Shiny: Part 3

{shinydashboards}

Daniel Anderson

Week 9, Class 1

Agenda

- Introduce shiny dashboard
- Introduce reactivity
- Some shiny best practices

Learning objectives

- Be able to create basic shiny dashboards
- Have at least a basic understanding of reactivity
- Recognize use cases where your functional programming skills can help make more efficient and/or clear apps

{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 `menuItem`s

Example

```
sidebarMenu(
  menuItem("Histogram", tabName = "histo", icon = icon("chart-ba
  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
- See here for a mostly comprehensive list
- 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 a 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, "!")
  })
}
```

From Mastering Shiny

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

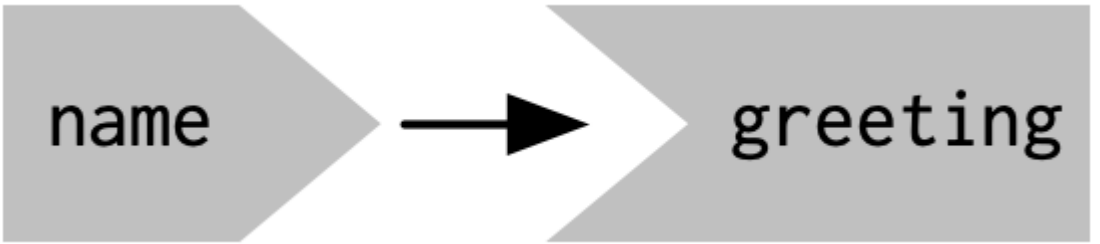
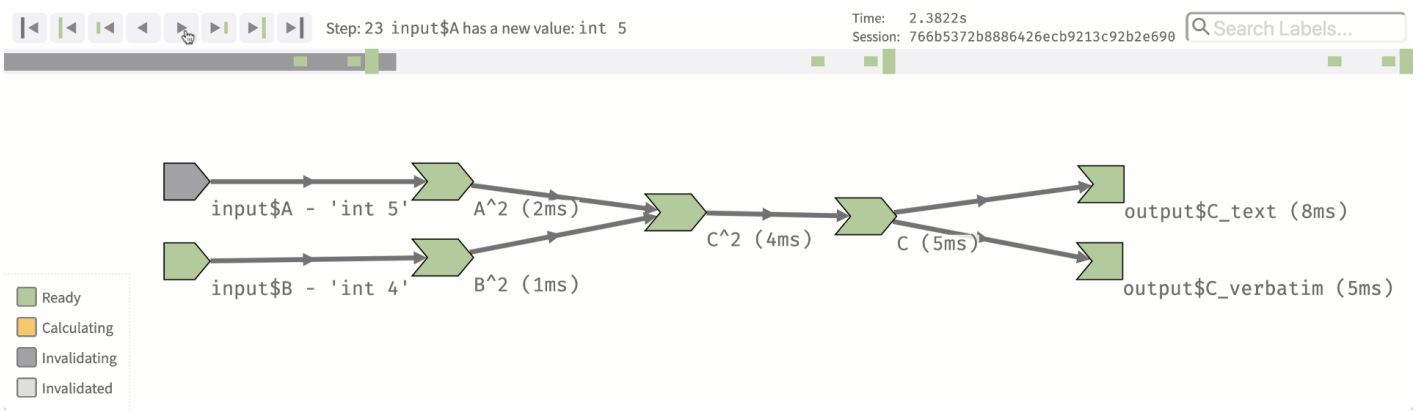


image from Mastering Shiny

reactlog


Step: 23 input$A has a new value: int 5
Time: 2.3822s
Session: 766b5372b8886426ecb9213c92b2e690
Search Labels...
input$A - 'int 5'
input$B - 'int 4'
A^2 (2ms)
B^2 (1ms)
C^2 (4ms)
C (5ms)
output$C_text (8ms)
output$C_verbatim (5ms)
Ready
Calculating
Invalidating
Invalidated

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()
```

Why reactivity?

Imagine we want to have a simple app converting temperatures from Fahrenheit to Celsius

Do this with variables

```
temp_f <- 72
temp_c <- (temp_f - 32) * (5/9)
temp_c
```

```
## [1] 22.22222
```

But changing `temp_f` has no impact on `temp_c`

```
temp_f <- 50
temp_c
```

```
## [1] 22.22222
```

Use a function?

Let's instead make this a function that depends on the object in the global environment.

