# LESSON 1 Welcome to Shiny

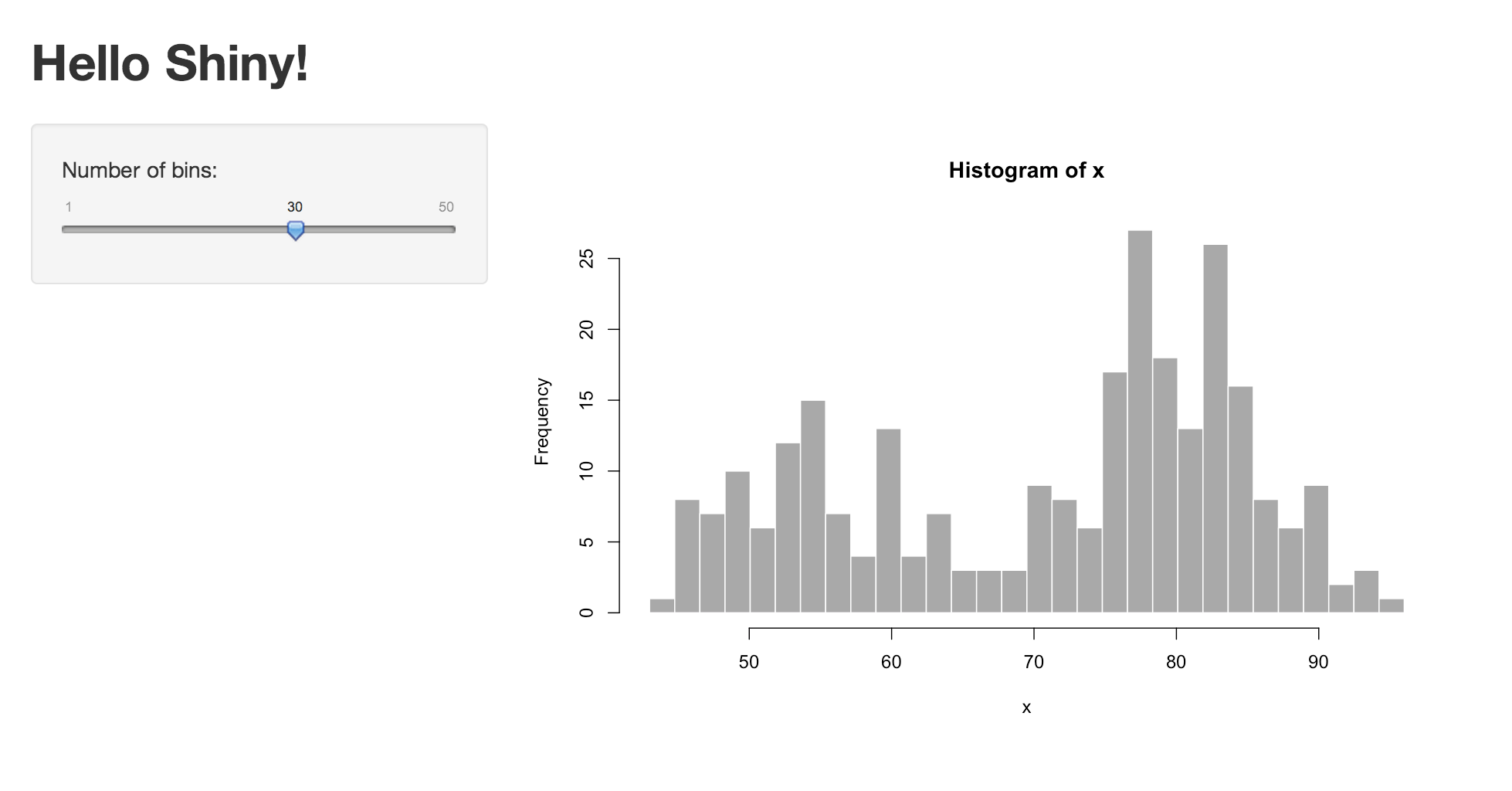
Shiny is an R package that makes it easy to build interactive web applications (apps) straight from R. This lesson will get you started building Shiny apps right away.

If you still haven’t installed the Shiny package, open an R session, connect to the internet, and run

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

Note that the preview release of RStudio IDE contains new features designed for Shiny. Download it [here](http://www.rstudio.com/ide/download/preview).

## Examples



The Shiny package has [eleven built-in examples](http://shiny.rstudio.com/tutorial/lesson1/#Go Further) that each demonstrate how Shiny works. Each example is a self-contained Shiny app.

The **Hello Shiny** example plots a histogram of R’s faithful dataset with a configurable number of bins. Users can change the number of bins with a slider bar, and the app will immediately respond to their input. You’ll use **Hello Shiny** to explore the structure of a Shiny app and to create your first app.

To run **Hello Shiny**, type:

**>** library(shiny)

**>** runExample("01\_hello")

## Structure of a Shiny App

Shiny apps have two components:

* a user-interface script
* a server script

The user-interface (ui) script controls the layout and appearance of your app. It is defined in a source script named ui.R. Here is the ui.R script for the **Hello Shiny** example.

#### ui.R

library(shiny)

*# Define UI for application that draws a histogram*

shinyUI(fluidPage(

*# Application title*

titlePanel("Hello Shiny!"),

*# Sidebar with a slider input for the number of bins*

sidebarLayout(

sidebarPanel(

sliderInput("bins",

"Number of bins:",

min = 1,

max = 50,

value = 30)

),

*# Show a plot of the generated distribution*

mainPanel(

plotOutput("distPlot")

)

)

))

The server.R script contains the instructions that your computer needs to build your app. Here is theserver.R script for the **Hello Shiny** example.

#### server.R

library(shiny)

*# Define server logic required to draw a histogram*

shinyServer(function(input, output) {

*# Expression that generates a histogram. The expression is*

*# wrapped in a call to renderPlot to indicate that:*

*#*

*# 1) It is "reactive" and therefore should re-execute automatically*

*# when inputs change*

*# 2) Its output type is a plot*

output$distPlot <- renderPlot({

x <- faithful[, 2] *# Old Faithful Geyser data*

bins <- seq(min(x), max(x), length.out = input$bins + 1)

*# draw the histogram with the specified number of bins*

hist(x, breaks = bins, col = 'darkgray', border = 'white')

})

})

At one level, the **Hello Shiny** server.R script is very simple. The script does some calculations and then plots a histogram with the requested number of bins.

However, you’ll also notice that most of the script is wrapped in a call to renderPlot. The comment above the function explains a bit about this, but if you find it confusing, don’t worry. We’ll cover this concept in much more detail soon.

Play with the **Hello Shiny** app and review the source code. Try to develop a feel for how the app works.

Your R session will be busy while the **Hello Shiny** app is active, so you will not be able to run any R commands. R is monitoring the app and executing the app’s reactions. To get your R session back, hit escape or click the stop sign icon (found in the upper right corner of the RStudio console panel).

## Running an App

Every Shiny app has the same structure: two R scripts saved together in a directory. At a minimum, a Shiny app has ui.R and server.R files.

**Note:** As of version 0.10.2, Shiny supports single-file applications. You no longer need to build separateserver.R and ui.R files for your app; you can just create a file called app.R that contains both the server and UI components. You can learn more about building a Shiny app in a single file [here](http://shiny.rstudio.com/articles/single-file.html), however this tutorial will focus on the two file structure for building a Shiny app.

You can create a Shiny app by making a new directory and saving a ui.R and server.R file inside it. Each app will need its own unique directory.

You can run a Shiny app by giving the name of its directory to the function runApp. For example if your Shiny app is in a directory called my\_app, run it with the following code:

**>** library(shiny)

**>** runApp("my\_app")

Note: runApp is similar to read.csv, read.table, and many other functions in R. The first argument ofrunApp is the filepath from your [working directory](http://www.rstudio.com/ide/docs/using/workspaces) to the app’s directory. The code above assumes that the app directory is in your working directory. In this case, the filepath is just the name of the directory.

(In case you are wondering, the **Hello Shiny** app’s files are saved in a special system directory called"01\_hello". This directory is designed to work with the runExample ("01\_hello") call.)

## Your Turn

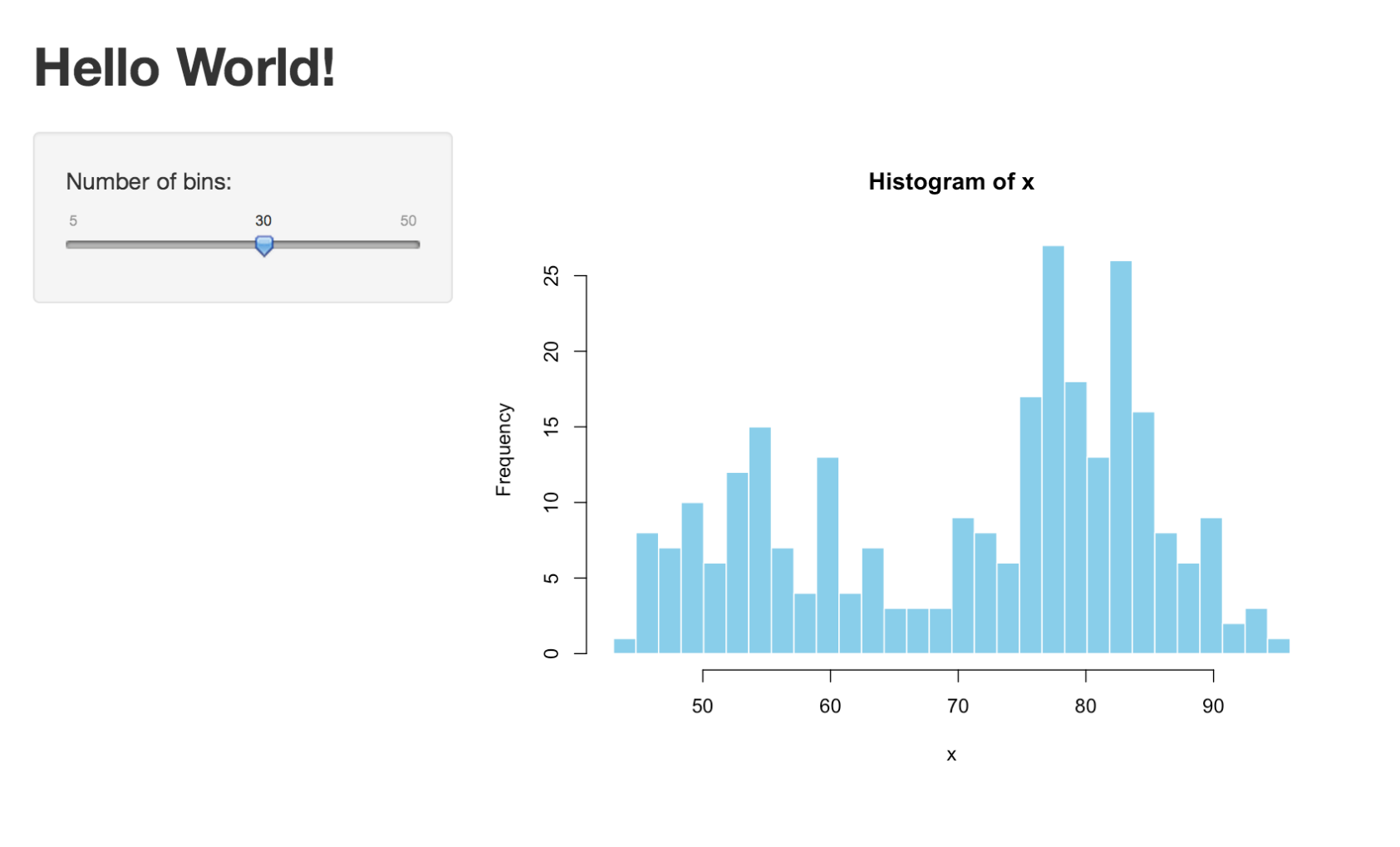
Create a new directory named App-1 in your [working directory](http://www.rstudio.com/ide/docs/using/workspaces). Then copy and paste the ui.R andserver.R scripts above into your directory (the scripts from **Hello Shiny**). When you are finished the directory should look like this:



Launch your app by running runApp("App-1"). Then click escape and make some changes to your app:

1. Change the title from “Hello Shiny!” to “Hello World!”.
2. Set the minimum value of the slider bar to 5.
3. Change the histogram color from "darkgray" to "skyblue".

When you are ready, launch your app again. Your new app should match the image below. If it doesn’t, or if you want to check your code, press the model answers button to reveal how we did these tasks.



By default, Shiny apps display in “normal” mode, like the app pictured above. **Hello Shiny** and the other built in examples display in “showcase mode”, a different mode that displays the server.R and ui.R` scripts alongside the app.

If you would like your app to display in showcase mode, you can runrunApp("App-1", display.mode = "showcase").

### Model Answers

Change the title of your app and the slider bar values in the ui.R script.

