

# Introduction to Shiny Introduction to Data Science

Joseph N Paulson

paulson.joseph@gene.com

# What is Shiny?

Shiny is an R package developed by RStudio that allows one to create interactive websites and visualizations.

- Overview:
  - <http://shiny.rstudio.com/>
- Gallery:
  - <http://shiny.rstudio.com/gallery/>
- Documentation:
  - <http://shiny.rstudio.com/articles/>
  - <http://shiny.rstudio.com/reference/shiny/>
- Deployment:
  - <http://shiny.rstudio.com/deploy/>
- Cheat sheet:
  - [Link here](#)



For this presentation, shiny is not a shiny art installation

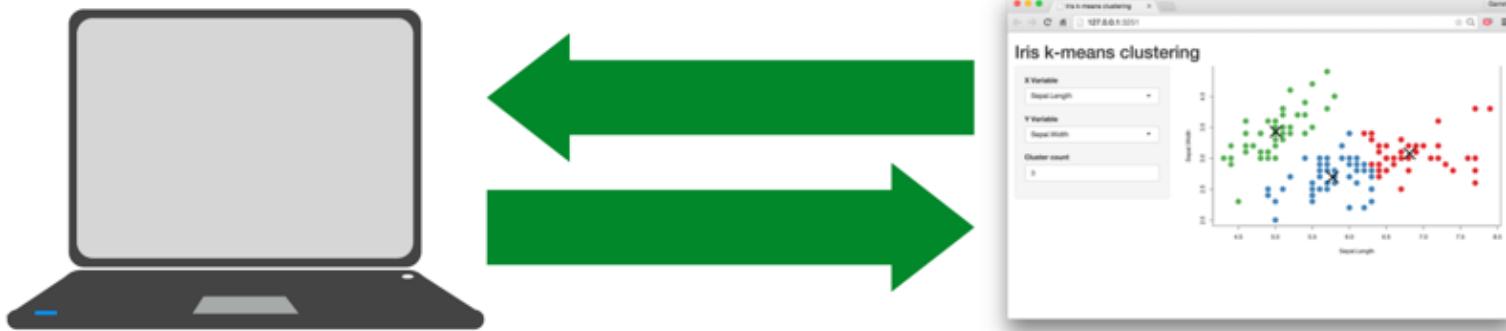
# Examples

- A few examples we will go over (Lots of examples in [the gallery](#))
  - Simple
    - [Malnutrition visualization example](#) – [Github Code](#)
    - <http://shiny.rstudio.com/gallery/single-file-shiny-app.html>
    - <http://shiny.rstudio.com/gallery/kmeans-example.html>
  - Slightly more advanced
    - [Diarrheal microbiome cohort](#) – [Github Code](#)
    - <http://shiny.rstudio.com/gallery/movie-explorer.html>

# Shiny overview



Every Shiny app is maintained by a computer (laptop, desktop or server) running R





Server computer  
running R and Shiny



A Web browser

# Template

- Lets create an R app through RStudio.
- It'll come with a ton of stuff.
- Lets remove it and replace with a minimalist product that will produce an empty (visually) webpage.

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

# Just Do It.

The following can be put within a single file, eg.  
**app.R** the ui and server portions can be separated into two files:  
**ui.R** and **server.R**

and run locally with the **runApp** function.

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

# Assessment – run this

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

# Overview - 1

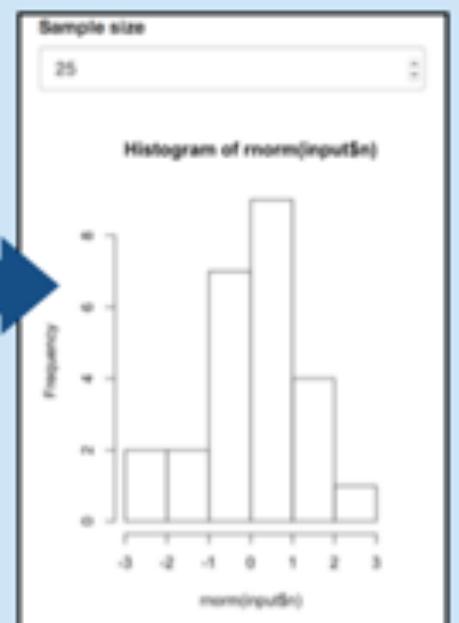
Add inputs to the UI with **\*Input()** functions

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)
```



# Overview - 2

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")
)
```

