Creating Custom Interactive Applications with R and Shiny

Chris Runyon, Josh Goodman,
and Marcus Walker

Day 1

Workshop Overview

Workshop Resources

- https://tinyurl.com/NCMEShiny2021

Day 1: Basics

Day 2 Part 1: More Basics / Developing a Shiny App

Day 2 Part 2: App Development Resources

Additional Resources in above Github repository

Logistic Details

Learning Shiny - like many things - requires having a bit of tolerance for ignoring the things you that aren’t covered until they are covered / you can look them up later.

- The app has several components / aspects to it, really no other way to cover the material.

This workshop has several hands-on components / in-class activities.

Day 1 Learning Objectives

- Be able to describe the basic shiny environment.

- Have a basic understanding of the fundamental aspects of the layout of a shiny app and be able to manipulate these layouts.

- Have a basic understanding of the input() and output() components of a shiny app and be able to select and utilize these commands.

- Know how to incorporate tabular and graphical displays into your shiny app.
  - Will learn how to download / export tables and graphics.

- Have an initial understanding of the reactivity components of shiny.

Brief Introduction

Chris Runyon, Measurement Scientist, NBME

crunyon@nbme.org

Josh Goodman, Senior Director of Psychometrics Test Development, NCCPA

joshuag@nccpa.net

Marcus Walker, Data Scientist, NCCPA

marcusw@nccpa.net

What is Shiny?

"Shiny is an R package that makes it easy to build interactive web apps straight from R. You can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards. You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions." - https://shiny.rstudio.com/

It is an interface to help you, your colleagues, your clients, and your customers to be able to do *something* without having to know any R programming.

- *something* = access information / data, enter information / data, perform an analysis, create a report, etc.

What is Shiny?

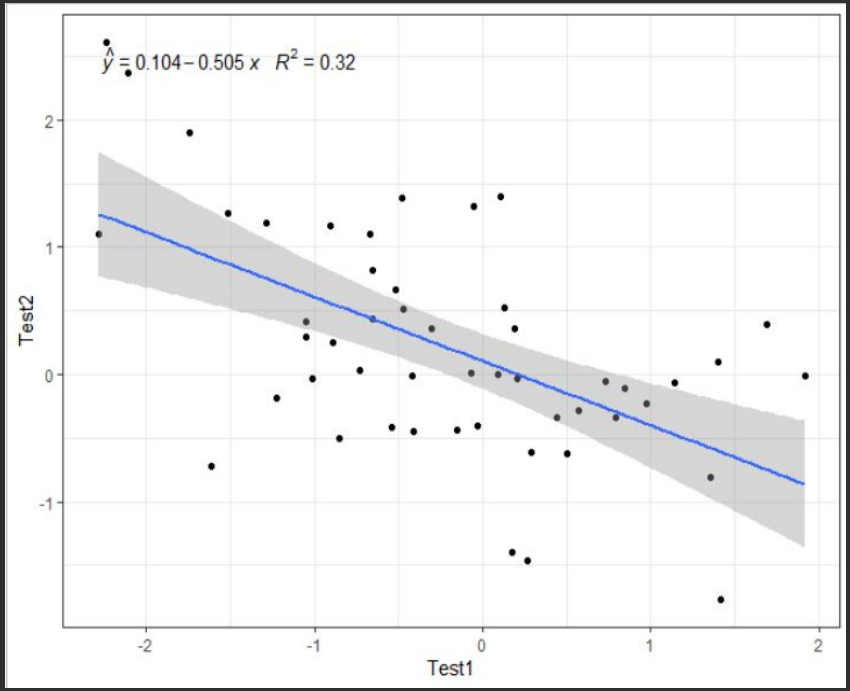
Shiny utilizes R syntax as a shortcut to write HTML / CSS / JavaScript.

- No knowledge of these languages is necessary to create your app.
- Gaining some small knowledge of one of these languages can extend your Shiny functionality.
- Knowing (or having a colleague that is fluent) in these languages can significantly extend the capabilities of the app.

Why use shiny?

Take it away Josh!

A Simple Shiny App



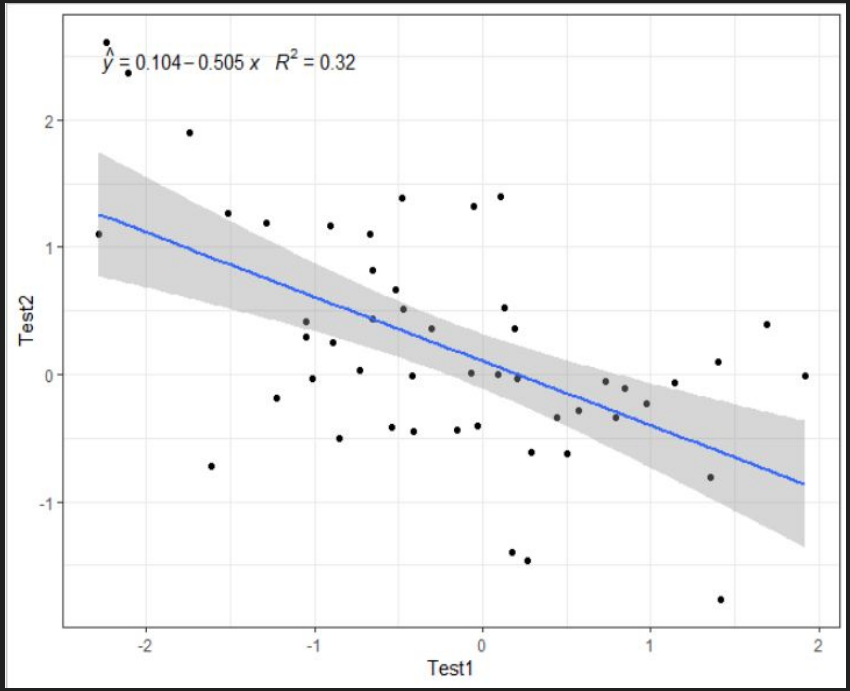
Shiny apps are just R code

- A Shiny application is developed using the same basic R coding methods you currently use, but with some package specific functions.
- We can leverage what we already know about R and make use of it within our applications.
  - You can use your favorite packages within Shiny applications.
  - You can use build an implement custom functions within applications.
  - Your data wrangling tricks will still work.

Shiny apps are just R code

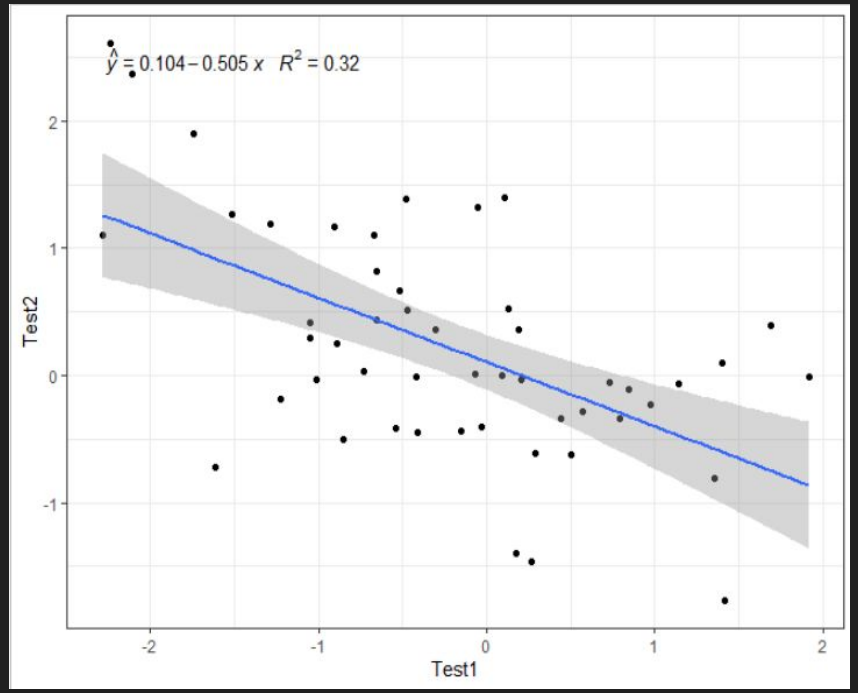
Let's consider a task that we might tackle with some routine R coding.

- We want to generate some bivariate data.
- We want the two variables to have a known correlation.
- Our desired output is a scatter plot that shows the a line of best linear fit and a confidence interval around that line.



Simple R code

```
######################### Simple R CODE ####################
library(tidyverse)
library(ggpmisc)
library(MASS)
######## INPUTS ##########
means<-c(0,0)
sds<-c(1,1)
N<-50
r<- -.65

############# Make the Plot ################################
my.formula <- y ~ x
ggplot(as.data.frame(
  mvrnorm(N,means, matrix(c(1,r,r,1),2,2))) %>%
    rename(Test1=V1,Test2=V2) , aes(x=Test1,y=Test2))+
  geom_point()+geom_smooth(method=lm)+
  stat_poly_eq(formula = my.formula,
               eq.with.lhs = "italic(hat(y))~`=`~",
               aes(label = paste(..eq.label.., ..rr.label..,
                                 sep = "~~~")),
               parse = TRUE)+theme_bw()
#############################################################
```



As a function

```r
########################## SIMPLE FUNCTION ################
ShinyDemoFunction<-function(N,r){
  library(tidyverse)
  library(ggpmisc)
  library(MASS)

  my.formula <- y ~ x
  display<-ggplot(as.data.frame(
    mvrnorm(N,c(0,0), matrix(c(1,r,r,1),2,2))) %>%
      rename(Test1=V1,Test2=V2), aes(x=Test1,y=Test2))+
    geom_point()+geom_smooth(method=lm)+
    stat_poly_eq(formula = my.formula,
                 eq.with.lhs = "italic(hat(y))~`=`~",
                 aes(label = paste(..eq.label..,..rr.label..,
                                   sep = "~~~")),
                 parse = TRUE)+theme_bw()
  display
}
```

- If we wanted to simulate data with new parameters, we'd have to change our inputs and rerun the code.
- If this was task we would need to run repeatedly or we wanted to incorporate it into some larger context, we might turn it into a function.

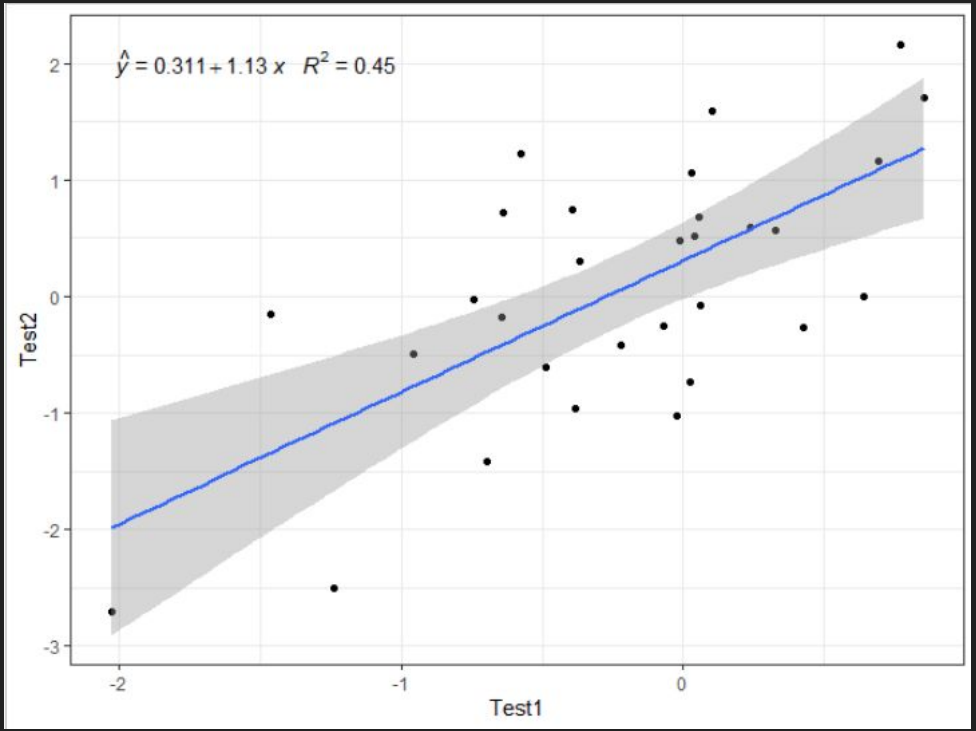
As a function

```
ShinyDemoFunction(N=30,r=.6)


ShinyDemoFunction(N=150,r=.83)


ShinyDemoFunction(N=50,r=-.83)
```

- As a function, we can easily manipulate our arguments to produce plots.
- We could embed and use the function in other code.
- We would be actively working within R.
- We would need to execute the code repeatedly to see new plots.