#### ui.R

library(shiny)

*# Define UI for application that draws a histogram*

shinyUI(fluidPage(

*# Application title*

titlePanel("Hello World!"),

*# Sidebar with a slider input for the number of bins*

sidebarLayout(

sidebarPanel(

sliderInput("bins",

"Number of bins:",

min = 5,

max = 50,

value = 30)

),

*# Show a plot of the generated distribution*

mainPanel(

plotOutput("distPlot")

)

)

))

Use the server.R script to change how your computer builds the histogram.

library(shiny)

*# Define server logic required to draw a histogram*

shinyServer(function(input, output) {

*# Expression that generates a histogram. The expression is*

*# wrapped in a call to renderPlot to indicate that:*

*#*

*# 1) It is "reactive" and therefore should*

*# re-execute automatically when inputs change*

*# 2) Its output type is a plot*

output$distPlot <- renderPlot({

x <- faithful[, 2] *# Old Faithful Geyser data*

bins <- seq(min(x), max(x), length.out = input$bins + 1)

*# draw the histogram with the specified number of bins*

hist(x, breaks = bins, col = 'skyblue', border = 'white')

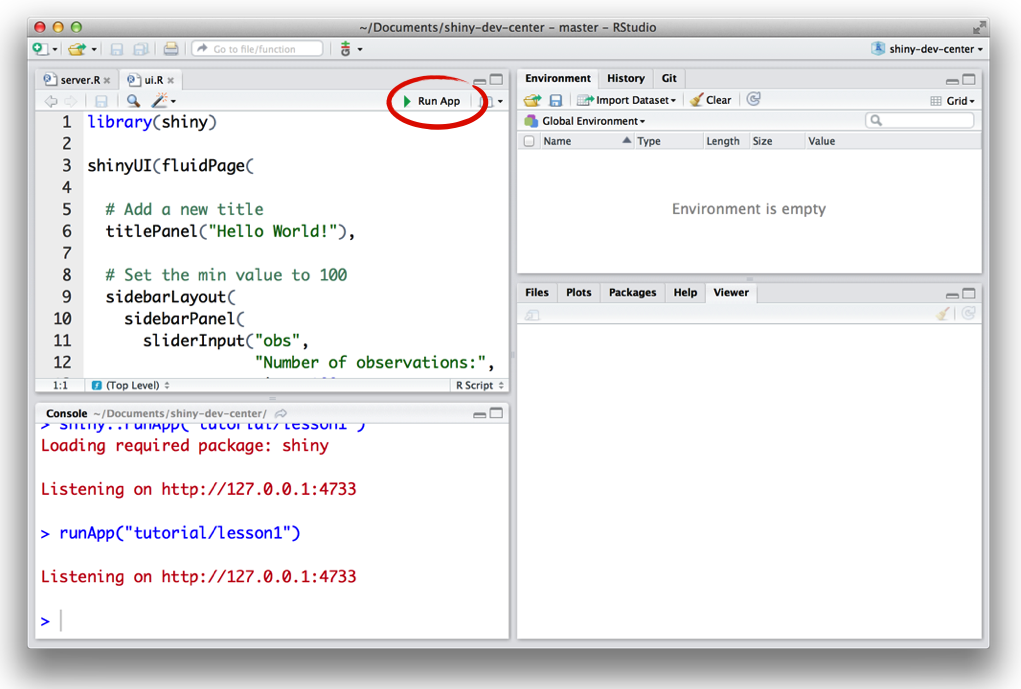
})

})

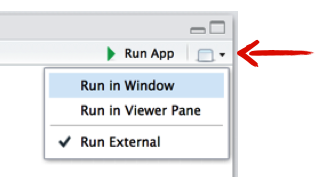
## Relaunching Apps

To relaunch your Shiny app:

* Run runApp("App-1"), or
* Open the ui.R or server.R scripts in your RStudio editor. RStudio will recognize the Shiny script and provide a Run App button (at the top of the editor). Either click this button to launch your app or use the keyboard shortcut: Command+Shift+Enter (Control+Shift+Enter on Windows).



RStudio will launch the app in a new window by default, but you can also choose to have the app launch in a dedicated viewer pane, or in your external web browser. Make your selection by clicking the icon next to Run App.



## Recap

To create your own Shiny app:

* Make a directory named for your app.
* Save your app’s server.R and ui.R script inside that directory.
* Launch the app with runApp or RStudio’s keyboard shortcuts.
* Exit the Shiny app by clicking escape.

## Go Further

You can create Shiny apps by copying and modifying existing Shiny apps. The Shiny [gallery](http://shiny.rstudio.com/gallery) provides some good examples, or use the eleven pre-built Shiny examples listed below.

system.file("examples", package="shiny")

runExample("01\_hello") *# a histogram*

runExample("02\_text") *# tables and data frames*

runExample("03\_reactivity") *# a reactive expression*

runExample("04\_mpg") *# global variables*

runExample("05\_sliders") *# slider bars*

runExample("06\_tabsets") *# tabbed panels*

runExample("07\_widgets") *# help text and submit buttons*

runExample("08\_html") *# Shiny app built from HTML*

runExample("09\_upload") *# file upload wizard*

runExample("10\_download") *# file download wizard*

runExample("11\_timer") *# an automated timer*

Each demonstrates a feature of Shiny apps. All Shiny example apps open in “showcase” mode (with theui.R and server.R scripts in the display).

But why limit yourself to copying other apps? The next few lessons will show you how to build your own Shiny apps from scratch. You’ll learn about each part of a Shiny app, and finish by deploying your own Shiny app online.

When you are ready, continue to [Lesson 2](http://shiny.rstudio.com/tutorial/lesson2/), where you will learn how to build the layout and appearance of your Shiny apps.

**LESSON 2**  
Build a user-interface

Now that you understand the structure of a Shiny app, it’s time to build your first app from scratch.

This lesson will show you how to build a user-interface for your app. You will learn how to lay out the user-interface and then add text, images, and other HTML elements to your Shiny app.

We’ll use the App-1 app you made in [Lesson 1](http://shiny.rstudio.com/tutorial/lesson1/). To get started, open its server.R and ui.R files. Edit the scripts to match the ones below:

**ui.R**

shinyUI(fluidPage(

))

**server.R**

shinyServer(function(input, output) {

})

This code is the bare minimum needed to create a Shiny app. The result is an empty app with a blank user-interface, an appropriate starting point for this lesson.

Layout

Shiny ui.R scripts use the function fluidPage to create a display that automatically adjusts to the dimensions of your user’s browser window. You lay out your app by placing elements in the fluidPagefunction.

For example, the ui.R script below creates a user-interface that has a title panel and then a sidebar layout, which includes a sidebar panel and a main panel. Note that these elements are placed within thefluidPage function.

*# ui.R*

shinyUI(fluidPage(

titlePanel("title panel"),

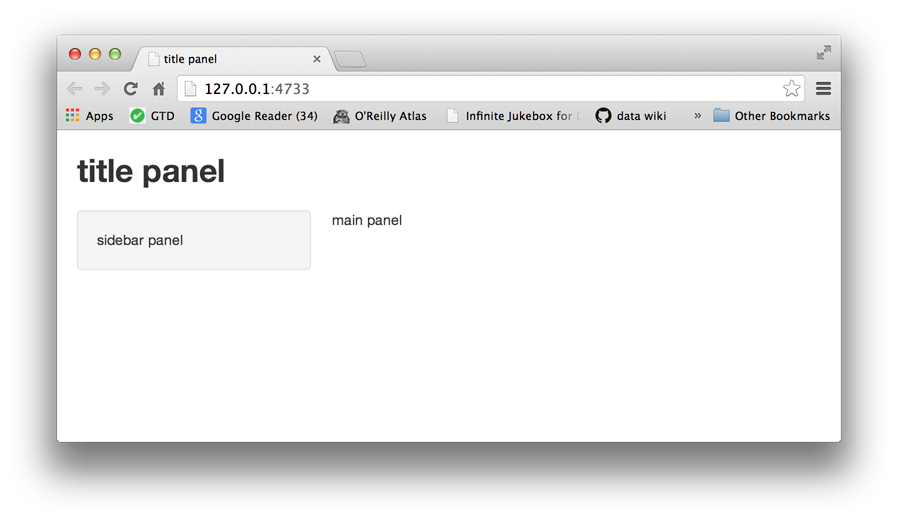
sidebarLayout(

sidebarPanel( "sidebar panel"),

mainPanel("main panel")

)

))



titlePanel and sidebarLayout are the two most popular elements to add to fluidPage. They create a basic Shiny app with a sidebar.

sidebarLayout always takes two arguments:

* sidebarPanel function output
* mainPanel function output

These functions place content in either the sidebar or the main panels. The sidebar panel will appear on the left side of your app by default. You can move it to the right side by giving sidebarLayout the optional argument position = "right".

*# ui.R*

shinyUI(fluidPage(

titlePanel("title panel"),

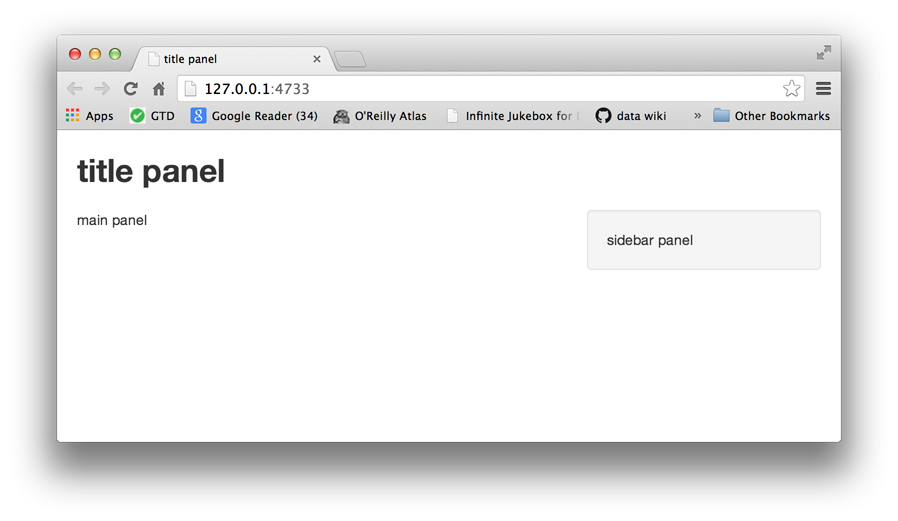
sidebarLayout(position = "right",

sidebarPanel( "sidebar panel"),

mainPanel("main panel")

)

))



titlePanel and sidebarLayout create a basic layout for your Shiny app, but you can also create more advanced layouts. You can use navbarPage to give your app a multi-page user-interface that includes a navigation bar. Or you can use fluidRow and column to build your layout up from a grid system. If you’d like to learn more about these advanced options, read the [Shiny Application Layout Guide](http://shiny.rstudio.com/articles/layout-guide.html). We will stick withsidebarLayout in this tutorial.

HTML Content

You can add content to your Shiny app by placing it inside a \*Panel function. For example, the apps above display a character string in each of their panels. The words “sidebar panel” appear in the sidebar panel, because we added the string to the sidebarPanel function, e.gsidebarPanel("sidebar panel"). The same is true for the text in the title panel and the main panel.

To add more advanced content, use one of Shiny’s HTML tag functions. These functions parallel common HTML5 tags. Let’s try out a few of them.

| **shiny function** | **HTML5 equivalent** | **creates** |
| --- | --- | --- |
| p | <p> | A paragraph of text |
| h1 | <h1> | A first level header |
| h2 | <h2> | A second level header |
| h3 | <h3> | A third level header |
| h4 | <h4> | A fourth level header |
| h5 | <h5> | A fifth level header |
| h6 | <h6> | A sixth level header |
| a | <a> | A hyper link |
| br | <br> | A line break (e.g. a blank line) |
| div | <div> | A division of text with a uniform style |
| span | <span> | An in-line division of text with a uniform style |
| pre | <pre> | Text ‘as is’ in a fixed width font |
| code | <code> | A formatted block of code |
| img | <img> | An image |
| strong | <strong> | Bold text |
| em | <em> | Italicized text |
| HTML |  | Directly passes a character string as HTML code |

Headers

To create a header element:

* select a header function (e.g., h1 or h5)
* give it the text you want to see in the header

For example, you can create a first level header that says “My title” with h1("My title"). If you run the command at the command line, you’ll notice that it produces HTML code.

> library(shiny)

> h1("My title")

<h1>My title</h1>

To place the element in your app:

* pass h1("My title") as an argument to titlePanel, sidebarPanel, or mainPanel

The text will appear in the corresponding panel of your web page. You can place multiple elements in the same panel if you separate them with a comma.

Give this a try. The new script below uses all six levels of headers. Update your ui.R to match the script and then relaunch your app. Remember to relaunch a Shiny app you may run runApp("App-1"), click the Run App button, or use your keyboard shortcuts.

*# ui.R*

shinyUI(fluidPage(

titlePanel("My Shiny App"),

sidebarLayout(

sidebarPanel(),

mainPanel(

h1("First level title"),

h2("Second level title"),

h3("Third level title"),

h4("Fourth level title"),

h5("Fifth level title"),

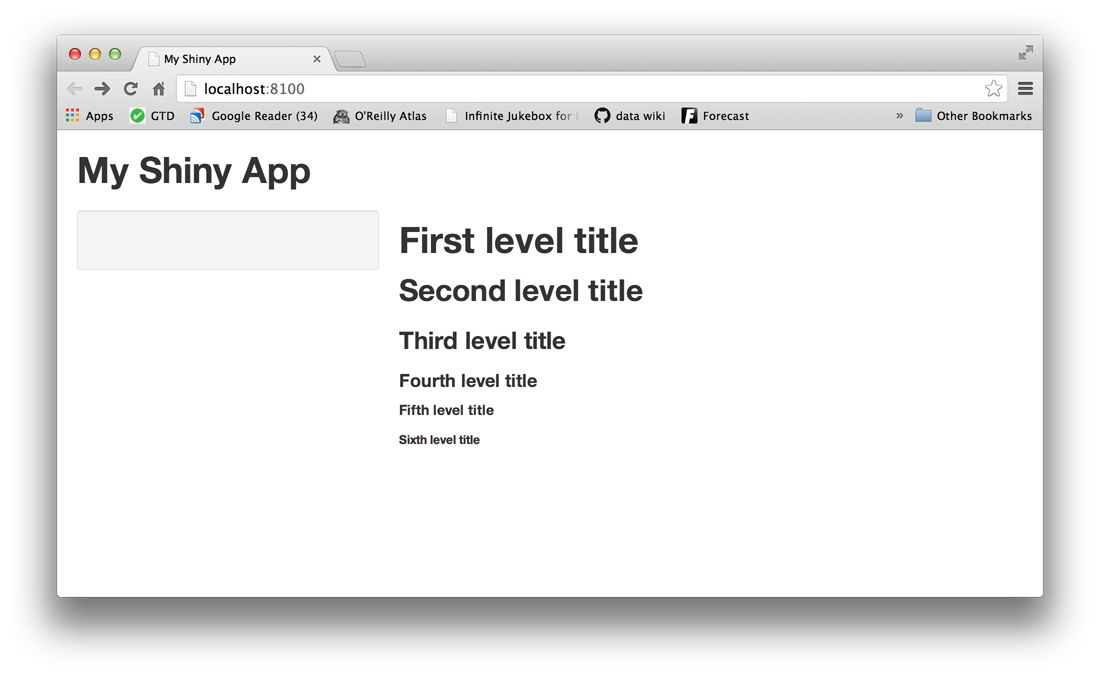
h6("Sixth level title")

