Decision Sciences Learning Session

Interactive Apps with R Shiny

Bill Petti

Maritz Motivation

2020-06-22


Shiny

Agenda

- Walk through Shiny basics
- View more complex Shiny apps
- Live changes to a Shiny app

What is Shiny?

What is Shiny (more helpful edition)?

- Shiny is an R library that makes it very easy to build interactrive web applications without the need to directly code in JavaScript
- You can build an application completely using R, or layer in custom CSS, HTML, or JavaScript
- Apps can be run locally on your machine, deployed to shinyapps.io, or deployed to our own Shiny server for greater security

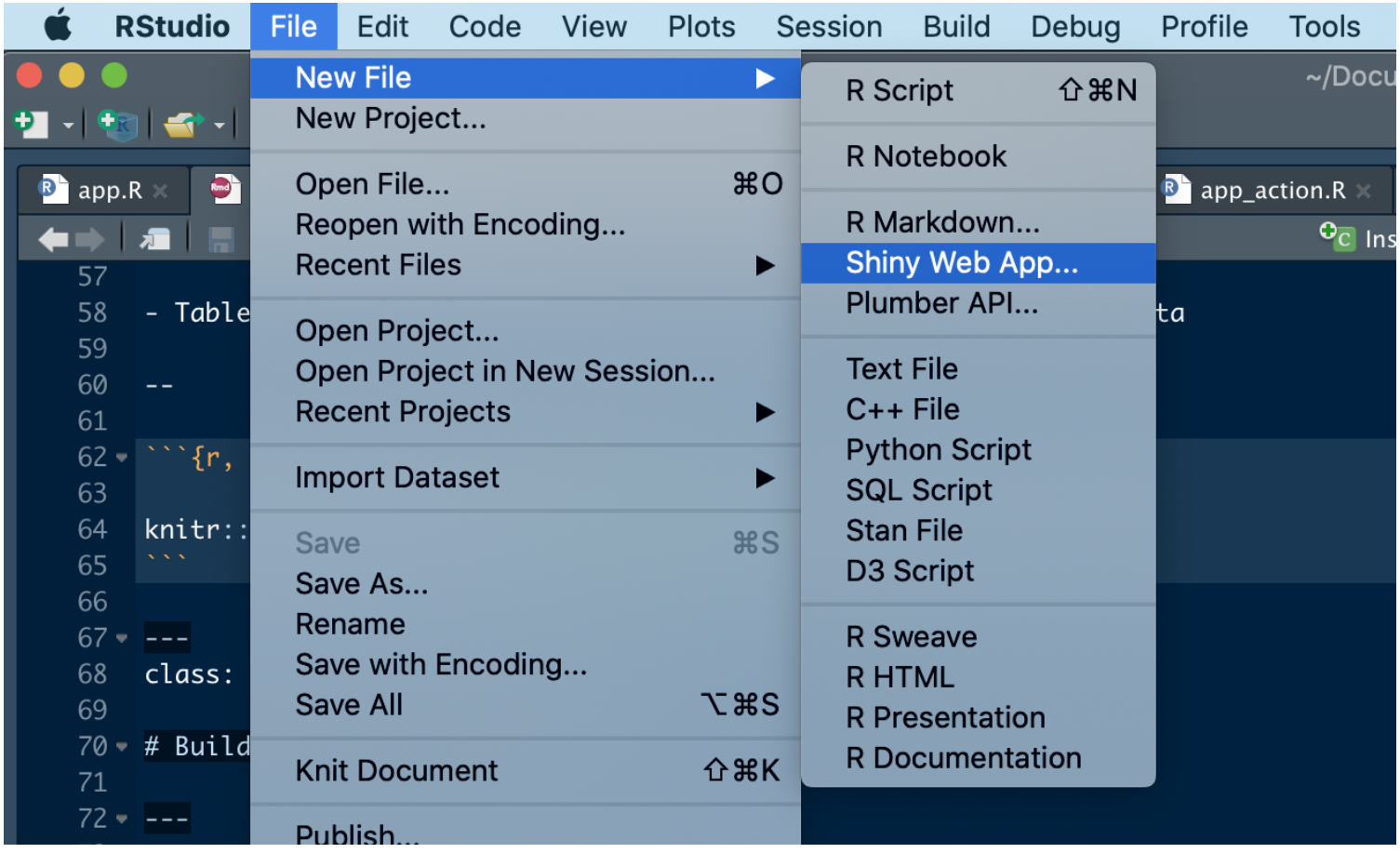
What is Shiny (more helpful edition)?

