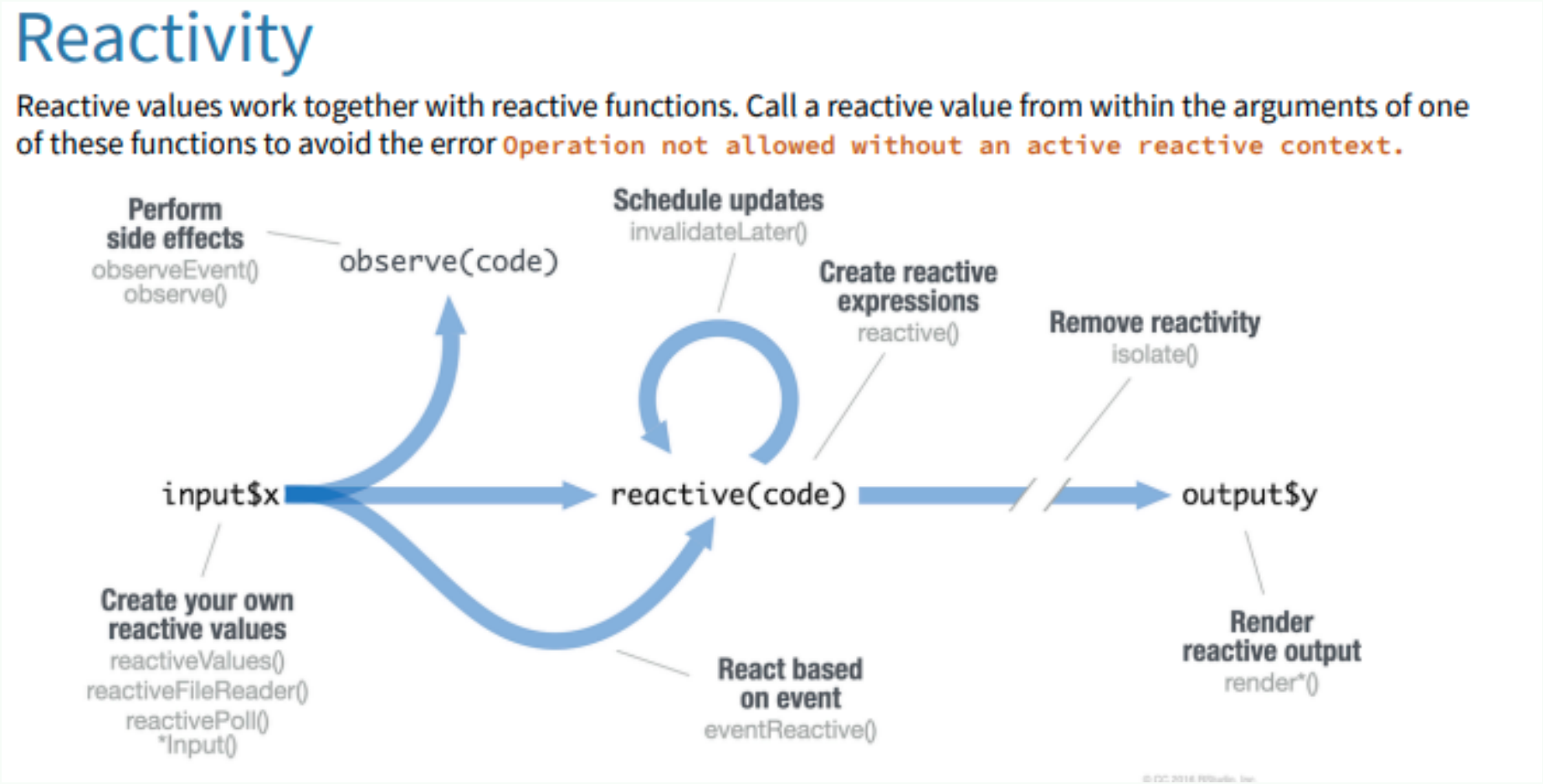
Shiny Reactivity

Spring 2023

May 10 2023

Shiny reactivity



Shiny reactivity 101

Shiny Reactivity: Responding to User Input

- *Shiny reactivity is a system for managing data dependencies and updates in response to user input.*
- *Unlike typical R code, Shiny apps are event-driven and react to user interactions.*
- *Shiny monitors events and triggers the necessary code to update the app's state and output.*

Reactive Expressions: Connecting Inputs and Outputs

"Reactive expressions transform reactive inputs into reactive outputs."

