Developing a Shiny App

- Sometimes the goal is for the viewer to interact with your visualization in some way. This allows the viewer to pick out variables or relationships of interest. Or to zero in on smaller components.
- We will begin by making interactive visualiztions by constructing a Shiny App. This will be accomplished entirely in R.
- One example is a military spending example here Military Spending. We were able to interact with the visualization to see how alterations in the response variable or graph type helped develop our understanding of military spending on a global level.
- Other Shiny Apps are available to explore at shiny.rstudio.com/gallery/.

Components of a Shiny app

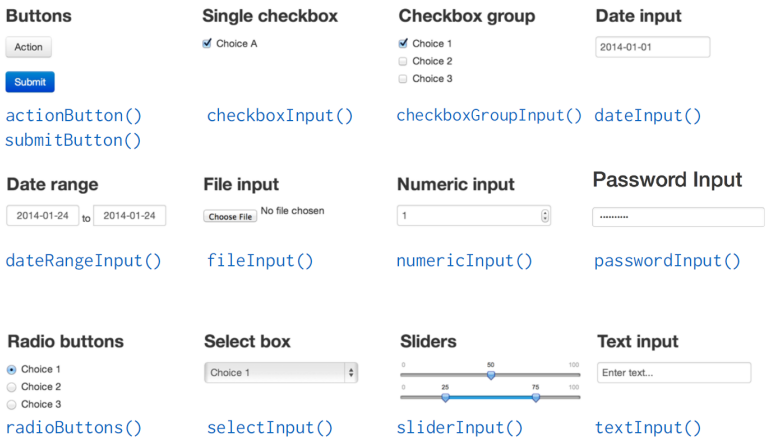
- Every Shiny app is maintained by a computer running R. In the beginning stage of development, that will be your personal computer or the lab computer. Once it is ready to publish, a web server can service the Shiny app.
- Shiny apps have two components
  - a user-interface script
  - a server script
- The user-interface (ui) script controls the layout and appearance of your app. It creates html code to create a webpage. It creates this code through R.
- The server script runs the R code that is used to create the Shiny app based on user-specified inputs.
- Every Shiny app should begin with the following template:

```
require(shiny)

ui<-fluidPage()

server<-function(input, output){}

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

- Build your app around your inputs and outputs.
- Add elements to your app as arguments to `fluidPage()`

```
ui<-fluidPage(
  # Input functions,
  # Output functions
)
```

- There are 12 different input functions. You can add buttons, checkboxes, date inputs, date ranges, file inputs, numeric inputs, password inputs, radio buttons, select boxes.



- Each of the input functions take the same basic syntax. They all start with inputId. These id's should be unique between different objects in your app. Second, the label argument will simply display appropriate text near the input object. Next, are input specific arguments. Let's begin by adding a slider to our webpage with `sliderInput`. Sliders require min, max, and value as arguments. There are many optional arguments you can add to customize your slider. In the example below, we just add `step`.

```
ui<-fluidPage(
  sliderInput(inputId="num",
              label="Choose a number",
              value=0,min=0,max=100,
              step=10
              )
)

server<-function(input, output){}

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

- Shiny provides a second family of functions called output functions. Output functions will place into your app different types of R output - plots, text, tables, etc.

Function	Inserts
dataTableOutput()	an interactive table
htmlOutput()	raw HTML
imageOutput()	an image
plotOutput()	plot
tableOutput()	table
textOutput()	text
uiOutput()	a Shiny UI element
verbatimTextOutput()	text

- Each output function must have an argument `outputId`.

```
ui<-fluidPage(
  sliderInput(inputId="num",
              label="Choose a number",
              value=0,min=0,max=100,
              step=10
```

```
              ),
  plotOutput(outputId="histogram")
)

server<-function(input, output){}

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

- The above code might feel a bit disappointing! It doesn't look like anything happened. Why is that?
- We need to use the server function to help us assemble inputs into outputs.
- There are three rules we need to follow:
  1. Save objects to display to `output$`
  2. Build objects to display with a `render` function. This comprises the 3rd family of functions that we use to build a Shiny app.

Function	Creates
renderDataTable()	an interactive table
renderImage()	an image
renderPlot()	a plot
renderPrint()	a block of printed output
renderTable()	table
renderText()	a character string
renderUI()	a Shiny UI element

  3. Use input values with `input$` to correspond with your inputId's. The input value changes whenever a user changes the input.
- Let's use the `diamonds` data set to plot a histogram to display the distribution of price.

```
ui<-fluidPage(
  sliderInput(inputId="num",
              label="Choose a number",
              value=100,min=100,max=1000,
              step=100
              ),
  plotOutput(outputId="histogram")
)

server<-function(input, output){
  output$histogram<-renderPlot({
    ggplot(data=diamonds)+
      geom_histogram(aes(x=price), binwidth=input$num, color="black", fill="blue")
  })
}

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

- Create a Shiny app that utilizes the 'diamonds' data and displays a scatter plot of price vs. carat based on a user-specified value of clarity. What type of input function makes sense for this?

