NC STATE UNIVERSITY

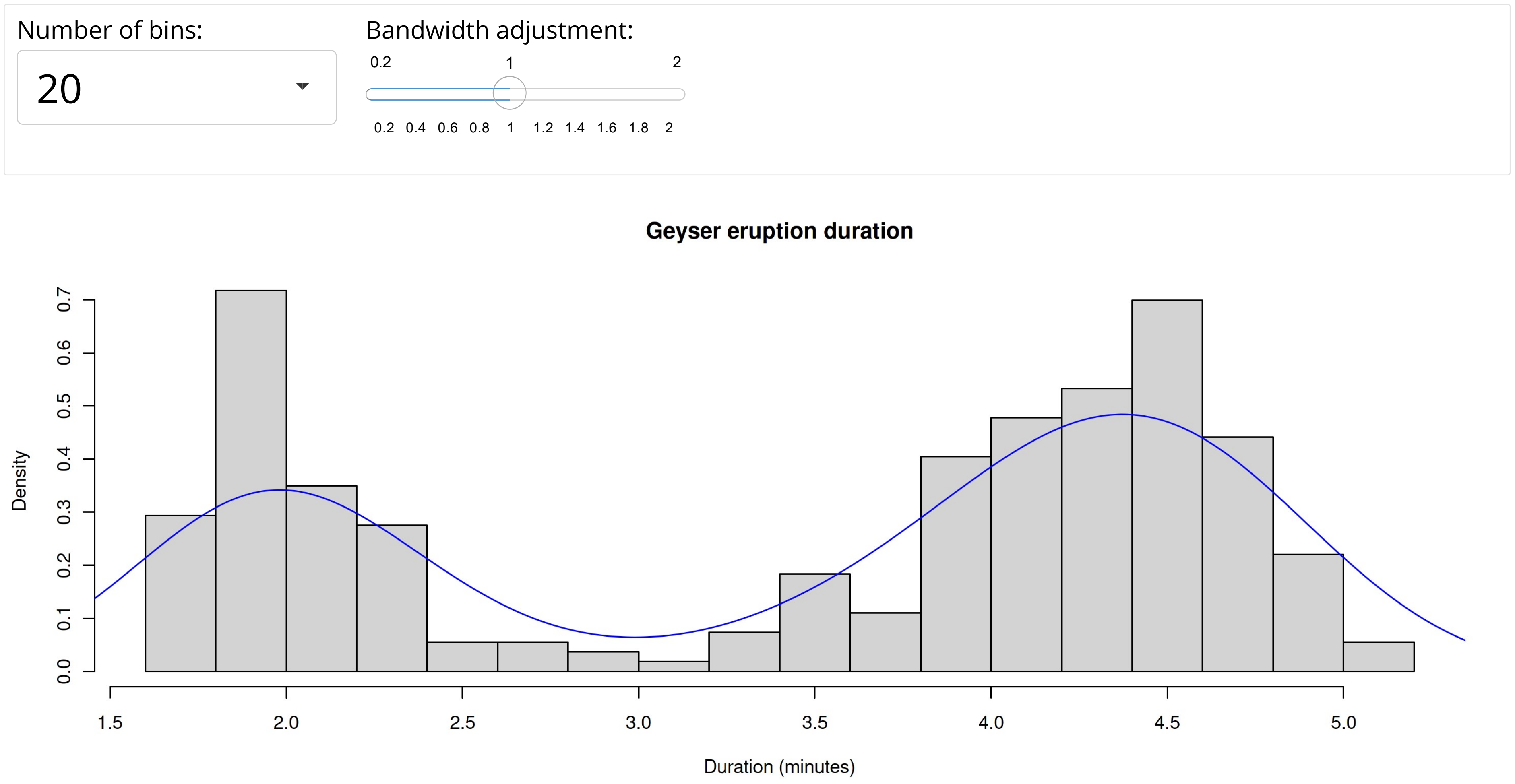
R Shiny

Justin Post

What is R Shiny?

- R Shiny Package (http://shiny.rstudio.com/)
  - Developed by RStudio
  - Allows for creation of apps and dashboards
- Usually a .R file (or two) with special code to create an app
  - `ui.R` (User Interface)
  - `server.R` (R functions that run/respond to UI)
  - `app.R` (both UI and server combined)
- But you can also just add them to HTML documents
- Requires no HTML, CSS, or JavaScript!

Example App



Number of bins: 20

Bandwidth adjustment: 1

Available Apps

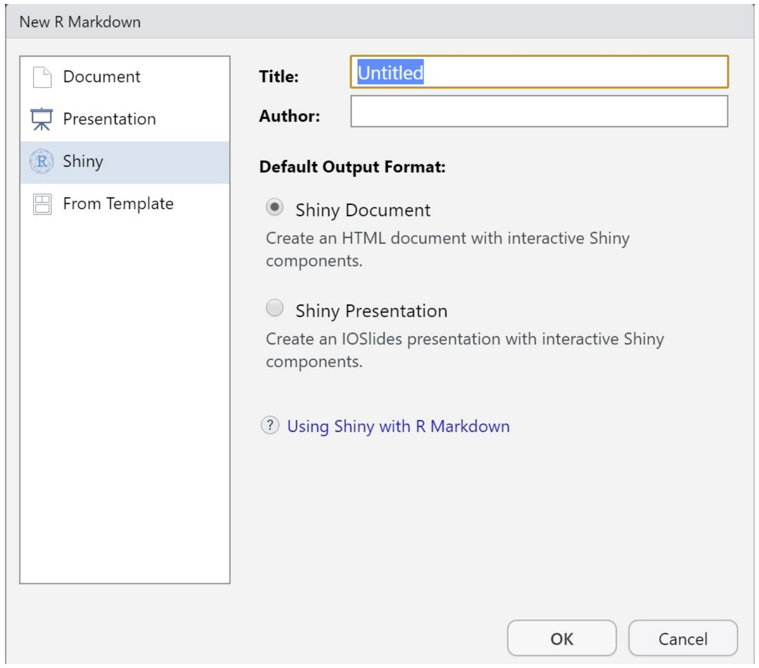
- Many available resources. Many have their source code available on github!
- Book of Apps for Statistics Teaching (BOAST) (https://sites.psu.edu/shinyapps/)
- Stat Concepts (https://github.com/gastonstat/shiny-introstats/)
- More Stat Concepts (https://www.researchgate.net/publication /298786680_Web_Application_Teaching_Tools_for_Statistics_Using_R_and_Shiny)
- Even more! (http://www.statistics.calpoly.edu/shiny)
- Shiny Gallery/Showcase (https://shiny.rstudio.com/gallery/)

Where to Start?

- Learn about user interface (UI) elements
  - Input widgets (sliders, numeric inputs, etc.)
  - Formatting of text
  - UI layout
- Understand how the server (R) backend works with the UI elements
  - Accessing UI inputs
  - Creating outputs

Create a shiny markdown doc

- File –> New file –> R Markdown
- Really, we just need to add `runtime: shiny` to the YAML header for an HTML doc!



Title: Untitled

Author:

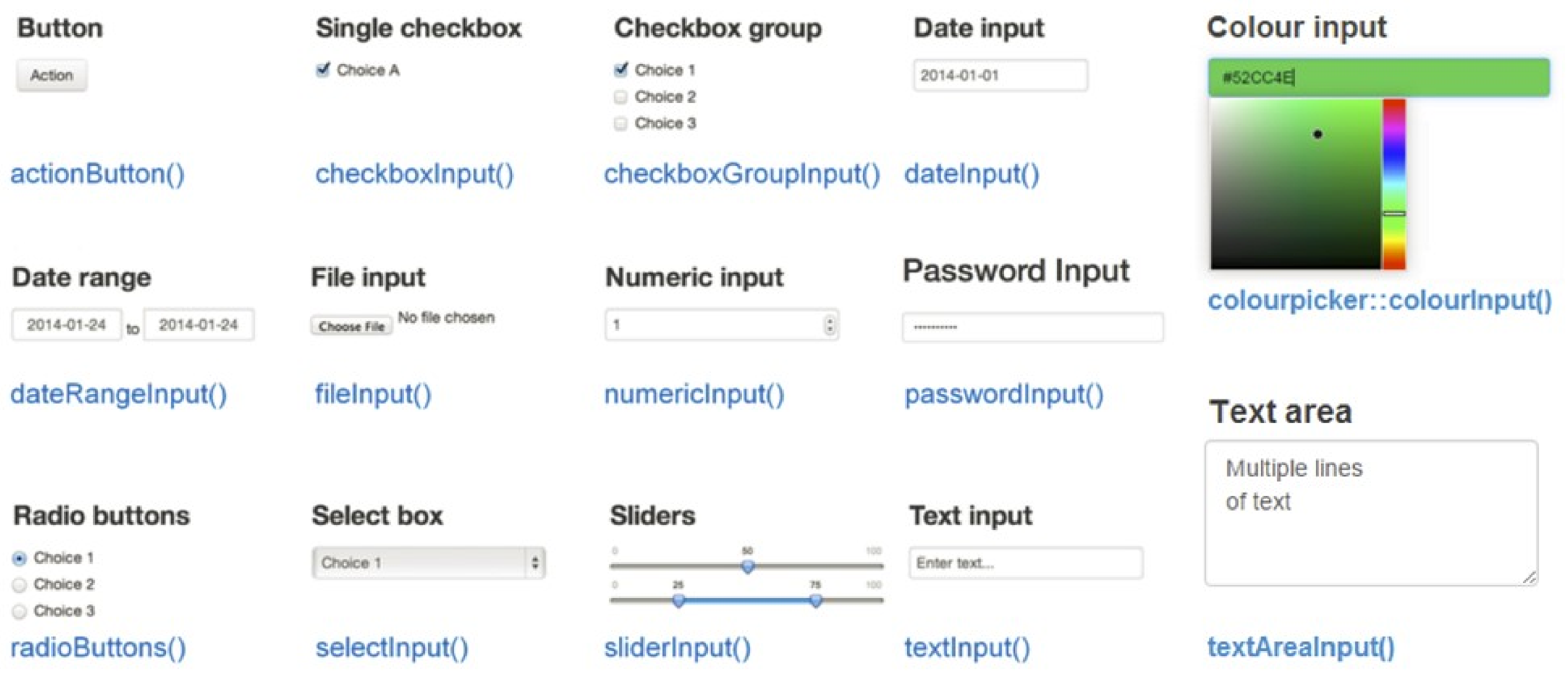
Default Output Format:

[✓ Shiny Document] Create an HTML document with interactive Shiny components.