- ***Reactive expressions** are the building blocks of Shiny reactivity.*
- *They are used to manage data dependencies and cache results.*
- *They automatically update when input values change, ensuring your app stays up-to-date.*

Examples of common use cases

- Accessing a database based on user input.
- Reading data from a file when the user selects a new file.
- Downloading data over the network in response to user actions.
- Performing expensive computations that depend on user input.

Reactive values: Storing and Sharing Data

"Reactive values contain values that can be read by other reactive objects."

- ***Reactive values** allow you to store and share data across different parts of your Shiny app.*
- *The input object is a special instance of **ReactiveValues** that holds user inputs.*

```
library(shiny)
ui <- fluidPage(
  titlePanel("Updating Plot Based on User Input"),
  sidebarLayout(
    sidebarPanel(
      numericInput("obs", "Number of observations:",
                   value = 100, min = 1)
    ),
    mainPanel( plotOutput("distPlot"))
  )
)
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
shinyApp(ui, server)
```

Triggering reactivity with `eventReactive()`

- ***`eventReactive()` is used to create reactive expressions that only update when a specific event occurs.***
- ***It takes an input, typically an action button, to trigger the update.***
- ***Useful for controlling when calculations or updates occur, reducing unnecessary computation.***

```
ui <- fluidPage(
  mainPanel(
    actionButton("addButton", "Add 1"),
    textOutput("result")
  )
)

server <- function(input, output) {
  sum_so_far <- eventReactive(input$addButton, {
    if (is.null(input$addButton)) {
      0
    } else {
      input$addButton
    }
  })

  output$result <- renderText({
    paste("Sum so far:", sum_so_far())
  })
}

shinyApp(ui, server)
```

`actionButton()`: Triggering Actions on Demand

"actionButton() creates a button that lets users manually trigger reactive events."

- *Use `actionButton()` to create a button in the UI that users can click to trigger specific actions.*
- *Combine with `eventReactive()` to execute code only when the button is clicked.*
- *Ideal for cases where you want to give users control over when certain actions are executed, rather than updating automatically.*

```
ui <- fluidPage(
  mainPanel(
    numericInput("obs", "Number of observations", v
    actionButton("updateButton", "Update Plot"),
    plotOutput("distPlot")
  )
)

server <- function(input, output) {
  data_to_plot <- eventReactive(input$updateButton,
    rnorm(input$obs)
  })

  output$distPlot <- renderPlot({
    hist(data_to_plot())
  })
}

shinyApp(ui, server)
```

Storing and managing state with `reactiveValues()`

- *`reactiveValues()` is used to create mutable, reactive objects.*
- *They can store multiple named values that can be updated independently.*
- *Useful for managing complex state or sharing data between multiple reactive expressions.*

```
ui <- fluidPage(
  mainPanel(
    numericInput("numInput", "Enter a number:", value = 1),
    actionButton("incrementButton", "Increment"),
    textOutput("incrementedValue")
  )
)
server <- function(input, output) {
  values <- reactiveValues(num = 1)
  incrementedValue <- reactive({
    if (input$incrementButton > 0) {
      values$num <- input$numInput + input$incrementButton
    }
    values$num
  })
  output$incrementedValue <- renderText(paste("Incremented
  incrementedValue()))
}
shinyApp(ui, server)
```

Observers: Responding to Changes in Reactive Values

`observeEvent()` is used to create observers that react to specific events, often triggered by user interactions like button clicks.

