Shiny App

LISA BERER, JONAS KAISER, ANNIKA STRATTON

Overview

Shiny

Component 1: User Interface

- Input and output functions

Component 2: Server

- Render functions and reactivity

Layout

Conclusion

What is Shiny?

R package from RStudio

can be used to build interactive web applications with R

can be shared with anyone

no R needed to view

Shiny app Template

- /load 1 example

```
library(shiny)

# Define UI ----
ui <- fluidPage(

)

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

}

# Run the app ----
shinyApp(ui = ui, server = server)
```

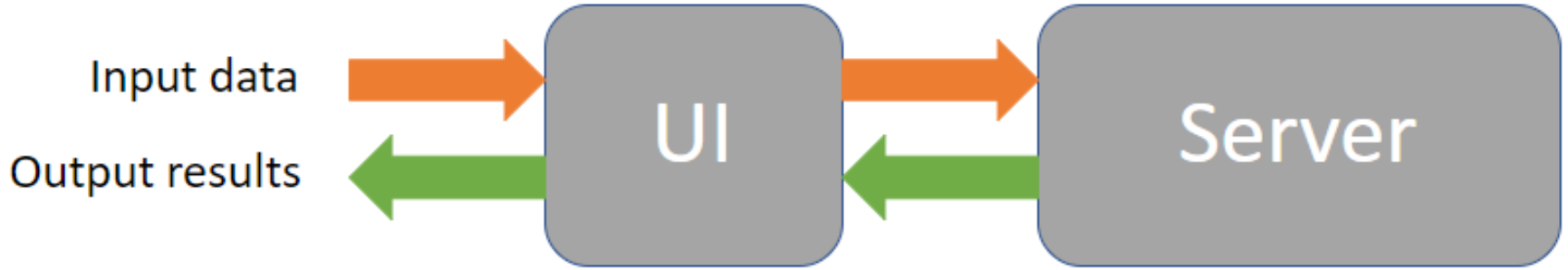
Behind the scenes

The web page (UI) of the shiny app is connected to a computer running a live R session (server)

Users can interact with the UI, which will cause the server to update the UI's display (by running R code)

UI is the frontend that accepts the user's input data

Server is the backend that processes the input data and produces output results that are displayed on the web page



Examples and useful add-on packages

Shiny - Gallery (rstudio.com)

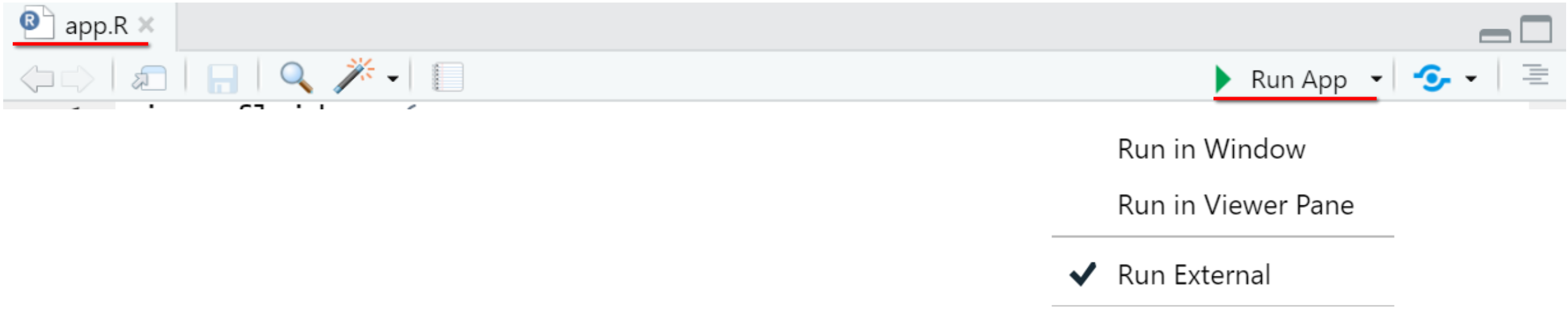
Several add-on packages to enhance shiny:

- shinyjs: Easily improve the user interaction and user experience in your Shiny apps in seconds
- shinythemes: Easily alter the appearance of your app
- leaflet: Add interactive maps to your apps
- ggvis: Similar to ggplot2, but the plots are focused on being web-based and are more interactive
- shinydashboard: Gives you tools to create visual "dashboards"

Save your app

Save your file in one folder as a

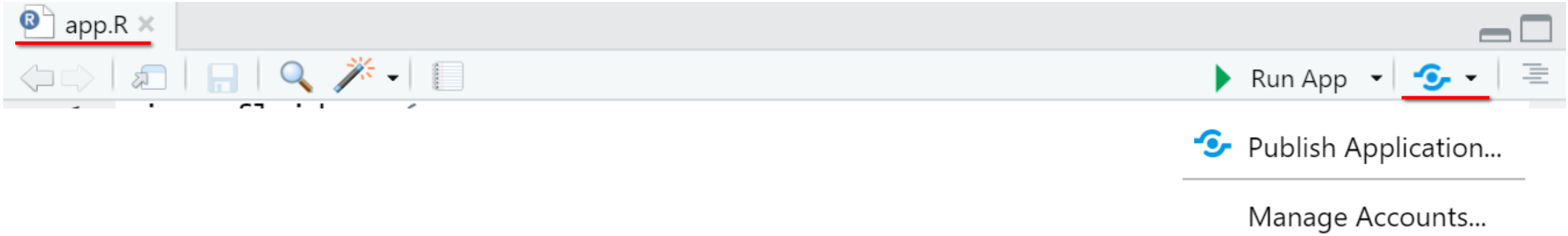
- single file with "app.R"
- multiple file with "ui.R/server.R"



Share your app

Share your app with anyone using

- Shinyapps.io (shinyapps.io) - server maintained by RStudio
- Shiny server (Shiny Server - RStudio) - build your own linux web server, other operating systems are currently not supported as server platforms



Component 1: User Interface

Defines how the app looks

The UI of a Shiny app is a web document

Displays what is shown on the app (inputs, outputs, layout,…)

The UI collects input values from the user and displays the result as an output

The UI calls R functions that output HTML code

Shiny turns this code into a web app

*Input() and *Output()

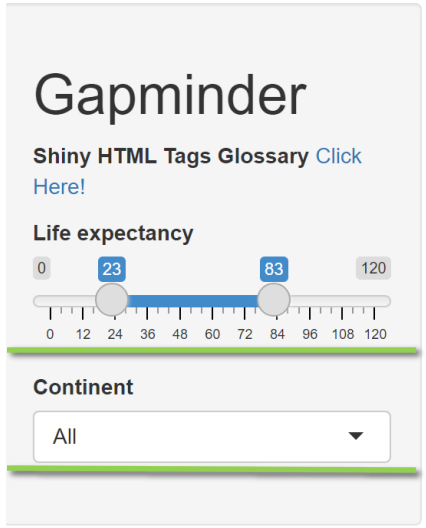
Shiny web app needs input values to generate an output