Shiny Presentation Create an IOSlides presentation with interactive Shiny components.

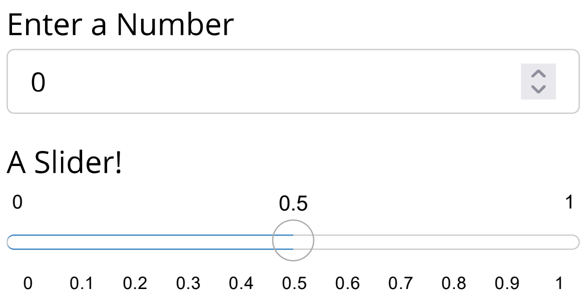
UI: Adding Widgets

- Shiny components (widgets & outputs) go in R code chunks
- Widgets can be added using their `*Input` functions



Widget Example

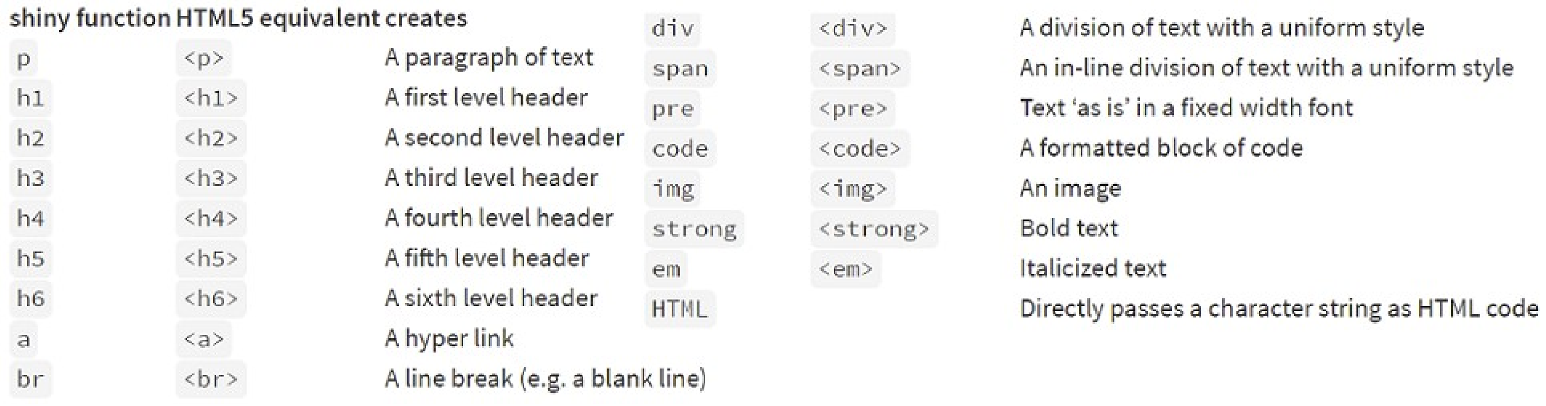
```
library(shiny)
numericInput("num", "Enter a Number", value = 0, min = 0, max = 100)
sliderInput("slide", label = "A Slider!", min = 0, max = 1, value = 0.5, step = 0.05)
```



UI: Adding Formatted Text

Can also add:

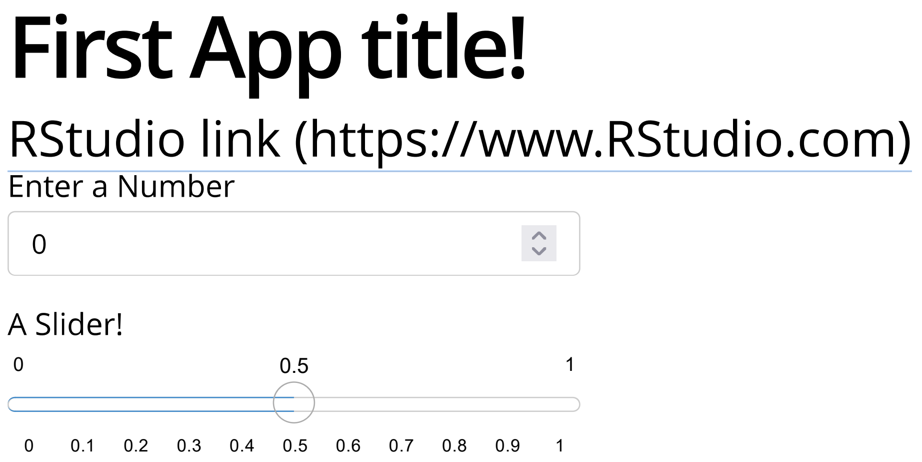
- Any plain strings
- Formatted text (using HTML type functions)



shiny function	HTML5 equivalent	creates
p	`<p>`	A paragraph of text
h1	`<h1>`	A first level header
h2	`<h2>`	A second level header
h3	`<h3>`	A third level header
h4	`<h4>`	A fourth level header
h5	`<h5>`	A fifth level header
h6	`<h6>`	A sixth level header
a	`<a>`	A hyper link
br	`<br>`	A line break (e.g. a blank line)
div	`<div>`	A division of text with a uniform style
span	`<span>`	An in-line division of text with a uniform style
pre	`<pre>`	Text 'as is' in a fixed width font
code	`<code>`	A formatted block of code
img	`<img>`	An image
strong	`<strong>`	Bold text
em	`<em>`	Italicized text
HTML		Directly passes a character string as HTML code

Widget & Text Example

```
h2("First App title!")
a("RStudio link", href = "https://www.RStudio.com")
numericInput("num", "Enter a Number", value = 0, min = 0, max = 100)
sliderInput("slide", label = "A Slider!", min = 0, max = 1, value = 0.5, step = 0.05)
```


First App title!

RStudio link (https://www.RStudio.com)

Enter a Number

0

A Slider!

UI: Formatting

- `inputPanel()` allows you to add user inputs, text, etc. in a single row
- Syntax:

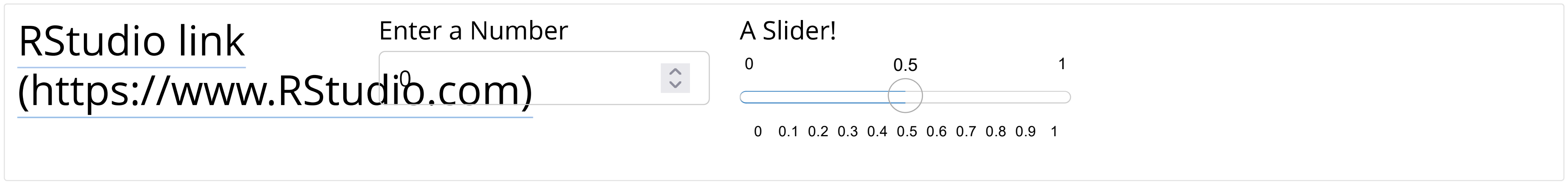
```
inputPanel(
 widgetName1(...),
 textFormatting(...),
 widgetName2(..),
)
```

Example

```
h2("First App title!")

inputPanel(
  a("RStudio link", href = "https://www.RStudio.com"),
  numericInput("num", "Enter a Number", value = 0, min = 0, max = 100),
  sliderInput("slide", label = "A Slider!", min = 0, max = 1, value = 0.5, step = 0.05)
)
```

First App title!