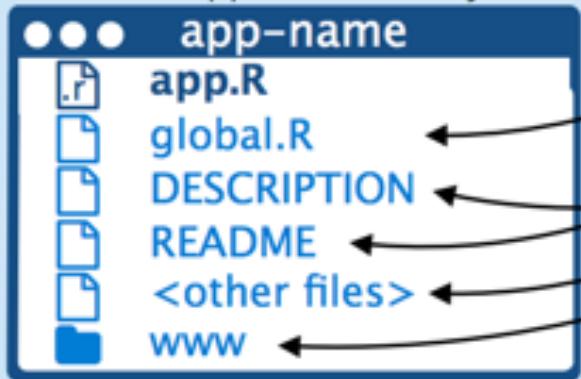
```
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

**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 contains 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>)**

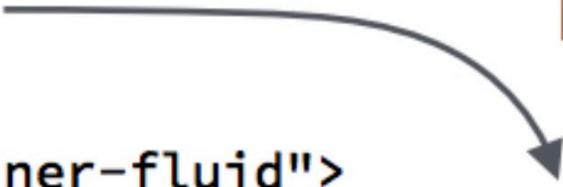
# UI Breakdown

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

# Digging a bit deeper.

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

# UI - Inputs

Inputs for the shiny app go within the ui component of the shiny app.

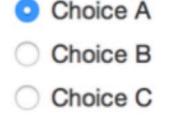
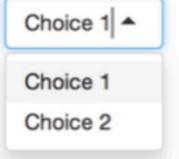
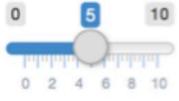
They create the sliders, radio buttons, and clickable values for the HTML user interface.

The values from the webpage (UI) are transferred to R in a list called 'input'.

The values can be extracted and processed through R and are available as `input$<inputID>`.

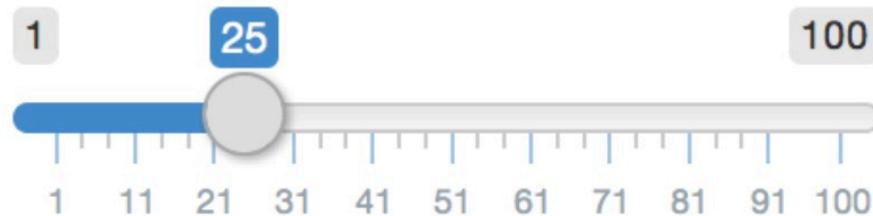
**Inputs - collect values from the user**

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

Action	<code>actionButton(inputId, label, icon, ...)</code>
Link	<code>actionLink(inputId, label, icon, ...)</code>
<input checked="" type="checkbox"/> Choice 1 <input checked="" type="checkbox"/> Choice 2 <input type="checkbox"/> Choice 3  <input checked="" type="checkbox"/> Check me	<code>checkboxGroupInput(inputId, label, choices, selected, inline)</code>  <code>checkboxInput(inputId, label, value)</code>
	<code>dateInput(inputId, label, value, min, max, format, startview, weekstart, language)</code>  <code>dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)</code>
<code>Choose File</code>	<code>fileInput(inputId, label, multiple, accept)</code>
	<code>numericInput(inputId, label, value, min, max, step)</code>
	<code>passwordInput(inputId, label, value)</code>
	<code>radioButtons(inputId, label, choices, selected, inline)</code>
	<code>selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())</code>
	<code>sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)</code>
	<code>submitButton(text, icon)</code> (Prevents reactions across entire app)
	<code>textInput(inputId, label, value)</code>

# Syntax

**Choose a number**



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

input name  
(for internal use)

Notice:  
Id not ID

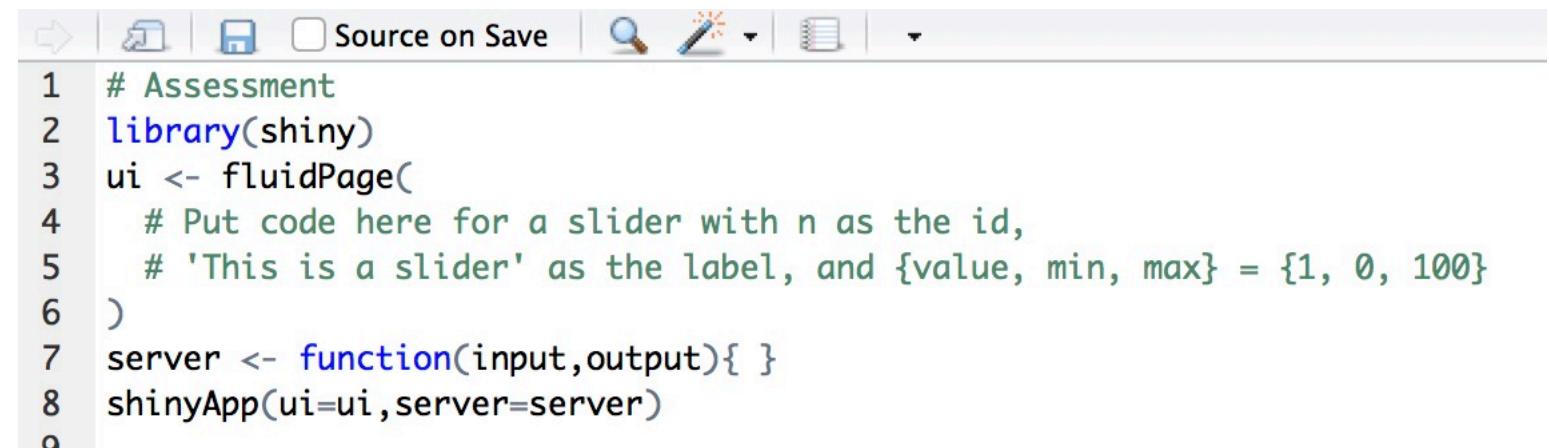
label to  
display

input specific  
arguments

?sliderInput

# Assessment – UI inputs

Extend the basic template to create a **sliderInput** with a starting value of 1, min of 0 and max of 100.

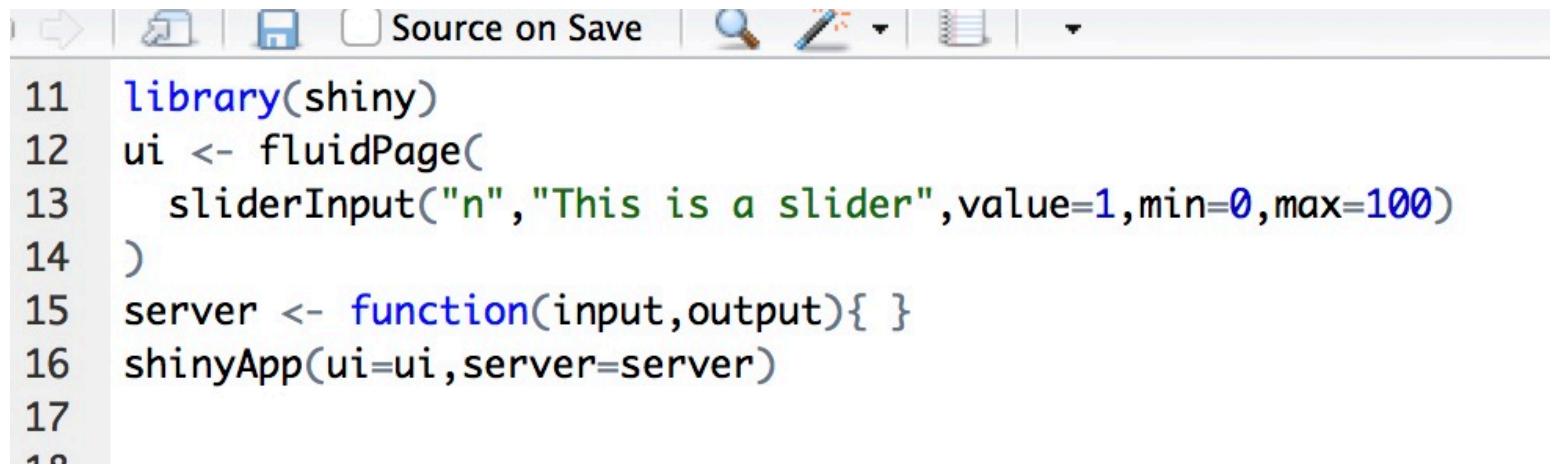


The image shows a screenshot of the RStudio IDE. The top menu bar includes options like File, Edit, View, Insert, Tools, Session, Help, and a Source on Save checkbox. Below the menu is a toolbar with icons for back, forward, file operations, and search. The main workspace contains the following R code:

```
1 # Assessment
2 library(shiny)
3 ui <- fluidPage(
4   # Put code here for a slider with n as the id,
5   # 'This is a slider' as the label, and {value, min, max} = {1, 0, 100}
6 )
7 server <- function(input,output){}
8 shinyApp(ui=ui,server=server)
```

# Solution

Extend the basic template to create a **sliderInput**.



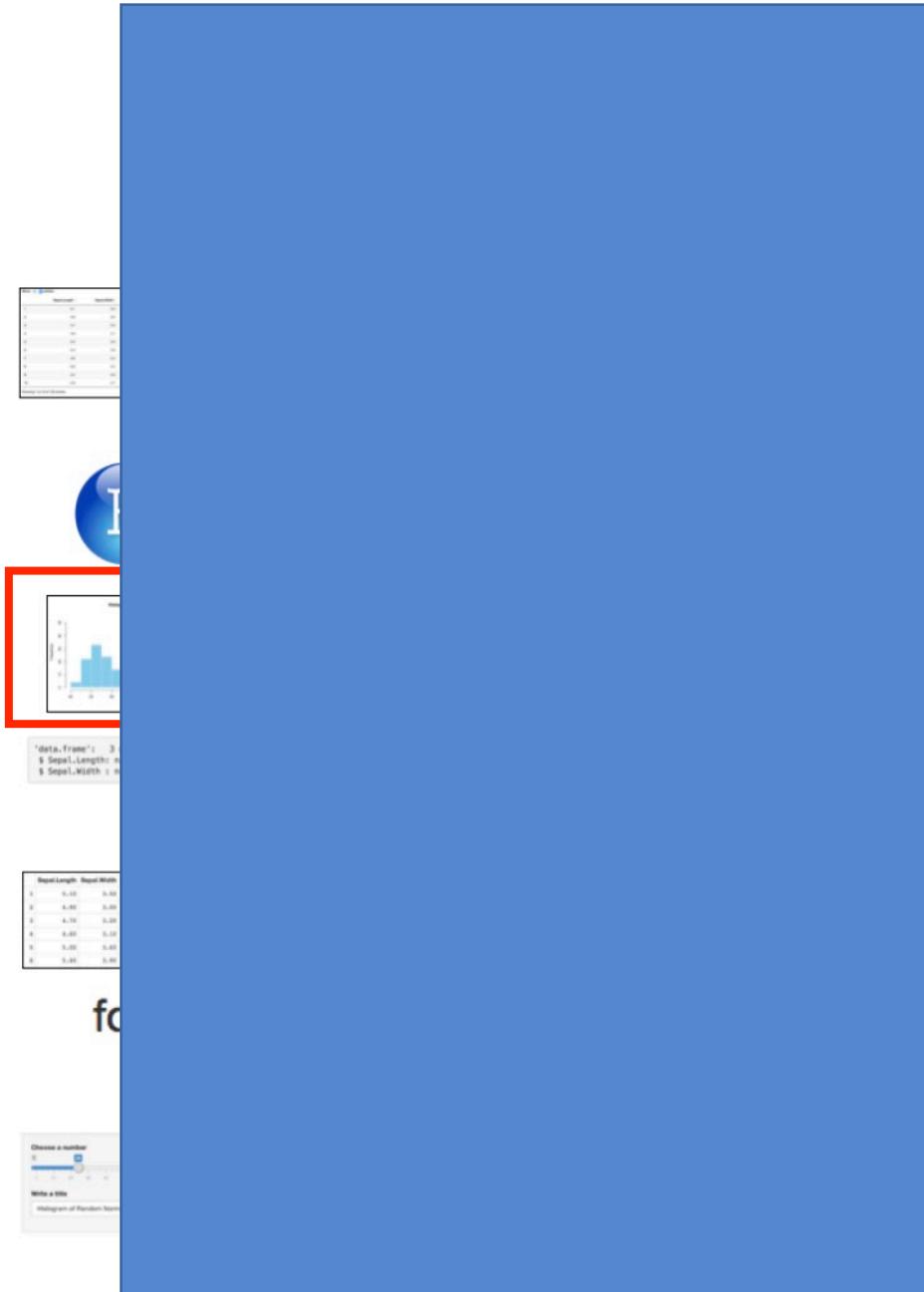
The screenshot shows a code editor window in RStudio. The title bar includes standard icons for file operations and a "Source on Save" button. Below the title bar is a toolbar with a magnifying glass icon for search and a pencil icon for edit. The main area contains the following R code:

```
11 library(shiny)
12 ui <- fluidPage(
13   sliderInput("n","This is a slider",value=1,min=0,max=100)
14 )
15 server <- function(input,output){}
16 shinyApp(ui=ui,server=server)
```

# UI – Output

Other pieces go within the UI specifically to display the 'server' processed information.

The **\*Output** functions take the output of functions rendered within the server. The rendered values are transferred from the server in **output\$outputID** and rendered in the webpage (UI) using the **\*Output** functions.



Put into **ui** function

**dataTableOutput(outputId, icon, ...)**

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

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

**verbatimTextOutput(outputId)**

**tableOutput(outputId)**

**textOutput(outputId, container, inline)**

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

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

# \*Output()

To display output, add it to `fluidPage()` with an  
`*Output()` function

