Understanding reactivity

Mine Çetinkaya-Rundel

@minebocek
mine-cetinkaya-rundel
mine@rstudio.com

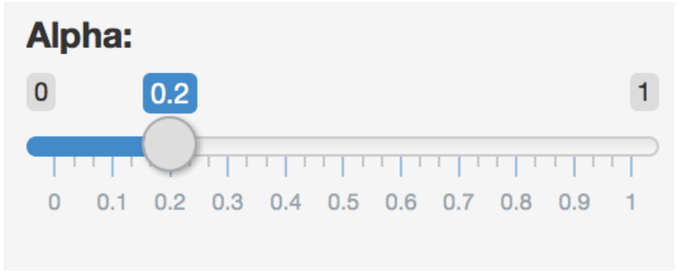
Reactivity 101

Reactions

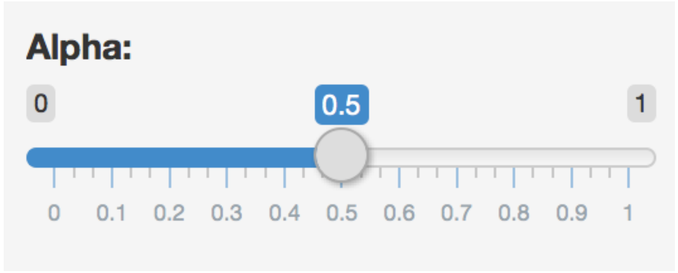
The **input$** list stores the current value of each input object under its name.

```
# Set alpha level
sliderInput(inputId = "alpha",
            label = "Alpha:",
            min = 0, max = 1,
            value = 0.5)
```

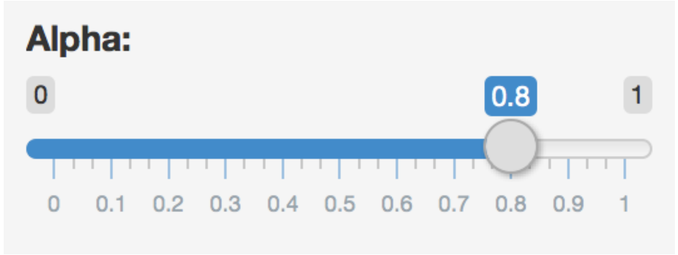
input$alpha



input$alpha = 0.2



input$alpha = 0.5



input$alpha = 0.8

Reactivity 101

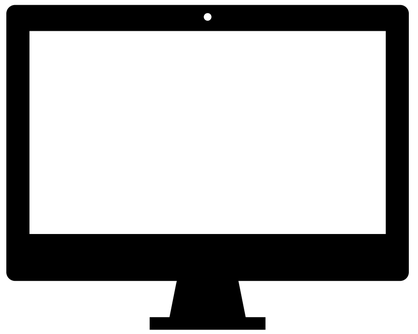
Reactivity automatically occurs when an input value
is used to render an output object

```
# Define server function required to create the scatterplot
server <- function(input, output) {
   # Create the scatterplot object the plotOutput function is expecting
    output$scatterplot <- renderPlot(
     ggplot(data = NHANES, aes_string(x = input$x, y = input$y,
                                      color = input$z)) +
       geom_point(alpha = input$alpha)
    )
}
```

Reactive flow

Suppose you want the option to plot only certain education level(s) as well as report how many such participants are plotted:

1. Add a UI element for the user to select which education level(s) they want to plot
2. Filter for chosen education level(s) and save as a new (reactive) expression
3. Use new data frame (which is reactive) for plotting
4. Use new data frame (which is reactive) also for reporting number of observations



DEMO

1. Add a UI element for the user to select which education level(s) they want to plot

```
# Select which education level(s) to plot
checkboxGroupInput(inputId = "education",
                   label = "Select education level(s):”,
                   choices = levels(NHANES$Education),
                   selected = "College Grad")
```

2. Filter for chosen education level(s) and save as a new (reactive) expression

```
# Server
# Create a subset of data filtering for chosen educ
NHANES_subset <- reactive({
  req(input$education)
  filter(NHANES, title_type %in% input$education)
})
```

Creates a **cached expression** that knows it is out of date when input changes

3. Use new data frame (which is reactive) for plotting

```
# Create the scatterplot object the plotOutput function is expecting
output$scatterplot <- renderPlot({
  ggplot(data = NHANES_subset(), aes_string(x =
                                            col
    geom_point(…) +
    …
})
```

