Shiny: Parts 3 & 4

Daniel Anderson

Week 9

Agenda

- Introduce reactivity
- Some shiny best practices

Learning objectives

- 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

Increase readability

- As I've mentioned, shiny apps can become unreadable quickly

Consider creating objects for each piece, possibly even in separate R files.

Example

```
sidebar <- dashboardSidebar(
  sidebarMenu(
    menuItem("Histogram", tabName = "histo", icon = icon("chart-
    sliderInput("slider", "Number of observations:", 1, 100, 50)
  )
)

body <- dashboardBody(
  fluidRow(
    tabItems(
      tabItem(
        "histo",
        box(plotOutput("plot1", height = 250))
      )
    )
  )
)
```

UI part

```r
ui <- dashboardPage(
  dashboardHeader(title = "Basic dashboard"),
  sidebar,
  body
)
```

Note you could create intermediary objects within each of the `sidebar` and `body` parts as well.

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 basically a list object that contains objects from the ui

```r
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.

```r
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}

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

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

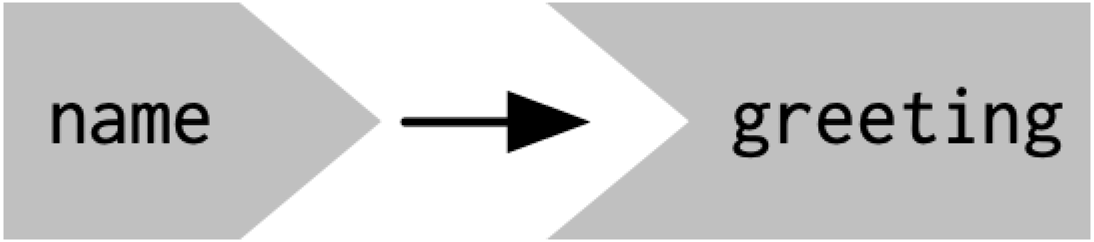
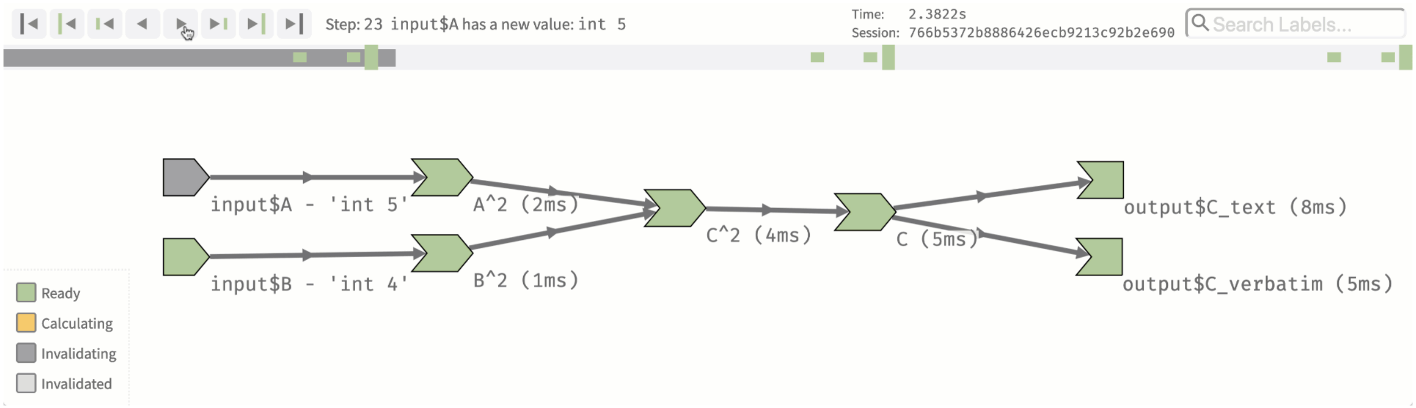


image from Mastering Shiny

reactlog



Step: 23 input$A has a new value: int 5

Time: 2.3822s
Session: 766b5372b8886426ecb9213c92b2e690

Basic example

```r
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

```r
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`

```r
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.

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

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

```r
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 often is.