)

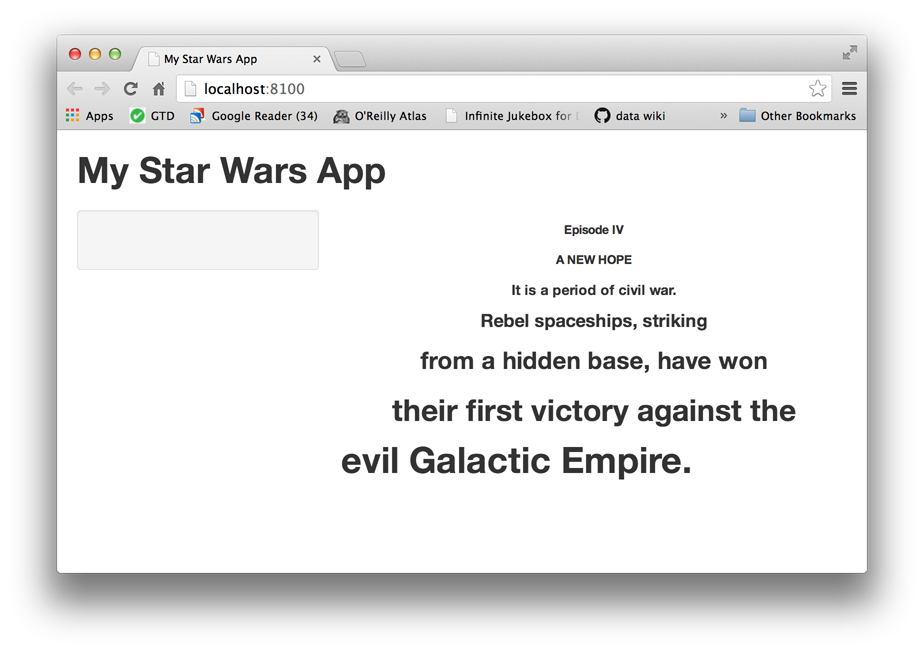
)

))

Now your app should look like this.



If George Lucas had a first app, it might look like this.



You can create this effect with align = "center", as in h6("Episode IV", align = "center"). In general, any HTML tag attribute can be set as an argument in any Shiny tag function.

If you are unfamiliar with HTML tag attributes, you can look them up in one of the many free online HTML resources such as [w3schools](http://www.w3schools.com/tags/tag_hn.asp).

Here’s the code that made the Star Wars-inspired user-interface:

*# ui.R*

shinyUI(fluidPage(

titlePanel("My Shiny App"),

sidebarLayout(

sidebarPanel(),

mainPanel(

h6("Episode IV", align = "center"),

h6("A NEW HOPE", align = "center"),

h5("It is a period of civil war.", align = "center"),

h4("Rebel spaceships, striking", align = "center"),

h3("from a hidden base, have won", align = "center"),

h2("their first victory against the", align = "center"),

h1("evil Galactic Empire.")

)

)

))

Formatted text

Shiny offers many tag functions for formatting text. The easiest way to describe them is by running through an example.

Paste the ui.R script below into your ui.R file and save it. If your Shiny app is still running, you can refresh your web page or preview window, and it will display the changes. If your app is closed, just relaunch it.

Compare the displayed app to your updated ui.R script to discover how to format text in a Shiny app.

*# ui.R*

shinyUI(fluidPage(

titlePanel("My Shiny App"),

sidebarLayout(

sidebarPanel(),

mainPanel(

p("p creates a paragraph of text."),

p("A new p() command starts a new paragraph. Supply a style attribute to change the format of the entire paragraph.", style = "font-family: 'times'; font-si16pt"),

strong("strong() makes bold text."),

em("em() creates italicized (i.e, emphasized) text."),

br(),

code("code displays your text similar to computer code"),

div("div creates segments of text with a similar style. This division of text is all blue because I passed the argument 'style = color:blue' to div", style = "color:blue"),

br(),

p("span does the same thing as div, but it works with",

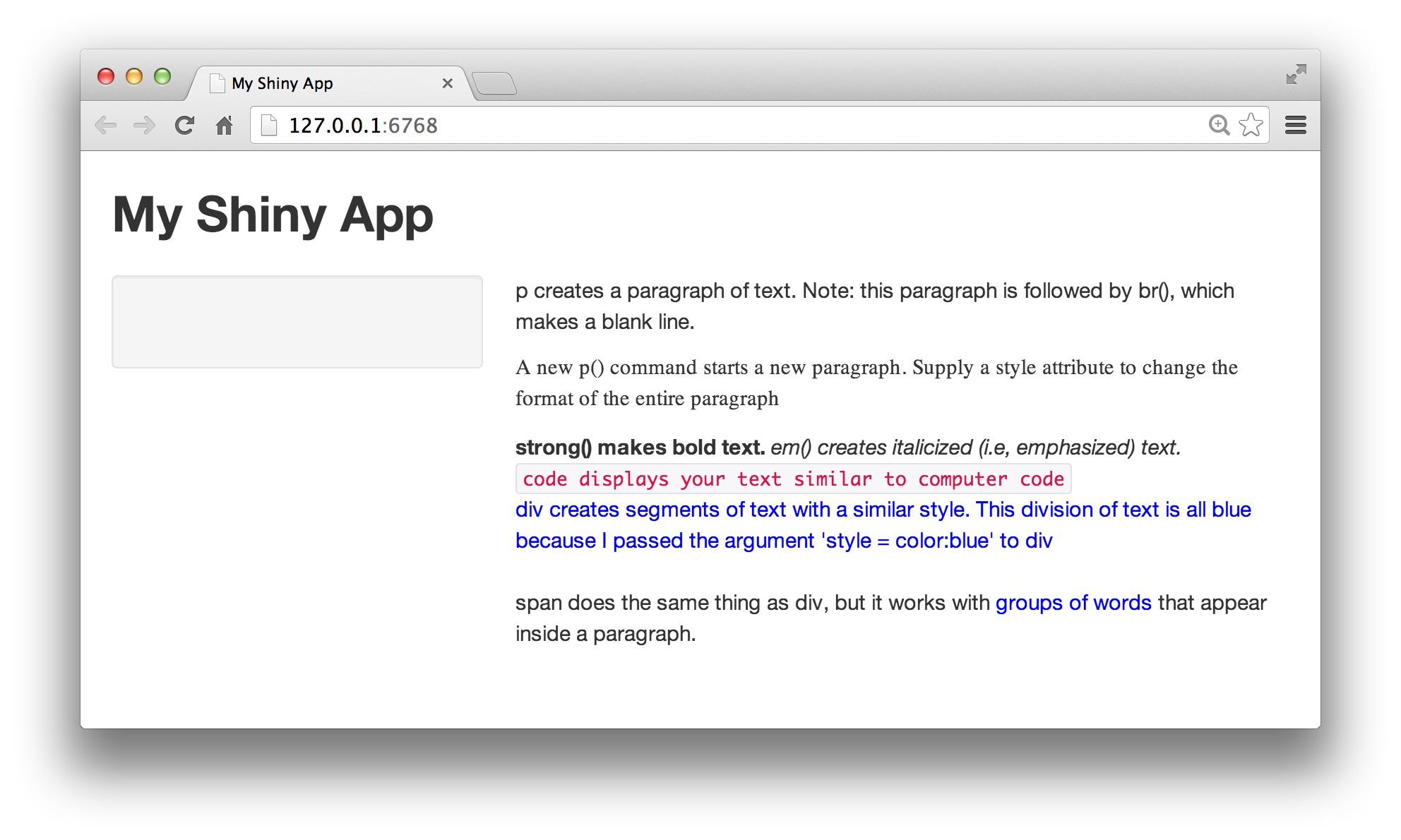
span("groups of words", style = "color:blue"),

"that appear inside a paragraph.")

)

)

))



Images

Images can enhance the appearance of your app and help your users understand the content. Shiny looks for the img function to place image files in your app.

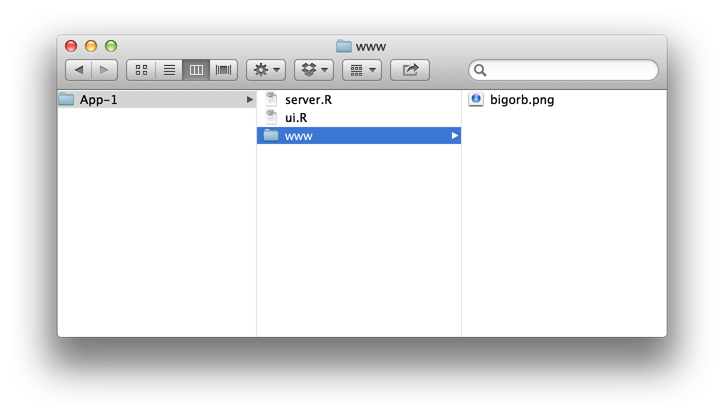
To insert an image, give the img function the name of your image file as the src argument (e.g.,img(src = "my\_image.png")). You must spell out this argument since img passes your input to an HTML tag, and src is what the tag expects.

You can also include other HTML friendly parameters such as height and width. Note that height and width numbers will refer to pixels.

img(src = "my\_image.png", height = 72, width = 72)

The img function looks for your image file in a specific place. Your file *must* be in a folder named www in the same directory as the ui.R script. Shiny treats this directory in a special way. Shiny will share any file placed here with your user’s web browser, which makes www a great place to put images, style sheets, and other things the browser will need to build the wep components of your Shiny app.

So if you want to use an image named [bigorb.png](http://shiny.rstudio.com/tutorial/lesson2/www/bigorb.png), your App-1 directory should look like this one:



With this file arrangment, the ui.R script below can create this app. Download bigorb.png [here](http://shiny.rstudio.com/tutorial/lesson2/www/bigorb.png) and try it out.

*# ui.R*

shinyUI(fluidPage(

titlePanel("My Shiny App"),

sidebarLayout(

sidebarPanel(),

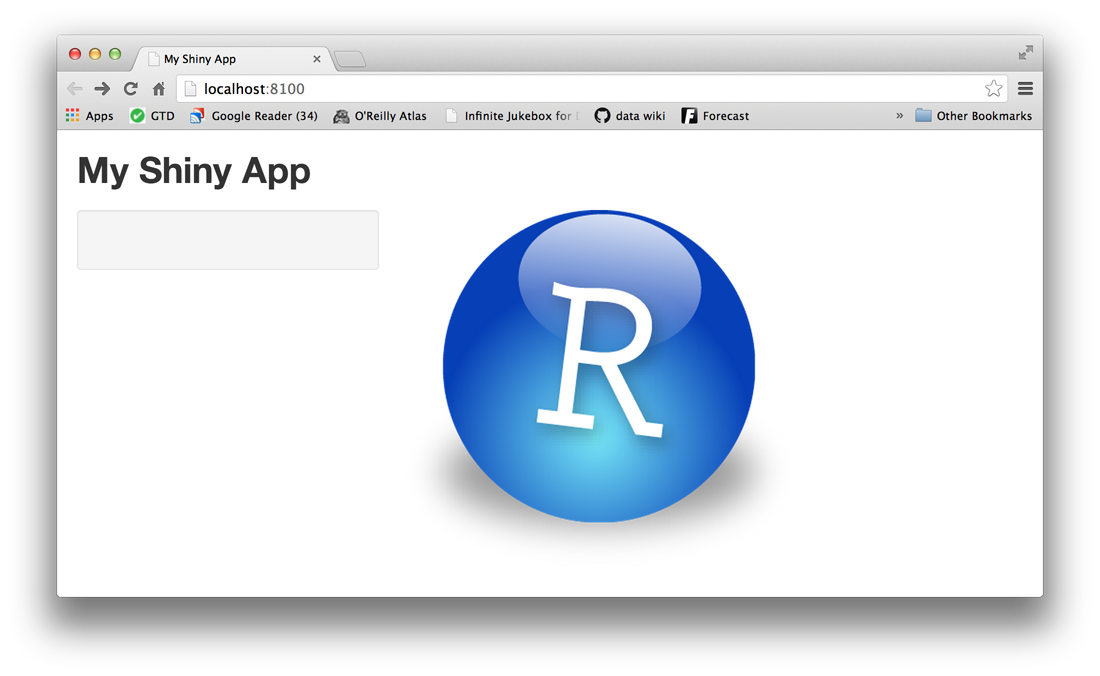
mainPanel(

img(src="bigorb.png", height = 400, width = 400)

)

)

))



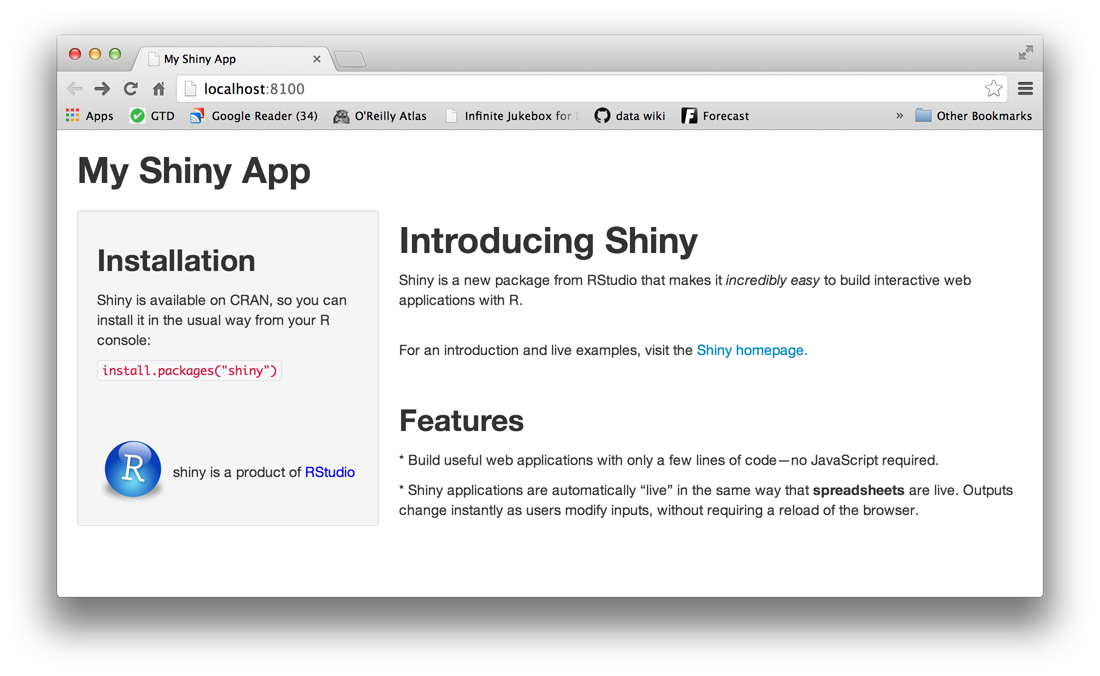
Other tags

This lesson covers the most popular Shiny tag functions, but there are many more tag functions for you to use. You can learn about additional tag functions in [Customize your UI with HTML](http://shiny.rstudio.com/articles/html-tags.html) and the [Shiny HTML Tags Glossary](http://shiny.rstudio.com/articles/tag-glossary.html).

