

Shiny: Part 2

Daniel Anderson

Week 8, Class 2

Agenda

- Review Lab 3
- Review shiny basics
- Review some layout options
- Introduce shiny dashboard
- If time allows – introduce reactivity

Learning objectives

- Solidify your understanding of the UI/Server relation
- Be able create tabsets and a navbar
- Be able to create basic shiny dashboards

Review Lab 3

Basic shiny

Create a basic shiny template, as we did before.

03:00

UI

- The **ui** defines the look and feel of the app – the user interface
- Use it to define where output "lives"
- It also defines the inputs for the **server**, where functions are actually evaluated.
- In the template/default case, a **sliderInput** has been defined, which we're calling **"bins"**. It will take on values from 1 to 50, and will start at 30.
- Access the specific value the user selects within the **server**, through **input\$bins**.

Server

- The `server` function takes the `input` from the UI and produces `output` with normal R code.
- In this case, we're creating one output object, called `distPlot`. The result is then called through the `ui` on line 30

Change the input


Let's switch from a slider to a drop down menu.

How?

Even if you don't know the specific code, what would we change?

`sliderInput` will become `selectInput`. The arguments will also be slightly different

Try!



02:00

Tabs

- Let's say we wanted a tabset with different things.
- First, we need at least two things!
- Let's create a table that has the lower/upper bound of the bin, and the counts within that range.

Table creation

```
input_bins <- 30 # placeholder for whatever the input is
x <- faithful[, 2] # data you're working with

bins <- seq(min(x),
            max(x),
            length.out = input_bins + 1)

h <- hist(x, breaks = bins, plot = FALSE)

tibble(lower = lag(h$breaks),
        upper = h$breaks) %>%
  drop_na(lower) %>%
  mutate(counts = h$counts) %>%
  mutate_if(is.numeric, round, 2)
```

Because this is base syntax, I'll give you the basics for the table, you focus on the shiny part

Your turn

You take it from here! Add a table below the plot with `DT::datatable` or `reactable::reactable`

08:00

[demo]

Move it to a tabset

- Just create a `tabsetPanel` within the `mainPanel`, then put the output for each tab within `tabPanel`.

05:00

Different pages

Add a navbar

- Instead of using a tabset with `tabsetPanel`, you might want to have a navbar at the top of the page, which you can create with `navbarPage`.
- Can be a bit more complicated – each `tabset` needs to include everything, including the `sidebarPanel` (if present), could include tabsets, `mainPanel`, etc.
- Essentially each tab from the `navbar` becomes an entirely new page.

Important things with navbar

- Move the title of your application to the first argument of `navbar` otherwise you'll end up with weird renderings
- Use separate input IDs even if you're repeating the same thing (otherwise it won't be responsive)

More on the navbar

- Can really help with organization/flexibility (you could even have tabs within a page)
- Refactoring can help organization A LOT
 - Pull pieces out to try to make code more readable/less buggy.
- You might consider shiny modules

{shinydashboard}

Getting started

First dashboard – ui

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(title = "Basic dashboard"),
  dashboardSidebar(),
  dashboardBody(
    # Boxes need to be put in a row (or column)
    fluidRow(
      box(plotOutput("plot1", height = 250)),
      box(
        title = "Controls",
        sliderInput("slider", "Number of observations:", 1, 100,
        )
      )
    )
  )
)
```

First dashboard – server

```
server <- function(input, output) {  
  set.seed(122)  
  histdata <- rnorm(500)  
  
  output$plot1 <- renderPlot({  
    data <- histdata[seq_len(input$slider)]  
    hist(data)  
  })  
}
```

Run it

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

[demo]

Main differences

- You now have `dashboardSidebar` and `dashboardBody`
- You also now have `fluidRow` and `box` arguments to arrange things in the main body

Sidebar

- Probably the defining characteristic of the dashboard
 - Define a `sidebarMenu` with `menuItems`

Example

```
sidebarMenu(  
  menuItem("Histogram", tabName = "histo", icon = icon("chart-bar")),  
  menuItem("Bin Counts", tabName = "bins", icon = icon("table"))  
)
```

You can also do things like put the slider in the `sidebarMenu`

[demo]

Referencing menu items

- If you define **menuItem**s, you'll have to give them a **tabName** (see previous slide).
- In the **dashboardBody**, create a **tabItems** with specific **tabItem** pieces. This should be how you control/refer to the **menuItem**.

[demo]

Put slider in sidebar

In this case, it's less than ideal to have the slider separated.

Instead, we can put it right in the sidebar

Bonus – it can then control things across tabs

[demo]

Extension

- There's lots of extensions for shiny, and quite a few (but not as many) for shinydashboard
- Consider themeing shiny apps with `{shinythemes}` and dashboards with `{dashboardthemes}`
- Consider themeing figures to match your shiny theme with `{thematic}`

reactivity

What is it?

- What you've been doing when writing shiny code
- Specify a graph of dependencies
 - When an input changes, all related output is updated

Inputs

- `input` is a basically a list object that contains objects from the ui