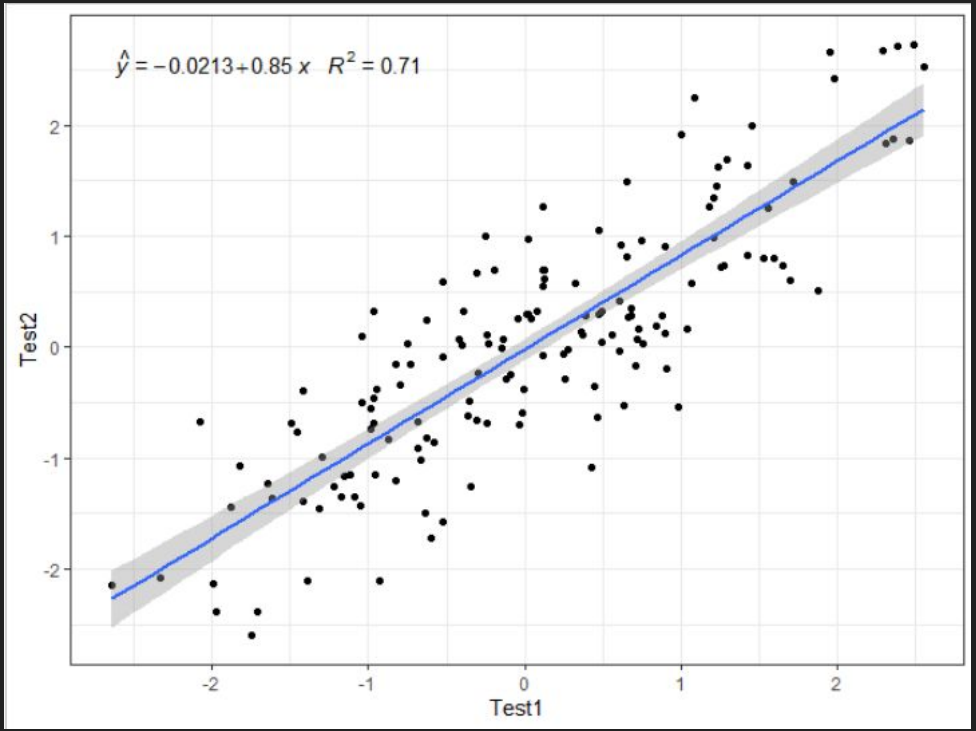
As a function

```
ShinyDemoFunction(N=30,r=.6)

ShinyDemoFunction(N=150,r=.83)

ShinyDemoFunction(N=50,r=-.83)
```



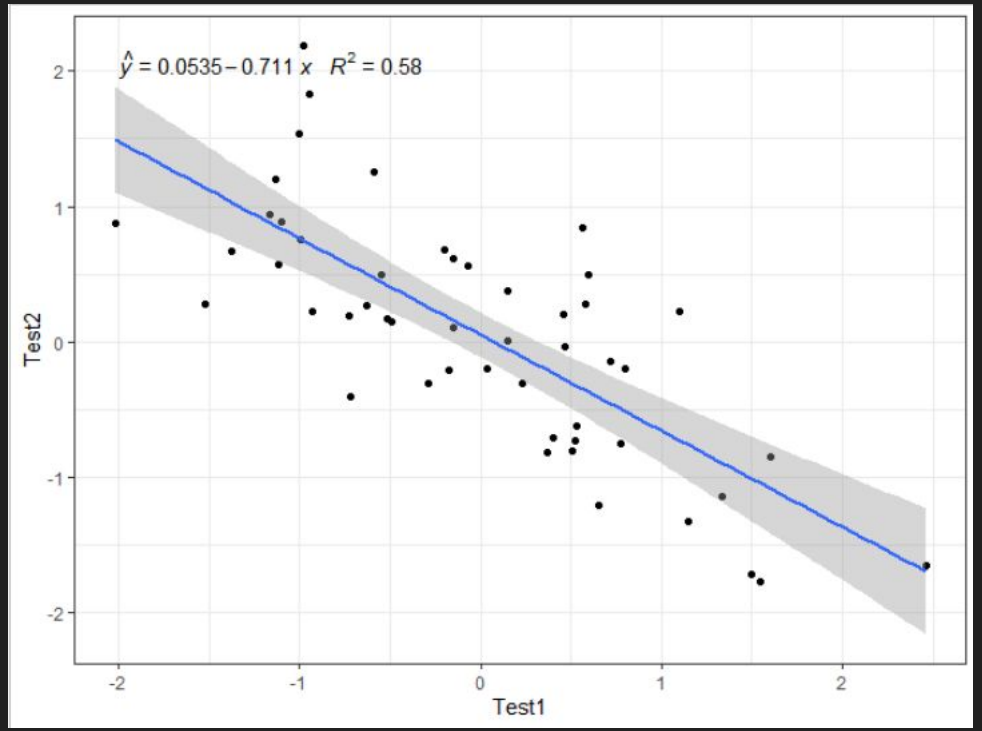
As a function

```
ShinyDemoFunction(N=30,r=.6)

ShinyDemoFunction(N=150,r=.83)

ShinyDemoFunction(N=50,r=-.83)
```



As a function

```
ShinyDemoFunction(N=30,r=.6)


ShinyDemoFunction(N=150,r=.83)


ShinyDemoFunction(N=50,r=-.83)
```



The same task...but as a Shiny app

- We can use the same base code to create a Shiny app.
  - The app can rerun the simulation and generate a new plot without executing the code over and over.
  - We can make the simulation interactive--as we manipulate the inputs, we can see changes in the output
- Let’s check out the app version of this task (0_simpleApp.r)

Peeking under the hood

```
library(tidyverse)
library(ggpmisc)
library(MASS)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput(inputId = "N",  "Number of observations:", min = 10, max = 500, value = 30),
      sliderInput(inputId = "R",  "Cor(x,y):", min = -1.00, max = 1.00, value = .6,step=.01)
    ),
    mainPanel(
      plotOutput(outputId = "regPlot")
      )
    )
  )
```

Peeking under the hood

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

  output$regPlot<- renderPlot({
    my.formula <- y ~ x
    ggplot(data=as.data.frame(mvrnorm(input$N,c(0,0), matrix(c(1,input$R,input$R,1),2,2))) %>%
             rename(Test1=V1,Test2=V2), aes(x=Test1,y=Test2))+
      geom_point()+geom_smooth(method=lm)+
      stat_poly_eq(formula = my.formula,
                   eq.with.lhs = "italic(hat(y))~`=`~",
                   aes(label = paste(..eq.label.., ..rr.label.., sep = "~~~")),
                   parse = TRUE)+theme_bw()
  })

}
```

Shiny is an analytic solution

- We can build internal applications and professional-looking external tools.
- The ability to create custom application is limited only by your R and Shiny programming skills.
- Applications are shareable—tasks can be completed by any user, at anytime, with predictable outcomes.
- Once an app is published or shared, reliance on R is minimized.

Building Your Shiny App

What is in a Shiny application?

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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

User **I**nterface - controls layout and appearance

The engine of the app

Function that launches the app locally

UI and Server

UI = User Interface

- Preset options for what the user can do.
- Result of what the user does.

Server

- Takes the information from the preset options, does something to that information, and then returns the result of that something back to the user.

UI

Layout Functions - set up the visual structure of the page

Input Control - lets the user interact with the app

Output Control - where the output will appear on the UI

Server

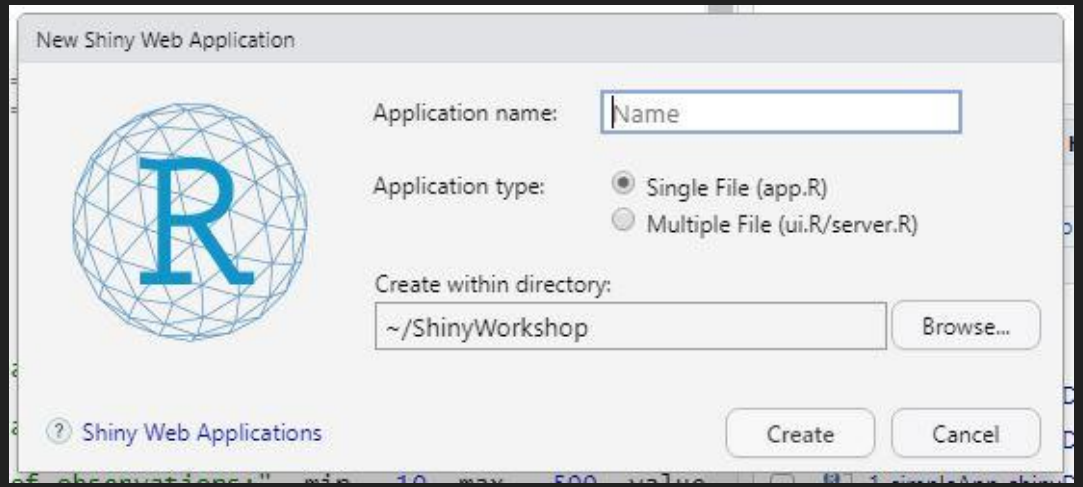
A function that takes the information provided by the input controls and produces the output

UI and Server

Two options for structuring the Shiny app:

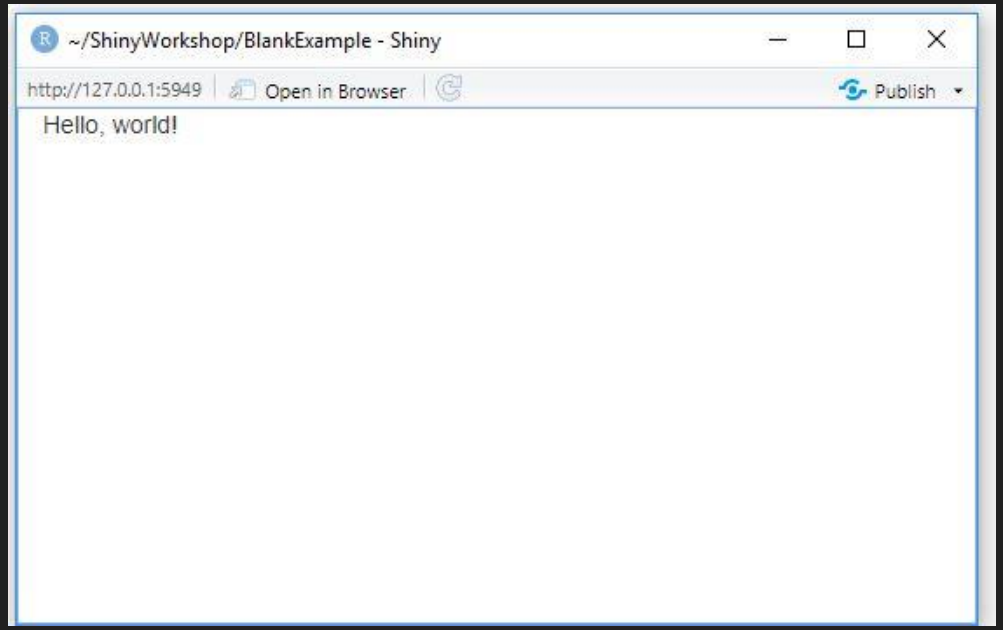
Single File

Multiple File

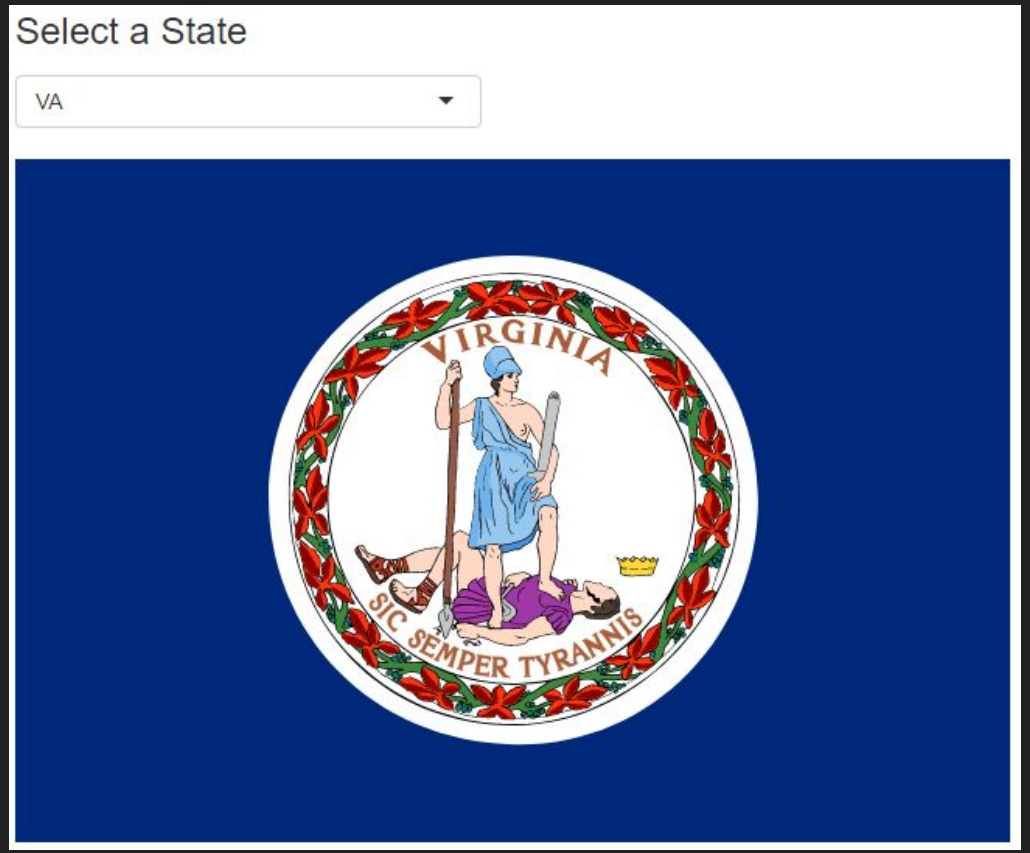


UI and Server (HelloWorld.R)