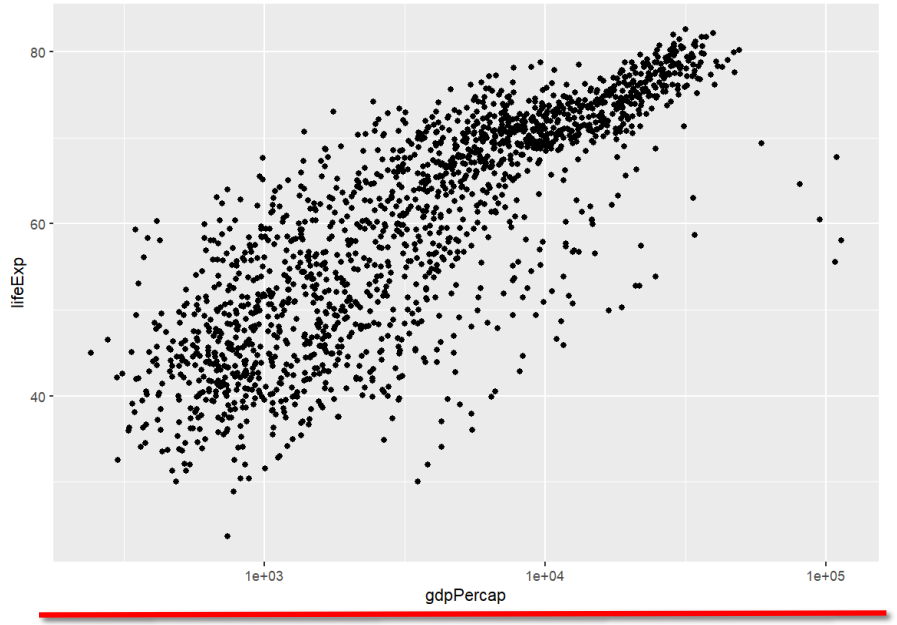
Inputs: user can interact on the UI

Outputs: objects which get displayed on the UI

ui

```
fluidPage(
    # *Input() functions,
    # *Output() functions
)
```



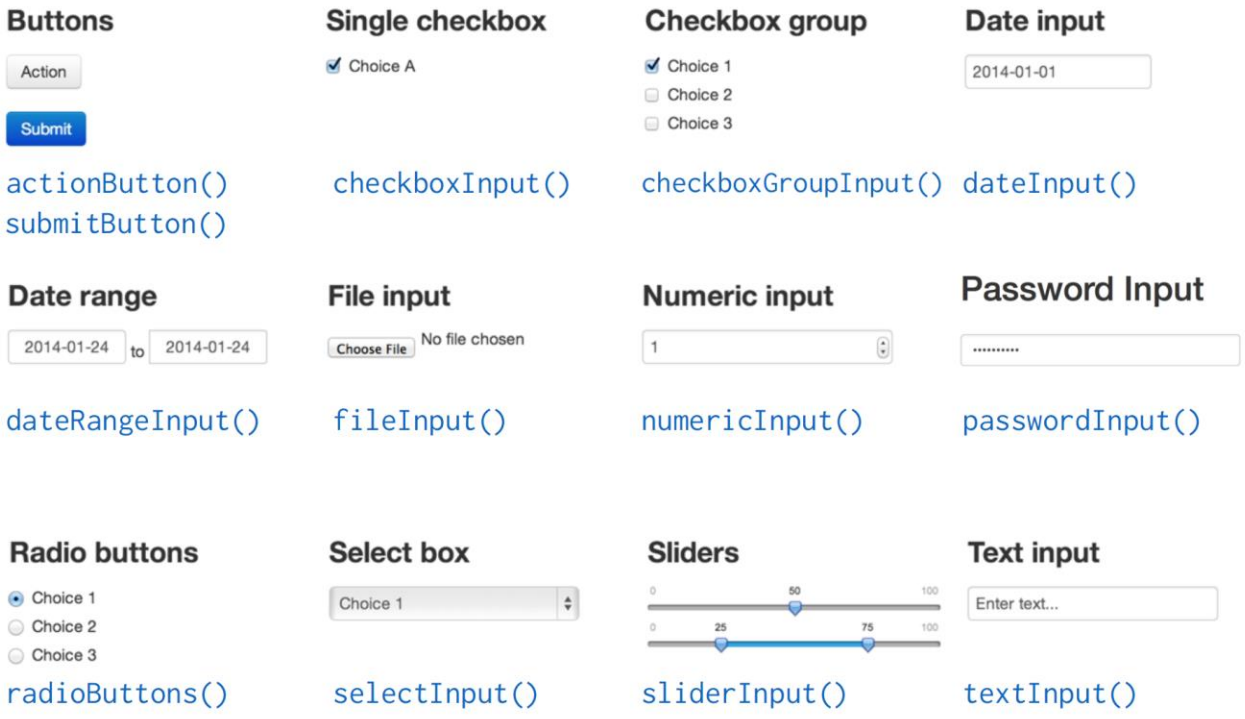


Syntax for *Input( )

Other than button inputs, replace * with type of input

In general, at least two required arguments:

- inputId
- label



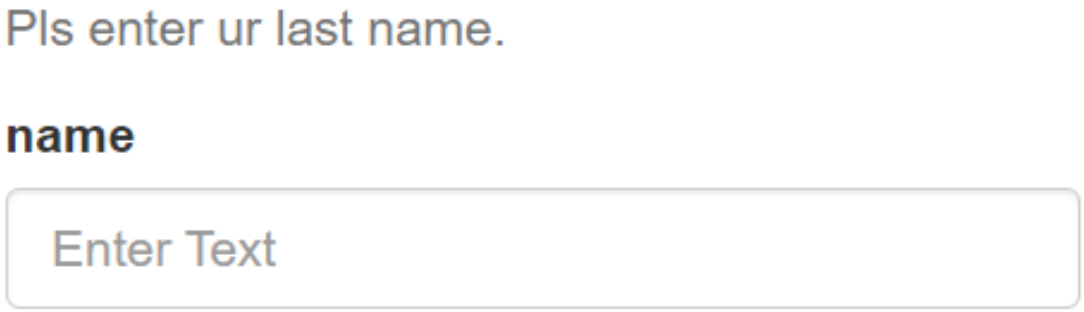
textInput()

- /load 2 example

Creates an input control entry for a text value

helpText() can be useful to further describe the *Input()

```
library(shiny)

ui <- fluidPage(
    helpText("Pls enter ur last name."),
    textInput(inputId = "txt",label = "name", placeholder = "Enter Text")
)

server <- function(input, output) {

}

shinyApp(ui, server)
```

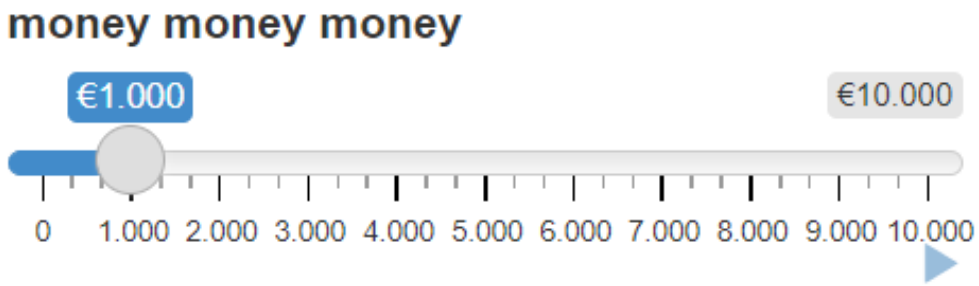


sliderInput()

•/load 3 example

Constructs a slider widget to select a number, date, or date-time from a range

```
library(shiny)

ui <- fluidPage(
sliderInput(inputId = "sliderId", label = "money money money",
            min = 0, max = 10000,
            value = 1000, step = 500,
            pre = "€", sep = ".",
            animate = animationOptions(interval = 500, loop = TRUE))
)

server <- function(input, output) {

}

shinyApp(ui, server)
```



submitButton()/actionButton()

•/load 4 example

A submit button isolates the input functions, so only by clicking the button the output values will be updated and displayed.

Font Awesome

An action button has an initial value, and the value gets updated every time it gets pushed. With the eventReactive() function it can be used as a submit Button as well.

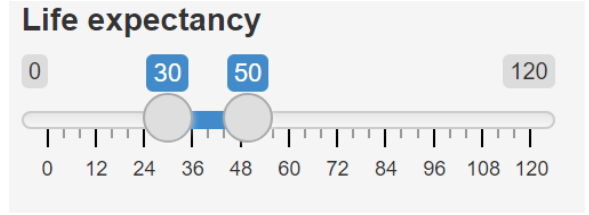
Users + People Icons | Font Awesome

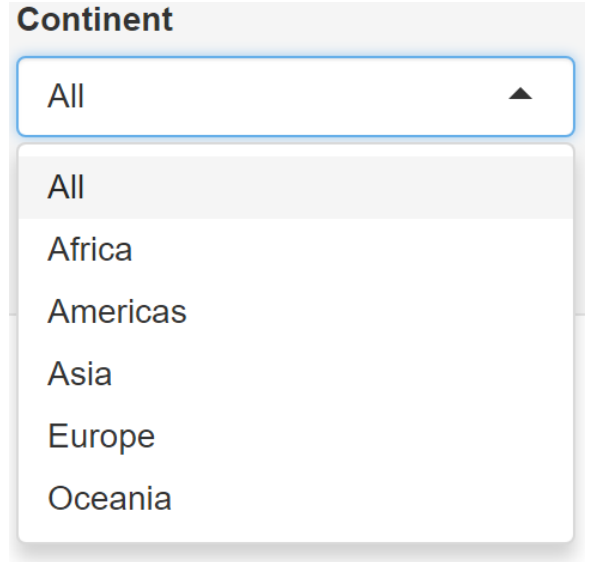
Bootstrap Buttons (w3schools.com)

Exercise 1

Please load the "gapminder" package for this exercise. You can get further information about gapminder through the following link: gapminder package - RDocumentation

1. build a web application with the following input functions:
   - A slider which has two buttons to specify the range. The inputId should be "life".
   - A select box. Use the different continents of the gapminder dataset for the levels and in addition to that an option to select all. The inputId should be "continent".
   - A submit button if you think it is useful for this application. In that case explain why so.





