Introduction to R Shiny

Summer 2017

Contents

[**The Basic Structure of a Shiny App** 2](#_Toc486253299)

[**Building a Basic User-Interface** 5](#_Toc486253300)

[Adding a Title 7](#_Toc486253301)

[Adding a Layout 7](#_Toc486253302)

[Adding Control Widgets to Collect Inputs from Users 9](#_Toc486253303)

[Adding Placeholders for Outputs (\*Output Functions) 10](#_Toc486253304)

[**Building Output with the Server Function** 11](#_Toc486253305)

[Using the render\* Functions 11](#_Toc486253306)

[Making the Output Reactive 12](#_Toc486253307)

[Creating and Using Reactive Variables 14](#_Toc486253308)

[**An Example: Ames Housing Data Set** 15](#_Toc486253309)

[Loading a Data Set in an App 15](#_Toc486253310)

[Building the User-Interface 15](#_Toc486253311)

[Adding Placeholders for Outputs 17](#_Toc486253312)

[Building Output with the Server Function 17](#_Toc486253313)

[Creating a Reactive Variable 19](#_Toc486253314)

[Using Information from the Data Set to Determine Input Selector Choices 19](#_Toc486253315)

[Allowing the User to Select Multiple Items from a List 21](#_Toc486253316)

[**Creating Separate User-interface and Server Files** 22](#_Toc486253317)

[**Sharing your app using Shinyapps.io** 22](#_Toc486253318)

[**Some Comments on Scoping Rules for Shiny Apps** 22](#_Toc486253319)

[**Links to Learn More** 22](#_Toc486253320)

Introduction to R Shiny

Summer 2017

Shiny is an R package that allows users to build interactive web applications using R. This handout will provide an introduction to using this package. For more information, note that RStudio has put together an excellent tutorial on Shiny: <http://shiny.rstudio.com/tutorial/>.   
  
To begin, you’ll need to install and load the Shiny package:

|  |
| --- |
| install.packages("shiny")  library(shiny) |

To verify that it has installed correctly, try running one of the demo apps:

|  |
| --- |
| runExample("01\_hello") |

To close the app, either press *Escape* or click on the *STOP* icon at the top of your console.

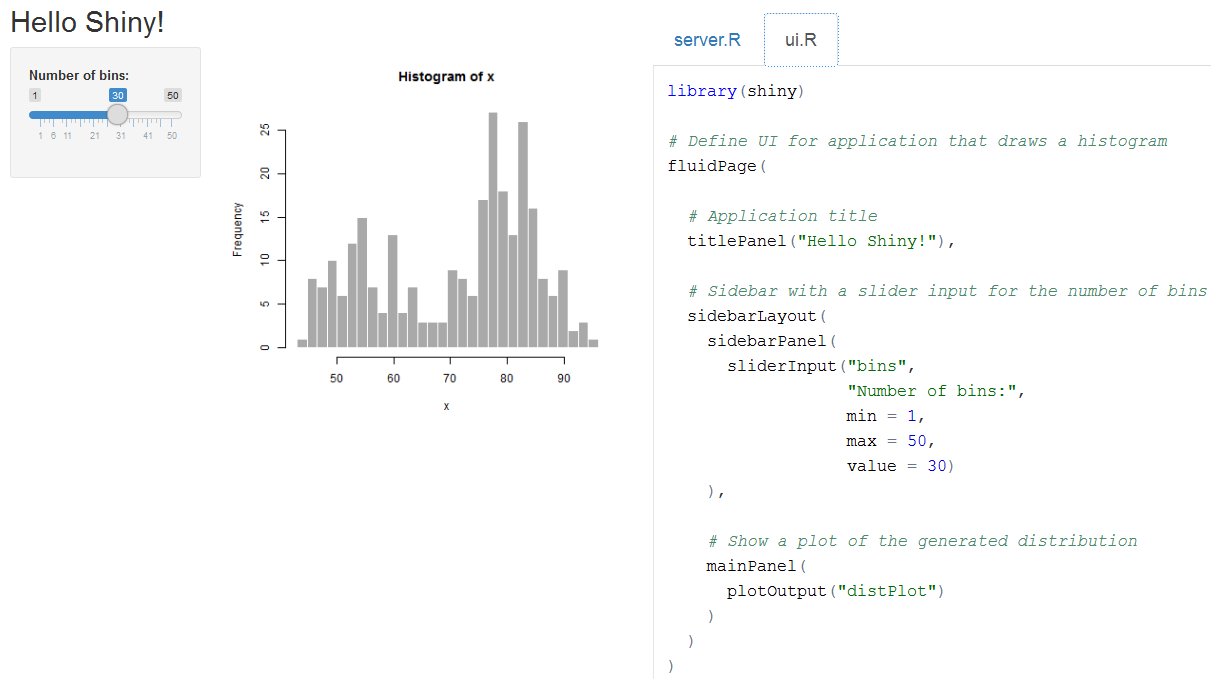
# **The Basic Structure of a Shiny App**

Shiny apps consist of two components:

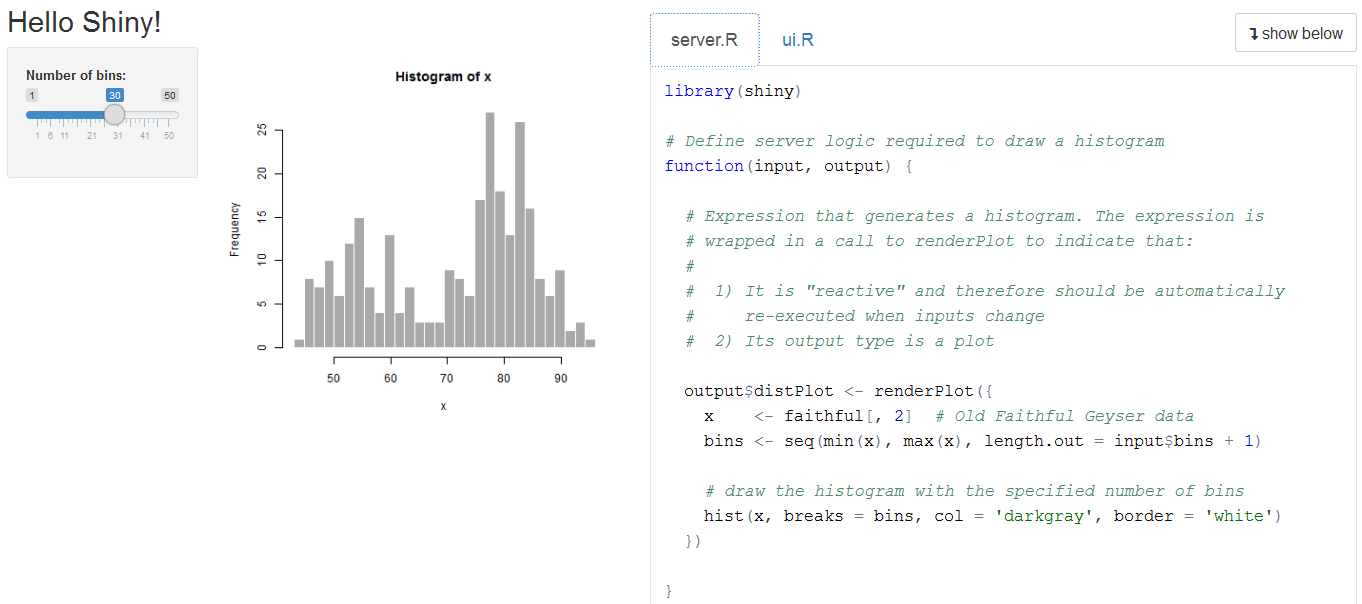
* The user-interface code (which defines the layout and appearance of the app)
* The server script (which contains the code needed to build the app)

*Creates the controls and tells Shiny where to place things*

Consider the code from the example app:



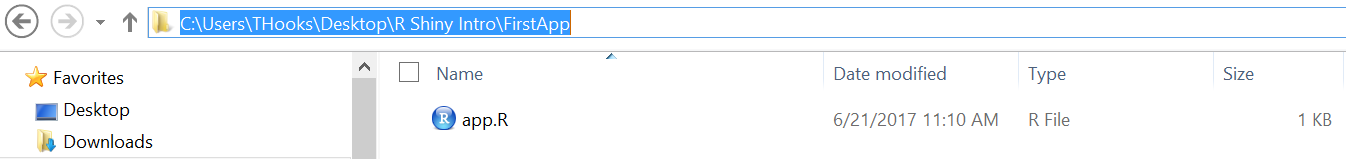
*Creates the actual plot*



To learn more about how Shiny works, let’s recreate the example app from above, starting from scratch. The bare minimum code needed to create a Shiny app is shown below:

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

The above code simply creates an empty user interface and an empty server. Copy and paste this template into a new file named *app.R*, and save it in a new folder named *FirstApp*.



After you have saved the file, RStudio should recognize that this is a Shiny app and the *Run App* button should appear.



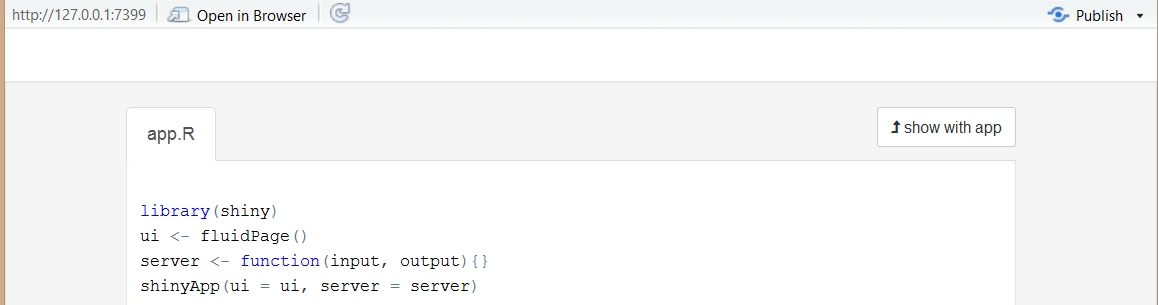
Try running it to verify that the app works, though not much happens.

When you click the *Run App* button, note that the following command runs in the R console:

> runApp('C:/Users/THooks/Desktop/R Shiny Intro/FirstApp')

So, if you prefer, you can run the app by passing the name of its directory to the function runApp.

The app is shown by default in “normal” mode. You can also display the app in “showcase” mode, which will display the source code in addition to the user-interface.  
  