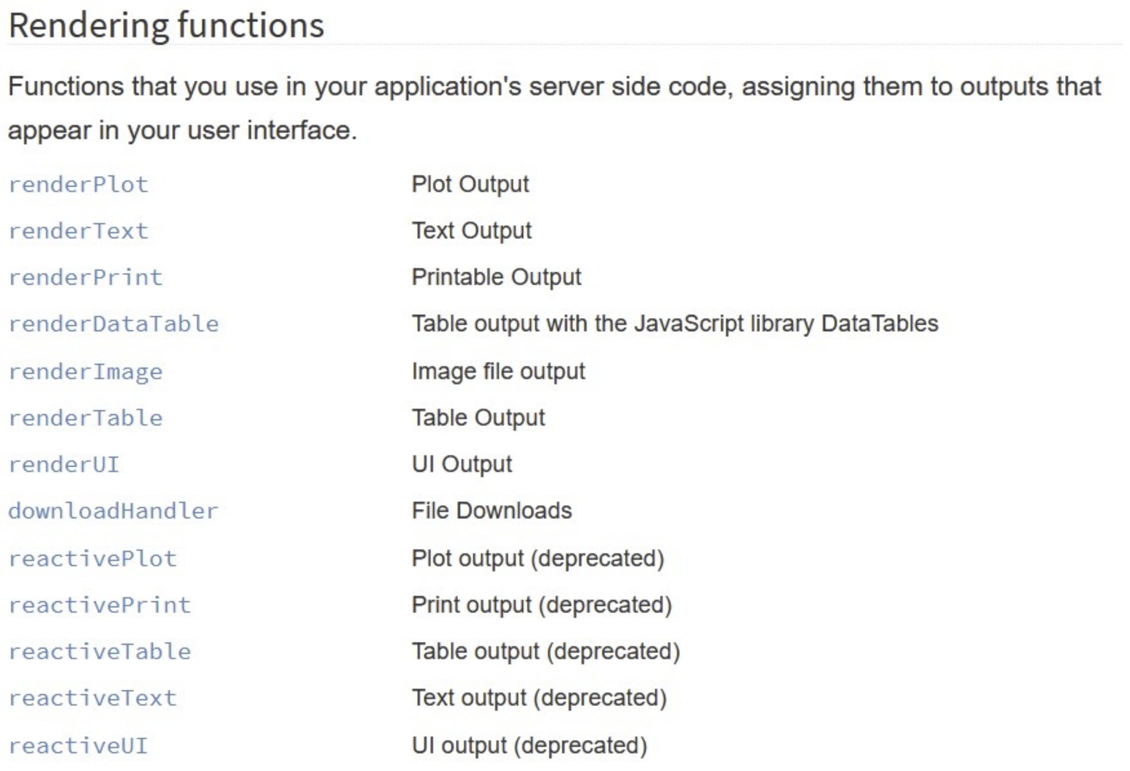


UI: More About Widgets

- Widgets all follow the same structure
- `widgetName("internalID", label = "Title the user sees", ...)`
- The `internalID` is how you access the inputs when creating plots, summaries, etc.

Server: Creating Outputs

- Outputs can be created using their `render*` functions



Rendering functions

Functions that you use in your application's server side code, assigning them to outputs that appear in your user interface.

`renderPlot`	Plot Output
`renderText`	Text Output
`renderPrint`	Printable Output
`renderDataTable`	Table output with the JavaScript library DataTables
`renderImage`	Image file output
`renderTable`	Table Output
`renderUI`	UI Output
`downloadHandler`	File Downloads
`reactivePlot`	Plot output (deprecated)
`reactivePrint`	Print output (deprecated)
`reactiveTable`	Table output (deprecated)
`reactiveText`	Text output (deprecated)
`reactiveUI`	UI output (deprecated)

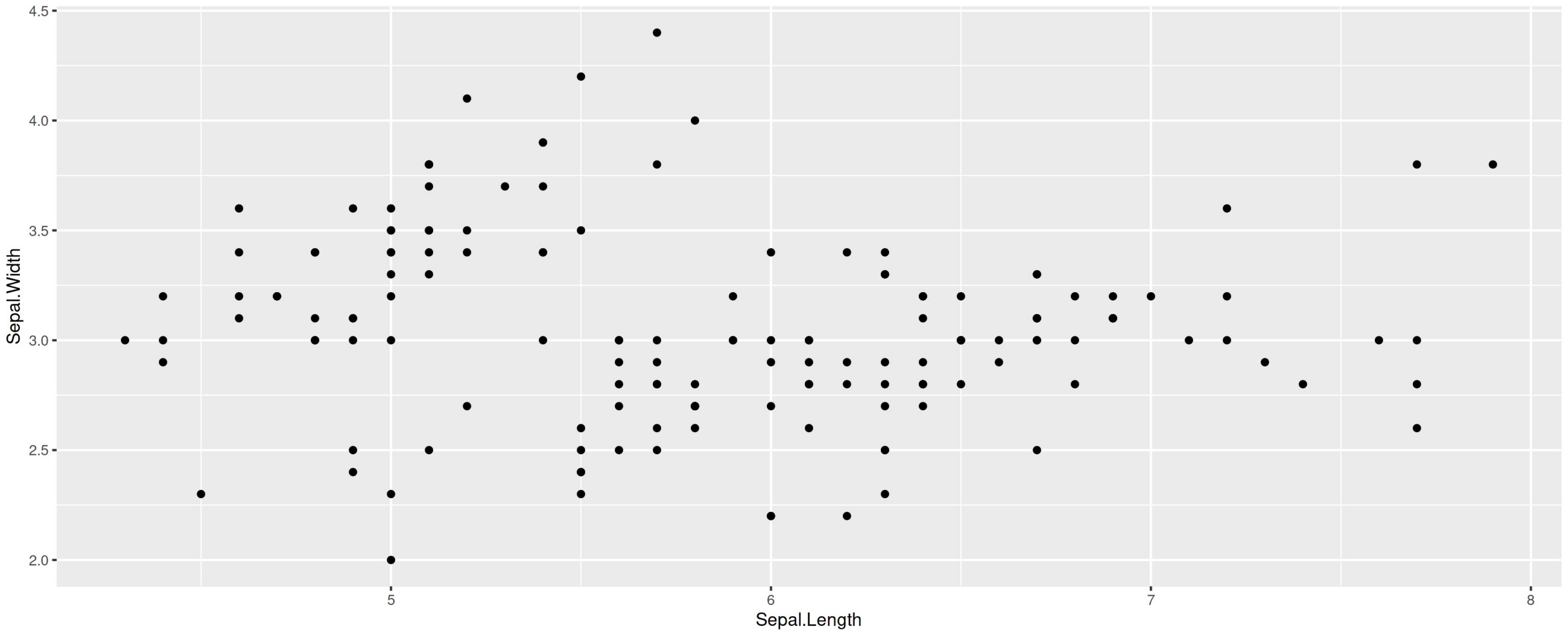
Plot Example

```
inputPanel(
  checkboxInput("addColor", "Color")
)

renderPlot({
  g <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))
  if(input$addColor){
    g + geom_point(aes(color = Species))
  } else {
      g + geom_point()
  }
})
```

Plot Example

Color ☐



Much more to learn!

- Stand alone apps
- Shiny themes
- Dashboards
- UI Layouts
- Reactive contexts
- Dynamic UIs
- Hosting an app https://www.shinyapps.io/ (shinyapps.io)

Example

Using the R Markdown shiny template file, do the following:

- Add a `checkboxInput()` after the `sliderInput()`
- Internally, reference the checkbox as `'prob'` and set the default value to `TRUE`
- In the `renderPlot()`, replace the code for the histogram with

```
hist(faithful$eruptions, probability = input$prob, breaks = as.numeric(input$n_breaks),
     xlab = "Duration (minutes)", main = "Geyser eruption duration")
if(input$prob){
  dens <- density(faithful$eruptions, adjust = input$bw_adjust)
  lines(dens, col = "blue")
}
```

Stand Alone Apps: Two File Approach

- Create folder for each App you create
- Each App's folder should have `ui.R` and `server.R` files
- (If single file, `app.R` in each folder)
- Can create with File –> New File –> Shiny Web App (Go ahead and make a two file app)

`ui.R` Basic Layout

```
library(shiny)

ui <- fluidPage(
    titlePanel(),

  sidebarLayout(
    sidebarPanel(#usually widgets
      ),
    mainPanel(#usually output
      )
  )
)
```

UI Common Layout

title panel

```
sidebar panel
|
|
|
|
|
|
V
Usually User Inputs
```

```
main panel content-------------------->
|
|
|
| Usually Output
| that Reacts to User Input
|
V
```

`server.R` Basic File

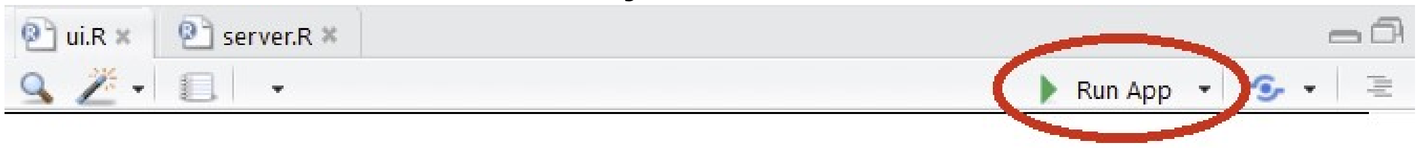
```
library(shiny)

shinyServer(function(input, output, session) {

})
```

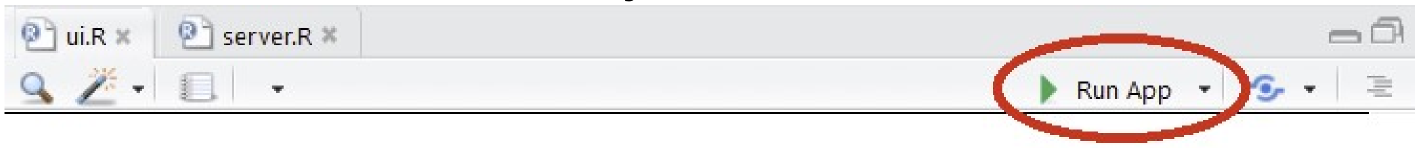
Running an App

- While `ui.R` or `server.R` is your active window, click the **Run App** button



Running an App

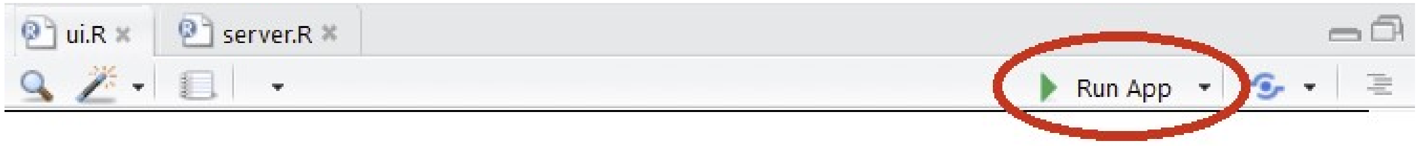
- While `ui.R` or `server.R` is your active window, click the **Run App** button