- Anything that you can code in R can be displayed and integrated into a Shiny app
- Tables, plots, downloading data, uploading data, transforming data

Building your first Shiny app

First Shiny App

Within RStudio, click File -> New File -> Shiny Web App

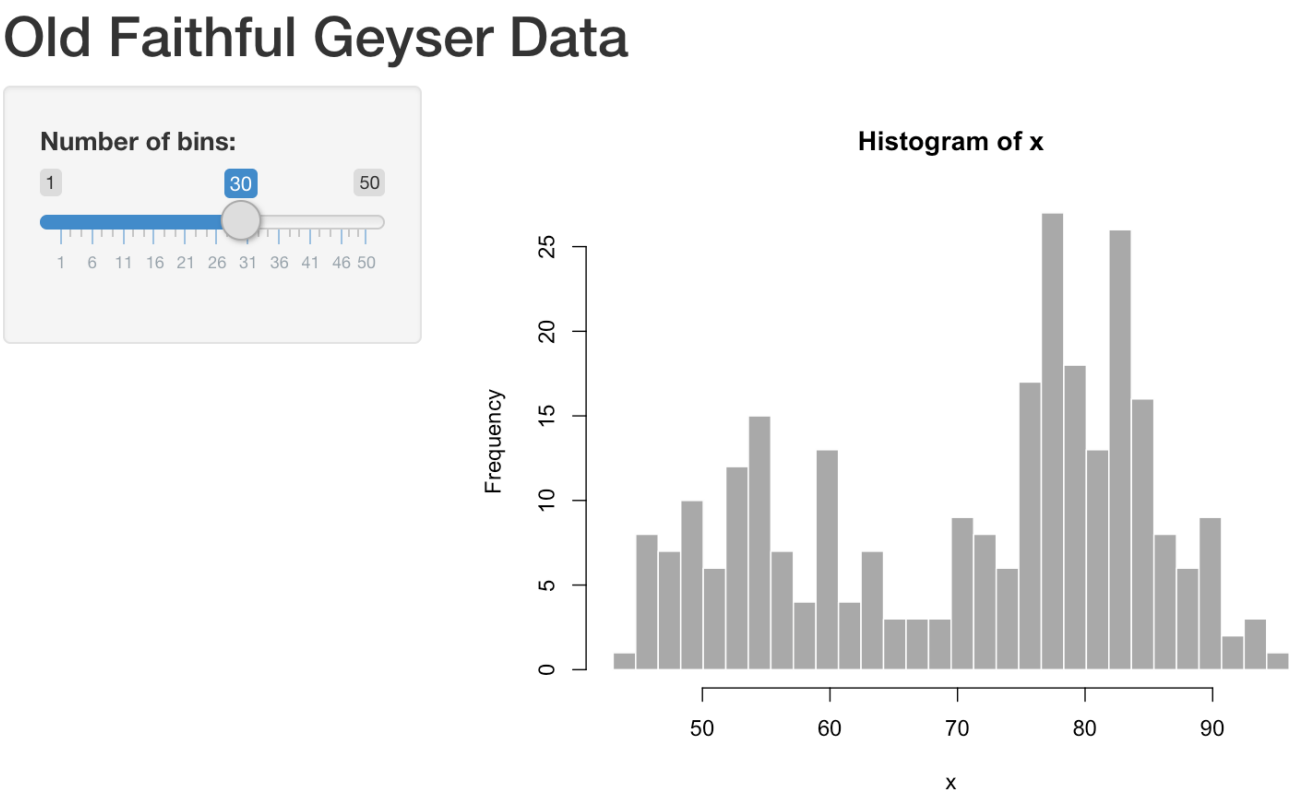


First Shiny App

- The file will provide a skeleton or template for fleshing out your app
- The file has two main parts
  - `ui`: This is where you code the user interface elements (what users see, interact with)
  - `server`: This is where you code the action and outputs of the app

First Shiny App

- This app allows users to adjust the bin size when plotting a histogram of the faithful waiting variable
- To run the app, click Run App in the upper right-hand corner of the app file