```r
library(shiny)

# Define UI
ui <- fluidPage(
    "Hello, world!"
)

# Define server logic required
server <- function(input, output, session){

}

# Run the application
shinyApp(ui, server)
```



UI and Server (PickState.R)



UI and Server (PickState.R)

```r
ui <- fluidPage(

  fluidRow(
    column(4,
      ### Input
      selectInput("select", label = h3("Select a State"),
                  choices = list(state.abb = state.abb),
                  selected = 1),
      ### Output
      imageOutput("state_flag")
    ))
  )
```

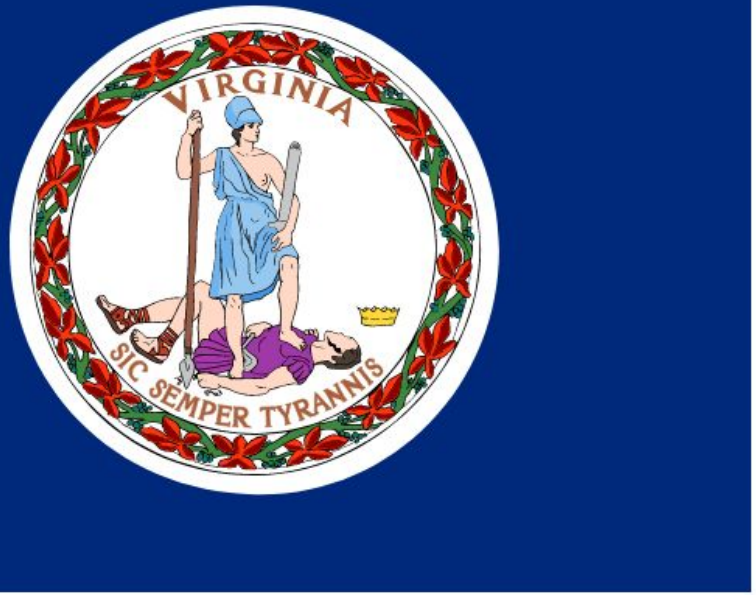
```r
server <- function(input, output) {

  output$state_flag <- renderImage({

    return(list(
      src = paste0("www/", tolower(input$select), ".png"),
      contentType = "www/png",
      alt = "Flag"
      ))
    }, deleteFile = FALSE)
}
```

UI and Server (PickState.R)

Input

Select a State

VA



Output

Layout

The *layout* of the shiny app is the basic visual structure of the page. The layout is shiny is powered by Bootstrap, which is a popular (and powerful) HTML/CSS framework.

The layout can range from very simple (such as the PickState.R example) to the very complex.

Basic concepts now. On Day 2, after learning more about the inputs, outputs, reactivity, etc. of Shiny, we will introduce some more complex layout options.

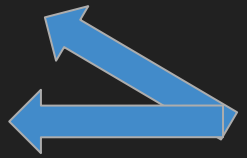
fluidPage()

This is the container for your UI. Sets up HTML, CSS, and JavaScript behind the scenes to enable easy programming in shiny.

fluidPage() - width is fluid; width of the app depends on components specified in the other parts of the UI

fixedPage() - utilizes a maximum fixed width; prevents display from being too wide

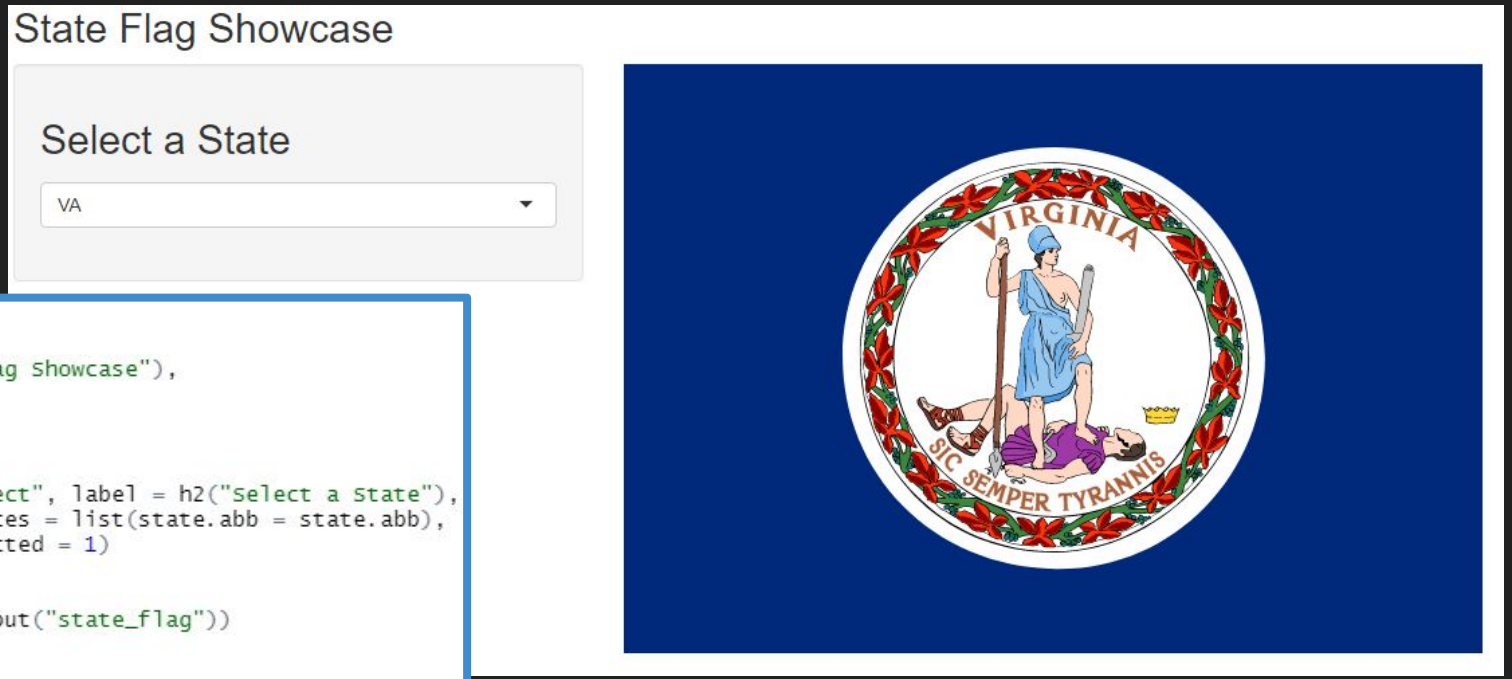
fillPage() - helps to fill the whole screen



Especially helpful when you expect users to be on a variety of devices / interfaces.

sidebarLayout() (PickStateSidebar.R)

Divides the app into a sidebarPanel and a mainPanel, which are displayed side-by-side.




```
ui <- fluidPage(

  titlePanel("State Flag Showcase"),

  sidebarLayout(

    sidebarPanel(
      selectInput("select", label = h2("Select a State"),
                  choices = list(state.abb = state.abb),
                  selected = 1)
    ),

    mainPanel(imageOutput("state_flag"))
  )
)
```

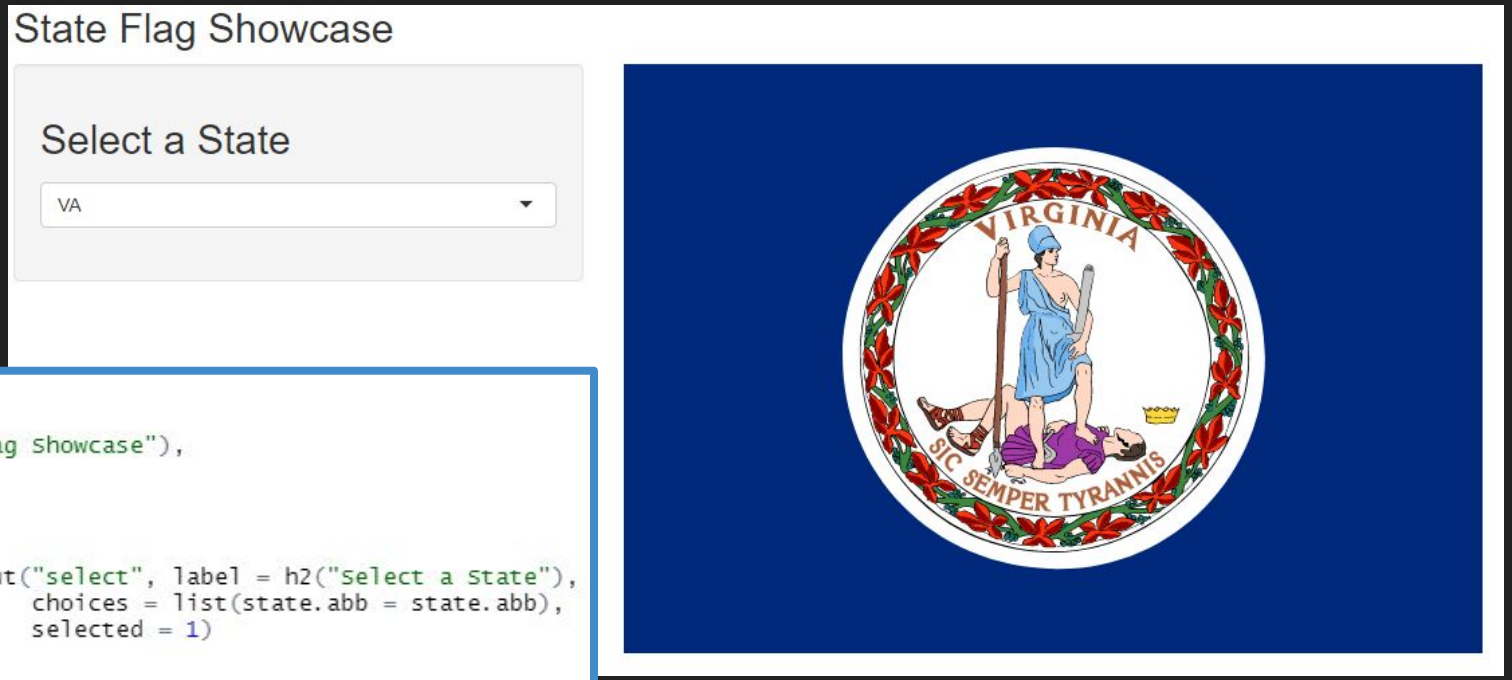
Grid Layout

The Bootstrap system on which shiny is based works off a 12-wide grid system.

You can specify widths of panels in fluidRows; needs to add up to 12. If the objects in your rows exceed 12, the object(s) exceeding 12 are put onto another row.

We can re-create the sidebarLayout() above using the grid layout system.

Grid Layout (PickStateGrid.R)



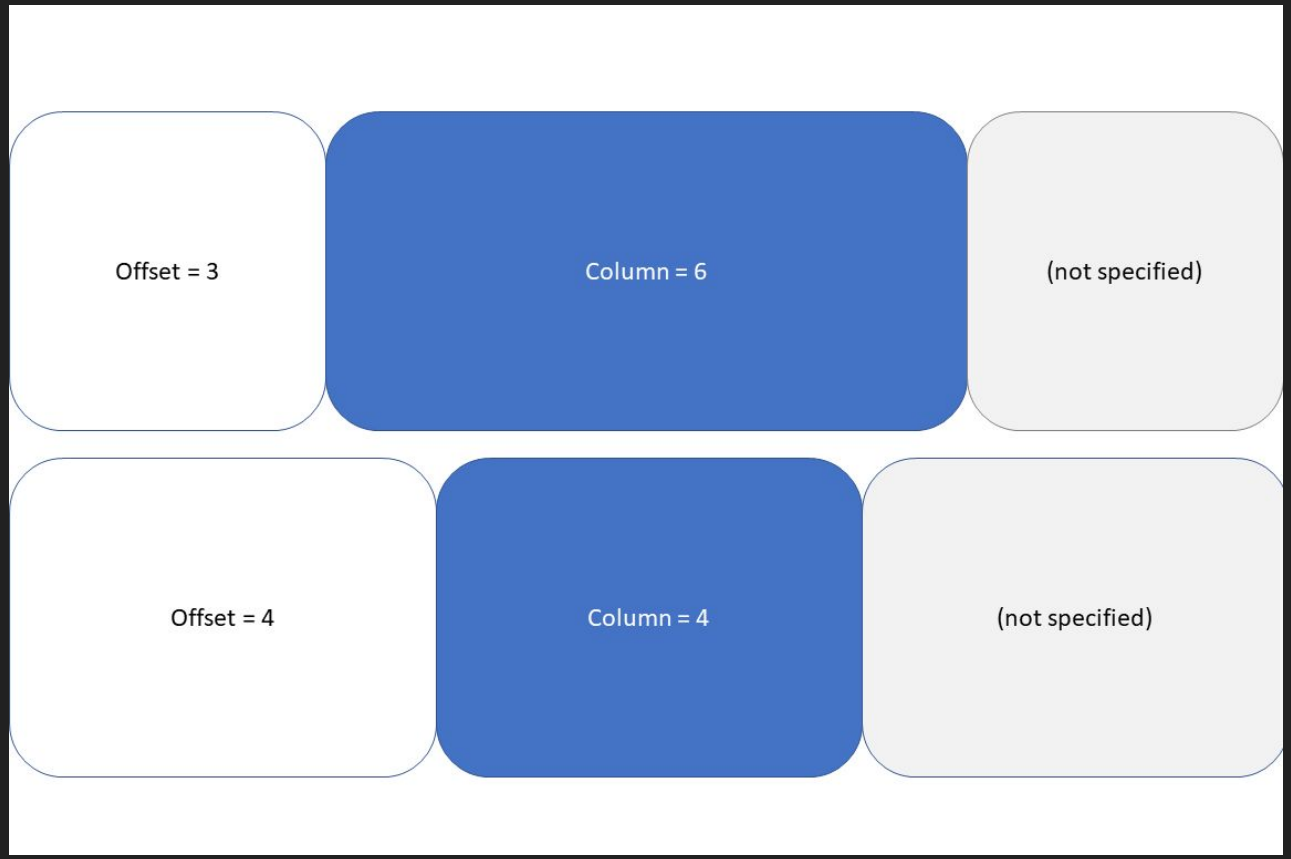