> runApp('C:/Users/THooks/Desktop/R Shiny Intro/FirstApp',   
 display.mode = "showcase")



There are a few things you should keep in mind when creating new Shiny apps:

* You must name the file containing your user-interface and server code *app.R*
* The line containing the code shinyApp(ui = ui, server = server) must be the last line in your file.
* You should get in the habit of placing each app in its own folder (and not in a folder containing other R scripts or files not used by your app)

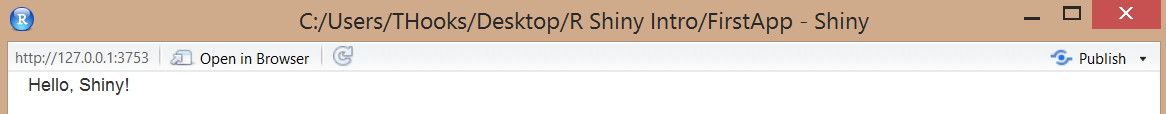
*Finally, note that it is possible to create Shiny apps by separating the ui and server code into two separate files (which might be preferable if the app is very complex). We’ll discuss this later in the handout.*

# **Building a Basic User-Interface**

The user-interface (ui) code uses the fluidPage function to create the app’s display. You can lay out the app by placing elements in the fluidPage function. For example, let’s modify the ui code in our app named “FirstApp.”

|  |
| --- |
| library(shiny)  ui <- fluidPage("Hello, Shiny!")  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

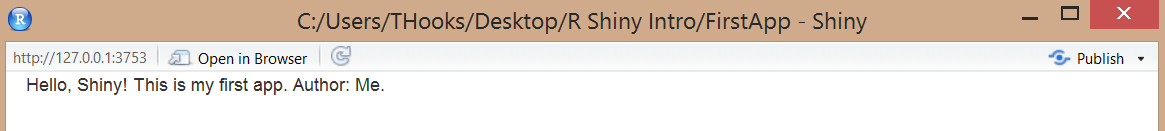
Now, when you re-run the app, you should see the following:



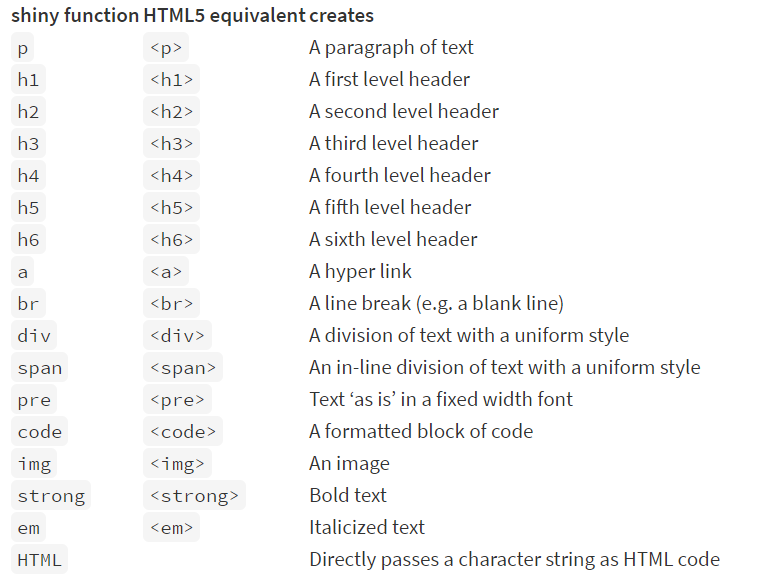
Next, modify the code again and re-run the app:

|  |
| --- |
| library(shiny)  AmesHousing <- read.csv("AmesHousing.csv")  ui <- fluidPage("Hello, Shiny!", "This is my first app.", "Author: Me.")  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

You should see that the text appears in one contiguous block (which doesn’t look very nice).



Shiny has several HTML tag functions available which allow you to easily format the text.

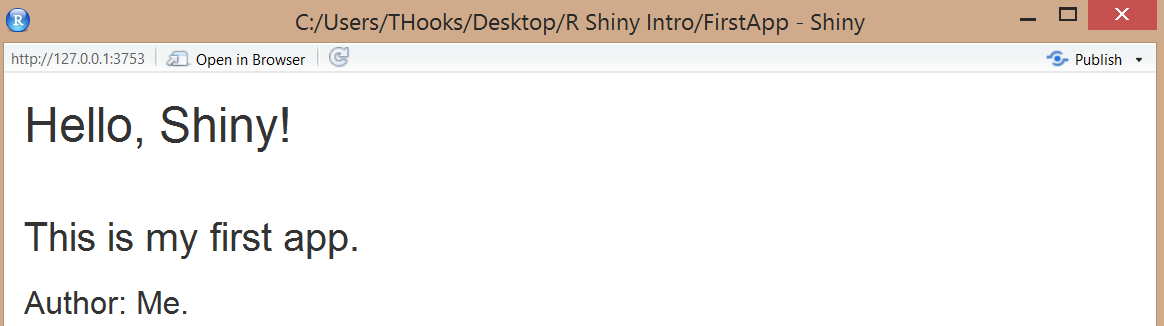


Source: <http://shiny.rstudio.com/tutorial/lesson2/>

To demonstrate, modify the code as shown below and re-run the app.

|  |
| --- |
| library(shiny)  ui <- fluidPage(  h1("Hello, Shiny!"),  br(),  h2("This is my first app."),  h3("Author: Me.")  )  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

You should see the following:



Task: take some time to experiment with different HTML tag functions inside fluidPage().

## Adding a Title

You can use the HTML tag function h1() to add a title, but Shiny has a special function that both adds a visible title to the app and sets the title of the web page.

