R Shiny Workshop

Christina Maimone

christina.maimone@northwestern.edu

Handouts

Workshop Information

Shiny cheat sheet

Get Ready

Files - see options on handout

Open RStudio (desktop or cloud)

Christina Maimone
Frank Elavsky

Northwestern University
Research Computing Services
http://www.it.northwestern.edu/research/

Introductions

Say hello to your neighbors

What is Shiny?

An R Package

An R Package

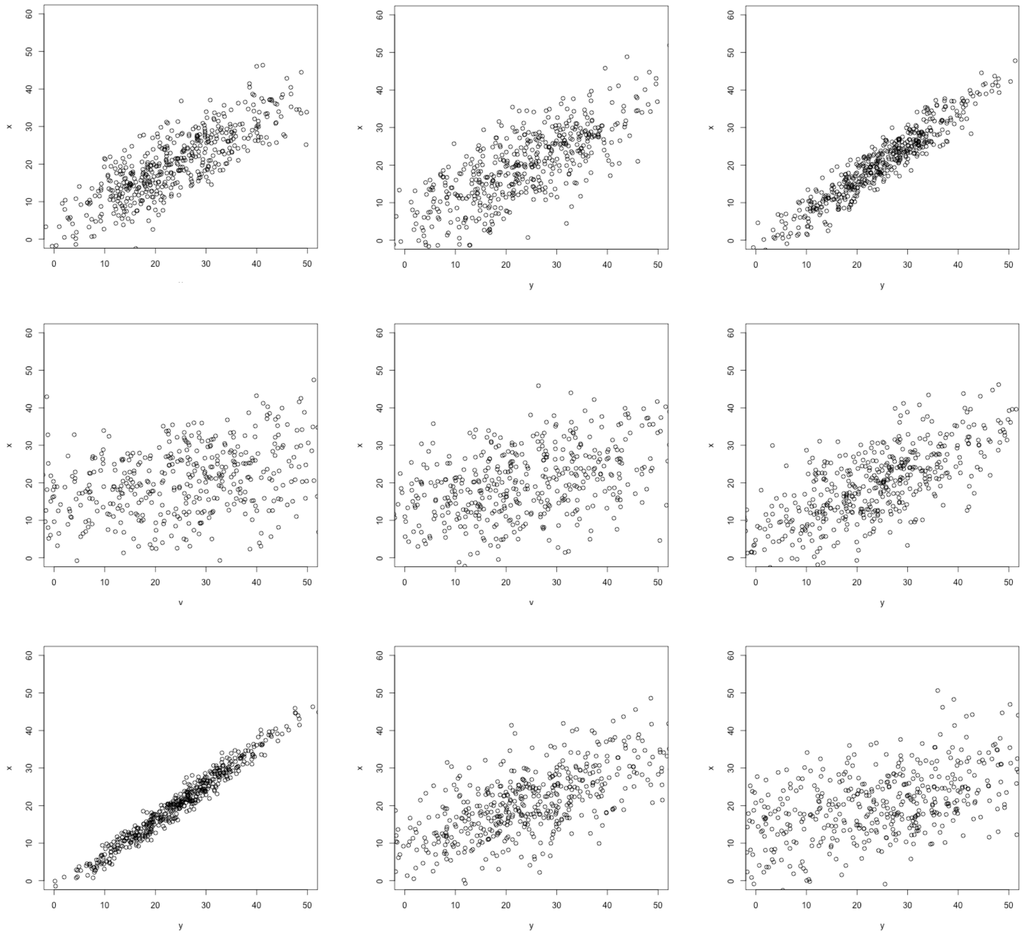
for creating interactive

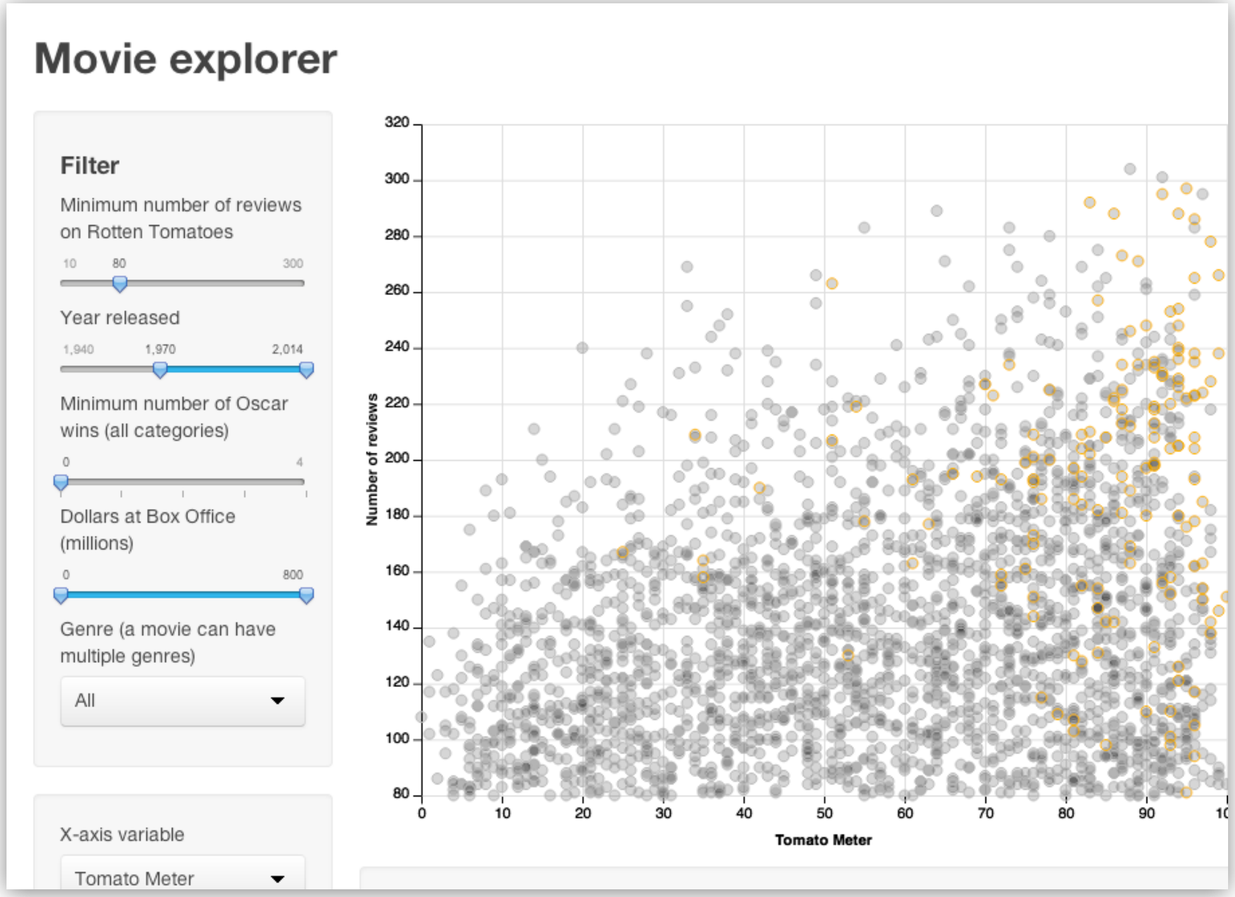
web applications

An R Package
for creating interactive
web applications
without needing to know
HTML, CSS, or JavaScript

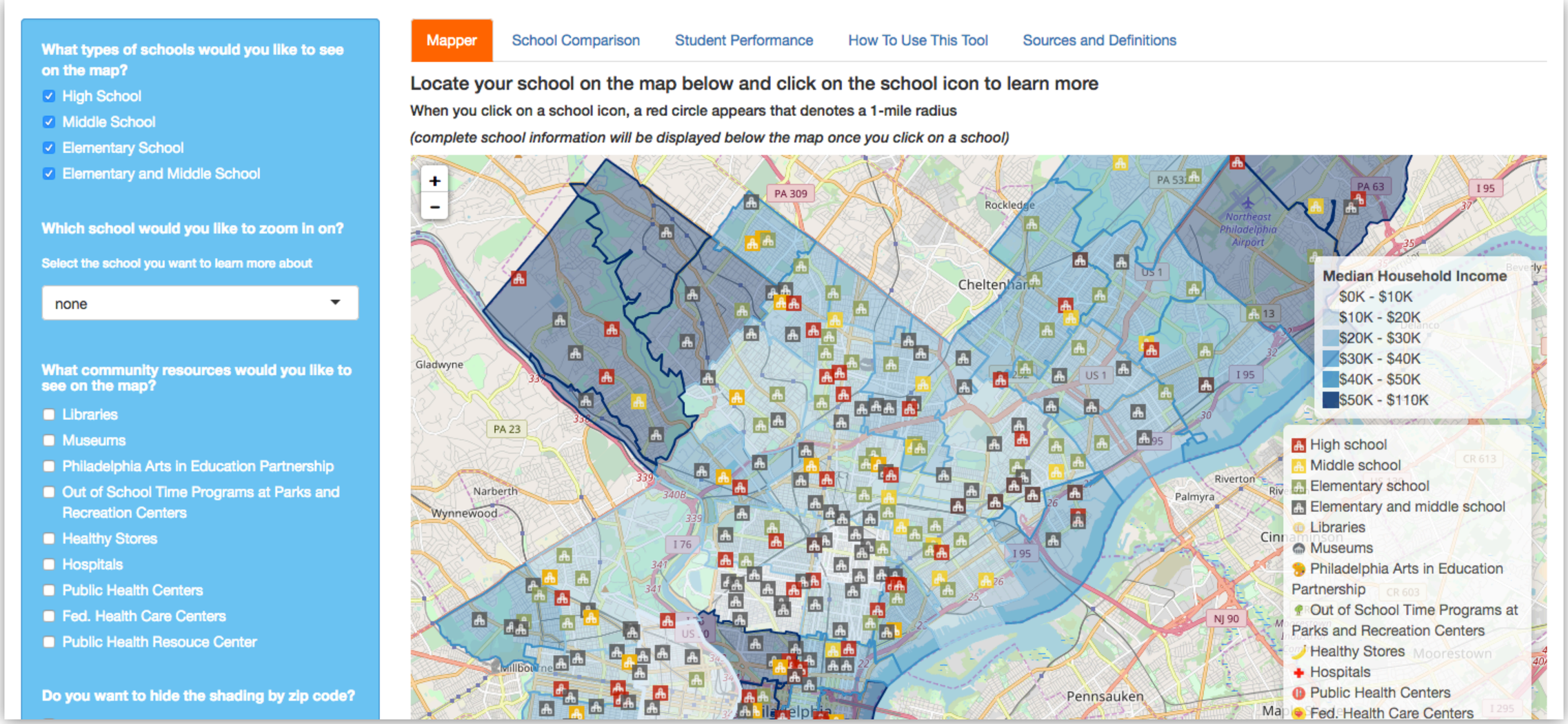
Why use Shiny?

Exploratory Data Analysis



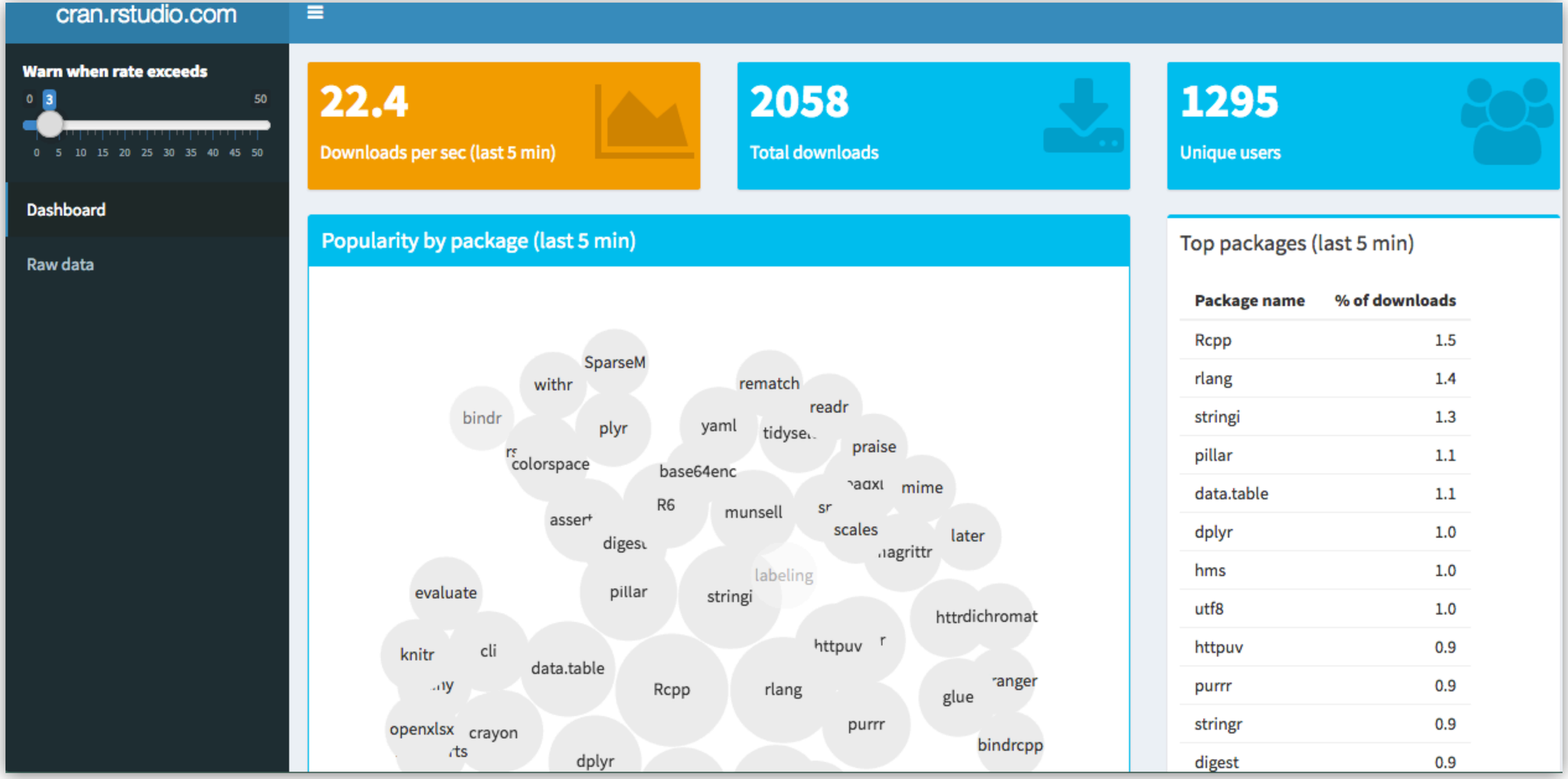


Contextual Learning and Exploration



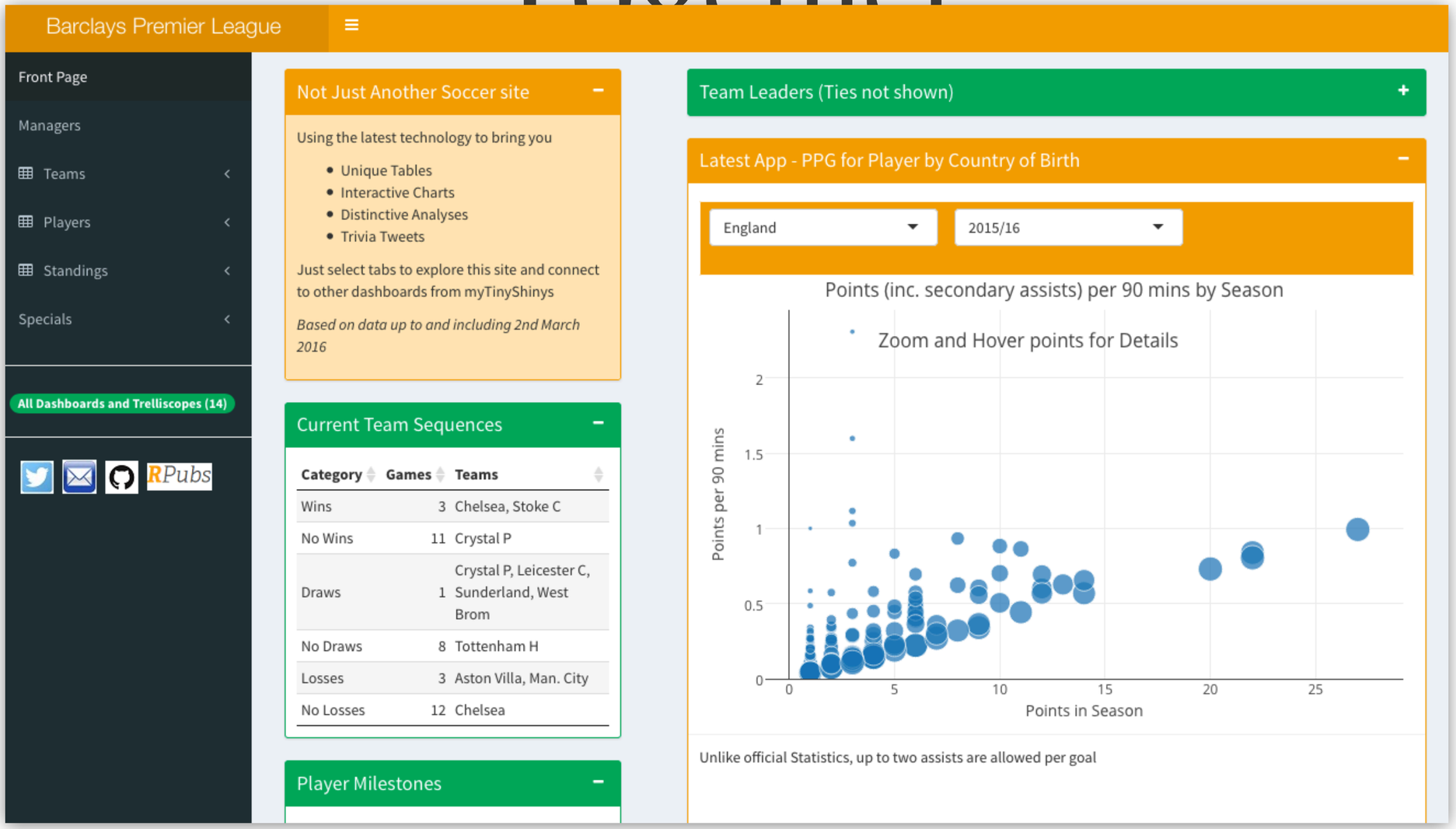
https://datasociety.com/kitamba-the-opportunity-project/

Dashboards



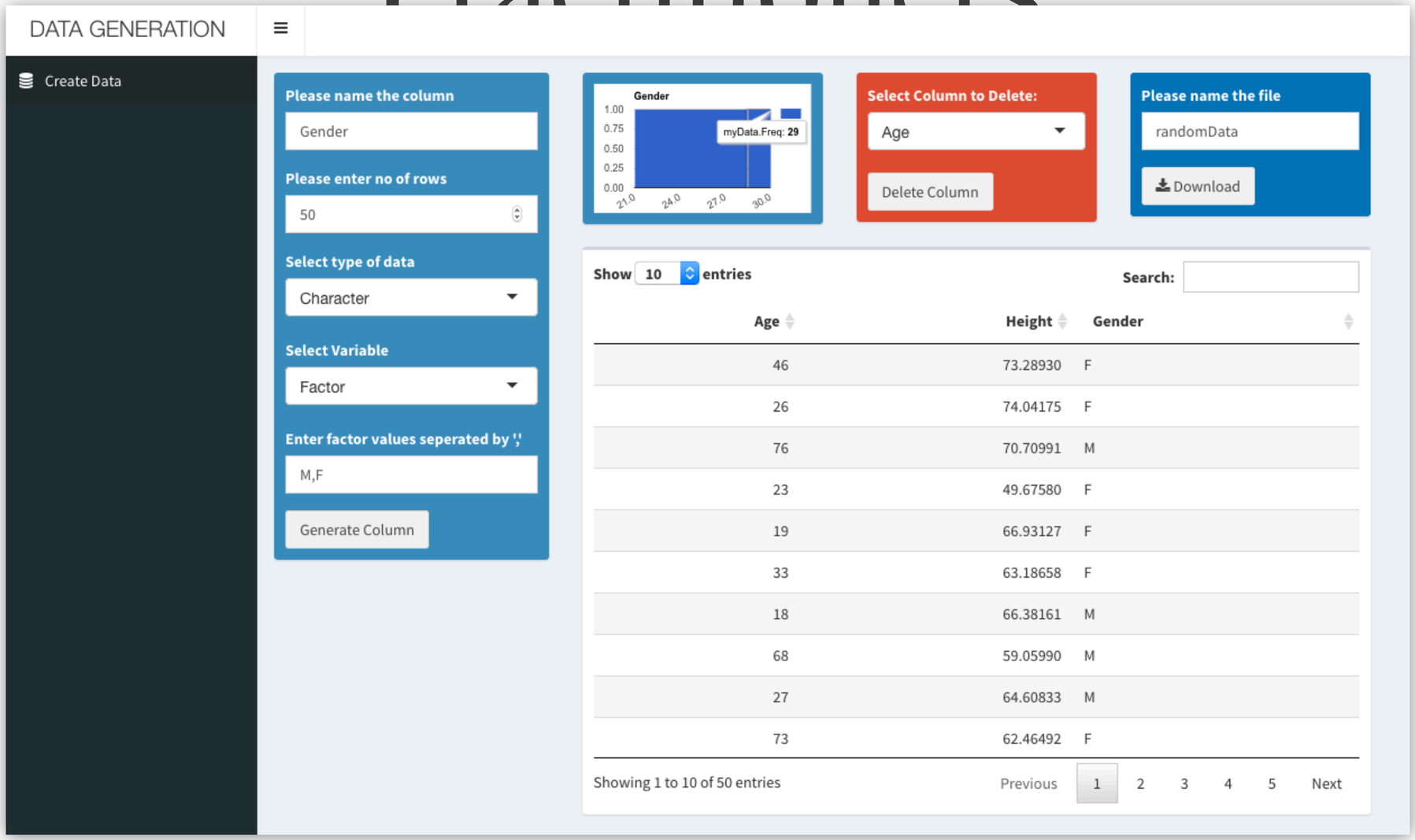
https://gallery.shinyapps.io/087-crandash/

Data, Text, and Visualizations Together



https://gallery.shinyapps.io/premierLeague/

Tools for Your Colleagues or Practitioners



https://ebsubudhi.shinyapps.io/DataGeneration/

Teaching

Influence Analysis

This simple Shiny app demonstrates the leverage and influence of an adjustable point that is part of a dataset with 101 points; 100 of which are normally distributed. You can select the X-Y coordinates of the adjustable point using the following sliders with ranges from [-10,10] for both X and Y. The adjustable point appears in red on the graph.

