HDAT9800

Visualisation and Communication of Health Data

Chapter 8

Drs James Farrow And Tim Churches

More sophisticated Shiny apps

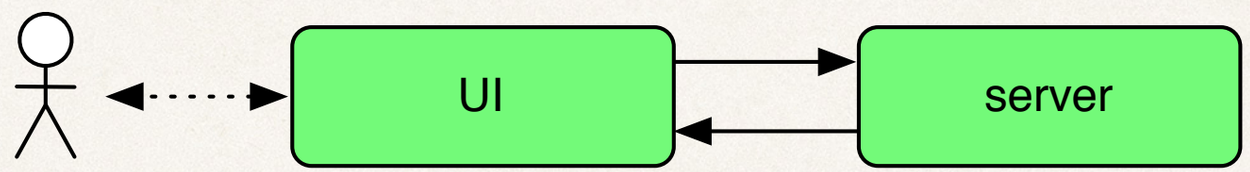
Recall the architecture of a shiny app

- UI portion
- server portion

Shiny app architecture

Recall the architecture of a shiny app

- UI portion
- server portion



User interacts with UI

Server builds and responds to changes in UI to update UI

UI

UI is usually created with a called to *fluidPage* to construct the UI

- creates named *inputs*
- has places for names *outputs*

```
# ui.R
library(shiny)

fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25,
    min = 1, max = 100),
  plotOutput("hist")
)
```

Server

A function of two variables *input* and *output*

Has slots for each *output* which creates the output for each *input*

Shiny keeps track of input changes to determine which outputs need to be regenerated

Uses render functions

```
# server.R
library(shiny)

function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
```

Reactive values

Shiny keeps track of which outputs depend on which inputs

When inputs change it knows which outputs need to be regenerated

The outputs are regenerated and the UI updated

This is convenient but not always desirable

It can lead to unnecessary time-consuming (re)computation

Reactive values tutorial

Part 2 of the Shiny Video Tutorial deals with reactive values and controlling reactivity

- https://shiny.rstudio.com/tutorial/

See also, *Effective reactive programming*

- https://www.rstudio.com/resources/videos/effective-reactive-programming/

Reactive values

We've already seen reactivity at work

Input changes force output updates

```
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
```

The input slider gets changed: the output graph gets updated

Reactive values must be used inside reactive functions (like *renderPlot*)

Reactive functions

Reactive functions take inputs to create outputs

- *code chunk*
- *reactive values*

The code chunk is provided as an argument to the reactive function as a closure (code surrounded by {…})

```
renderPlot({
   ...
})
```

Note that the entire code chunk has to be run (this will be significant later)

Render functions

Render functions

- renderDataTable()
- renderImage()
- renderPlot()
- renderPrint()
- renderTable()
- renderText()
- renderUI()

Unnecessary (re)computation

Reactive values are convenient but not always desirable

They can lead to unnecessary time-consuming (re)computation

The whole chunk of code in a reactive function must be run

What if we only want to run part of it?

What if the code chunk uses multiple input values but only one has changed?

Unnecessary (re)computation

```
output$popmap <- renderLeaflet({

  CED_polyinfo <- CED_info %>%
    mutate(CED_CODE_2016 = paste0("CED", CED_info$CED_CODE_2016)) %>%
    group_by(CED_CODE_2016, CED_NAME_2016) %>%
    summarise(total_area = sum(AREA_ALBERS_SQKM)) %>%
    left_join(CED_popinfo) %>%
    mutate(pop_density = Tot_P_P / total_area)

  # let's use white -> orange
  pop_pal <- colorNumeric(c(input$zeropopcolour, input$maxpopcolour),
    c(0, input$maxpop))

  # join the CED info to our polygons plotting
  Sydney_polys@data <- left_join(Sydney_polys@data, CED_polyinfo)

  # draw our map
  leaflet() %>% ...

})
```

Unnecessary (re)computation

Our output (`popmap`) depends on three inputs

- `zeropopcolour`
- `maxpopcolour`
- `maxpop`

We want the output to be re-rendered (redrawn) whenever any of these change

However the output also contains a calculation heavy step that does not depend on those variables

- the pipeline which constructs `CED_polyinfo`

Dependencies

```
output$hist <- renderPlot({
  hist(rnorm(input$num))
})

output$stats <- renderPrint({
  summary(rnorm(input$num))
})
```

We're calling *rnorm* twice here

- unnecessary computation
- the stats displayed will be different to the histogram!
  - it's a different random sample from the normal distribution!

