

# Shiny: Part 1 & 2

---

Daniel Anderson

Week 8

# Agenda

---

- Review Lab 3
- Shiny!
  - template
  - ui/server

# Learning objectives

---

- Be able to create basic interactive plots with shiny
- Understand basic layouts
- Be able to create basic shiny dashboards

# Review Lab 3

---



It's required you make this joke when talking about shiny (especially if you're a parent of young kids)

# The basics!

---

Do the following:

- Create a new R project
- Select "Shiny Web Application"
- Select the directory
- Click "create project"

[demo]

# First thing – Run App!

---

- I like to open in the browser
- What's going on here? Talk with your neighbor and see if you can deduce the basic functionality.

02:00

# 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 this case – we have defined a `sliderInput`, 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 puts it in 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
- Try removing lines 39–44, and replacing it with `print(input$bins)`
- Feels like it should work, but it doesn't. Any ideas why?

# Text output

---

We are no longer rendering a plot. We are rendering text.

We need to change `renderPlot` to `renderText` and `plotOutput` to `textOutput`.

Try

Does it work now?

# Challenge

---

- Try producing essentially the same thing as the default, but use `ggplot` instead of base plotting.
- Reminder, the data are called `faithful`, and you'll be showing the distribution of `waiting`

10:00

[demo]

---

# Change the input

---

- Let's say instead of a slider, we want 5 options: 1, 5, 20, 50, or 100
- We can change the input from a slider (representing continuous intervals) to radio buttons, representing discrete choices.
- Just change `sliderInput` to `radioButtons`, and put in the appropriate `choices`. You can also add a `selected` argument for which it defaults to.

Try it out!

05:00

[demo]

---

# More complicated

---

- Show distribution of highway mpg that could be faceted by `year`, `trans`, or `class`.
- Keep slider to control bins.
- Use `radioButtons` for the variables, and `sliderInput` for the bins.
- You'll need to create a second entry to your `ui`
- Use the `mpg` dataset (from `ggplot2`)

[demo]

- Change the width of the sidebar with `sidebarPanel(width = 2)` if you want
- Change the height of the main panel plot with `plotOutput("distPlot", height = "800px")` if you want

[demo]



# Tables

---

- My recommendation: Use one of the following:
  - `DT::datatable`
  - `reactable::reactable`
- Let's build a table to go below our distribution(s) that shows `mean`, `sd`, `min`, and `max` of the corresponding distribution(s)
- You'll need to create a new output object in the server, and tell the ui where it should be rendered

- Use `renderDataTable` in the server and `dataTableOutput` in the ui or `renderReactable` and `reactableOutput`
- If you want to use `{dplyr}`, as I would, then you have to deal with NSE. In this case it's not that big of a deal though. Just use `!!sym(input$var)`, where `sym` transforms a string to a symbol.

[demo]

# Last thing before break

---

## Change the layout

- Let's say instead of having the table below, we wanted it within a tabset
- Just create a `tabsetPanel` within the `mainPanel`, then put the output for each tab within `tabPanel`.

[demo]

# Break

---

05:00

# Alternative

---

- You might want to have a navbar at the top of the page, which you can create with `navbarPage()`.
- Create each page with `tabPanel()`
- Each `tabPanel` needs to include everything, including the `sidebarPanel()` (if present), could include `tabsetPanel()`, `mainPanel()`, etc.
- Move your title so it's `navbarPage("App title",` or the rest of the pages won't look right
- Make sure not to share `inputId`'s across pages – it won't work.

# Conclusions

---

- Shiny is super customizable – just scratched the surface today
- Mastering Shiny should be your guide for going deeper
- Great for building interactive plots, but you can use it for all sorts of other things too (including text and tables)
- Apps get complicated quickly
  - Consider pulling pieces out as a refactoring method or using shiny modules

Shiny

dashboards

---

# 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]

# More complicated

---

Use the `forcats::gss_cat` dataset

- Plots on one page
  - boxplots, with `tvhours` on y-axis
  - counts of x variable
  - proportion of x variable
- Table on another page
- Select the variable for x-axis on the sidebar
- Variable selection should produce table with means by the variable

[walkthrough]

# 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

---

My guess – we're mostly out of time, but these slides are here just in case.



# 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.

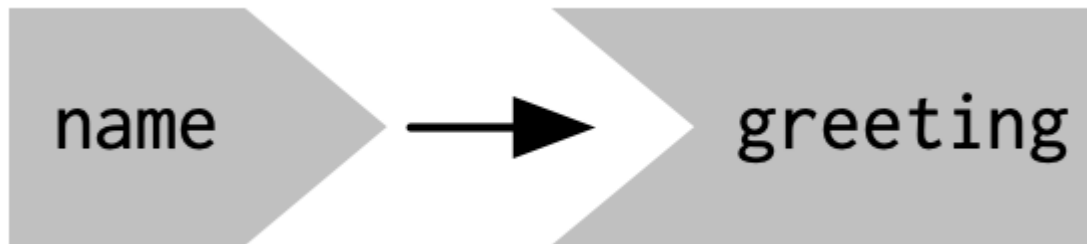
you describe your overall goals, and the software figures out how to achieve them

(from Hadley)

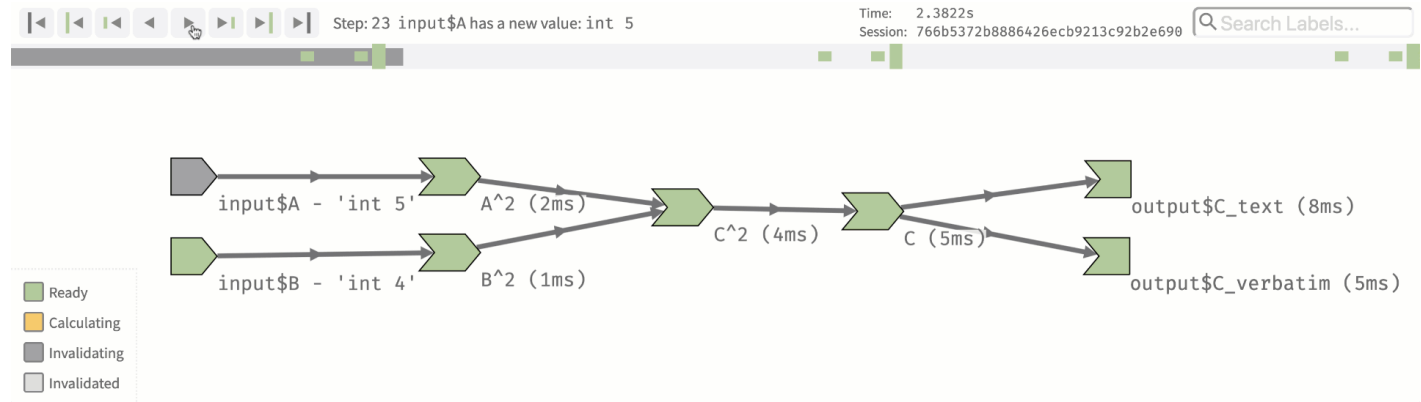
# Reactive graph

---

- 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](#))
- shinydashboard provides, basically, an alternative layout
- 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 more about reactivity and, perhaps, shiny modules