Reactive alternative

First create a reactive variable

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

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

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

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

Reactive function

Next create a reactive function

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

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

```r
to_celsius()
```

```
## Converting...
```

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

```r
to_celsius()
```

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

```r
temp_f(100)
to_celsius()
```

```
## Converting...
```

```
## [1] 37.77778
```

```r
to_celsius()
```

```
## [1] 37.77778
```

Unless the value of `temp_f` changes, the code will not be re–run.

An app example

Please follow along

We'll build an app to show the results of different regression models.

First, an extension!

We want to be able to select which variables are predictors in the model.

Ideally, we should be able to select 1, 2, ... n predictors.

`shinyWidgets::multiInput()` to the rescue!

Super basic example

```r
library(shiny)
library(shinyWidgets)
library(palmerpenguins)

ui <- fluidPage(
  titlePanel("Regression Example"),
  sidebarLayout(
    sidebarPanel(
      multiInput(
        inputId = "xvars",
        label = "Select predictor variables :",
        choices = names(penguins)[-3], # column 3 will be outcome
        selected = "island"
      )
    ),
    mainPanel(

    )
  )
)
```

Leave the server blank

Try!

```
server <- function(input, output) {

}

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

[demo]

Write the server

- We now want it to fit a model with `bill_length_mm` as the outcome, and whatever variables are selected as predictors

- We can do this by creating a *reactive* model, that only is estimated when something changes.

Example

```
model <- reactive({
  form <- paste(
    "bill_length_mm ~ ",
    paste(input$xvars, collapse = " + ")
  )

  lm(as.formula(form), penguins)
})
```

The above creates a string, which is then converted to a formula

It will only be estimated when `input$xvars` is changed

We can refer to the model object elsewhere in our server with `model()`.

Build a table

- Use the gt and gtsummary packages to pull the model coefficients
- Render the result in **gt** table

Remember to load additional packages

```
library(shiny)
library(shinyWidgets)
library(palmerpenguins)
library(gtsummary)
library(gt)
```

In the server:

```
output$tbl <- render_gt({
  as_gt(tbl_regression(model(), intercept = TRUE))
})
```

In the `mainPanel()` of the `ui`:

```
gt_output("tbl")
```

Try!

[demo]

Make it fancier

- Let's add an equation with equatiomatic.

You try first:

- Use `extract_eq()` to extract the equation, and the `renderEq()` and `eqOutput()` functions for shiny.

- Make sure to include `withMathJax()` in your `ui` somewhere (I put it after the title).

- Check out the help page for `extract_eq()` for additional arguments, including `use_coefs`, and play around with these options.

[demo]

06:00

Going further

- Partial regression plots? (see visreg).
- The more we add, the more we might want to think about layouts – maybe a dashboard would be preferred?
- Could make it so you can change the outcome as well (but this gets a bit tricky)
- Others?

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, s
  sliderInput("beta",  "beta",  min = 0, max = 1, value = 0.5, s
  sliderInput("gamma", "gamma", min = 0, max = 1, value = 0.5, s
  sliderInput("delta", "delta", min = 0, max = 1, value = 0.5, s
)
```

