MANGO SOLUTIONS

Data analysis that delivers

Introduction to Shiny

Workshop for BirminghamR

Trainer: Graham Parsons

Level: Foundation

@ training@mango-solutions.com

+44 (0)1249 75 450

mango-solutions.com

Chapter 1
Introduction to Shiny

1.1 Introduction to the Training

1.1.1 Overview

This course is designed to provide a more advanced knowledge of the R language. It is designed as a follow-on to Mango's Introduction to R course and it is therefore assumed that attendees are familiar with the contents of that course. In this course we will introduce the **shiny** package for developing web frameworks from R.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
> This is a section of code                 # This is a comment
```

🔴	A warning: typically describing non-intuitive aspects of the R language
🟠	A tip: additional features of R or "shortcuts" based on user experience
🟢	Exercises: to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

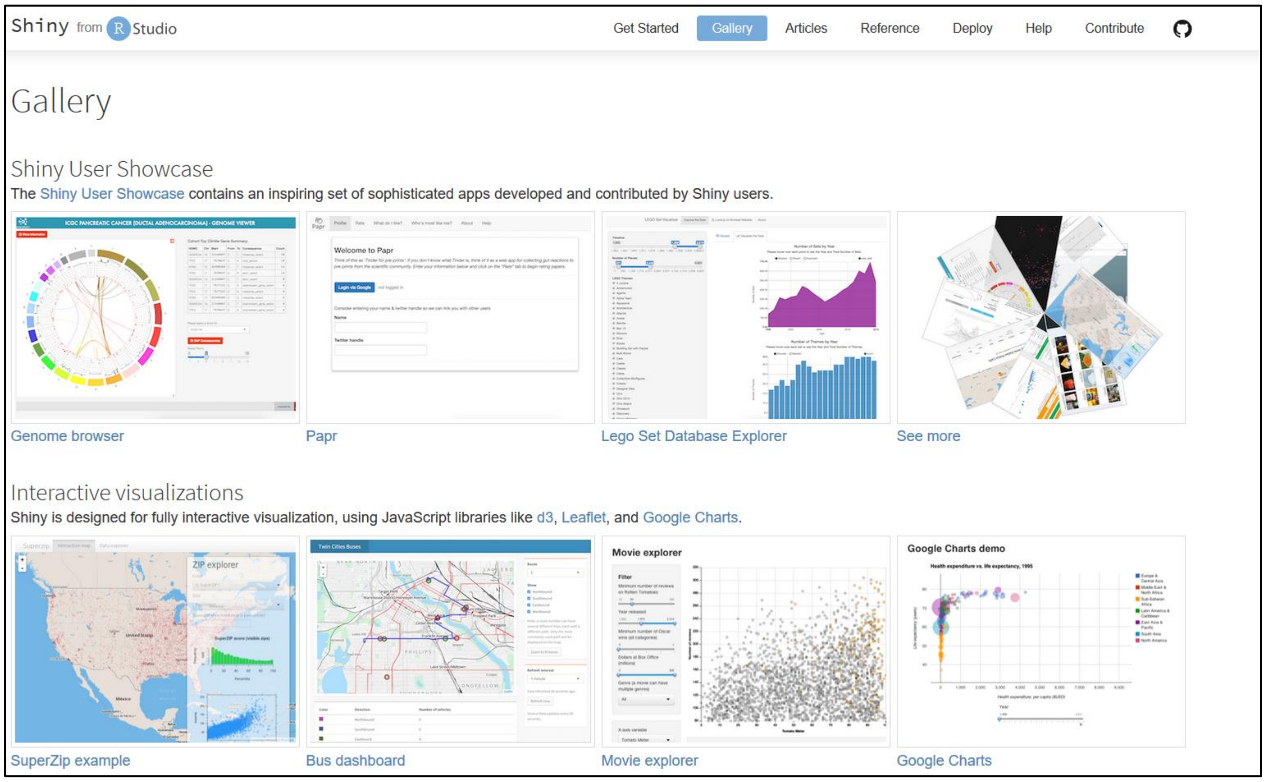
A great deal of code will be executed within R by the consultant during the delivery of this training. This includes the answers to each exercise, as well as other code written to answer questions that arise. Following the course, each attendee will be sent a script containing all the code that was executed.

1.2 Introduction to Shiny

The **shiny** R package allows us to develop interactive web applications directly from R. We can make our analysis, data models and reports available to anyone with a web browser, and enable our users to benefit from the power of the R language without having to know how to code.

Shiny is based entirely in R so we can use the same tools for analysis and graphics that we are already familiar with. Shiny takes care of the web technologies behind the scenes, meaning we do not need to know HTML, JavaScript, CSS, etc. in order to produce great interactive apps.

Shiny is developed and maintained by RStudio, the makers of the RStudio development environment. RStudio host a Gallery of existing applications that is a great way to understand what shiny is capable of and to get inspiration for your apps.



Gallery

Shiny User Showcase

The Shiny User Showcase contains an inspiring set of sophisticated apps developed and contributed by Shiny users.

Interactive visualizations

Shiny is designed for fully interactive visualization, using JavaScript libraries like d3, Leaflet, and Google Charts.

1.3 Getting Up and Running

1.3.1 *Installing shiny*

The **shiny** framework is a regular R package that is published on CRAN, so we can install it just like most other packages.

```
> install.packages("shiny")
```

1.3.2 *Running the Example Apps*

A good starting point to make sure **shiny** is working correctly is to run the example apps that come bundled with the package.

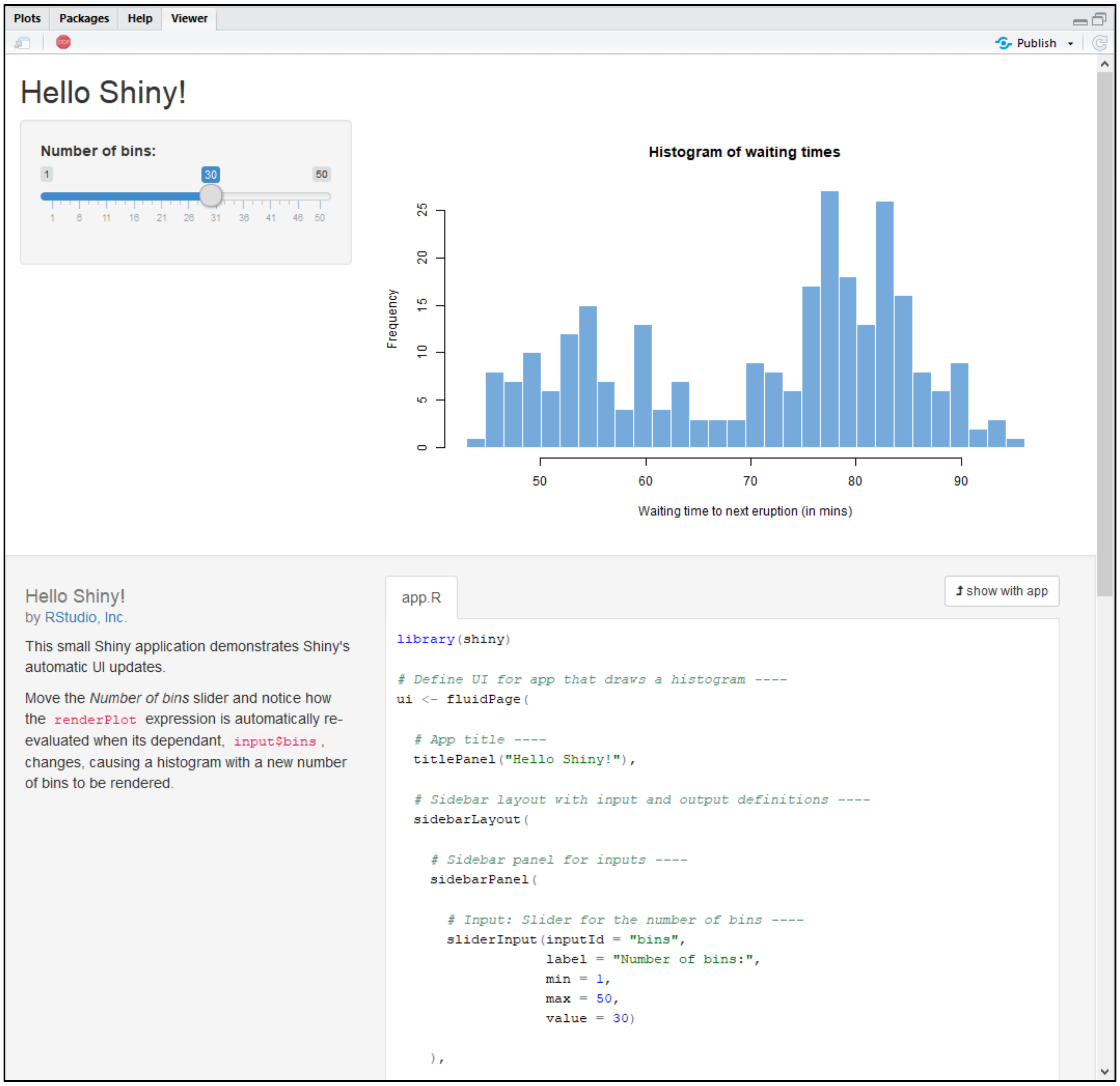
We can run these examples using the `runExample` function. Calling the function with no arguments displays the available examples:

```
> library(shiny)
> runExample()
Valid examples are "01_hello", "02_text", "03_reactivity", "04_mpg",
"05_sliders", "06_tabsets", "07_widgets", "08_html", "09_upload",
"10_download", "11_timer"
```

We can run an example by providing its name as a string to the `runExample` function:

```
> runExample("01_hello")
```

The app will open in a web browser, which will either be RStudio's built-in browser, or a web browser you have installed on your system, for example Chrome or Firefox.



Hello Shiny!
by RStudio, Inc.

This small Shiny application demonstrates Shiny's automatic UI updates.

Move the *Number of bins* slider and notice how the `renderPlot` expression is automatically re-evaluated when its dependant, `input$bins`, changes, causing a histogram with a new number of bins to be rendered.

```
library(shiny)

# Define UI for app that draws a histogram ----
ui <- fluidPage(

  # App title ----
  titlePanel("Hello Shiny!"),

  # Sidebar layout with input and output definitions ----
  sidebarLayout(

    # Sidebar panel for inputs ----
    sidebarPanel(

      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)

    ),
```

The `runExample` function displays apps in two halves:

- The top half shows the app itself, with an input slider and an output chart showing the distribution of wait times between eruptions for the “Old Faithful” geyser.
- The bottom half shows a brief description of the app alongside its R code

Later we will learn how this code works.

1.3.2.1 *Inputs and Outputs*

In **shiny** we describe elements on the page as either *inputs* or *outputs*. Inputs provide widgets for the user of our apps to interactive with, such as the slider widget. Outputs are the results of our app that the user wants to see, such as the histogram above.

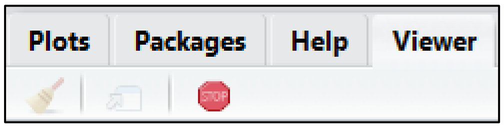
We can look at the example files directly using the `find.package` function to find the shiny package on disk, then navigate to the `examples` subfolder:

```
> find.package("shiny")
[1] "C:/Users/Mango/Documents/R/win-library/3.4/shiny"
```

The valid example names are just the subfolders within `examples`.

1.3.2.2 Stopping an App

When shiny apps are running you will find that you cannot run any R commands as the console is busy. To stop your running app click the “stop” icon, which will appear at the top of your Console tab and also at the top of the Viewer tab.



1.3.2.3 The Example Apps

The examples currently bundled with **shiny** are as follows:

App Name	Description
01_hello	A simple histogram with slider
02_text	Shows different text output options
03_reactivity	Demonstrates shiny’s core concept of reactivity
04_mpg	Demonstrates using global variables and reactivity
05_sliders	Shows different ways of using the slider widget
06_tabsets	Shows an example of a tabbed interface
07_widgets	Demonstrates some other widgets such as helpText
08_html	Shows shiny can use custom HTML files for layout
09_upload	Demonstrates uploading files into an app
10_download	Demonstrates downloading data from an app
11_timer	Demonstrates triggering events on a schedule

1. Install and load the shiny library
2. Run the “01_hello” example and test its functionality
3. Pick another example app and run it

Chapter 2
Building a Basic App

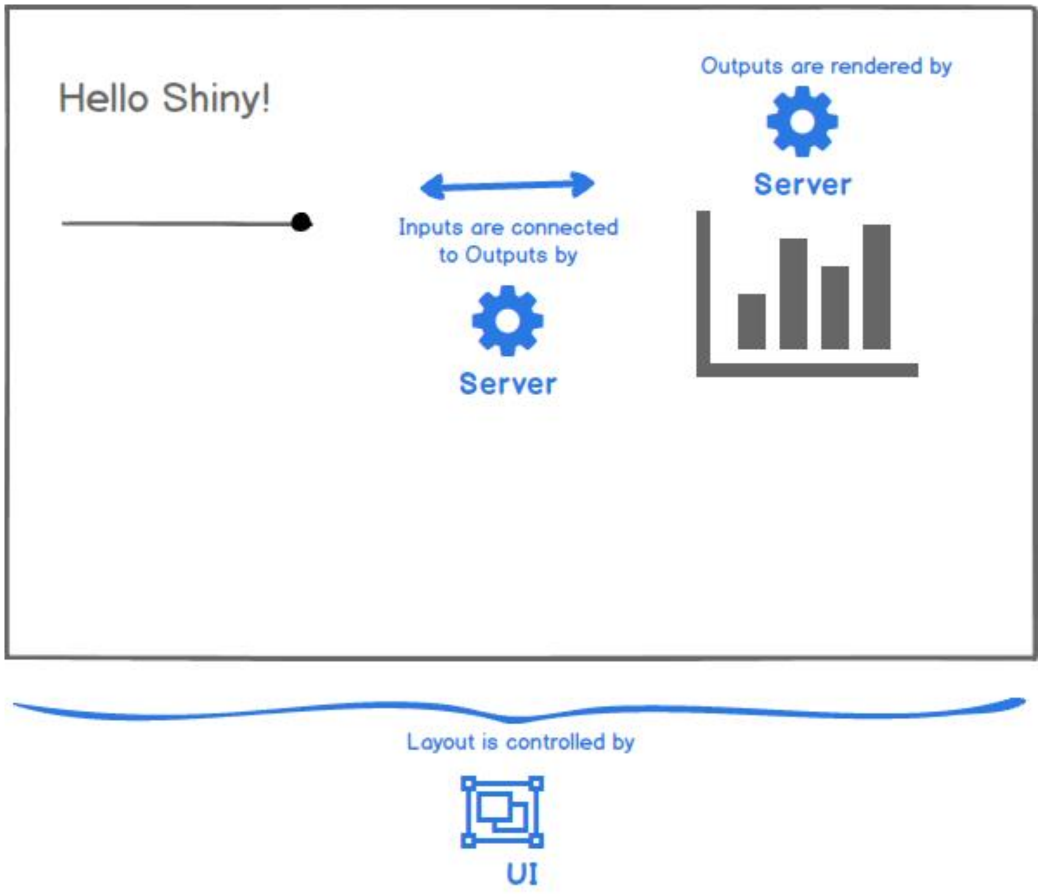
2.1 Overview

In this chapter we will familiarise ourselves with the structure of a **shiny** application and then build our first application.

2.2 Understanding an App's Architecture

Each app is constructed from two fundamental components:

- The "User Interface" component
  - Describes the app's layout and appearance
- The "Server" component:
  - Describes the R code needed to create the outputs, e.g. how to make the histogram
  - Describes how the input elements are connected to the output elements, e.g. how the slider is connected to the histogram



2.3 Creating an App

We will create our own version of the interactive histogram we have seen already by building it up from basic components.
In order to create a minimal shiny application we need to load the **shiny** library, define the user interface (UI) and define the server behaviour. We can do this in a new R script as follows:

```
library(shiny)
# Define the user interface component
ui <- fluidPage()
# Define the server component
server <- function(input, output) {}
# Combine the two components
shinyApp(ui = ui, server = server)
```

2.3.1 Saving Apps

To work correctly the shiny application files need to adhere to specific naming conventions. We need to save our application file as **app.R** in order for it to be recognised as a shiny app.

Since we can only give one name to our app file we cannot have more than one app in the same directory. Each app should be in its own directory, and we use the directory name to give names to each of our apps.

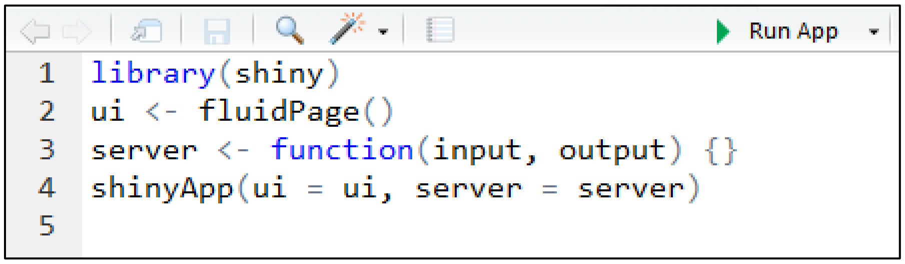
For the minimal app above we would create a directory called Minimal_App and save app.R in it.

We will be creating several apps throughout the course, so ensure you create a new directory each time you want to create a new app.

2.4 Running apps

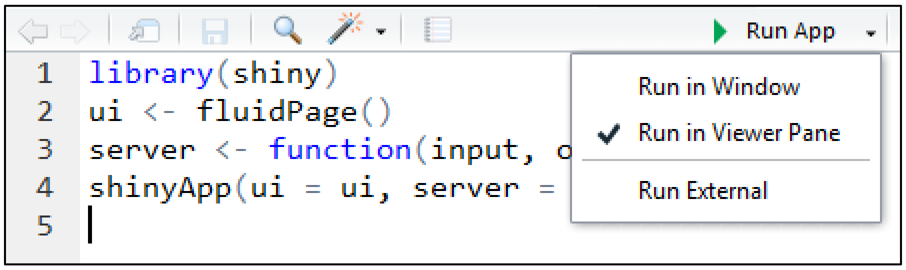
2.4.1 Using RStudio's Run App Button

If we have saved our file with the correct name RStudio will detect that it is a shiny app and display the Run App button.



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

We can click on the dropdown arrow next to Run App to choose where we want the app to run.



```
library(shiny)
ui <- fluidPage()
server <- function(input, o
shinyApp(ui = ui, server =
```

Run in Window will run in RStudio in a new window
Run in Viewer Pane will run in RStudio's integrated Viewer tab
Run External will run the app in your computer's default web browser

There can be minor differences in how elements are displayed between different browsers. If your users have a particular browser then set this as your default browser in your operating system and choose Run External to see your app exactly how your users will see it.

2.4.2 Using the runApp Function

We can also run an application using the function `runApp`. As a minimum we only need to provide the path to the directory that contains our app.R file. As with any file reading/writing in R, we can provide the full or relative file paths to this location.

```
> runApp("Minimal_App")

Listening on http://127.0.0.1:3968
```

Where the app opens will depend on RStudio's Run App button settings as we have seen before. We can override this behaviour using by setting the `launch.browser` argument:

```
> runApp("Minimal_App", launch.browser = TRUE)

Listening on http://127.0.0.1:3968
```

●

1. In a new R Script create the minimal example.
2. Save your script in a folder called "Minimal_App" and ensure the script is named correctly.
3. Run the app using RStudio's Run App button and experiment with its options. Use the Stop button to stop your app. Note the app will only show a blank screen at the moment!
4. Run the app from the console specifying that it should run in the external browser.

Extension:

5. Use `runApp` to run the "01_hello" example application we ran earlier.

2.5 Developing the App

Now that we have the bare bones of our app we will develop it into the basic interactive histogram we have seen before.

Splitting an application into the UI and server components is very useful as it allows us to think about how the app *looks* and how the app *works* separately. Firstly we will build out how the app *looks* by adding UI components.

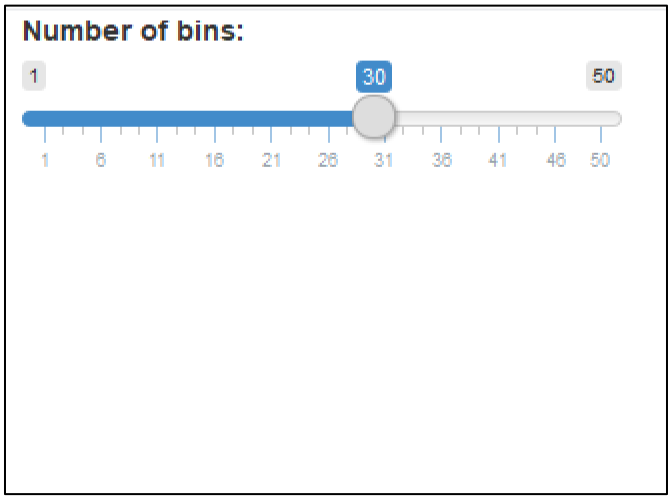
2.5.1 Adding UI Components

The user interface is specified by providing UI components as *arguments* to the `fluidPage` function. Each UI component is a single argument, so each component must be separated by a comma, just like any other R function.

We will add the slider input and a space for the histogram output to be presented. Each UI component uses one of shiny's built-in functions to define it. We will add the `sliderInput` and `plotOutput` components. The new code is in bold.

```
library(shiny)
# Define the user interface component
ui <- fluidPage(
  sliderInput(inputId = "bins", label = "Number of bins:",
    min = 1, max = 50, value = 30),
  plotOutput(outputId = "hist")
)

# Define the server component
server <- function(input, output) {}
# Combine the two components
shinyApp(ui = ui, server = server)
```

Running the app will look similar to the following:



The plot output does not display anything yet, we have just allocated some space for it to appear after the slider. We will define how to render the plot in the server component.

When your app is running RStudio's Run App button changes to a "Reload App" button. You can develop your app while it is running and then click "Reload App" to see the changes.

2.5.2 *Adding the Server Component*

In the server part of the app we need to connect the inputs and outputs together and define how to render the output histogram. We do this by adding R code to the *body* of the `server` function.

The `server` function has parameters named “input” and “output” which allow us to refer to the input and output elements of our user interface within our server code. Each parameter is represented as a named list, just like any other list in R.

2.5.2.1 Inputs

The input list contains elements for each of our input elements defined in our user interface, named by their `inputId`. In this case we have just one element called `input$bins`. When our app is running, `input$bins` will hold the current value of our input slider.

2.5.2.2 Outputs

The output list provides the mechanism for defining how our outputs should be rendered. For example, we can define how our histogram is rendered by assigning behaviour to the `output$hist` element, where “hist” corresponds to our plot’s `outputId` defined in our user interface.

The table below summarises the input and output elements we have so far in our app, and how they are referred to in the UI and the server components.

User Interface	Server
`sliderInput(inputId = ”bins”)`	`input$bins`
`plotOutput(outputId = ”hist”)`	`output$hist`

2.5.2.3 Combining Inputs and Outputs

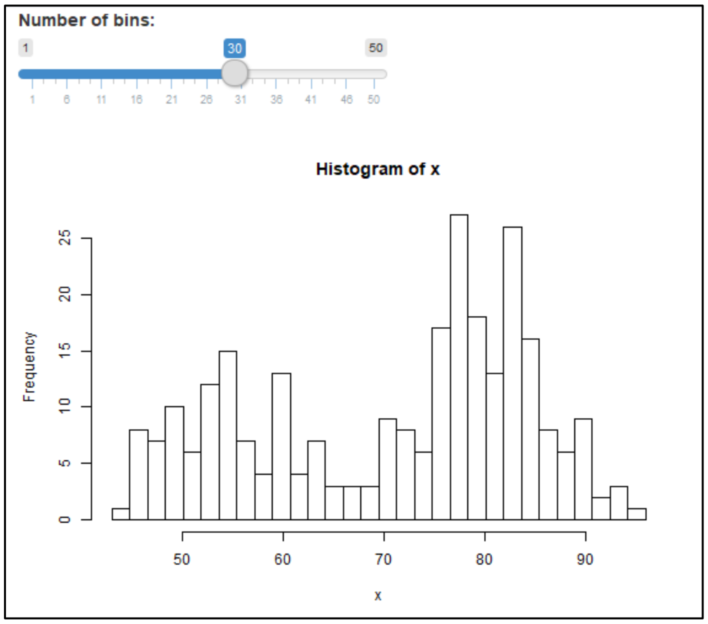
We will define how to render our histogram based on the number of bins selected by the input slider:

```
library(shiny)
# Define the user interface component
ui <- fluidPage(
  sliderInput(inputId = "bins", label = "Number of bins:",
    min = 1, max = 50, value = 30),
  plotOutput(outputId = "hist")
)

# Define the server component
server <- function(input, output) {
  output$hist <- renderPlot({
    x <- faithful$waiting
    binBreaks <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = binBreaks)
  })
}
# Combine the two components
shinyApp(ui = ui, server = server)
```

We have created a sequence of `binBreaks` to feed into R's standard `hist` function. Crucially, we have included `input$bins` as part of this sequence definition, and this is what links the input slider to the output histogram when our app runs.

The `renderPlot` function is from the **shiny** package. We can include any R code within it, provided the final result actually produces a plot, for example by calling `hist`.

Running the app should now look like this:



1. Create and run a new app as outlined above in a folder called Eruptions_Hist.
2. Change the app to plot `eruptions` instead of `waiting` time.
3. Adjust the slider input so it starts at 20 and goes up to 40.
4. Change the colour of the histogram to red.
5. What happens if you remove `input$bins` from the `binBreaks` calculation?

Extension

6. Add another sliderInput to the UI, ranging from 1 to 10. Link it to the histogram in the server by using it to control the histogram's colour. Hint: colours in `hist` can be defined by integers as well as text.

2.6 Mutiple File Applications

So far we have just used a single `app.R` file to hold both the UI and the server components of our app. When our apps start to become more complex it can be useful to split the UI and the server components into their own files, named `ui.R` and `server.R`.

The example below shows the same application that we created above but this time splitting the UI and server components into two files.

```
# ui.R
library(shiny)
fluidPage(
  sliderInput(inputId = "bins", label = "Number of bins:",
    min = 1, max = 50, value = 30),
  plotOutput(outputId = "hist")
)
```

```
# server.R
server <- function(input, output) {
  output$hist <- renderPlot({
    x <- faithful$waiting
    binBreaks <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = binBreaks)
  })
}
```

We can launch the app in the same way as before, in this case either from within `ui.R` or from `server.R`. If we run the app from `ui.R` RStudio will look for the `server.R` file in the same directory, and vice versa.

Chapter 3
Input and Output Widgets

3.1 Overview

Once we know the building blocks for a simple shiny application we can start to build more complex, interactive applications.

In this chapter we will look at how we can expand the types of input our users can use to interact with our application, such as selecting dates or uploading files. We will also look at how we can display different output types such as tables, text and graphics. Importantly we will look at how we can combine them so our inputs and outputs are working together.

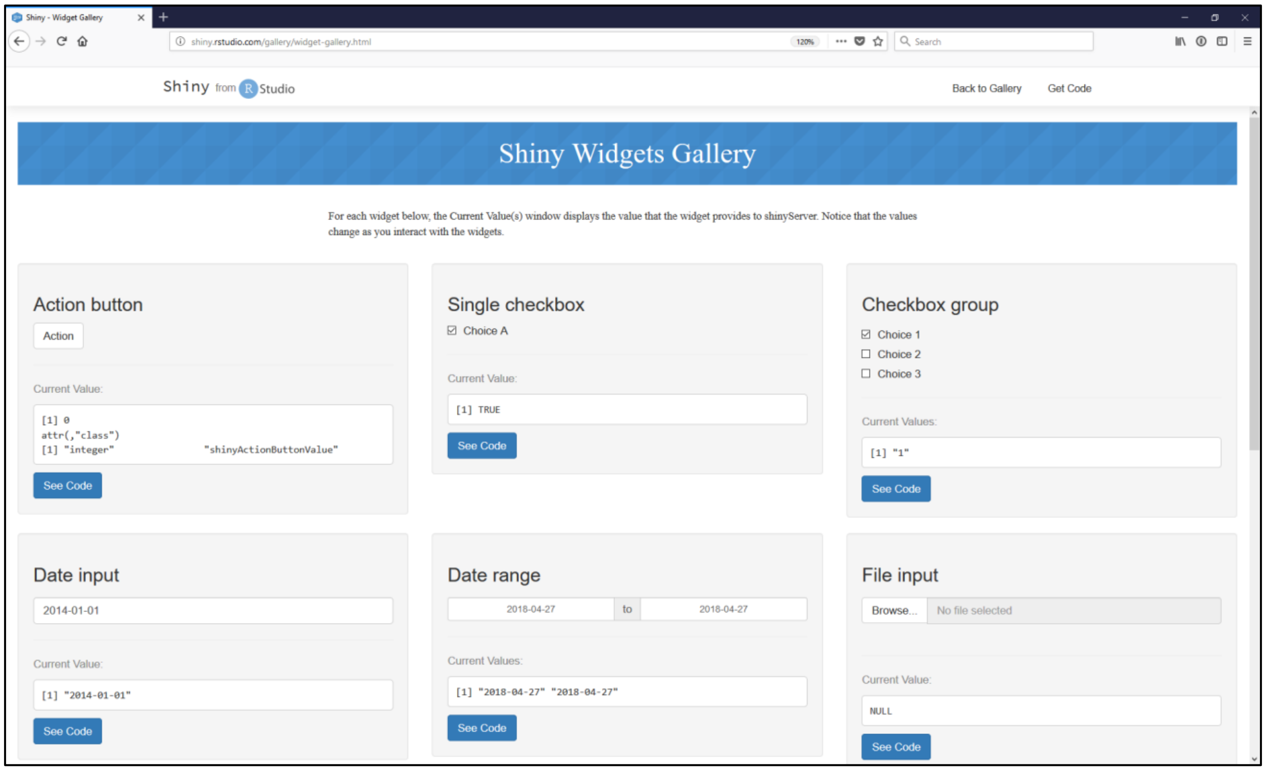
3.2 Defining User Inputs

3.2.1 Input Widget Options

There are a range of input types we can use:

Input Function	Description
`textInput`	Text string input
`numericInput`	Numeric value input (with optional validation)
`selectInput`	Select single or multiple values from drop down list
`sliderInput`	Numeric range "slider" input
`dateInput`	Date selector with popup calendar widget
`dateRangeInput`	Date range selector with popup calendar widget
`radioButtons`	Set of radio button inputs
`fileInput`	File upload control
`checkboxInput`	Single check box input
`checkboxGroupInput`	Group of check box inputs

The Shiny Widgets Gallery (shiny.rstudio.com/gallery/widget-gallery.html) provides an excellent demonstration of the various inputs and the types of values they return for us to use in our apps.



Shiny Widgets Gallery

For each widget below, the Current Value(s) window displays the value that the widget provides to shinyServer. Notice that the values change as you interact with the widgets.

Action button

Current Value:

```
[1] 0
attr(,"class")
[1] "integer"              "shinyActionButtonValue"
```

Single checkbox

☑ Choice A

Current Value:

```
[1] TRUE
```

Checkbox group

☑ Choice 1
☐ Choice 2
☐ Choice 3

Current Values:

```
[1] "1"
```

Date input

2014-01-01

Current Value:

```
[1] "2014-01-01"
```

Date range

2018-04-27 to 2018-04-27

Current Values:

```
[1] "2018-04-27" "2018-04-27"
```

File input

Current Value:

```
NULL
```

3.2.2 *Understanding Input Widgets*

Each input widget is an R function that returns the right web markup (HTML) to be displayed in our web browser. Calling a widget function in the console will show you the HTML it produces:

```
> sliderInput(inputId = "slider", label = "Pick a number",
    min = 1, max = 10, value = 5)
<div class="form-group shiny-input-container">
  <label class="control-label" for="slider">Pick a number</label>
  <input class="js-range-slider" id="slider" data-min="1" data-
max="10" data-from="5" data-step="1" data-grid="true" data-grid-
num="9" data-grid-snap="false" data-prettify-separator="," data-
prettify-enabled="true" data-keyboard="true" data-keyboard-
step="11.1111111111111" data-data-type="number"/>
</div>
```

This is how we can write web applications using just R code and not have to worry about the underlying web technologies.

Since each widget is a function, we can get more information on what a widget can do by consulting its help page.