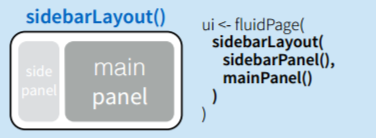
To demonstrate, add the titlePanel function to fluidPage(), as shown below.

|  |
| --- |
| library(shiny)  ui <- fluidPage(  titlePanel("Hello, Shiny!")  )  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

Task: Look at the documentation for the titlePanel() function and note that it has an optional argument. Experiment with this to see what it does.

## Adding a Layout

By default, adding text in the fluidPage() function with HTML tags simply results in a single column of text that is not formatted very nicely. Shiny has several functions available to add more structure to the user-interface. For example, consider the sidebarLayout function, which provides a simple two-column layout as shown below.



Source: <http://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

You can control what appears in each of the panels with the following functions:

* sidebarPanel()
* mainPanel()

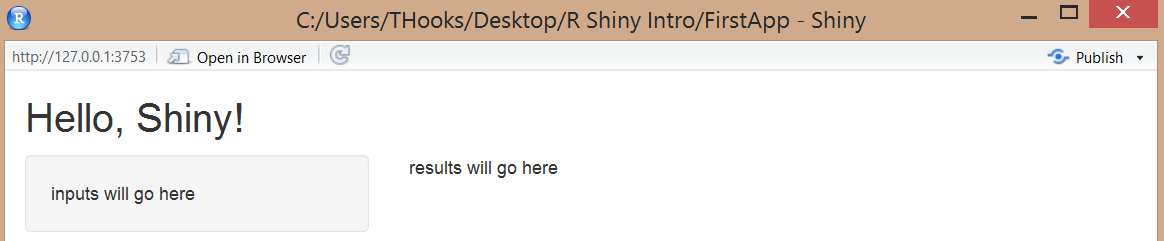
The side panel is displayed with a distinct background color and typically contains input controls.   
The main panel typically contains outputs.

Modify your code as shown below and re-run the app.

|  |
| --- |
| library(shiny)  AmesHousing <- read.csv("AmesHousing.csv")  ui <- fluidPage(  titlePanel("Hello, Shiny!"),  sidebarLayout(  sidebarPanel("inputs will go here"),  mainPanel("results will go here")  )  )  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

Note that the arguments inside fluidPage() are separated by commas! The arguments inside sidebarLayout() are separated by commas, as well.

The app now appears as follows:



Options do exist to create more advanced layouts. For example, navbarPage allows you to create a multi-page user-interface that includes a navigation bar. Or, fluidRow and column can be used to build the layout up from a grid system.

## Adding Control Widgets to Collect Inputs from Users

In Shiny, control widgets can be used to collect input values from the app’s user. When the user changes the widget(s), the value(s) changes, as well. Several pre-built widgets are available in Shiny:

| **Function** | **Widget** |
| --- | --- |
| actionButton | Action Button |
| checkboxGroupInput | A group of check boxes |
| checkboxInput | A single check box |
| dateInput | A calendar to aid date selection |
| dateRangeInput | A pair of calendars for selecting a date range |
| fileInput | A file upload control wizard |
| helpText | Help text that can be added to an input form |
| numericInput | A field to enter numbers |
| radioButtons | A set of radio buttons |
| selectInput | A box with choices to select from |
| sliderInput | A slider bar |
| submitButton | A submit button |
| textInput | A field to enter text |

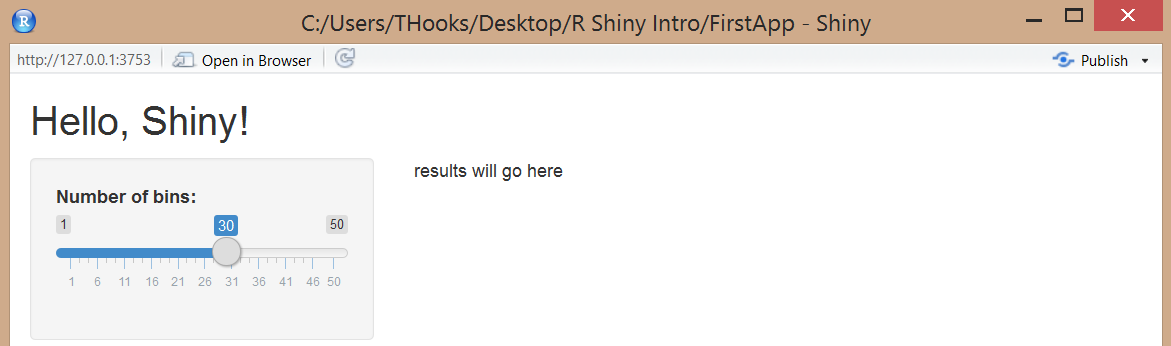
You can add control widgets to your app by placing any of the above functions in either the sidebarPanel or mainPanel functions. Each widget function requires *at least* these first two arguments:

* inputId – this is your name for the widget (you will use this to access the widget’s value)
* label - this text will appear with the widget in your app (note that it can be an empty string, “”)

For example, consider the following modification to our user-interface code which adds a slider bar to allow the user to specify the number of bins for the histogram.

|  |
| --- |
| ui <- fluidPage(  titlePanel("Hello, Shiny!"),  sidebarLayout(  sidebarPanel(  sliderInput("bins",  "Number of bins:",  min = 1,  max = 50,  value = 30)  ),  mainPanel("results will go here")  )  ) |

The app should now appear as follows:



Task: Identify the purpose of each argument in the sliderInput() function, and change some of the parameters of sliderInput() to see how it affects the app. Then, change the app so that it collects the information from the user with the numericInput() function, instead.