X

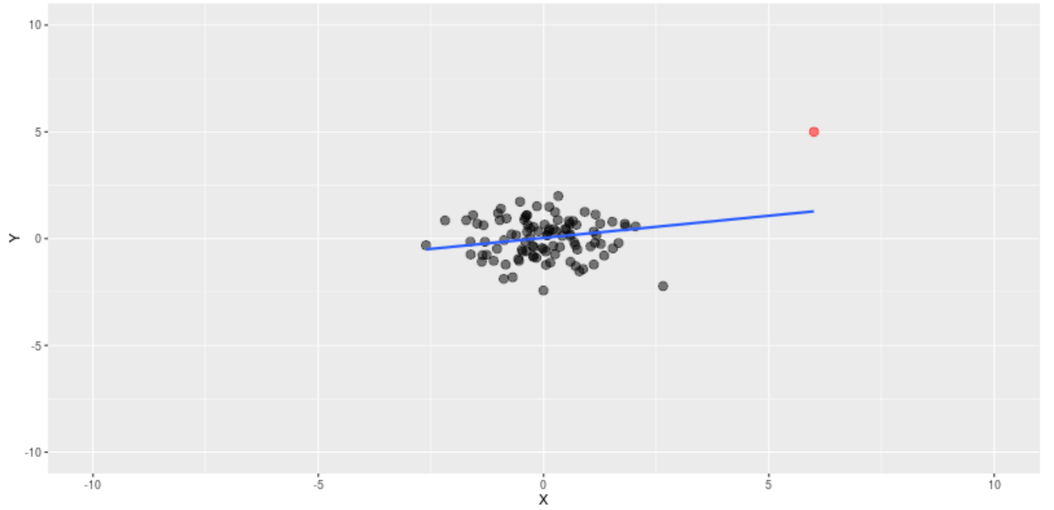
-10 6 10

-10 -8 -6 -4 -2 0 2 4 6 8 10

Y

-10 5 10

-10 -8 -6 -4 -2 0 2 4 6 8 10



Linear regression model coefficients

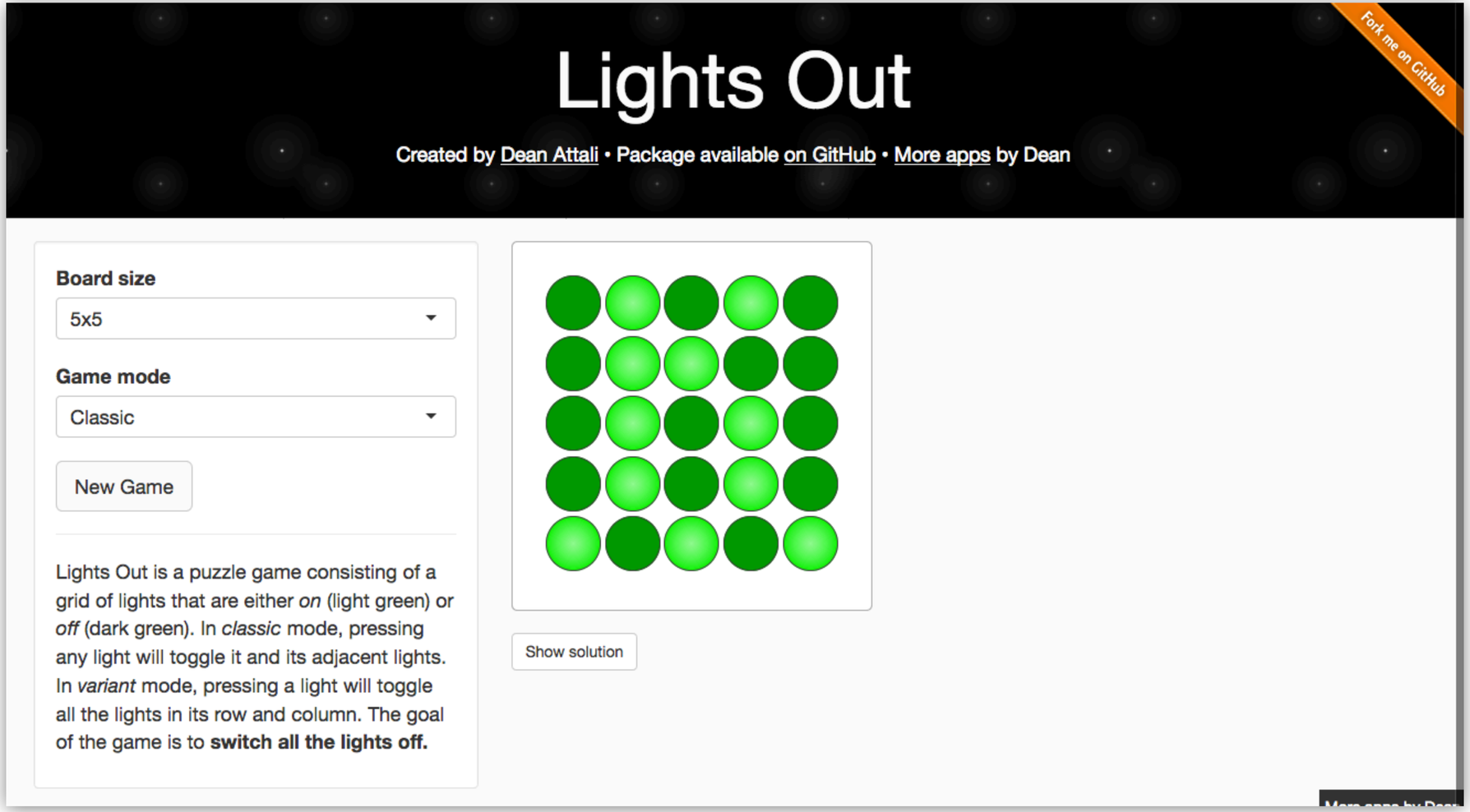
	Coefficient	Value
1	Intercept	0.03721
2	Slope	0.20699

Influence measures for the adjustable point

	Measure	Value
1	hatvalue	0.28682
2	residual	3.72084
3	dfbeta.1	0.41251
4	dfbeta.x	3.03286

https://omaymas.shinyapps.io/Influence_Analysis/

Fun



https://daattali.com/shiny/lightsout/

Demo

examples/120-goog-index/app.R

- Structure of Shiny Applications
- UI: Layout, Inputs and Outputs
- Server
- Bringing it Together
- Design Elements
- Interactive Graphics
- Sharing Your Shiny Application

Exercise 1A

- Create a new Shiny app
- File > New File > Shiny web app
- Application Type: Single file
- Name: exercise1
- Run App

BONUS

Change the x axis label and title on the plot

- `x="Minutes between Eruptions"`
- `main="Waiting Time Distribution"`

Shiny Application Structure

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

Shiny : : CHEAT SHEET

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

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

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE or run:
   rsconnect::deployApp("<path to directory>")

Build or purchase your own Shiny Server
at www.rstudio.com/products/shiny-server/

RStudio

Building an App

Complete the template by adding arguments to fluidPage() and a body to the server function.

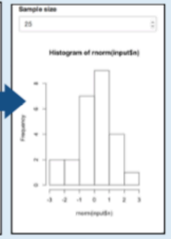
Add inputs to the UI with *Input() functions

Add outputs with *Output() functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

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

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



Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

ui.R contains everything you would save to ui.

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

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

The directory name is the name of the app

(optional) defines objects available to both ui.R and server.R

(optional) used in showcase mode

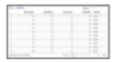
(optional) data, scripts, etc.

(optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

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

works with

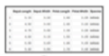


DT::**renderDataTable**(expr, options, callback, escape, env, quoted) — **dataTableOutput**(outputId, icon, …)

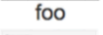
renderImage(expr, env, quoted, deleteFile) — **imageOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

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)



renderText(expr, env, quoted, func) — **textOutput**(outputId, container, inline)

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

Inputs

collect values from the user

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)

checkboxGroupInput(inputId, label, choices, selected, inline)

checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

fileInput(inputId, label, multiple, accept)

numericInput(inputId, label, value, min, max, step)

passwordInput(inputId, label, value)

radioButtons(inputId, label, choices, selected, inline)

selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

submitButton(text, icon)
(Prevents reactions across entire app)

textInput(inputId, label, value)

Shiny Application Structure

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

Shiny Application Structure

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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

Shiny Application Structure

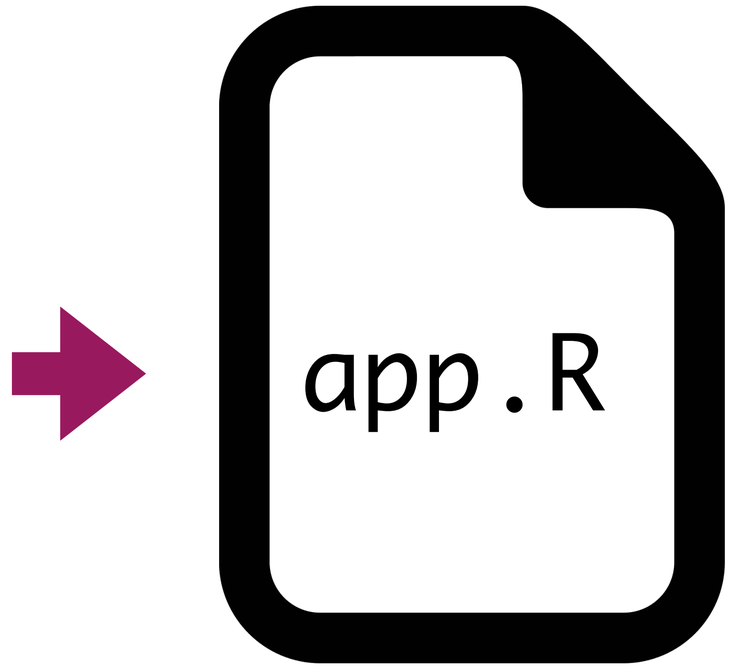
```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

Shiny Application Structure

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

Shiny Application Structure

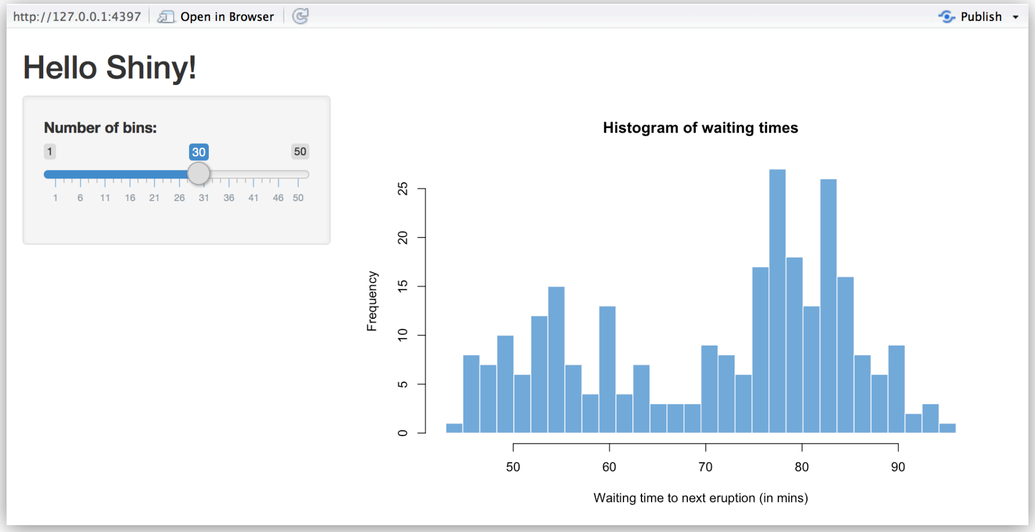
```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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



`runApp('file_directory/')`

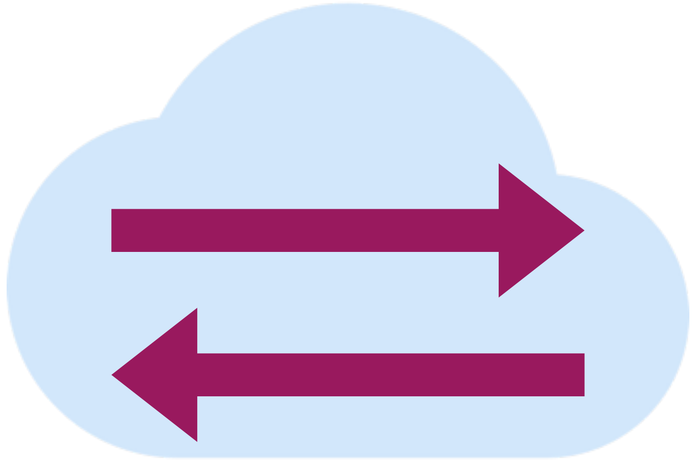
starts `shinyApp`

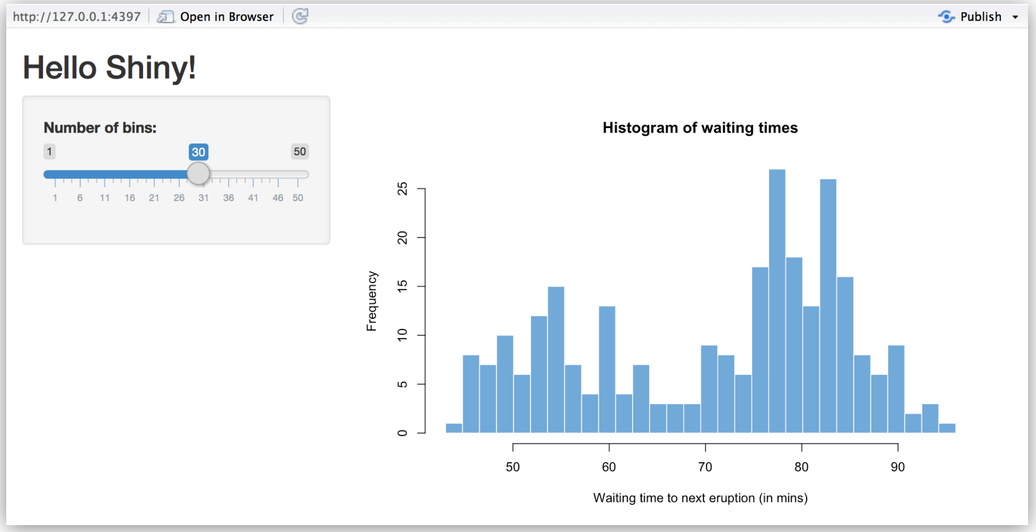
Locally Running Application




```
Listening on http://127.0.0.1:5859
```

Application Server





- Structure of Shiny Applications
- UI: Layout, Inputs and Outputs
- Server
- Bringing it Together
- Design Elements
- Interactive Graphics
- Sharing Your Shiny Application

UI: Layout

~/training/rshiny/exercises/exercise1 - Shiny

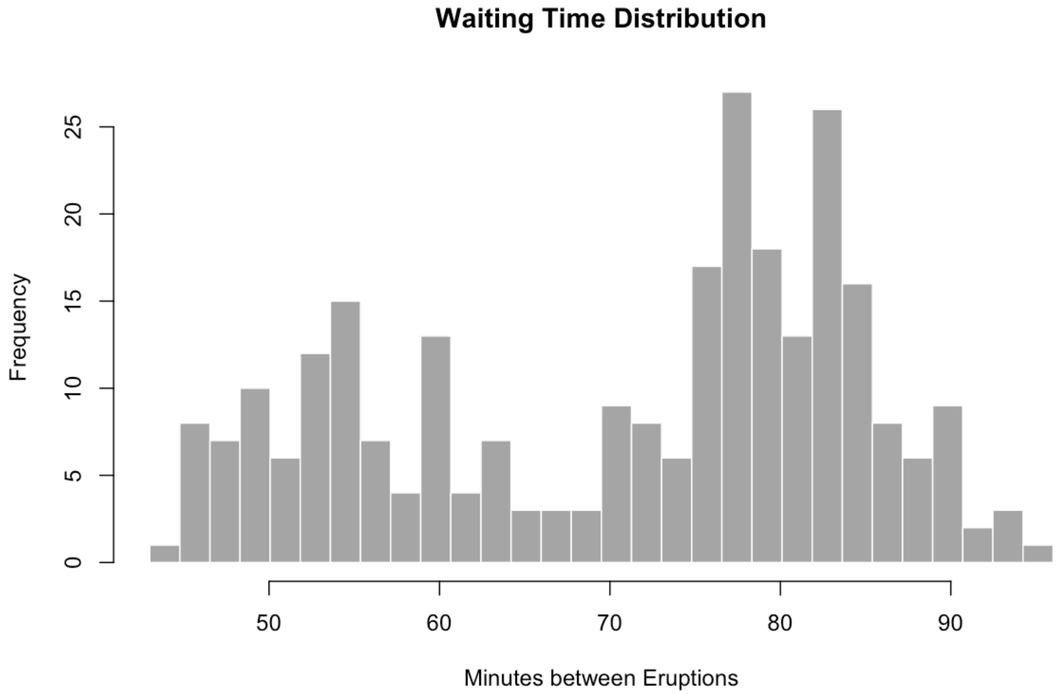
http://127.0.0.1:5859 | Open in Browser | Publish

Old Faithful Geyser Data

Number of bins:

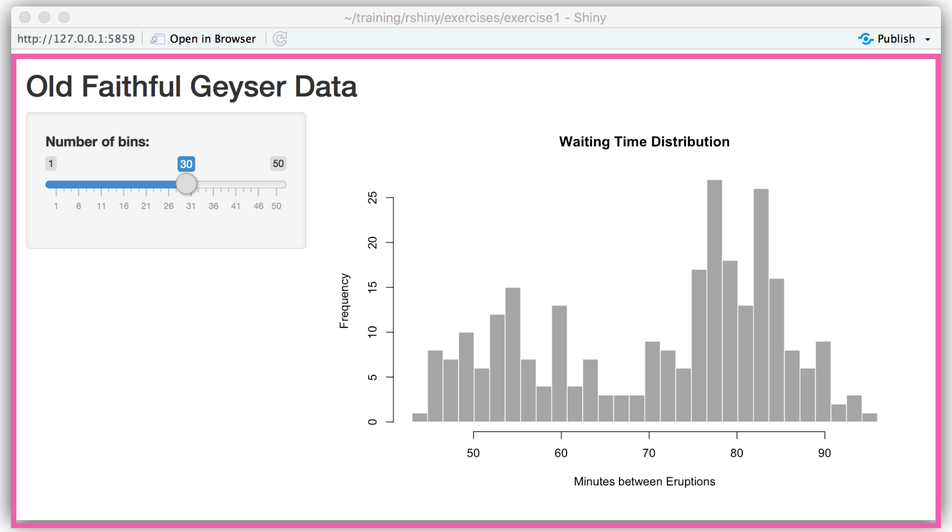
1 30 50

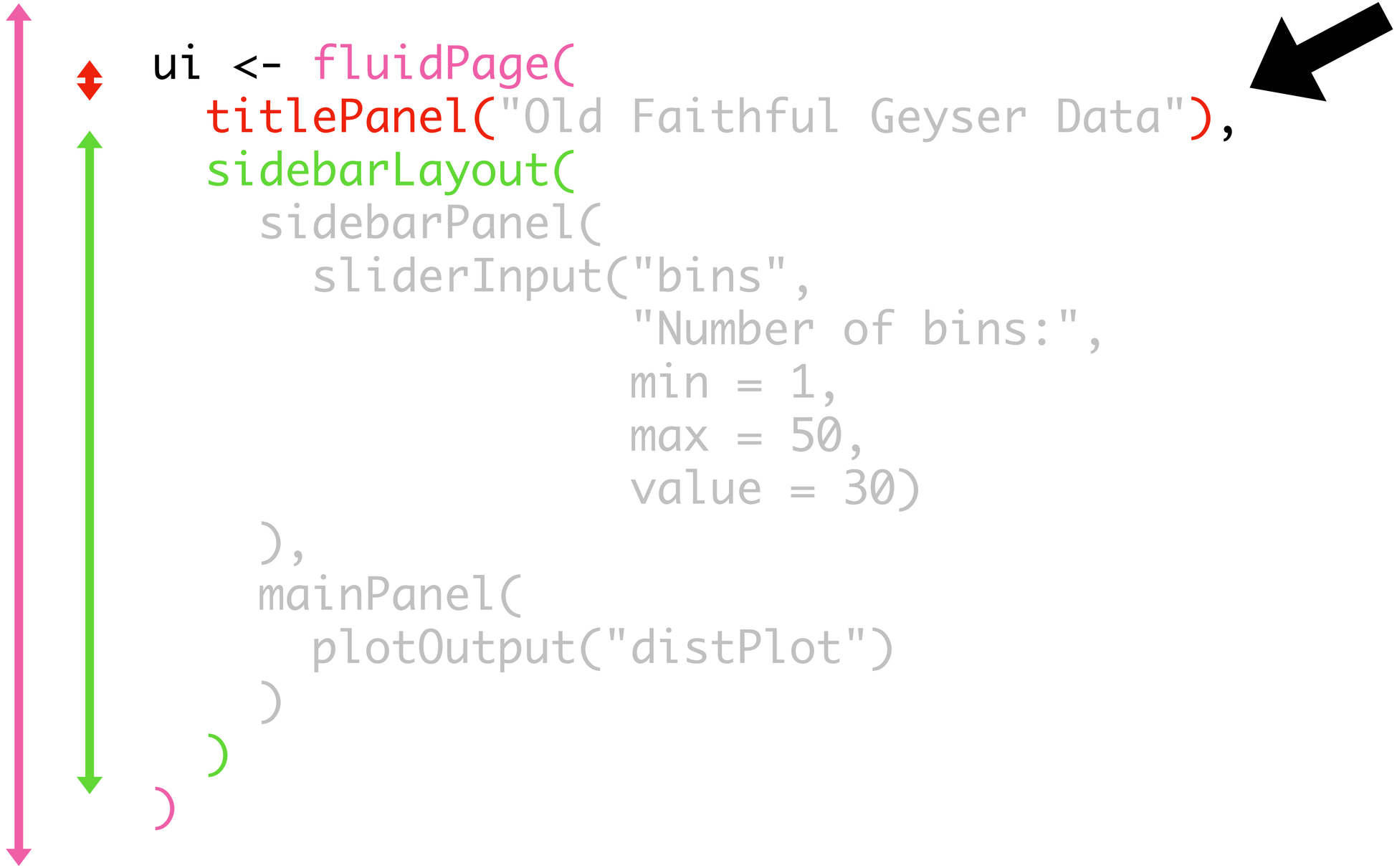
1 6 11 16 21 26 31 36 41 46 50



```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

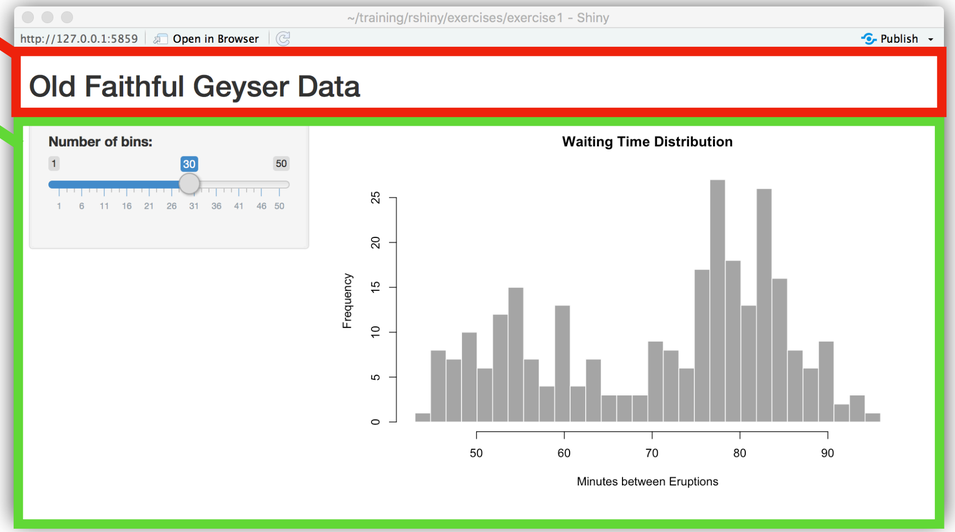
```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

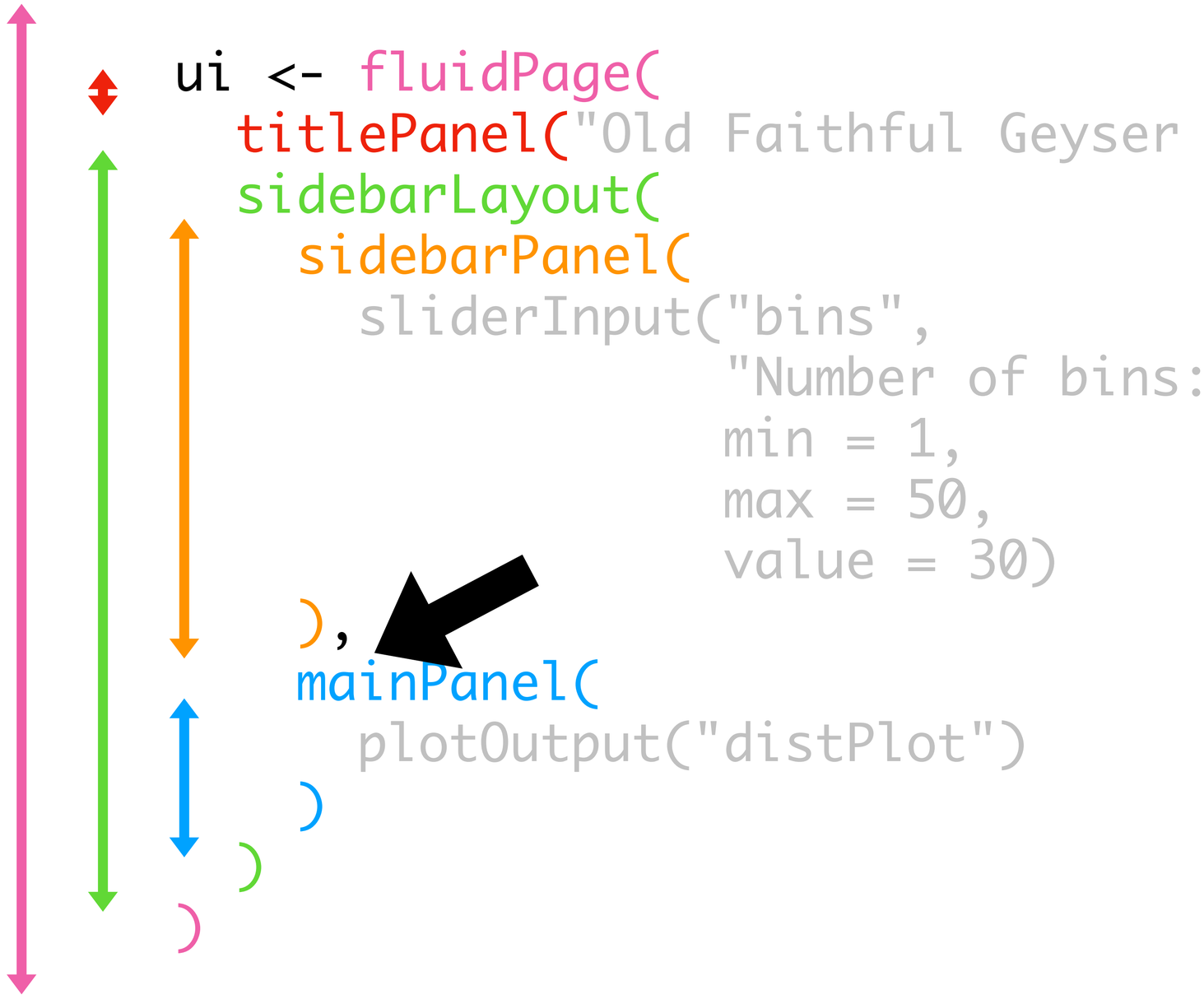




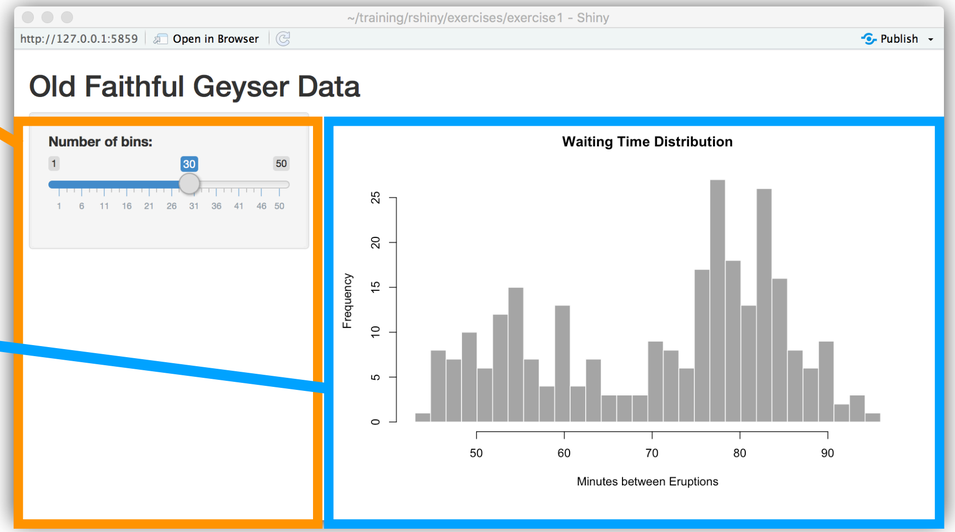
```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



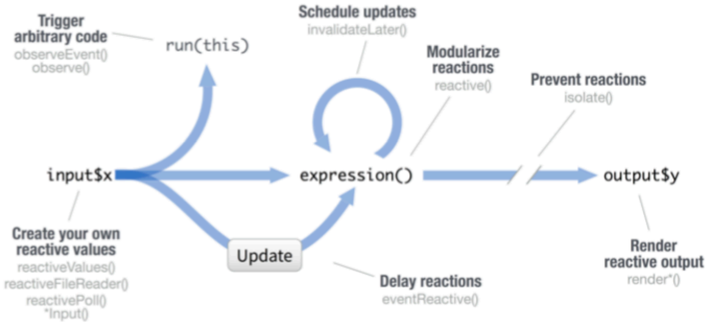

ui <- fluidPage(
titlePanel("Old Faithful Geyser Data"),
sidebarLayout(
sidebarPanel(
sliderInput("bins",
"Number of bins:",
min = 1,
max = 50,
value = 30)
),
mainPanel(
plotOutput("distPlot")
)
)
)

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error `Operation not allowed without an active reactive context.`



CREATE YOUR OWN REACTIVE VALUES

```
# example snippets

ui <- fluidPage(
 textInput("a","","A")
)

server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}
```

***Input() functions**
(see front page)

reactiveValues(...)

Each input function creates a reactive value stored as **input$<inputId>**

reactiveValues() creates a list of reactive values whose values you can set.

RENDER REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

render*() functions
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**

PREVENT REACTIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

TRIGGER ARBITRARY CODE

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <-
function(input,output){
 observeEvent(input$go,{
  print(input$a)
 })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

MODULARIZE REACTIONS

```
ui <- fluidPage(
 textInput("a","","A"),
 textInput("z","","Z"),
 textOutput("b"))

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input$z)})
 output$b <- renderText({
  re()
 })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain**)**

Creates a **reactive expression** that

- **caches** its value to reduce computation
- can be called by other code
- notifies its dependencies when it ha been invalidated

Call the expression with function syntax, e.g. `re()`

DELAY REACTIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
  re()
 })
}

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, domain, ignoreNULL)

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

UI - An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a","")
)
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##    <label for="a"></label>
##    <input id="a" type="text"
##      class="form-control" value=""/>
##  </div>
## </div>
```

Returns HTML

Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

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$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

The most common tags have wrapper functions. You do not need to prefix their names with **tags$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>")
)
```

To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$script(src = "<file name>"))
```

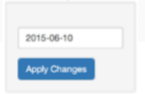
IMAGES

To include an image
1. Place the file in the **www** subdirectory
2. Link to it with `img(src="<file name>")`

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

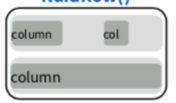
```
wellPanel(dateInput("a", ""),
          submitButton()
)
```



absolutePanel()	navlistPanel()
conditionalPanel()	sidebarPanel()
fixedPanel()	tabPanel()
headerPanel()	tabsetPanel()
inputPanel()	titlePanel()
mainPanel()	wellPanel()

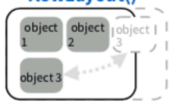
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()



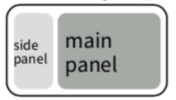

```
ui <- fluidPage(
 fluidRow(column(width = 4),
   column(width = 2, offset = 3)),
 fluidRow(column(width = 12))
)
```

flowLayout()



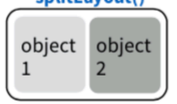

```
ui <- fluidPage(
 flowLayout( # object 1,
             # object 2,
             # object 3
 )
)
```

sidebarLayout()



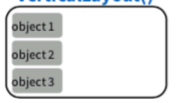

```
ui <- fluidPage(
 sidebarLayout(
  sidebarPanel(),
  mainPanel()
 )
)
```

splitLayout()




```
ui <- fluidPage(
 splitLayout( # object 1,
              # object 2
 )
)
```

verticalLayout()




```
ui <- fluidPage(
 verticalLayout( # object 1,
                 # object 2,
                 # object 3
 )
)
```

Layer tabPanels on top of each other, and navigate between them, with:

```
ui <- fluidPage(tabsetPanel(
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents")))
```

```
ui <- fluidPage(navlistPanel(
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents")))
```

```
ui <- navbarPage(title = "Page",
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents"))
```

Exercise 1B

Using the app you created in the first exercise:

1. Change the `sidebarLayout` to `verticalLayout`

   > Run app to see changes

2. Change the `sidebarPanel` to `inputPanel`

   > Run/Reload app to see changes

3. Between the `inputPanel` and the `mainPanel`, insert a `headerPanel("The Plot")`

   > Run/Reload app to see changes

~/training/rshiny/exercises/exercise1 - Shiny

http://127.0.0.1:5610 | Open in Browser | Publish

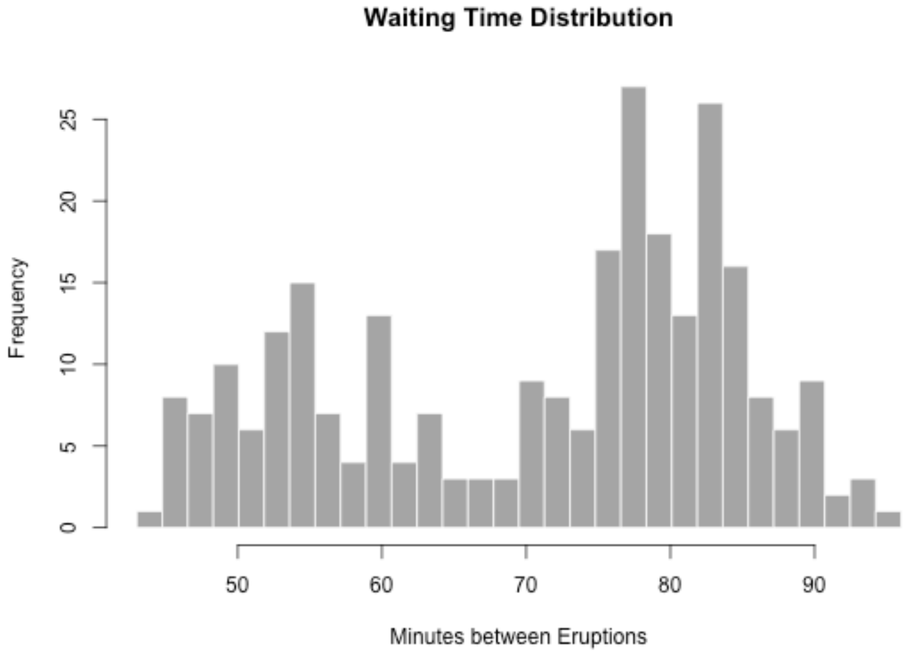
Old Faithful Geyser Data

Number of bins:

1 30 50

1 6 11 16 21 26 31 36 41 46 50

The Plot

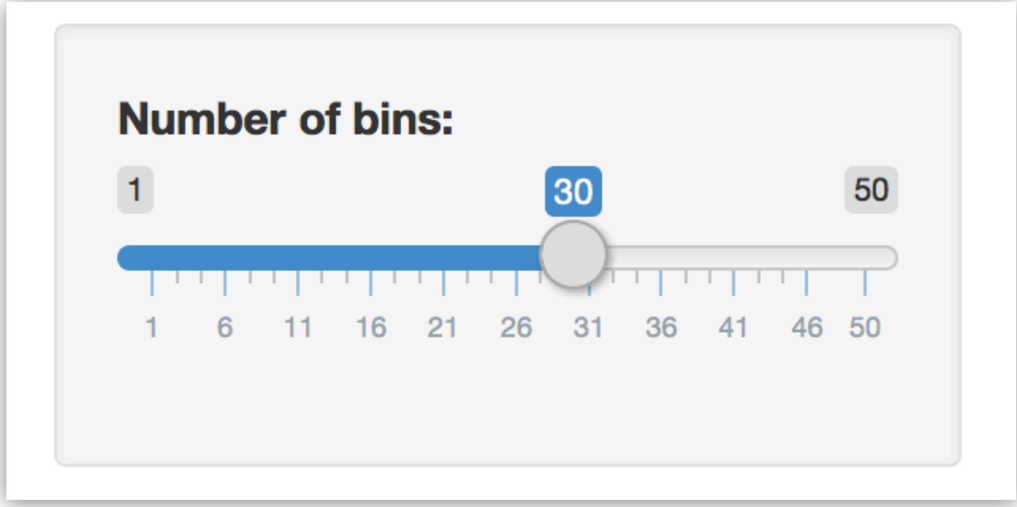


```
verticalLayout(
      inputPanel(
         sliderInput("bins",
                     "Number of bins:",
                     min = 1,
                     max = 50,
                     value = 30)
      ),
      headerPanel("The Plot"),
      mainPanel(
         plotOutput("distPlot")
      )
   )
```

UI: Inputs and Outputs

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

Name of the input: `inputId`



Shiny : : CHEAT SHEET

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

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

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE or run:
   rsconnect::deployApp("<path to directory>")

Build or purchase your own Shiny Server
at www.rstudio.com/products/shiny-server/

RStudio

Building an App

Complete the template by adding arguments to fluidPage() and a body to the server function.

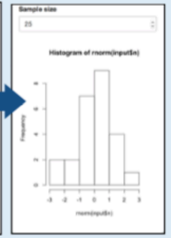
Add inputs to the UI with *Input() functions

Add outputs with *Output() functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

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

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



Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

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

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

The directory name is the name of the app

(optional) defines objects available to both ui.R and server.R

(optional) used in showcase mode

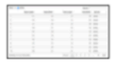
(optional) data, scripts, etc.

(optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

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

works with

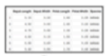


DT::**renderDataTable**(expr, options, callback, escape, env, quoted) — **dataTableOutput**(outputId, icon, …)

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)

renderText(expr, env, quoted, func) — **textOutput**(outputId, container, inline)

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

Inputs

collect values from the user

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)

checkboxGroupInput(inputId, label, choices, selected, inline)

checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

fileInput(inputId, label, multiple, accept)

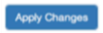
numericInput(inputId, label, value, min, max, step)

passwordInput(inputId, label, value)

radioButtons(inputId, label, choices, selected, inline)

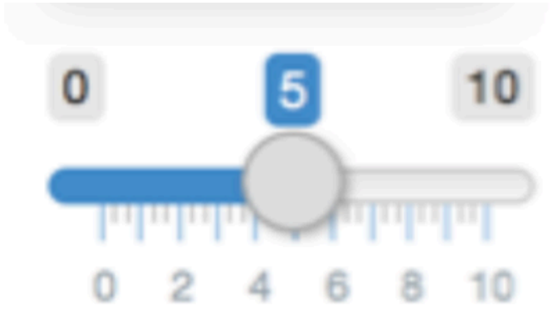
selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)



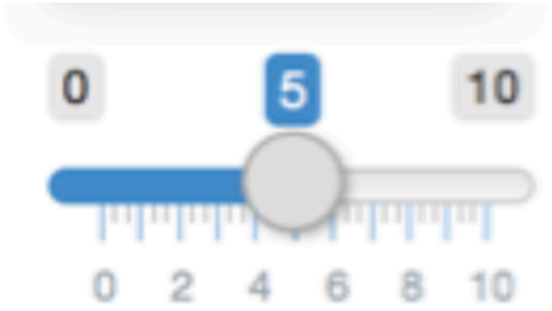
submitButton(text, icon)
(Prevents reactions across entire app)

textInput(inputId, label, value)



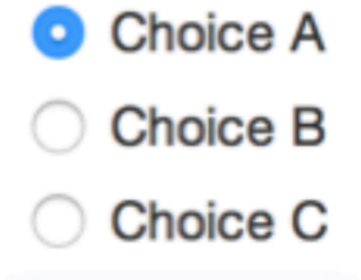
sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