```
> help("sliderInput")
```

●

1. Build a simple Shiny application called “Hist_Inputs” that plots a histogram of the Old Faithful geyser wait times (`faithful$waiting`) allowing the user to specify the number of bins using a slider from 10 to 50.
2. Adjust the slider widget so that it increments in steps of 5.
3. Swap the slider widget for a numeric input with the same constraints as before
4. Add a text input widget to appear before the slider, allowing the user to specify the main title for the histogram and reload the app.
5. Connect the text widget through to the histogram in your server component.

3.3 Defining Outputs

3.3.1 Output Widget Options

In the previous exercise we used the `renderPlot` function in the server to describe how to render our histogram, and the `plotOutput` function to place it in our user interface. In addition to plots we have the following options.

Output Type	Function to Layout (in UI)	Render Function (in server)
Text	`verbatimTextOutput, textOutput`	`renderText, renderPrint`
Data	`dataTableOutput`	`renderDataTable`
Table	`tableOutput`	`renderTable`
Image	`imageOutput`	`renderImage`
Plot	`plotOutput`	`renderPlot`

As with input widgets, we can find out more about each output widget by consulting the function documentation.

3.3.2 Rendering Text

There are two render functions for rendering text and two output functions for laying out the text outputs in the user interface.

3.3.2.1 Text User Interface Options

- `textOutput` places rendered text into the application’s layout. This is the standard way of outputting text into the user interface.

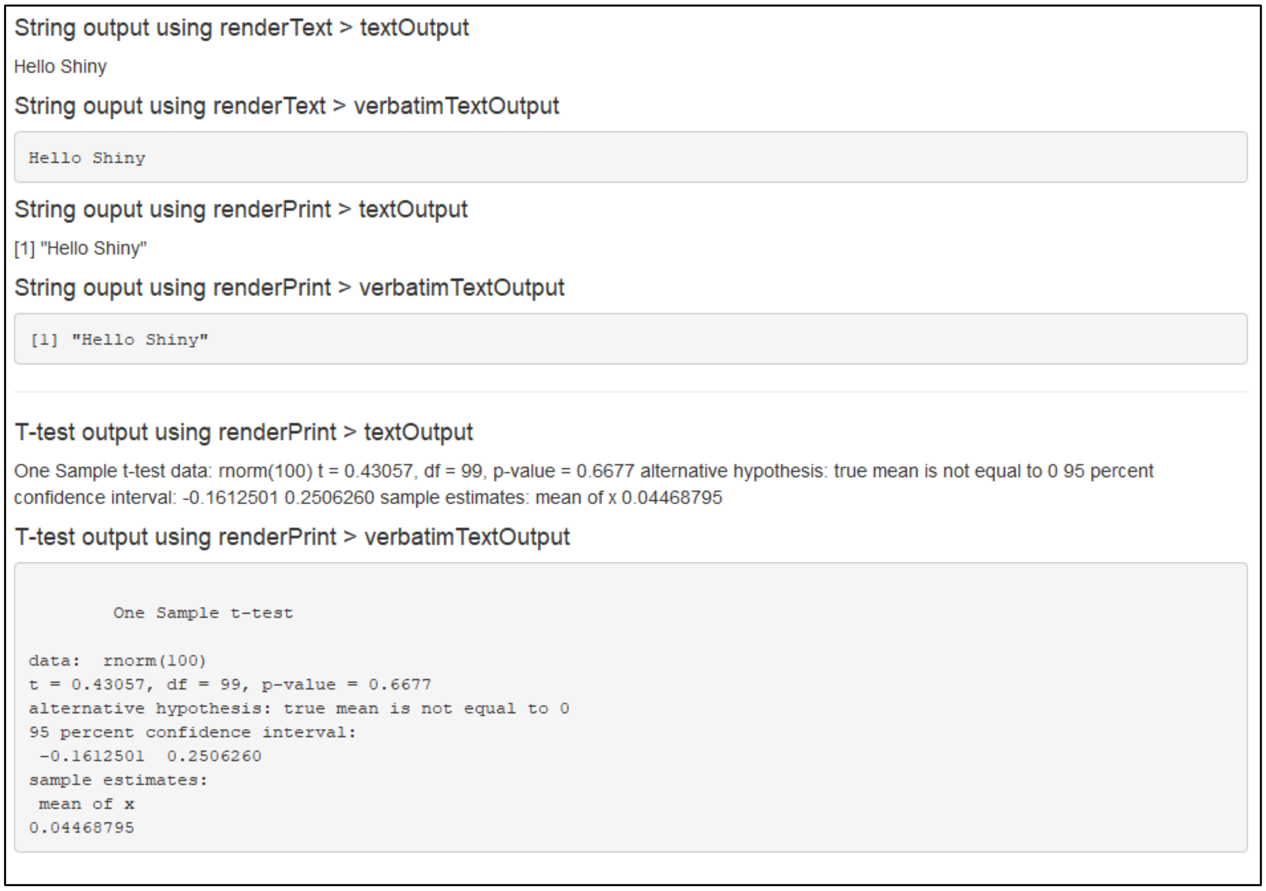
- `verbatimTextOutput` outputs “preformatted” text using a fixed-width font, similar to how objects print in the console. This is often used for printing outputs from function calls.

3.3.2.2 Text Server Rendering Options

- `renderPrint` will capture the print outputs of your functions, in the same way as `renderPlot` captured our plot outputs for our histogram. Think of this as the Shiny equivalent for `print`.
- `renderText` will capture the raw outputs of your functions and render them as text. Think of this as the Shiny equivalent for `cat`.

We will now build a Shiny app to illustrate their differences.

```
library(shiny)
ui <- fluidPage(
  h4("String output using renderText > textOutput"),
  textOutput("strRenderText1"),

  h4("String ouput using renderText > verbatimTextOutput"),
  verbatimTextOutput("strRenderText2"),

  h4("String ouput using renderPrint > textOutput"),
  textOutput("strRenderPrint1"),

  h4("String ouput using renderPrint > verbatimTextOutput"),
  verbatimTextOutput("strRenderPrint2"),

  hr(), # Horizontal Rule to divide the page

  h4("T-test output using renderPrint > textOutput"),
  textOutput("ttRenderPrint1"),

  h4("T-test output using renderPrint > verbatimTextOutput"),
  verbatimTextOutput("ttRenderPrint2")
)
server <- function(input, output) {
  # Strings
  txt <- "Hello Shiny"
  output$strRenderText1 <- renderText({txt})
  output$strRenderText2 <- renderText({txt})
  output$strRenderPrint1 <- renderPrint({txt})
  output$strRenderPrint2 <- renderPrint({txt})
  # Model outputs
  tt <- t.test(rnorm(100))
  output$ttRenderPrint1 <- renderPrint({tt})
  output$ttRenderPrint2 <- renderPrint({tt})
}
shinyApp(ui, server)
```


String output using renderText > textOutput

Hello Shiny

String ouput using renderText > verbatimTextOutput

```
Hello Shiny
```

String ouput using renderPrint > textOutput

[1] "Hello Shiny"

String ouput using renderPrint > verbatimTextOutput