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

Example from Mastering Shiny

Slider function

```r
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

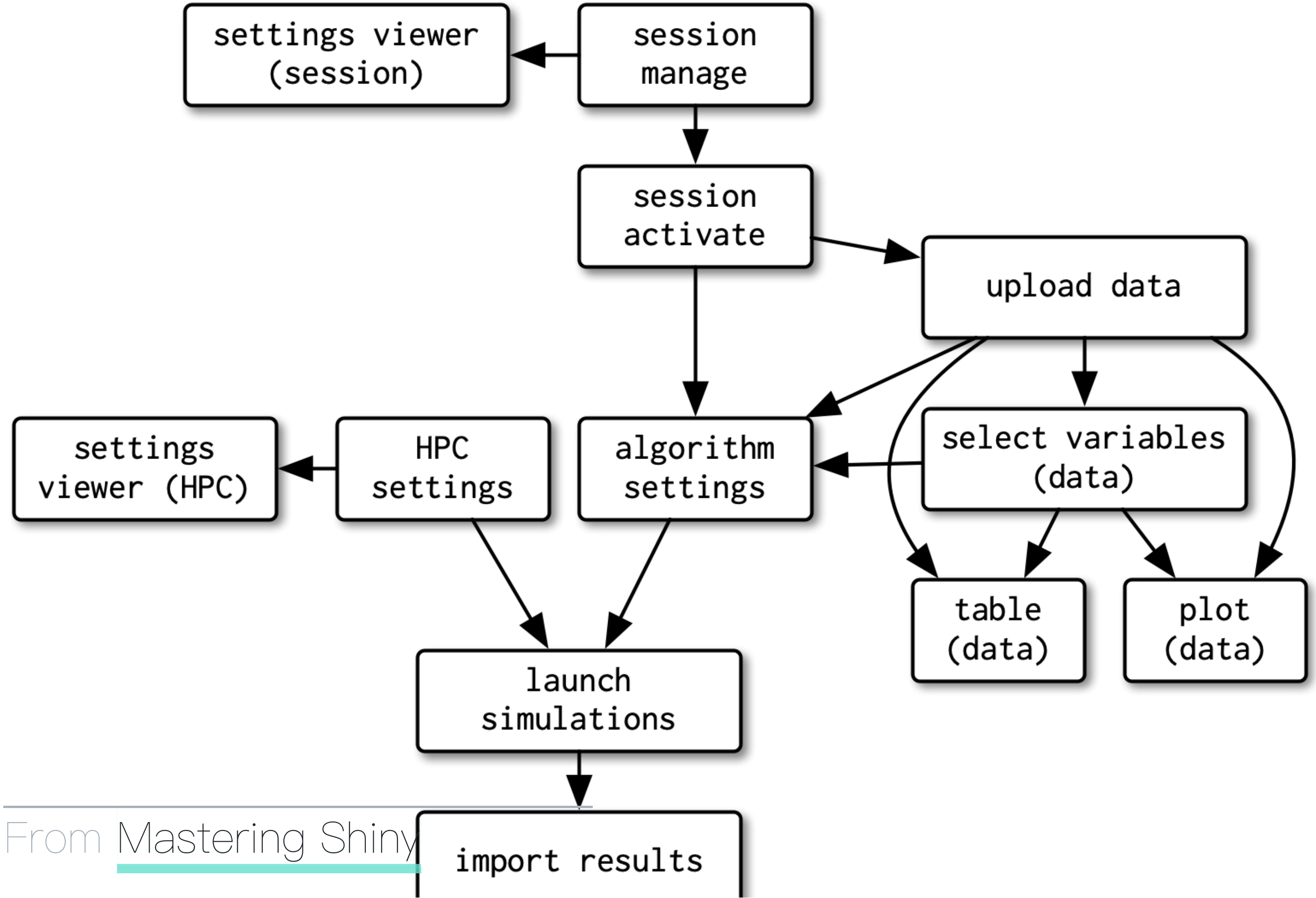
(briefly)

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

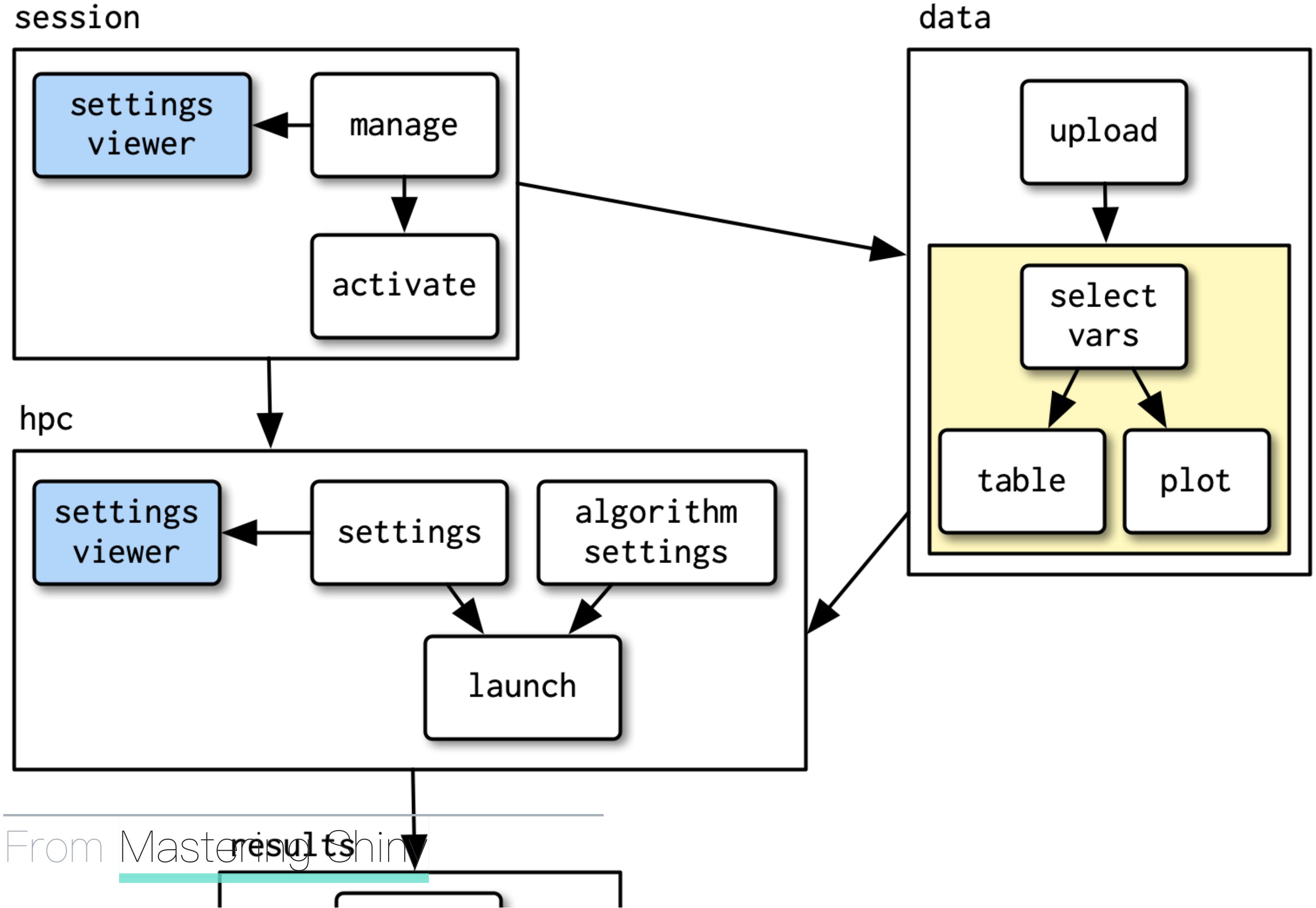
No modules



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

Let's look at a very basic use case from Mastering Shiny

The original app