Uses of Shiny

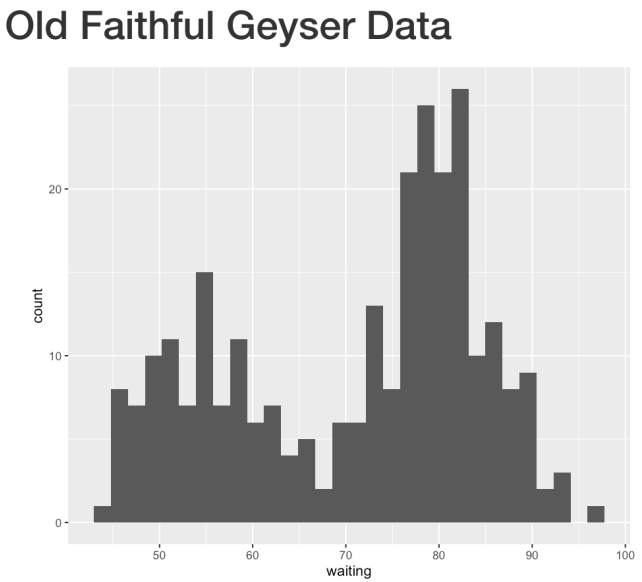
Shiny can be used for a number of broad needs:

- Display data or a simple plot without interactivity
- Display data with some interactivity (i.e. filtering, sorting)
- Generate different plots based on some interactivity
- Combine plots and tabular data
- Lots of other stuff...

Let's walk through examples of each of the examples above...

Apps that display plots

app_plot.R




```r
# display a plot of the data

library(shiny)
library(ggplot2)

# Define UI for application that draws a histogram
ui <- fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Show a table of the data
  mainPanel(
    plotOutput("faithful_plot")
  )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  plot <- reactive({

    ggplot(data = faithful,
           aes(x = waiting)) +
      geom_histogram()

  })

  output$faithful_plot <- renderPlot(expr = plot())

}

# Run the application
shinyApp(ui = ui, server = server)
```

app_datatable.R

Old Faithful Geyser Data

Show 10 entries Search:

	eruptions	waiting
1	3.6	79
2	1.8	54
3	3.333	74
4	2.283	62
5	4.533	85
6	2.883	55
7	4.7	88
8	3.6	85
9	1.95	51
10	4.35	85