Syntax for *Output( )

- /load 5 example

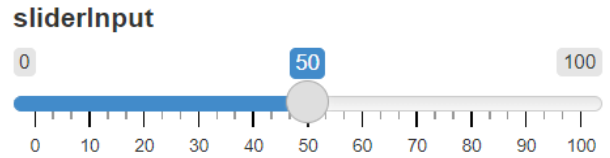
Replace * with type of output

One required argument: outputId

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

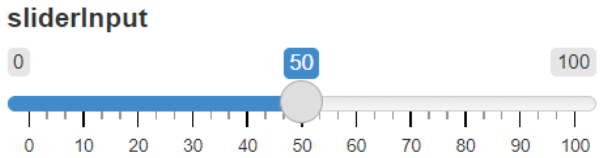
Why can't we see the plot?

- /load 6 example




```
ui <- fluidPage(
  sliderInput(inputId = "myslider", label = "sliderInput",
              min = 0, max = 100, value = 50),

  # creating placeholder for plot
  plotOutput(outputId = "myPlot"),

  # notice the gap between the slider and text box
  textInput(inputId = "mytext", label = "textInput"),
)

server <- function(input, output) {

}

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

Why can't we see the plot?




```
ui <- fluidPage(
  sliderInput(inputId = "myslider", label = "sliderInput",
              min = 0, max = 100, value = 50),

  # creating placeholder for plot
  plotOutput(outputId = "myPlot"),

  # notice the gap between the slider and text box
  textInput(inputId = "mytext", label = "textInput"),
)

server <- function(input, output) {

}

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

We will need to learn how to build the plot in the server function.

Component 2: Server

Defines how the app operates

R code that runs and updates the app

Uses reactive programming context for interactivity

- render*( ) functions for creating/updating output
- And other reactive context to customize reactions

render*()

Defines the code to build the output objects

Works with *Output() functions to add R output to the UI

*Output( ) (in ui)	render*( ) (in server)	Creates:
dataTableOutput( )	renderDataTable({ })	an interactive table
tableOutput( )	renderTable({ })	a table
textOutput( )	renderText({ })	a character string
verbatimTextOutput( )	renderPrint({ })	code block of printed output
plotOutput( )	renderPlot({ })	a plot
imageOutput( )	renderImage({ })	an image
uiOutput( )	renderUI({ })	a Shiny UI element

render*( ) and *Output( ) Syntax

ui:

*Output(outputId = **"uniqueName"**)

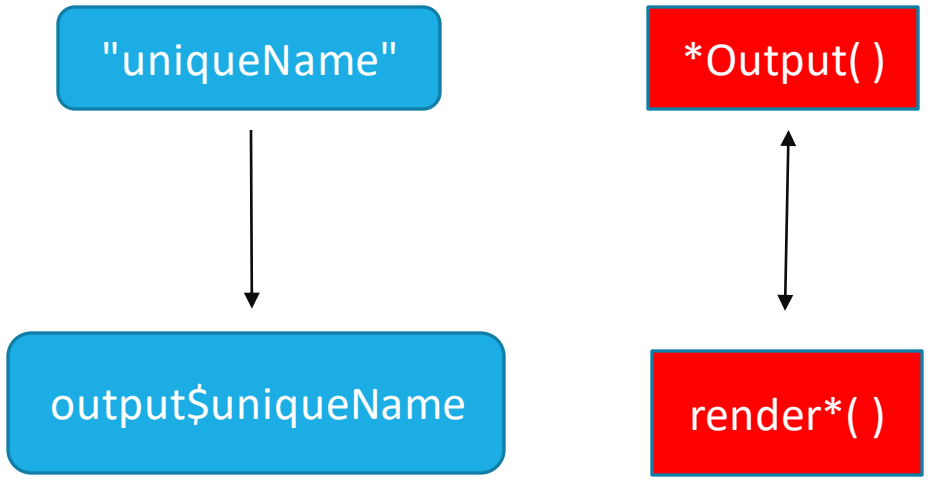
render*( ) and *Output( ) Syntax

ui:

```
*Output(outputId = "uniqueName")
```

server:

```
  output$uniqueName <- render*({
    # build output with R code here
  })
```



•/load 7 example

Tables

*Output( )	render*( )	Creates:
tableOutput( )	renderTable({ })	a table
dataTableOutput( )	renderDataTable({ })	an interactive table

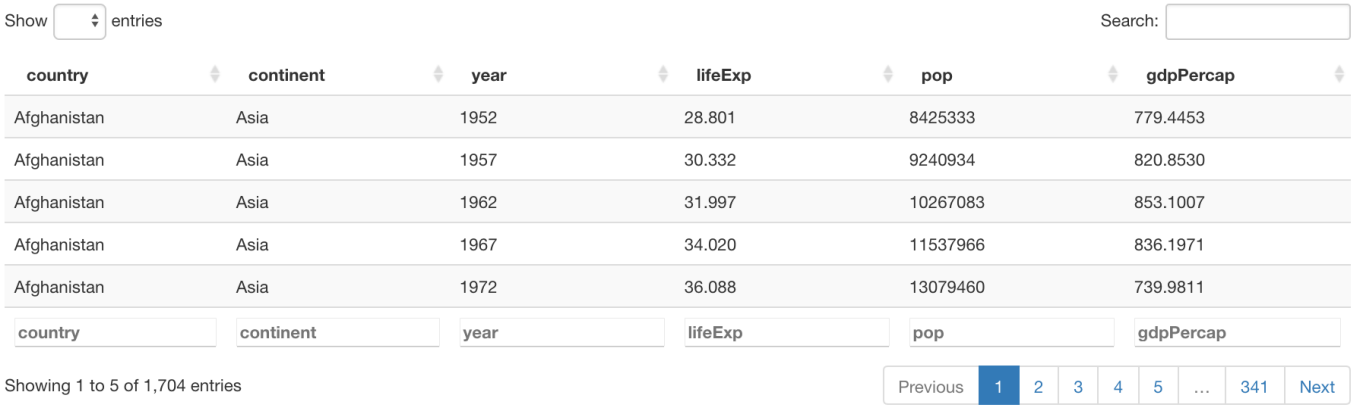
Table: HTML tables

- Best for small tables

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.80	8425333	779.45
Afghanistan	Asia	1957	30.33	9240934	820.85
Afghanistan	Asia	1962	32.00	10267083	853.10
Afghanistan	Asia	1967	34.02	11537966	836.20
Afghanistan	Asia	1972	36.09	13079460	739.98
Afghanistan	Asia	1977	38.44	14880372	786.11

Data table: to create interactive tables

- Best for showing entire data set



•/load 8 example

Texts

*Output( )	render*( )	Creates:
textOutput( )	renderText( )	a character string
verbatimTextOutput( )	renderPrint( )	code block of printed output

Two types:

- **Text**: normal text

Hello friend!