```r
ui <- fluidPage(

  titlePanel("State Flag Showcase"),

  fluidRow(
    column(4,
           wellPanel(
             selectInput("select", label = h2("Select a State"),
                         choices = list(state.abb = state.abb),
                         selected = 1)
           )),
    column(8,
           imageOutput("state_flag"))
  )
)
```

Grid Layout (PickStateGridTopFlag.R)

Grid Layout (PickStateGridTopFlag.R)

```
ui <- fluidPage(

  titlePanel("State Flag Showcase"),

  fluidRow(
    column(6, offset = 3,
           imageOutput("state_flag"))
    ),

  hr(),

  fluidRow(
    column(4, offset = 4,
           wellPanel(
             selectInput("select", label = h2("Select a State"),
                         choices = list(state.abb = state.abb),
                         selected = 1)
           ))
    )
  )
```

Grid Layout (PickStateGridTopFlag.R)



...Panel()

`absolutePanel()`
Panel position set rigidly (absolutely), not fluidly

`conditionalPanel()`
A JavaScript expression determines whether panel is visible or not.

`fixedPanel()`
Panel is fixed to browser window and does not scroll with the page

`headerPanel()`
Panel for the app's title, used with pageWithSidebar()

`inputPanel()`
Panel with grey background, suitable for grouping inputs

`mainPanel()`
Panel for displaying output, used with pageWithSidebar()

`navlistPanel()`
Panel for displaying multiple stacked tabPanels(). Uses sidebar navigation

`sidebarPanel()`
Panel for displaying a sidebar of inputs, used with pageWithSidebar()

`tabPanel()`
Stackable panel. Used with navlistPanel() and tabsetPanel()

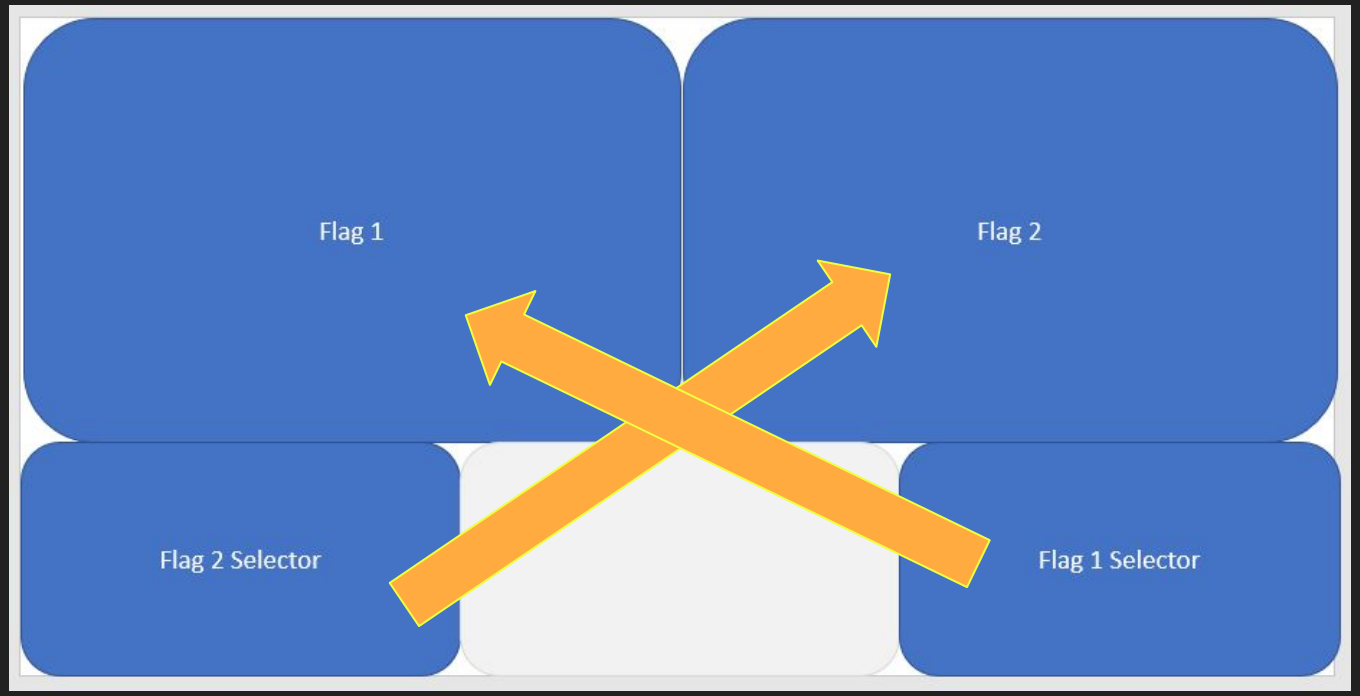
`tabsetPanel()`
Panel for displaying multiple stacked tabPanels(). Uses tab navigation

`titlePanel()`
Panel for the app's title, used with pageWithSidebar()

`wellPanel()`
Panel with grey background.

Activity 1a

Make an app where the top row is 2 state flag outputs, the bottom row is two selection bars spaced at both ends of the row, AND the selection bar under the left flag controls the right flag and vice versa.



Activity 1a Solution

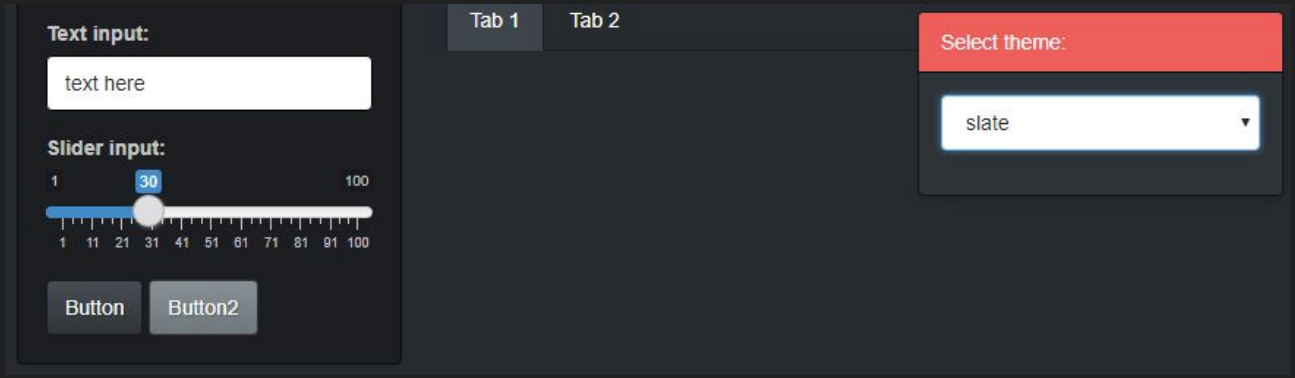
Themes

Themes are a quick and easy way to change the styling of the shiny app.

> “styling” = font, colors of the different elements

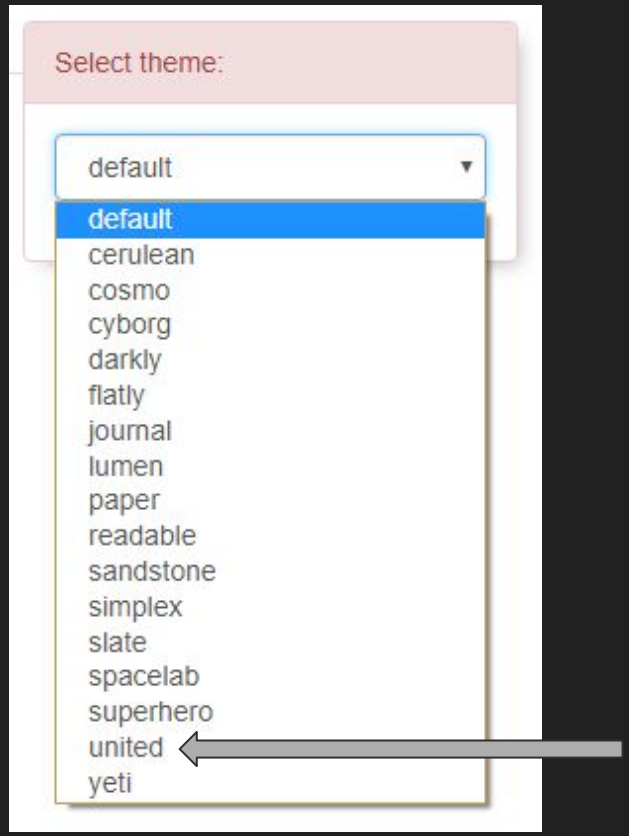
The ‘shinythemes’ package is one of the quickest ways to change the theme of your package / layout.

shinytheme examples (shinythemeExamples.R)

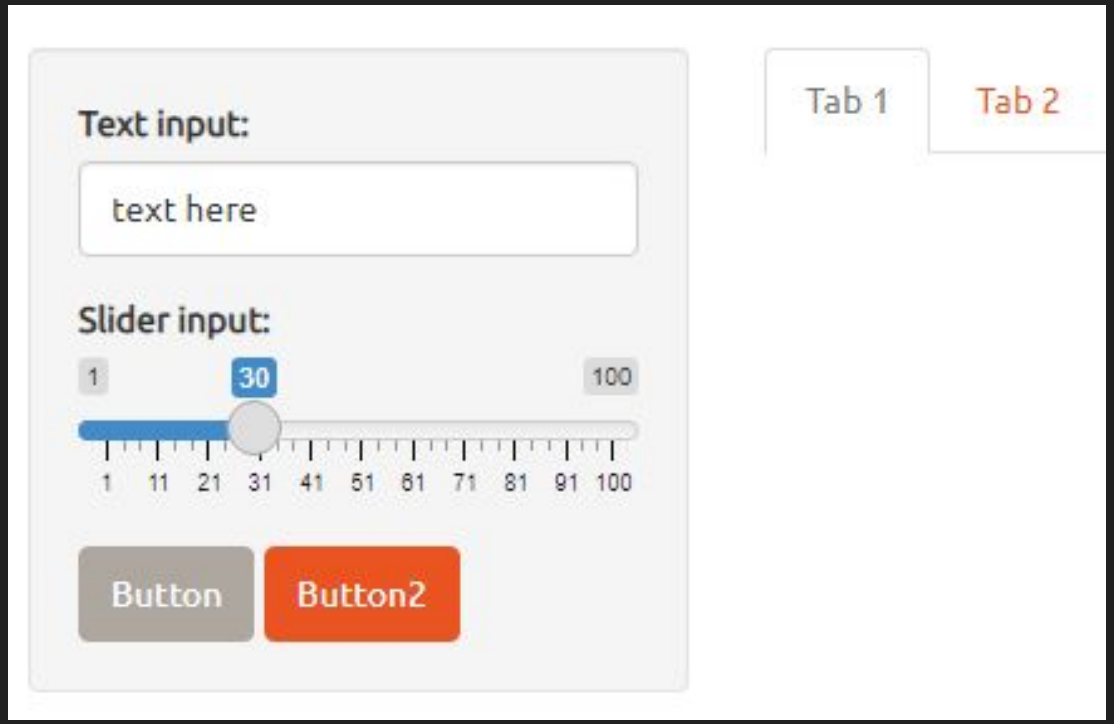


shinytheme examples (shinythemesUnited.R)