```
[1] "Hello Shiny"
```

T-test output using renderPrint > textOutput

One Sample t-test data: rnorm(100) t = 0.43057, df = 99, p-value = 0.6677 alternative hypothesis: true mean is not equal to 0 95 percent confidence interval: -0.1612501 0.2506260 sample estimates: mean of x 0.04468795

T-test output using renderPrint > verbatimTextOutput

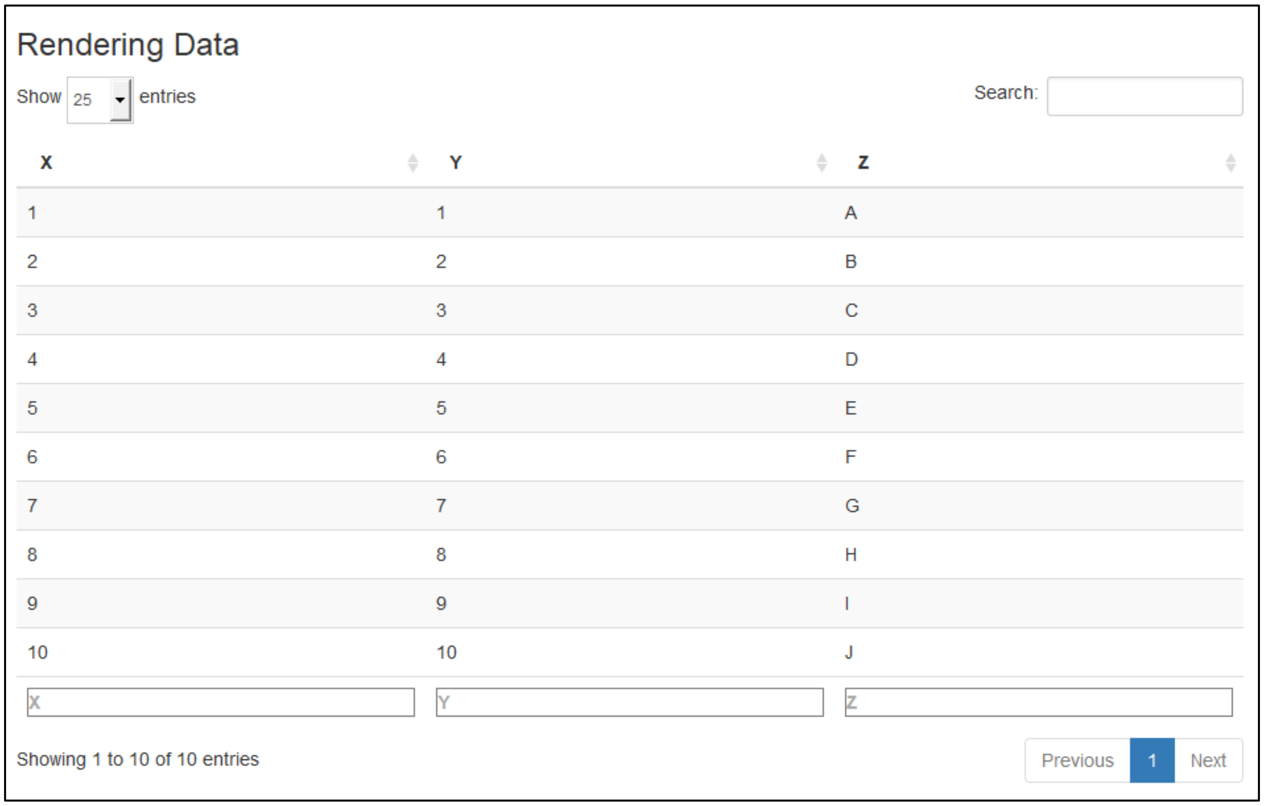
```

	One Sample t-test

data:  rnorm(100)
t = 0.43057, df = 99, p-value = 0.6677
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.1612501  0.2506260
sample estimates:
 mean of x 
0.04468795 
```

Note that we cannot use `renderText` on the t-test object, this would be the equivalent of calling `cat` on the `t.test` results, which outputs an object too complex for `cat` to deal with.

3.3.3 Rendering Data

We can use the `dataTableOutput` function to present data in the UI, and use the `renderDataTable` function in the server component to render it.

```
library(shiny)
ui <- fluidPage(
  h4("Data Frame"),
  dataTableOutput("dfHead")
)
server <- function(input, output) {
  myDf <- data.frame(X = 1:10, Y = 1:10, Z = LETTERS[1:10] )
  output$dfHead <- renderDataTable({myDf})
}
shinyApp(ui, server)
```


Rendering Data

Show 25 entries Search:

X	Y	Z
1	1	A
2	2	B
3	3	C
4	4	D
5	5	E
6	6	F
7	7	G
8	8	H
9	9	I
10	10	J
X	Y	Z

Showing 1 to 10 of 10 entries

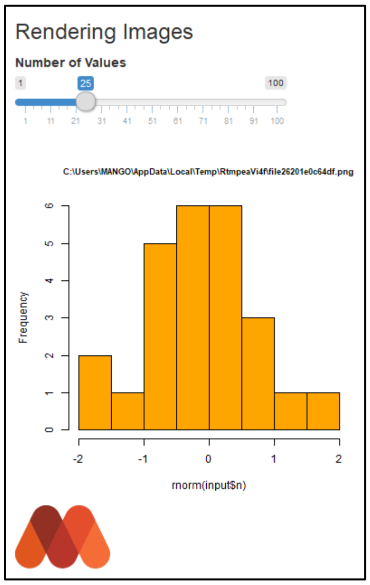
The result is an interactive table with built-in sorting, filtering and search functionality.

For a more basic table output we can use `tableOutput` and `renderTable` functions in the UI and server respectively. These produce tables in a simple format without the filtering and sorting illustrated above and are often used to output the results of R's `table`, or `xtabs` functions.

3.3.4 Rendering Images

We can use the `imageOutput` and `renderImage` functions to present an image to the user. This can either be an existing image or one that we create dynamically as part of our application:

```
library(shiny)
ui <- fluidPage(
  h3("Rendering Images"),
  sliderInput(inputId = "n", label = "Number of Values", 1, 100, 25),
  imageOutput("thePlot"),
  imageOutput("mangoLogo")
)
server <- function(input, output) {
  # render existing image (from working directory)
  output$mangoLogo <- renderImage({list(src = "mango_logo.png")},
                                  deleteFile = FALSE)
  # render an image created on the fly
  output$thePlot <- renderImage({
    outfile <- tempfile(fileext = '.png')
    png(outfile, width = 400 , height = 400)
    hist(rnorm(input$n), col = "orange", main = outfile)
    dev.off()
    list(src = outfile, alt = "I just created this plot")
  }, deleteFile = TRUE)
}
shinyApp(ui, server)
```



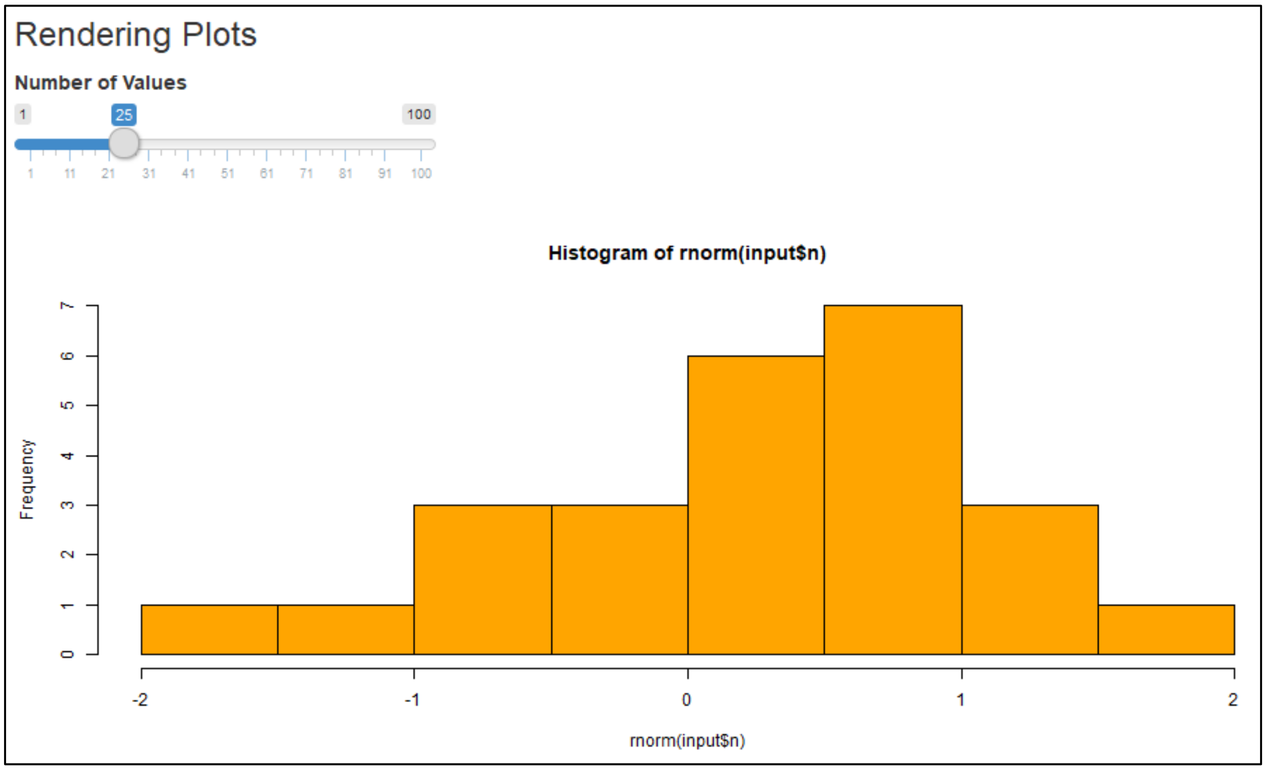
●

Then `renderImage` function is designed to work with transient image files (such as those created by R on the fly). As such, renderImage has an argument "deleteFile" which is set to TRUE by default – this deletes the image after the app is closed.

3.3.5 *Rendering Plots*

We can use the `plotOutput` and `renderPlot` functions to produce plots. This is distinct from the *imageOutput*/*renderImage* functionality in the last example which was used to render image *files*. With `plotOutput`/`renderPlot` we are working with plot *objects*, not files.

