# Shiny I - Excercises

## Excercise 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. Think about which lines you'll use and then copy and paste them into the right place in a Shiny app.

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

### 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".

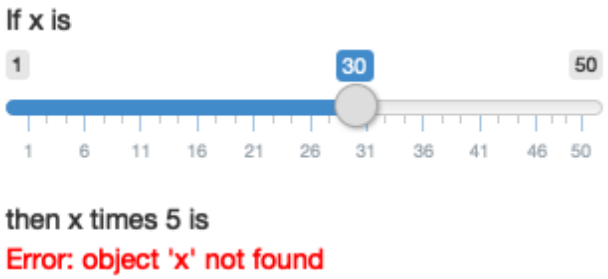
```
library(shiny)

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)
```

## Excercise 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:

```
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({
    x * 5
  })
}
shinyApp(ui, server)
```

But unfortunately it has an error:




Can you help them find and correct the error?

### Solution

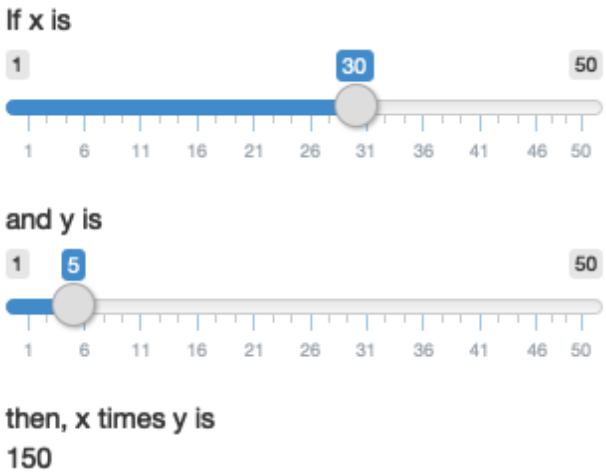
The error here arises because on the server side we need to write `input$x` rather than `x`. By writing `x`, we are looking for element `x` which doesn't
exist in the Shiny environment; `x` only exists within the read-only object `input`.

```
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)
```

## Excercise 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

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)
```

## Excercise 4:

Take the following app which adds some additional functionality to the last app described in the last exercise. What's new? How could you reduce the amount of duplicated code in the app by using a reactive expression.

```
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) {
  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
  })
}

shinyApp(ui, server)
```

### 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)
```

## Excercise 5:

Consider the following app. 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

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.

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)
```

## Excercise 6:

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?


### 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")
```

## Excercise 7:

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


### 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")
)
```

## Excercise 8:

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 the range automatically.

### 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)
```

## Excercise 9:

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

### 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.

## Excercise 10:

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

1. `renderPrint(summary(mtcars))`
2. `renderText("Good morning!")`
3. `renderPrint(t.test(1:5, 2:6))`
4. `renderText(str(lm(mpg ~ wt, data = mtcars)))`

### Solution

We can have a look at the documentation:

> **Description**
>
> Render a reactive output variable as text within an application page. `textOutput()` is usually paired with `renderText()` and puts regular text in `<div>` or `<span>`; `verbatimTextOutput()` is usually paired with `renderPrint()` and provides fixed-width text in a `<pre>`.
>
> **Usage**

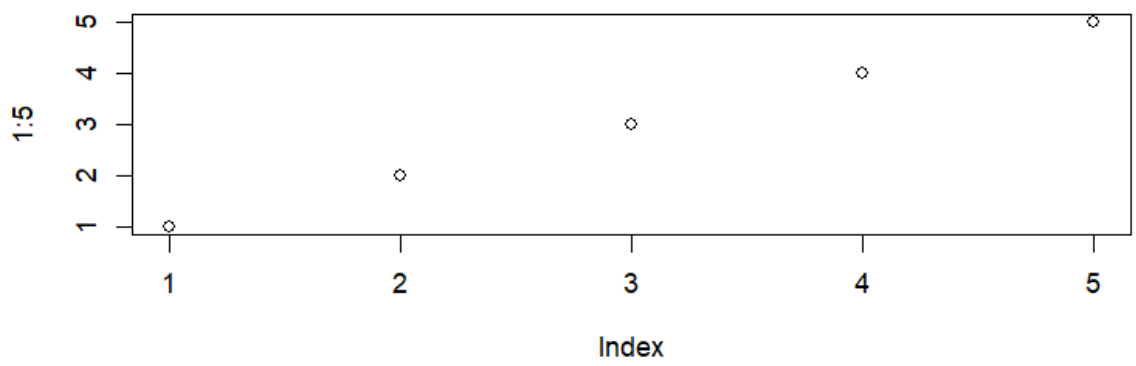
> textOutput(outputId, container = if (inline) span else div, inline = FALSE)
>
> verbatimTextOutput(outputId, placeholder = FALSE)

1. `renderPrint(summary(mtcars))` - should be paired with `verbatimTextOutput()`
2. `renderText("Good morning!")` - should be paired with `textOutput()`
3. `renderPrint(t.test(1:5, 2:6))` - should be paired with `verbatimTextOutput()`
4. `renderText(str(lm(mpg ~ wt, data = mtcars)))` - should be paired with `textOutput()`

## Excercise 11:

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

*Hint: plot(1:5)*




### 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)
```