Introduction to Shiny

Data Wrangling and Husbandry

04/13/2020

Shiny

Shiny is an RStudio product for web app development using R. There is lots of documentation at http://shiny.rstudio.com (some of which is used in these slides).

There are many examples of finished projects at www.rstudio.com/products/shiny/shiny-user-showcase/

Zev Ross has further guidance and 40(!) examples.

The `shiny` package itself includes 11 examples; you can see them with `runExample()`

The Shiny division of labor

Every Shiny app requires a machine, the server, running R. That machine has instructions about the R code and about the user interface (UI). The user interface be run from the same machine or another—it's really just a web page.

Example 01_hello

```
library(shiny)
runExample("01_hello")
```

Notice that

- The app is *reactive*, in that the analysis changes in reaction the user actions
- The code is divided into server and UI components
  - In this and other examples there are two separate files, but this is not a requirement—for smaller apps it's simpler to use just one file
- The code draws on both external objects, e.g., the `faithful` dataset, and objects determined by the UI

The Shiny template

All Shiny apps have the same basic template

```
library(shiny)
ui <- fluidPage()
server <- function(input, output, session) {}
shinyApp(ui = ui, server = server)
```

- Copy this text to a new file *that must be called app.R* and save the file in an otherwise *empty* directory (you can have other files need for your R code, however).
- RStudio should recognize that this is a Shiny app and offer a button called "Run App". You can run it, but it's rather boring.
- You can also use the File > New menu in RStudio to set up the directory and decide whether to go with one or two files.

A Shiny babynames app

- The loading of other packages, definitions of needed functions, setting of constants, can go at the top of the file
- You can set the page title with `titlePanel()`
- The `fluidPage()` function can take text strings and html code, although instead of writing, for example, `<h1>Header Text</h1>` you instead write `h1("Header Text")`.

```
fluidPage(
  titlePanel("Baby Names"),
  h1("Header Text"),
  "other text"
)
```

Page Format

Rather than having everything stacked up, we can get the side panel / main panel look by adding the code below *inside* the `fluidPage()` function

```
sidebarLayout(
  sidebarPanel("our inputs will go here"),
  mainPanel("the results will go here")
)
```

Our code so far

```
library(shiny)
library(babynames)



ui <- fluidPage(
  sidebarLayout(
    sidebarPanel("our inputs will go here"),
    mainPanel("the results will go here")
  ),
  titlePanel("Baby Names"),
  h1("Header Text"),
  "other text"
)
server <- function(input, output, session) {}
shinyApp(ui = ui, server = server)
```

Adding inputs to the interface

Users interact with Shiny apps via the user interface, so there are many ways to add inputs.

- Every input has two mandatory arguments, `inputID` and `label`. The values of `inputID` must be unique.
- The input functions go inside the `sidebarPanel()` function inside the `fluidPage()` function, separated by commas

```
sliderInput("yearInput", "Year", min = 1880, max = 2015,
            value = c(1900, 2000))
textInput("nameInput", "Name")
radioButtons("sexID", "Sex",
  choices = c("Female only", "Male only", "Both"), selecte
```

For now, we'll add a placeholder for the plot inside the `mainPanel()` function inside the `fluidPage()` function. Let's add a table of results, too.

```
plotOutput("main_plot")
tableOutput("results")
```

We now have

```
library(shiny)
library(babynames)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("yearInput", "Year", min = 1880, max = 20
        value = c(1900, 2000)),
      textInput("nameInput", "Name"),
      radioButtons("sexID", "Sex",
        choices = c("Female only", "Male only", "Both"), se
    mainPanel(
      plotOutput("main_plot"),
      tableOutput("results")
    )
  ),
  titlePanel("Baby Names")
)
```

The server side function

There are three rules for writing the output code

- Save the output object into the output list
- Build the object with a `render*` function, where `*` is the type of output
- Access input values using the input list

Example using the first two rules

Replace the empty server function with, say,

```
server <- function(input, output, session) {
  output$main_plot <- renderPlot({
    reduced_df <- filter(babynames,
                         name == "Leslie",
                         year >= 1920 & year <= 2010,
                         sex %in% c("F", "M")
                        )
    ggplot(data = reduced_df,
           aes(year, n, colour = sex)) +
      geom_line() + ggtitle("Leslie")
  })
}
```