## Adding Placeholders for Outputs (\*Output Functions)

Once all of the inputs have been created, we can add elements to the user-interface to display the outputs (e.g., a plot, table, or text). At this point, we can create only placeholders for the outputs; the actual output will be constructed later in the server portion of the code.   
  
There are several output functions available in Shiny.

| **Output Function** | **Creates…** |
| --- | --- |
| htmlOutput | Raw HTML |
| imageOutput | Image |
| plotOutput | Plot |
| tableOutput | Table |
| textOutput | Text |
| uiOutput | Raw HTML |
| verbatimTextOutput | Text |

Like the input functions, these output functions have an outputID argument that is used to identify each output. For example, modify the mainPanel() portion of your code as follows. This code adds a placeholder in the user-interface for a plot named *distPlot*.

|  |
| --- |
| ui <- fluidPage(  titlePanel("Hello, Shiny!"),  sidebarLayout(  sidebarPanel(  sliderInput("bins",  "Number of bins:",  min = 1,  max = 50,  value = 30)  ),  mainPanel(  plotOutput("distPlot")  )  )  ) |

# **Building Output with the Server Function**

The server portion of the code responds to changes in the inputs and then creates outputs to display in the app. The server function has two arguments: input and output, which are list-like objects. The input object is a list you read values from; the output object is a list you write values to.

## Using the render\* Functions

Recall that we have already created an output placeholder, *distPlot*. To build the output, we must first do the following in the server portion of the code:

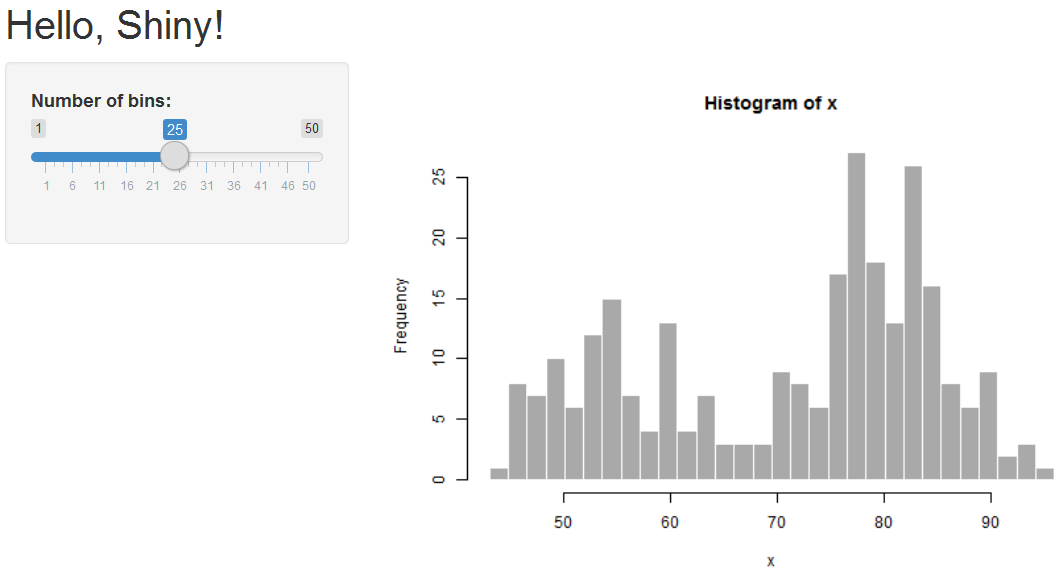
* Define the desired output element in the output list (the name should match the name of the placeholder defined in the user-interface portion of the code)
* Build the output object with one of Shiny’s render functions

| **Render Function** | **Creates…** |
| --- | --- |
| renderImage | Images |
| renderPlot | Plots |
| renderPrint | Any printed output |
| renderTable | Data frame, matrix, other table-like structures |
| renderText | Character strings |
| renderUI | A Shiny tag object or HTML |

For example, modify the server portion of the code as shown below to create a histogram of the waiting times between eruptions and send it to the *distPlot* output (*note that this code simply sets the number of bins to 30 – we’ll modify this later*).

|  |
| --- |
| server <- function(input, output){  output$distPlot <- renderPlot({  x <- faithful[, 2] # Old Faithful Geyser data  bins <- seq(min(x), max(x), length.out = 30 + 1)  hist(x, breaks = bins, col = 'darkgray', border = 'white')  })  } |

When the app is run, the following is displayed:



## Making the Output Reactive

Note that the app displays the plot; however, the plot is not yet [*reactive*](https://shiny.rstudio.com/articles/understanding-reactivity.html).

To make the histogram react to the slider (which will allow the user to choose the number of bins), you must ask Shiny to call a widget value when it builds the plot. This is accomplished using the input object you created in the user-interface portion of the code. The current values of all widgets in your app are saved in the input object under the names that were assigned to the widgets earlier.

|  |
| --- |
| library(shiny)  ui <- fluidPage(  titlePanel("Hello, Shiny!"),  sidebarLayout(  sidebarPanel(  sliderInput("bins",  "Number of bins:",  min = 1,  max = 50,  value = 30)  ),  mainPanel(  plotOutput("distPlot")  )  )  )  server <- function(input, output){  output$distPlot <- renderPlot({  x <- faithful[, 2] # Old Faithful Geyser data  bins <- seq(min(x), max(x), length.out = input$bins + 1)  hist(x, breaks = bins, col = 'darkgray', border = 'white')  })  }  shinyApp(ui = ui, server = server) |

Now, the app functions as originally intended. The histogram gets updated whenever the number of bins is changed by the user.