```
plotOutput(outputId = "plot")
```

the type of output  
to display

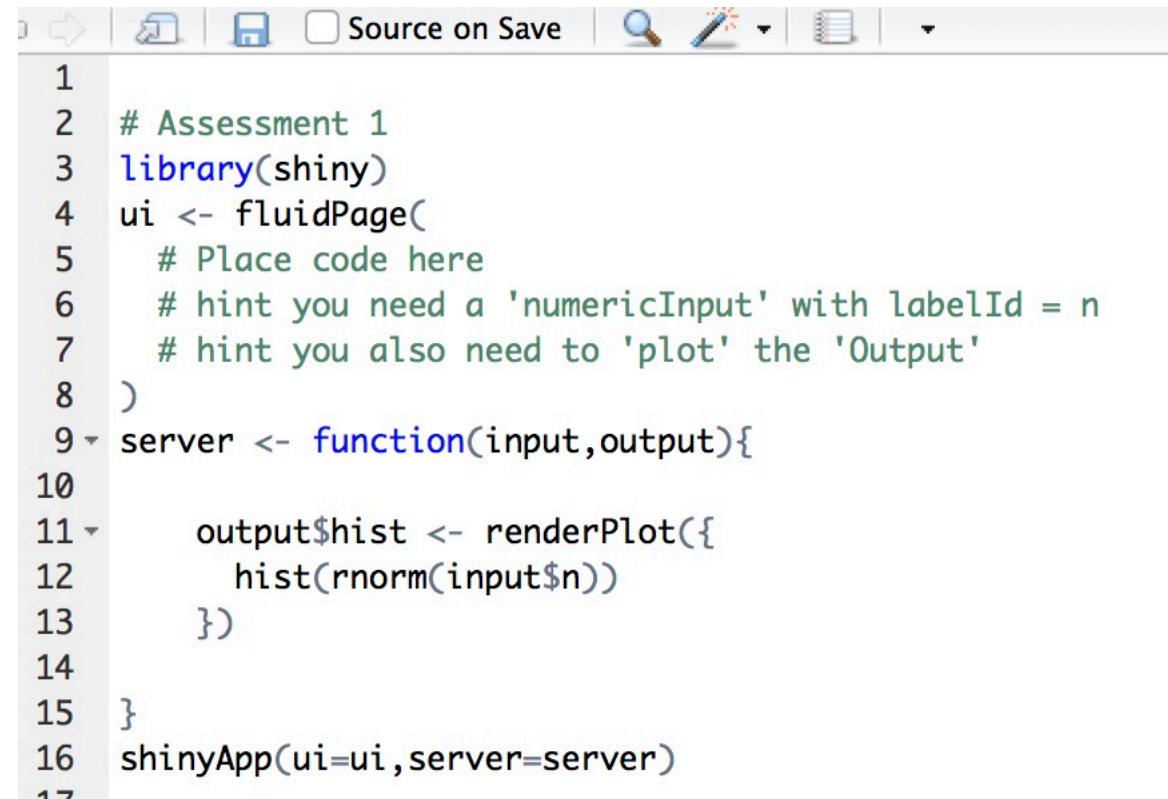
name to give to the  
output object

# Assessment - UI output

Create a Shiny App using the following template to create a histogram of a user-defined number of draws from a normal distribution (or any distribution).

The general structure for the ui and server are in place as is the code required for plotting the histogram and drawing the 'n' draws.

In particular, you only need to prepare the plot. We only need to create a **numericInput** that will say how many draws to make from a distribution and a **plotOutput** that will plot the **histogram**.

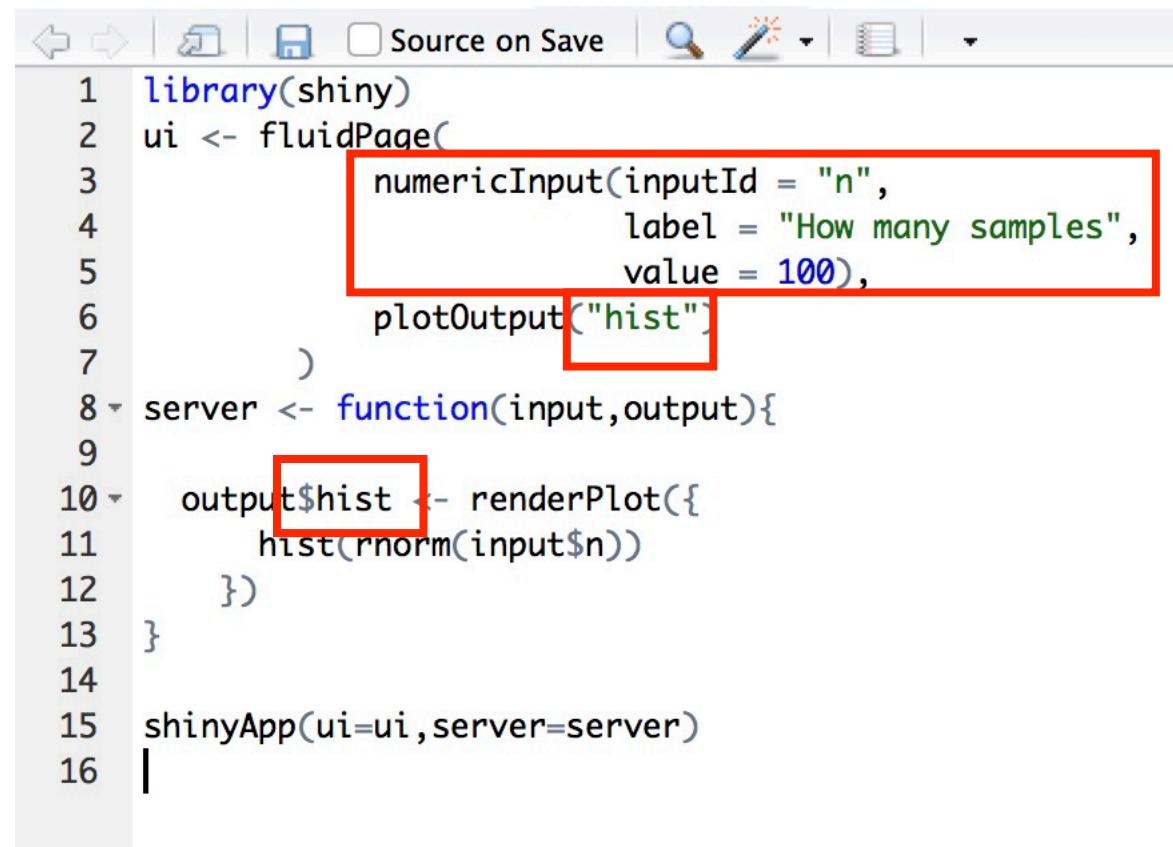


A screenshot of the RStudio interface showing a code editor window. The window contains R code for a Shiny application. The code is numbered from 1 to 17. It includes a header comment, the library function for shiny, the creation of a fluidPage, and a server function that uses renderPlot to create a histogram of rnorm draws based on user input. The RStudio toolbar at the top includes icons for file operations, source control, and search.

```
1 # Assessment 1
2 library(shiny)
3 ui <- fluidPage(
4   # Place code here
5   # hint you need a 'numericInput' with labelId = n
6   # hint you also need to 'plot' the 'Output'
7 )
8 server <- function(input,output){
9   output$hist <- renderPlot{
10     hist(rnorm(input$n))
11   }
12 }
13 shinyApp(ui=ui,server=server)
```

# Solution

We added a **numericInput** to say how many draws to make from a distribution (eg. **rnorm**) and provided the **hist** function within **output\$hist** the **input\$n** normal distribution draws.



The screenshot shows the RStudio interface with the code editor open. The code is a Shiny application (app.R) with the following structure:

```
1 library(shiny)
2 ui <- fluidPage(
3   numericInput(inputId = "n",
4               label = "How many samples",
5               value = 100),
6   plotOutput("hist")
7 )
8 server <- function(input,output){
9
10   output$hist <- renderPlot({
11     hist(rnorm(input$n))
12   })
13 }
14
15 shinyApp(ui=ui,server=server)
16 |
```

Two specific parts of the code are highlighted with red boxes:

- A red box surrounds the **numericInput** block (lines 3-5), which defines an input field for the number of samples.
- A red box surrounds the **renderPlot** call within the **server** function (line 10), which generates a histogram.

# Server Breakdown

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

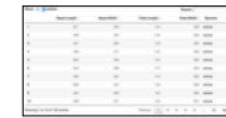
# Server

These pieces go within the **server** function and are the workhorse between taking the hard written functions and display.

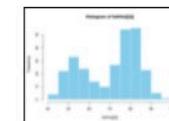
The render functions perform the opposite task of the input functions and grab the output values and converts it into HTML code.

The rendered values are transferred from the server in **output\$outputID** and rendered in the webpage (UI).

Put into **server** portion



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



'data.Frame': 3 obs. of 2 variables:  
# Sepal.Length: num 5.1 4.9 4.7  
# Sepal.Width : num 3.5 3.0 3.2

**renderImage(expr, env, quoted, deleteFile)**

**renderPlot(expr, width, height, res, ..., env,  
quoted, func)**

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

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.40	0.20	Iris-setosa
2	4.90	3.00	1.30	0.20	Iris-setosa
3	4.70	3.20	1.30	0.20	Iris-setosa
4	4.50	3.00	1.30	0.20	Iris-setosa
5	5.00	3.60	1.50	0.20	Iris-setosa
6	5.40	3.90	1.70	0.40	Iris-setosa

foo

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

**renderText(expr, env, quoted, func)**

works  
with

&

# render\*

Builds reactive output to display in UI