Reactive functions

Reactive functions remember their results

They return the same result until they become invalidated

When the inputs change the code chunk is rerun and the new value remembered

Reactive functions

```
ndata <- reactive({
  rnorm(input$num)
})

output$hist <- renderPlot({
  hist(ndata())
})

output$stats <- renderPrint({
  summary(ndata())
})
```

Reactive functions

We can also define a reactive function which just caches a value

```
get_CID_polyinfo <- reactive({

  CED_info %>%
    mutate(CED_CODE_2016 = paste0("CED", CED_info$CED_CODE_2016)) %>%
    group_by(CED_CODE_2016, CED_NAME_2016) %>%
    summarise(total_area = sum(AREA_ALBERS_SQKM)) %>%
    left_join(CED_popinfo) %>%
    mutate(pop_density = Tot_P_P / total_area)

})
```

This code has no input dependencies and so will never be invalidated

It's used to cache the final value and avoid recomputing it unnecessarily

Preventing reactivity

The *isolate()* function creates a value which is non-reactive

```
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  textInput(inputId = "title",
    label = "Write a title",
    value = "Histogram of Random Normal Values"),
  plotOutput(“hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num), main = isolate({input$title}))
  })
}
```

Responding to events

We can create a button for users to click

```
ui <- fluidPage(
  actionButton(inputId = "clicks",
    label = "Click me")
)

server <- function(input, output) {
  observeEvent(input$clicks, {
    print(input$clicks)
  })
}
```

The *observeEvent()* code block is only run when inputs listed in the first argument change (the first argument can be a vector of inputs)

Responding to events

There is also an *observe()* function which tracks all inputs in the code block and which therefore doesn't take a first argument

```
observe({
  ...
})
```

Delaying updates

We can create a reactive expression that only responds to changes in specific values

```
ui <- fluidPage(
  ...
  actionButton(inputId = "go", label = "Update"),
  plotOutput("hist")
)

server <- function(input, output) {
  data <- eventReactive(input$go, {
    rnorm(input$num)
  })
  output$hist <- renderPlot({
    hist(data())
  })
}
```

Event-specific reactivity

Compare

- *observe()* and *observeEvent()*
- *reactive()* and *eventReactive()*

Non-input reactive values

The *reactiveValues()* function creates a list of reactive values which can be used like inputs

```
rv <- reactiveValues()
```

We can add new reactive values to this list later using the $ syntax

```
rv$data = rnorm(100)
```

It can be given a list of initial values

```
rv <- reactiveValues(data = rnorm(100))
```