Task: Create an app that allows the user to select either the variable *eruptions* or w*aiting* from the Old Faithful data, displays the histogram (with default bin width), and displays summary statistics for the chosen variable.

Solution:

|  |
| --- |
|  |

## Creating and Using Reactive Variables

In the previous section, we already saw one way that reactive variables can be created in Shiny (i.e., using elements from the input list). Such reactive variables, however, can only be used inside a reactive context. For example, try to add the command print(input$bins) to the server function. What happens?

Essentially, a reactive context gets updated if the values of any of the reactive variables on which it depends are changed. There are three common reactive contexts in Shiny in which you can access reactive variables.

* Any render function (e.g., renderPlot) is by default a reactive context, so you can always use variables defined in the input list inside render functions.
* The observe({}) function allows you to *observe* the value of a reactive variable. To see how the observe({})function works, add observe({print(input$bins)}) to the server function and see what happens.
* The reactive({}) function allows you to *create* your own reactive variables not included in the input list.

The reactive({}) function actually returns a value. Often, this can be used to reduce code duplication. For example, in the task on the previous page, you probably used if statements to define the variable of interest in two different places in the code. Alternatively, we can define a reactive variable that will hold this value and then use it in each of the render functions.

|  |
| --- |
| server <- function(input, output){    x<-reactive({  faithful[[input$var]]  })  output$distPlot <- renderPlot({  hist(x(), col = 'darkgray', border = 'white', xlab = input$var)  })  output$summ <- renderPrint({  summary(x())  })  } |

The variable filtered is being defined almost like before, except for the following:

* now it is wrapped by the reactive({}) function
* it is defined in the server function instead of inside the individual render functions

Also, note that to access a variable defined with the reactive({}) function, you must add parentheses after the variable name (as if you’re calling a function).

# **An Example: Ames Housing Data Set**

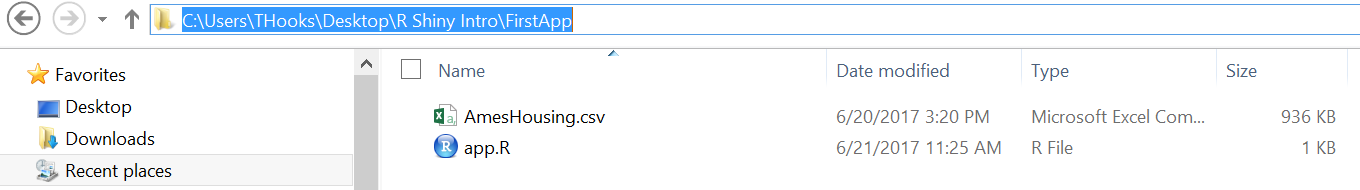
Next, we will consider an example app that first requires us to load an external data set. The app will simply create a histogram of the variable *Sale Price* but will allow the user to filter the observations based on *Year Built* and *Kitchen Quality*.

## Loading a Data Set in an App

The data set we will use in our example app is stored in the file **AmesHousing.csv**. These data describe the sales of individual residential properties in Ames, Iowa, from 2006 to 2010 ([source](http://ww2.amstat.org/publications/jse/v19n3/decock.pdf)).

A documentation file is available [here](https://ww2.amstat.org/publications/jse/v19n3/decock/DataDocumentation.txt).

To use these data in the app, start by saving the file in the same directory as the Shiny app.



Then, add a line in the app to load the data into an object named Ames. Place this command in the second line of your code, just after library(shiny).

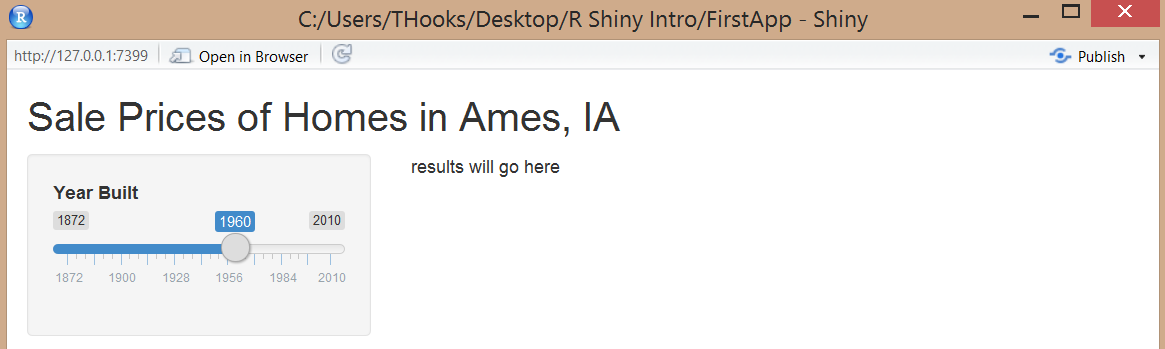
|  |
| --- |
| library(shiny)  AmesHousing <- read.csv("AmesHousing.csv")  ui <- fluidPage()  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

