

## Notes 4 - R Shiny

Rick Brown  
Southern Utah University

Math 3190

# What is Shiny?

Shiny is an **R** package that allows you to easily create rich, interactive web apps around your **R** functions and packages. Shiny allows you to take your work in **R** and disseminate it via a web browser so that anyone can use it. Shiny makes you look awesome by making it easy to produce polished web apps with a minimum amount of pain.

(Credit: Some of the images and slide text were taken and adapted from <https://mastering-shiny.org> and RStudio's Shiny tutorials)

# What is Shiny?

Shiny is designed primarily with data scientists in mind, and to that end, you can create pretty complicated Shiny apps with no knowledge of HTML, CSS, or JavaScript. On the other hand, Shiny doesn't limit you to creating trivial or prefabricated apps: its user interface components can be easily customized or extended, and its server uses reactive programming to let you create any type of back end logic you want.

# What is Shiny?

Some of the things people use Shiny for are:

- Create dashboards
- Replace hundreds of pages of PDFs with interactive apps
- Communicate complex models informative visualizations
- Provide self-service data analysis for common workflows
- Create interactive demos for teaching statistics and data science

In short, Shiny gives you the ability to pass on some of your **R** superpowers to anyone who can use the web.

# Getting Started

To create your first Shiny App you will need:

- **R** installed (of course).
- **RStudio** installed. (strongly recommended):  
<https://www.rstudio.com/products/rstudio/download>
- The **R** Shiny package: `install.packages("shiny")`
- Knowledge and experience in developing **R** packages (recommended).
- Git and GitHub (recommended): to share your Apps.

# Getting Started

If you haven't already installed Shiny, install it now with:

```
install.packages("shiny")
```

If you've already installed Shiny, use the following to check that you have version 1.5.0 or greater:

```
packageVersion("shiny")
```

```
## [1] '1.8.0'
```

**RStudio's** [Shiny Showcase](#) is an exciting assembly of exciting apps!

Or we can start with a very simple example

```
library(shiny)
runExample("01_hello")
```

# Introduction to Shiny Development

Here we'll create a simple `Shiny` app, starting with the minimum boilerplate, followed by the two key components of every `Shiny` app: the UI (short for user interface) which defines how your app looks, and the server function which defines how your app works. `Shiny` uses **reactive** programming to automatically update outputs when inputs change.

## Template: Shortest Viable App

The simplest way to start an app is to create a new **R** script and start with the following code:

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

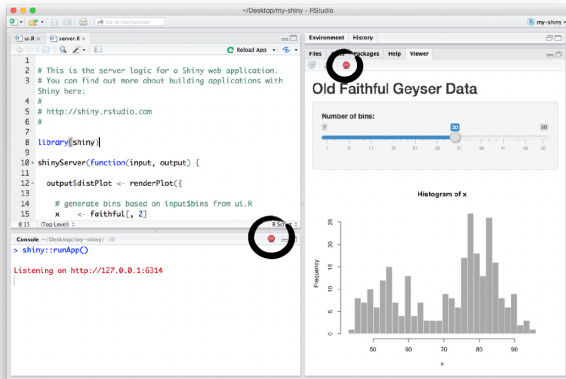
shinyApp(ui = ui, server = server)
```



# Close your App

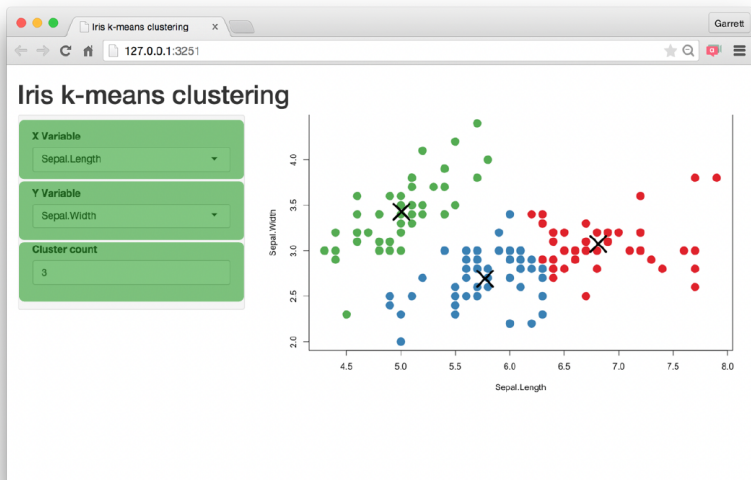
To close the Shiny app, press the stop button, close the window, or press escape from the console.

## Close an app



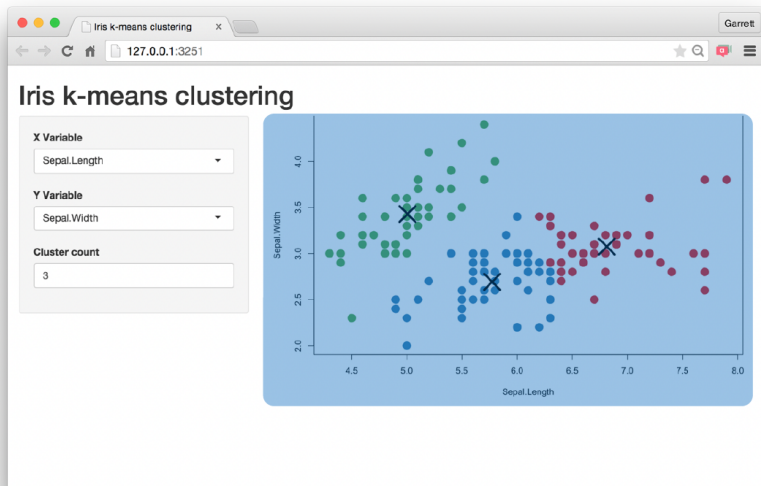
# Inputs and Outputs

Build your app around **inputs** and **outputs**



# Inputs and outputs

Build your app around **inputs** and **outputs**



# Adding Elements to the UI

Add elements to your app as arguments to `fluidPage()`

```
library(shiny)
```

```
ui <- fluidPage(  
  # Input() functions,  
  # Output() functions  
)
```

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

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

Take note that the `ui` (or `fluidPage` function) is what controls what the app looks like and takes inputs from the user. The `server` function controls what the app is doing.

# Adding Elements to the UI

Add elements to your app as arguments to `fluidPage()`

```
library(shiny)
```

```
ui <- fluidPage(  
  "Hello World!"  
)
```

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

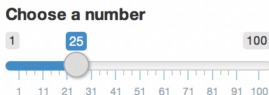
```
shinyApp(ui = ui, server = server)
```

# Input Functions

Create an input with a function that ends in **Input()**.

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

## Syntax



```
sliderInput(inputId = "num", label = "Choose a number", ...)
```

input name  
(for internal use)

Notice:  
Id not ID

label to  
display

input specific  
arguments

?sliderInput

# Input Functions in an App

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100)
)

server <- function(input, output) {}

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

The ID of in the Input() function, inputId in this case, creates an attribute of input that can be used in the server function inside of a render() function.

## Server inputs

You can access inputs from the UI using **input\$**

```
sliderInput(inputId = "num",...)
```

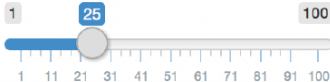


```
input$num
```



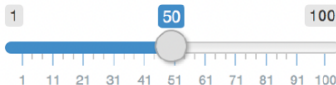
The input value changes whenever a user changes the input.

Choose a number



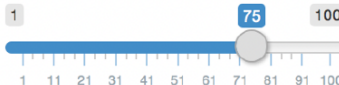
`input$num = 25`

Choose a number



`input$num = 50`

Choose a number



`input$num = 75`

# Render Functions in an App

A `render...()` function is used in the server function to create an attribute of output.

You can save your outputs to return to the UI using **output\$** and build objects to display with a function whose name begins with `render`.

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

## Render Functions in an App