Your turn

You can use Shiny’s layout, HTML, and img functions to create very attractive and useful user-interfaces. See how well you understand these functions by recreating the Shiny app pictured below. Use the examples in this tutorial to work on it and then test it out.

Our ui.R script is found under the Model Answer button, but don’t just copy and paste it. Make sure you understand how the code works before moving on.



Model Answer

*# ui.R*

shinyUI(fluidPage(

titlePanel("My Shiny App"),

sidebarLayout(

sidebarPanel(

h2("Installation"),

p("Shiny is available on CRAN, so you can install it in the usual way from your R console:"),

code('install.packages("shiny")'),

br(),

br(),

br(),

br(),

img(src = "bigorb.png", height = 72, width = 72),

"shiny is a product of ",

span("RStudio", style = "color:blue")

),

mainPanel(

h1("Introducing Shiny"),

p("Shiny is a new package from RStudio that makes it ",

em("incredibly easy"),

" to build interactive web applications with R."),

br(),

p("For an introduction and live examples, visit the ",

a("Shiny homepage.",

href = "http://www.rstudio.com/shiny")),

br(),

h2("Features"),

p("\* Build useful web applications with only a few lines of code—no JavaScript required."),

p("\* Shiny applications are automatically “live” in the same way that ",

strong("spreadsheets"),

" are live. Outputs change instantly as users modify inputs, without requiring a reload of the browser.")

)

)

))

Recap

With your new skills, you can:

* create a user-interface with fluidPage, titlePanel and sidebarLayout
* create an HTML element with one of Shiny’s tag functions
* set HTML tag attributes in the arguments of each tag function
* add an element to your web page by passing it to titlePanel, sidebarPanel or mainPanel
* add multiple elements to each panel by separating them with a comma
* add images by placing your image in a folder labeled www within your Shiny app directory and then calling the img function

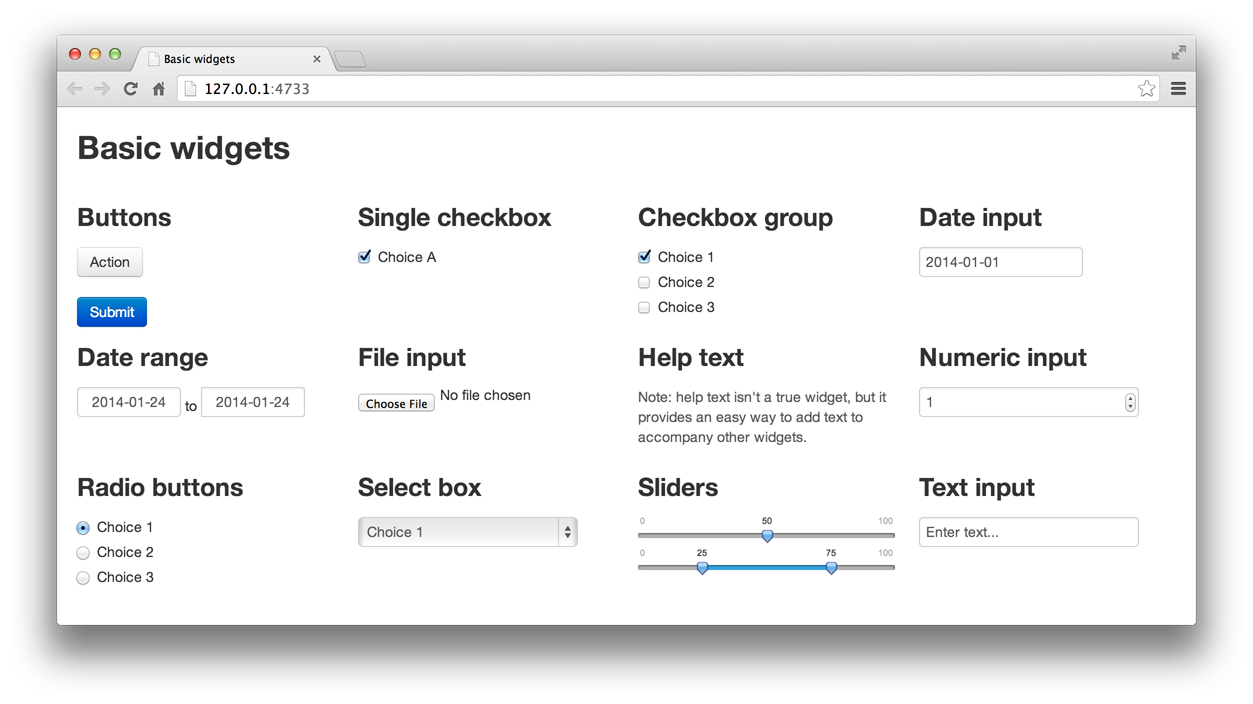
Now that you can place simple content in your user-interface, let’s look at how you would place more complicated content, like widgets. Widgets are interactive web elements that your user can use to control the app. They are also the subject of [Lesson 3](http://shiny.rstudio.com/tutorial/lesson3/).

**LESSON 3**  
Add control widgets

This lesson will show you how to add control widgets to your Shiny apps. What’s a widget? A web element that your users can interact with. Widgets provide a way for your users to send messages to the Shiny app.

Shiny widgets collect a value from your user. When a user changes the widget, the value will change as well. This sets up opportunities that we’ll explore in [Lesson 4](http://shiny.rstudio.com/tutorial/lesson4/).

Control widgets



Shiny comes with a family of pre-built widgets, each created with a transparently named R function. For example, Shiny provides a function named actionButton that creates an Action Button and a function named sliderInput that creates a slider bar.

The standard Shiny widgets are:

| **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 |

Some of these widgets are built using the [Twitter Bootstrap](http://getbootstrap.com/) project, a popular open source framework for building user-interfaces.

Adding widgets

You can add widgets to your web page in the same way that you added other types of HTML content in [Lesson 2](http://shiny.rstudio.com/tutorial/lesson3/.../lesson2/). To add a widget to your app, place a widget function in sidebarPanel or mainPanel in your ui.R file.

Each widget function requires several arguments. The first two arguments for each widget are

* A **Name for the widget.** The user will not see this name, but you can use it to access the widget’s value. The name should be a character string.
* A **label.** This label will appear with the widget in your app. It should be a character string, but it can be an empty string "".

In this example, the name is “action” and the label is “Action”:actionButton("action", label = "Action")

The remaining arguments vary from widget to widget, depending on what the widget needs to do its job. They include things the widget needs to do its job, like initial values, ranges, and increments. You can find the exact arguments needed by a widget on the widget function’s help page, (e.g., ?selectInput).

The ui.R script below makes the app pictured above. Change your own App-1 ui.R script to match it, and then launch the app (runApp("App-1"), select Run App, or use shortcuts).

Play with each widget to get a feel for what it does. Experiment with changing the values of the widget functions and observe the effects. If you are interested in the layout scheme for this Shiny app, read the description in the [application layout guide](http://shiny.rstudio.com/articles/layout-guide.html). This lesson will not cover this slightly more complicated layout scheme, but it is interesting to note what it does.

*# ui.R*

shinyUI(fluidPage(

titlePanel("Basic widgets"),

fluidRow(

column(3,

h3("Buttons"),

actionButton("action", label = "Action"),

br(),

br(),

submitButton("Submit")),

column(3,

h3("Single checkbox"),

checkboxInput("checkbox", label = "Choice A", value = TRUE)),

column(3,

checkboxGroupInput("checkGroup",

label = h3("Checkbox group"),

choices = list("Choice 1" = 1,

"Choice 2" = 2, "Choice 3" = 3),

selected = 1)),

column(3,

dateInput("date",

label = h3("Date input"),

value = "2014-01-01"))

),

fluidRow(

column(3,

dateRangeInput("dates", label = h3("Date range"))),

column(3,

fileInput("file", label = h3("File input"))),

column(3,

h3("Help text"),

helpText("Note: help text isn't a true widget,",

"but it provides an easy way to add text to",

"accompany other widgets.")),

column(3,

numericInput("num",

label = h3("Numeric input"),

value = 1))

),

fluidRow(

column(3,

radioButtons("radio", label = h3("Radio buttons"),

choices = list("Choice 1" = 1, "Choice 2" = 2,

"Choice 3" = 3),selected = 1)),

column(3,

selectInput("select", label = h3("Select box"),

choices = list("Choice 1" = 1, "Choice 2" = 2,

"Choice 3" = 3), selected = 1)),

column(3,

sliderInput("slider1", label = h3("Sliders"),

min = 0, max = 100, value = 50),

sliderInput("slider2", "",

min = 0, max = 100, value = c(25, 75))

),

column(3,

textInput("text", label = h3("Text input"),

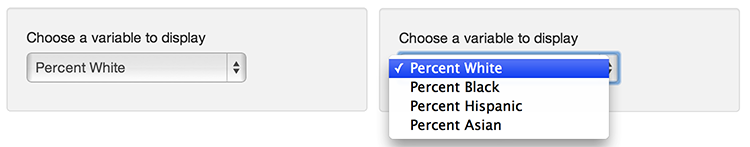
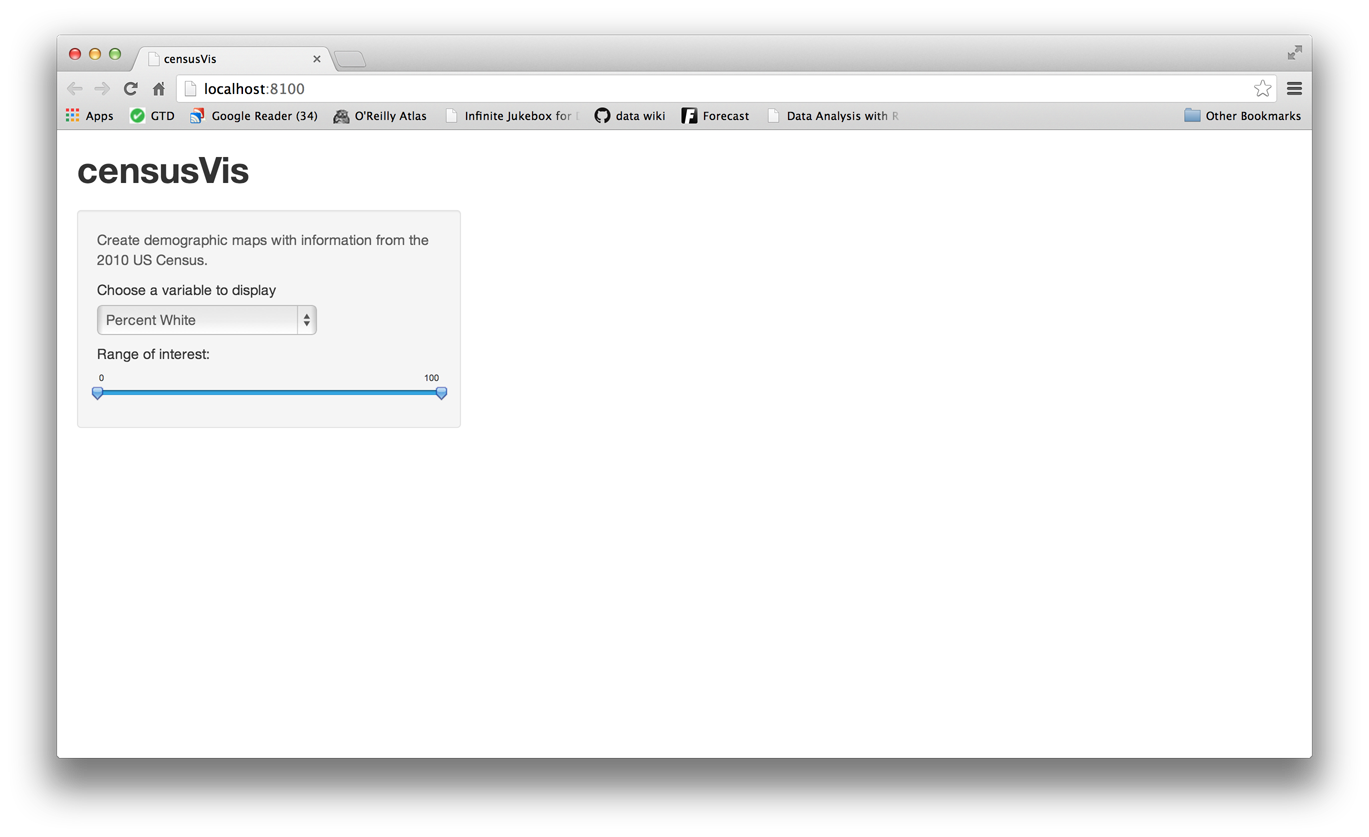
value = "Enter text..."))