## Building the User-Interface

The user-interface code shown below adds a slider bar to allow the user to select only houses built in a particular year (or years); recall that *Year.Built* is a variable in the data set.

|  |
| --- |
| ui <- fluidPage(  titlePanel("Sale Prices of Homes in Ames, IA"),  sidebarLayout(  sidebarPanel(  sliderInput("YrBlt","Year Built", min=min(AmesHousing$Year.Built),  max= max(AmesHousing$Year.Built), value = 1960, sep="")  ),  mainPanel("results will go here")  )  ) |

The app should now appear as follows when you run the code:



Note that at this point, the app allows the user to select only a single year. It would make much more sense for the user to select a range of years for consideration. This can be accomplished by supplying a vector of length two as the value argument to the sliderInput() function.

|  |
| --- |
| ui <- fluidPage(  titlePanel("Sale Prices of Homes in Ames, IA"),  sidebarLayout(  sidebarPanel(  sliderInput("YrBlt","Year Built", min=min(AmesHousing$Year.Built),  max= max(AmesHousing$Year.Built),   value = c(1960,max(AmesHousing$Year.Built)),  sep="")  ),  mainPanel("results will go here")  )  ) |

Task: Change some of the parameters of sliderInput() to see how it changes the result.

For illustrative purposes, let’s also add a select box to allow the user to specify houses based on their Kitchen Quality (recall that *Kitchen.Qual* is a variable in the data set).

|  |
| --- |
| ui <- fluidPage(  titlePanel("Sale Prices of Homes in Ames, IA"),  sidebarLayout(  sidebarPanel(    sliderInput("YrBlt","Year Built", min=min(AmesHousing$Year.Built),  max= max(AmesHousing$Year.Built),   value = c(1960,max(AmesHousing$Year.Built)),  sep=""),    selectInput("KQ", "Kitchen Quality",   choices = c("Ex", "Gd", "TA", "Fa", "Po"))  ),    mainPanel("results will go here")  )  ) |

## Adding Placeholders for Outputs

Modify your user-interface code as follows:

|  |
| --- |
| Library(shiny)  AmesHousing <- read.csv("AmesHousing.csv")  ui <- fluidPage(  titlePanel("Sale Prices of Homes in Ames, IA"),  sidebarLayout(  sidebarPanel(    sliderInput("YrBlt","Year Built", min=min(AmesHousing$Year.Built),  max= max(AmesHousing$Year.Built),   value = c(1960,max(AmesHousing$Year.Built)),  sep=""),    selectInput("KQ", "Kitchen Quality",   choices = c("Ex", "Gd", "TA", "Fa", "Po"))  ),    mainPanel(plotOutput("SPplot"))  )  )  server <- function(input, output){}  shinyApp(ui = ui, server = server) |

Note that this code adds a placeholder in the user-interface for a plot named *SPplot*.

## Building Output with the Server Function

Modify the server portion of the code as shown below to create a histogram of sales prices and send it to the *SPplot* output. (Note that we should add the command library(ggplot2) at the top to ensure that the ggplot2 package is loaded).

|  |
| --- |
| server <- function(input, output){  output$SPplot <- renderPlot({  ggplot(data=AmesHousing) + geom\_histogram(mapping=aes(SalePrice))  })  } |

When the app is run, the plot appears; however, the plot is not yet *reactive*. To make it react to the slider bar and select box, you must ask Shiny to call a widget value when it builds the plot. Recall that this is accomplished using the input object. The current values of all widgets in your app are saved in the input object under the names that were assigned in the ui portion of the code.

|  |
| --- |
| ui <- fluidPage(  titlePanel("Sale Prices of Homes in Ames, IA"),  sidebarLayout(  sidebarPanel(    sliderInput("YrBlt","Year Built", min=min(AmesHousing$Year.Built),  max= max(AmesHousing$Year.Built),   value = c(1960,max(AmesHousing$Year.Built)),  sep=""),    selectInput("KQ", "Kitchen Quality",   choices = c("Ex", "Gd", "TA", "Fa", "Po"))  ),    mainPanel(plotOutput("SPplot"))  )  ) |

We can filter the data set based on the values of these inputs. We’ll use dplyr functions to filter the data, so make sure that the dplyr package is loaded at the top of the code. Place the code below into your server function and re-run the app.

|  |
| --- |
| server <- function(input, output){  output$SPplot <- renderPlot({  filtered <-  AmesHousing %>%  filter(Year.Built >= input$YrBlt[1],  Year.Built <= input$YrBlt[2],  Kitchen.Qual == input$KQ)  ggplot(data=filtered) + geom\_histogram(mapping=aes(SalePrice))  })  } |

Now, the plot will update when the input values are changed.

Task: Add a new text output that indicates how many houses are included in the plot.

## Creating a Reactive Variable

Recall that the reactive({}) function can be used to reduce code duplication. In this example, when you added a new text output to indicate how many houses are included in the plot (see previous task), the original data set was filtered in two different places in the code. To be more efficient, we can define a reactive variable that will hold the filtered data set and then use that variable in the render functions.

|  |
| --- |
| server <- function(input, output){  # observe({print(input$KQ)})    filtered <- reactive({  AmesHousing %>%  filter(Year.Built >= input$YrBlt[1],  Year.Built <= input$YrBlt[2],  Kitchen.Qual == input$KQ)  })    output$SPplot <- renderPlot({  ggplot(data=filtered()) + geom\_histogram(mapping=aes(SalePrice))  })    output$NumHouses <- renderText({  nrow(filtered())  })  } |

## Using Information from the Data Set to Determine Input Selector Choices

In our example, there were only five values for *Kitchen Quality*, so it wasn’t difficult to enter these manually. It is possible, however, to use information from the data set to determine selector choices.