```
ui <- fluidPage(  
  numericInput("count", label = "Number of values", value = 100)  
)
```

After writing this code, `input$count` will be available in the server, and the value it takes will depend on the browser input (starting at 100)

These are read-only, and cannot be modified

Selective read permissions

It must be in a reactive context, or it won't work.

That's why this results in an error

```
server <- function(input, output, session) {  
  print(paste0("The value of input$count is ", input$count))  
}  
  
shinyApp(ui, server)  
# > Error in .getReactiveEnvironment()$currentContext() :  
# > Operation not allowed without an active reactive context.  
# > (You tried to do something that can only be done from inside
```

Output

- The **output** object is similar to **input**, in terms of being a list-like object.
- Create new components of the list for new output, and refer to them in the UI
- These also need to be in reactive contexts (e.g., **render***)

Simple example

Try this app. Type the letters in one at a time. Notice how it updates.

```
ui <- fluidPage(  
  textInput("name", "What's your name?"),  
  textOutput("greeting")  
)  
  
server <- function(input, output, session) {  
  output$greeting <- renderText({  
    paste0("Hello ", input$name, "!!")  
  })  
}
```


Programming style

- Notice you don't have to "run" the code each time the input updates
- Your app provides instructions to R. Shiny decides when it actually runs the code.

This is known as declarative programming

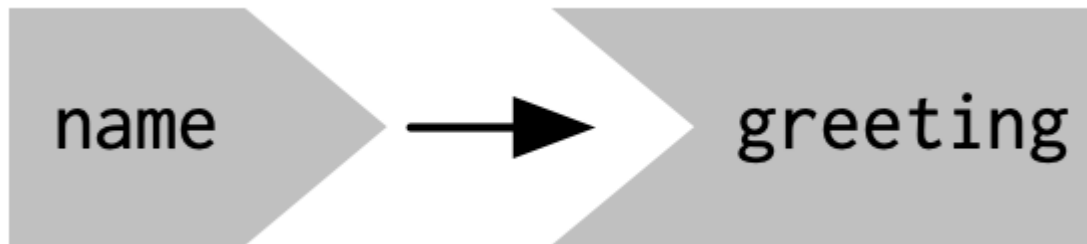
Normal R code is *imperative* programming – you decide when it's run. Declarative programming means you provide instructions, but don't actually run it.

you describe your overall goals, and the software figures out how to achieve them

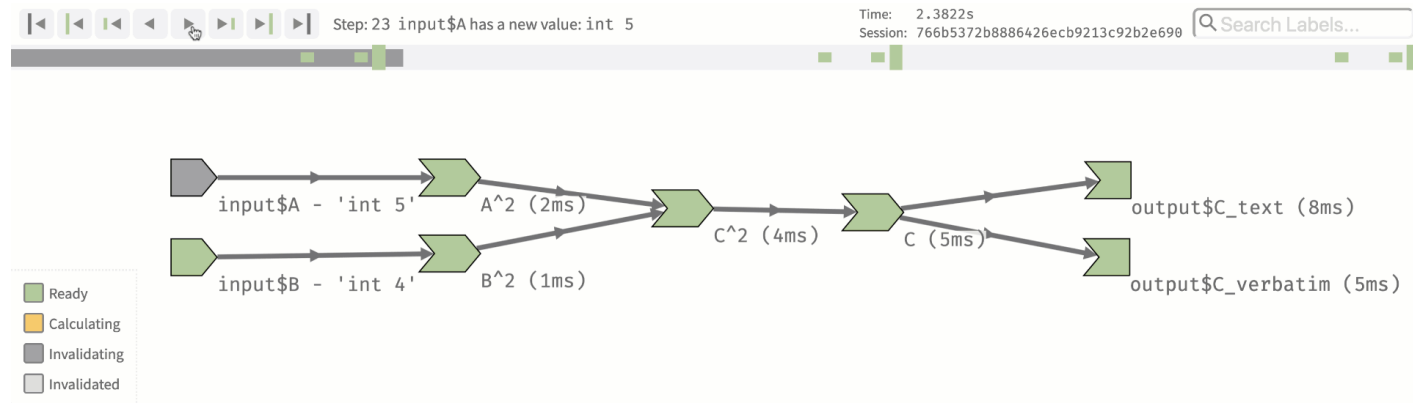
(from Hadley)

Reactive graph

- Normally, you understand R code by running it top to bottom
- This doesn't work with shiny
- Instead, we think through reactive graphs



reactlog



Basic example

```
library(shiny)
library(reactlog)

reactlog_enable()

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)

# close app, then
reactlogShow()
```

Conclusions

- Shiny is super customizable – almost limitless (see more examples [here](#))
- Great for building interactive plots, but you can use it for all sorts of other things too (including text and tables)
- Really helpful and fun way to build data tools for practitioners
- Takes some practice, but basically allows you to write normal R code, and get interactive websites

Next time

More Shiny

Specifically we'll talk about reactivity and, perhaps, shiny modules