```r
shinyApp(
  ui = fluidPage(
    theme = shinytheme("united"),
    sidebarPanel(
      textInput("txt", "Text input:", "text here"),
      sliderInput("slider", "Slider input:", 1, 100, 30),
      actionButton("action", "Button"),
      actionButton("action2", "Button2", class = "btn-primary")
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Tab 1"),
        tabPanel("Tab 2")
      )
    )
  ),
  server = function(input, output) {}
)
```



shinytheme examples (shinythemesUnited.R)



Themes

Themes are a quick and easy way to change the styling of the shiny app.

“styling” = font, colors of the different elements

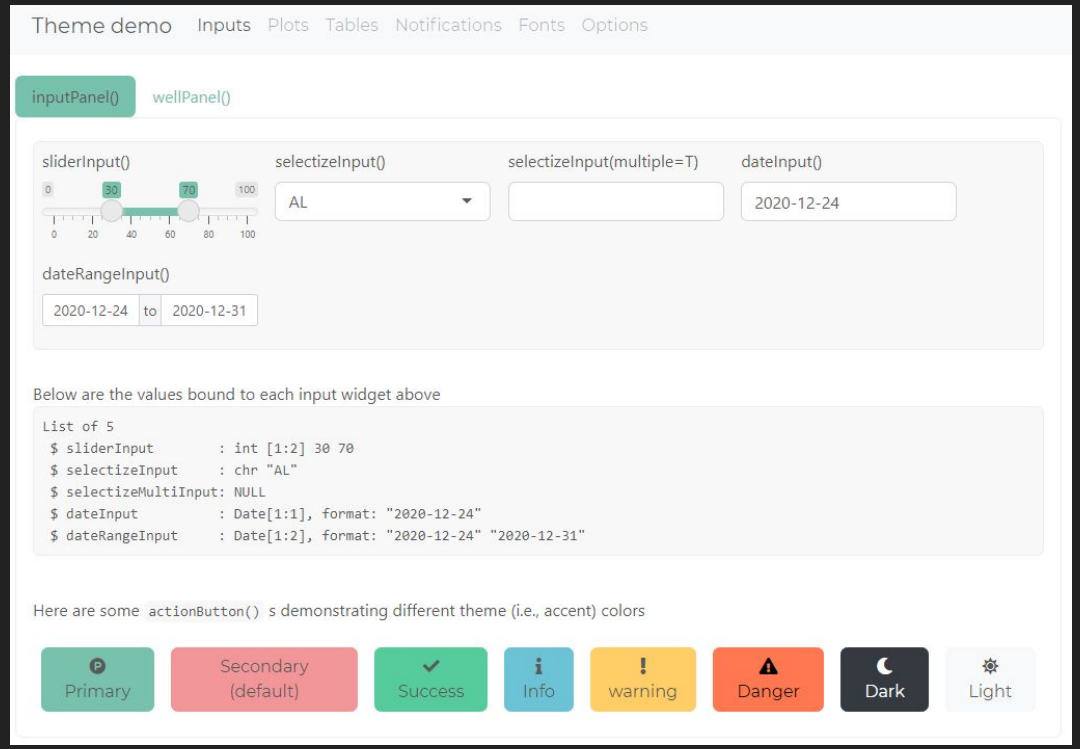
The ‘shinythemes’ package is one of the quickest ways to change the theme of your package / layout.

“bslib” package - “bootstrap library” - additional themes / customizing elements of the themes

- Works off of the Bootstrap framework mentioned earlier

bslib - preset theme example (bslibMinty.R)

```
shinyApp(
  navbarPage(
    theme = bs_theme(bootswatch = "minty"),
    title = "Theme demo",
    collapsible = TRUE,
    id = "navbar",
    tabPanel(
```



bslib - changing elements

```
# Shiny usage
navbarPage(
  theme = bs_theme(
    bg = "#101010",
    fg = "#FDF7F7",
    primary = "#ED79F9",
    base_font = font_google("Prompt"),
    code_font = font_google("JetBrains Mono")
  ),
  ...
)
```

pssst…. fonts.google.com is awesome!

bslib - changing elements (PickStateSidebarBSLIB.R)

State Flag Showcase

Select a State

WA

```
ui <- fluidPage(

  titlePanel("State Flag Showcase"),

  theme = bs_theme(
    bg = "#89cff0",
    fg = "#FFFF00",
    primary = "#FF0000",
    base_font = font_google("Odibee Sans")
  ),
```

Tags - Shiny HTML Tags Glossary

tags

The `shiny::tags` object contains R functions that recreate 110 HTML tags.

```
names(tags)
##   [1] "a"          "abbr"       "address"    "area"       "article"
##   [6] "aside"      "audio"      "b"          "base"       "bdi"
##  [11] "bdo"        "blockquote" "body"       "br"         "button"
##  [16] "canvas"     "caption"    "cite"       "code"       "col"
##  [21] "colgroup"   "command"    "data"       "datalist"   "dd"
##  [26] "del"        "details"    "dfn"        "div"        "dl"
##  [31] "dt"         "em"         "embed"      "eventsource" "fieldset"
##  [36] "figcaption" "figure"     "footer"     "form"       "h1"
##  [41] "h2"         "h3"         "h4"         "h5"         "h6"
##  [46] "head"       "header"     "hgroup"     "hr"         "html"
##  [51] "i"          "iframe"     "img"        "input"      "ins"
##  [56] "kbd"        "keygen"     "label"      "legend"     "li"
##  [61] "link"       "mark"       "map"        "menu"       "meta"
##  [66] "meter"      "nav"        "noscript"   "object"     "ol"
##  [71] "optgroup"   "option"     "output"     "p"          "param"
##  [76] "pre"        "progress"   "q"          "ruby"       "rp"
##  [81] "rt"         "s"          "samp"       "script"     "section"
##  [86] "select"     "small"      "source"     "span"       "strong"
##  [91] "style"      "sub"        "summary"    "sup"        "table"
##  [96] "tbody"      "td"         "textarea"   "tfoot"      "th"
## [101] "thead"      "time"       "title"      "tr"         "track"
## [106] "u"          "ul"         "var"        "video"      "wbr"
```

What is Shiny? - a tags example

"**Shiny** is an R package that makes it easy to build interactive web apps straight from R. You can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards. You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions." - https://shiny.rstudio.com/

It is an interface to help you, your colleagues, your clients, and your customers to be able to do *something* without having to know any R programming.

What is Shiny? (AboutShiny.R)

Shiny is an R package that makes it easy to build interactive web apps straight from R. You can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards. You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions. - https://shiny.rstudio.com/

It is an interface to help you, your colleagues, your clients, and your customers to be able to do *something* **without having to know any R programming.**

What is Shiny?

```
ui <- fluidPage(

  div(img(src='blueShiny.png', width = 500), style = "text-align: center;"),

  br(),

  fluidRow(
    column(6, offset = 3,
           br(),
           p(tags$b("Shiny"), "is an R package that makes it easy to build interactive
             web apps straight from R. You can host standalone apps on a webpage
             or embed them in ",
             a("R Markdown", href="https://rmarkdown.rstudio.com/", target="_blank"),
             " documents or build ",
             a("dashboards. ", href="http://rstudio.github.io/shinydashboard/", target="_blank"),
             "You can also extend your Shiny apps with ",
             a("CSS themes, ", href="http://rstudio.github.io/shinythemes/", target = "_blank"),
             a("htmlwidgets, ", href = "http://www.htmlwidgets.org/", target = "_blank"), "and JavaScript actions. - ",
             a("https://shiny.rstudio.com/", href = "https://shiny.rstudio.com/", target = "_blank")))),

  br(),
  fluidRow(
    column(6, offset = 3,
           p("It is an interface to help you, your colleagues, your clients, and your
             customers to be able to do ", em("something"),  tags$b("without having to know
             any R programming.")))
  ))
```

Images

Including images in a Shiny app is easy. The only trick is that images need to be included in a folder named “www” in the same directory where your app is located (check out the www folder in the PickState folder).

```
div(img(src='blueShiny.png', width = 500), style = "text-align: center;"),
```

This default behavior can be changed / overridden, but I’ve never had a need to do so. There are also additional ways to include images, but this has always worked fine for me.

Activity 1b

1. Make a Shiny page with a State Flag image, some detail about why you chose that flag (where you live / work / were born / favorite place to visit, etc.), and then a hyperlink - in italics - that takes the Shiny user to something related to that page (state tourism page, favorite restaurant, etc.).
   a. You must use at least 2 rows and/or the sidebar layout.
   b. You do not include any inputs; the server should be empty.

2. Update that page by changing the color of the background and the font used on the page.

Activity 1b Solutions

Shiny Inputs and Outputs

Shiny Inputs and Outputs

Input Functions:

*Input()

numericInput()

textInput()

sliderInput()

selectInput()

dateInput()

...

Output Functions:

*Output()

plotOutput()

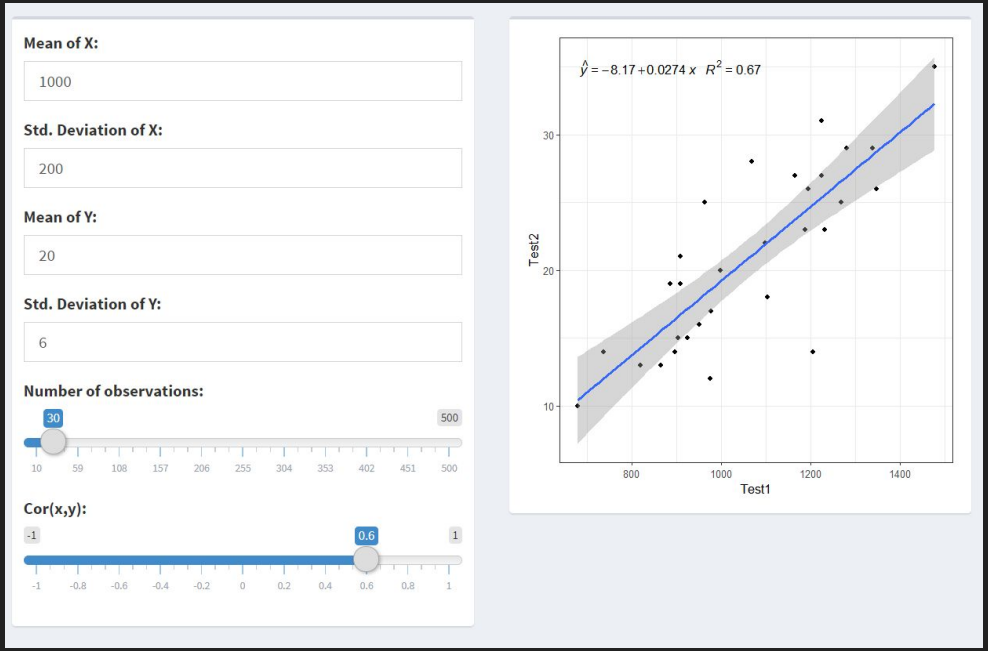
dataTableOutput()

textOutput()

htmlOutput()

uiOutput()

...



Input Function Structure

Common and Required Inputs:

sliderInput(inputId,

label,

...

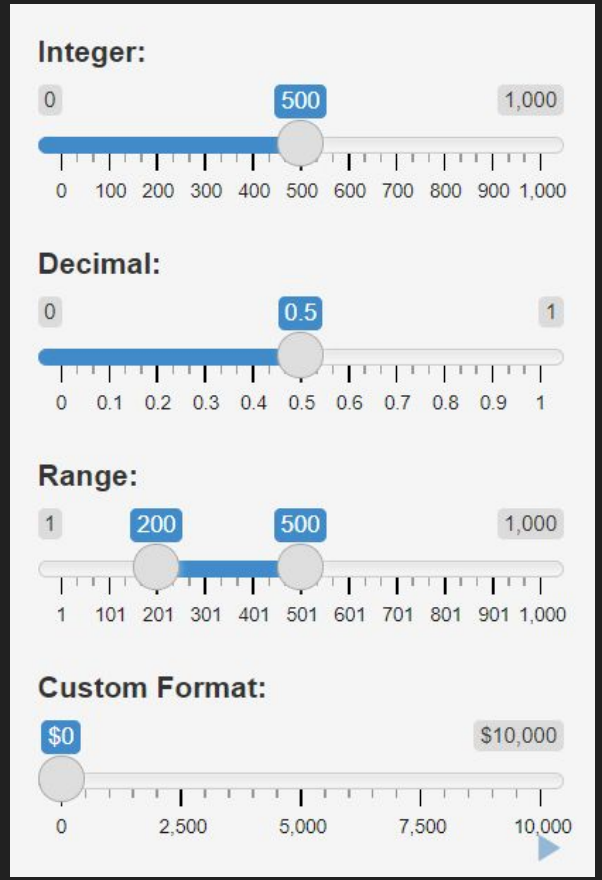
)

Full input list (including ones with defaults)

https://shiny.rstudio.com/reference/shiny/latest/sliderInput.html

Sometimes there are differences based on the Shiny package version

```
sliderInput(
  inputId,
  label,
  min,
  max,
  value,
  step = NULL,
  round = FALSE,
  format = NULL,
  locale = NULL,
  ticks = TRUE,
  animate = FALSE,
  width = NULL,
  sep = ",",
  pre = NULL,
  post = NULL,
  timeFormat = NULL,
  timezone = NULL,
  dragRange = TRUE
)
```



What the Input Functions Do

- Generate the HTML so you don't have to
- Signal to Shiny that something has changed
- Many functions used in the Shiny UI generate HTML like this

