mainPanel(
      plotOutput("hist")
    )
  )
)
server <- function(input, output, session) {
  output$hist <- renderPlot({
    means <- replicate(1e4, mean(runif(input$m)))
    hist(means, breaks = 20)
  }, res = 96)
}
```

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

Exercise 5

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

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)
```

Exercise 6

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

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 7

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

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 8

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

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 9

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

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.

```r
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)
```

Exercise 10

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

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 11

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

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 12

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

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)

```

```r
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 13

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

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)
```