```
sliderInput("bins",
            "Number of bins:",
            min = 1,
            max = 50,
            value = 30)
```



sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

```
sliderInput(inputId = “bins",
            label = “Number of bins:",
            min = 1,
            max = 50,
            value = 30)
```



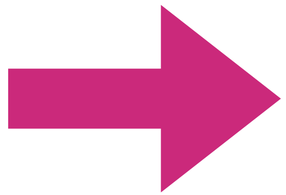
radioButtons(inputId, label, choices, selected, inline)

```
radioButtons(inputId = "choices",
             label = "Choose One:",
             choices=c("Choice A",
                       "Choice B",
                       "Choice C")
             )
```

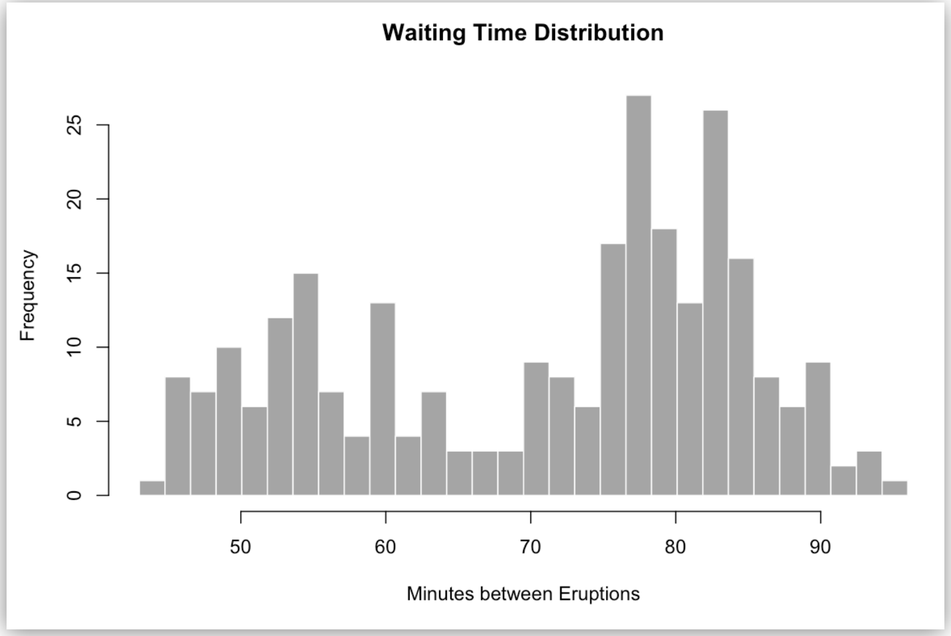
`inputId`

is how you get the value

of the input later

`inputId="bins"`  `input$bins`

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



Name of the output: `outputId`

Shiny : : CHEAT SHEET

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

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

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE or run:
   rsconnect::deployApp("<path to directory>")

Build or purchase your own Shiny Server
at www.rstudio.com/products/shiny-server/

RStudio

Building an App

Complete the template by adding arguments to fluidPage() and a body to the server function.

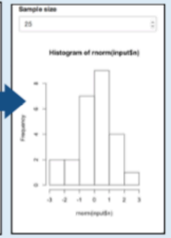
Add inputs to the UI with *Input() functions

Add outputs with *Output() functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

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

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



Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

ui.R contains everything you would save to ui.

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

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

The directory name is the name of the app

(optional) defines objects available to both ui.R and server.R

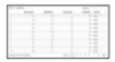
(optional) used in showcase mode

(optional) data, scripts, etc.

(optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

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



DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

works with

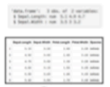
dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

renderPlot(expr, width, height, res, …, env, quoted, func)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)



renderPrint(expr, env, quoted, func, width)

renderTable(expr,…, env, quoted, func

verbatimTextOutput(outputId)

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, …)

Inputs

collect values from the user

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)

checkboxGroupInput(inputId, label, choices, selected, inline)

checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

fileInput(inputId, label, multiple, accept)

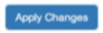
numericInput(inputId, label, value, min, max, step)

passwordInput(inputId, label, value)

radioButtons(inputId, label, choices, selected, inline)

selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)



submitButton(text, icon)
(Prevents reactions across entire app)

textInput(inputId, label, value)

plotOutput(outputId, width, height, click,
dblclick, hover, hoverDelay, inline,
hoverDelayType, brush, clickId, hoverId)

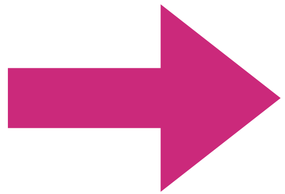
```
plotOutput("distPlot")
```

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

```
plotOutput(outputId="distPlot")
```

`outputId`

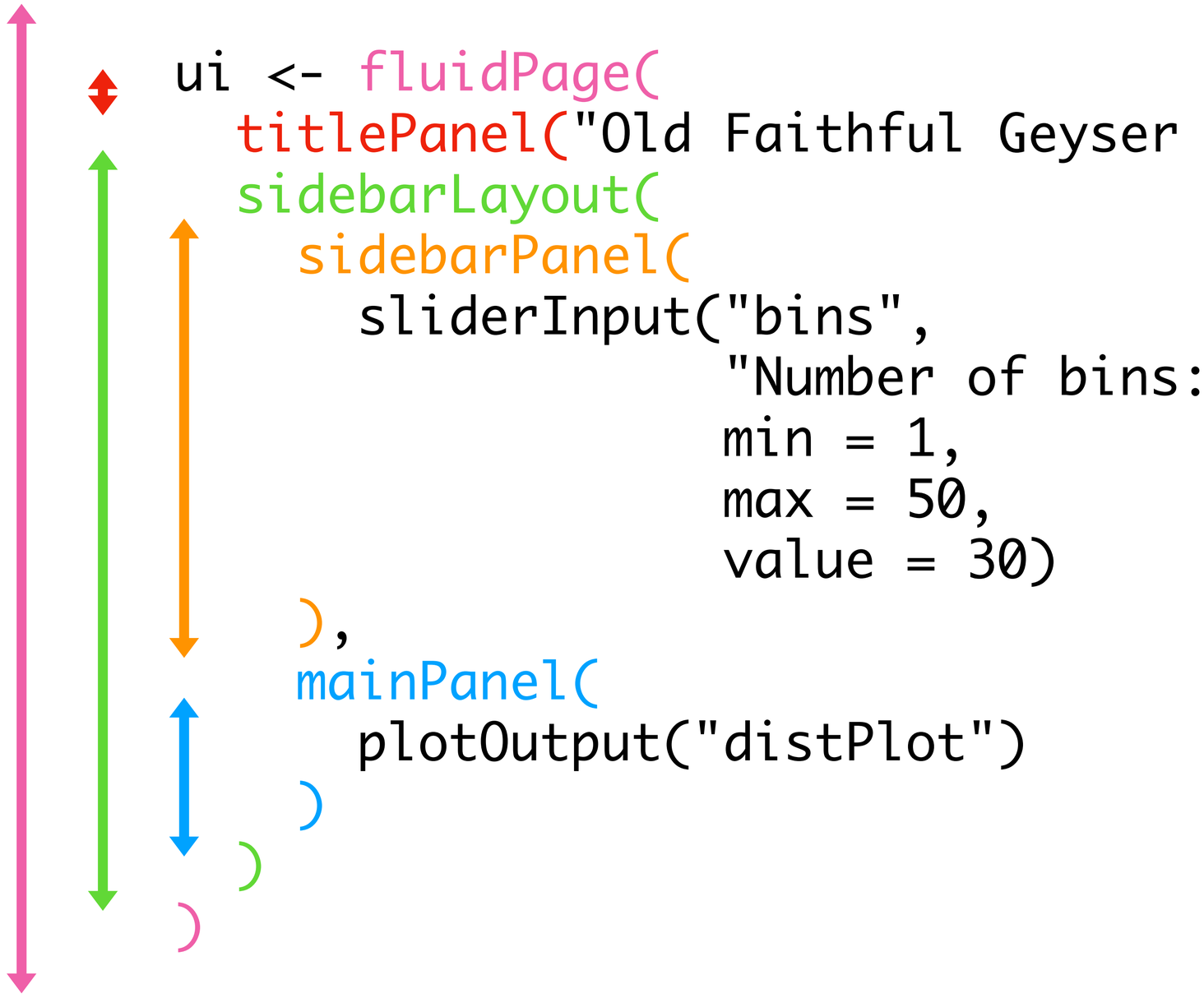
is how you set the value
of the output later

`outputId="distPlot"`  `output$distPlot`

Shiny makes two list-like objects:

`input`: all inputs, named by `inputIds`

`output`: all outputs, named by `outputIds`



```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

Exercise 1C

Create a new Shiny app, as you did in Exercise 1A, or work from the previous file

To the `sidebarPanel` (or `inputPanel` if you changed it last exercise), add a `radioButtons` input with

- `inputId="color"`
- `label="Bar Color"`
- `choices=c("red","blue","green")`

Run the app to make sure there aren't an errors. The radio input won't have an effect yet though.

~/training/rshiny/exercises/exercise1 - Shiny

http://127.0.0.1:5610 | Open in Browser | Publish

Old Faithful Geyser Data

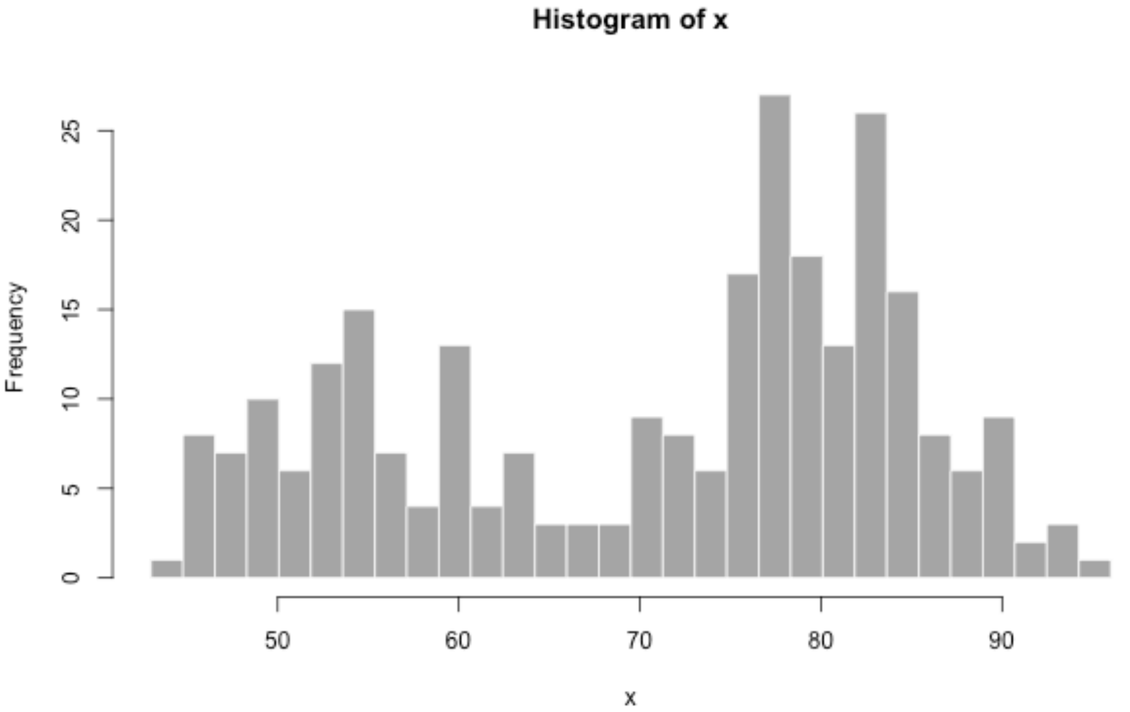
Number of bins:

1 30 50

1 6 11 16 21 26 31 36 41 46 50

Color

- (•) red
- ( ) blue
- ( ) green



```
sidebarPanel(
        sliderInput("bins",
                    "Number of bins:",
                    min = 1,
                    max = 50,
                    value = 30),
        radioButtons(inputId="color",
                     label="Color",
                     choices=c("red","blue","green"))
      ),
```

Structure of Shiny Applications

UI: Layout, Inputs and Outputs

Server

Bringing it Together

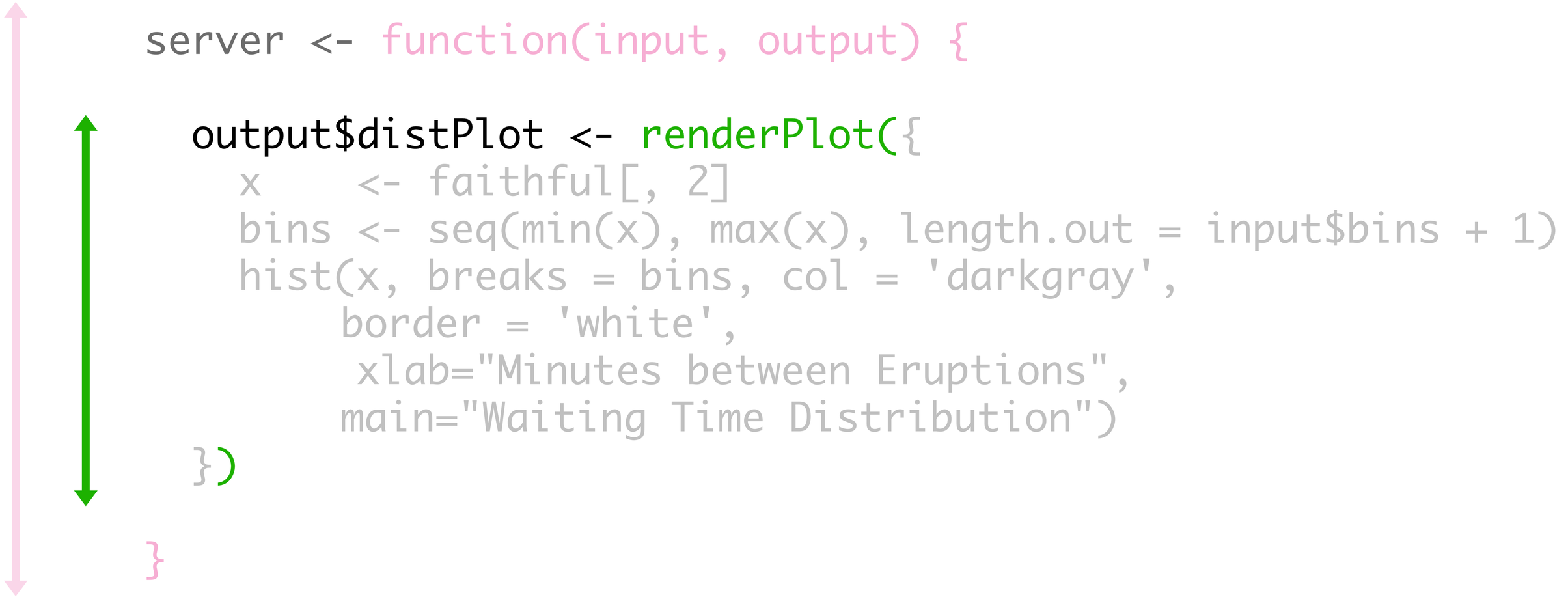
Design Elements

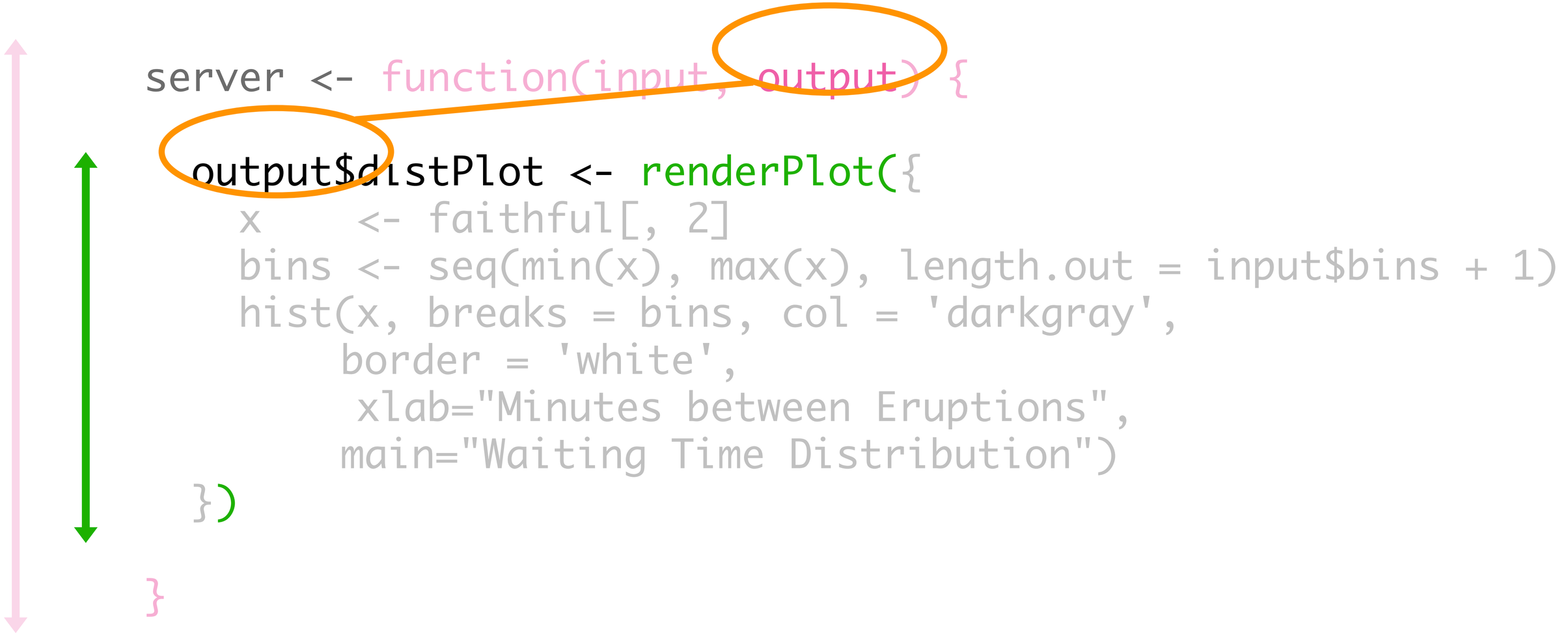
Interactive Graphics

Sharing Your Shiny Application

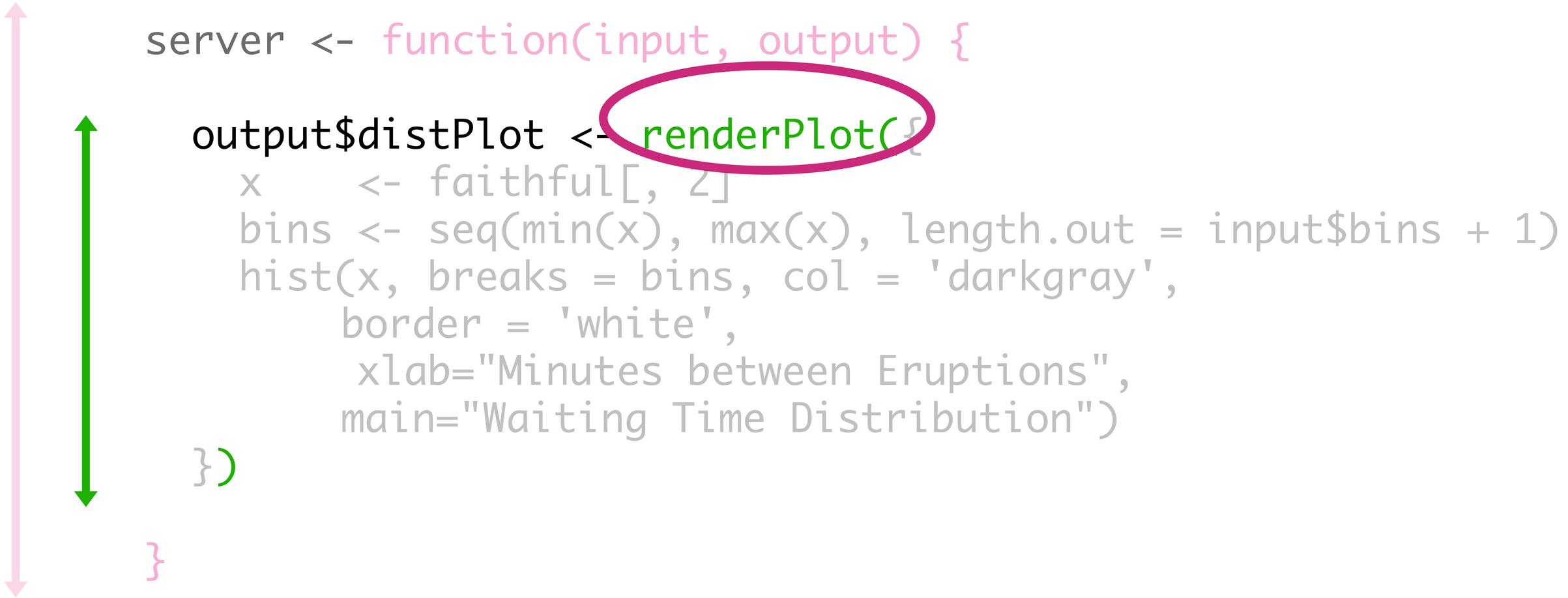
```
server <- function(input, output) {

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray',
         border = 'white',
          xlab="Minutes between Eruptions",
         main="Waiting Time Distribution")
  })

}
```