```
ui <- fluidPage(  
  sliderInput(inputId = "num",  
    label = "Choose a number",  
    value = 25, min = 1, max = 100)  
)  
  
server <- function(input, output) {  
  output$hist <- renderPlot({  
    hist(rnorm(input$num))  
  })  
}  
  
shinyApp(server = server, ui = ui)
```

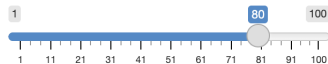
## Output Functions in an App

Now that an attribute of output has been created, we need an `Output()` function inside `fluidPage()` in the ui. The name of the output attribute created in the server function must go in quotes.

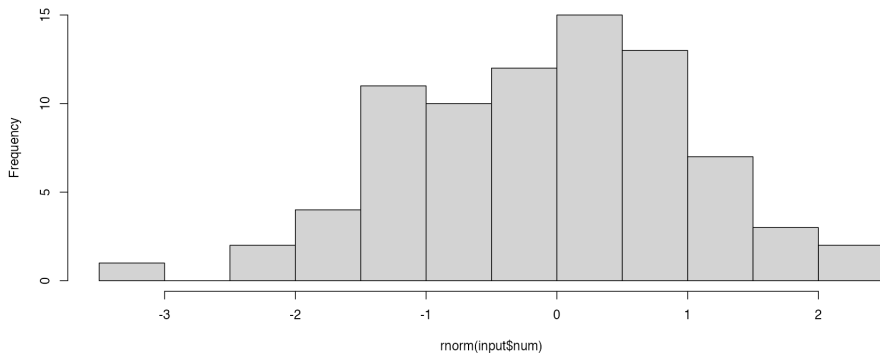
```
ui <- fluidPage(  
  sliderInput(inputId = "num",  
    label = "Choose a number",  
    value = 25, min = 1, max = 100), # Comma!  
  plotOutput("hist")  
)  
  
server <- function(input, output) {  
  output$hist <- renderPlot({  
    hist(rnorm(input$num))  
  })  
}
```

# Shiny App

Choose a number



Histogram of `rnorm(input$num)`



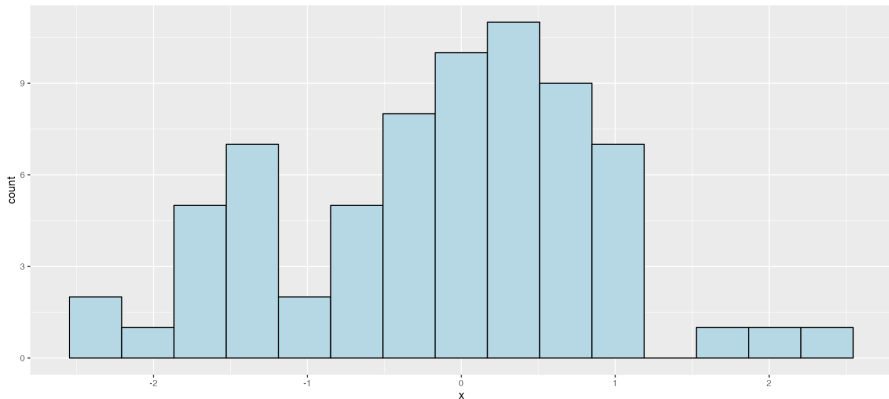
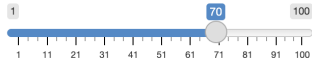
# Output Functions in an App

We can also make the graph look nicer with ggplot.