Notice that

- the plot is saved as `output$main_plot`, which is saving the output object into the output list
- that the list element `main_plot` is one that has an inputID in the `ui` function
- that the `renderPlot` function is wrapped around our actual plotting function
- that the plot is not interactive

Making the page reactive

We can replace the `year >= 1920` part of the code with `year >= input$yearInput[1]` to make the lower limit of the plot depend on the input, and similarly with `year <= input$yearInput[2]`.

```
  output$main_plot <- renderPlot({
    reduced_df <- filter(
      babynames,
      name == "Leslie",
      year >= input$yearInput[1] & year <= input$yearInput
      sex %in% c("F", "M")
   )
    ggplot(data = reduced_df,
           aes(year, n, colour = sex)) +
      geom_line() + ggtitle("Leslie")
  })
```

Did you notice that the creation of the data frame `reduced_df` is inside `renderPlot()`. Every reactive element has to be inside a reactive context such as `renderPlot()`.

Tuning up the plot

```
output$main_plot <- renderPlot({
    sex_vec <- switch(input$sexID,
      `Female only` = "F",
      `Male only` = "M",
      Both = c("F", "M")
    )
    reduced_df <- filter(
      babynames,
      name == input$nameInput,
      year >= input$yearInput[1] & year <= input$yearInput
      sex %in% sex_vec
   )
    ggplot(data = reduced_df,
           aes(year, n, colour = sex)) +
      geom_line() + ggtitle(input$nameInput)
  })
```

Adding the table

Adding the table is very similar to what we did to add the plot

```
output$results <- renderTable({
     sex_vec <- switch(input$sexID,
      `Female only` = "F",
      `Male only` = "M",
      Both = c("F", "M")
    )
    reduced_df <- filter(
      babynames,
      name == input$nameInput,
      year >= input$yearInput[1] & year <= input$yearInput
      sex %in% sex_vec
   )
  reduced_df
})
```

Use of `reactive()`

The current code is redundant, but `input$*` variables can only be used inside a "reactive context". Defining the `reduced_df` data frame inside the `reactive()` function will do the trick.

```
reduced_df <-
  reactive({
  sex_vec <- switch(input$sexID,
      `Female only` = "F",
      `Male only` = "M",
      Both = c("F", "M")
    )
    reduced_df <- filter(
      babynames,
      name == input$nameInput,
      year >= input$yearInput[1] & year <= input$yearInput
      sex %in% sex_vec
   )
})
output$results <- renderTable({
  reduced_df
})
```

However, this is a function, so `reduced_df` will have to be replaced with `reduced_df()`

The final server function

```
server <- function(input, output, session) {
  reduced_df <- reactive({
    sex_vec <- switch(input$sexID,
                      `Female only` = "F",
                      `Male only` = "M",
                      Both = c("F", "M")
    )
    filter(
      babynames,
      name == input$nameInput,
      year >= input$yearInput[1] & year <= input$yearInput
      sex %in% sex_vec
    )
  })
  output$main_plot <- renderPlot({
    ggplot(data = reduced_df(),
           aes(year, n, colour = sex)) +
      geom_line() + ggtitle(input$nameInput)
```

function	creates
renderDataTable()	An interactive table
renderImage()	An image (saved as a link to a source file)
renderPlot()	A plot
renderPrint()	A code block of printed output
renderTable()	A table
renderText()	A character string
renderUI()	a Shiny UI element

`render*()`

- Always save the result to `output$*`
- `render*()` functions make objects to display
- When notified that it is invalid, the object created by a `render*()` function will rerun the *entire block of code* associated with it

`reactive()`

- call a reactive expression like a function
- reactive expressions cache their values
- useful for reading reactive values (e.g., `input$*`), evaluating expressions, and making the results available to other server functions—principally `render*()` functions

Reduce repetition

Place code where it will be re-run as little as necessary

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num", label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

server <- function(input, output) {

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}

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

Code outside the server function will be run once per R session (worker)

Code inside the server function will be run once per end user (connection)

Code inside a reactive function will be run once per reaction (e.g. many times)