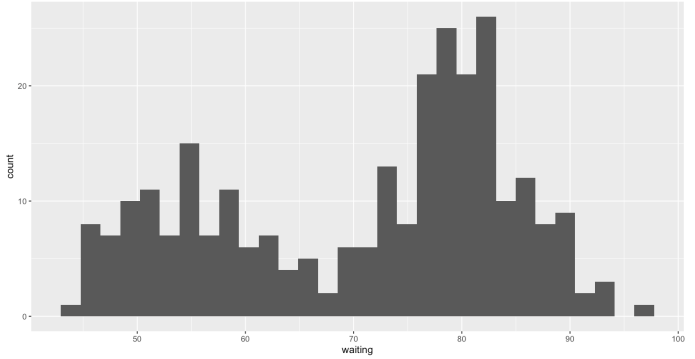
Showing 1 to 10 of 272 entries

Previous 1 2 3 4 5 … 28 Next

```r
# display a table of the data

library(shiny)
library(DT)

# Define UI for application that draws a histogram
ui <- fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Show a table of the data
    mainPanel(
      DTOutput("faithful_data")
    )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  output$faithful_data <- renderDT({
    DT::datatable(faithful)
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

app_combine_outputs.R

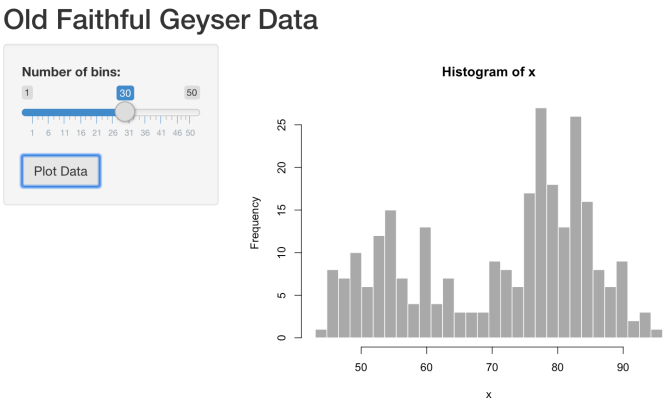
Old Faithful Geyser Data



Show 10 entries Search:

	eruptions	waiting
1	3.6	79
2	1.8	54
3	3.333	74
4	2.283	62
5	4.533	85
6	2.883	55
7	4.7	88
8	3.6	85
9	1.95	51

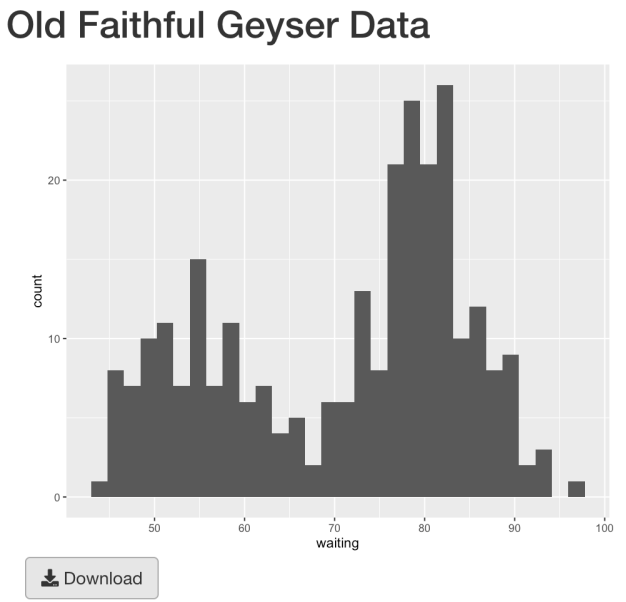
app_action.R




```r
# observe an event, then take action
# using action buttons

library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30),
      actionButton("button", "Plot Data")
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  bin_size <- eventReactive(input$button, {

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    return(bins)
  })

  output$distPlot <- renderPlot({
    # draw the histogram with the specified number of bins
    x <- faithful[, 2]
    hist(x, breaks = bin_size(), col = 'darkgray', border = 'white')
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```

Downloading Data and Plots

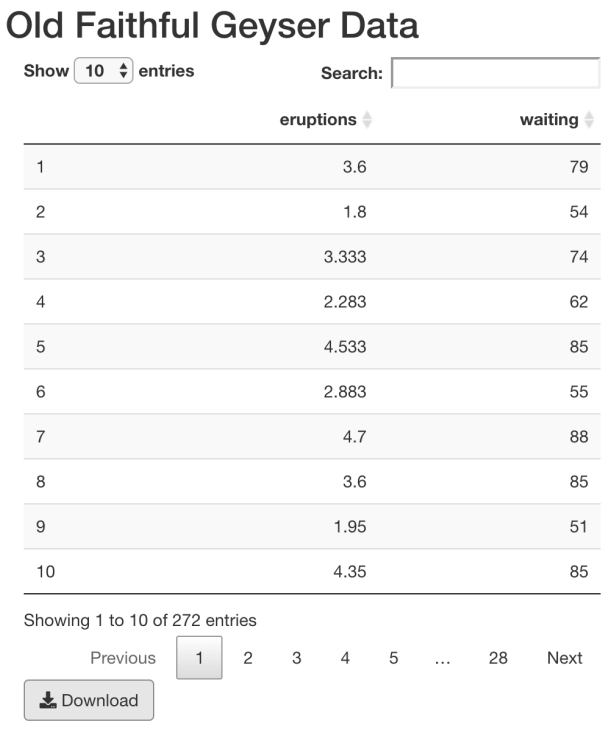
- You can allow users to download data or plots by using the `downloadButton()` function




```r
# display a plot of the data

library(shiny)
library(ggplot2)

# Define UI for application that draws a histogram
ui <- fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Show a table of the data
  mainPanel(
    plotOutput("faithful_plot"),
    downloadButton(outputId = "download_plot", label = "Download")
  )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  plot <- reactive({

    ggplot(data = faithful,
           aes(x = waiting)) +
      geom_histogram()

  })

  output$faithful_plot <- renderPlot(expr = plot())

  output$download_plot <- downloadHandler(
    filename = function(){
      paste('test', '.png', sep = '')
    },
    content = function(file){
      ggsave(file,
             plot=plot())
    }
  )
}

# Run the application
shinyApp(ui = ui, server = server)
```