```
library(tidyverse)
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100), # Comma!
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    DF <- data.frame(x = rnorm(input$num))
    ggplot(DF, aes(x = x)) +
      geom_histogram(color = "black", fill = "lightblue",
        bins = 15)
  })
}
shinyApp(server = server, ui = ui)
```

# Shiny App

Choose a number



# Shiny Buttons (Inputs)

## Buttons

Action

Submit

`actionButton()`  
`submitButton()`

## Single checkbox

☒ Choice A

`checkboxInput()`

## Checkbox group

☒ Choice 1  
☐ Choice 2  
☐ Choice 3

`checkboxGroupInput()`

## Date input

2014-01-01

`dateInput()`

## Date range

2014-01-24 to 2014-01-24

`dateRangeInput()`

## File input

Choose File No file chosen

`fileInput()`

## Numeric input

1

`numericInput()`

## Password Input

\*\*\*\*\*

`passwordInput()`

## Radio buttons

☒ Choice 1  
☐ Choice 2  
☐ Choice 3

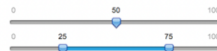
`radioButtons()`

## Select box

Choice 1

`selectInput()`

## Sliders



`sliderInput()`

## Text input

Enter text...

`textInput()`



# Shiny Renders

Use the **render\***() function that creates the type of output you wish to make.

function	creates
<code>renderDataTable()</code>	An interactive table <small>(from a data frame, matrix, or other table-like structure)</small>
<code>renderImage()</code>	An image (saved as a link to a source file)
<code>renderPlot()</code>	A plot
<code>renderPrint()</code>	A code block of printed output
<code>renderTable()</code>	A table <small>(from a data frame, matrix, or other table-like structure)</small>
<code>renderText()</code>	A character string
<code>renderUI()</code>	a Shiny UI element

# Shiny Outputs

Function	Inserts
<code>dataTableOutput()</code>	an interactive table
<code>htmlOutput()</code>	raw HTML
<code>imageOutput()</code>	image
<code>plotOutput()</code>	plot
<code>tableOutput()</code>	table
<code>textOutput()</code>	text
<code>uiOutput()</code>	a Shiny UI element
<code>verbatimTextOutput()</code>	text

# Interactive Plots with `plotly`

A package that pairs very nicely with Shiny is the `plotly` package. This package allows us to create interactive plots with hover features. Some especially nice functions in `plotly` are:

- `ggplotly()`: This takes a `ggplot` object and converts it to a `plotly` object to make it interactive.
- `renderPlotly()`: This can be put in the server function to render an interactive plot that can be saved to an output.
- `plotlyOutput()`: This can be put in the `fluidPage()` function to generate an interactive plot in the Shiny app.

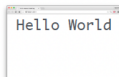
# Interactive Plots with plotly

Taking the plot from the last output and making it interactive:

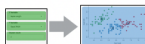
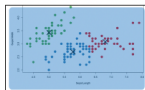
```
library(shiny); library(tidyverse); library(plotly)
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100), # Comma!
  plotlyOutput("hist") # Note: changed from plotOutput to plotlyOutput
)
server <- function(input, output) {
  output$hist <- renderPlotly({ # Note: changed to renderPlotly
    DF <- data.frame(x = rnorm(input$num))
    p <- ggplot(DF, aes(x = x)) + # Save object as p
      geom_histogram(color = "black", fill = "lightblue",
        bins = 15)
    ggplotly(p) # Make the ggplot interactive
  })
}
shinyApp(server = server, ui = ui)
```

## Recap

```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```



Hello World



Begin each app with the template

Add elements as arguments to **fluidPage()**

Create reactive inputs with an **\*Input()** function

Display reactive results with an **\*Output()** function

Assemble outputs from inputs in the server function

# Recap: Server



Use the server function to assemble inputs into outputs. Follow 3 rules:

`output$hist <-`

1. Save the output that you build to `output$`

```
renderPlot({  
  hist(rnorm(input$num))  
})
```

2. Build the output with a `render*()` function

`input$num`

3. Access input values with `input$`



Create reactivity by using `Inputs` to build `rendered Outputs`

# Building the Server

There are 2 basic rules for your **fluidPage()** function:

- ① Save input values with one of the Shiny buttons. Most of these functions end with `Input()` or `Button()`.
- ② Use a function that ends with `output()` in `fluidPage()` to display objects built with a `render...()` function in the server.