)

))

Your turn

Rewrite your ui.R script to create the user-interface displayed below. Notice that this Shiny app uses a basic Shiny layout (no columns) and contains three of the widgets pictured above. The other values of the select box are shown below the image of the app.



Model Answer

Be sure your ui.R script is identical to the one displayed below before you move on. You will use the script in [Lesson 4](http://shiny.rstudio.com/tutorial/lesson4/) and [Lesson 5](http://shiny.rstudio.com/tutorial/lesson5/), as part of an app that visualizes census data.

In particular, make sure that your select box widget is named “var”, and your slider widget is named “range”.

Also notice that the slider widget has *two* values, not one.

*# ui.R*

shinyUI(fluidPage(

titlePanel("censusVis"),

sidebarLayout(

sidebarPanel(

helpText("Create demographic maps with

information from the 2010 US Census."),

selectInput("var",

label = "Choose a variable to display",

choices = list("Percent White", "Percent Black",

"Percent Hispanic", "Percent Asian"),

selected = "Percent White"),

sliderInput("range",

label = "Range of interest:",

min = 0, max = 100, value = c(0, 100))

),

mainPanel()

)

))

Recap

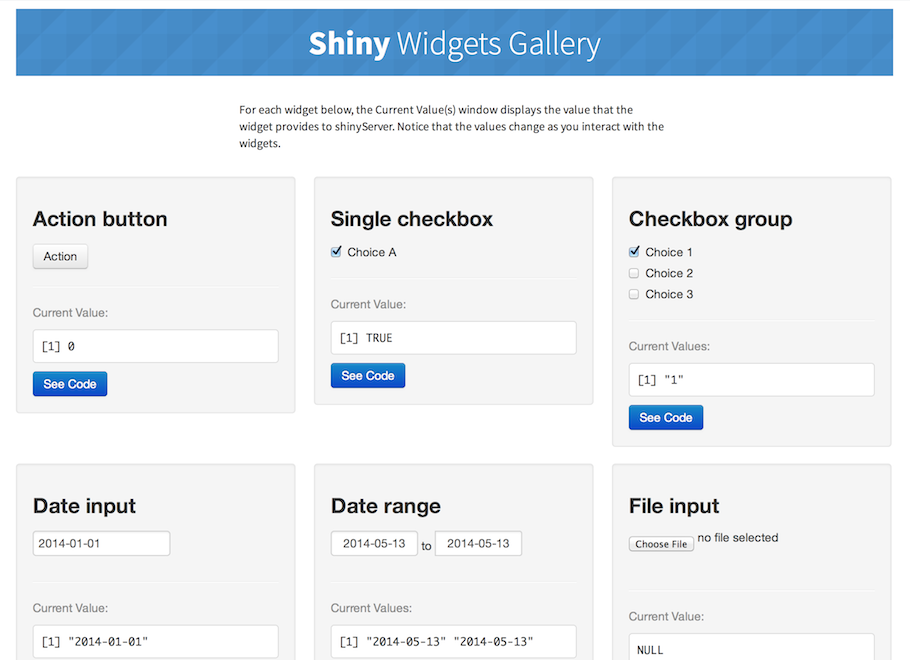
It is easy to add fully functional widgets to your Shiny app.

* Shiny provides a family of functions to create these widgets.
* Each function requires a name and a label.
* Some widgets need specific instructions to do their jobs.
* You add widgets to your Shiny app just like you added other types of HTML content (see [Lesson 2](http://shiny.rstudio.com/tutorial/lesson2/))

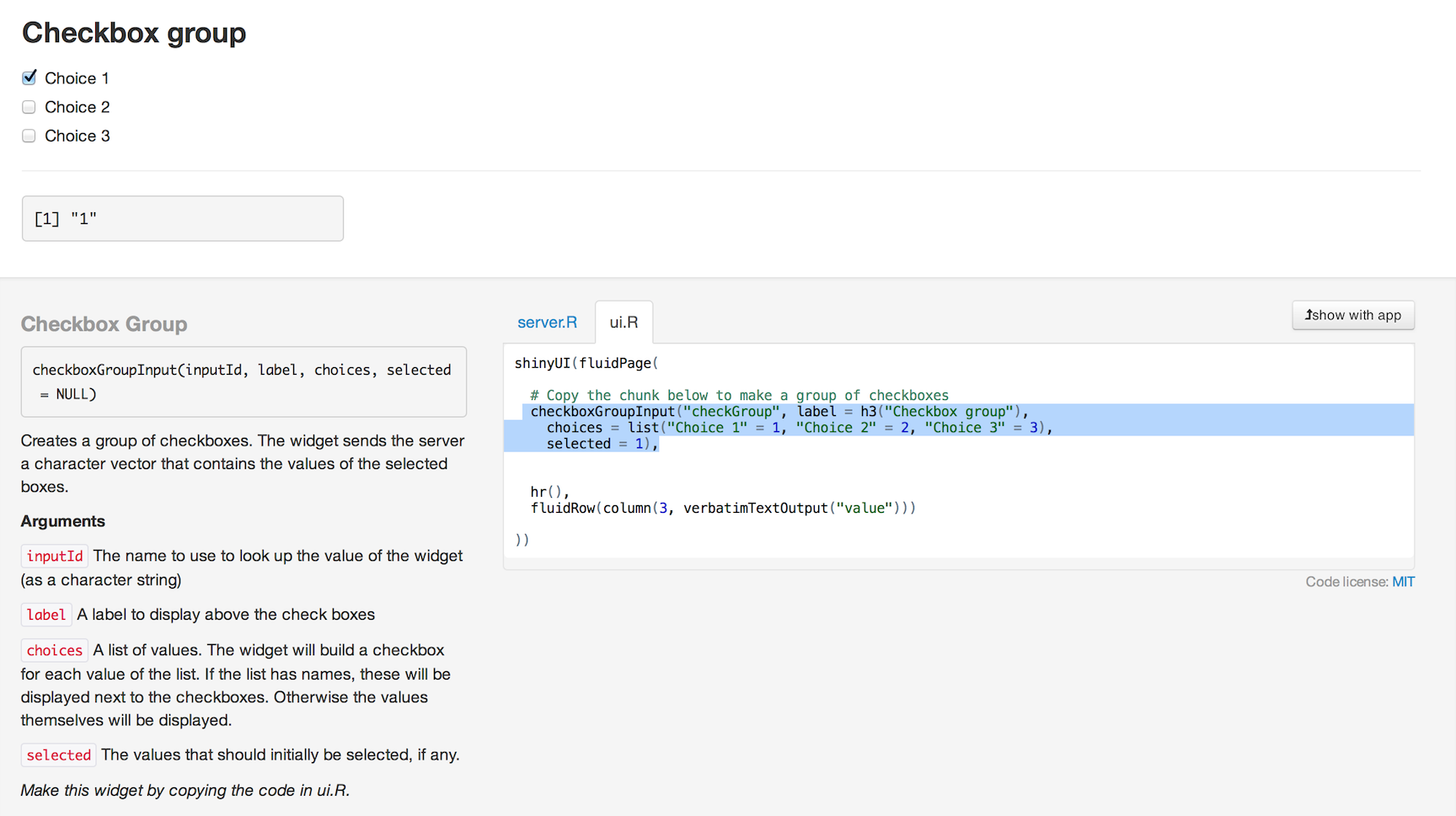
Go Further

The [Shiny Widgets Gallery](http://shiny.rstudio.com/gallery/widget-gallery.html) provides templates that you can use to quickly add widgets to your Shiny apps.

To use a template, visit the [gallery](http://shiny.rstudio.com/gallery/widget-gallery.html). The gallery displays each of Shiny’s widgets, and demonstrates how the widgets’ values change in response to your input.



Select the widget that you want and click the “See Code” button below the widget. The gallery will take you to an example app that describes the widget. To use the widget, copy and paste the code in the example’sui.R file to your ui.R file.



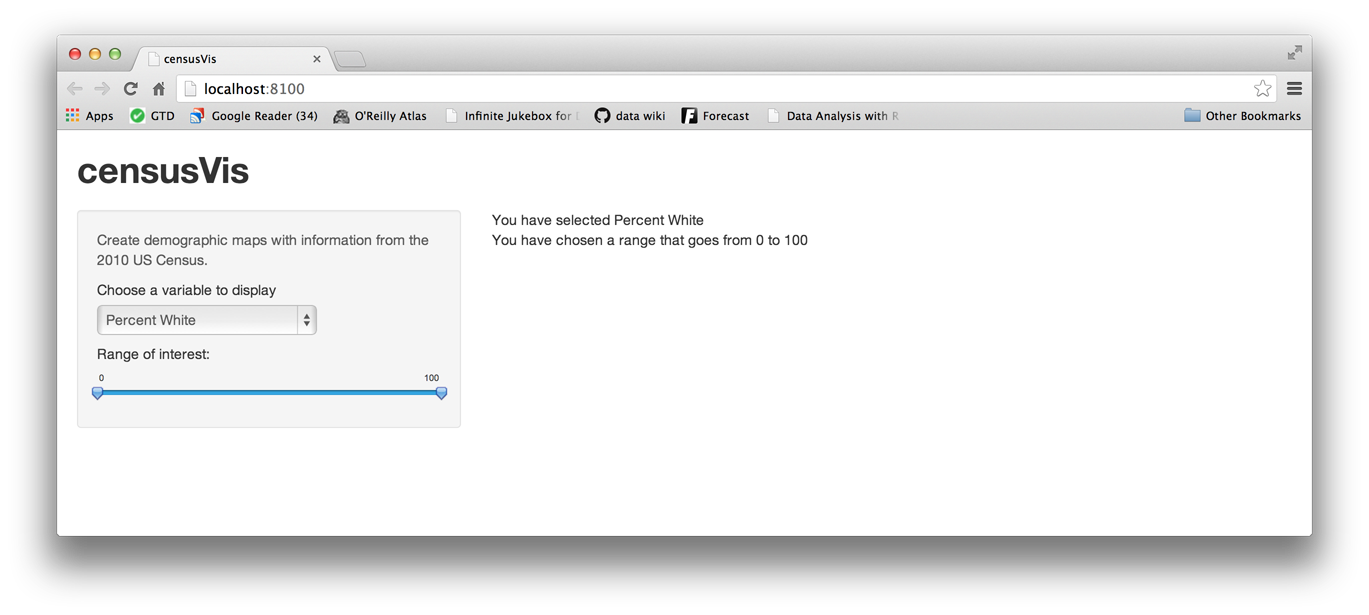
In [Lesson 4](http://shiny.rstudio.com/tutorial/lesson4/), you will learn how to connect widgets to reactive output, objects that update themselves whenever your user changes a widget.

[Continue to lesson 4](http://shiny.rstudio.com/tutorial/lesson4)

**LESSON 4**  
Display reactive output

Time to give your Shiny app a “live” quality! This lesson will teach you how to build reactive output to display in your Shiny app. Reactive output automatically responds when your user toggles a widget.

By the end of this lesson, you’ll know how to make a simple Shiny app with two reactive lines of text. Each line will display the values of a widget based on your user’s input.



This new Shiny app will need its own, new directory. Create a folder in your working directory namedcensus-app. This is where we’ll save the ui.R and server.R files that you make in this lesson.

Two steps

You can create reactive output with a two step process.

1. Add an R object to your user-interface with ui.R.
2. Tell Shiny how to build the object in server.R. The object will be reactive if the code that builds it calls a widget value.

Step 1: Add an R object to the UI

Shiny provides a family of functions that turn R objects into output for your user-interface. Each function creates a specific type of output.

| **Output function** | **creates** |
| --- | --- |
| htmlOutput | raw HTML |
| imageOutput | image |
| plotOutput | plot |
| tableOutput | table |
| textOutput | text |
| uiOutput | raw HTML |
| verbatimTextOutput | text |

You can add output to the user-interface in the same way that you added HTML elements and widgets. Place the output function inside sidebarPanel or mainPanel in the ui.R script.

For example, the ui.R file below uses textOutput to add a reactive line of text to the main panel of the Shiny app pictured above.

*# ui.R*

shinyUI(fluidPage(

titlePanel("censusVis"),

sidebarLayout(

sidebarPanel(

helpText("Create demographic maps with

information from the 2010 US Census."),

selectInput("var",

label = "Choose a variable to display",

choices = c("Percent White", "Percent Black",

"Percent Hispanic", "Percent Asian"),

selected = "Percent White"),

sliderInput("range",

label = "Range of interest:",

min = 0, max = 100, value = c(0, 100))

),

mainPanel(

textOutput("text1")

)

)

))

Notice that textOutput takes an argument, the character string “text1”. Each of the \*Output functions require a single argument: a character string that Shiny will use as the name of your reactive element. Your users will not see this name, but you will use it later.

Step 2: Provide R code to build the object.

Placing a function in ui.R tells Shiny where to display your object. Next, you need to tell Shiny how to build the object.

Do this by providing R code that builds the object in server.R. The code should go in the unnamed function that appears inside shinyServer in your server.R script.

The unnamed function plays a special role in the Shiny process; it builds a list-like object named outputthat contains all of the code needed to update the R objects in your app. Each R object needs to have its own entry in the list.

You can create an entry by defining a new element for output within the unnamed function, like below. The element name should match the name of the reactive element that you created in ui.R.

In the script below, output$text1 matches textOutput("text1") in your ui.R script.