```
renderPlot({ hist(iris$Sepal.Length) })
```

type of object to build

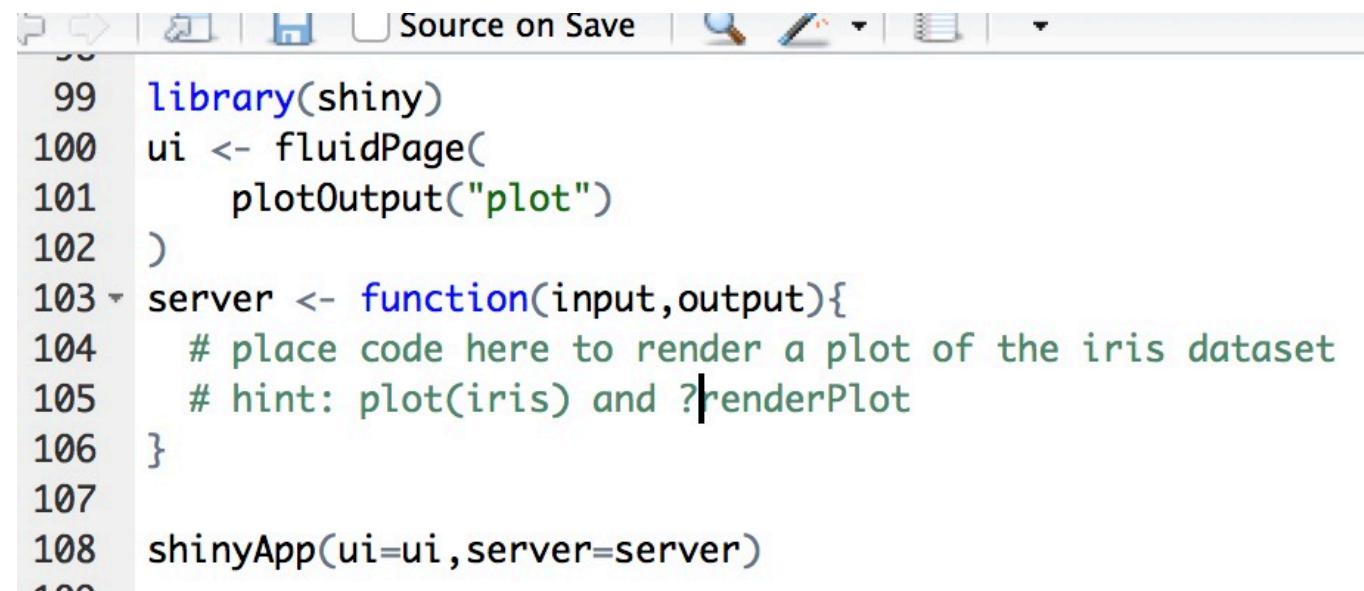
code that builds the object

# Assessment - Render output

Create a Shiny App using the following template to plot the iris dataset given:

The general structure for the ui and server are in place as is the code to plot the output from the server.

In particular, you only need to prepare and **renderPlot** the plot.



A screenshot of a code editor window showing R code. The code is a template for a Shiny application. It includes a header bar with icons for file operations and a 'Source on Save' button. The code itself is numbered from 99 to 108. Lines 99-102 define the UI as a fluid page with a plot output. Line 103 starts the server function, which contains a comment about rendering a plot of the iris dataset and a hint to use `plot(iris)` and `?renderPlot`. Lines 104-107 complete the server function. Line 108 creates the shinyApp object with the defined UI and server.

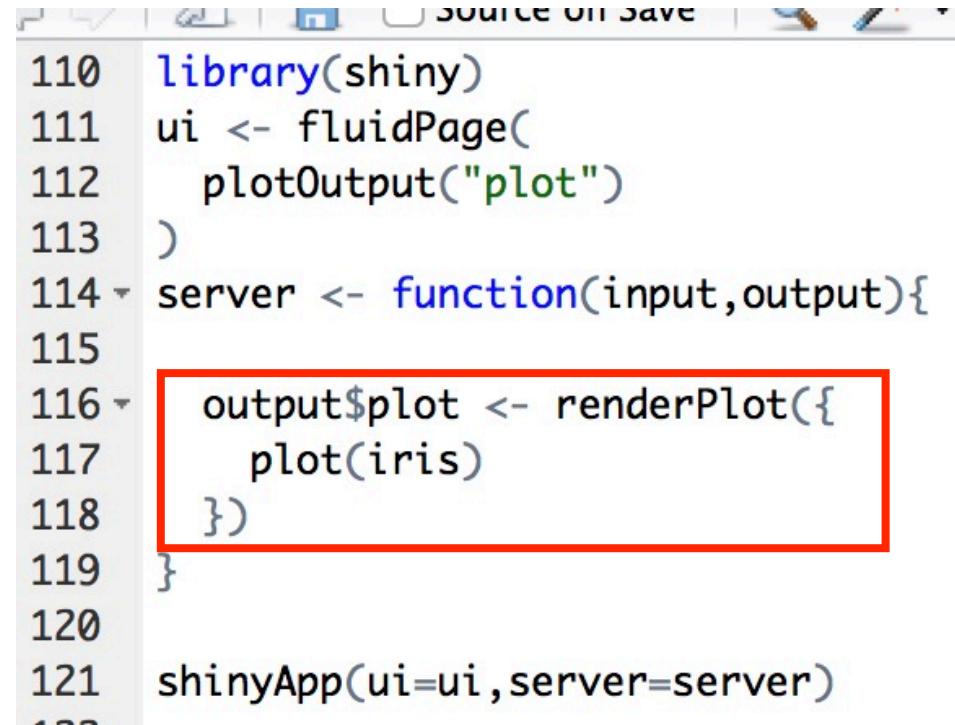
```
99 library(shiny)
100 ui <- fluidPage(
101   plotOutput("plot")
102 )
103 server <- function(input,output){
104   # place code here to render a plot of the iris dataset
105   # hint: plot(iris) and ?renderPlot
106 }
107
108 shinyApp(ui=ui,server=server)
```

# Solution

Create a Shiny App using the following template to plot the iris dataset given:

The general structure for the ui and server are in place as is the code to plot the output from the server.

In particular, you only need to prepare and **renderPlot** the plot.