```r
ui <- fluidPage(
  selectInput("var", "Variable", names(mtcars)),
  numericInput("bins", "bins", 10, min = 1),
  plotOutput("hist")
)

server <- function(input, output, session) {
  data <- reactive(mtcars[[input$var]])

  output$hist <- renderPlot({
    hist(
      x = data(),
      breaks = input$bins,
      main = input$var
    )
  })
}

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

Histogram UI module

- A function for the UI components of the histogram
- Wrap all internals into a single function with an `id` argument
- Wrap all inputs in a `tagList()`, with inputs separated by commas
- Wrap each ID in `NS()`
    - Allows the multiple id's to be referenced by a single id

```r
uiHist <- function(id) {
  tagList(
    selectInput(NS(id, "var"), "Variable", names(mtcars)),
    numericInput(NS(id, "bins"), "bins", 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

Histogram server module

- Wrap the internals in a function with an `id` argument
- Inside this function, wrap the internals again with the `moduleServer()` function
  - First argument is `id`
  - Second argument is `function(input, output, session) { <internals> }`

```r
serverHist <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(mtcars[[input$var]])

    output$hist <- renderPlot({
      hist(
        x = data(),
        breaks = input$bins,
        main = input$var
      )
    })
  })
}
```

New app

```r
ui <- fluidPage(
  uiHist("histo")
)

server <- function(input, output, session) {
  serverHist("histo")
}
```

The modules can then live in their own files, and your UI and server functions become much more clean

Conclusions

- Shiny is fun!
- Bigger apps get pretty complicated pretty quickly
- Consider modules
- Do try to be efficient with your code whenever possible
- Think carefully about reactivity – it's a fundamentally different approach.

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