- What if we want to have 'R' recognize a string as a variable name?
  - We need to use 'eval(as.symbol())' for this. (The alternative is to change the data from wide to long, which can be tricky when you have variables of different types). Here is an example of this in action:

```
require(shiny)
require(ggplot2)

ui<-fluidPage(
  radioButtons(inputId="criteria1",
               label="What would you like to predict?",
               choices=names(diamonds),
               selected="price"),
  radioButtons(inputId="criteria2",
               label="Pick an explanatory variable?",
               choices=names(diamonds),
               selected="depth"),
  plotOutput(outputId="scatterplot1")
)

server<-function(input, output){
  output$scatterplot1<-renderPlot({
    ggplot(data=diamonds)+
      geom_point(aes(x=eval(as.symbol(input$criteria2)),
                     y=eval(as.symbol(input$criteria1))))+
      xlab(input$criteria2)+
      ylab(input$criteria1)

  })
}

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

- How to have user select two values for the slider input.

```
ui<-fluidPage(
  sliderInput(inputId="range",
              label="Select the price range of interest",
              min=c(0,10000), max=c(10000,20000),
              value=c(0,20000)),
  plotOutput("Plot")
)

server<-function(input, output){
  output$Plot<-renderPlot({
    ggplot(data=filter(diamonds,
                       price<=input$range[2]&
                         price>=input$range[1]))+
    geom_point(aes(x=carat, y=price))
})}

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

Creating Tables in your Shiny App

- Creating tables is very similar to creating plots in a shiny app. Suppose you want to display mean diamond prices for each clarity level based on a user-specified carat range.

```
ui<-fluidPage(
  sliderInput(inputId="caratrange",
              label="Select a Carat Range",
              min=0,
              max=8,
              step=0.25,
              value=c(0,2)),
  tableOutput(outputId="Table1")
)
server<-function(input, output){
  output$Table1<-renderTable({
    tempdata<-filter(diamonds, carat>=input$caratrange[1]&carat<=input$caratrange[2])
    tempdata2<-summarise(group_by(tempdata, clarity), AvgPrice=mean(price))
    names(tempdata2)<-c("Clarity Level", "Average Price")
    tempdata2
  })
}

shinyApp(ui, server)
```

Incorporating Multiple rendered objects in your Shiny App

- You have had practice in this document creating multiple user-specified decisions. Notice there is a comma between all of the components in your ui. It is worth noting that there is no comma in between the different rendered objects in your server function.
- Let's return to the Obese dataset. Create an interactive graph that allows the user to select whether they want their map to display adult obesity proportion, adult overweight proportion, or children overweight proportion. In addition, have it display the top 5 states and the bottom 5 states in terms of the selected proportion.

```
ui<-fluidPage(
  radioButtons(inputId="criteria",
               label="Select a predictor",
               choices=names(Obese)[2:4],
               selected="AdultsObese"),
  plotOutput(outputId="plotastic", width="80%"),
  textOutput(outputId="HighestFiveLabel"),
  tableOutput(outputId="HighestFive"),
  textOutput(outputId="LowestFiveLabel"),
  tableOutput(outputId="LowestFive")
)

server<-function(input,output){
  output$plotastic<-renderPlot({
    names(Obese)[1]<-"region"
    Obese=mutate(Obese, region=tolower(region))
    all_states=map_data("state")
    ggplot()+
      geom_map(data=Obese, aes(map_id=region,
                               fill=eval(as.symbol(input$criteria))),
               map=all_states)+
      expand_limits(x=all_states$long, y=all_states$lat)+
      scale_fill_distiller("Proportion", palette = "RdPu", direction=1)+
      ggtitle(input$criteria)

  })
  output$HighestFiveLabel<-renderText({
    "States with the Top 5 Highest Rates"
  })
  output$HighestFive<-renderTable({
    arrange(Obese, desc(eval(as.symbol(input$criteria))))[1:5,c("State", input$criteria)]
  })
  output$LowestFiveLabel<-renderText({
    "States with the Lowest 5 Rates"
  })
  output$LowestFive<-renderTable({
    arrange(Obese, eval(as.symbol(input$criteria)))[1:5,c("State", input$criteria)]
  })

}


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

- How to take your Shiny App to the next level:
  - There are a series of 7 written tutorials that the makers of Shiny App have created. It may be useful to go through them if you'd like to take your interactive even further. http://shiny.rstudio.com/tutorial/lesson1/
  - You can change the theme of your Shiny App. You will need to install the package shinythemes and look for options here (https://rstudio.github.io/shinythemes/).
  - You can also change the layout of your Shiny App. Currently we have only done a simple layout. It may be useful to add tabs or have the ui portions be on the left and the output on the right. See different layouts here (https://shiny.rstudio.com/articles/layout-guide.html).
  - Currently, you are using your own machine to service your shiny app. To share your shiny app with the world, go here: http://shiny.rstudio.com/deploy/