- **VerbatimText**: creates R code text

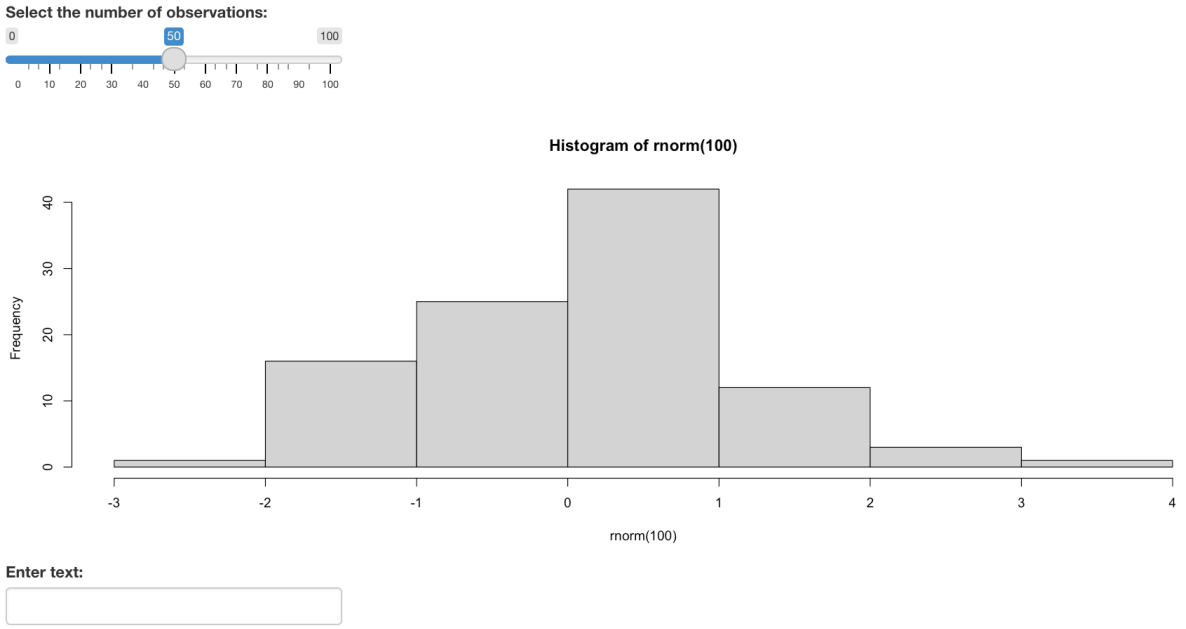
```
[1] "Happy Monday!"
```

```
ui <- fluidPage(
  textOutput(outputId ="text"),
  verbatimTextOutput(outputId = "code")
)

server <- function(input, output) {

   # regular text
  output$text <- renderText({
    "Hello friend!"
  })

  # code-like text
  output$code <- renderPrint({
    "Happy Monday!"
  })
}
shinyApp(ui = ui, server = server)
```

Finish creating the plot

- /load 9 example

```
ui <- fluidPage(

  sliderInput(inputId = "mySlider",
              label = "Select the number of observations:",
              min = 0, max = 100, value = 50),

  # creating placeholder for plot
  plotOutput(outputId = "myPlot"),

  # notice the gap between the slider and text box
  textInput(inputId = "myText", label = "Enter text:")
)

server <- function(input, output) {

  # plot output (fill in the blanks!)----
  ______ <- render_____({
    hist(rnorm(100))
  })
```



Reactive Outputs

Assemble input value in output objects for interactive outputs

Call input value in render*( ) by: **input$<inputId>**

input$<inputId> values are called **reactive values**

- Reactive values can ONLY be called from reactive context

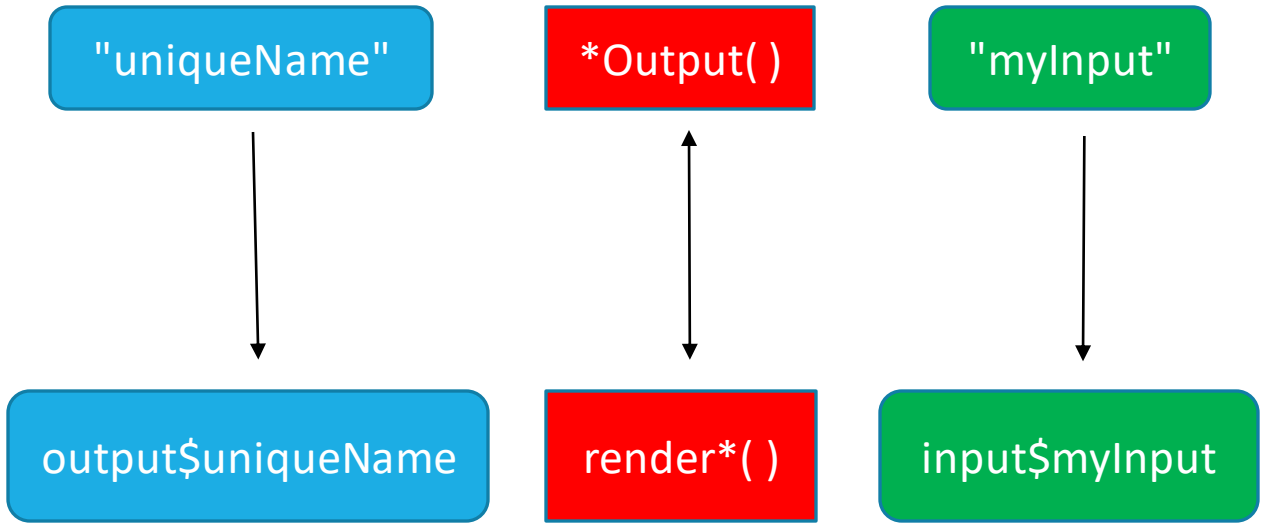
Build Reactive Object Syntax

ui:

```
*Input(inputId = "myInput")
*Output(outputId = "uniqueName")
```

server:

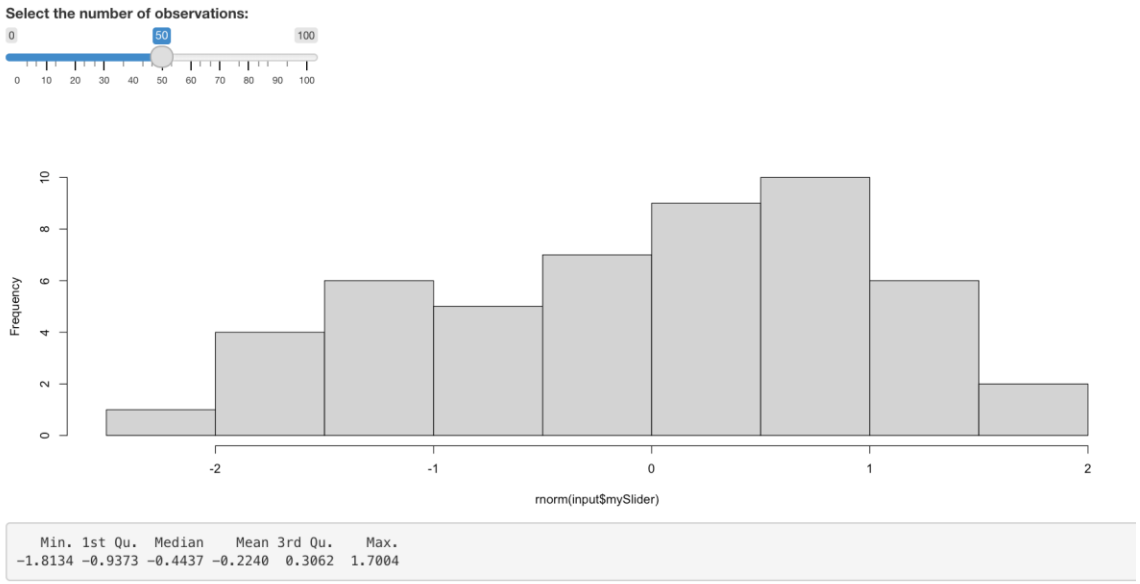
```
  output$uniqueName <- render*({
    R code(input$myInput)
  })
```