*# server.R*

shinyServer(function(input, output) {

output$text1 <- renderText({

"You have selected this"

})

}

)

You do not need to arrange for the unnamed function to return output in its last line of code. R will automatically update output through reference class semantics.

Each entry to output should contain the output of one of Shiny’s render\* functions. These functions capture an R expression and do some light pre-processing on the expression. Use the render\* function that corrresponds to the type of reactive object you are making.

| **render function** | **creates** |
| --- | --- |
| renderImage | images (saved as a link to a source file) |
| renderPlot | plots |
| renderPrint | any printed output |
| renderTable | data frame, matrix, other table like structures |
| renderText | character strings |
| renderUI | a Shiny tag object or HTML |

Each render\* function takes a single argument: an R expression surrounded by braces, {}. The expression can be one simple line of text, or it can involve many lines of code, as if it were a complicated function call.

Think of this R expression as a set of instructions that you give Shiny to store for later. Shiny will run the instructions when you first launch your app, and then Shiny will re-run the instructions every time it needs to update your object.

For this to work, your expression should return the object you have in mind (a piece of text, a plot, a data frame, etc). You will get an error if the expression does not return an object, or if it returns the wrong type of object.

Use widget values

If you run the server.R script above, the Shiny app will display “You have selected this” in the main panel. However, the text will not be reactive. It will not change even if you manipulate the widgets of your app.

You can make the text reactive by asking Shiny to call a widget value when it builds the text. Let’s look at how to do this.

Take a look at the first line of code in server.R. Do you notice that the unnamed function mentions *two*arguments, input and output? You already saw that output is a list-like object that stores instructions for building the R objects in your app.

input is a second list-like object. It stores the current values of all of the widgets in your app. These values will be saved under the names that you gave the widgets in ui.R.

So for example, our app has two widgets, one named “var” and one named “range” (you gave the widgets these names in [Lesson 3](http://shiny.rstudio.com/tutorial/lesson3/)). The values of “var” and “range” will be saved in input as input$var andinput$range. Since the slider widget has two values (a min and a max), input$range will contain a vector of length two.

Shiny will automatically make an object reactive if the object uses an input value. For example, the server.R file below creates a reactive line of text by calling the value of the select box widget to build the text.

*# server.R*

shinyServer(

function(input, output) {

output$text1 <- renderText({

paste("You have selected", input$var)

})

}

)

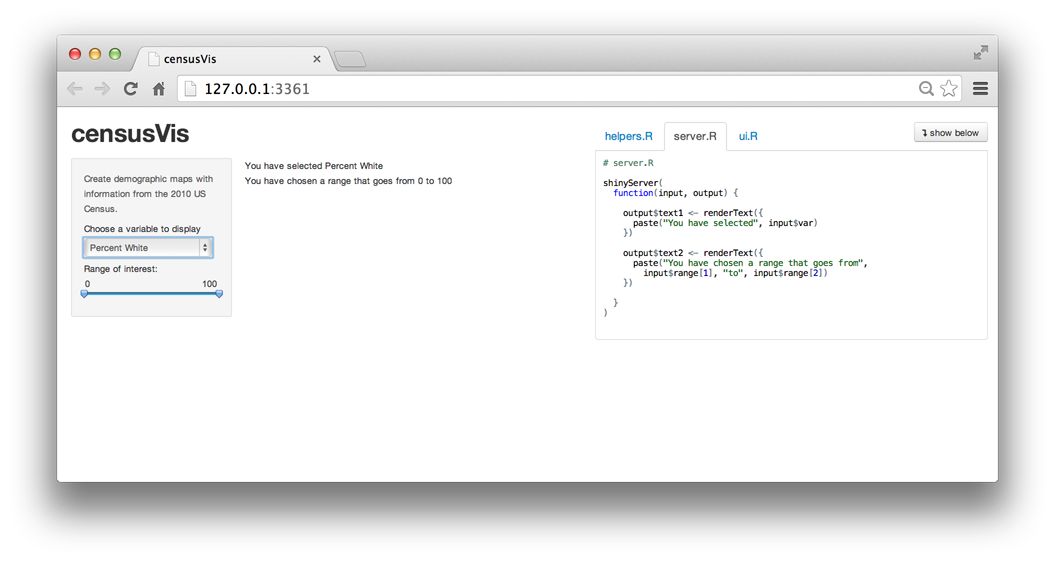
Shiny tracks which outputs depend on which widgets. When a user changes a widget, Shiny will rebuild all of the outputs that depend on the widget, using the new value of the widget as it goes. As a result, the rebuilt objects will be completely up-to-date.

This is how you create reactivity with Shiny, by connecting the values of input to the objects in output. Shiny takes care of all of the other details.

Launch your app and see the reactive output

When you are ready, update your server.R and ui.R files to match those above. Then launch your Shiny app by running runApp("census-app", display.mode = "showcase") at the command line. Your app should look like the app below, and your statement should update instantly as you change the select box widget.

Watch the server.R script. When Shiny rebuilds an output, it highlights the code it is running. This temporary highlighting can help you see how Shiny generates reactive output.



Your turn

Add a second line of reactive text to the main panel of your Shiny app. This line should display “You have chosen a range that goes from *something* to *something*”, and each *something* should show the current minimum (min) or maximum (max) value of the slider widget.

Don’t forget to update both your ui.R and server.R files.

Model answer

*# ui.R*

shinyUI(fluidPage(

titlePanel("censusVis"),

sidebarLayout(

sidebarPanel(

helpText("Create demographic maps with

information from the 2010 US Census."),

selectInput("var",

label = "Choose a variable to display",

choices = c("Percent White", "Percent Black",

"Percent Hispanic", "Percent Asian"),

selected = "Percent White"),

sliderInput("range",

label = "Range of interest:",

min = 0, max = 100, value = c(0, 100))

),

mainPanel(

textOutput("text1"),

textOutput("text2")

)

)

))

And

*# server.R*

shinyServer(

function(input, output) {

output$text1 <- renderText({

paste("You have selected", input$var)

})

output$text2 <- renderText({

paste("You have chosen a range that goes from",

input$range[1], "to", input$range[2])

})

}

)

Discussion

Add the second line of text in the same way that you added the first one. Use textOutput in ui.R to place the second line of text in the main panel. Use renderText in server.R to tell Shiny how to build the text. You’ll need to use the same name to refer to the text in both scripts (e.g., “text2”).

Your text should use both the slider’s min value (saved as input$range[1]) and its max value (saved asinput$range[2]).

Remember that your text will be reactive as long as you connect input values to output objects. Shiny creates reactivity automatically when it recognizes these connections.

Recap

In this lesson, you created your first reactive Shiny app. Along the way, you learned to

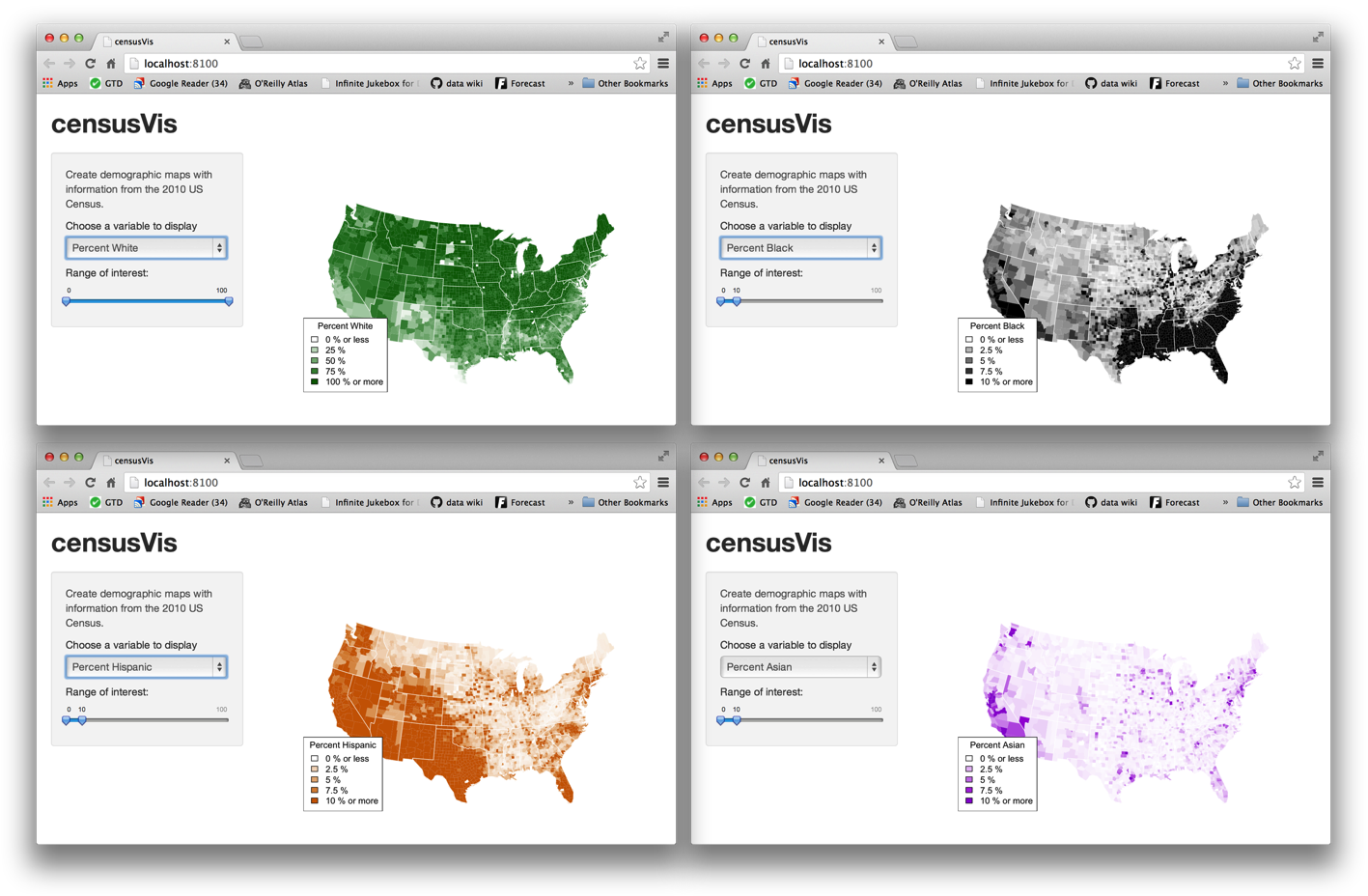
* use an \*Output function in the ui.R script to place reactive objects in your Shiny app
* use a render\* function in the server.R script to tell Shiny how to build your objects
* surround R expressions by braces, {}, in each render\* function
* save your render\* expressions in the output list, with one entry for each reactive object in your app.
* create reactivity by including an input value in a render\* expression

If you follow these rules, Shiny will automatically make your objects reactive.

In [Lesson 5](http://shiny.rstudio.com/tutorial/lesson5/) you will create a more sophisticated reactive app that relies on R scripts and external data.

**LESSON 5**  
Use R scripts and data

This lesson will show you how to load data, R Scripts, and packages to use in your Shiny apps. Along the way, you will build a sophisticated app that visualizes US Census data.



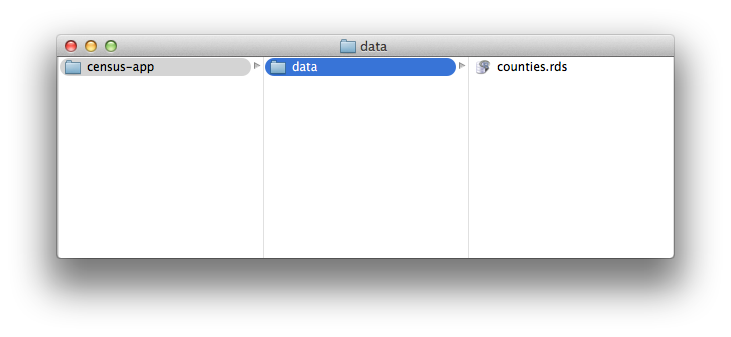
counties.rds

counties.rds is a dataset of demographic data for each county in the United States, collected with theUScensus2010 R package. You can download it [here](http://shiny.rstudio.com/tutorial/lesson5/census-app/data/counties.rds).

Once you have the file,

* Create a new folder named data in your census-app directory.
* Move counties.rds into the data folder.

When you’re done, your census-app folder should look like this.



The dataset in counties.rds contains

* the name of each county in the United States
* the total population of the county
* the percent of residents in the county who are white, black, hispanic, or asian

counties <- readRDS("census-app/data/counties.rds")

head(counties)

name total.pop white black hispanic asian

1 alabama,autauga 54571 77.2 19.3 2.4 0.9

2 alabama,baldwin 182265 83.5 10.9 4.4 0.7

3 alabama,barbour 27457 46.8 47.8 5.1 0.4

4 alabama,bibb 22915 75.0 22.9 1.8 0.1

5 alabama,blount 57322 88.9 2.5 8.1 0.2

6 alabama,bullock 10914 21.9 71.0 7.1 0.2

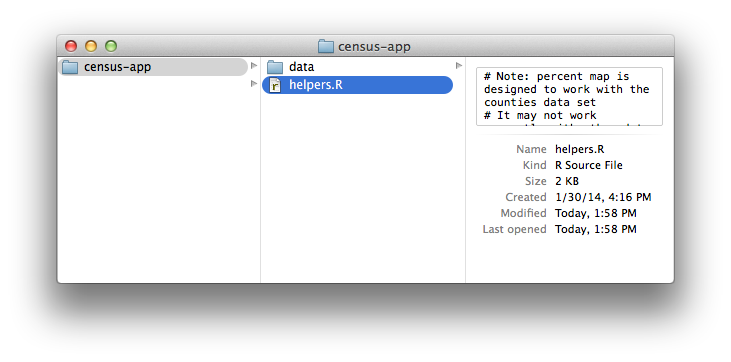
helpers.R

helpers.R is an R script that can help you make [choropleth maps](http://en.wikipedia.org/wiki/Choropleth_map), like the ones pictured above. A choropleth map is a map that uses color to display the regional variation of a variable. In our case,helpers.R will create percent\_map, a function designed to map the data in counties.rds. You can download helpers.R [here](http://shiny.rstudio.com/tutorial/lesson5/census-app/helpers.R).

helpers.R uses the maps and mapproj packages in R. If you’ve never installed these packages before, you’ll need to do so before you make this app. Run

**>** install.packages(c("maps", "mapproj"))

Save helpers.R inside your census-app directory, like below.