Cached - only re-run when inputs change

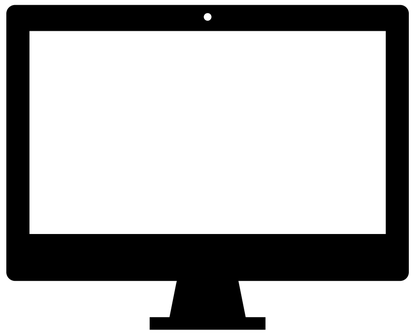
4. Use new data frame (which is reactive) also for printing number of observations

```
# UI
mainPanel(
  …
  # Print number of obs plotted
  uiOutput(outputId = "n"),
  …
  )
# Server
output$n <- renderUI({
  types <- NHANES_subset()$title_type %>%
    factor(levels = input$selected_type)
  counts <- table(types)

  HTML(paste("There are", counts, input$selected_type, “participants in this dataset.<br>"))
})
```

Putting it altogether

`nhanes-apps/nhanes-05.R`



DEMO

Also notice

- HTML tags for visual separation
- `req()`

When to use reactives

- By using a reactive expression for the subsetted data frame, we were able to get away with subsetting once and then using the result twice
- In general, reactive conductors let you
  - not repeat yourself (i.e. avoid copy-and-paste code) which is a maintenance boon)
  - decompose large, complex (code-wise, not necessarily CPU-wise) calculations into smaller pieces to make them more understandable
- These benefits are similar to what happens when you decompose a large complex R script into a series of small functions that build on each other

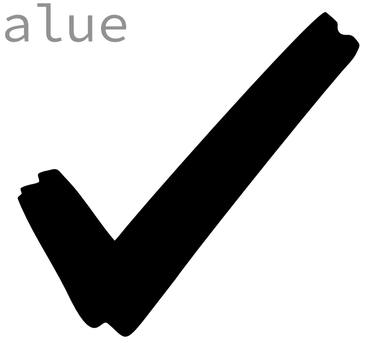
Suppose we want to plot only a random sample of participants, of size determined by the user. What is wrong with the following?

```
# Server
# Create a new data frame that is a sample of n_samp
# observations from NHANES
NHANES_sample <- sample_n(NHANES_sample(), input$n_samp)

# Plot the sampled participants
output$scatterplot <- renderPlot({
  ggplot(data = NHANES_sample,
         aes_string(x = input$x, y = input$y,
                    color = input$z)) +
    geom_point(…)
})
```

```
# Server
# Create a new data frame that is a sample of n_samp
# observations from NHANES
NHANES_sample <- reactive({
  req(input$n_samp)      # ensure availability of value
  sample_n(NHANES_sample(), input$n_samp)
})

# Plot the sampled participants
output$scatterplot <- renderPlot({
  ggplot(data = NHANES_sample(),
         aes_string(x = input$x,
                    y = input$y,
                    color = input$z)) +
    geom_point(…)
})
```



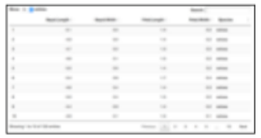
SOLUTION

Render functions

Render functions

```
render*({ [code_chunk] })
```

- Provide a code chunk that describes how an output should be populated
- The output will update in response to changes in any reactive values or reactive expressions that are used in the code chunk



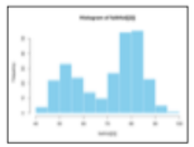
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

works with

dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

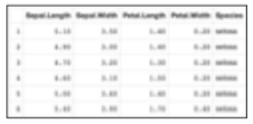


renderPlot(expr, width, height, res, …, env, quoted, func)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

renderPrint(expr, env, quoted, func, width)

verbatimTextOutput(outputId)



renderTable(expr,…, env, quoted, func)

tableOutput(outputId)

foo

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, …)
& **htmlOutput**(outputId, inline, container, …)

Recap

```
render*({ [code_chunk] })
```

- These functions make objects to display
- Results should always be saved to `output$`
- They make an observer object that has a block of code associated with it
- The object will rerun the entire code block to update itself whenever it is invalidated

Implementation

Implementation of reactives

- **Reactive values** – `reactiveValues()`:
  - e.g. `input:` which looks like a list, and contains many individual reactive values that are set by input from the web browser
- **Reactive expressions** – `reactive()`: they depend on reactive values and observers depend on them
  - Can access reactive values or other reactive expressions, and they return a value
  - Useful for caching the results of any procedure that happens in response to user input
  - e.g. reactive data frame subsets we created earlier
- **Observers** – `observe()`: they depend on reactive expressions, but nothing else depends on them
  - Can access reactive sources and reactive expressions, but they don't return a value; they are used for their side effects
  - e.g. output object is a reactive observer, which also looks like a list, and contains many individual reactive observers that are created by using reactive values and expressions in reactive functions

Reactive expressions vs. observers

- Similarities: Both store expressions that can be executed
- Differences:
  - Reactive expressions return values, but observers don't
  - Observers (and endpoints in general) eagerly respond to reactives, but reactive expressions (and conductors in general) do not
  - Reactive expressions must not have side effects, while observers are only useful for their side effects

Your turn

Debug the following app scripts:

- `review/whats-wrong.R`
- `review/mult-3.R`
- `review/add-2.R`

5m 00s