```
> sliderInput(inputId = "N", "Number of observations:", min = 10, max = 500,
 value = 30)
<div class="form-group shiny-input-container">
  <label class="control-label" for="N">Number of observations:</label>
  <input class="js-range-slider" id="N" data-min="10" data-max="500" data-fr
om="30" data-step="1" data-grid="true" data-grid-num="10" data-grid-snap="fa
lse" data-prettify-separator="," data-prettify-enabled="true" data-keyboard
="true" data-data-type="number"/>
</div>
>
```

Rules for accessing your inputs

Access like a list (or similar to a dataframe):

input$

Input[[ ]]

MUST be accessed in a reactive component inside the server code

ui

```
checkboxInput(inputId = "check1",
              label = “Choice A”)
```

server

```
input$check1
```

Activity: Add inputs to your UI (2a_AddInputs.R)

- Add a sliderInput() using some information from the dataset
  - inputId = “sliderValue”
- Add a selectInput() using some information from the dataset
  - inputId = “selectValue”
  - label
  - choices = loadedData$Month
- Assess the values returned in your outputs to see the differences
- Make your slider input provide a range rather than a single value
  - Hint: value = c(1, 10)
- Explore how to modify selectInput()
  - Examples: https://shiny.rstudio.com/gallery/selectize-vs-select.html
- Get creative!

Available Inputs: Shiny Widgets Gallery

Explore:

- Input options
- The object type of the returned value
- Code to add it to your app

Go to:
https://shiny.rstudio.com/gallery/widget-gallery.html

{shiny} Input Types:

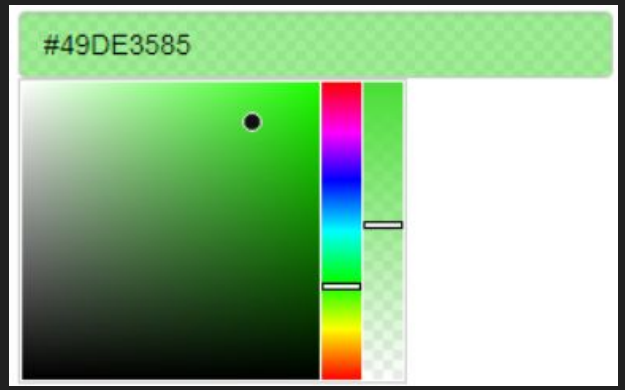
- Free text:
  - textInput()
  - passwordInput()
  - textAreaInput()
- Numeric:
  - numericInput()
  - sliderInput()
- Dates:
  - dateInput()
  - dateRangeInput()
- Set choice:
  - selectInput()
  - radioButtons()
- Upload files:
  - fileInput()
- Action buttons:
  - actionButton()

Spice up your inputs

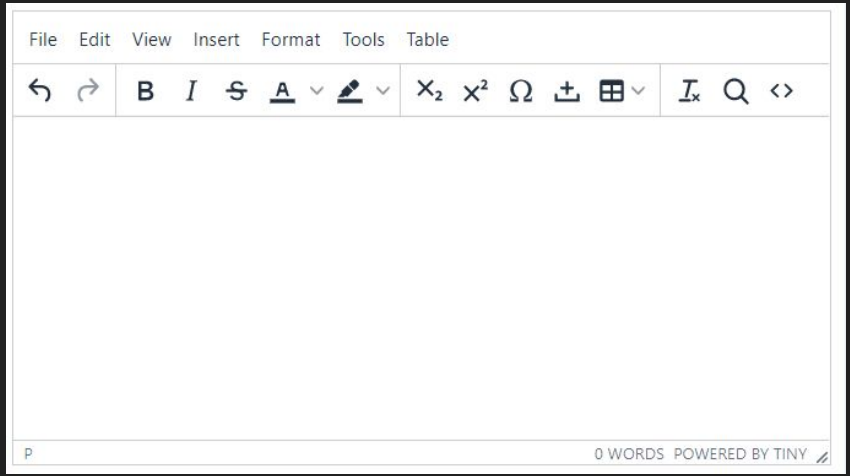
All Shiny Gallery Examples:
https://shiny.rstudio.com/gallery/

Interactive Button Styler - Includes the colourInput from {colourpicker}
https://shiny.rstudio.com/gallery/button-styler.html

selectInput() and it's many, many options:
https://shiny.rstudio.com/gallery/selectize-examples.html
https://shiny.rstudio.com/gallery/selectize-vs-select.html
https://shiny.rstudio.com/gallery/option-groups-for-selectize-input.html
https://shiny.rstudio.com/articles/selectize.html



https://cran.r-project.org/web/packages/colourpicker/vignettes/colourpicker.html



How Are Action Buttons Different?

With buttons, it's better to write code that will happen when the button is click than to try to use the VALUE of the button input.

observeEvent(input$MyButton, {

....

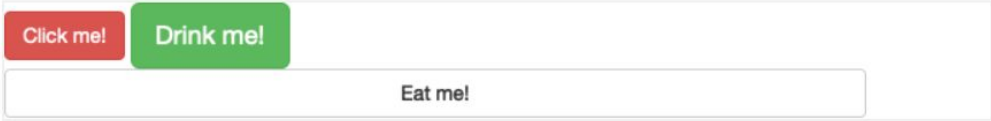
})

Rather than:

buttonCount <- input$MyButton

```
ui <- fluidPage(
  fluidRow(
    actionButton("click", "Click me!", class = "btn-danger"),
    actionButton("drink", "Drink me!", class = "btn-lg btn-success")
  ),
  fluidRow(
    actionButton("eat", "Eat me!", class = "btn-block")
  )
)
```

Copy



https://mastering-shiny.org/basic-ui.html#action-buttons

Outputs

Act like functions that re-run when an input changes (and they are viewable on the page -- lazy evaluation).

Two components:

1. UI Side: A location in your app
2. Server Side: Render function telling Shiny what to show

output$* <- render*({ })

ui (WHERE to display)

plotOutput(outputId = "regPlot")

server (WHAT to display)

output$regPlot <- renderPlot({ })

Outputs

Act like functions that re-run when an input changes (and they are viewable on the page -- lazy evaluation).

Two components:

1. UI Side: A location in your app
2. Server Side: Render function telling Shiny what to show

```
output$* <- render*({ })
```

ui

```
plotOutput(outputId = "regPlot")
```

server

```
output$regPlot <- renderPlot({ })
```

Outputs

Act like functions that re-run when an input changes (and they are viewable on the page -- lazy evaluation).

Two components:

1. UI Side: A location in your app
2. Server Side: Render function telling Shiny what to show

```
output$* <- render*({ })
```

ui

```
plotOutput(outputId = "regPlot")
```

server

```
output$regPlot <- renderPlot({ })
```

Outputs

Act like functions that re-run when an input changes (and they are viewable on the page -- lazy evaluation).

Two components:

1. UI Side: A location in your app
2. Server Side: Render function telling Shiny what to show

```
output$* <- render*({ })
```

ui

```
plotOutput(outputId = "regPlot")
```

server

```
output$regPlot <- renderPlot({ })
```

Outputs

Act like functions that re-run when an input changes (and they are viewable on the page -- lazy evaluation).

Two components:

1. UI Side: A location in your app
2. Server Side: Render function telling Shiny what to show

output$* <- render*({ })

ui

plotOutput(outputId = "regPlot")

server

output$regPlot <- renderPlot({ })

```
output$regPlot <- renderPlot({
  plot(reactiveData(), pch = 16)
})
```

Output Syntax Variations

```
output$displayValue1 <- renderPrint({
    input$sliderValue
})
output$displayValue1 <- renderPrint({
    return(input$sliderValue)
})
output$displayValue1 <- renderPrint(input$sliderValue)
```

Other Outputs & Their Render Functions

Outputs - render*() and *Output() functions work together to add R output to the UI

		works with	
	DT::**renderDataTable**(expr, options, callback, escape, env, quoted)		**dataTableOutput**(outputId, icon, …)
	renderImage(expr, env, quoted, deleteFile)		**imageOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
	renderPlot(expr, width, height, res, …, env, quoted, func)		**plotOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
	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, …)

Image from the R Shiny Cheatsheet https://shiny.rstudio.com/articles/cheatsheet.html

Plot and Table Output Syntax Activity (2b_AddOutputs.R)

Short Individual Activity:

- Add plotOutput and renderPlot
- Add dataTableOutput and renderDataTable
- Copy code from "#Code to place: ----" to make your data display

ui

```
plotOutput(outputId = "regPlot")
```

server

```
output$regPlot <- renderPlot({ })
```

Group Activity: Connection Outputs and Inputs (2c_ConnectInputsToOutputs.R)

- **Connect your outputs to your inputs**
- **Plot output**
  - **Ways to step it up: Plot click events or hover**
  - **Examples: https://shiny.rstudio.com/articles/plot-interaction.html**
- **Table output**
  - **Ways to step it up: Click a row in the table and get back it's information**
  - **Examples: https://rstudio.github.io/DT/shiny.html**
- **Table feedback**
  - **input$<TableOutputId>_rows_selected**
  - **https://yihui.shinyapps.io/DT-info/**

renderImage()

ui

```
imageOutput(outputId = "sample1")
```

server

```
output$sample1 <- renderImage({ })
```

renderPrint() and renderText()

```r
library(shiny)
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("print")
)
server <- function(input, output, session) {
  output$text <- renderText("renderText output")
  output$print <- renderPrint("renderPrint output")
}
shinyApp(ui, server)
```

renderText output

```
[1] "renderPrint output"
```

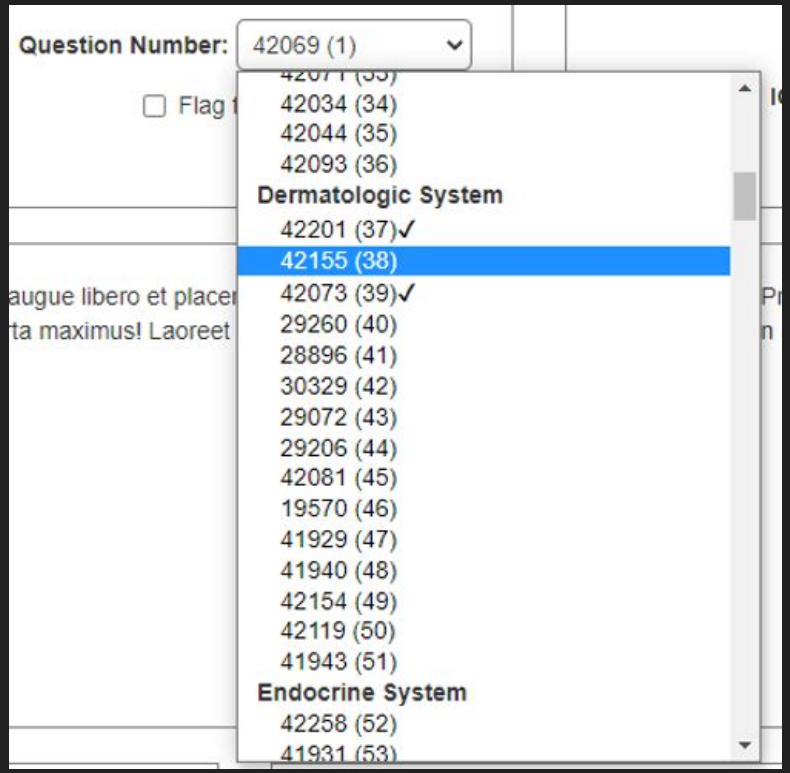
ui

```r
textOutput(outputId = "sample2")
```

server

```r
output$sample2 <- renderText({ })
```

renderUI() for HTML and tags




```
ui

uiOutput(outputId = "sample3")


server

output$sample3 <- renderUI({ })
```

There are 0 enemy items included!

The mean IRT b-value for all items used is: -0.16 (median: -0.287).

The mean b-values for all forms are 0.13 apart.

Other packages add more output options

{leaflet}

leafletOutput() renderLeaflet()

https://rstudio.github.io/leaflet/shiny.html

{plotly}

plotlyOutput() renderPlotly() ggplotly()

https://plotly-r.com/linking-views-with-shiny.html

{reactable}

reactableOutput() renderReactable()

https://glin.github.io/reactable/articles/examples.html#shiny

Exporting Tables and Plots

Export Options

1. Save to your drive (when running locally)
   a. write_csv()
   b. ggsave()
2. Save to external/cloud drives (good for deployed apps)
3. Download button (good for local and deployed apps)
   a. Saving function/code is provided and the user decides where to save the file you create

ui

```
downloadButton(outputId = “downloadRegPlot”,
               label = “Download Plot”)
```

server