```
library(shiny)
ui <- fluidPage(
  sliderInput(inputId = "n", label = "Number of Values", 1, 100, 25),
  plotOutput("thePlot")
)
server <- function(input, output) {
  output$thePlot <- renderPlot({
    hist(rnorm(input$n), col = "orange")
  })
}
shinyApp(ui, server)
```



●

1. Create a “Hist_Text” Shiny application takes:
   a. A numeric value between 1 and 500
   b. A colour
   c. A main title
2. Use these inputs to create an output histogram
3. Add a check box – if the check box is selected, add a vertical reference line at the median of the data
4. Add an appropriate text output to the app that displays the R code used to create the displayed histogram, including the interactive parameters.

Extension

5. Create a “Timeseries_Simulation” app that plots a simulated time series using data drawn from a particular distribution. Allow the user to specify the date range with an appropriate widget, and the distribution using radio buttons (with options “Normal”, “Poisson” and “Uniform”). Output the resulting series in a line chart.

Chapter 4
Reactivity

4.1 Reactivity

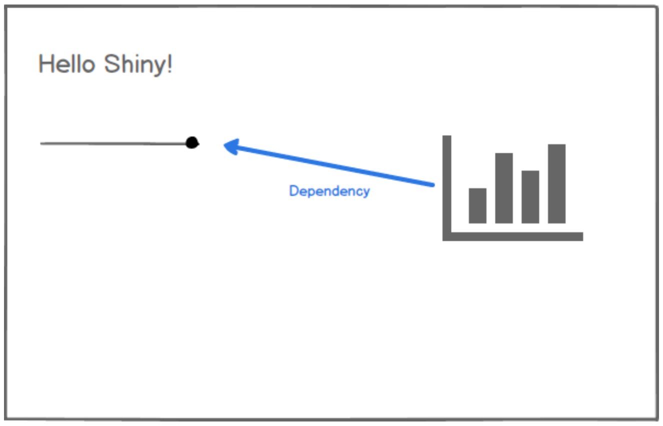
4.1.1 What is Reactivity?

Shiny applications are different to normal R code in that they connect directly with our users' actions when they interact with our app. Ordinarily our R code runs only when we, as programmers, tell it to. In Shiny we give up control of when our code will run as this becomes dependent on external events, such as a user dragging an input slider.

We have seen how to connect our output to user inputs by including the input values as part of our rendering instructions in our server. For example, the snippet below describes how to render a histogram based on a sampleSize input.

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

We can say that the code to render the histogram *depends* on the input slider. When the `sampleSize` slider changes the app *reacts* by redrawing the histogram.



When we build larger apps these dependencies become more complex, and we can create chains of dependencies where one thing depends on another, which depends on another. The **shiny** framework calculates this dependency graph behind the scenes and figures out what needs to change when our app changes state. This process is known as reactivity.

4.1.2 *Reactive Objects and Reactive Contexts*

The input widgets we have seen so far are examples of reactive objects. These are the “live” aspects of our app that drive the interactivity that is unique to **shiny** applications. Input widgets are not the only type of reactive object, we will look at others later in the chapter.

Reactive objects, such as an input widget, must always be evaluated within a *reactive context*. Consider a simple example of plotting a histogram of some random numbers:

```
sampleSize <- 100
hist(rnorm(sampleSize))
```

Ordinarily, when we run this code it will run once and only once. Now imagine that, instead of being assigned the value of `100`, `sampleSize` can have its value changed at any point in time. How will R know when to recreate the histogram?

We need to encapsulate the code within a reactive expression, which will allow **shiny** to take care of re-running the histogram whenever `sampleSize` changes.

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

In this example, `renderPlot` is the **shiny** reactive expression providing the reactive context around input$sampleSize. So far the reactive expressions we’ve seen so far are `render` family (`renderPlot`, `renderText`, `renderTable`, etc.), but we will see other types later.

A common mistake when starting with Shiny programming is to try to evaluate a reactive value outside of a reactive context, for example:

```
library(shiny)
ui <- fluidPage(
  numericInput("sampleSize", "Select size of data:",
               min = 10, max = 500, value = 100),
  plotOutput("hist")
)
server <- function(input, output){
  data <- rnorm(input$sampleSize)  # outside renderPlot – bad!
  output$hist <- renderPlot({
    hist(data)
  })
}
shinyApp(ui, server)

Listening on http://127.0.0.1:3085
Warning: Error in .getReactiveEnvironment()$currentContext: Operation
not allowed without an active reactive context. (You tried to do
something that can only be done from inside a reactive expression or
observer.)
```

The error message is very informative, explaining that we have tried to evaluate `input$sampleSize` outside of a reactive context. The solution is to move the line (in bold) inside the `renderPlot` block of code.

4.2 Controlling Reactivity

So far the apps we've made have been hyper-reactive – whenever an input has changed all of our outputs have reacted. Often we want to have more control over our app's reactivity and we will look at two common use cases:

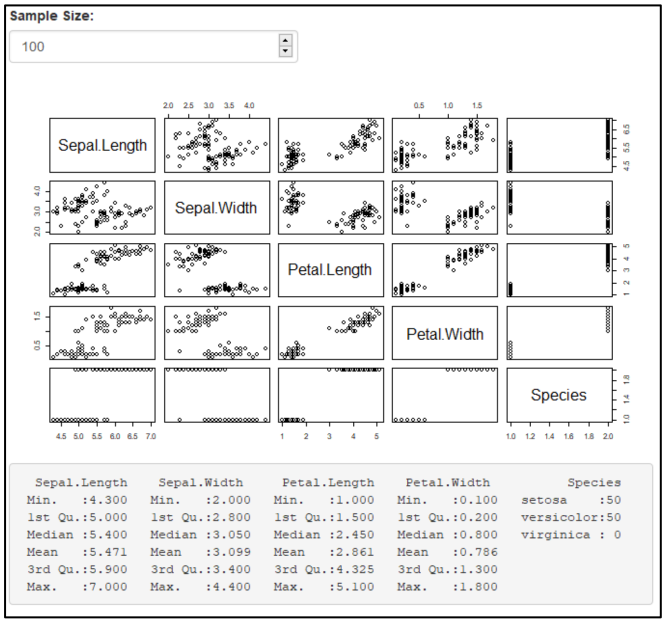
1. We want to produce several outputs that all depend on the same input data calculation.
2. We want to delay updates to our plot outputs until we have finished entering our inputs.

4.2.1 Creating Reactive Expressions

A common example for needing greater control over reactivity is when we have multiple outputs that rely on the results of a shared calculation that we have carried out, e.g. computing selected summary statistics, or filtering a dataset by user-specified criteria.

In the app below we have a plot output and a text output both summarising a random sample of the `iris` dataset.