- Use `shiny::runApp()` function
  - ex: `runApp("path/to/ui_or_server_or_app.R")`

Running an App

- While `ui.R` or `server.R` is your active window, click the **Run App** button



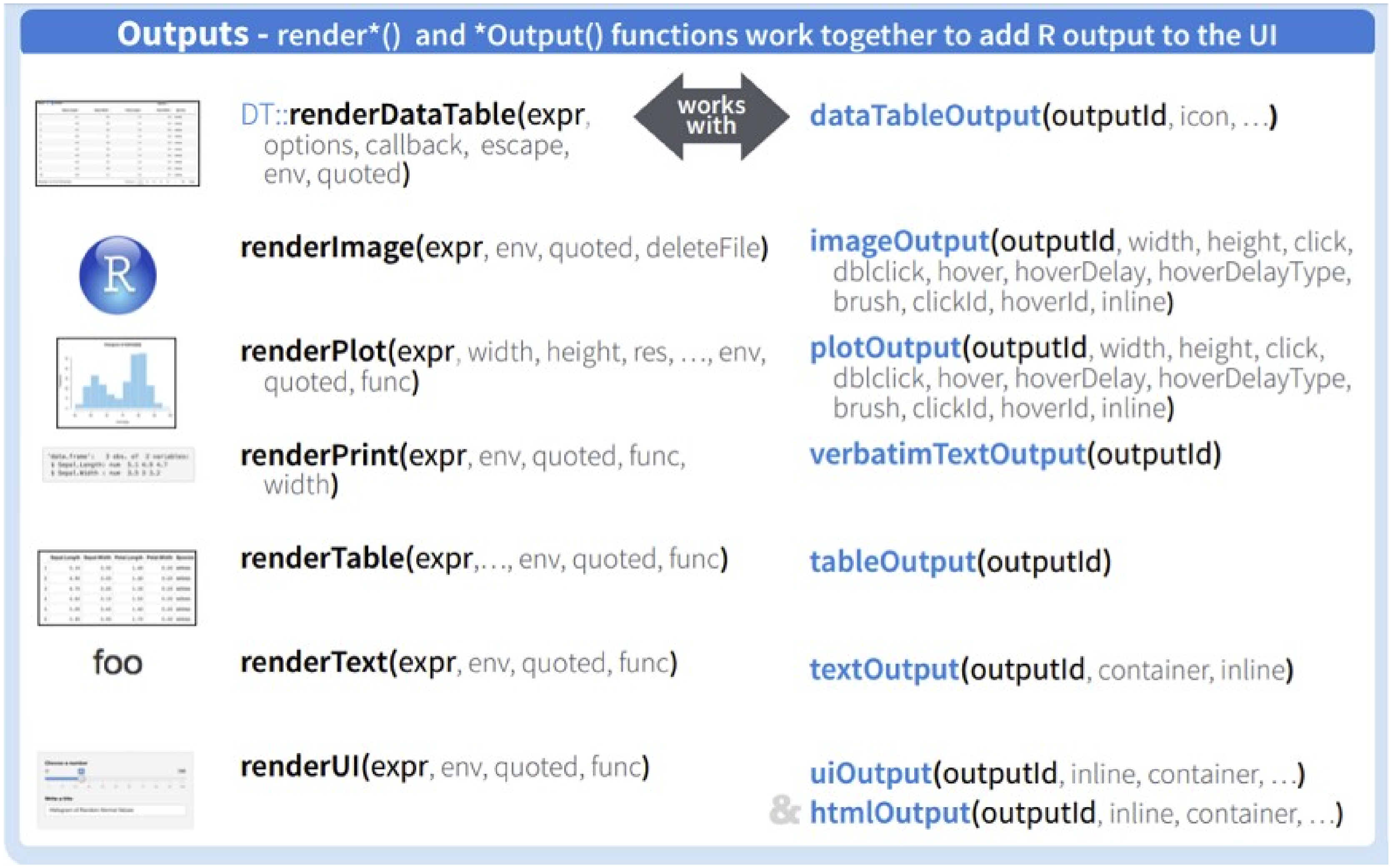
- Use `shiny::runApp()` function
    - ex: `runApp("path/to/ui_or_server_or_app.R")`
- Running App will tie up R console!
- End by hitting Esc or closing shiny app
- Take a minute and run the template app

Adding to the UI

Using a comma to separate items, you can add

- Any plain strings
- Widgets
- Formatted text (using HTML type functions)
- Output from things created in the `server.R` file

Sharing Between Server and UI



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, 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)**
renderText(expr, env, quoted, func**)**		**textOutput(outputId**, container, inline**)**
renderUI(expr, env, quoted, func**)**		**uiOutput(outputId**, inline, container, …**)** & **htmlOutput(outputId**, inline, container, …**)**

Adding to the UI - Example Syntax

```
library(shiny)
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      h2("Widgets/Text"),
      numericInput("NI",label="Intercept",value=10),
      sliderInput("SI",label="Slope",min=-1,max=1,value=0,step=0.1),
      "More text",
      br(),
      a(href="http://www.rstudio.com",target="_blank","Link to RStudio")
    ),
    mainPanel(plotOutput("dataPlot"), #dataPlot is name of "plot" object in server
              textOutput("dataInfo"), #dataInfo is name of "text" object in server
              dataTableOutput("dataTable") #dataTable is name of "data" object in server
    )
  )
)
```

Widgets/Text

Intercept

10

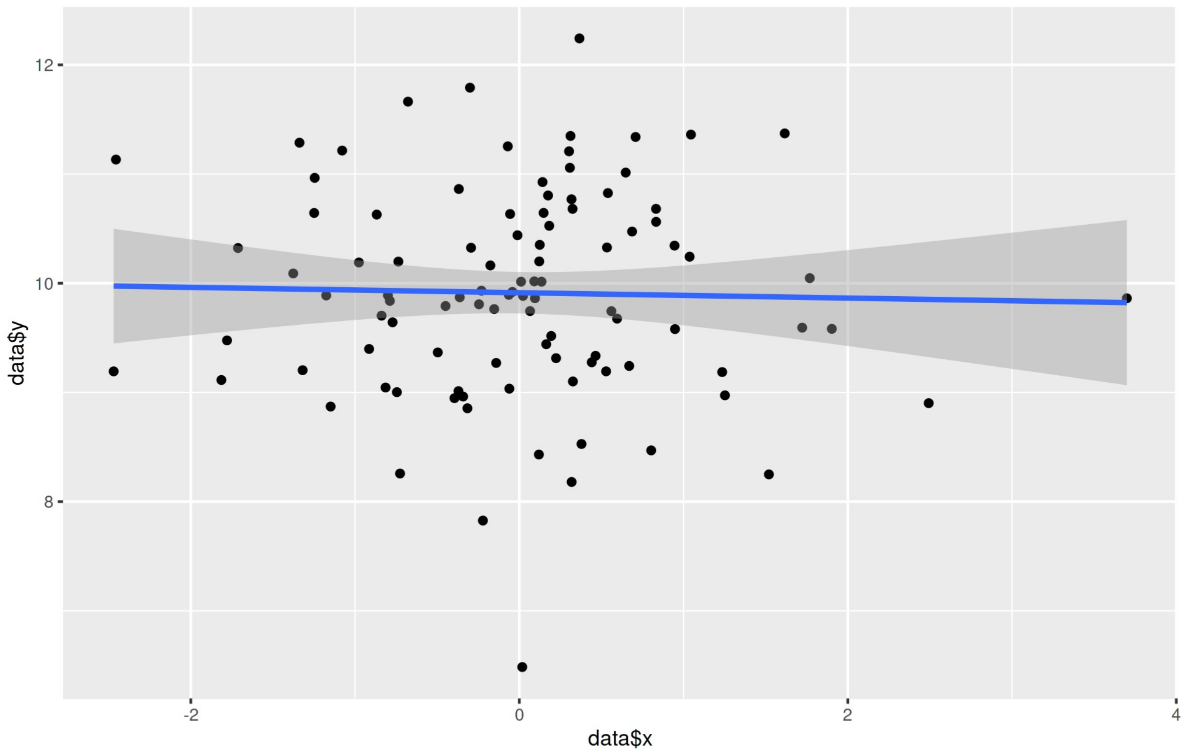
Slope

-1 0 1

-1 -0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8 1

More text

Link to RStudio (http://www.rstudio.com)



The true intercept is 10. The true slope is 0.