Exercise 2

1. Make the plot reactive: make the slider input value determine the number of observations

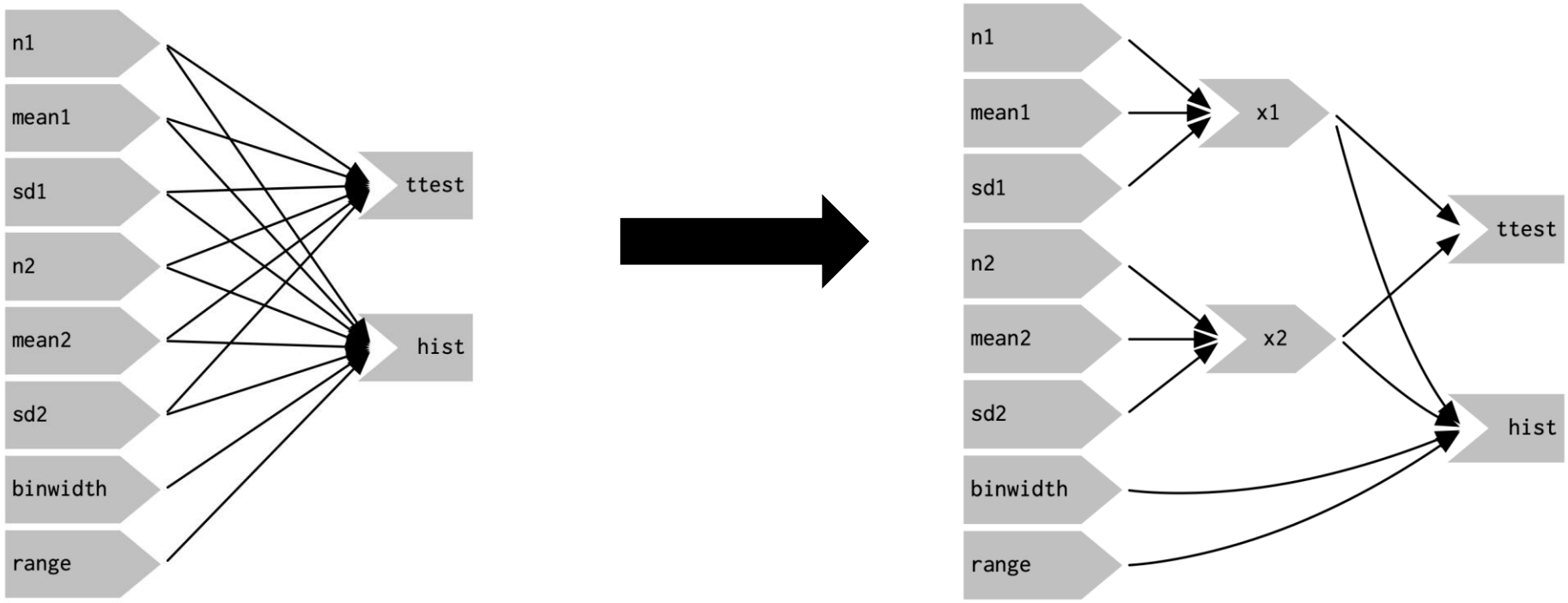
2. Plot summary stats: Replace the text box with the correct type of text output function to display its statistics based on the current value of the slider. Name the outputId: "mySummary".



Reactive Context and Dependency Trees

Reactive context creates and customizes reactions

Easier to understand input and output relationship with dependency tree

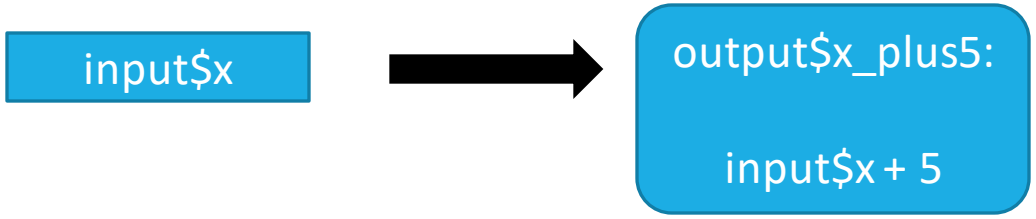


render*( )

•/load 10 example

Output automatically updates whenever the reactive value in block of code changes

```
ui <- fluidPage(
  numericInput(inputId = "x", label = "Choose a number:",
               value = 1),

  textOutput(outputId = "x_plus5")

)

server <- function(input, output) {
  output$x_plus5 <- renderText({
     input$x + 5

  })

}

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



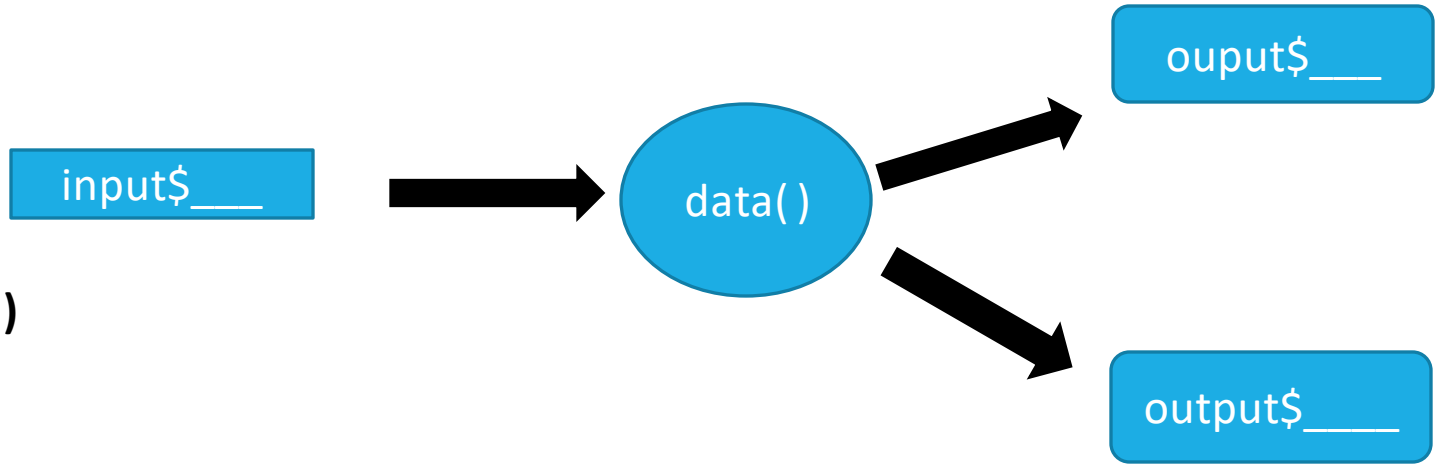
reactive( )

•/load 11 example

Used to reduce duplication of code

Is a reactive expression

- Its value is called as a FUNCTION
  - Named data( ) in example



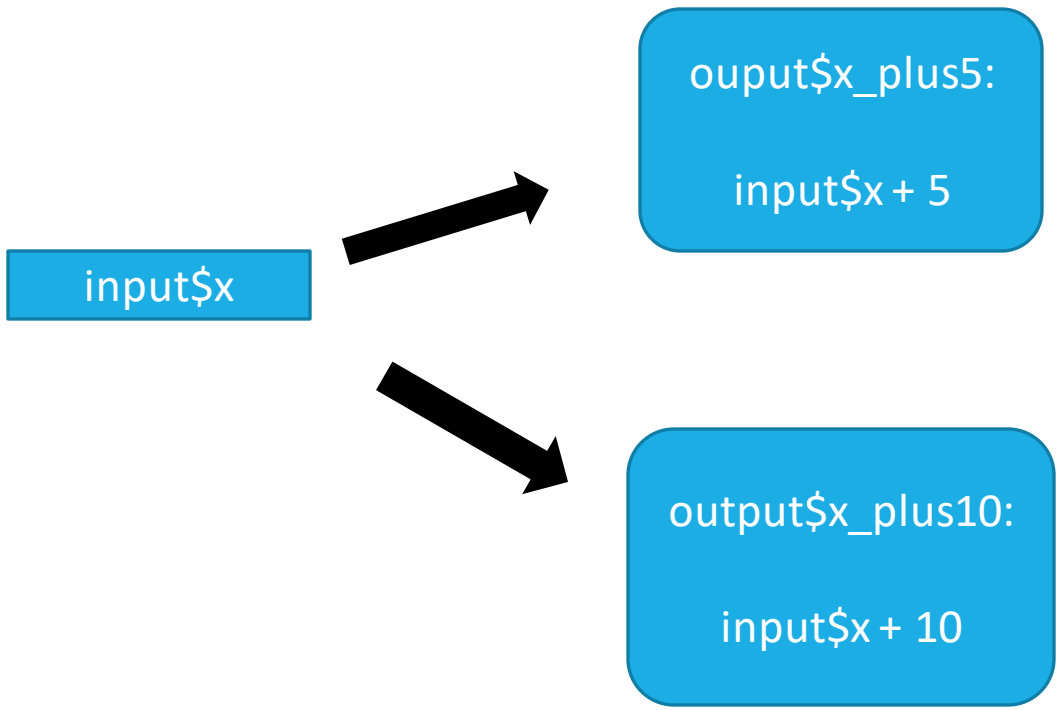

```
data <- reactive( { reactive value } )

output$name <- renderText({
    data( )
})
```

reactive( )

- /load 11 example

```
ui <- fluidPage(
  numericInput(inputId = "x", label = "Choose a number:",
               value = 1),

  textOutput(outputId = "x_plus5"),
  textOutput(outputId = "x_plus10")

)

server <- function(input, output) {
  output$x_plus5 <- renderText({
    input$x + 5