Shiny supplies these from the UI


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins", "Number of bins:", min = 1, max = 50, value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)

server <- function(input, output) {
  output$distPlot <- renderPlot({
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray', border = 'white',
         xlab="Minutes between Eruptions", main="Waiting Time Distribution")
  })
}
```


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}

Shiny : : CHEAT SHEET

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

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

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE or run:
   rsconnect::deployApp("<path to directory>")

Build or purchase your own Shiny Server
at www.rstudio.com/products/shiny-server/

RStudio

Building an App

Complete the template by adding arguments to fluidPage() and a body to the server function.

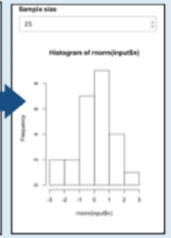
Add inputs to the UI with *Input() functions

Add outputs with *Output() functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

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

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



Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

ui.R contains everything you would save to ui.

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

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

The directory name is the name of the app

(optional) defines objects available to both ui.R and server.R

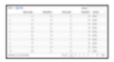
(optional) used in showcase mode

(optional) data, scripts, etc.

(optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

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



DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

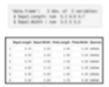
dataTableOutput(outputId, icon, ...)

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)

renderTable(expr,..., env, quoted, func)

verbatimTextOutput(outputId)

tableOutput(outputId)

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, ...)

htmlOutput(outputId, inline, container, ...)

Inputs

collect values from the user

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

actionButton(inputId, label, icon, ...)

actionLink(inputId, label, icon, ...)

checkboxGroupInput(inputId, label, choices, selected, inline)

checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

fileInput(inputId, label, multiple, accept)

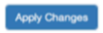
numericInput(inputId, label, value, min, max, step)

passwordInput(inputId, label, value)

radioButtons(inputId, label, choices, selected, inline)

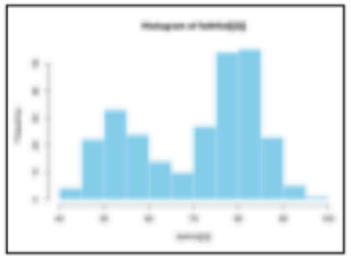
selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)



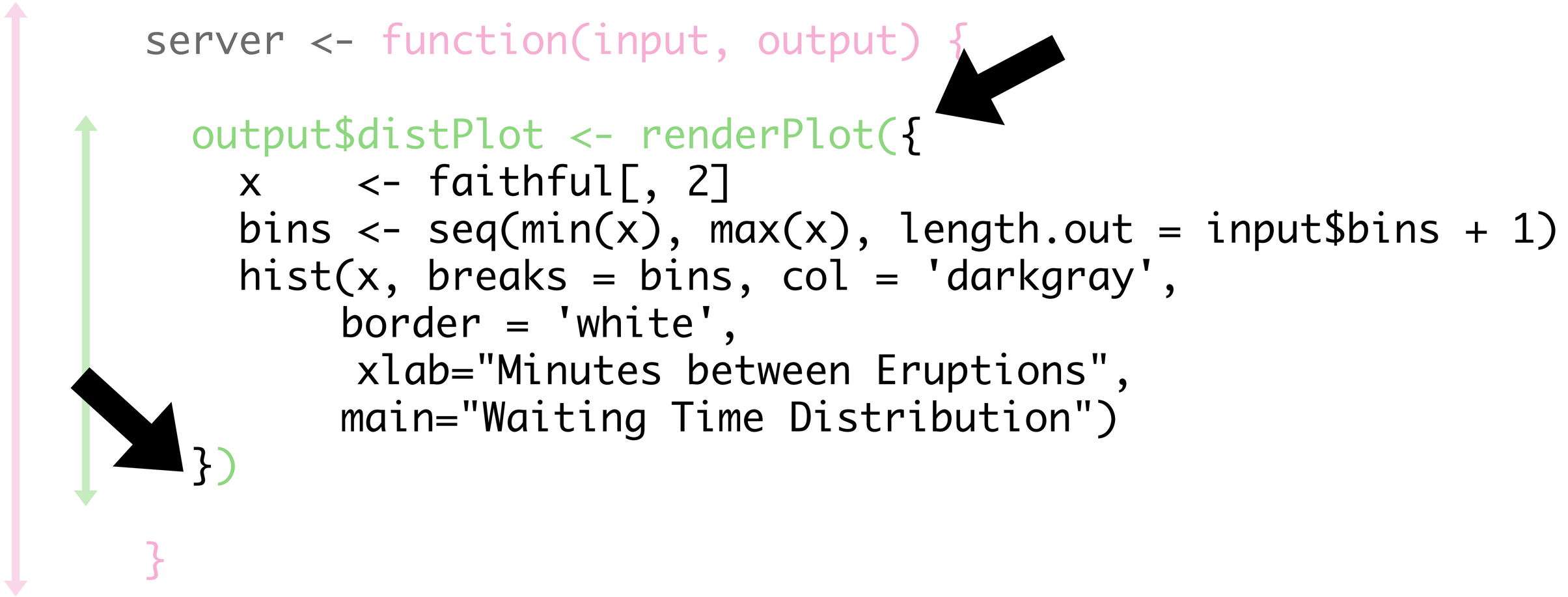
submitButton(text, icon)
(Prevents reactions across entire app)

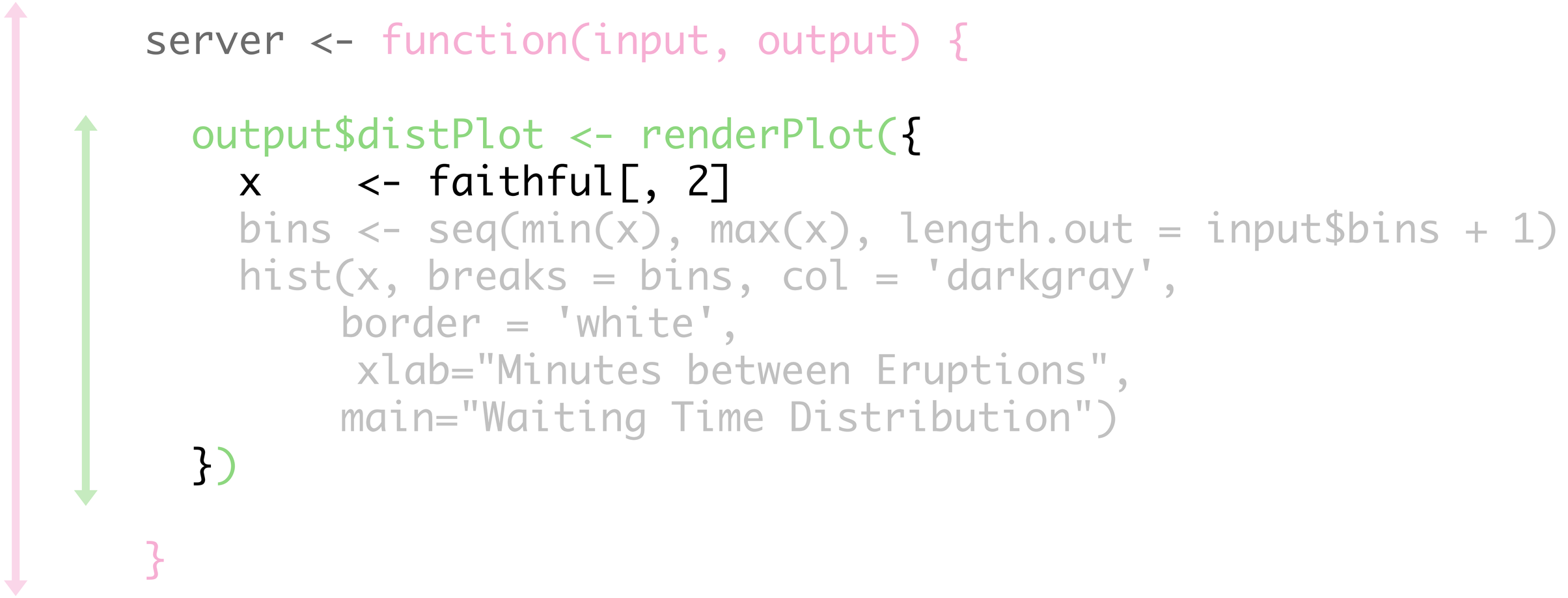
textInput(inputId, label, value)



renderPlot(expr, width, height, res, …,
env, quoted, func)

- Any R expression
- Multiple lines in {}


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}

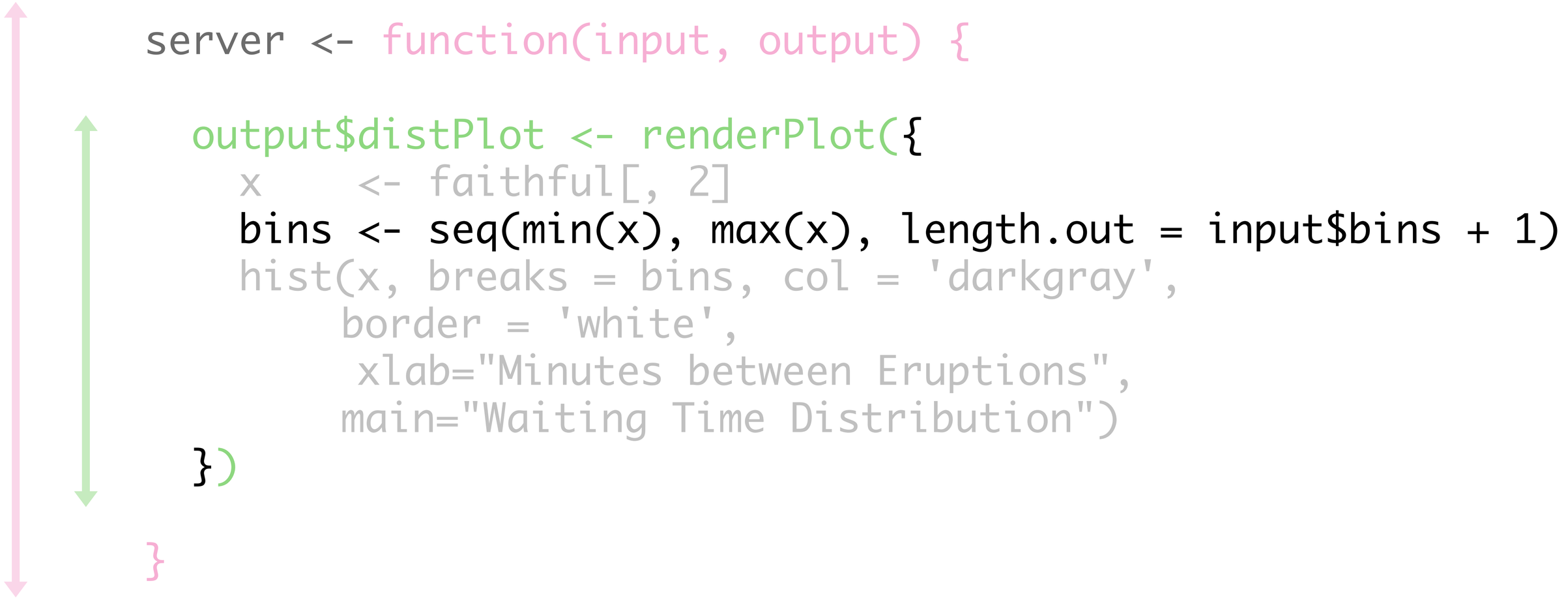


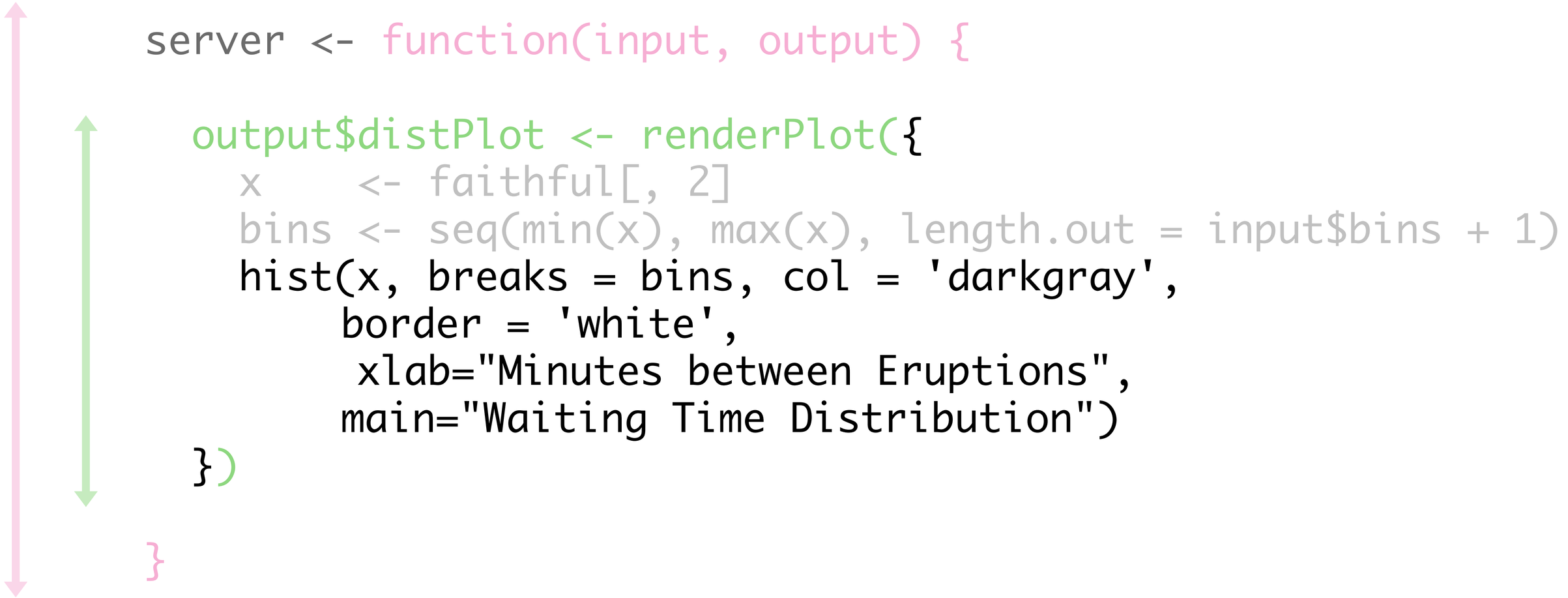
```
server <- function(input, output) {

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray',
         border = 'white',
          xlab="Minutes between Eruptions",
         main="Waiting Time Distribution")
  })

}
```

Console | Terminal

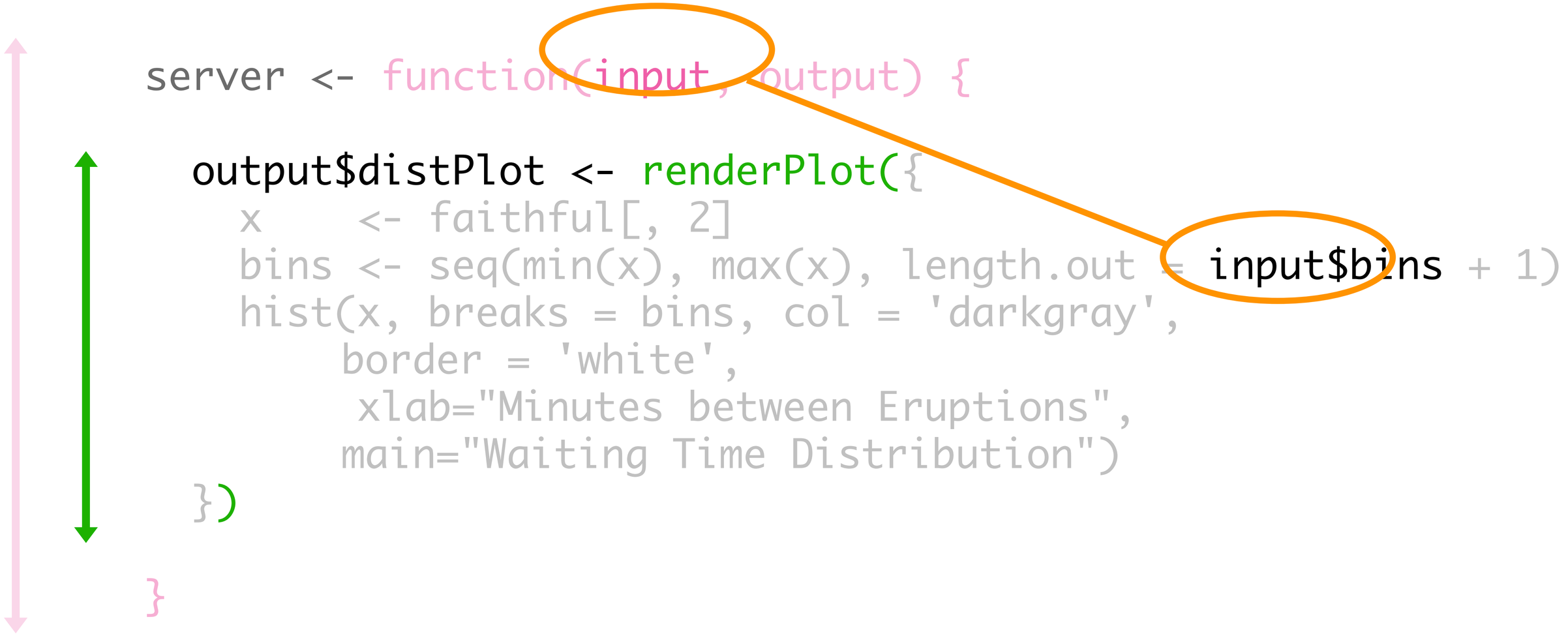
~/training/rshiny/

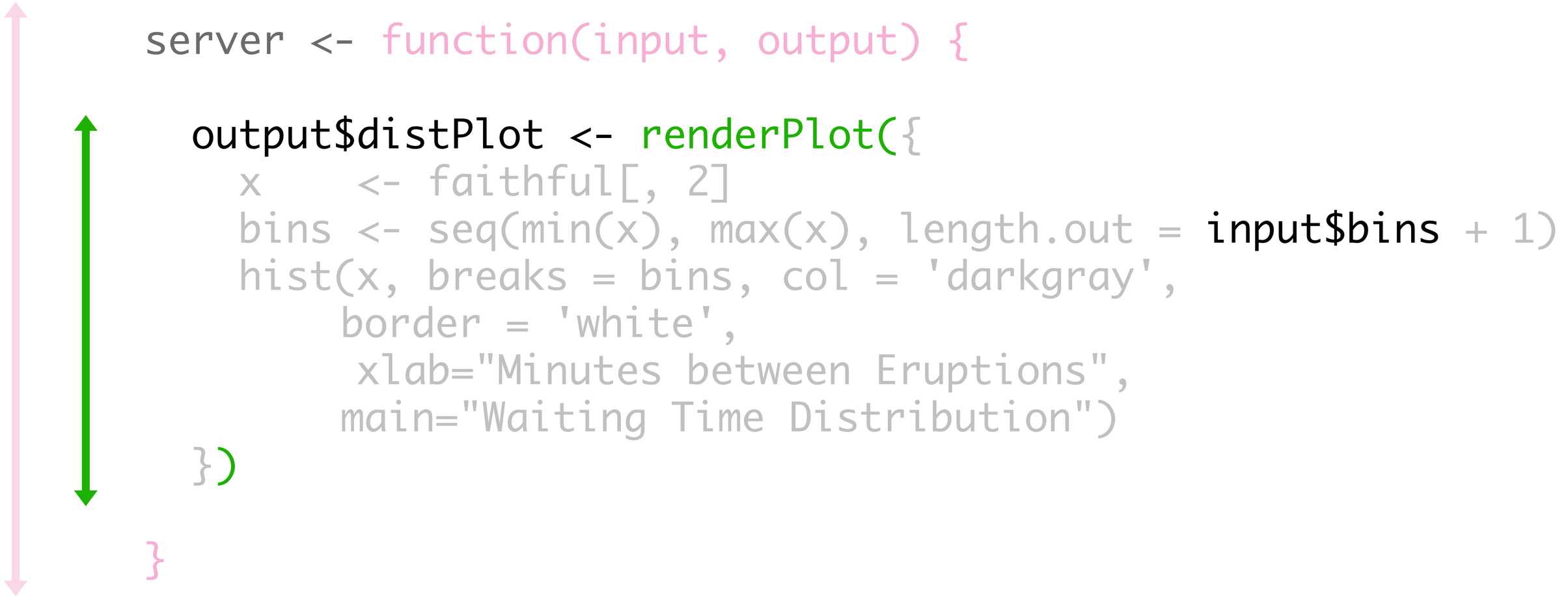
```
> faithful
    eruptions waiting
1       3.600      79
2       1.800      54
3       3.333      74
4       2.283      62
5       4.533      85
6       2.883      55
7       4.700      88
8       3.600      85
9       1.950      51
10      4.350      85
11      1.833      54
12      3.917      84
13      4.200      78
14      1.750      47
15      4.700      83
16      2.167      52
17      1.750      62
```


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}



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

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray',
         border = 'white',
          xlab="Minutes between Eruptions",
         main="Waiting Time Distribution")
  })

}
```