```
to_celsius <- function() {
  (temp_f - 32) * (5/9)
}
to_celsius()
```

```
## [1] 10
```

```
temp_f <- 30
to_celsius()
```

```
## [1] -1.111111
```

This works, but it's less than ideal computationally.

Even if `temp_f` hasn't changed, the conversion is re–computed Not a big deal in this case, but is a big deal with

Reactive alternative

First create a reactive variable

```
library(shiny)
reactiveConsole(TRUE)
temp_f <- reactiveVal(72)
temp_f()
```

```
## [1] 72
```

```
temp_f(50)
temp_f()
```

```
## [1] 50
```

Reactive function

Next create a reactive function

```
to_celsius <- reactive({
  message("Converting...")
  (temp_f() - 32) * (5/9)
})
```

Now it will convert **only** when the value of `temp_f()` changes

```
to_celsius()
```

```
## [1] 10
```

```
to_celsius()
```

```
## [1] 10
```

Some shiny best practices

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand. — Martin Fowler

Things to consider

- Are variable/function names clear and concise?
- Is code commented, where needed?
- Is there a way to break up complex functions?
- Are there many instances of copy/pasting?
- Can I manage different components of my app independently? Or are they too tangled together?

From Mastering Shiny

Functions

Something new I learned

Last time I showed you how to `source()` files. It turns out you don't need to do this!

- Create a new folder, `R/`
- Place `.R` files in that folder
- shiny will source them automatically!!

UI example

Imagine we have a bunch of different sliders we want to create. We could do something like this:

```
ui <- fluidRow(
  sliderInput("alpha", "alpha", min = 0, max = 1, value = 0.5, st
  sliderInput("beta",  "beta",  min = 0, max = 1, value = 0.5, st
  sliderInput("gamma", "gamma", min = 0, max = 1, value = 0.5, st
  sliderInput("delta", "delta", min = 0, max = 1, value = 0.5, st
)
```

Ideas on how you could write a function to reduce the amount of code here?

Example from Mastering Shiny

Slider function

```
my_slider <- function(id) {
  sliderInput(id, id, min = 0, max = 1, value = 0.5, step = 0.1)
}
ui <- fluidRow(
  my_slider("alpha"),
  my_slider("beta"),
  my_slider("gamma"),
  my_slider("delta")
)
```

Anyway to make this even less repetitive?

Loop through the ids