```
output$downloadRegPlot <- downloadHandler(
  filename = "RegressionPlot.png",
  content = function(file) {
    <saving code>
  }
)
```

Activity: Exporting Tables and Plots (2d_AddDownloadButton.R)

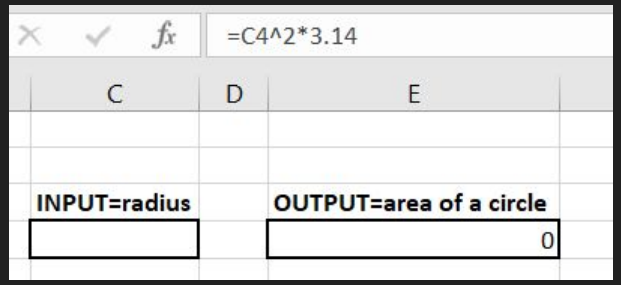
1. Add export buttons
2. Copy & Paste display code into the “content” area of the Handler and assign the plot/table to a variable
3. Save those variables using ggsave() or write_csv()

```
ui

downloadButton(outputId = “downloadRegPlot”,
                        label = “Download Plot”)

server

output$downloadRegPlot <- downloadHandler(

    filename = "RegressionPlot.png",

    content = function(file) {

        <saving code>

    }

)
```

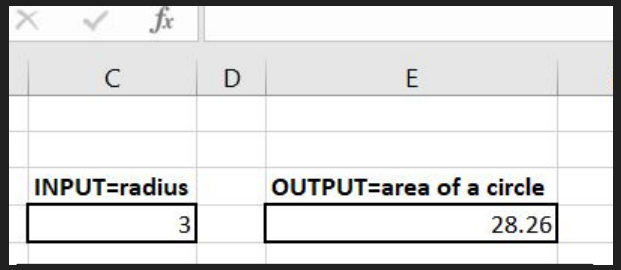
Reactivity in Shiny Apps

What is Reactivity?

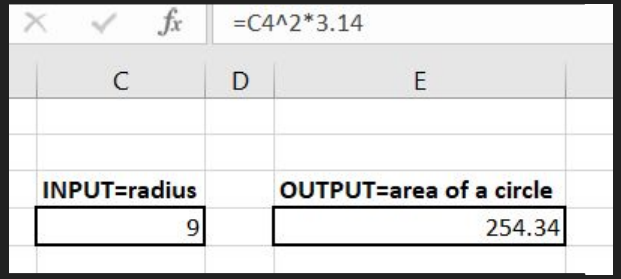
- One the great strengths of Shiny is that the applications we create are interactive.
- Adding reactivity to your application is what creates interactive features.
- Think about a spreadsheet:
  - If we input a value in one cell and use that value in formula in another cell, what happens?
  - What happens when we update the value in the first cell?



input$radius → output$area

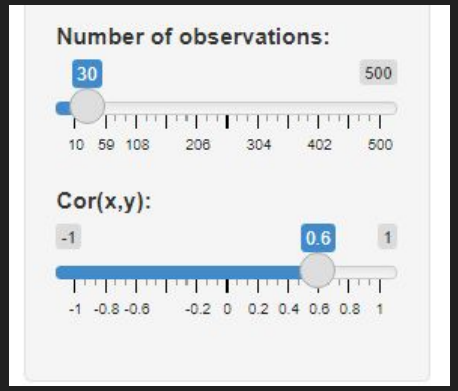


input$radius → output$area



input$radius → output$area

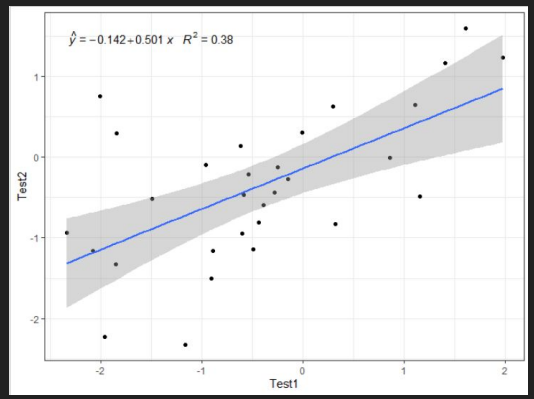
Reactivity controls the data flow through an App



input$N

input$R

output$plot



Reactive values

- We have talked pretty extensively about inputs in Shiny applications (thanks, Marcus!)
- The list of inputs in your app are reactive values by definition.
- When we slide a slider bar, the input value associated with bar stores the current value.
- The input value will change when an app user changes the input.

Reactive functions

- Reactive values are only useful within a reactive function.
- If you attempt to use a reactive value in a regular function, you'll get an error in your R console.

```
Error in
.getReactiveEnvironment()
$currentContext() :

Operation not allowed
without an  active reactive
context. (You tried  to do
something that can only be
done  from inside a reactive
expression or  observer.)
```

Reactivity happens in two steps

- When a reactive value changes, it becomes invaild and notifies the functions that use that value.
- The reactive function responds by updating the stored value to the new value and reruns the function.
- Let’s take another look at our simple app from the opening.

The reactive toolbox

There are six major types of reactive functions that are useful when building Shiny apps.

1. **Display output:** The render*() functions (you have seen some of these)
2. **Modularing:** The reactive() function
3. **Preventing reactions:** The isolate() function
4. **Triggering events:** The observeEvent() function
5. **Delaying reactions:** The eventReactive() function
6. **Managing states:** The reactiveValues() function

Displaying output with render*

- We already know some of these functions from generating output.
- The structure of all render*() functions is the same:

```
render*({ CODE BLOCK })
```

The code block contain one or more reactive values.

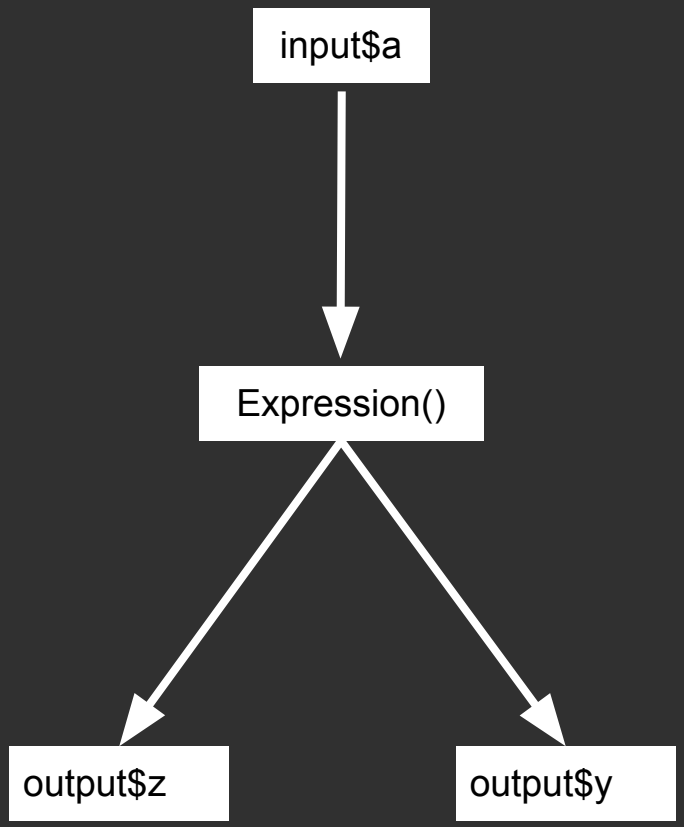
The code block will respond with each update to a reactive value within the code block.

You can include as much or as little code as needed in between the curly brackets.

Displaying output with render*

Function	Outcome
renderDataTable()	An interactive table
renderImage()	An image (saved as a link to a source file)
renderPlot()	A plot
renderPrint()	A code block of printed output
renderTable()	A simple table
renderText()	A character string
renderVerbatimtext()	A character string format like console output

Modularizing Shiny code with reactive()

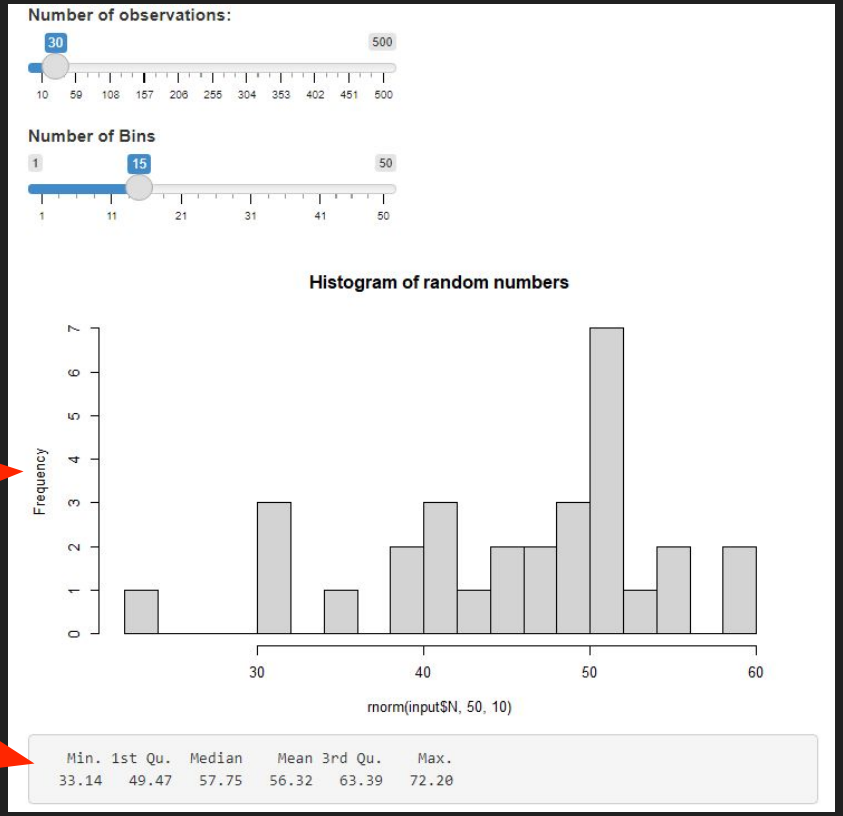


Making modular code

- We might need to perform some operations on your reactive values BEFORE you render the output.
  - You do not want to update an expression in two places.
  - you want to run a function using inputs and need the result to be reactive.
- The **reactive({})** function allows you to operate on reactive values to create a reactive expression that behaves a little like a data frame or tibble.
- We can then call on this reactive object within other reactive functions.

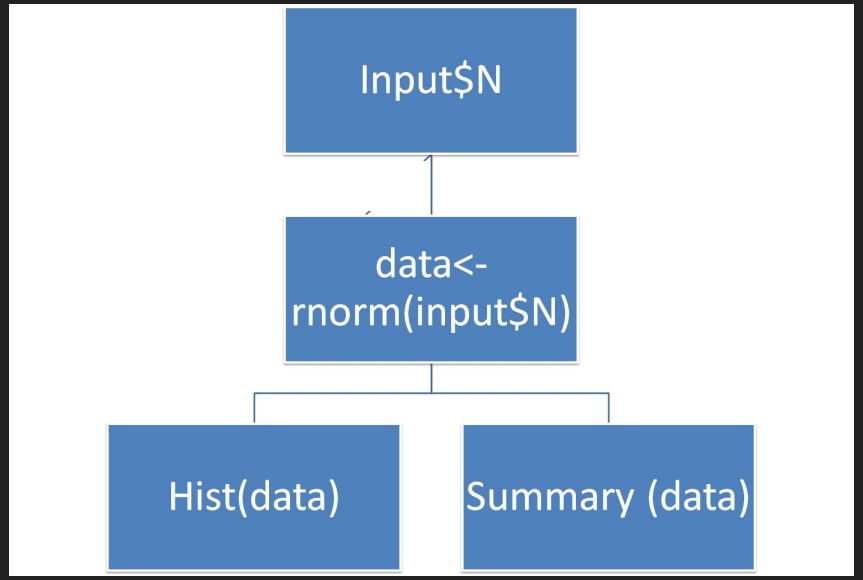
Using reactive()

```
ui <- fluidPage(
    mainPanel(
      sliderInput(inputId  = "N","Number of observ
                  min = 10, max = 500, value = 30)
      sliderInput(inputId  = "bins","Number of Bir
                  min = 1.00, max = 50.00, value =
      plotOutput(outputId = "Plot"),
      verbatimTextOutput(outputId = "SumStats")
    )
  )
server <- function(input, output){
  # Draw a histogram;
  output$Plot <- renderPlot({
    hist(rnorm(input$N,50,10),
         breaks=input$bins,
         main="Histogram of random numbers")
    })
  # Print summary statistics;
  output$SumStats <- renderPrint({
    summary(rnorm(input$N,50,10))
    })
}
```



Using reactive()

- The app is it is now is generating two sets of random numbers.
- What we would like to do is generate one set of numbers and use this set in both render*() functions
- We need to build an in between step that contains our reactive values



Building reactive expressions in your code

```
data < - reactive( { rnorm(input$N, 50, 10})
```

The code block here contains reactive values and will respond when a reactive value is invalidated.

the code between the curly brackets will be used to regenerate the expression with each update.

Using reactive()

```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- reactive({rnorm(input$N,50,10)})
  # Draw a histogram;
    output$Plot <- renderPlot({
      hist(Data(),
           breaks=input$bins,
           main="Histrogram of random numbers")
    })
    # Print summary statistics;
    output$SumStats <- renderPrint({
      summary(Data())
    })
}
```

Side-by-side code

Before