Show 10 **entries** **Search:**

y x

Summary So Far

`ui.r`

- Controls layout of app
- Basic layout uses a sidebar panel and a main panel
- Use strings, formatted (html style) text, widgets (`*Input` functions), and output from `server.r` (`*Output` functions)
- Separate items with commas

Quick Try

In the two-file template app:

- In the sidebar, above the slider, add a link to the old faithful page here:

https://www.yellowstonepark.com/things-to-do/geysers-hot-springs/about-old-faithful/ (https://www.yellowstonepark.com/things-to-do/geysers-hot-springs/about-old-faithful/)

- Below the slider in the sidebar, Add a radio button widget that is titled `Summaries of Data`
  - This should allow the user to select `none`, `eruptions`, `waiting` or `both`
  - The default value should be `none`

Server file

`server.r` also called the 'back-end' because it works behind-the-scenes

```
## set up server
shinyServer(function(input, output, session) {
  # add stuff
})
```

Server file

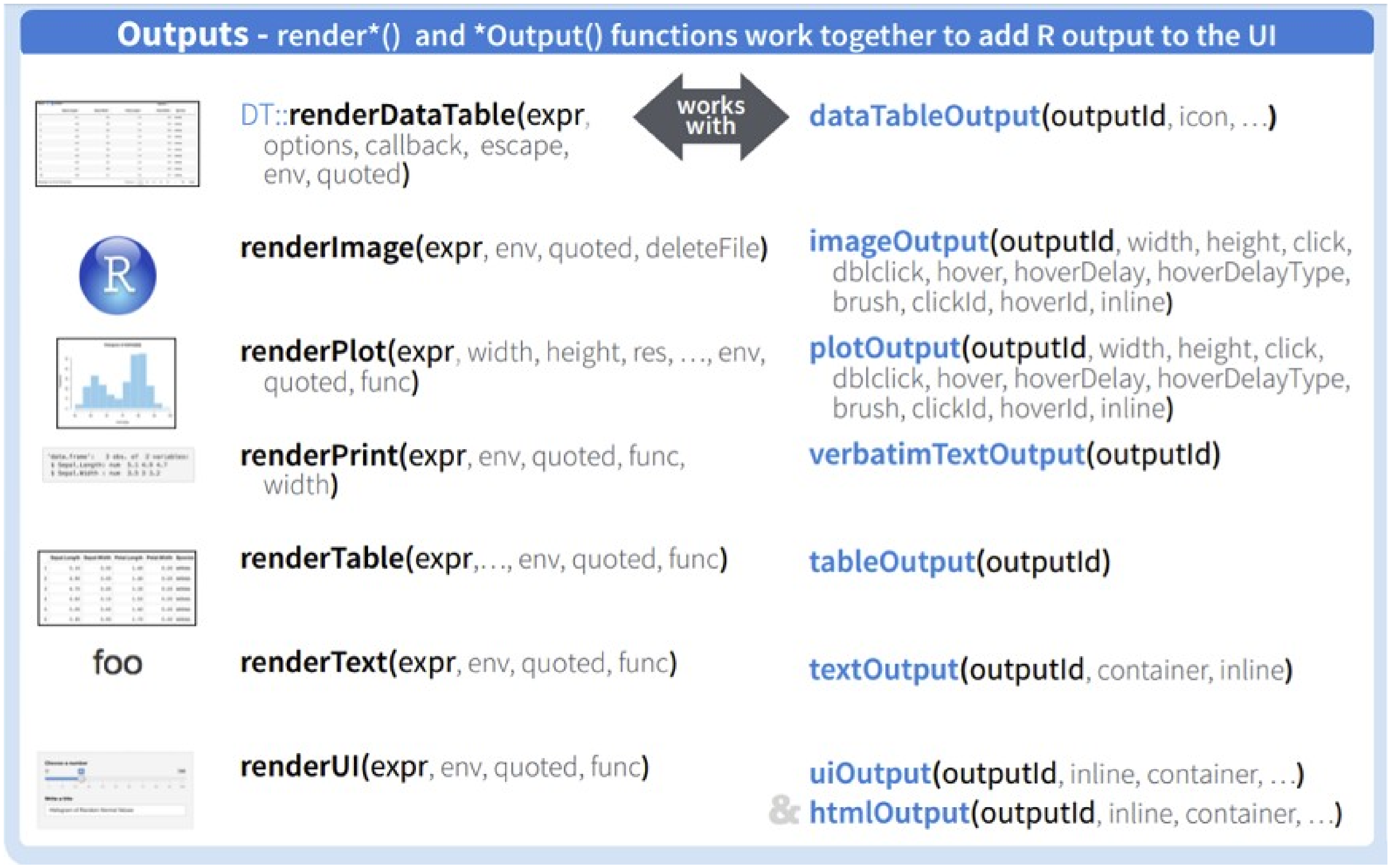
`server.r` also called the 'back-end' because it works behind-the-scenes

```
## set up server
shinyServer(function(input, output, session) {
  # add stuff
})
```

The arguments for the server are `input`, `output`, and `session`. Allow us to

1. Take in inputs from the UI
2. Run functions on them
3. Create outputs to send back

Creating Output to Send to UI



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, 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, …)

Creating Output to Send to UI

Example syntax

```
shinyServer(function(input,output){
  output$nameOfOutputObject <- renderPlot(
    #code that will return a plot
  )

  output$otherOutput <- renderText(
    #code that will return something that R can coerce to a string
  )
})

#in ui.r file, reference would look like
plotOutput("nameOfOutputObject")
textOutput("otherOutput")
```

Accessing Input Values in server.R

- Every input object has an `inputId`

Accessing Input Values in server.R

- Every input object has an `inputId`
- In `server.r`, reference input value by

```
input$inputId
```

Accessing Input Values in server.R

- Every input object has an `inputId`
- In `server.r`, reference input value by

```
input$inputId
```

- Example

```
#input widget code from ui.r file
sliderInput(inputId = "slide",label = "Select the Range Here",min = 0,max = 1,
            value = c(0,1))
#reference in server.r might look like
output$userPlot<-renderPlot({
  range<-input$slide
  #create plot that changes based on user input
  plot(data,xlim=range)
})
```

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them
- Notice how we name our output objects

```
output$nameOfOutputObject <- renderPlot(...)
```

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them
- Notice how we name our output objects

```
output$nameOfOutputObject <- renderPlot(...)
```

- Notice how we access our inputs

```
output$nameOfOutputObject <- renderPlot(
   range <- input$slide
   ...
)
```

Quick Try: Using the template app

- Add a text output object in the `server.R` file (use `renderText()`) that returns the current value of the input slider
- To do this, call `renderText(input$bins)` and just reference the input
- Add a `textOutput()` in the `ui.R` file!
- Add a `renderPrint()` that outputs the numeric summary requested from the `radioButtons()` created earlier (you'll also need a corresponding `verbatimTextOutput()` in the UI)

```
if(input$radio == "eruptions"){
  summary(faithful$eruptions)
} else if(input$radio == "waiting"){
  summary(faithful$waiting)
} else if(input$radio == "both"){
  summary(faithful)
}
```

Reactivity

- Output objects do not have to depend on an input
- Those that don't will be static
- Any 'chunk' of code in `server.r` that references a user input must be **reactive**
- When a user changes an input the `input$` value associated invalidates and causes appropriate chunks of code to **re-evaluate** in the server

Example Reactivity

```
##code chunk "reacts" to and re-evaluates if
##input$sampleSize or input$otherInput changes

output$dataPlot <- renderPlot({

  n <- input$sampleSize
  input$otherInput #not used anywhere else, but entire
                   #renderPlot chunk still re-evaluates
                   #if changed

  hist(rbinom(n = 1, size = n, prob = 0.4))

})
```

- type `shiny::runExample("01_hello")` into the console

Reactivity

- `server.r` can run any R code, but can't access inputs unless put into a reactive context
- All `render*` functions are reactive contexts

Error Using Reactive Variables

Following returns the error:

```
shinyApp(ui <- fluidPage(
                numericInput("NI", "Give me a number", value = 10),
                textOutput("string")
                ),

         shinyServer(function(input, output){
                print(input$NI + 10)
                output$string <- renderText(paste("value plus 10 is", input$NI + 10))
         }
))
```

Warning: 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.)

Other Reactive Contexts

- `reactive({})` creates a reactive context and allows for the creation of a new variable
- `reactiveValues({})` similar to `reactive` but is easier to create multiple items
- `observe({})` function allows for reactivity and reevaluation of code
- `observeEvent({})` similar to observe but allows for more control
- `eventReactive({})` similar to `observeEvent()` but you can return something like `reactive()` does

More on `reactive({})`

- 'Wraps' a normal expression to create a reactive expression (code user can cause to change)
- Can read reactive values and call other reactive expressions
- Usually used to save something you'll call in multiple other places
- Access object as though calling it as a function

More on `reactive({})`

- Access object as though calling it as a function

```
shinyServer(function(input,output){
  #Creates a new reactive variable
  newVar <- reactive({
    val <- c(input$NI + 10, input$NI * 3)
  })

  output$textString <- renderText({
    value <- newVar()  #access like a function!
    paste0("Input plus 10 is ", value[1], " and Input times 3 is ", value[2])
  })

  output$otherString <- renderText({
    value <- newVar()
    paste0(value[1], ",", value[2])
  })
})
```

`reactiveValues()`

- Create list like object with `reactiveValues()`
- Access elements via `$`
- Elements can be changed in a reactive context

```
shinyServer(function(input,output){
  #Creates a new reactive values
  vals <- reactiveValues(data = rnorm(150), sliderval = 0)

  observe({vals$sliderval <- input$slider})

  output$textString <- renderText({
    paste0("The value is ", vals$sliderval)
  })

  output$hist <- renderPlot({
    hist(vals$data)
  })
})
```

`observe({})`

- Can read reactive values and call reactive expressions

```
shinyServer(function(input,output){
  #would now print to console
  observe({print(input$NI+10)})

    #update UI
    observe({
        input$noPitch
        updateCheckboxGroupInput(session, "pitchTypeChoice", selected = c(""))
    })
})
```

`observeEvent({})`

- Similar to observe but allows for control of dependencies
- Place explicit dependencies prior to `{}`

```
shinyServer(function(input,output){
  #would now print to console
  observeEvent(input$NI, {print(input$data)})

    #update UI
    observe({
        input$noPitch
        updateCheckboxGroupInput(session, "pitchTypeChoice", selected = c(""))
    })
})
```

`eventReactive()`

- Similar to `observeEvent()` but returns object similar to `reactive()` (use like a function)

```
shinyServer(function(input,output){
  #would now print to console
  update <- eventReactive(input$submit, {list(gamma = input$gamma, alpha = input$alpha)})

    #update UI
    output$gammaDist <- renderText({
      paste0("The parameters are ", update())
    })
})
```