There are 3 basic rules for your **server** function:

- ① Access input values with `input$`.
- ② Save objects to display as `output$`.
- ③ Build objects to display with `render...()`.

# Reactivity

Shiny is built on reactive objects. The term **reactivity** refers to the idea that the inputs and outputs are connected to each other: when you change an input, you change all outputs that require the input.

If we ever want to do anything with the input out of a `render...()` function, we need to use the `reactive()` function around it. Then, to use any variable defined with a `reactive()` function, we need to call it with parentheses. This is most common if you want to use that object in multiple `render...()` functions.



# Reactivity

```
ui <- fluidPage(  
  sliderInput(inputId = "num", label = "Choose a number",  
    value = 25, min = 1, max = 100), # Comma!  
  plotOutput("hist"),  
  plotOutput("box")  
)  
server <- function(input, output) {  
  rand_DF <- reactive(data.frame(x = rnorm(input$num)))  
  # rand_values <- input$num /> rnorm() /> reactive()  
  output$hist <- renderPlot({  
    ggplot(rand_DF(), aes(x = x)) +  
      geom_histogram(color = "black", fill = "lightblue", bins = 15) +  
      geom_vline(xintercept = median(rand_DF()$x), linewidth = 1) +  
      coord_cartesian(xlim = c(-3, 3))  
  })  
  output$box <- renderPlot({  
    ggplot(rand_DF(), aes(x = x)) +  
      geom_boxplot(color = "black", fill = "lightblue") +  
      coord_cartesian(xlim = c(-3, 3))  
  })  
}  
shinyApp(server = server, ui = ui)
```

# Without Reactivity

```
ui <- fluidPage(  
  sliderInput(inputId = "num", label = "Choose a number",  
    value = 25, min = 1, max = 100), # Comma!  
  plotOutput("hist"),  
  plotOutput("box")  
)  
server <- function(input, output) {  
  output$hist <- renderPlot({  
    rand_DF <- data.frame(x = rnorm(input$num))  
    ggplot(rand_DF, aes(x = x)) +  
      geom_histogram(color = "black", fill = "lightblue", bins = 15) +  
      geom_vline(xintercept = median(rand_DF$x), linewidth = 1) +  
      coord_cartesian(xlim = c(-3, 3))  
  })  
  output$box <- renderPlot({  
    rand_DF <- data.frame(x = rnorm(input$num))  
    ggplot(rand_DF, aes(x = x)) +  
      geom_boxplot(color = "black", fill = "lightblue") +  
      coord_cartesian(xlim = c(-3, 3))  
  })  
} # In this case, different values are plotted in each plot!  
shinyApp(server = server, ui = ui)
```

## Multi-file apps:

When these Shiny apps get larger, it is common practice to create two files: a **ui.R** file and a **server.R** file.

```
#ui.R
library(shiny)
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)
```

```
#server.R
library(shiny)
server <- function(input, output){
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
```

## Multi-file apps:

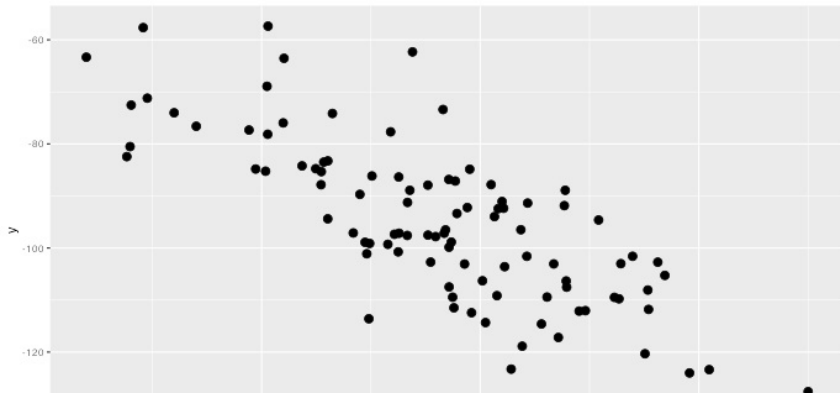
And then finally the program file, **app.R**:

```
#app.R  
library(shiny)  
  
source("ui.R")  
source("server.R")  
  
shinyApp(server = server, ui = ui)
```