  })

  output$x_plus10 <- renderText({
    input$x + 10
  })

}

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

Before adding reactive( ):

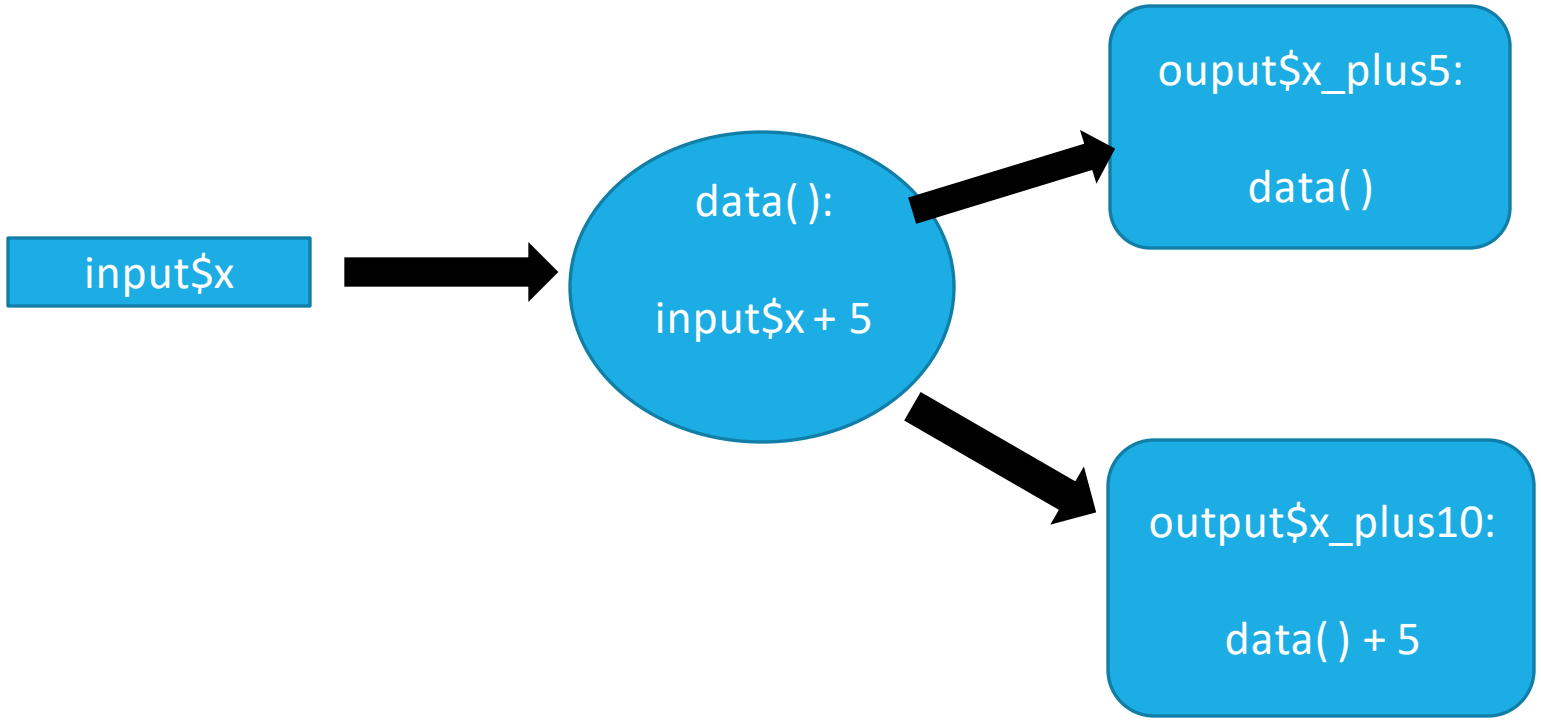


reactive( )

- /load 11 example

```
server <- function(input, output) {
 # create reactive() and call data() in output
  data <- reactive({
    input$x + 5
  })

   output$x_plus5 <- renderText({
    data()

  })

   output$x_plus10 <- renderText({
     data() + 5
   })

}

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

After adding reactive( ):



Exercise 3

Instructions:

1. Improve the code by adding reactive( ) to reduce code duplication.

2. What else does this fix?

Want more practice with reactive() expressions?

Draw the dependency trees for the following server functions:

```
server1 <- function(input, output, session) {
  c <- reactive(input$a + input$b)
  e <- reactive(c() + input$d)
  output$f <- renderText(e())
}
server2 <- function(input, output, session) {
  x <- reactive(input$x1 + input$x2 + input$x3)
  y <- reactive(input$y1 + input$y2)
  output$z <- renderText(x() / y())
}
server3 <- function(input, output, session) {
  d <- reactive(c() ^ input$d)
  a <- reactive(input$a * 10)
  c <- reactive(b() / input$c)
  b <- reactive(a() + input$b)
}
```

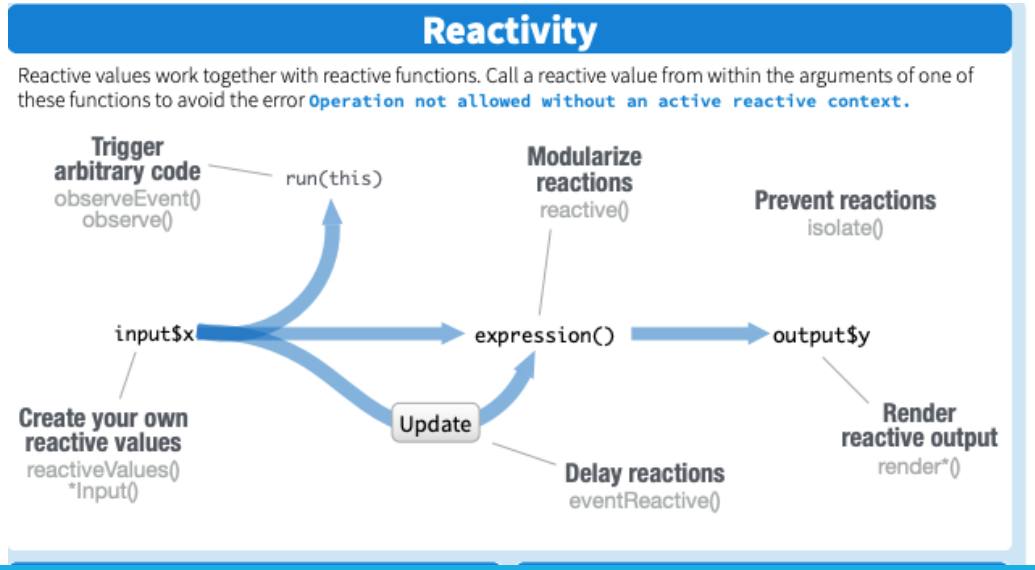
Recap: Reactivity

To create reactive objects:

- Save output objects as **output$<outputId>**
- Build the output by assigning it to a **render*( )** function
- Access input values with **input$<inputId>**

Can use dependency trees to help visualize reactivity

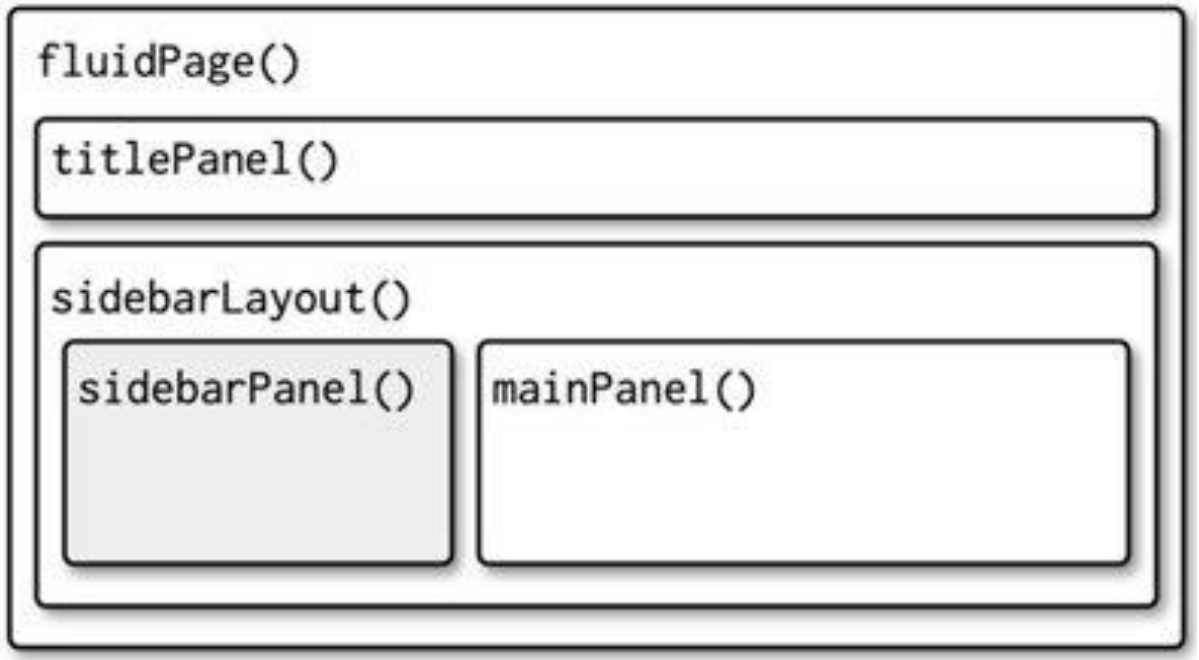
See our sources to learn more about reactive context



Layout

Layout Hierarchy

HTML hierarchy applies to ui when generating a layout