The percent\_map function in helpers.R takes five arguments:

| **Argument** | **Input** |
| --- | --- |
| var | a column vector from the counties.rds dataset |
| color | any character string you see in the output of colors() |
| legend.title | A character string to use as the title of the plot’s legend |
| max | A parameter for controlling shade range (defaults to 100) |
| min | A parameter for controlling shade range (defaults to 0) |

You can use percent\_map at the command line to plot the counties data as a choropleth map, like this.

library(maps)

library(mapproj)

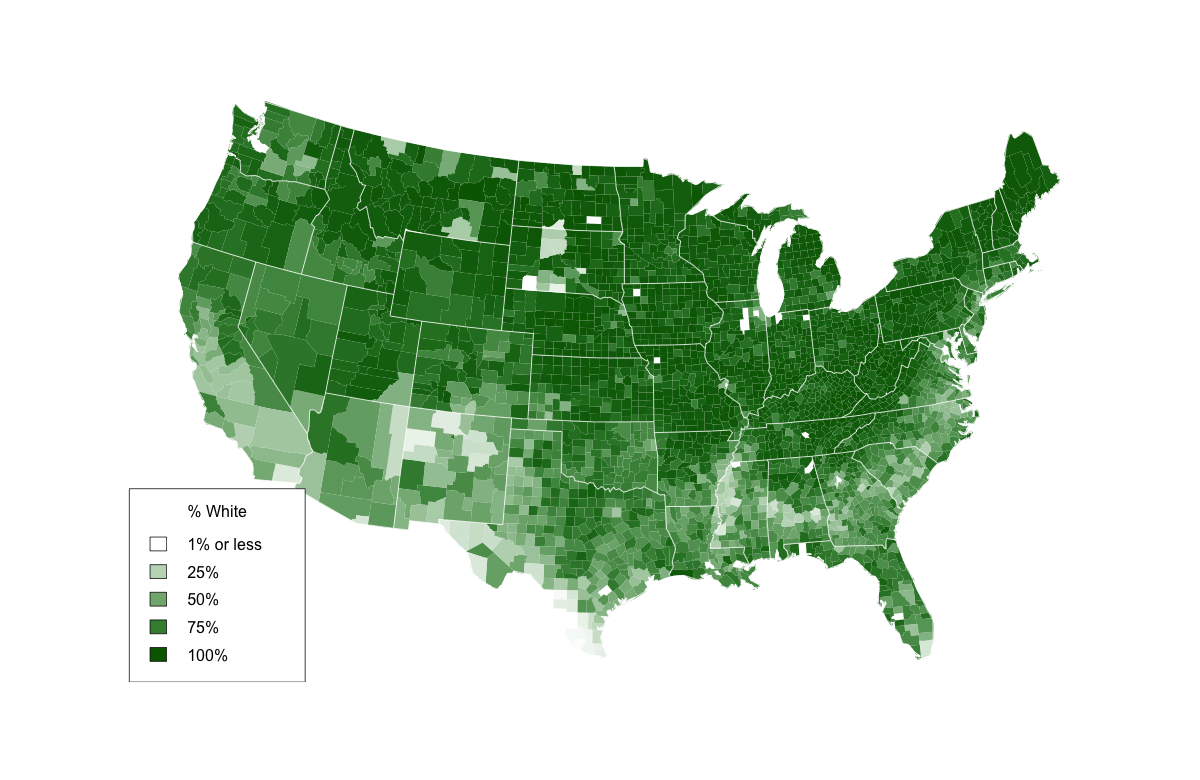
source("census-app/helpers.R")

counties <- readRDS("census-app/data/counties.rds")

percent\_map(counties$white, "darkgreen", "% White")

*Note: The code above assumes that census-app is a sub-directory in your working directory. Make certain to set your working directory as the parent directory for census-app. To change your working directory location, click on Session > Set Working Directory > Choose Directory… in the RStudio menu bar.*

percent\_map plots the counties data as a choropleth map. Here it will plot the percent of white residents in the counties in the color dark green.



Loading files and file paths

Take a look at the above code. To use percent\_map, we first ran helpers.R with the source function, and then loaded counties.rds with the readRDS function. We also ran library(maps) andlibrary(mapproj).

You will need to ask Shiny to call the same functions before it uses percent\_map in your app, but how you write these functions will change. Both source and readRDS require a file path, and file paths do not behave the same way in a Shiny app as they do at the command line.

When Shiny runs the commands in server.R, it will treat all file paths as if they begin in the same directory as server.R. In other words, the directory that you save server.R in will become the working directory of your Shiny app.

Since you saved helpers.R in the same directory as server.R, you can ask Shiny to load it with

source("helpers.R")

Since you saved counties.rds in a sub-directory (named data) of the directory that server.R is in, you can load it with.

counties <- readRDS("data/counties.rds")

You can load the maps and mapproj packages in the normal way with

library(maps)

library(mapproj)

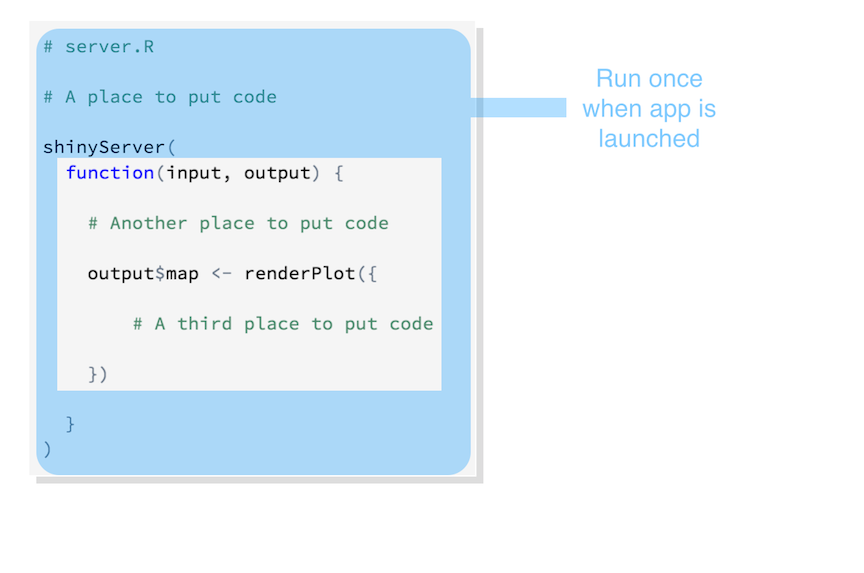
which does not require a file path.

Execution

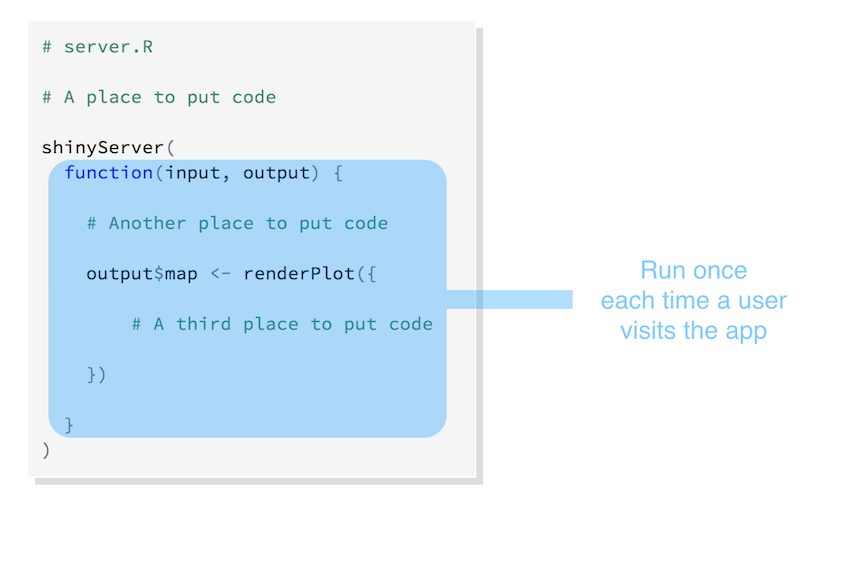
Shiny will execute all of these commands if you place them in your server.R script. However, where you place them in server.R will determine how many times they are run (or re-run), which will in turn affect the performance of your app.

Shiny will run some sections of server.R more often than others.

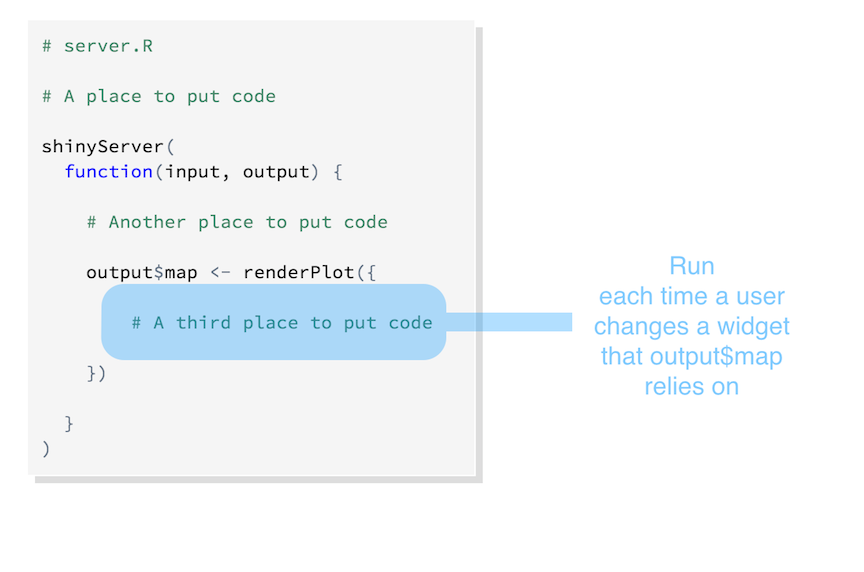
Shiny will run the whole script the first time you call runApp. This causes Shiny to execute shinyServer.shinyServer then gives Shiny the unnamed function in its first argument.



Shiny saves the unnamed function until a new user arrives. Each time a new user visits your app, Shiny runs the unnamed function again, one time. The function helps Shiny build a distinct set of reactive objects for each user.



As users change widgets, Shiny will re-run the R expressions assigned to each reactive object. If your user is very active, these expressions may be re-run many, many times a second.



Here’s what we’ve learned so far:

* The server.R script is run once, when you launch your app
* The unnamed function inside shinyServer is run once *each time* a user visits your app
* The R expressions inside render\* functions are run many times. Shiny runs them once each time a user changes a widget.

How can you use this information?

Source scripts, load libraries, and read data sets at the beginning of server.R *outside* of theshinyServer function. Shiny will only run this code once, which is all you need to set your server up to run the R expressions contained in shinyServer.

Define user specific objects inside shinyServer’s unnamed function, but outside of any render\* calls. These would be objects that you think each user will need their own personal copy of. For example, an object that records the user’s [session information](http://shiny.rstudio.com/articles/client-data.html). This code will be run once per user.

Only place code that Shiny *must* rerun to build an object inside of a render\* function. Shiny will rerun *all* of the code in a render\* chunk each time a user changes a widget mentioned in the chunk. This can be quite often.

You should generally avoid placing code inside a render function that does not need to be there. The code will slow down the entire app.

Your Turn 1

Copy and paste the following ui.R and server.R files to your census-app directory. Then add

source("helpers.R")

counties <- readRDS("data/counties.rds")

library(maps)

library(mapproj)

to server.R. Be sure to place the commands in an efficient location.

*Note: This is the first of two steps that will complete your app. Choose the best place to insert the code above, but do not try to run the app. Your app will return an error until you replace # some arguments with real code in Your Turn 2.*

**ui.R**

*# ui.R*

shinyUI(fluidPage(

titlePanel("censusVis"),

sidebarLayout(

sidebarPanel(

helpText("Create demographic maps with

information from the 2010 US Census."),

selectInput("var",

label = "Choose a variable to display",

choices = c("Percent White", "Percent Black",

"Percent Hispanic", "Percent Asian"),

selected = "Percent White"),

sliderInput("range",

label = "Range of interest:",

min = 0, max = 100, value = c(0, 100))

),

mainPanel(plotOutput("map"))

)

))

**server.R**

*# server.R*

shinyServer(

function(input, output) {

output$map <- renderPlot({

percent\_map( *# some arguments )*

})

}

)

Model Answer 1

Since your app only needs to load helpers.R and counties.rds once, they should go outside of theshinyServer function. This is also a good place to load the maps library (which percent\_map uses).

*# server.R*

library(maps)

library(mapproj)

source("helpers.R")

counties <- readRDS("data/counties.rds")

shinyServer(

function(input, output) {

output$map <- renderPlot({

percent\_map( *# some arguments )*

})

}

)

You may wonder, “Won’t each user need their own copy of counties and percent\_map?” (which would imply that the code should go inside of shinyServer’s unnamed function). No, each user will not.

Keep in mind that your user’s computer won’t run any of the R code in your Shiny app. In fact, their computer won’t even see the R code. The computer that you use as a server will run all of the R code necessary for all of your users. It will send the results over to your users as HTML elements.

Your server can rely on a single global copy of counties.rds and percent\_map to do all of the R execution necessary for all of the users. You only need to build a separate object for each user if the objects will have different values for each of your users.

Finishing the app

The censusVis app has one reactive object, a plot named “map”. The plot is built with the percent\_mapfunction, which takes five arguments.