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}


server <- function(input, output) {
output$distPlot <- renderPlot({
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = input$bins + 1)
hist(x, breaks = bins, col = 'darkgray',
border = 'white',
xlab="Minutes between Eruptions",
main="Waiting Time Distribution")
})
}

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins", "Number of bins:", min = 1, max = 50, value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)

server <- function(input, output) {
  output$distPlot <- renderPlot({
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray', border = 'white',
         xlab="Minutes between Eruptions", main="Waiting Time Distribution")
  })
}
```

Exercise 1D

03:00

Using the app you created in Exercise 1C (with the color `radioButtons`), or exercises/exercise1/answer1c.R:

Set the color (`col=`) of the bars in the histogram (`hist`) using the value from the color `radioButtons` input

BONUS

1. Change the default color choice to green instead of red using `selected=` as part of the `radioButtons` input
2. Add additional color choices

~/training/rshiny/exercises/exercise1 - Shiny

http://127.0.0.1:5610 | Open in Browser | Publish

Old Faithful Geyser Data

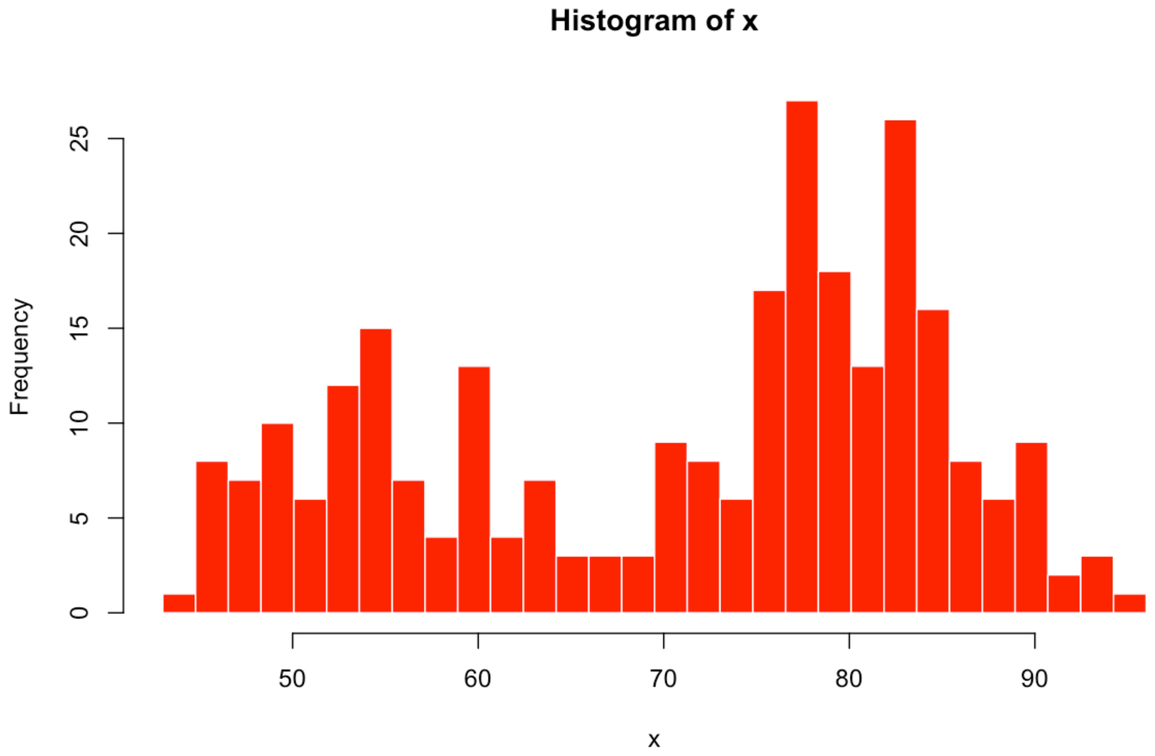
Number of bins:

1 30 50

1 6 11 16 21 26 31 36 41 46 50

Color

- (•) red
- ( ) blue
- ( ) green



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

   output$distPlot <- renderPlot({
      # generate bins based on input$bins from ui.R
      x    <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)

      # draw the histogram with the specified number of bins
      hist(x, breaks = bins, col = input$color, border = 'white')
   })
}
```

Structure of Shiny Applications

UI: Layout, Inputs and Outputs

Server

Bringing it Together

Design Elements

Interactive Graphics

Sharing Your Shiny Application

Debugging

Watch out for formatting errors!

1. Run code
2. Look at error messages
3. Try to fix
4. Run again

Location of the error may be slightly above the line number referenced in the error message

Exercise 2A & B

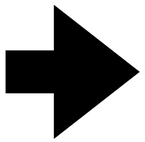
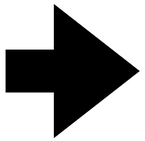
05:00

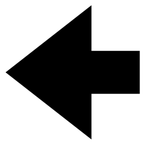
exercises/exercise2/app.R

A: The code contains errors — try to fix them and get the code to run.

B (if time): Fill in the code to render a DataTable of the entire data set.

- Hints:
  - What is the name of the element on output you need to set? `output$_____`
  - Using the cheat sheet, what is the render function for a DataTable that goes with `dataTableOutput`?
  - The only expr you need for the render function is the name of the data object. What did we call the dataset when importing it at the top of the file?



```
...
    mainPanel(
      plotOutput("plot1")
    )
  ),
  verticalLayout(
    DT::dataTableOutput("datatable1")
  )
)
server <- function(input, output) {
  output$plot1 <- renderPlot({
    ggplot(data, aes_string(x="area", y=input$yval)) +
      geom_point() +
      geom_smooth(color="red", method="lm") +
      xlab("Area (sq km)") +
      theme_minimal()
  })
  output$datatable1 <- DT::renderDataTable(data)
}
shinyApp(ui = ui, server = server)
```

From above, in UI:

```
DT::dataTableOutput("datatable1")
```

```
output$datatable1 <- DT::renderDataTable(data)
```

From above, after library imports:

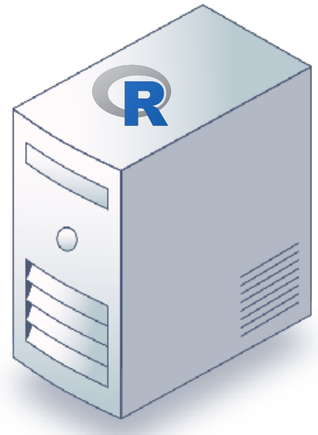
```
data <- read_csv("africadata.csv")
```

Applications and Sessions

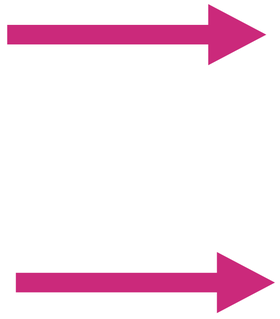
Application vs. Sessions

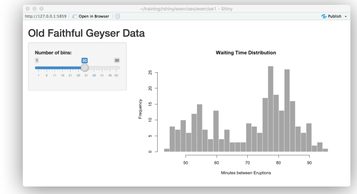
Multiple Applications

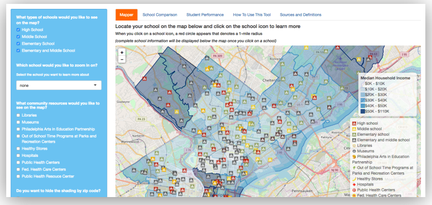
Multiple Sessions

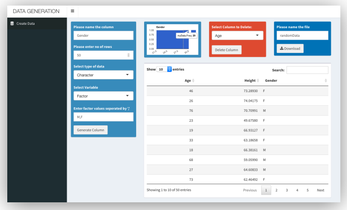


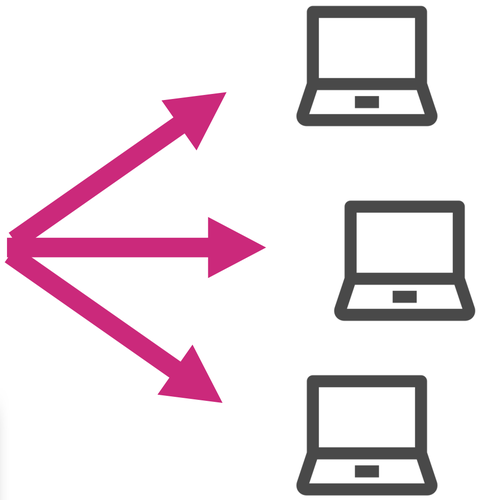
Shiny Server

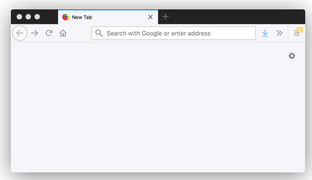


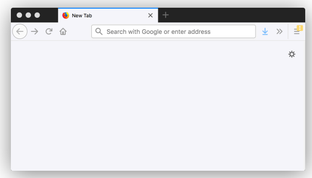












Shiny Application Structure

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

Shiny Application Structure

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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

Called once when application starts

Shiny Application Structure

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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

Called once when
a user connects —
once for each session

Shiny Application Structure

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

Chunks of code
inside `server` function
can be called many times

Shiny Application Structure

```
library(shiny)


ui <- fluidPage()

server <- function(input, output) {}

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

Add code that should only be called
once per application here

Shiny Application Structure

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {

    output$x <- …
}

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

Add code that should only be calle
once per session here

Reactivity

- `input` is a reactive source
- `output` is a reactive endpoint
- Shiny manages `input` and `output` for you
- You can
  - Define other relationships
  - Create more complicated conditions
  - Trigger events
  - Interrupt relationships

Exercise 2

```
library(shiny)
library(readr)
library(ggplot2)

data <- read_csv("africadata.csv")

ui <- fluidPage(
  titlePanel("Africa: Country Size"),
  sidebarLayout(
    sidebarPanel(
      selectInput(inputId = "yval",
                  label = "Y-axis:",
                  choices = c("GDP" = "gdp_2017", "Population" = "pop_2017",
                              "Railroads" = "rail", "Roads" = "road"),
                  selected = "gdp_2017")
    ),
    mainPanel(
      plotOutput("plot1")
    )
  ),
  verticalLayout(
    DT::dataTableOutput("datatable1")
  )
)
```

```
library(shiny)
library(readr)
library(ggplot2)

data <- read_csv("africadata.csv")

ui <- fluidPage(
  titlePanel("Africa: Country Size"),
  sidebarLayout(
    sidebarPanel(
      selectInput(inputId = "yval",
                  label = "Y-axis:",
                  choices = c("GDP" = "gdp_2017", "Population" = "pop_2017",
                              "Railroads" = "rail", "Roads" = "road"),
                  selected = "gdp_2017")
    ),
    mainPanel(
      plotOutput("plot1")
    )
  ),
  verticalLayout(
    DT::dataTableOutput("datatable1")
  )
)
```

input$yval

output$plot1

output$datatable1

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

  output$plot1 <- renderPlot({
      ggplot(data, aes_string(x="area", y=input$yval)) +
        geom_point() +
        geom_smooth(color="red", method="lm") +
        xlab("Area (sq km)") +
        theme_minimal()
  })

  output$datatable1 <- DT::renderDataTable(data)
}
```

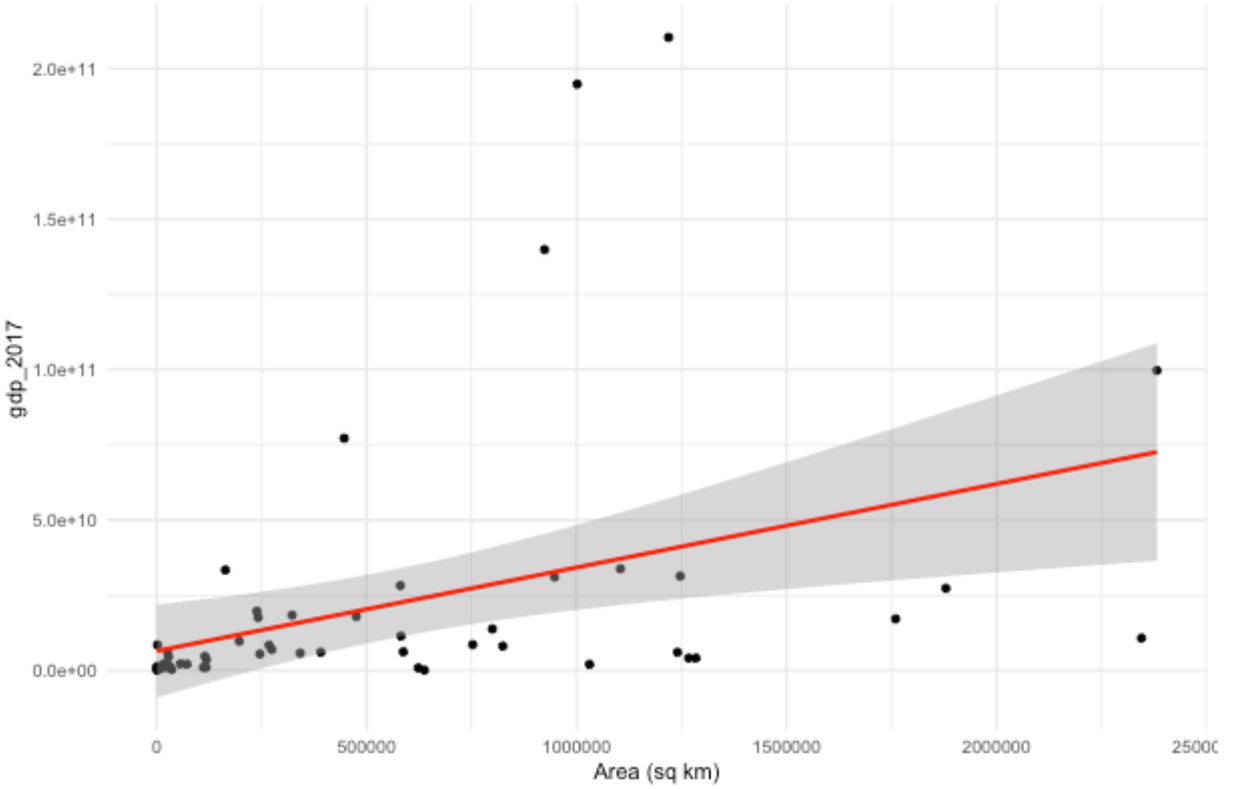
```
server <- function(input, output) {

  output$plot1 <- renderPlot({
      ggplot(data, aes_string(x="area", y=input$yval)) +
        geom_point() +
        geom_smooth(color="red", method="lm") +
        xlab("Area (sq km)") +
        theme_minimal()
  })

  output$datatable1 <- DT::renderDataTable(data)
}
```

Structure of Shiny Applications
UI: Layout, Inputs and Outputs
Server
Bringing it Together
Design Elements
Interactive Graphics
Sharing Your Shiny Application

Africa: Country Size

Y-axis:

GDP



Show 10 **entries**

Search:

country	region	gdp_2017	pop_2017	area	rail	road
No data available in table						

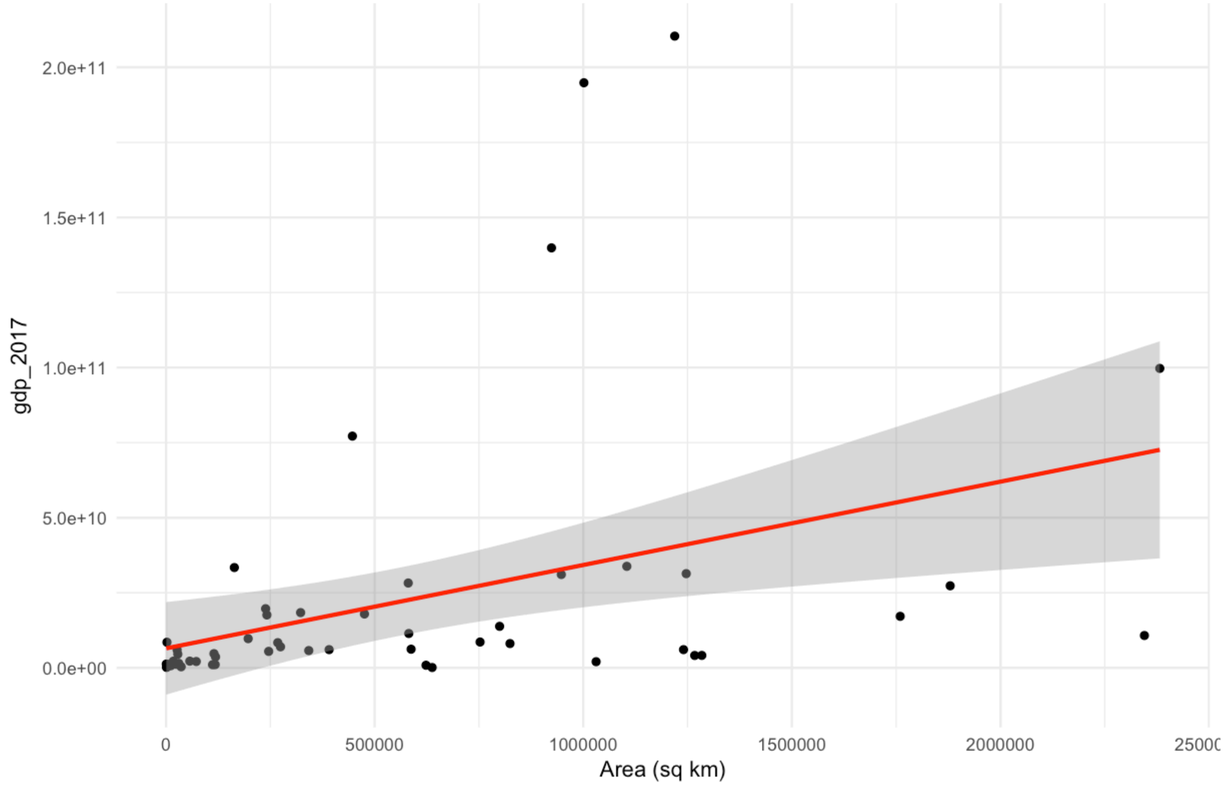
Showing 0 to 0 of 0 entries

Previous Next

Africa: Country Size

Y-axis:

GDP



Data Source: African Development Bank Statistical Data Portal

Show 10 **entries**

Search:

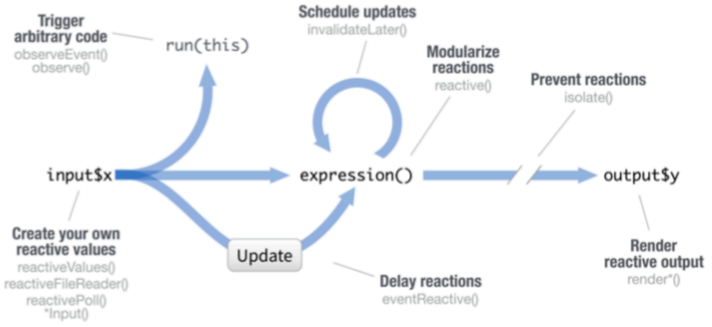
	country	region	gdp_2017	pop_2017	area	rail	road
1	Algeria	north	99722535234	41063753	2381740	4691	113655

Tags

- Add static content
- Wrap tags around textOutput(), htmlOutput()
- Named for, and translated into, HTML tags
- Can be nested
- tags$____
- Common tags don't need `tags$`: p(), br(), h1(), etc.
- Write your own HTML with HTML()

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error `Operation not allowed without an active reactive context.`



CREATE YOUR OWN REACTIVE VALUES

```
# example snippets

ui <- fluidPage(
 textInput("a","","A")
)

server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}
```

***Input() functions**
(see front page)

reactiveValues(...)