```
ids <- c("alpha", "beta", "gamma", "delta")
sliders <- map(ids, my_slider)
ui <- fluidRow(sliders)
```

Other use cases

- Maybe you want to use a shiny function, but assign it with some standard formatting
    - e.g., icons
- Maybe there are functions where you only want to change 1–3 things? Create a function that allows you to modify those, but keep other things at the values you want

Server functions

- I often find it helpful to create functions for output, even if I'm not repeating them a lot
    - Can help keep server code clean and concise
- Inspect your code and consider refactoring in a similar way you would standard functions
- Consider keeping these in a separate `.R` file from your UI functions

shiny modules

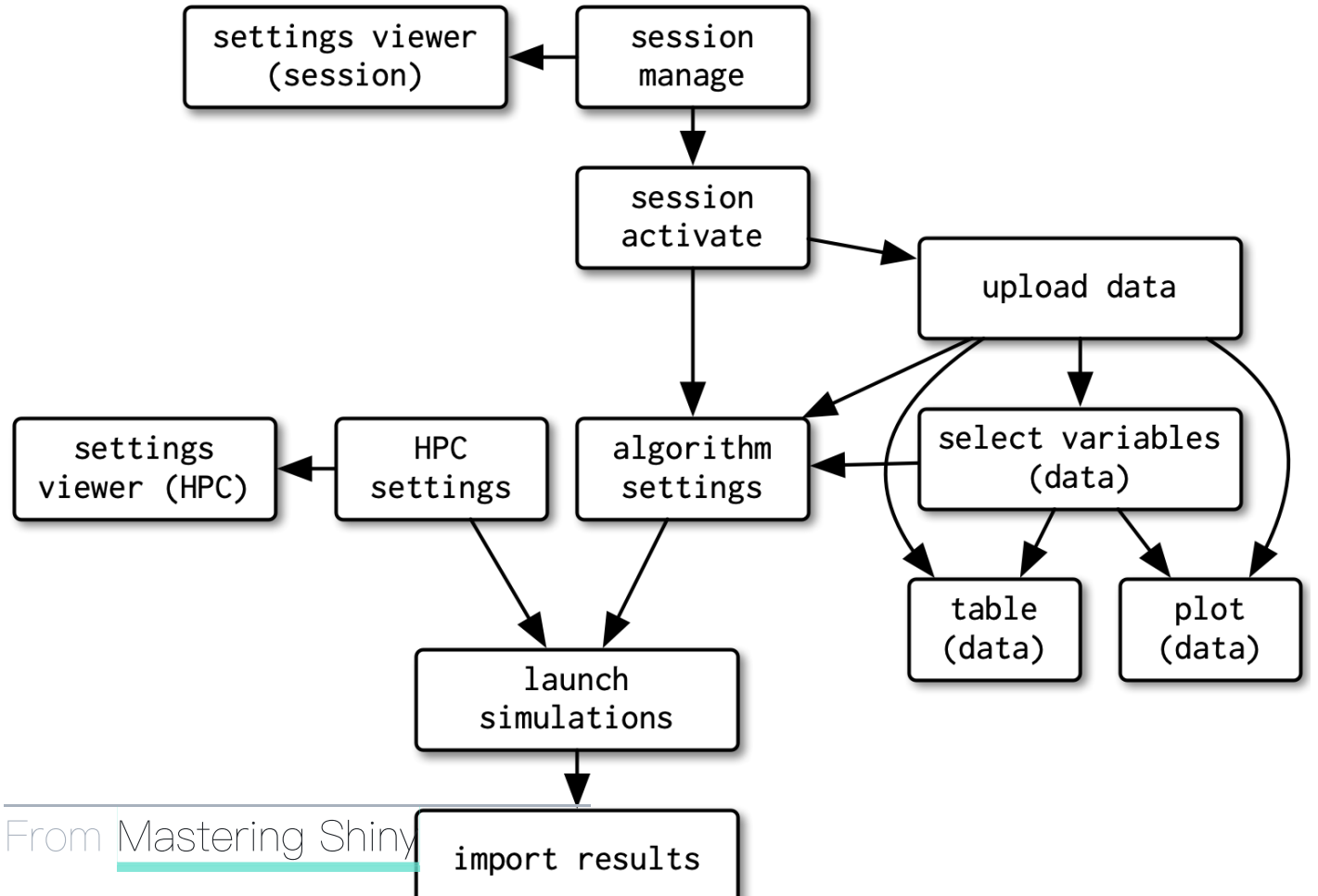
(briefly)

What is a module?

- Allows you to modularize parts of your shiny app
- This can help "de–tangle" these pices from the rest of the app