Quick Try

- Continue building from the template app
- Add a `numericInput()` element and an `actionButtion()`
- Create a `square <- reactive({})` object that stores the square of the numeric input
- Use the output of `reactive()` to add to the `renderText()` using something like `paste("Bins:", input$bins, "The square is ", square())`
- Use `observeEvent()` to print the value of `square()` (`print(square)`) to the console when the action button is pressed

Developing an App

- **Highly Recommended:**

Draw out what you want the app to look like

- Write R code to complete your app in a static manner!
- Determine dependencies and consider reactive functions to use
- Translate to appropriate Shiny output functions

Summary So Far

`ui.r`

- Controls layout of app
- Basic layout uses a sidebar panel and a main panel
- Use strings, formatted (html style) text, widgets (`*Input` functions), and output from `server.r` (`*Output` functions)
- Separate items with commas

`server.r`

- Back-end for app
- Create outputs that react to inputs (`render*` functions)
- To respond to input, must be in a reactive context

Dynamic UI

- Often want to update UI based on user input!
- Methods for updating UI
  - `update*` functions
  - `renderUI()`/`uiOutput()`
  - `conditionalPanel()`

Using update* Functions

- Every input widget has a corresponding update function
    - `updateActionButton()`
    - `updateCheckboxInput()`
    - `updateNumericInput()`
    - …

Using update* Functions

- Every input widget has a corresponding update function
  - `updateActionButton()`
  - `updateCheckboxInput()`
  - `updateNumericInput()`
  - ...
- Require session argument on server() function

```
shinyServer(function(input, output, session) {
  ##  do stuff
})
```

Using update* Functions

- Every input widget has a corresponding update function
    - `updateActionButton()`
    - `updateCheckboxInput()`
    - `updateNumericInput()`
    - …
- Require session argument on server() function

```
shinyServer(function(input, output, session) {
  ##  do stuff
})
```

- After all observers (reactive things) evaluate, updater sends message back to client

Using update* Functions

- Syntax of `update*` functions similar to the functions that created the inputs

Example syntax:

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA,
  width = NULL)

updateNumericInput(session, inputId, label = NULL, value = NULL,
  min = NULL, max = NULL, step = NULL)
```

Using update* Functions

- Syntax of `update*` functions similar to the functions that created the inputs

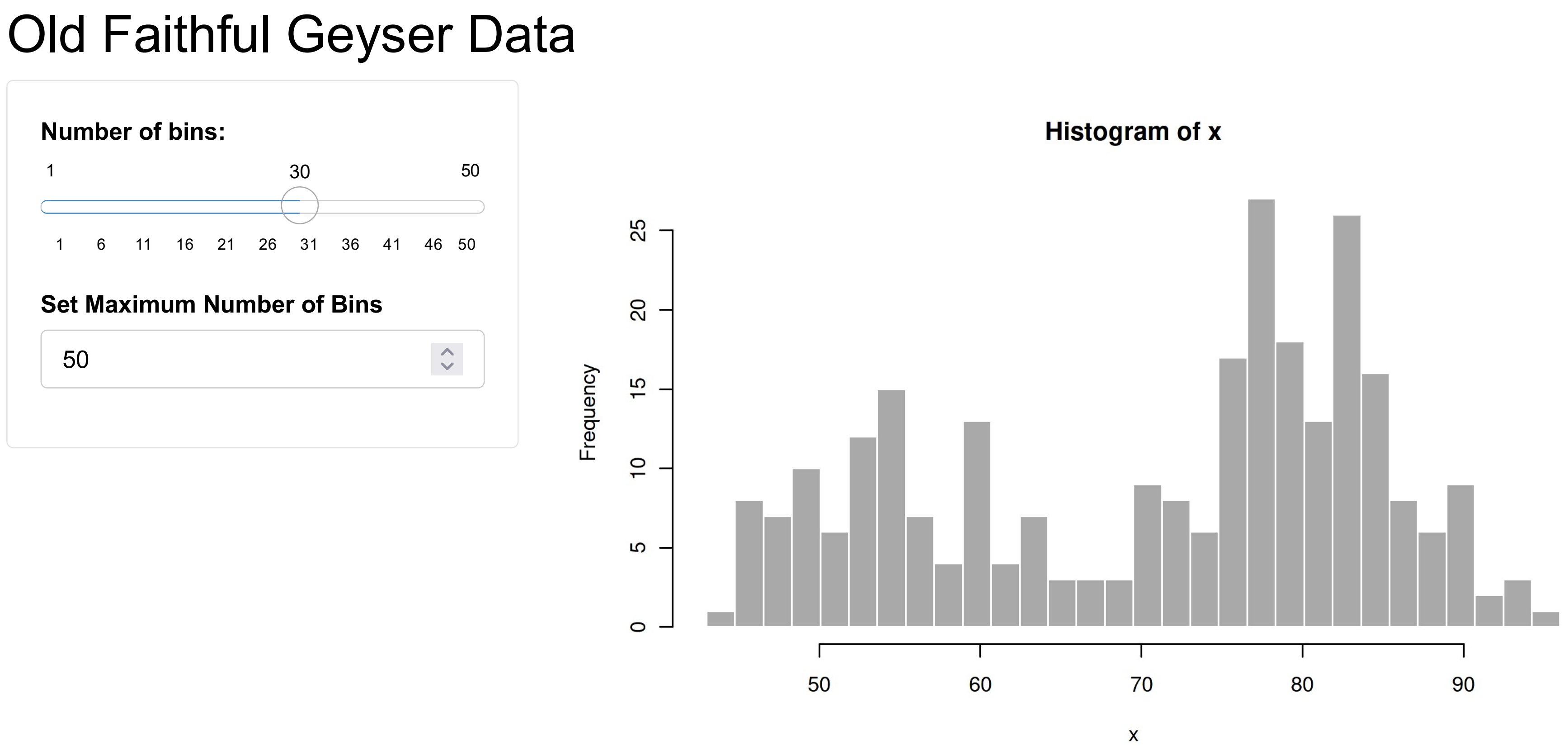
Example syntax:

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA,
  width = NULL)

updateNumericInput(session, inputId, label = NULL, value = NULL,
  min = NULL, max = NULL, step = NULL)
```

- Any arguments with `NULL` values ignored (i.e. will not result in any changes to the input object)
- For `radioButtons()`, `checkboxGroupInput()`, and `selectInput()`, the set of choices can be cleared by using `choices = character(0)` (similary for the set of selected)

Using `update*` Functions


Old Faithful Geyser Data

Number of bins: 30

Set Maximum Number of Bins 50

`updateSliderInput()` (First Attempt)

```
  ui <- fluidPage(
        ...
        sidebarPanel(
           sliderInput("bins", "Number of bins:",
                       min = 1, max = 50, value = 30),
           numericInput("maxBins", label = "Set Maximum Number of Bins",
                        value = 50, min = 1, max = 100)
        ),
        ...
  ),
  server <- function(input, output, session) {
      ...
      updateSliderInput(session, "bins", max = input$maxBins)
  }
)
```

What is our issue?

`updateSliderInput()` (Fixed)

```
ui <- fluidPage(
      ...
      sidebarPanel(
         sliderInput("bins", "Number of bins:",
                     min = 1, max = 50, value = 30),
         numericInput("maxBins", label = "Set Maximum Number of Bins",
                      value = 50, min = 1, max = 100)
      ),
      ...
)
server <- function(input, output, session) {
    ...
    observe({updateSliderInput(session, "bins", max = input$maxBins)})
}
```

`update*` UI Functions

- Use the template app
- Try to add a numeric input for the user to specify the largest value of the slider
- Use the `updateSliderInput` function to update the max of the slider
- Don't forget `observe`!
- Then, use the `actionButton` to only update the slider when it is pressed (so no intermediate updates while typing)

`renderUI()` and **`uiOutput()`**

- Alternatively, `renderUI()` and `uiOutput()` can be used

`renderUI()` and `uiOutput()`

- Alternatively, `renderUI()` and `uiOutput()` can be used
- Shiny essentially writes HTML/JavaScript for us!

```
print(fluidPage(titlePanel(title = "Hi"),
                sidebarLayout(sidebarPanel(), mainPanel())))
```

```
## <div class="container-fluid">
##   <h2>Hi</h2>
##   <div class="row">
##     <div class="col-sm-4">
##       <form class="well" role="complementary"></form>
##     </div>
##     <div class="col-sm-8" role="main"></div>
##   </div>
## </div>
```

`renderUI()` and `uiOutput()`

- Alternatively, `renderUI()` and `uiOutput()` can be used
- Shiny essentially writes HTML/JavaScript for us!

```
print(numericInput("id", "Label User Sees", value = 10))
```

```
## <div class="form-group shiny-input-container">
##   <label class="control-label" id="id-label" for="id">Label User Sees</label>
##   <input id="id" type="number" class="form-control" value="10"/>
## </div>
```

`renderUI()` and `uiOutput()`

`renderUI()`

- Makes a **reactive version** of a function that generates HTML (like any widget)
- Have `renderUI()` return a shiny 'tag object,', HTML, or a list of these

- Use with `uiOutput()` in UI file
- Interprets the HTML and outputs appropriately (usually a `div` element)