Each input function creates a reactive value stored as **input$<inputId>**

reactiveValues() creates a list of reactive values whose values you can set.

RENDER REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

render*() functions
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**

PREVENT REACTIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

TRIGGER ARBITRARY CODE

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <-
function(input,output){
 observeEvent(input$go,{
  print(input$a)
 })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

MODULARIZE REACTIONS

```
ui <- fluidPage(
 textInput("a","","A"),
 textInput("z","","Z"),
 textOutput("b"))

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input$z)})
 output$b <- renderText({
  re()
 })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain)

Creates a **reactive expression** that

- **caches** its value to reduce computation
- can be called by other code
- notifies its dependencies when it ha been invalidated

Call the expression with function syntax, e.g. `re()`

DELAY REACTIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
  re()
 })
}

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, domain, ignoreNULL)

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

RStudio

UI - An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a","")
)
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##    <label for="a"></label>
##    <input id="a" type="text"
##      class="form-control" value=""/>
##  </div>
## </div>
```

Returns HTML

HTML

Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

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$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

The most common tags have wrapper functions. You do not need to prefix their names with **tags$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>")
)
```

Header 1

bold

italic

`code`

link

Raw html

CSS

To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

JS

To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$script(src = "<file name>"))
```

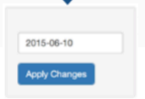
IMAGES

To include an image
1. Place the file in the **www** subdirectory
2. Link to it with `img(src="<file name>")`

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

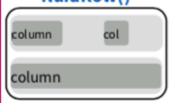
```
wellPanel(dateInput("a", ""),
          submitButton()
)
```



absolutePanel()	navlistPanel()
conditionalPanel()	sidebarPanel()
fixedPanel()	tabPanel()
headerPanel()	tabsetPanel()
inputPanel()	titlePanel()
mainPanel()	wellPanel()

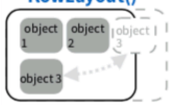
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()



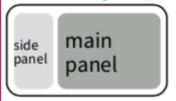

```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()



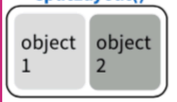

```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()



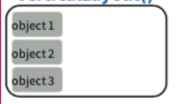

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()




```
ui <- fluidPage(
  splitLayout( # object 1,
               # object 2
  )
)
```

verticalLayout()




```
ui <- fluidPage(
  verticalLayout( # object 1,
                  # object 2,
                  # object 3
  )
)
```

Layer tabPanels on top of each other, and navigate between them, with:

```
ui <- fluidPage(tabsetPanel(
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents")))
```

```
ui <- fluidPage(navlistPanel(
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents")))
```

```
ui <- navbarPage(title = "Page",
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents"))
```

Tag Examples

```
ui <- fluidPage(
  p("This is a sentence with", strong("bold text"),
    "and", a("a link", href="http://www.google.com")
  ),
  hr(),
  h1("Header 1"),
  h2("Header 2"),
  tags$ul(tags$li("item 1"),
          tags$li("item 2"))
)
```

```
ui <- fluidPage(
   titlePanel("Old Faithful Geyser Data"),
   h3(em("Built-in Data")),
   sidebarLayout(
      sidebarPanel(
         sliderInput("bins", "Number of bins:",
                     min = 1, max = 50, value = 30)
      ),
      mainPanel(
        plotOutput("distPlot"),
        tags$caption("Here is my plot caption")
      )
   )
)
```

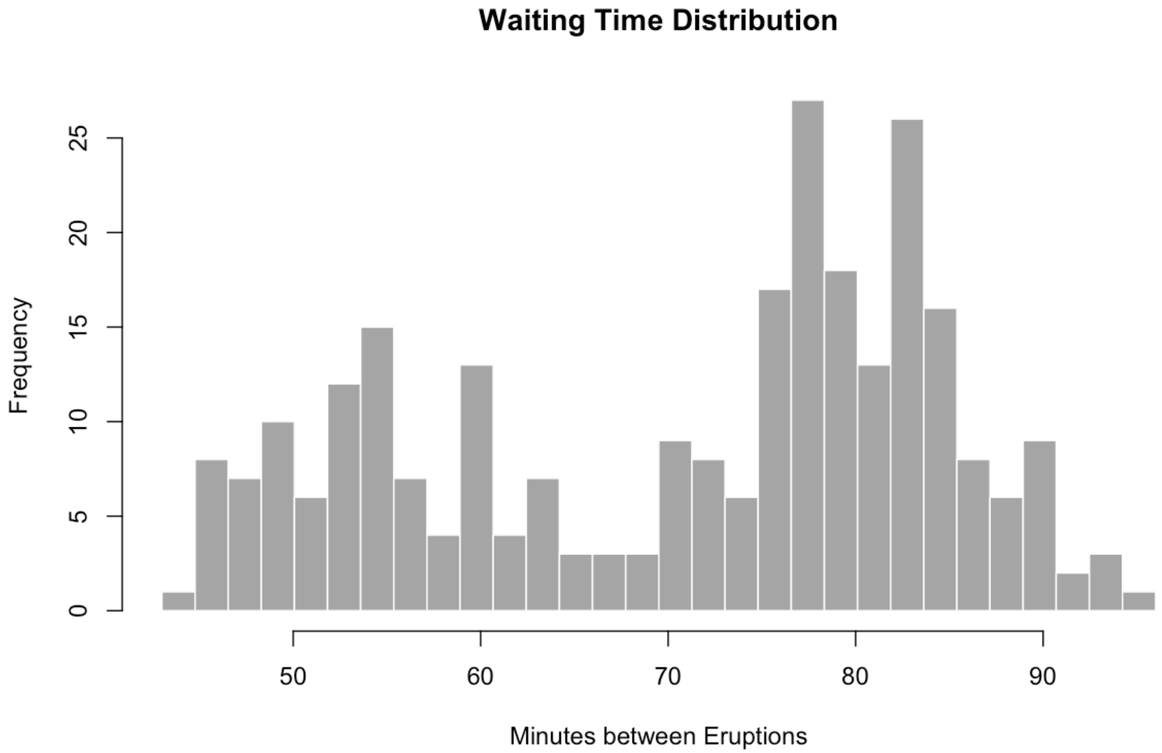
Old Faithful Geyser Data

Built-in Data

Number of bins:

1 30 50

1 6 11 16 21 26 31 36 41 46 50



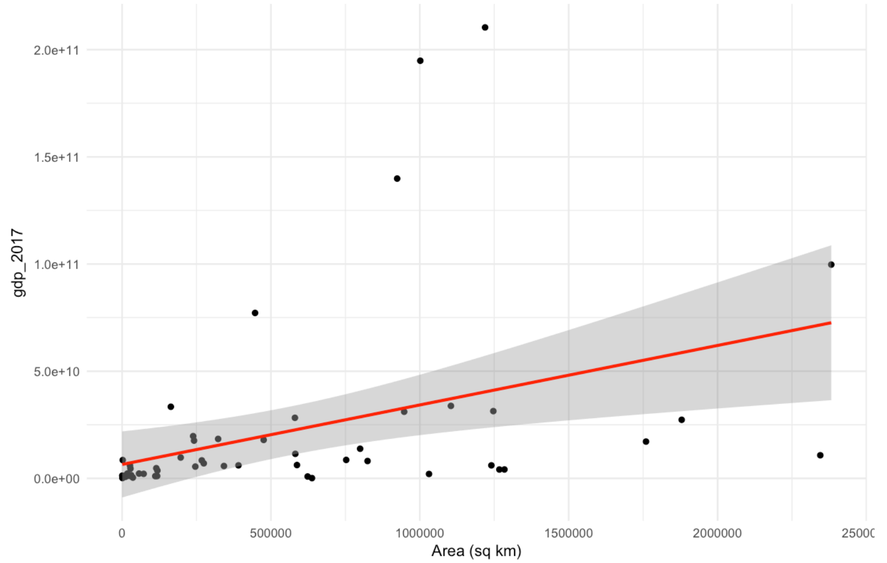
Here is my plot caption

Exercise 2E

Africa: Country Size

Y-axis:

GDP



Data Source: African Development Bank Statistical Data Portal

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

	country	region	gdp_2017	pop_2017	area	rail	road
1	Algeria	north	99722535234	41063753	2381740	4691	113655

Exercise 2E

04:00

exercises/exercise2/app.R *or* answer2c.R

Add a horizontal rule `hr()` between the `mainPanel` and `verticalLayout`.

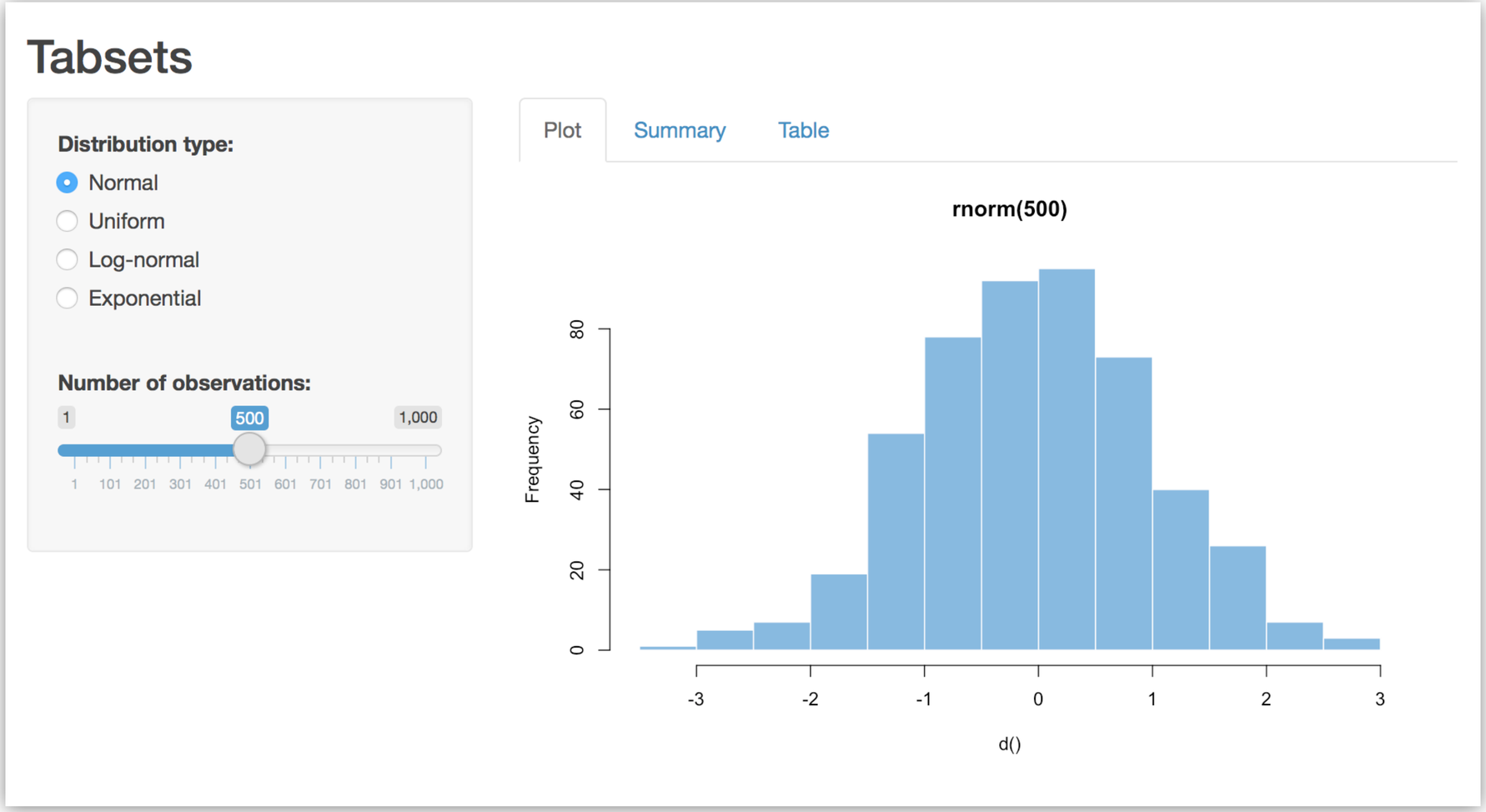
Add a link to the source of the data:

African Development Bank Statistical Data Portal
http://dataportal.opendataforafrica.org

Hint: There's `tags$caption`. You'll also need an `a` tag for a link.

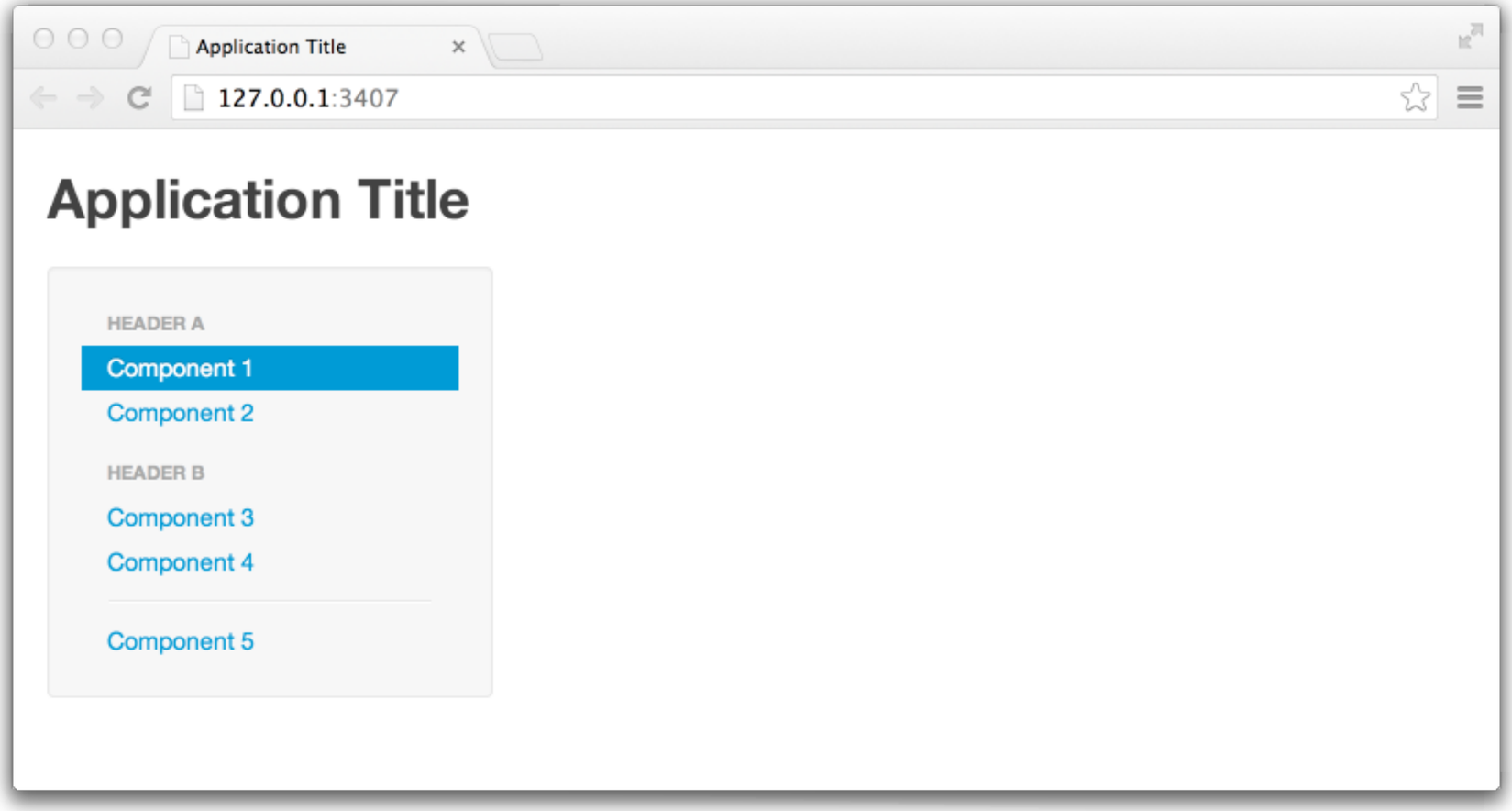
```
mainPanel(
  plotOutput("plot1", brush="plot1_brush"),
  tags$caption("Data Source: ",
               a("African Development Bank Statistical Data Portal",
                 href="http://dataportal.opendataforafrica.org"))
)
), # ends sidebar layout
hr(),
verticalLayout(DT::dataTableOutput("datatable1"))
```

Navigation: tabsetPanel



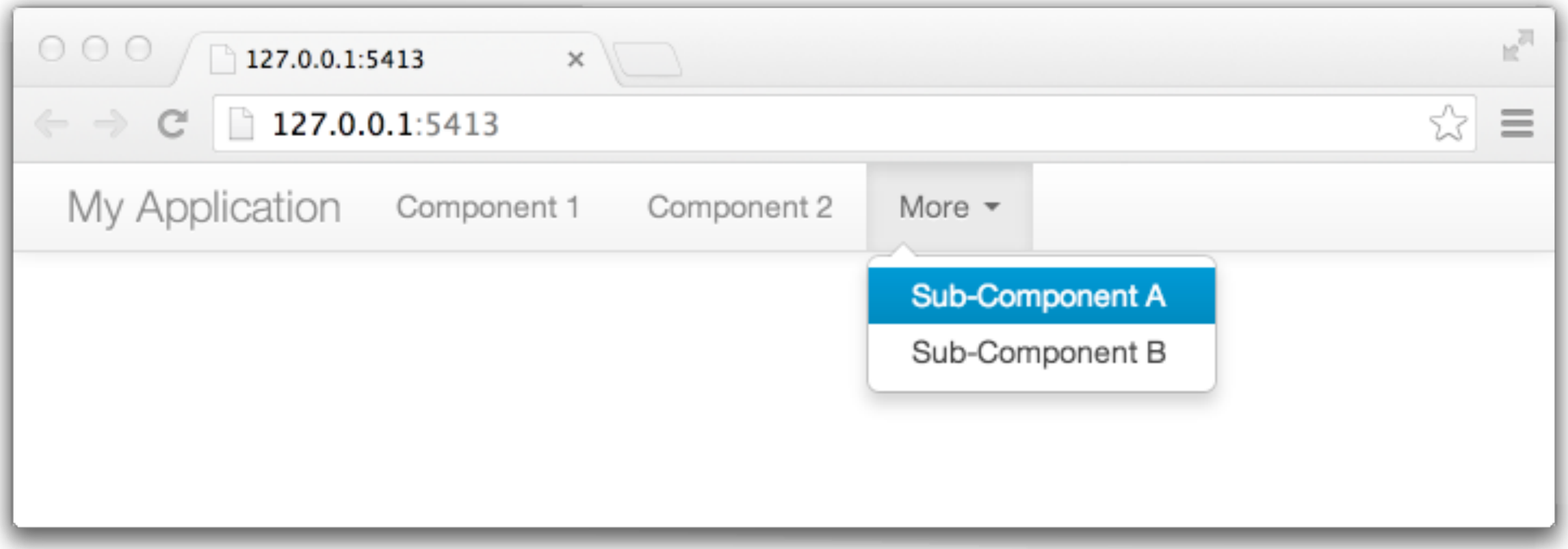
https://shiny.rstudio.com/gallery/tabsets.html

Navigation: navlistPanel



https://shiny.rstudio.com/gallery/navlistpanel-example.html

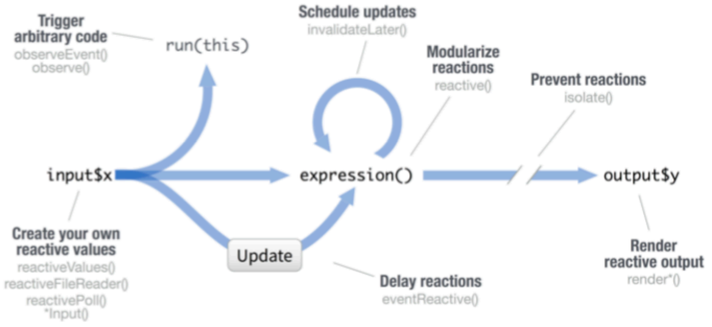
Navigation: navbarPage



https://shiny.rstudio.com/gallery/navbar-example.html

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error `Operation not allowed without an active reactive context.`



CREATE YOUR OWN REACTIVE VALUES

```
# example snippets

ui <- fluidPage(
 textInput("a","","A")
)

server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}
```

***Input() functions**
(see front page)

reactiveValues(...)

Each input function creates a reactive value stored as **input$<inputId>**

reactiveValues() creates a list of reactive values whose values you can set.