```
library(shiny)
ui <- fluidPage(
  plotOutput("plot")
)
server <- function(input,output){
  output$plot <- renderPlot({
    plot(iris)
  })
}
shinyApp(ui=ui,server=server)
```

# Reactivity

When the sliders, text, numeric, really any of the inputs are modified the functions within the server *react* to the changed input and re-render

## Reactivity 101

Reactivity automatically occurs whenever you use an input value to render an output object

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

# Reactions

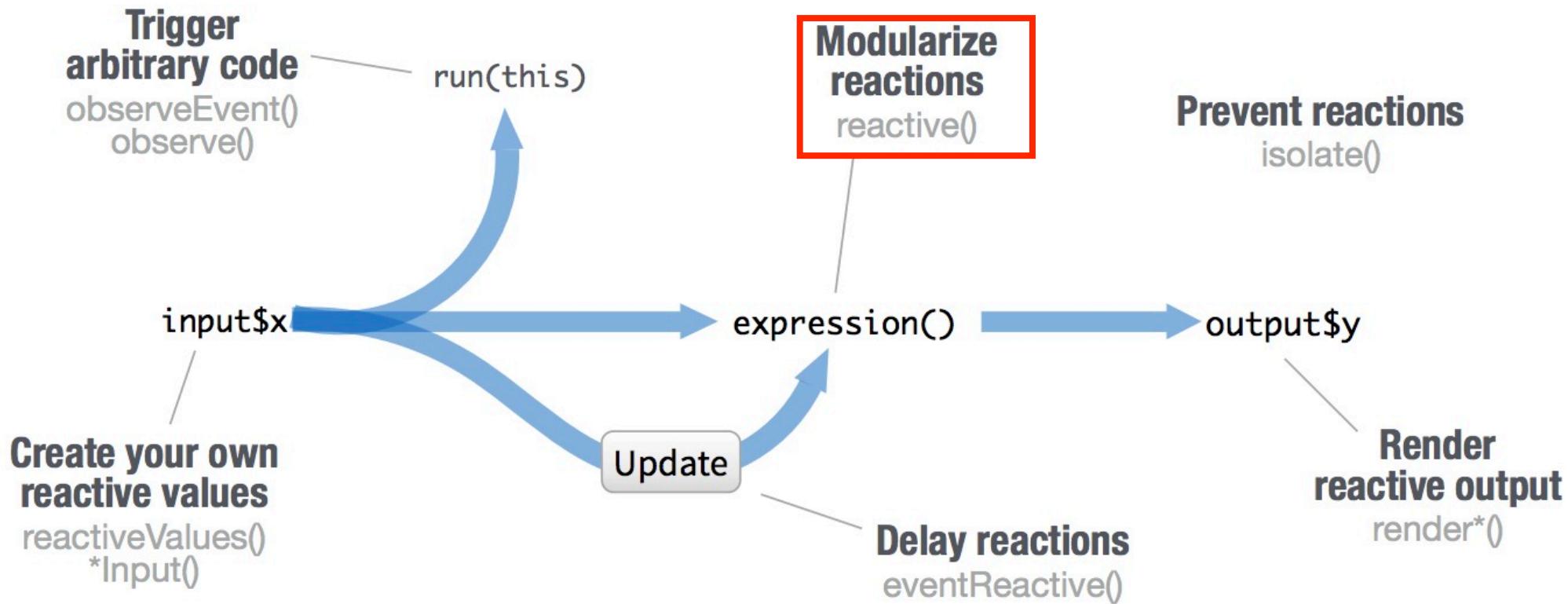
```
library(shiny)
ui <- fluidPage(
  sliderInput("num", "", 1, 100, 25),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
shinyApp(ui = ui, server = server)
```

Reactivity automatically occurs whenever you use an input value to render an output object

An input value

# 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.**



# Reactivity

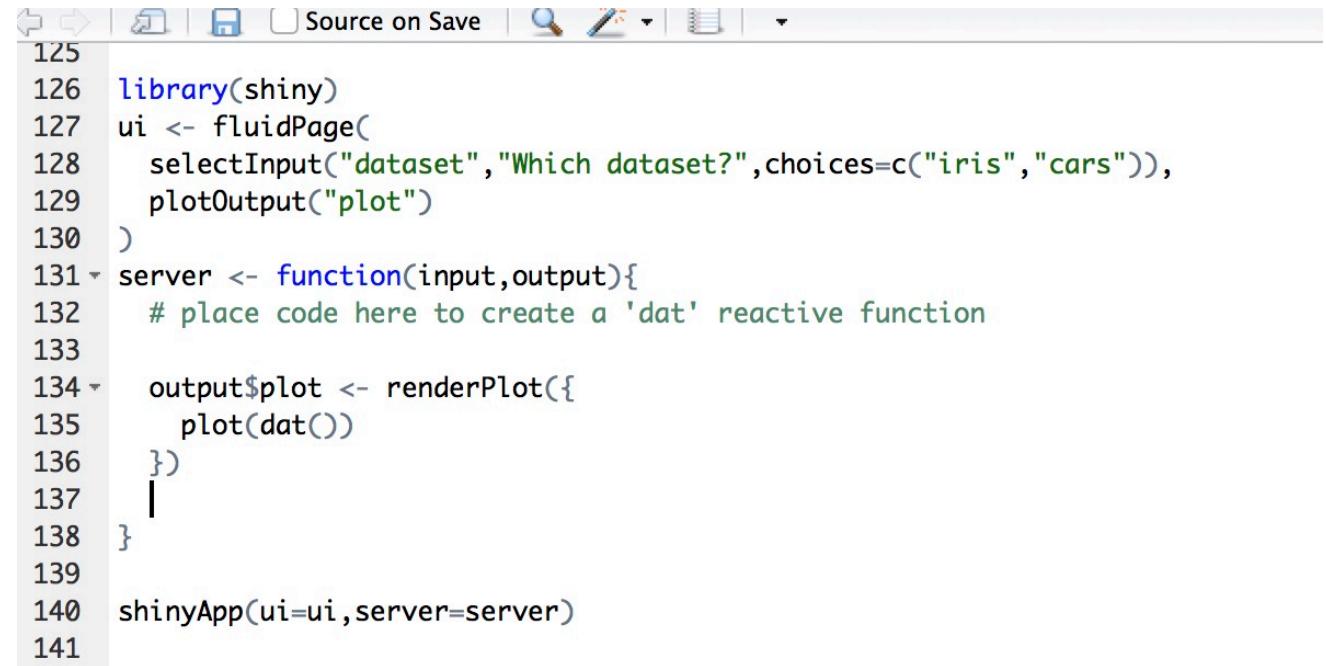
- As your apps become more complicated, it can be useful to simplify your code and create **reactive** functions within the server to use across the multiple functions within the server.
- For example, we can create **reactive** data to use within multiple functions

# Assessment - Reactivity

Create a Shiny App using the following template to plot the iris dataset given:

The general structure for the ui and server are in place as is the code to plot the output from the server.

We want to display a plot for either the Iris or Car datasets depending on the selection



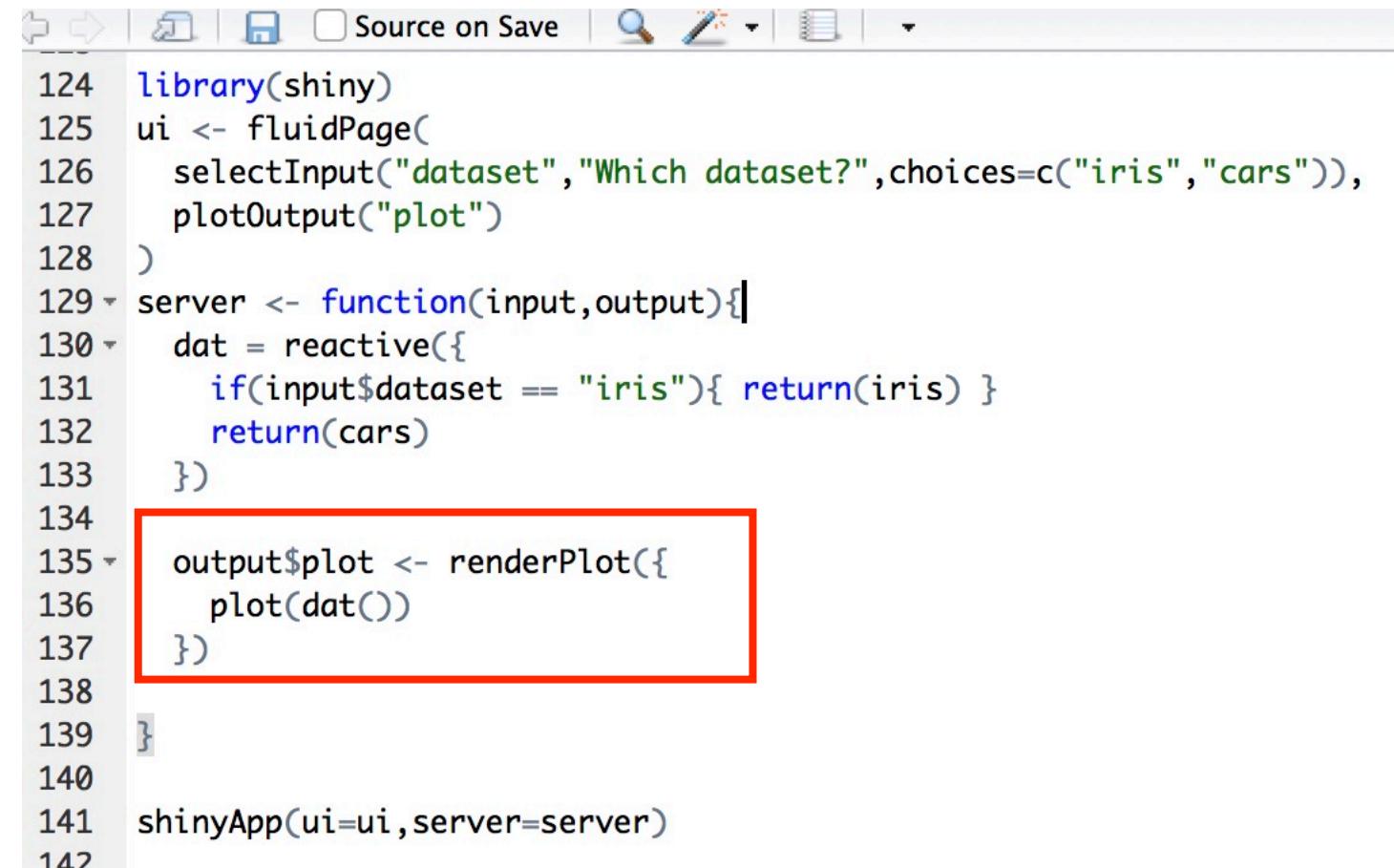
```
125 library(shiny)
126 ui <- fluidPage(
127   selectInput("dataset", "Which dataset?", choices=c("iris", "cars")),
128   plotOutput("plot")
129 )
130 }
131 server <- function(input, output){
132   # place code here to create a 'dat' reactive function
133 
134   output$plot <- renderPlot({
135     plot(dat())
136   })
137 }
138 }
139 
140 shinyApp(ui=ui, server=server)
141 }
```

# Solution

Create a Shiny App using the following template to plot the iris dataset given:

The general structure for the ui and server are in place as is the code to plot the output from the server.

We want to display a plot for either the Iris or Car datasets depending on the selection



A screenshot of the RStudio IDE showing an R script for a Shiny application. The script includes code for the user interface (ui) and the server logic (server). The server logic uses reactive programming to return either the 'iris' or 'cars' dataset based on the user's selection. A specific section of the code, which renders a plot using the 'renderPlot' function, is highlighted with a red rectangular box.