Downloading Data and Plots

- It's a little different for `csv` files




```server.R
# display a table of the data and download the underlying data

library(shiny)
library(DT)

# Define UI for application that draws a histogram
ui <- fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Show a table of the data
  mainPanel(
    DTOutput("faithful_data"),
    downloadButton(outputId = "download_data", label = "Download")
  )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  output$faithful_data <- renderDT({
    DT::datatable(faithful)
  })

  output$download_data <- downloadHandler(

    filename = function() {
      paste("dataset", ".csv", sep = "")
    },
    content = function(file) {
      write.csv(faithful, file, row.names = FALSE)
    }


  )
}

# Run the application
shinyApp(ui = ui, server = server)
```

Inputs

- Most of the useful interactivity in Shiny is based on inputs.
- Inputs are typically generated through the `ui` and send information to the `server` so that some kind of actoin can take place
- The most common inputs are widgets that allow you to filter data (honestly, filtering data is about 90% of what you will use inputs for)

Inputs

- Here, we can filter the `faithful` data set by the length of the eruptions before binning and plotting

- We filter the data based on the same action button, and then use that filtered data set for binning and plotting (notice once you make an objection reactive it needs to be followed by `()` for the server to recognize it)

```r
# observe an event, then take action
# using action buttons

library(shiny)
library(tidyverse)

# Define UI for application that draws a histogram
ui <- fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30),
      sliderInput("eruption_time",
                  "Length of Eruption:",
                  min = 0,
                  max = max(faithful$eruptions),step = .05,
                  value = 0),
      actionButton("button", "Plot Data")
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  faithful_filtered <- eventReactive(input$button, {

    df <- faithful %>%
      filter(eruptions >= input$eruption_time)

    return(df)

  })

  bin_size <- eventReactive(input$button, {

    # generate bins based on input$bins from ui.R
    x    <- faithful_filtered()[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    return(bins)
  })

  output$distPlot <- renderPlot({
    # draw the histogram with the specified number of bins
    x <- faithful_filtered()[, 2]
    hist(x, breaks = bin_size(), col = 'darkgray', border = 'white')
  })

}

# Run the application
shinyApp(ui = ui, server = server)

```

Inputs

There are lots of different inputs you can use:

function	widget
`actionButton`	Action Button
`checkboxGroupInput`	A group of check boxes
`checkboxInput`	A single check box
`dateInput`	A calendar to aid date selection
`dateRangeInput`	A pair of calendars for selecting a date range
`fileInput`	A file upload control wizard
`helpText`	Help text that can be added to an input form
`numericInput`	A field to enter numbers
`radioButtons`	A set of radio buttons
`selectInput`	A box with choices to select from
`sliderInput`	A slider bar
`submitButton`	A submit button
`textInput`	A field to enter text

Conditional Inputs

- Sometimes you might want the available values for unputs to be conditional on selections from earlier filters
- Consider an app that displays data about MLB games
  - The users first selects a date from a calendar input
  - Then user can select from a list of games available on that day
- Conditional Inputs can be useful in a few ways
  - Too many potential options for a drop down
  - Save users from themselves (prevent them from selecting options that may not make sense)

Conditional Inputs

First, we create the calendar input in the `ui` section and create a `uiOutput` that will reference an object from the `server` names `matchup`

```
ui <- fluidPage(

  # Application title
  titlePanel("Conditional Filters"),

  dateInput("date",
            "Choose or enter a date (YYYY-MM-DD)",
            value = '2019-04-01'),
  uiOutput('matchups')

)
```

Conditional Inputs

Then, we render the ui `matchups` server-side, which can then be referenced in the `ui` by `uiOutput`

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

  output$matchups <- renderUI({

    games <- get_game_pks_mlb(input$date) %>%
      mutate(matchup =
               paste0(teams.away.team.name, " at ",
                      teams.home.team.name))

    matchups <- games %>%
      pull(matchup)

    selectizeInput("matchups_for_date",
                   "Select a game",
                   matchups,
                   width = 350)
  })
}
```

Conditional Inputs