```r
server <- function(input, output){
  # Draw a histogram;
  output$Plot <- renderPlot({
    hist(rnorm(input$N,50,10),
         breaks=input$bins,
         main="Histogram of random numbers")
    })
  # Print summary statistics;
  output$SumStats <- renderPrint({
    summary(rnorm(input$N,50,10))
    })
}
```

After

```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- reactive({rnorm(input$N,50,10)})
  # Draw a histogram;
    output$Plot <- renderPlot({
      hist(Data(),
           breaks=input$bins,
           main="Histrogram of random numbers")
    })
    # Print summary statistics;
    output$SumStats <- renderPrint({
      summary(Data())
    })
}
```

A few notes about reactive()

- We need to call a reactive expression as if it was function with no arguments (e.g., in our example we used data() instead of data).

- Reactive expression retain their values until a reactive value used in the expression is invalidated.

Activity 3a

Now, we will add some reactivity to a Shiny function. Open the file “3c_Task_1.r”

a. Use reactive() to create a reactive expression named “xy” that uses the code that generates data contained within renderPlot()

b. Replace the generating code in renderPlot with your new reactive expression

c. Create a new output named “xyvalues” using renderDataTable that displays the values of your newly created expression, “xy”, in a table just below the plot.

d. bonus: use options within renderDataTable to limit the display to 10 observations per page

Activity solution

Let’s look at the solution and walk through the code

Open the file “3c_task_1_solution.r”

Preventing reactions

- Sometimes immediate reactivity can be problematic or inconvenient.
- We can use the isolate() function to circumvent the reactivity of certain values.
- if wrapped in isolate(), your app will treat a reactive value like its a normal R value.

Using isolate() to suppress a reactive value

```
isolate( { rnorm(input$N, 50, 10})
```

The object will not respond to changes in any reactive value in the code.

the code between the curly brackets will be used to create.

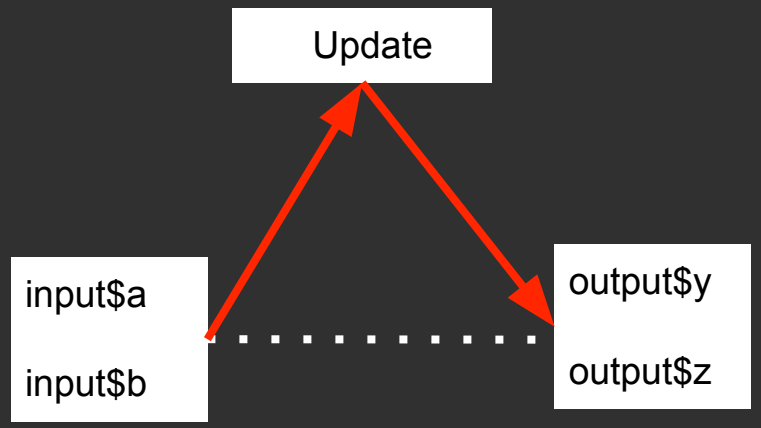
Using isolate() to suppress a reactive value

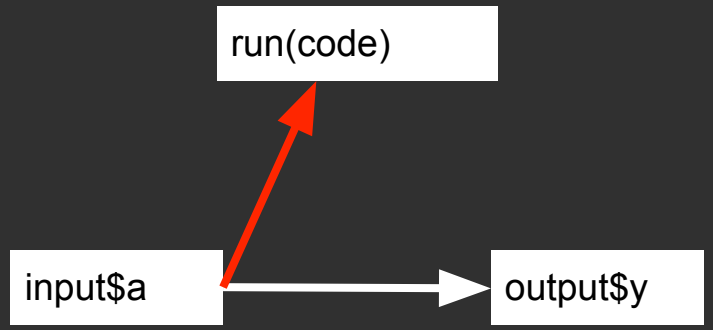
```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- reactive({rnorm(input$N,50,10)})
  # Draw a histogram;
    output$Plot <- renderPlot({
      hist(Data(),
           breaks=input$bins,
           main="Random Values")
    })
    # Print summary statistics;
    output$SumStats <- renderPrint({
      summary(Data())
    })
}
```

```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- reactive({rnorm(input$N,50,10)})
  # Draw a histogram;
    output$Plot <- renderPlot({
      hist(Data(),
           breaks=isolate({input$bins}),
           main="Random Values")
    })
    # Print summary statistics;
    output$SumStats <- renderPrint({
      summary(Data())
    })
}
```

Triggering or delaying a reaction

Using observeEvent() and eventReactive()





Triggering or delaying a reaction

- There are two reactive functions that are highly in shiny apps.
  - observeEvent() allow us to trigger a code chunk to run when a certain action takes place.
  - eventReactive() creates a reactive expression after a certain action takes place.
- There are a number of ways to capture an action, but the most frequently used is with an action button.

Adding an action button

```
actionButton(inputId = "go", label = "Do it")
```

input function for an action button

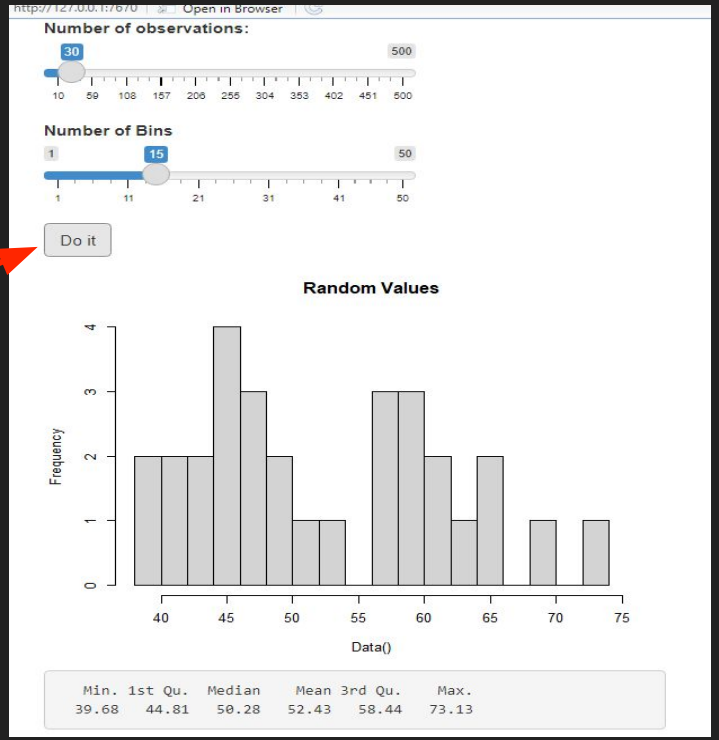
name of input value

Display text

- We can add a clickable button as an input on the UI side of our shiny app.
- When we click on the button the input value takes on an integer values.
- Each click increases the value by one.
- The value of the action button is not useful, but we can use it within other reactive functions to trigger actions.

Adding action buttons

```r
ui <- fluidPage(

    mainPanel(
      sliderInput(inputId  = "N",      "Number of observations:",
                  | min = 10, max = 500, value = 30),
      sliderInput(inputId  = "bins",      "Number of Bins",
                   min = 1.00, max = 50.00, value = 15,step=10),


      actionButton(inputId="go", label="Do it"),


      plotOutput(outputId = "Plot"),
      verbatimTextOutput(outputId = "SumStats")
    )
  )
```



Triggering an action with observeEvent()

- We can use observeEvent() to watch for updates to a reactive value (i.e., a click occurs) to trigger a piece of code to run.
- We tie the observer to a reactive value (i.e, a action button input value).
- When the reactive value changes, the observer invalidates and reruns.

```
observeEvent(input$go, {print(input$clicks) })
```

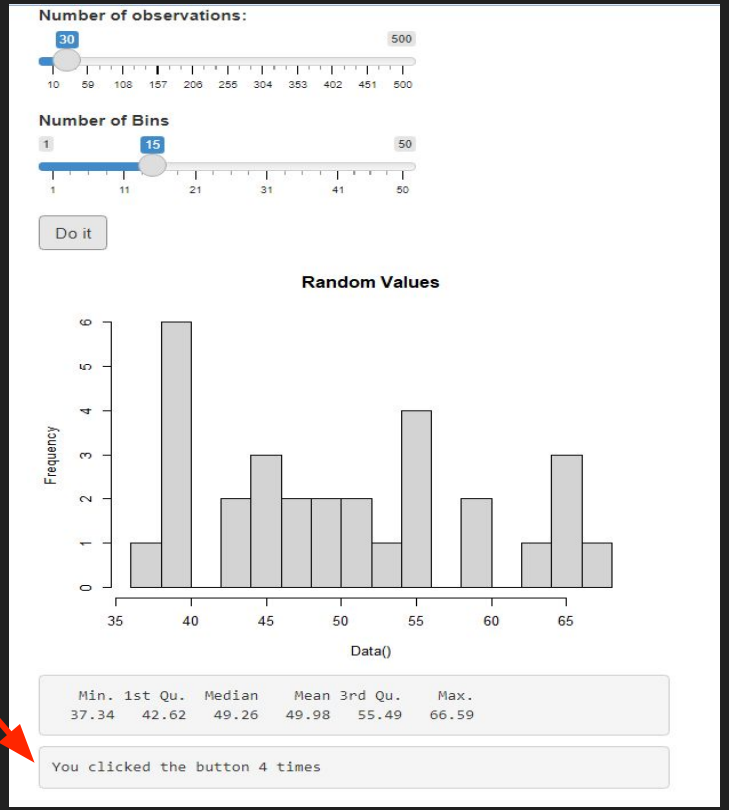
This is the reactive values that triggers the observer.

This code runs when the observer is invalidated (i.e., the reactive value updates).

The code in the brackets is treated like we used isolate()

Triggering an action with observeEvent()

```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- reactive({rnorm(input$N,50,10)})
  # Draw a histogram;
    output$Plot <- renderPlot({
      hist(Data(),
           breaks=isolate({input$bins}),
           main="Random Values")
    })
    # Print summary statistics;
    output$SumStats <- renderPrint({
      summary(Data())
    })

    observeEvent(input$go,{
      output$ClickMessage<-renderPrint(
        {paste0("You clicked the button ", input$go, " times")
          })
    })

}
```



Delaying an action with eventReactive()

We tie a reactive expression to delay creation until a reactive value (like the input from clicking on an update button) is satisfied.

```
data<-eventReactive(input$go, {rnorm(input$N)})
```

This is the reactive values that triggers the observer.

This code runs when one the action takes place and created the reactive object.

The expression in the brackets is treated like we used isolate().

Updating an expression with eventReactive

```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- reactive({rnorm(input$N,50,10)})
  # Draw a histogram;

  output$Plot <- renderPlot({
      hist(Data(),
           breaks=isolate({input$bins}),
           main="Random Values")
    })
    # Print summary statistics;

  output$SumStats <- renderPrint({
      summary(Data())
    })

    observeEvent(input$go,{
      output$ClickMessage<-renderPrint(
        {paste0("You clicked the button ",
                input$go, " times")
          })
    })

}
```

```r
server <- function(input, output) {
  # Create a reactive expression;
  Data <- eventReactive(input$go,{rnorm(input$N,50,10)})
  # Draw a histogram;

  output$Plot <- renderPlot({
      hist(Data(),
           breaks=isolate({input$bins}),
           main="Random Values")
    })
    # Print summary statistics;

    output$SumStats <- renderPrint({
      summary(Data())
    })

    observeEvent(input$go,{
      output$ClickMessage<-renderText(
        {paste0("You have updated this plot ",
                input$go, " times")
          })
    })

}
```

Activity 3h

Let's build on the app we started in the last activity to add an update button that will delay the generation of our plot and table. Open the file "3h_task_2.r"

a. Add an update button below the other inputs in your app.

b. Delay the creation of your reactive expression until the update button is clicked using eventReactive.

c. Use observeEvent and renderPrint to create a printed message below your plot that says "You have updated your plot XX times" when you click the update button.

Activity solution

Let’s look at the solution and walk through the code

Open the file “3h_task_2_solution.r”

Creating your own reactive values with an input

reactiveValues()

Using reactiveValues()

- Shiny applications often rely on user specified values gathered through our input widgets
- Inputs cannot be overwritten programmatically, but this might be handy to do sometimes.
- We can use reactiveValues() to create list of objects, not specified by the user, than can be used within a reactive context and manipulated programmatically.

One last useful input: inFile

- We can us inFile on the ui side to navigate to and import external data files
- inFile creates a set of reactive values with these elements
  - name (the file name)
  - size (uploaded file size)
  - type
  - datapath(the path to a temp file created on upload)
- The values are used on the server side in reactive functions.

An short example of reactiveValues and InFile

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      fileInput("file1", "Choose CSV File")
    ),
      mainPanel(
        textOutput("report"),
        dataTableOutput("contents")
      )
    )
)

  server <- function(input, output) {
    message<-reactiveValues(display="No data file has been uploaded")

    data<-eventReactive(input$file1,{read.csv(input$file1$datapath)})
    output$contents <- renderDataTable({ data() })

    observeEvent(input$file1,{ message$display=input$file1$name })
    output$report<-renderText({paste0("You are viewing the file:
                                      ",message$display )})
  }
```