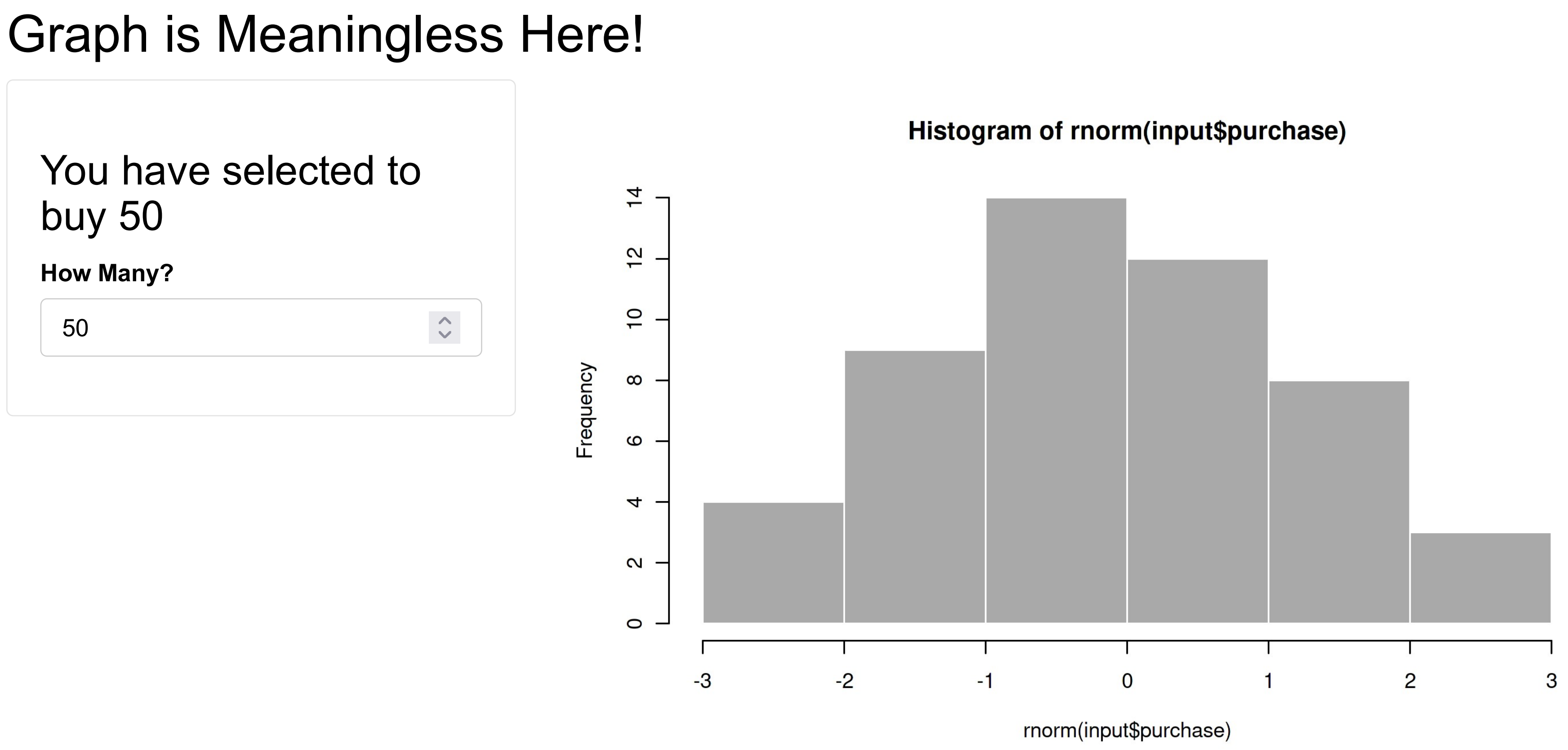
`renderUI()` and `uiOutput()` (updating a widget)

```
ui <- fluidPage(
      ...
      sidebarPanel(
          uiOutput("slider"),
          numericInput("maxBins", label = "Set Maximum Number of Bins",
                       value = 50, min = 1, max = 100)
      ),
      ...
),
server <- function(input, output, session) {
    ...
    output$slider <- renderUI({
        sliderInput("bins", "Number of bins:", min = 1,
                    max = input$maxBins, value = 30)
    })
}
```

`renderUI()` and `uiOutput()` (outputting HTML)

```
ui <- fluidPage(
      ...
      sidebarPanel(
           uiOutput("info"),
           numericInput("purchase", label = "How Many?",
                        value = 50, min = 0, max = 100)
      ),
      ...
),
server <- function(input, output, session) {
    ...
      output$info <- renderUI({
        text <- paste0("You have selected to buy ", input$purchase)
        h3(text)
      })
}
```

`renderUI()` and `uiOutput()`



Graph is Meaningless Here!

You have selected to buy 50

How Many?

50

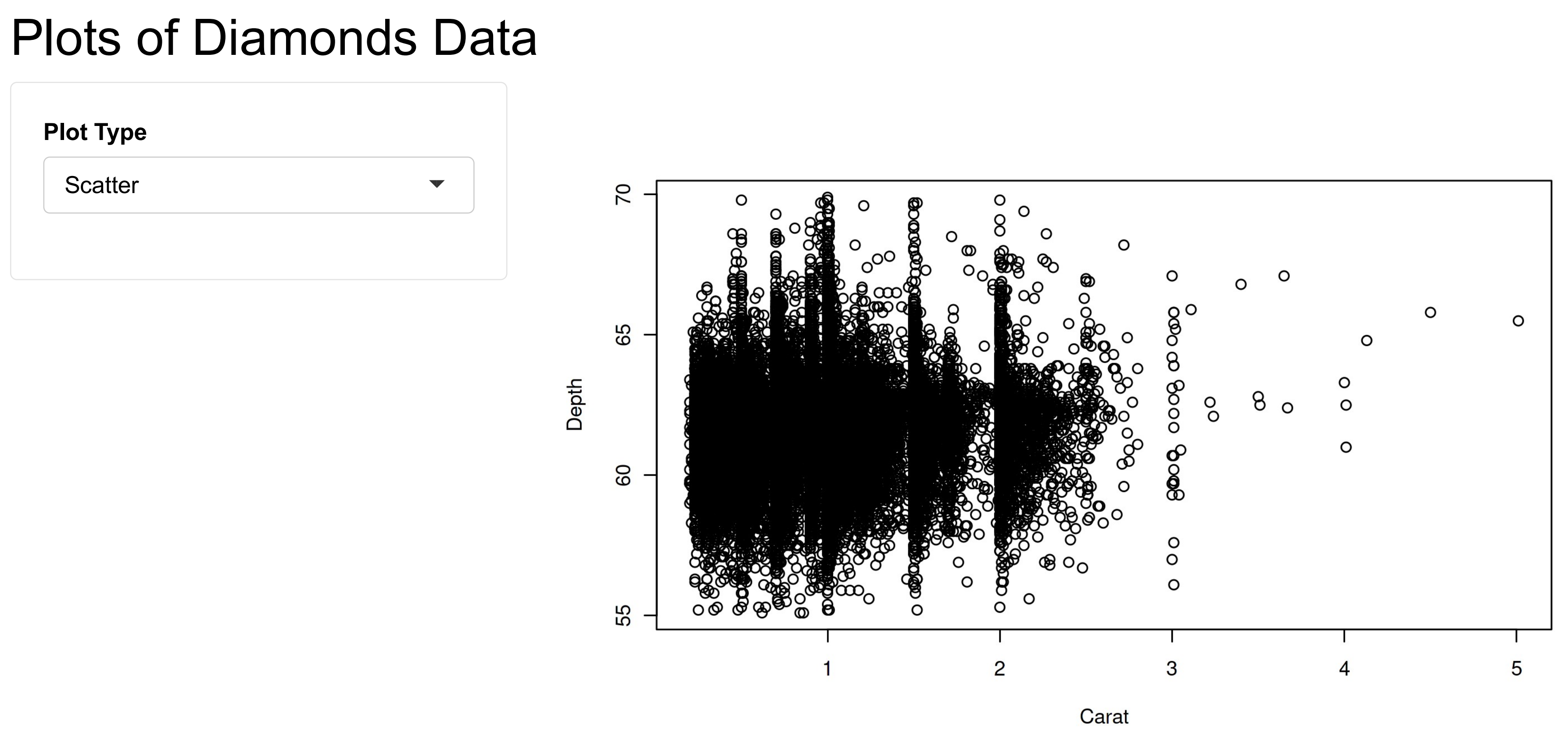
`renderUI()` and `uiOutput()`

- Use the template app
- Try to add some dynamic updating text to the UI

`conditionalPanel()`

- Create a 'panel' that is only visible if a condition is met
- Condition can depend on input or output value
- Accessed differently! (Use a '.' not a '$')

`conditionalPanel()`



Plots of Diamonds Data

Plot Type

Scatter

conditionalPanel()

```
...
sidebarPanel(
  selectInput("plotType", "Plot Type",
            c(Scatter = "scatter", Histogram = "hist")),

  # Only show this panel if the plot type is a histogram
  conditionalPanel(condition = "input.plotType == 'hist'",
          selectInput("breaks", "Breaks",
              c("Sturges", "Scott", "Freedman-Diaconis", "[Custom]" = "custom")),

      # Secondary conditonalPanel, Only show this panel if Custom is selected
      conditionalPanel(
          condition = "input.breaks == 'custom'",
          sliderInput("breakCount", "Break Count", min = 1, max = 200, value = 40)
      )
  )
)
```

`conditionalPanel()`

- Use the template app
- Try to add a new UI element if a condition on the slider is met

Dynamic UI Recap

- Often want to update UI based on user input!
- Recall: UI and server basically pass lists back and forth
- Methods for updating UI
    - `update*` functions
    - `renderUI()`/`uiOutput()`
    - `conditionalPanel()`

Custom UI Layout

- Contents of UI wrapped in `fluidPage()`
- Content can be wrapped in `fluidRow()`'s
- Columns can be created with `column(width, offset)`
- Columns should sum to 12 in total width for an 'area'!