```
124 library(shiny)
125 ui <- fluidPage(
126   selectInput("dataset", "Which dataset?", choices=c("iris", "cars")),
127   plotOutput("plot")
128 )
129 server <- function(input, output){
130   dat = reactive({
131     if(input$dataset == "iris"){ return(iris) }
132     return(cars)
133   })
134   output$plot <- renderPlot{
135     plot(dat())
136   }
137 }
138 }
139 }
140 }
141 shinyApp(ui=ui, server=server)
142 }
```

# Digging deeper.

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

Digging a bit deeper.

```
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

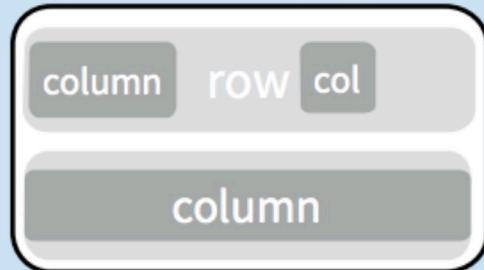
link

Raw html

# Digging a bit deeper.

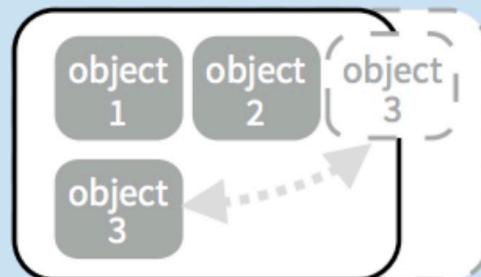
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

## fluidRow()



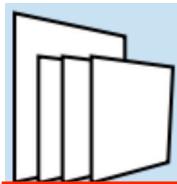
```
ui <- fluidPage(  
  fluidRow(column(width = 4),  
           column(width = 2, offset = 3)),  
  fluidRow(column(width = 12))  
)
```

## flowLayout()



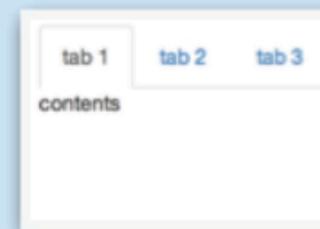
```
ui <- fluidPage(  
  flowLayout( # object 1,  
             # object 2,  
             # object 3  
)  
)
```