```
ui <- fluidPage(
  titlePanel(
    # app title/description
  ),
  sidebarLayout(
    sidebarPanel(
      # inputs
    ),
    mainPanel(
      # outputs
    )
  )
)
```

sidebarLayout()

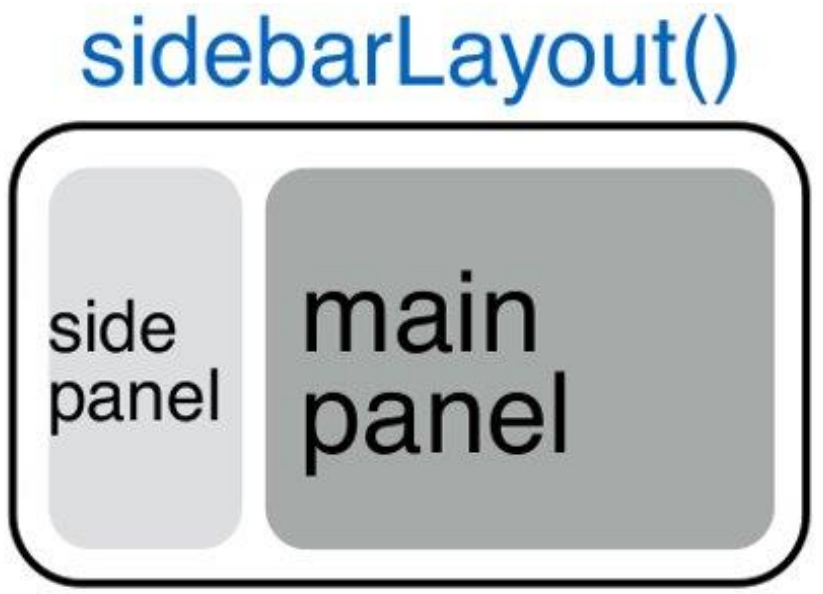
•/load 12 example

Creates a basic layout for the app

Provides a sidebar panel for inputs and a main area for outputs

Consists of two arguments:

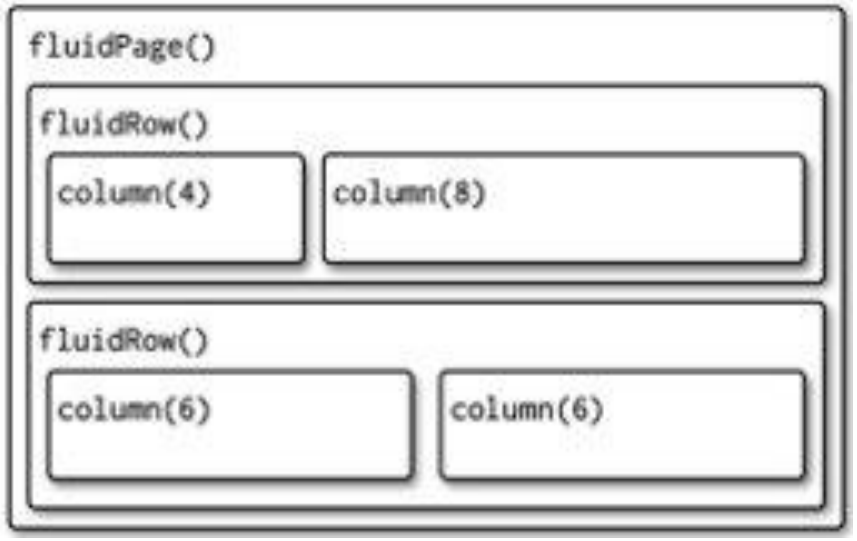
- sidebarPanel()
- mainPanel()



Grid Layout

The fluidRow() and column() functions are used to build a layout from a grid system.

Grid layouts can be used anywhere within fluidPage( ) and can also be nested in each other.




```
fluidPage(
  fluidRow(
    column(4,
      ...
    ),
    column(8,
      ...
    )
  ),
  fluidRow(
    column(6,
      ...
    ),
    column(6,
      ...
    )
  )
)
```

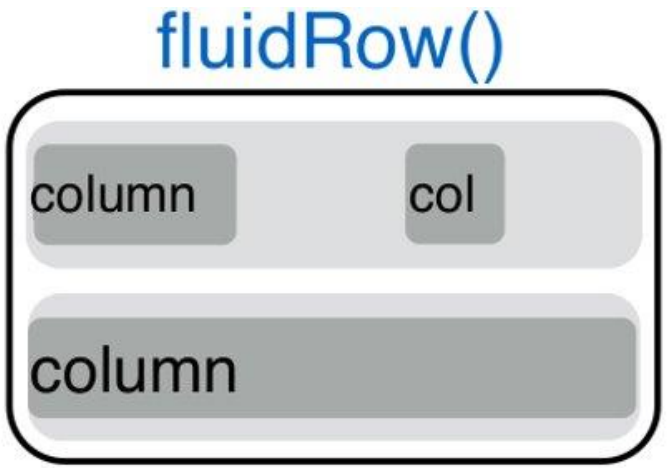
fluidRow()/column()

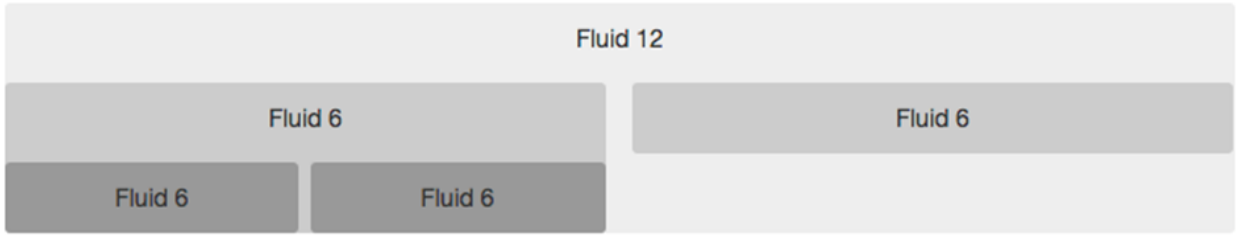
- /load 13 example

fluidRow() adds rows to the grid. Each new row goes below the previous rows.

Column is included in fluidRow() and creates a new column. By default, there are 12 Columns in a fluidRow() container.

The offset argument indicates the offset from the end of the previous column.





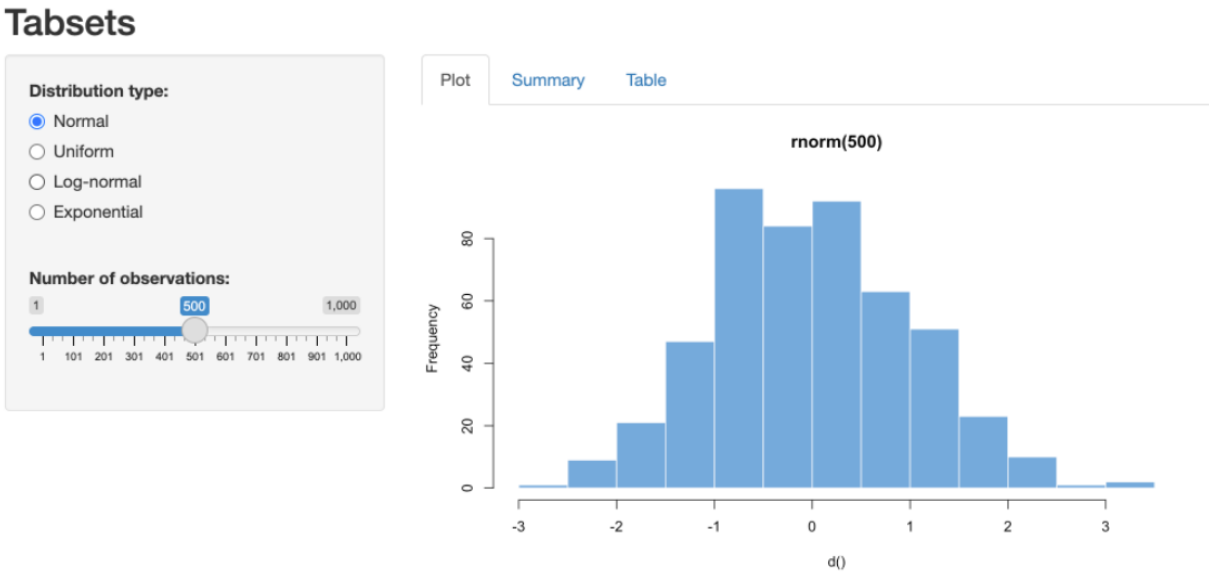
tabsetPanel()

•/load 14 example

tabsetPanel() is a container for tabPanels(). It can be used to divide an app into sections.

Tabs are located on top by default.