Customized Layout

fluidRow with columns

2nd column

column widths in a given row should add to 12

2nd fluidRow below above row

Columns can contain their own fluidRow as well, allowing for a lot of customization of layouts!

subcol subcol

```
shinyUI(fluidPage(
  fluidRow(
    column(2,"fluidRow with columns--------...---------"),
    column(6,"2nd column------------...--------"),
    column(4,"column widths in a given row must add to 12------...---------")),
  fluidRow(tags$hr()),
  fluidRow(
    column(6,"2nd fluidRow below above row----...-----"),
    column(6,
           fluidRow("Columns can contain their own fluidRow as well, allowing for a lot o
           fluidRow(
             column(3,"subcol ----...-----"),
             column(9,"subcol ----...-----")
           ))
  )
))
```

More customization

- Can wrap UI elements with `wellPanel()`
    - Puts elements together in grey area

More customization

- Can wrap UI elements with `wellPanel()`
    - Puts elements together in grey area
- `tabPanel()` is like a UI page of its own
    - Used with `tabsetPanel()` or `navbarPage()`

```
tabsetPanel(
tabPanel("Title1", "contents"),
tabPanel("Title2", "contents")
)
```

More customization

- `navbarMenu()`
- `navlistPanel()`
- `fixedPanel()`
- `fixedrow()`

Quick Try

- Let's use `wellPanel()`, `fluidRow()`, and `columns()`
- Recreate side-bar layout

Recap

`ui.r`

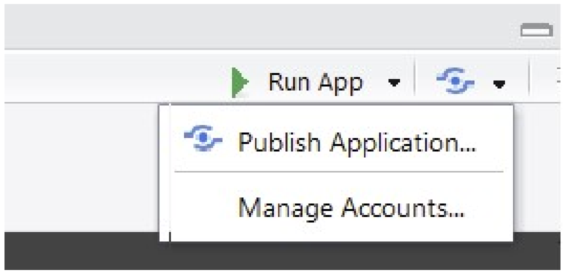
- Controls layout of app (can use standard layouts or customize)
- Use strings, formatted (html style) text, widgets (`*Input` functions), and output from `server.r` (`*Output` functions)
- Separate items with commas
- Update inputs, render HTML reactively, conditionally show input

`server.r`

- Back-end for app
- Create outputs that react to inputs (`render*` functions)
- To respond to input, must be in a reactive context
- Code can be included prior to `shinyServer()`

Sharing App

- Running App locally ties up your system
- Others can't access it!
- Can host as a URL on a shiny server
- Can host apps on shinyapps.io (powered by RStudio) (https://shiny.rstudio.com/articles/shinyapps.html)
  - Free, but number of connects and hours limited
  - Gives stats about usage
  - Integrated into R Studio



`shinyapps.io`

- First install the `rsconnect` package
- Go to shinyapps.io, log in, and then click on tokens in the top right
- Click on show on the right hand side
- In the box that pops up, click on show secret
- Copy that line of code and run it in your console

`shinyapps.io`

- Go to your `ui.R` or `server.R`
- Click on the publish icon in the top right
- Click publish on the box that pops up
- It may take a few minutes to deploy… but that's it!
- Go to shinyapps.io to see your app

(You can view usage stats and what-not)

Sharing App

- Can host files on gitHub and run locally from there using

```
shiny::runGitHub("<your repository name>", "<your user name>")
```

- Can host at a URL and run using `shiny::runURL("<the weblink>")`
- More info about hosting via a URL or just posting files here (https://shiny.rstudio.com/tutorial/written-tutorial/lesson7/)

Useful Things (Static Code)

Code can be placed prior to shinyServer

```
## Code here that you only need to evaluate once.
## This can include reading in data, creation of
##      functions common to all sessions, and
##      reading of other common r scripts.

shinyServer(function(input, output) {

## Code here that can be reactive.  Differs for
##      every instance of your app that runs.

})
```

Useful Things (Static Code)

Including Other Files

```
##  top of server.R, output from here is common to all users

#data set only read in once
dat <- read_csv("dataset.csv")

#function created and not modified
helper <- function(item1, item2) {item1 + item2}

shinyServer(function(input, output) {
  ## reactive things, instance of app dependent
})
```

Useful Things (Static Code)

Including Other Files

If you have a lot of code, you can read in a separate script

Useful Things (Static Code)

Including Other Files

If you have a lot of code, you can read in a separate script

- If external script is `helpers.R` in same folder as app:

```
##  top of server.R
source("helpers.R")

shinyServer(function(input, output) {
  ##  do stuff
})
```

Improve Efficiency

- Can use `isolate()` to improve code efficiency

```
renderPlot({
         input$submit  # Do take a dependency on input$submit
         hist(data, breaks = isolate(input$slider)) #don't depend on slider
})
```

Missing data or object

- Used to need to check if data/object existed and return `NULL` from your `render*` function
- Now can use `req()` to check (see this page (https://shiny.rstudio.com/articles/req.html) for more info)

```
data <- reactive({
  req("input$dist")
  if(input$dist == "norm"){
    dataVals <- rnorm(100)
  } else if (input$dist == "unif"){
    dataVals <- runif(100)
  }
})

output$plot <- renderPlot({
  hist(data())
})
```

Other Useful Things

- Create "dashboards" with `shinydashboard` or `flexdashboard` packages
- Use mouse over and click inputs
- Click Input Example (http://shiny.stat.ncsu.edu/jbpost2/BasketballCharting)
- Include Shiny in your Markdown slides!
- Use ioslides and add `runtime: shiny`

Debugging

- Much harder in shiny!
- Shiny debugging page (https://shiny.rstudio.com/articles/debugging.html)
- Recommendation: Get static working code, then transfer to shiny

Debugging

- Can use `observe({print(...)})`

```
shinyServer(function(input,output){

  #would now print to console
  observe({print(input$NI + 10)})

}
```

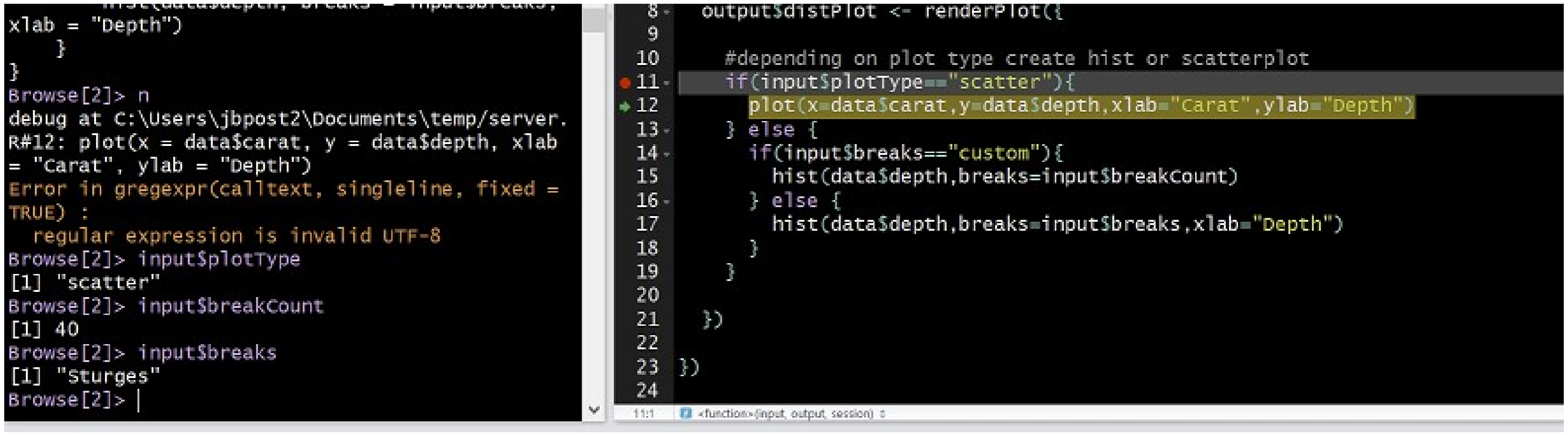
Debugging

Three major approaches:

1. Breakpoints - Pausing execution of your program
2. Tracing - Collecting information as your program runs
3. Error handling - Finding the source of errors (both on the client and server side) and ascertaining their cause.

Breakpoints

- Can be used in `server.r`
- Click to the left of the line number



```
xlab = "Depth")
    }
}
Browse[2]> n
debug at C:\Users\jbpost2\Documents\temp/server.
R#12: plot(x = data$carat, y = data$depth, xlab
= "Carat", ylab = "Depth")
Error in gregexpr(calltext, singleline, fixed =
TRUE) :
  regular expression is invalid UTF-8
Browse[2]> input$plotType
[1] "scatter"
Browse[2]> input$breakCount
[1] 40
Browse[2]> input$breaks
[1] "Sturges"
Browse[2]> |
```

```
 8    output$distPlot <- renderPlot({
 9
10      #depending on plot type create hist or scatterplot
11      if(input$plotType=="scatter"){
12        plot(x=data$carat,y=data$depth,xlab="Carat",ylab="Depth")
13      } else {
14        if(input$breaks=="custom"){
15          hist(data$depth,breaks=input$breakCount)
16        } else {
17          hist(data$depth,breaks=input$breaks,xlab="Depth")
18        }
19      }
20
21    })
22
23  })
24
```

- Now can access values and step through program
- Can also use browser()

Enter Debug Mode on Error

- Can make Shiny enter the debugger when an error occurs by using the following statement:

```
options(shiny.error = browser)
```

- Overall, experience helps!

Recap

- Shiny a great way to share results
- `ui.R` and `server.R` files or one file using `app.R`
- Reactive contexts important
- Can make UI dynamic
- Deploy in multiple ways
- Lots of add-ons/packages to make things nicer
- Debugging can be tricky, best to make static code work first!