The uiOutput() function renders a reactive output variable as HTML within the app; in other words, it allows you to create inputs dynamically.

To populate the select box with the values Kitchen Quality assumes in the data set, you must first replace the selectInput(“KQ”, …) in the user-interface portion of the code with the following:

|  |
| --- |
| uiOutput("KQOutput") |

Then, you must create the output in the server portion of the code as follows:

|  |
| --- |
| output$KQOutput <- renderUI({  selectInput("KQ","Kithen Quality",  sort(unique(AmesHousing$Kitchen.Qual)),selected = "Ex")  }) |

The code for the app now is as shown below in its entirety:

|  |
| --- |
| library(shiny)  library(ggplot2)  library(dplyr)  AmesHousing <- read.csv("AmesHousing.csv")    ui <- fluidPage(  titlePanel("Sale Prices of Homes in Ames, IA"),  sidebarLayout(  sidebarPanel(    sliderInput("YrBlt","Year Built", min=min(AmesHousing$Year.Built),  max= max(AmesHousing$Year.Built),   value = c(1960,max(AmesHousing$Year.Built)),  sep=""),    uiOutput("KQOutput")  ),    mainPanel(plotOutput("SPplot"),br(),  "Number of houses included in plot:", textOutput("NumHouses"))  )  )  server <- function(input, output){    output$KQOutput <- renderUI({  selectInput("KQ","Kitchen Quality",  sort(unique(AmesHousing$Kitchen.Qual)),selected = "Ex")  })    filtered <- reactive({  AmesHousing %>%  filter(Year.Built >= input$YrBlt[1],  Year.Built <= input$YrBlt[2],  Kitchen.Qual == input$KQ)  })    output$SPplot <- renderPlot({  ggplot(data=filtered()) + geom\_histogram(mapping=aes(SalePrice))  })    output$NumHouses <- renderText({  nrow(filtered())  })  }  shinyApp(ui = ui, server = server) |

When you run this app, you may notice that R produces an error message. This happens because filtered is trying to access the Kitchen Quality input which technically hasn’t been created yet.

To circumvent this error, revise the code so that it checks to see if the Kitchen Quality input exists and returns NULL if it does not. Note that you must also do a similar check in the renderPlot() function since ggplot will not work with a NULL dataset.

|  |
| --- |
| filtered <- reactive({  if(is.null(input$KQ)){return(NULL)}  AmesHousing %>%  filter(Year.Built >= input$YrBlt[1],  Year.Built <= input$YrBlt[2],  Kitchen.Qual == input$KQ)  })    output$SPplot <- renderPlot({  if(is.null(filtered())){return()}  ggplot(data=filtered()) + geom\_histogram(mapping=aes(SalePrice))  }) |

## Allowing the User to Select Multiple Items from a List

So far, the app we have developed allows the user to select only one level for Kitchen Quality at a time. To allow the user to select multiple levels, you can modify the code as follows:

|  |
| --- |
| server <- function(input, output){  output$KQOutput <- renderUI({  selectInput("KQ","Kitchen Quality",sort(unique(AmesHousing$Kitchen.Qual)),  selected = "Ex",multiple=TRUE)  })  filtered <- reactive({  if(is.null(input$KQ)){return(NULL)}    AmesHousing %>%  filter(Year.Built >= input$YrBlt[1],  Year.Built <= input$YrBlt[2],  Kitchen.Qual %in% input$KQ)  })  output$SPplot <- renderPlot({    if(is.null(filtered())){return()}    ggplot(data=filtered()) + geom\_histogram(mapping=aes(SalePrice))  })    output$NumHouses <- renderText({  nrow(filtered())  })  } |

You may prefer the checkboxGroupInput() option, instead:

|  |
| --- |
| checkboxGroupInput("KQ","Kitchen Quality",  sort(unique(AmesHousing$Kitchen.Qual)),selected = "Ex") |

# **Creating Separate User-interface and Server Files**

If desired, you can separate the ui and server code into two different files: ui.R and server.R. Simply put all of the code assigned to ui in the ui.R file; likewise, put the code assigned to server in the server.R file. These two files must be stored in the same folder for RStudio to recognize your code as a Shiny app.   
  
Note that you do not need to include the ShinyApp(ui=ui, server=server) line in your code when you use the multiple file method.

Task: Recreate the Ames Housing App using the multiple file method.

# **Sharing your app using Shinyapps.io**

RStudio provides a service called shinyapps.io which lets you host your apps for free. You can publish your apps with the click of a button in Rstudio. Go to [www.shinyapps.io](http://www.shinyapps.io/) and sign up for an account. When you’re ready to publish your app, click on the “Publish Application” button in RStudio and follow their instructions.



You might be asked to install a couple packages if it’s your first time.

# **Some Comments on Scoping Rules for Shiny Apps**

Where you define objects in an R Shiny app matters, especially when you are supporting multiple users. You should read the following [article](http://shiny.rstudio.com/articles/scoping.html) to learn more about scoping rules for Shiny apps.

# **Links to Learn More**

Click [here](http://shiny.rstudio.com/articles/scoping.html) for access to several articles that will help you learn more about R Shiny.