- Primarily useful when the thing you want to modularize spans *both* the UI and the server

Example

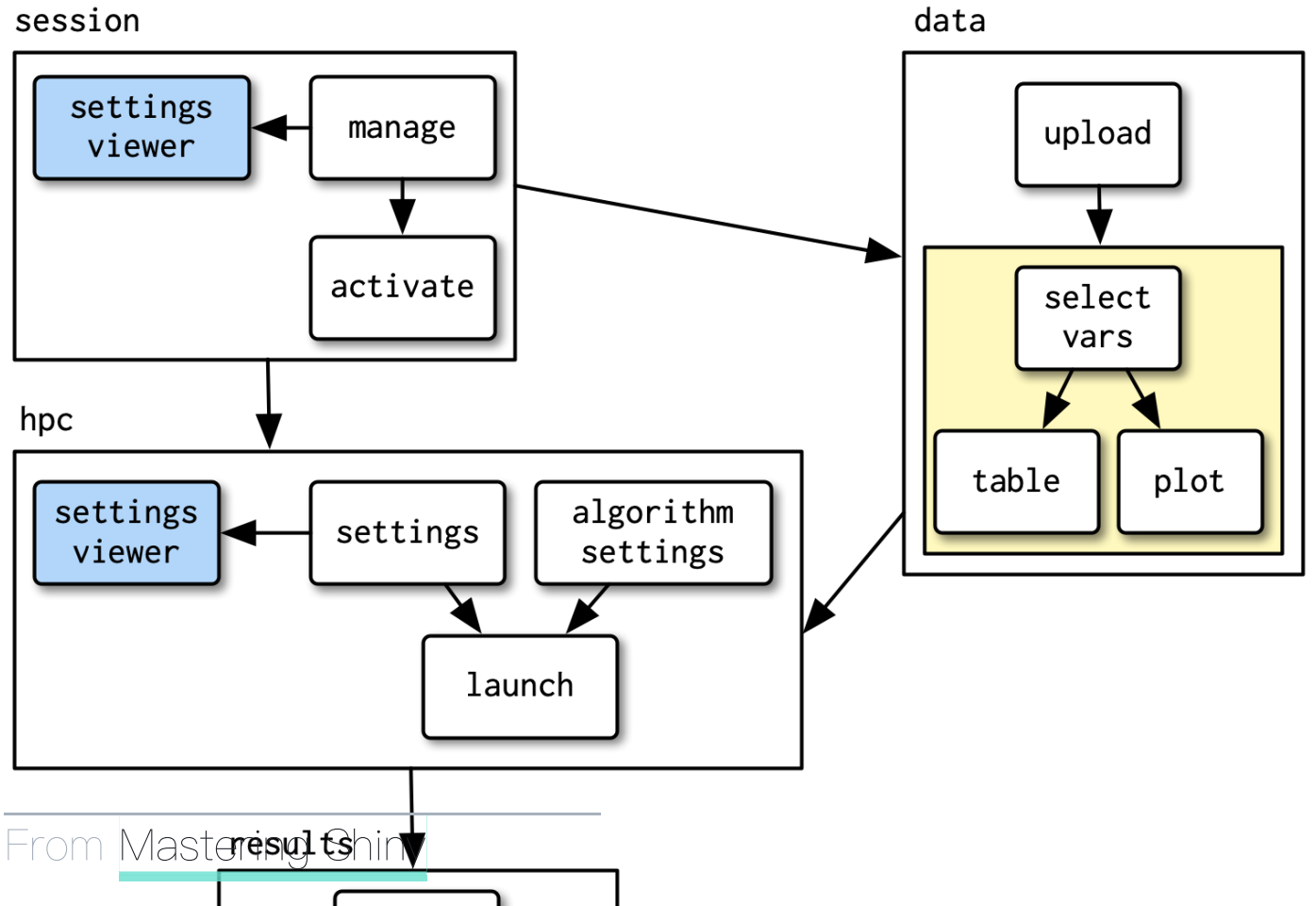
No modules



From Mastering Shiny

Example

Same app, but with modules



From Mastering Shiny

Create modules

- It's fairly complicated. See the chapter from Mastering Shiny on the topic
- Probably not worth it until you're creating complicated apps, but I wanted to make you aware of them

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

Any time left?

Challenge

- Create a shiny app or shiny dashboard with the `palmerpenguins` dataset
- Allow the x and y axis to be selected by the user
    - These should be any numeric variables
- Allow the points to be colored by any categorical variable
    - For an added challenge, try to add in a "no color" option, which should be the default

Next time

Review