# Shiny Example

```
##  
## Listening on http://127.0.0.1:5310  
Guess the correlation!
```

Submit Guess



# Hosting Shiny Apps

You can put your Shiny app in a GitHub repository and then run it (or have someone else run it) by using the `runGitHub()` function in the `shiny` library. The syntax is either of the following:

```
runGitHub("repository_name", "username")  
runGitHub("username/repository_name")
```

By default, this will only work if your entire app is in a file called `app.R` with `shinyApp(ui = ui, server = server)` as the last line or if your app is split up into two files: `ui.R` and `server.R`.

If the Shiny app is in a different directory in your repository, you can specify what folder it is in by using the `subdir` option in the `runGitHub()` function:

```
runGitHub("repository_name", "username",  
          subdir = "path/to/shiny/app/in/repo")
```

# Building Apps in R Packages

We can also add a Shiny app to any **R** package we create. In the package directory, create an `inst/shiny/app_name` directory and place either the `app.R` file or the `ui.R` and `server.R` files in there. Once this is done, be sure to add “shiny” to the “Depends:” section of your `DESCRIPTION` file.

Then, in the package `R/` folder, add a function that calls the Shiny app. A template for this is on the next slide.

Also:

- Write the package functions first!
  - You can use any functions defined in your package in your shiny app.
- Shiny app should call functions, provide inputs, display results, etc.

You can check out the `mypackage` repo in my GitHub page for an example.

# Building Apps in R Packages:

```
## Shiny App Name
##
## App description
##
## @export

# Change the name of the function to be relevant.
# It is convention to begin the name of this function with "run"
runShinyApp <- function() {
  appDir <- system.file("shiny", "app_name", package = "package_name")
  if (appDir == "") {
    stop("Could not find example directory. Try re-installing `package_name`.",
        call. = FALSE)
  }

  shiny::runApp(appDir, display.mode = "normal")
}
```



It is also possible to run Shiny completely in a browser using WebR and shinylive. In fact, you can host Shiny apps on github. This is a beyond the scope of our class, but here are some links to some tutorials:

- 1 <https://github.com/posit-dev/r-shinylive>
- 2 <https://medium.com/@rami.krispin/deploy-shiny-app-on-github-pages-b4cbd433bdc>
- 3 <https://hypebright.nl/index.php/en/2023/10/02/run-a-shiny-app-in-the-browser-with-shinylive-for-r/>

An example of this is given at this link:

<https://rbrown53.github.io/ShinyCorrelation/>

# Session info

```
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31)
## Platform: aarch64-apple-darwin20 (64-bit)
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib; LAPACK version 3
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats      graphics  grDevices  utils      datasets
## [6] methods    base
##
## other attached packages:
## [1] shiny_1.8.0      dslabs_0.7.6     lubridate_1.9.3
## [4] forcats_1.0.0    stringr_1.5.1    dplyr_1.1.4
## [7] purrr_1.0.2      readr_2.1.5      tidyr_1.3.0
## [10] tibble_3.2.1     ggplot2_3.4.4    tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
## [1] sass_0.4.8        utf8_1.2.4
## [3] generics_0.1.3    stringi_1.8.3
## [5] hms_1.1.3         digest_0.6.34
## [7] magrittr_2.0.3    evaluate_0.23
## [9] grid_4.3.2        timechange_0.3.0
## [11] fastmap_1.1.1     isonlite_1.8.8
```