Non-input reactive values

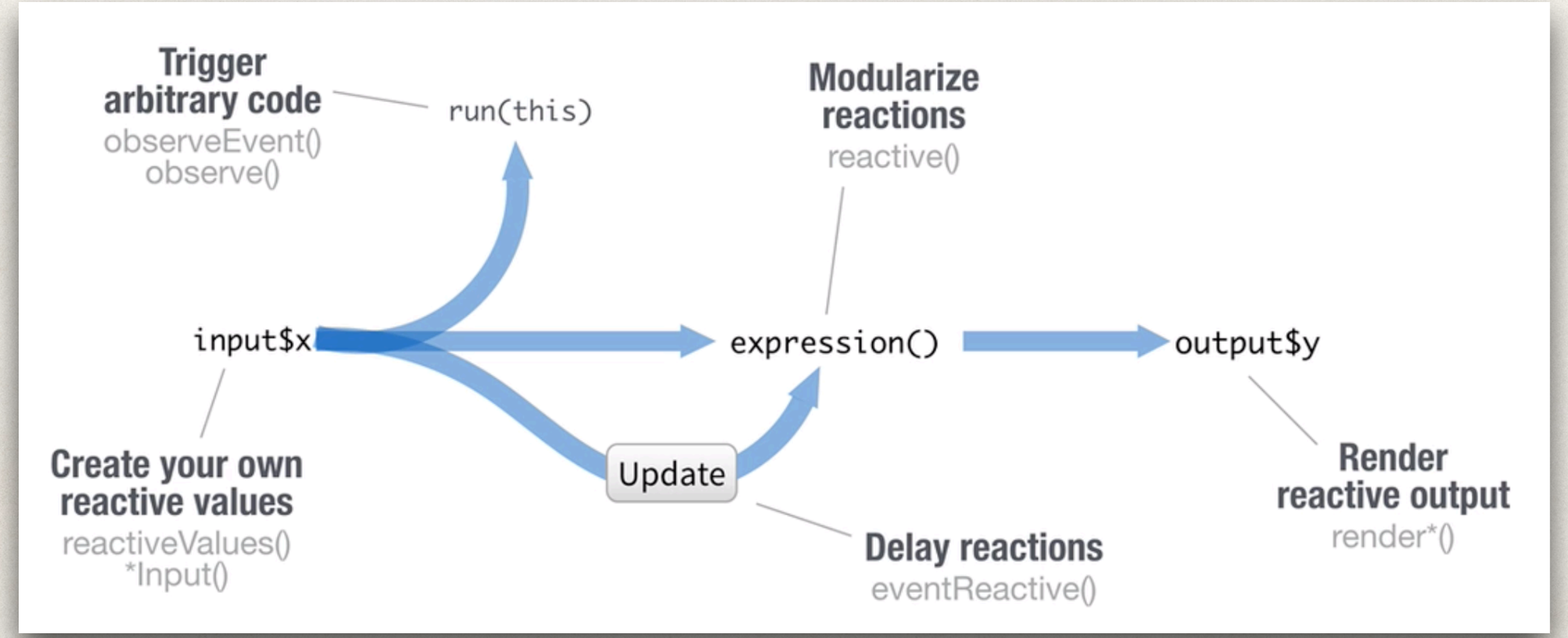
```
ui <- fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
  actionButton(inputId = "unif", label = "Uniform"),
  plotOutput(hist)
)

server <- function(input, output) {
  rv <- reactiveValues(data = rnorm(100))

  observeEvent(input$norm, {rv$data <- rnorm(100)})
  observeEvent(input$unif, {rv$data <- runif(100)})

  ...
}
```

Reactivity


Trigger arbitrary code
observeEvent()
observe()
run(this)
Modularize reactions
reactive()
input$x
expression()
output$y
Create your own reactive values
reactiveValues()
*Input()
Update
Delay reactions
eventReactive()
Render reactive output
render*()

Reactivity

Carefully consider the data flows in your application

- what needs to respond to changes in inputs?
- should those changes be automatic or manually triggered?
- what computation can be cached using reactive values?
- try and minimise how often code is run

Code execution

Code outside of the server function gets run once per session

Code inside the server function gets run once per user (connection)

Code inside a reactive function get run once per reaction (value change)

Practical

Building a shiny app

Interactivity

Useful for the group assignment (option A and possibly for option B)

Optional practice exercise

Turn the example and your solution to the Chapter 7 assessment into a shiny app.

Provide a pair of radio buttons which will switch between the population density map and the median income difference map.

Provide a checkbox which will turn the legend on and off.

Provide appropriately labelled text entry boxes which provide the three colours used as the ends of the scales in these two maps. Changing these boxes should ***not*** force a map redraw until a 'Redraw map' button is pressed (provide a 'Redraw map' button), *i.e.* the map rendering should use reactive value wrappers for these colours.