# Digging a bit deeper.



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")))
```



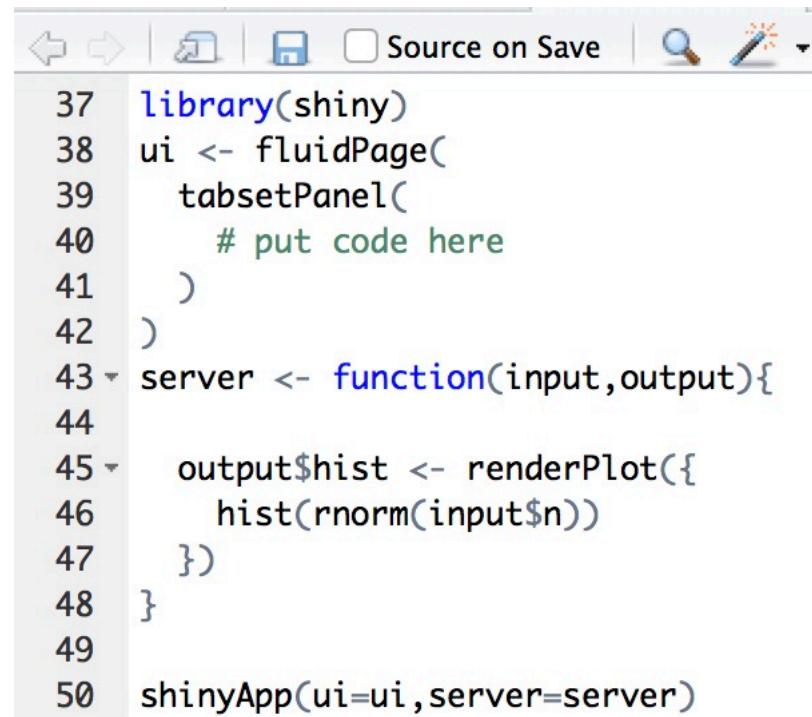
```
ui <- navbarPage(title = "Page",  
  tabPanel("tab 1", "contents"),  
  tabPanel("tab 2", "contents"),  
  tabPanel("tab 3", "contents"))
```



# Assessment – tabs and other HTML

Create a Shiny App using the following template to create a '**tabsetPanel**'ized app with **tabPanels** that will recreate the app we made earlier.

But this time, will make one tab for inputs (**numericInput** for our n observations) and another tab for the histogram output (using **plotOutput**)



The image shows a screenshot of the RStudio IDE. The code editor pane displays the following R script:

```
37 library(shiny)
38 ui <- fluidPage(
39   tabsetPanel(
40     # put code here
41   )
42 )
43 server <- function(input, output){
44
45   output$hist <- renderPlot({
46     hist(rnorm(input$n))
47   })
48 }
49
50 shinyApp(ui = ui, server = server)
```

The code uses the `library(shiny)` command to load the shiny package. It defines a user interface (`ui`) using the `fluidPage` function, which contains a `tabsetPanel`. Inside the `tabsetPanel`, there is a comment indicating where to put the code. The server part of the application (`server`) is defined as a function that takes `input` and `output` arguments. It contains a single `renderPlot` call that generates a histogram for the input `n`. Finally, the `shinyApp` function is used to run the application with the defined `ui` and `server`.

# Solution

Create a Shiny App using the following template to create a '**tabsetPanel**'ized app with **tabPanels** that will recreate the app we made earlier.

But this time, will make one tab for inputs (**numericInput** for our n observations) and another tab for the histogram output (using **plotOutput**)

```
58 library(shiny)
59 ui <- fluidPage(
60   tabsetPanel(
61     # put code here
62     tabPanel("Input",
63       numericInput(inputId = "n",
64                     label = "How many samples",
65                     value = 100)
66     ),
67     tabPanel("Output",
68       plotOutput("hist")
69   )
70 )
71 )
72 server <- function(input,output){
73
74   output$hist <- renderPlot({
75     hist(rnorm(input$n))
76   })
77 }
78
79 shinyApp(ui=ui,server=server)
80
```

# App design

## Two file apps

```
library(shiny)

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

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

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

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

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

# App design

## runApp

You can launch any app from the command line  
with runApp

```
runApp("~/Documents/App-1")
```

File path to app directory.

R will append the file path to the working directory,  
if path does not begin at the home directory

# Deploying your app

- Shinyapps.io
- Local shiny server

# Shinyapps.io

A server maintained by RStudio

- easy to use
- secure
- scalable



© 2016 RStudio, Inc. All rights reserved.

# Example deployment workflow

- Create cloud server instance
  - I prefer to use either Amazon Web Services (AWS) or DigitalOcean
- Install [Shiny](#) Server [optionally [Rstudio](#) Server as well]
- Go to Shiny App directory [default: /srv/shiny-server/shiny]
- Make folder for your project along with server.R and ui.R
  - Optionally do this through Rstudio server and allow it to create the files and example scripts
- Modify
- Enjoy and Profit @ [your-ip-address:3838/shiny/your-project-folder-name](#)

# Example deployment workflow - 2

- Can place shiny scripts into a github repository and anyone can run your apps locally with the runGithub function (or url and runUrl).
  - For example
    - <https://github.com/jnpaulson/MSD1000>
    - <https://github.com/jnpaulson/helpmeviz-malnutrition>

```
library("shiny")
runGitHub("helpmeviz-malnutrition", "nosson")
```

# Example workflow - 3

- Develop shiny code within an Rmarkdown file.
- This allows for static components and then shiny/interactive components within the HTML produced document.
  - [Download the example here](#)
    - Note: requires no ShinyApp function call (nor ui / server function definitions)

# Other shiny topics

- Modules - <https://shiny.rstudio.com/articles/modules.html>
- Reactive Inputs and other Reactivity -  
<https://shiny.rstudio.com/articles/reactivity-overview.html>

# Internship @ Genentech (Summer 2017)

## Genentech

*South San Francisco, California*

**Number of Positions:** 4–6

**Type of Student:** PhD in statistics, biostatistics, or related field

**Deadline for Applying:** January 29, 2017, but offers may be made earlier

Our biostatistics summer interns work for 10–12 weeks under the supervision of experienced biostatisticians on theoretical or applied problems with direct relevance to ongoing clinical or nonclinical drug development research in areas such as oncology, immunology, infectious disease, ophthalmology, and neuroscience. Specific topics include problems in genomics and, more generally, translational research to late-stage clinical trials and post-marketing evaluations. At the end of the internship, students give a department-wide presentation on their research topic. It is not uncommon for an intern to summarize their work in a peer-reviewed publication.

Applicants must be at least 18 and pursuing a PhD in statistics or biostatistics. They must have at least one year of graduate work by May 2016 and be returning to school in the fall of 2016. The applicant must be legally authorized to work in the United States. In addition, applicants should have a good working knowledge of R, S-Plus, or SAS and good communication skills.

**Contact:** Please send CV, personal statement of interest, and a letter of recommendation to  
[gnebiostatsummerintern@gene.com](mailto:gnebiostatsummerintern@gene.com).