```
titlePanel("Tabsets"),

sidebarLayout(

  sidebarPanel(
    # Inputs excluded for brevity
  ),

  mainPanel(
    tabsetPanel(
      tabPanel("Plot", plotOutput("plot")),
      tabPanel("Summary", verbatimTextOutput("summary")),
      tabPanel("Table", tableOutput("table"))
    )
  )
)
```



navlistPanel()

- /load 15 example

navlistPanel() is a good alternative to tabsetPanel() if you have a bigger amount of tabPanels()

Components are listed in a sidebar instead of using tabs

It supports section heading and separators for longer lists

```
ui <- fluidPage(

  titlePanel("Application Title"),

  navlistPanel(
    "Header A",
    tabPanel("Component 1"),
    tabPanel("Component 2"),
    "Header B",
    tabPanel("Component 3"),
    tabPanel("Component 4"),
    "-----",
    tabPanel("Component 5")
  )
)
```

navbarPage()/navbarMenu()

navbarPage() creates a horizontal Menu.

navbarPage() replaces fluidPage() and requires a title

A drop-down menu can be added with navbarMenu()

```
ui <- navbarPage("My Application",
  tabPanel("Component 1"),
  tabPanel("Component 2"),
  navbarMenu("More",
    tabPanel("Sub-Component A"),
    tabPanel("Sub-Component B"))
)
```

Customizing UI with HTML

HTML elements can be used to customize an app

Shiny turns code into a web app. Each ui object calls R functions that return HTML.

You do not need to know HTML to use shiny

CSS and JavaScript can also be included to style the app and make it interactive

HTML Tags

The shiny::tags object contains
R functions that create HTML tags.

```
library(shiny)
shiny::tags
names(tags)
```

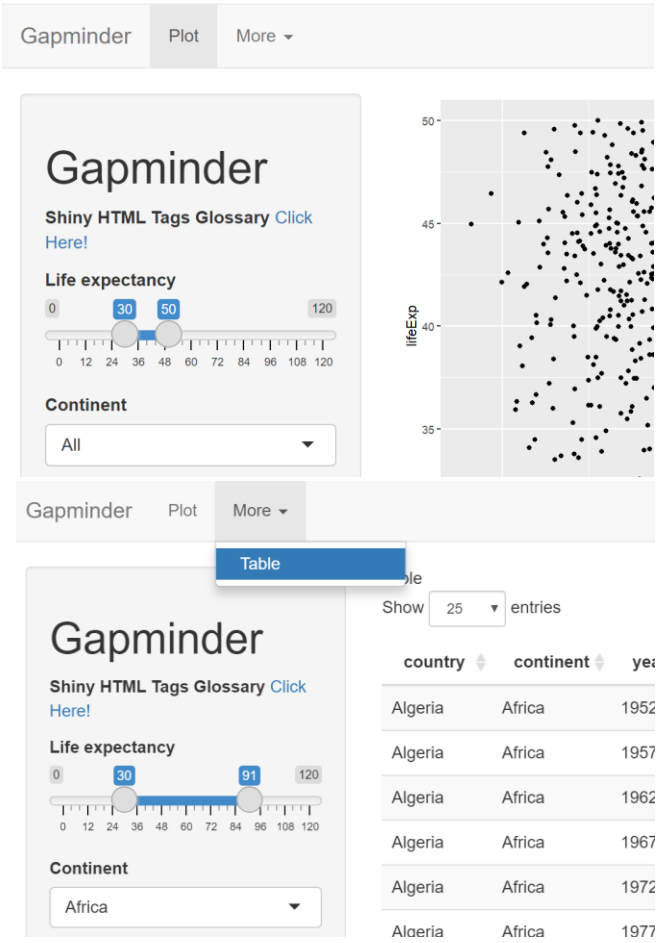
To create a tag, run
an element of tags as a function

tags$a	tags$data	tags$h6	tags$nav	tags$span
tags$abbr	tags$datalist	tags$head	tags$noscript	tags$strong
tags$address	tags$dd	tags$header	tags$object	tags$style
tags$area	tags$del	tags$hgroup	tags$ol	tags$sub
tags$article	tags$details	tags$hr	tags$optgroup	tags$summary
tags$aside	tags$dfn	tags$HTML	tags$option	tags$sup
tags$audio	tags$div	tags$i	tags$output	tags$table
tags$b	tags$dl	tags$iframe	tags$p	tags$tbody
tags$base	tags$dt	tags$img	tags$param	tags$td
tags$bdi	tags$em	tags$input	tags$pre	tags$textarea
tags$bdo	tags$embed	tags$ins	tags$progress	tags$tfoot
tags$blockquote	tags$eventsource	tags$kbd	tags$q	tags$th
tags$body	tags$fieldset	tags$keygen	tags$ruby	tags$thead
tags$br	tags$figcaption	tags$label	tags$rp	tags$time
tags$button	tags$figure	tags$legend	tags$rt	tags$title
tags$canvas	tags$footer	tags$li	tags$s	tags$tr
tags$caption	tags$form	tags$link	tags$samp	tags$track
tags$cite	tags$h1	tags$mark	tags$script	tags$u
tags$code	tags$h2	tags$map	tags$section	tags$ul
tags$col	tags$h3	tags$menu	tags$select	tags$var
tags$colgroup	tags$h4	tags$meta	tags$small	tags$video
		tags$meter	tags$source	tags$wbr

```
tags$a(href = "https://shiny.rstudio.com/articles/tag-glossary.html", "Click Here!"),

#Shiny HTML Tags glossary
```

Exercise 4

1. build a layout with navbarPage():
   - Create two tabs using tabPanel() and a drop-down menu using navbarMenu()
   - Place the content for each page in a sidebarLayout(). Place the input in the sidebarPanel() and the output in the mainPanel()
   - Please name the sliderInput in the second tab inputId="life1" and the selectInput inputId="content1"
   - Use at least 5 HTML-Tags tags from the glossary: https://shiny.rstudio.com/articles/tag-glossary.html
   - You can use example 5 and example 17



Conclusion

Shiny combines the computational power of R with the interactivity of the modern web

Shiny apps are easy to write

Huge userbase with plenty of examples

Very flexible

Enhance your app with a lot of add-on packages

No web development skills are required

Can share/view app on the internet

Easy to handle, hard to master

Limited number of free apps to publish via shinyapps.io

Sources

Shiny (rstudio.com)

https://mastering-shiny.org/index.html

(91) R Shiny for Data Science Tutorial – Build Interactive Data-Driven Web Apps - YouTube

Building Shiny apps - an interactive tutorial (deanattali.com)

Case Studies: Building Web Applications with Shiny in R Course | DataCamp

https://shiny.rstudio.com/articles/layout-guide.html

https://shiny.rstudio.com/articles/tag-glossary.html

https://mastering-shiny.org/action-layout.html