- *Executes code in response to a specified event*
- *Can be used with `actionButton()` to perform actions when a button is clicked*
- *Doesn't return a value, but causes side effects (e.g., updating output or triggering other reactive expressions)*

```
ui <- fluidPage(
  mainPanel(
    actionButton("showAlert", "Show Alert"),
    textOutput("alertCount")
  )
)
server <- function(input, output) {
  alert_counter <- reactiveValues(count = 0)
  observeEvent(input$showAlert, {
    showModal(modalDialog(
      title = "Alert",
      "This is an alert message!"
    ))
    alert_counter$count <- alert_counter$count + 1
  })
  output$alertCount <- renderText({
    paste("Number of alerts shown:", alert_counter$count)
  })
}
shinyApp(ui, server)
```

Controlling reactivity with `isolate()`

- *The `isolate()` function is used to prevent reactivity within a reactive context.*
- *It allows you to use an input or reactive value without triggering a reaction.*
- *Useful when you want to control when a reactive expression or output updates.*

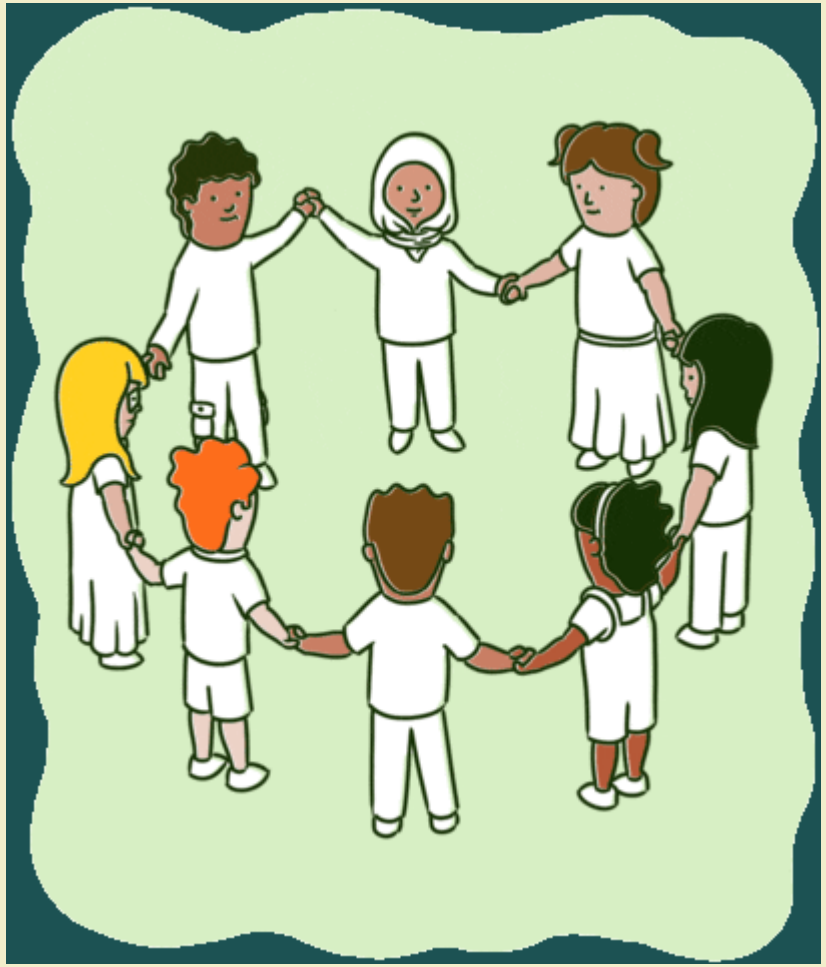
```
ui <- fluidPage(
  mainPanel(
    textInput("textInput", "Enter some text"),
    actionButton("submitButton", "Submit"),
    textOutput("outputText")
  )
)

server <- function(input, output) {
  observeEvent(input$submitButton, {
    output$outputText <- renderText({
      paste("You submitted:", isolate(input$textInput))
    })
  })
}

shinyApp(ui, server)
```

GROUP ACTIVITY 1

15:00



- *Let's go over to maize server/ local Rstudio and our class moodle*
- *Get the class activity 19.Rmd file*
- *Let's work on the class activity together*
- *Ask me questions*