RENDER REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

render*() functions
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**

PREVENT REACTIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

TRIGGER ARBITRARY CODE

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <-
function(input,output){
 observeEvent(input$go,{
  print(input$a)
 })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

MODULARIZE REACTIONS

```
ui <- fluidPage(
 textInput("a","","A"),
 textInput("z","","Z"),
 textOutput("b"))

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input$z)})
 output$b <- renderText({
  re()
 })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain)

Creates a **reactive expression** that

- **caches** its value to reduce computation
- can be called by other code
- notifies its dependencies when it ha been invalidated

Call the expression with function syntax, e.g. `re()`

DELAY REACTIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
  re()
 })
}

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, domain, ignoreNULL)

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

UI - An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a","")
)
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##    <label for="a"></label>
##    <input id="a" type="text"
##      class="form-control" value=""/>
##  </div>
## </div>
```

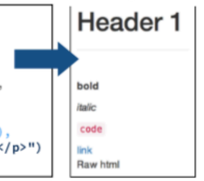
Returns HTML

Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

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$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

The most common tags have wrapper functions. You do not need to prefix their names with **tags$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>")
)
```



To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$script(src = "<file name>"))
```

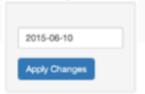
IMAGES

To include an image
1. Place the file in the **www** subdirectory
2. Link to it with `img(src="<file name>")`

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

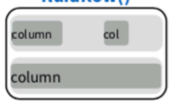
```
wellPanel(dateInput("a", ""),
          submitButton()
)
```



absolutePanel()	navlistPanel()
conditionalPanel()	sidebarPanel()
fixedPanel()	tabPanel()
headerPanel()	tabsetPanel()
inputPanel()	titlePanel()
mainPanel()	wellPanel()

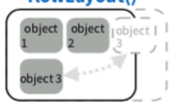
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()



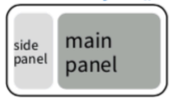

```
ui <- fluidPage(
 fluidRow(column(width = 4),
   column(width = 2, offset = 3)),
 fluidRow(column(width = 12))
)
```

flowLayout()



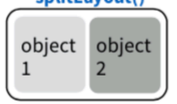

```
ui <- fluidPage(
 flowLayout( # object 1,
             # object 2,
             # object 3
 )
)
```

sidebarLayout()



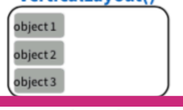

```
ui <- fluidPage(
 sidebarLayout(
   sidebarPanel(),
   mainPanel()
 )
)
```

splitLayout()




```
ui <- fluidPage(
 splitLayout( # object 1,
              # object 2
 )
)
```

verticalLayout()




```
ui <- fluidPage(
 verticalLayout( # object 1,
                 # object 2,
                 # object 3
 )
)
```

Layer tabPanels on top of each other, and navigate between them, with:

```
ui <- fluidPage(tabsetPanel(
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents")))
```

```
ui <- fluidPage(navlistPanel(
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents")))
```

```
ui <- navbarPage(title = "Page",
  tabPanel("tab 1","contents"),
  tabPanel("tab 2","contents"),
  tabPanel("tab 3","contents"))
```

Exercise 2F

exercises/exercise2/exercise2f.R

Put a `tabsetPanel` inside the `mainPanel` with 2 tabs:

- "All" `tabPanel` containing the existing `plotOutput()`
- "By Region" `tabPanel` containing a new `plotOutput` with `outputId="regionplot"`
  - There is already code in the server section to render `output$regionplot`

Refer to documentation as needed.

Shiny Themes

https://rstudio.github.io/shinythemes/

Exercise 2G

03:00

exercises/exercise2/ — any working app

Go to the Shiny themes page:
https://rstudio.github.io/shinythemes/

Find a Shiny theme you like.
Or, try the `shinythemes::themeSelector()`

Apply it to the app.

Structure of Shiny Applications
UI: Layout, Inputs and Outputs
Server
Bringing it Together
Design Elements
Interactive Graphics
Sharing Your Shiny Application

Selecting Plot Points

```
plotOutput(outputId, width = "100%", height = "400px", click = NULL,
  dblclick = NULL, hover = NULL, hoverDelay = NULL,
  hoverDelayType = NULL, brush = NULL, clickId = NULL, hoverId = NULL,
  inline = FALSE)
```

click

dblclick

hover

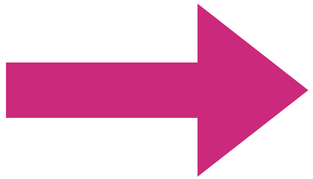
brush

Demo

examples/104-plot-interaction-select

click

```
plotOutput(outputId = "plot1",
    click = "plot1_click"
)
```

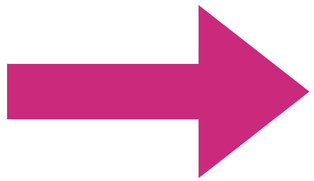


input$plot1_click

- input$plot1_click$x
- input$plot1_click$y

Click Helper Function

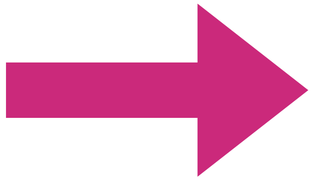
```
nearPoints(dataframe_name,
           input$plot1_click,
           xvar = "x_variable_name",
           yvar = "y_variable_name")
```



Subset of the data frame (or NULL)

Brush

```
plotOutput(outputId = "plot1",
    brush = "plot1_brush"
)
```

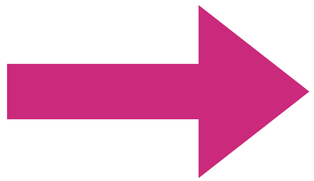


`input$plot1_brush`

- `input$plot1_brush$xmin`
- `input$plot1_brush$xmax`
- `input$plot1_brush$ymin`
- `input$plot1_brush$ymax`

Brush Helper Function

```
brushedPoints(dataframe_name,
              input$plot1_brush,
              xvar = "x_variable_name",
              yvar = "y_variable_name")
```



Subset of the data frame (or NULL)

Brush Helper Function

```
brushedPoints(dataframe_name,
              input$plot1_brush,
              xvar = "x_variable_name",
              yvar = input$selectedvar)
```

Demo Code

Exercise 2C

exercises/exercise2/app.R *or* answer2b.R

- Set the `brush` argument to `plotOutput` to "plot1_brush"
- Change `DT::renderDataTable()` to use output from `brushedPoints()` instead of using `data` directly

```
mainPanel(
    plotOutput("plot1", brush="plot1_brush")
)



output$datatable1 <- DT::renderDataTable(
    brushedPoints(data, input$plot1_brush,
                  xvar="area", yvar=input$yval)
)
```

Structure of Shiny Applications
UI: Layout, Inputs and Outputs
Server
Bringing it Together
Design Elements
Interactive Graphics
Sharing Your Shiny Application

Sharing Your Application

Share your Files

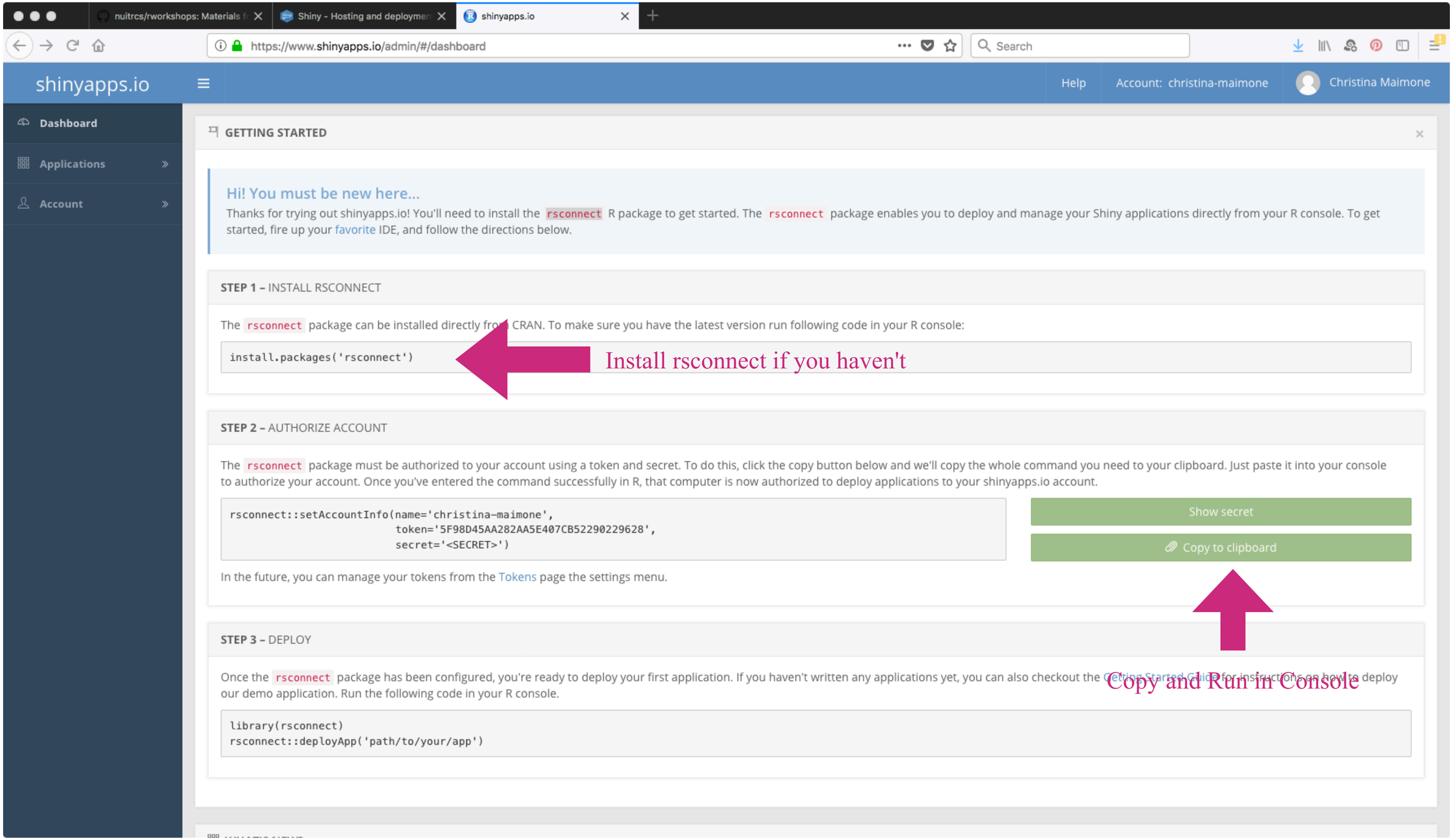
Other R Users can run your app locally

Deploy to a Server

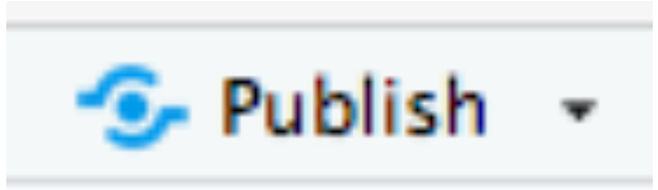
shinyapps.io or a custom server

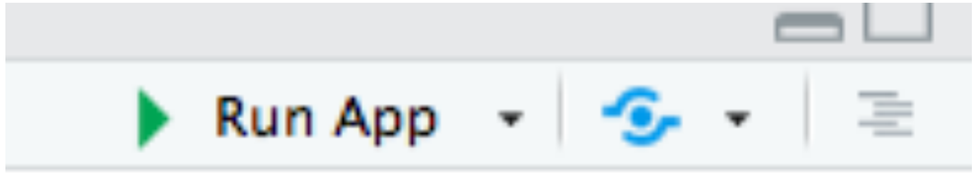
http://shinyapps.io

Make an Account or Sign In


nuitrcs/rworkshops: Materials
Shiny - Hosting and deploymen
shinyapps.io
https://www.shinyapps.io/admin/#/dashboard
Search
shinyapps.io
Help
Account: christina-maimone
Christina Maimone
Dashboard
Applications
Account
GETTING STARTED
Hi! You must be new here...
Thanks for trying out shinyapps.io! You'll need to install the rsconnect R package to get started. The rsconnect package enables you to deploy and manage your Shiny applications directly from your R console. To get started, fire up your favorite IDE, and follow the directions below.
STEP 1 – INSTALL RSCONNECT
The rsconnect package can be installed directly from CRAN. To make sure you have the latest version run following code in your R console:
install.packages('rsconnect')
Install rsconnect if you haven't
STEP 2 – AUTHORIZE ACCOUNT
The rsconnect package must be authorized to your account using a token and secret. To do this, click the copy button below and we'll copy the whole command you need to your clipboard. Just paste it into your console to authorize your account. Once you've entered the command successfully in R, that computer is now authorized to deploy applications to your shinyapps.io account.
rsconnect::setAccountInfo(name='christina-maimone',
token='5F98D45AA282AA5E407CB52290229628',
secret='<SECRET>')
Show secret
Copy to clipboard
In the future, you can manage your tokens from the Tokens page the settings menu.
STEP 3 – DEPLOY
Once the rsconnect package has been configured, you're ready to deploy your first application. If you haven't written any applications yet, you can also checkout the Getting Started Guide for instructions on how to deploy our demo application. Run the following code in your R console.
Copy and Run in Console
library(rsconnect)
rsconnect::deployApp('path/to/your/app')

shinyapps.io






```
rsconnect::deployApp('exercises/exercise3')
```

How Do I Get Started?

Use the Shiny website:
Lots of well-written documentation and articles

Explore the gallery:
Find an app, get the code, and modify it

That's It

Questions?

Christina Maimone

christina.maimone@northwestern.edu

@maimonecr

Extras

Reactivity

How does this all work?

~/training/rshiny/exercises/exercise1 - Shiny

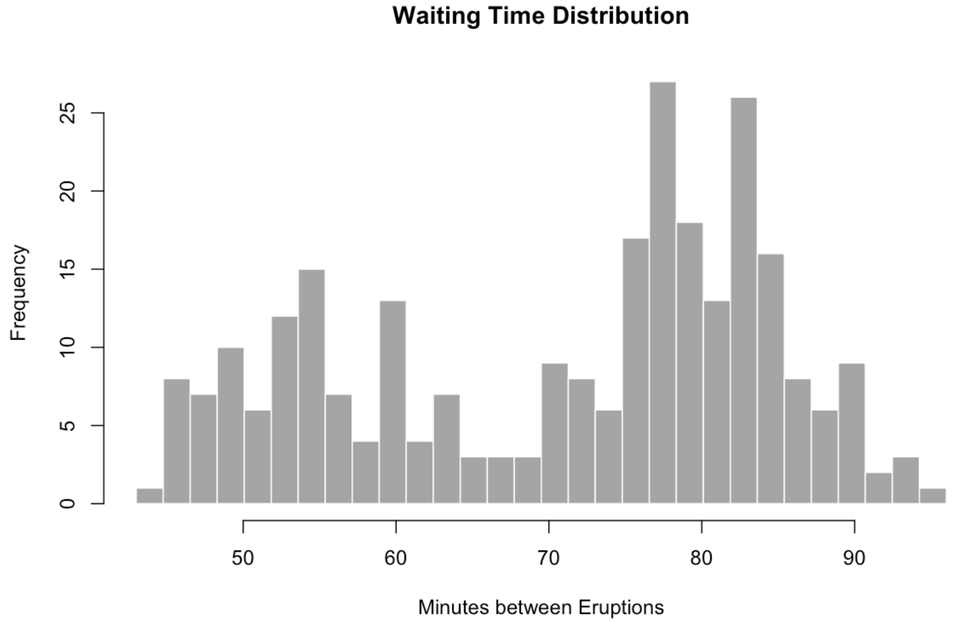
http://127.0.0.1:5859 | Open in Browser | Publish

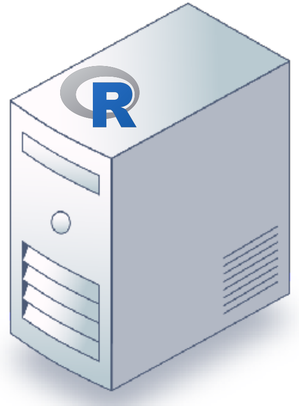
Old Faithful Geyser Data

Number of bins:

1 30 50

1 6 11 16 21 26 31 36 41 46 50





Shiny Server

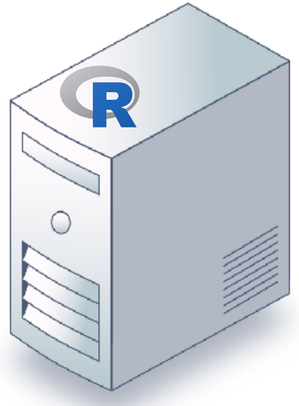
output$distplot

I see you're 30.
I need to know when you change. Send this pigeon.

`input$bins` ← `output$distplot`

30

https://shiny.rstudio.com/articles/understanding-reactivity.html



Shiny Server

Something you need changed

`output$distplot`

`input$bins`

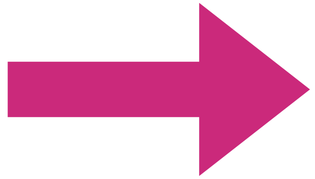
`output$distplot`

30

https://shiny.rstudio.com/articles/understanding-reactivity.html

Reactive Sources and Endpoints

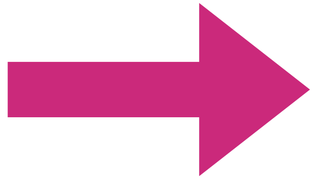
`input$_____`



Reactive Source

Sends signals

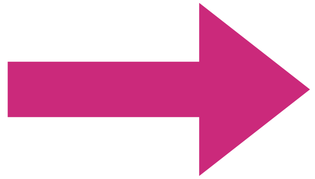
`output$_____`

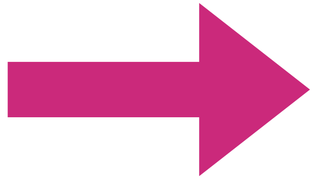


Reactive Endpoint

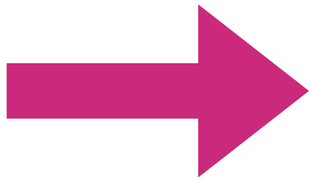
1. Told to re-execute
2. Request values from sources

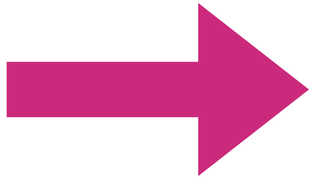
Reactive Sources and Endpoints

`input$_____`  Reactive Value

`output$_____`  Observer

Reactive Sources and Endpoints

`input`  `reactiveValues`

`input$_____`  `reactiveVal`

Reactivity: Next Steps

- More complicated relationships

  ➡ Reactive expressions

- Interruptions to normal reactive flow
- Triggering events explicitly

Verifying and Validating

Verifying Input

The `req()` function specifies requirements for the code that follows to work.

Arguments are the values that should exist or conditional tests.

It stops execution of the expression (code block) if the conditions aren't met.

Verifying Input

```
output$letters <- renderText({
    req(input$n)
    paste(rep('*', input$n), collapse=' ')
  })
```

Verifying Input

```
output$letters <- renderText({
    req(input$n > 0 & input$n <= 30)
    paste(rep('*', input$n), collapse=' ')
  })
```

Verifying Input

The `validate()` function checks if certain conditions are true, and returns error messages if not.

Can take multiple conditions.

Specify each condition with `need(test, message)`

Verifying Input

```
output$letters <- renderText({
    validate(need(input$n,
                  "Please enter a number"))
    paste(rep('*', input$n), collapse=' ')
  })
```

Verifying Input

```
output$letters <- renderText({
    validate(need(input$n > 0 & input$n <= 30,
                  "Please enter a number between 1 and 30"))
    paste(rep('*', input$n), collapse=' ')
  })
```

Exercise 3A & B

05:00

exercises/exercise3/app.R

- Instructions in the file
- A: Run 2 sessions of the application. Examine the behavior.
  - Laptop users: Copy the URL from upper left and open two web browser tabs or windows with this URL.
  - Cloud users: Find a partner who is running from their laptop.
- B: Make some updates to validate inputs and change behavior

Challenge Exercise 2D

05:00

exercises/exercise2/app.R *or* answer2c.R

Fix the application so that the DataTable doesn't display empty rows for missing data.

- Subset the data returned by `brushedPoints()` to filter out rows with missing values in the `input$yval` column
- Use `req()` to check that there's at least one row in the subset
- Make sure the last line in the {} is the data to display

BONUS

If nothing is highlighted on the plot, show all of the data in the DataTable.

```
output$datatable1 <- DT::renderDataTable({
    req(input$plot1_brush)
    pts <- brushedPoints(data, input$plot1_brush,
                         xvar="area",
                         yvar=input$yval)
    pts <- subset(pts, !is.na(pts[input$yval]))
    req(nrow(pts) > 0)
    pts
})
```

```
output$datatable1 <- DT::renderDataTable({
    if(is.null(input$plot1_brush)) {
      data
    } else {
      req(input$plot1_brush)
      pts <- brushedPoints(data,
                input$plot1_brush, xvar="area",
                yvar=input$yval)
      pts <- subset(pts, !is.na(pts[input$yval]))
      req(nrow(pts) > 0)
      pts
    }
})
```

01:00

03:00

05:00

10:00