Shiny Lecture 2 (Shinier)

Data Products

Brian Caffo, Jeff Leek, Roger Peng
Johns Hopkins Bloomberg School of Public Health

Shiny revisited

- In the last lecture, we covered basic creation of Shiny applications
- If you tried it and are like most, you had an easy time with `ui.R` but a harder time with `server.R`
- In this lecture, we cover some more of the details of shiny
- Since writing the last lecture, a more detailed tutorial has been created that is worth checking out (http://shiny.rstudio.com/tutorial/)

Details

- Code that you put before `shinyServer` in the `server.R` function gets called once when you do `runApp()`
- Code inside the unnamed function of `shinyServer(function(input, output){` but not in a reactive statement will run once for every new user (or page refresh)
- Code in reactive functions of `shinyServer` get run repeatedly as needed when new values are entered (reactive functions are those like `render*`)

Experiment (code in the slidify document)

`ui.R`

```
shinyUI(pageWithSidebar(
  headerPanel("Hello Shiny!"),
  sidebarPanel(
      textInput(inputId="text1", label = "Input Text1"),
      textInput(inputId="text2", label = "Input Text2")
  ),
  mainPanel(
      p('Output text1'),
      textOutput('text1'),
      p('Output text2'),
      textOutput('text2'),
      p('Output text3'),
      textOutput('text3'),
      p('Outside text'),
      textOutput('text4'),
      p('Inside text, but non-reactive'),
      textOutput('text5')
  )
))
```

`server.R` Set `x <- 0` before running

```
library(shiny)
x <<- x + 1
y <<- 0

shinyServer(
  function(input, output) {
    y <<- y + 1
    output$text1 <- renderText({input$text1})
    output$text2 <- renderText({input$text2})
    output$text3 <- renderText({as.numeric(input$text1)+1})
    output$text4 <- renderText(y)
    output$text5 <- renderText(x)
  }
)
```

Try it

- type `runApp()`
- Notice hitting refresh incriments `y` but enterting values in the textbox does not
- Notice `x` is always 1
- Watch how it updated `text1` and `text2` as needed.
- Doesn't add 1 to text1 every time a new `text2` is input.
- *Important* try `runApp(display.mode='showcase')`

Reactive expressions

- Sometimes to speed up your app, you want reactive operations (those operations that depend on widget input values) to be performed outside of a `render*`1 statement
- For example, you want to do some code that gets reused in several `render*` statements and don't want to recalculate it for each
- The `reactive` function is made for this purpose

Example

`server.R`

```
shinyServer(
  function(input, output) {
    x <- reactive({as.numeric(input$text1)+100})
    output$text1 <- renderText({x()                          })
    output$text2 <- renderText({x() + as.numeric(input$text2)})
  }
)
```

As opposed to

```
shinyServer(
  function(input, output) {
    output$text1 <- renderText({as.numeric(input$text1)+100  })
    output$text2 <- renderText({as.numeric(input$text1)+100 +
        as.numeric(input$text2)})
  }
)
```

Discussion

- Do `runApp(display.mode='showcase')`
- (While inconsequential) the second example has to add 100 twice every time text1 is updated for the second set of code
- Also note the somewhat odd syntax for reactive variables

Non-reactive reactivity (what?)

- Sometimes you don't want shiny to immediately perform reactive calculations from widget inputs
- In other words, you want something like a submit button

ui.R

```
shinyUI(pageWithSidebar(
  headerPanel("Hello Shiny!"),
  sidebarPanel(
      textInput(inputId="text1", label = "Input Text1"),
      textInput(inputId="text2", label = "Input Text2"),
      actionButton("goButton", "Go!")
  ),
  mainPanel(
      p('Output text1'),
      textOutput('text1'),
      p('Output text2'),
      textOutput('text2'),
      p('Output text3'),
      textOutput('text3')
  )
))
```

Server.R

```
shinyServer(
  function(input, output) {
    output$text1 <- renderText({input$text1})
    output$text2 <- renderText({input$text2})
    output$text3 <- renderText({
        input$goButton
        isolate(paste(input$text1, input$text2))
    })
  }
)
```

Try it out

- Notice it doesn't display output `text3` until the go button is pressed
- `input$goButton` (or whatever you named it) gets increased by one for every time pushed
- So, when in reactive code (such as `render` or `reactive`) you can use conditional statements like below to only execute code on the first button press or to not execute code until the first or subsequent button press

```
if (input$goButton == 1){ Conditional statements }
```

Example

Here's some replaced code from our previous `server.R`

```
output$text3 <- renderText({
    if (input$goButton == 0) "You have not pressed the button"
    else if (input$goButton == 1) "you pressed it once"
    else "OK quit pressing it"
})
```

More on layouts

- The sidebar layout with a main panel is the easiest.
- Using `shinyUI(fluidpage(` is much more flexible and allows tighter access to the bootstrap styles
- Examples here (http://shiny.rstudio.com/articles/layout-guide.html)
- `fluidRow` statements create rows and then the `column` function from within it can create columns
- Tabsets, navlists and navbars can be created for more complex apps

Directly using html

- For more complex layouts, direct use of html is preferred (http://shiny.rstudio.com/articles/html-ui.html)
- Also, if you know web development well, you might find using R to create web layouts kind of annoying
- Create a directory called `www` in the same directory with `server.R`
- Have an `index.html` page in that directory
- Your named input variables will be passed to `server.R` `<input type="number" name="n" value="500" min="1" max="1000" />`
- Your `server.R` output will have class definitions of the form `shiny-` `<pre id="summary" class="shiny-text-output"></pre>`

Debugging techniques for Shiny

- Debugging shiny apps can be tricky
- We saw that `runApp(displayMode = 'showcase')` highlights execution while a shiny app runs
- Using `cat` in your code displays output to stdout (so R console)
- The `browser()` function can interupt execution and can be called conditionally (http://shiny.rstudio.com/articles/debugging.html)