```
ui <- fluidPage(
  numericInput("sampleSize", "Select size of data:",
               min = 50, max = 150, value = 100),
  plotOutput("pairs"),
  verbatimTextOutput("summary")
)
server <- function(input, output){
  output$pairs <- renderPlot({
    pairs(iris[sample(input$sampleSize), ])
  })
  output$summary <- renderPrint({
    summary(iris[sample(input$sampleSize), ])
  })
}
shinyApp(ui, server)
```



```
 Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
1st Qu.:5.000   1st Qu.:2.800   1st Qu.:1.500   1st Qu.:0.200   versicolor:50
Median :5.400   Median :3.050   Median :2.450   Median :0.800   virginica : 0
Mean   :5.471   Mean   :3.099   Mean   :2.861   Mean   :0.786
3rd Qu.:5.900   3rd Qu.:3.400   3rd Qu.:4.325   3rd Qu.:1.300
Max.   :7.000   Max.   :4.400   Max.   :5.100   Max.   :1.800
```

Can you see what is wrong with the design of this application?

The problem with the app above is that the plot and the text summary are each based on a *different* sample from the iris dataset, but we want to summarise the *same* dataset in both outputs (i.e. visually and numerically).

We might consider pulling the duplicated sampling code line out of the render functions and creating a shared common dataset beforehand. However, because the sample is based on the reactive `input$sampleSize`, it must be evaluated inside a reactive context.

The only reactive contexts we have seen so far have been the render functions, but in this case we just want to create a shared version of the sampling code, not render anything. We can do this by creating our own reactive expression using `reactive`.

```
ui <- fluidPage(
  numericInput("sampleSize", "Sample Size:",
               min = 50, max = 150, value = 100),
  plotOutput("pairs"),
  verbatimTextOutput("summary")
)
server <- function(input, output){

  getSample <- reactive({iris[sample(input$sampleSize), ]})

  output$pairs <- renderPlot({
    pairs(getSample())
  })
  output$summary <- renderPrint({
    summary(getSample())
  })
}
shinyApp(ui, server)
```

The sampling code has been pulled out of the render functions and we have created a new *reactive expression*. The `reactive` function provides a reactive context, so we can safely evaluate `input$sampleSize` within it.

A crucial point is that **reactive expressions cache their outputs**. A reactive expression will only recalculate its output if the reactive values it depends on change. In the example above, the first call to `getSample` will calculate the iris sample and, provided the `input$sampleSize` does not change, the subsequent call to `getSample` will return its cached value.

● A common reason to use reactive expressions is when there is an expensive processing operation, such as reading from a database/file, or performing a complex calculation. We can use reactive expression to do the processing once and cache the result.

1. Create a “Reactive_Hist” app that plots a histogram of randomly generated values. Add input options to control the number of values and the plot colour.
2. Add a textOutput showing the `summary` of the same data, ensuring that it is summarising the same data as the plot.
3. In your reactive expression, add a `Sys.sleep(5)` statement before generating the random values.
4. How long does the app take to refresh after changing the number of random values? What about changing the colour?

4.2.2 Delaying Reactivity

Another common scenario where we want more control over reactivity is to delay refreshes to our app’s output until the user explicitly requests it, e.g. by clicking a “Go” button. We can do this using a combination of `eventReactive` and `actionButton` functions.

The `eventReactive` function is identical to the `reactive` function we have just seen, with the exception that it will only react to inputs that we explitly define in its first argument.

Action buttons are simple input widgets created using `actionButton(inputId, label)`. When clicked their value increments by one. The actual value the action buttons takes is not important, they just give us a mechanism to trigger reactive events.

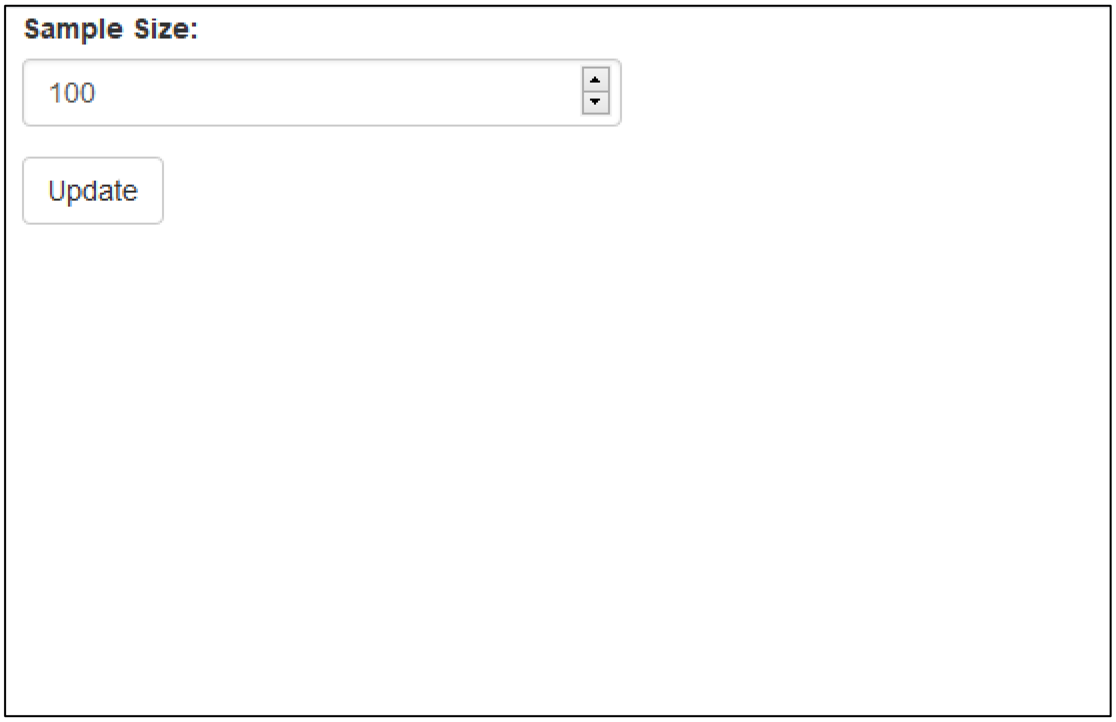
We can use an `actionButton` as the first argument to `eventReactive` in order to make it update only when the user clicks the button.

In the following example app we have the same iris-sampling app as before but this time we want the user to be able to set the input sample size before clicking a button to update the histogram, instead of it updating on every minor input adjustment.

Reactivity is delayed by converting the `reactive` expression to an `eventReactive` which depends updates when the `actionButton` is clicked:

```
ui <- fluidPage(
  numericInput("sampleSize", "Sample Size:",
                  min = 50, max = 150, value = 100),
  actionButton(inputId = "update", label = "Update"),
  plotOutput("pairs"),
  verbatimTextOutput("summary")
)
server <- function(input, output){
  getSample <- eventReactive(input$update, {
    iris[sample(input$sampleSize), ]
  })

  output$pairs <- renderPlot({
    pairs(getSample())
  })
  output$summary <- renderPrint({
    summary(getSample())
  })
}
shinyApp(ui, server)
```

The app will show a blank plot output until the user clicked the “Update” button. Changing the input will not update the plot until the user clicks “Update”.



1. Create a “Petal_Boxplot” app that creates a boxplot of Petal.Length by Species using a random sample from the iris dataset. Add a numeric input option to control the size of the sample (between 50 and 150), see the hint below for the example R code.
2. Ensure that the boxplot only displays after the user has selected their desired sample size.

Hint: you can base the plot on the following code snippet:

```
boxplot(Petal.Length ~ Species, data = ???)
```

4.3 Summary

- If we want to create an output that depends on reactive inputs we use the `render` family of functions.
- If we want to create intermediate calculations that depend on reactive inputs we use `reactive` to create a reactive expression.
- If we want to delay reactivity until the user requests it we can combine an `actionButton` with an `eventReactive`.