* The first three arguments, var, color, and legend.title, depend on the value of the select box widget.
* The last two arguments, max and min, should be the max and min values of the slider bar widget.

The server.R script below shows one way to craft reactive arguments for percent\_map. R’s switchfunction can transform the output of a select box widget to whatever you like. However, the script is incomplete. It does not provide values for color, legend.title, max, or min. **Note: the script will not run as is. You will need to finish the script before you run it, which is the task of Your Turn 2.**

*# server.R*

library(maps)

library(mapproj)

counties <- readRDS("data/counties.rds")

source("helpers.R")

shinyServer(

function(input, output) {

output$map <- renderPlot({

data <- switch(input$var,

"Percent White" = counties$white,

"Percent Black" = counties$black,

"Percent Hispanic" = counties$hispanic,

"Percent Asian" = counties$asian)

percent\_map(var = data, color = ?, legend.title = ?, max = ?, min = ?)

})

}

)

Your Turn 2

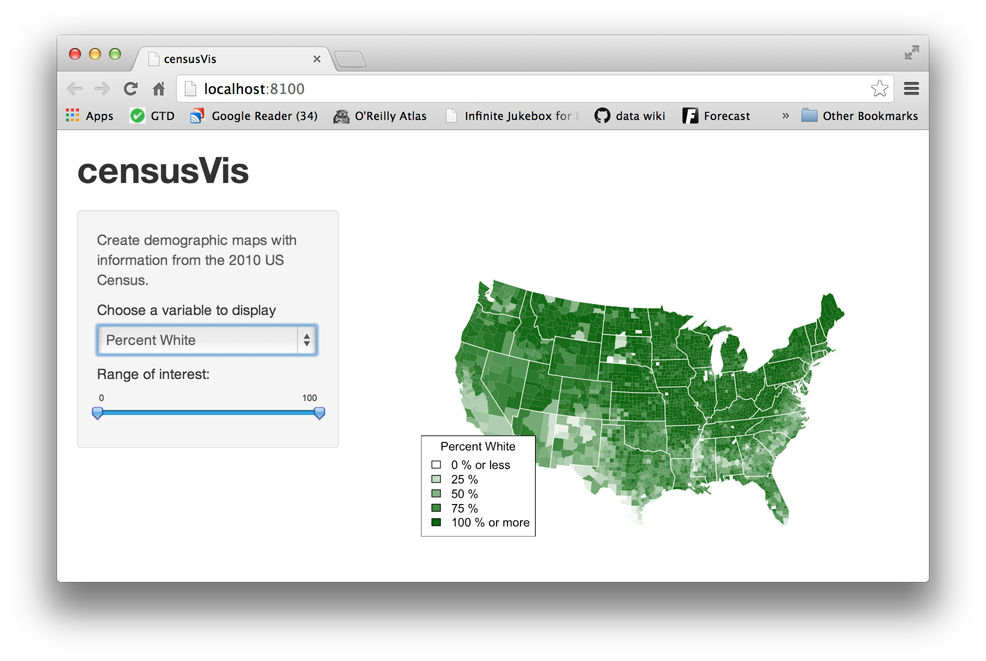
Complete the code to build a working censusVis app.

When you’re ready to deploy your app, save your server.R and ui.R files and runrunApp("census-app"). If everything works, your app should look like the picture below.

You’ll need to decide

* how to create the argument values for percent\_map, and
* where to put the code that creates these arguments.

Remember, you’ll want the argument values to switch whenever a user changes the associated widget. When you are finished, or if you get stuck, read on below for a model answer.



Model Answers 2

A simple version of server.R:

*# server.R*

library(maps)

library(mapproj)

counties <- readRDS("data/counties.rds")

source("helpers.R")

shinyServer(

function(input, output) {

output$map <- renderPlot({

data <- switch(input$var,

"Percent White" = counties$white,

"Percent Black" = counties$black,

"Percent Hispanic" = counties$hispanic,

"Percent Asian" = counties$asian)

color <- switch(input$var,

"Percent White" = "darkgreen",

"Percent Black" = "black",

"Percent Hispanic" = "darkorange",

"Percent Asian" = "darkviolet")

legend <- switch(input$var,

"Percent White" = "% White",

"Percent Black" = "% Black",

"Percent Hispanic" = "% Hispanic",

"Percent Asian" = "% Asian")

percent\_map(var = data,

color = color,

legend.title = legend,

max = input$range[2],

min = input$range[1])

})

}

)

A more concise version of server.R:

*# server.R*

library(maps)

library(mapproj)

counties <- readRDS("data/counties.rds")

source("helpers.R")

shinyServer(

function(input, output) {

output$map <- renderPlot({

args <- switch(input$var,

"Percent White" = list(counties$white, "darkgreen", "% White"),

"Percent Black" = list(counties$black, "black", "% Black"),

"Percent Hispanic" = list(counties$hispanic, "darkorange", "% Hispanic"),

"Percent Asian" = list(counties$asian, "darkviolet", "% Asian"))

args$min <- input$range[1]

args$max <- input$range[2]

do.call(percent\_map, args)

})

}

)

and

*# ui.R*

shinyUI(fluidPage(

titlePanel("censusVis"),

sidebarLayout(

sidebarPanel(

helpText("Create demographic maps with

information from the 2010 US Census."),

selectInput("var",

label = "Choose a variable to display",

choices = c("Percent White", "Percent Black",

"Percent Hispanic", "Percent Asian"),

selected = "Percent White"),

sliderInput("range",

label = "Range of interest:",

min = 0, max = 100, value = c(0, 100))

),

mainPanel(plotOutput("map"))

)

))

Recap

You can create more complicated Shiny apps by loading R Scripts, packages, and data sets.

Keep in mind:

* The directory that server.R appears in will become the working directory of the Shiny app
* Shiny will run code placed at the start of server.R, before shinyServer, only once during the life of the app.
* Shiny will run code placed inside shinyServer multiple times, which can slow the app down.

You also learned that switch is a useful companion to multiple choice Shiny widgets. Use switch to change the values of a widget into R expressions.

As your apps become more complex, they can become inefficient and slow. [Lesson 6](http://shiny.rstudio.com/tutorial/lesson6/) will show you how to build fast, modular apps with reactive expressions.

**LESSON 6**  
Use reactive expressions

Shiny apps wow your users by running fast, instantly fast. But what if your app needs to do a lot of slow computation?

This lesson will show you how to streamline your Shiny apps with reactive expressions. Reactive expressions let you control which parts of your app update when, which prevents unnecessary work.

To get started:

* Create a new folder named stockVis in your working directory.
* Download the following files and place them inside stockVis: [ui.R](http://shiny.rstudio.com/tutorial/lesson6/stockVis/ui.R), [server.R](http://shiny.rstudio.com/tutorial/lesson6/stockVis/server.R), and [helpers.R](http://shiny.rstudio.com/tutorial/lesson6/stockVis/helpers.R).
* Launch the app with runApp("stockVis")

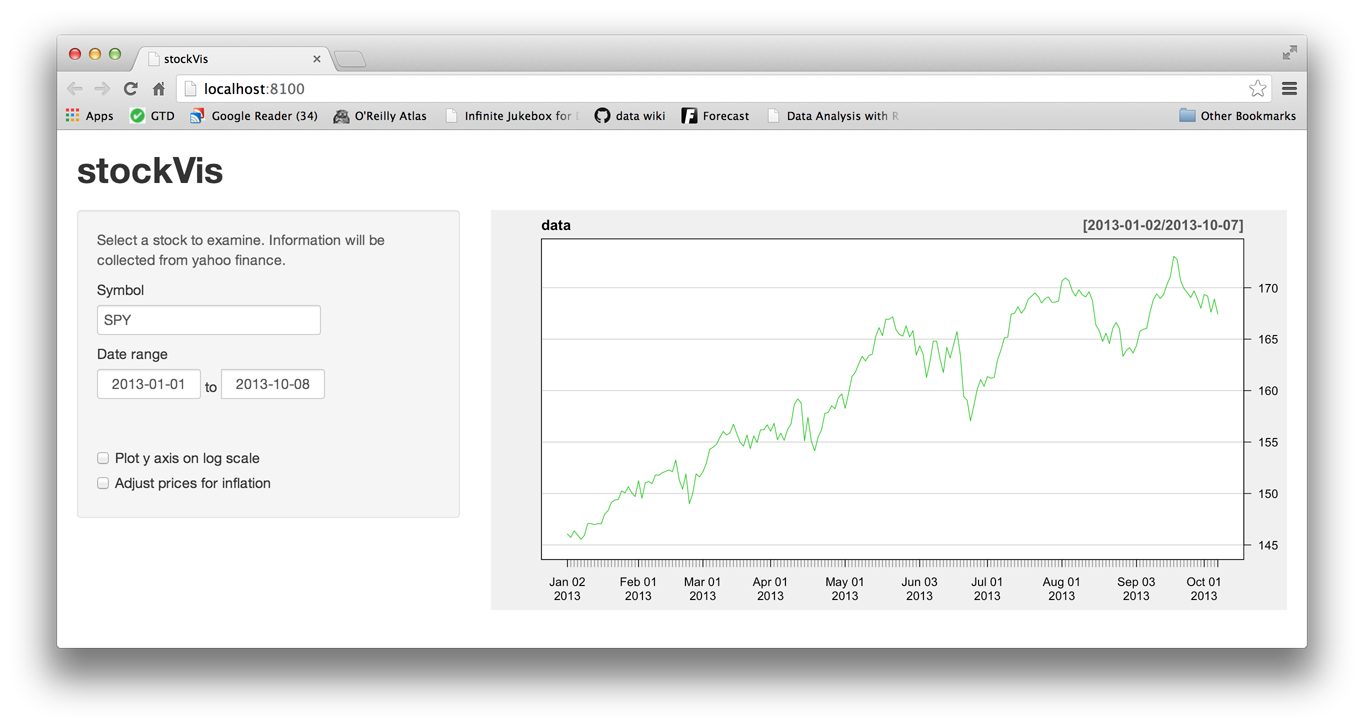
StockVis use R’s quantmod package, so you’ll need to install quantmod withinstall.packages("quantmod") if you do not already have it.

runApp("stockVis")

A new app: stockVis

The stockVis app looks up stock prices by ticker symbol and displays the results as a line chart. The app lets you

1. Select a stock to examine
2. Pick a range of dates to review
3. Choose whether to plot stock prices or the log of the stock prices on the y axis, and
4. Decide whether or not to correct prices for inflation.



Note that the “Adjust prices for inflation” check box doesn’t work yet. One of our tasks in this lesson is to fix this check box.

By default, stockVis displays the SPY ticker (an index of the entire S & P 500). To look up a different stock, type in a stock symbol that Yahoo finance will recognize. You can look up Yahoo’s stock symbols [here](http://finance.yahoo.com/lookup). Some common symbols are GOOG (Google), AAPL (Apple), and GS (Goldman Sachs).

StockVis relies heavily on two functions from the quantmod package:

1. It usesgetSymbols to download financial data straight into R from websites like [Yahoo finance](http://finance.yahoo.com/) and the [Federal Reserve Bank of St. Louis](http://research.stlouisfed.org/fred2/).
2. It uses chartSeries to display prices in an attractive chart.

StockVis also relies on an R script named helpers.R, which contains a function that adjusts stock prices for inflation.

Check boxes and date ranges

The stockVis app uses a few new widgets.

* a date range selector, created with dateRangeInput, and
* a couple of check boxes made with checkboxInput. Check box widgets are very simple. They return a TRUE when the check box is checked, and a FALSE when the check box is not checked.

The check boxes are named log and adjust in the ui.R script, which means you can look them up asinput$log and input$adjust in the server.R script. If you’d like to review how to use widgets and their values, check out [Lesson 3](http://shiny.rstudio.com/tutorial/Lesson-3/) and [Lesson 4](http://shiny.rstudio.com/tutorial/Lesson-4/).

Streamline computation

The stockVis app has a problem.

Examine what will happen when you click “Plot y axis on the log scale.” The value of input$log will change, which will cause the entire expression in renderPlot to re-run:

output$plot <- renderPlot({

data <- getSymbols(input$symb, src = "yahoo",

from = input$dates[1],

to = input$dates[2],

auto.assign = FALSE)

chartSeries(data, theme = chartTheme("white"),

type = "line", log.scale = input$log, TA = NULL)

})

Each time renderPlot re-runs

1. it re-fetches the data from Yahoo finance with getSymbols, and
2. it re-draws the chart with the correct axis.

This is not good, because you do not need to re-fetch the data to re-draw the plot. In fact, Yahoo finance will cut you off if you re-fetch your data too often (because you begin to look like a bot). But more importantly, re-running getSymbols is unnecessary work, which can slow down your app and consume server bandwidth.

Reactive expressions

You can limit what gets re-run during a reaction with reactive expressions.

A reactive expression is an R expression that uses widget input and returns a value. The reactive expression will update this value whenever the original widget changes.

To create a reactive expression use the reactive function, which takes an R expression surrounded by braces (just like the render\* functions).

For example, here’s a reactive expression that uses the widgets of stockVis to fetch data from Yahoo.

dataInput <- reactive({

getSymbols(input$symb, src = "yahoo",

from = input$dates[1],

to = input$dates[2],

auto.assign = FALSE)

})

When you run the expression, it will run getSymbols and return the results, a data frame of price data. You can use the expression to access price data in renderPlot by calling dataInput().

output$plot <- renderPlot({

chartSeries(dataInput(), theme = chartTheme("white"),

type = "line", log.scale = input$log, TA = NULL)

})

Reactive expressions are a bit smarter than regular R functions. They cache their values and know when their values have become outdated. What does this mean? The first time that you run a reactive expression, the expression will save its result in your computer’s memory. The next time you call the reactive expression, it can return this saved result without doing any computation (which will make your app faster).

The reactive expression will only return the saved result if it knows that the result is up-to-date. If the reactive expression has learned that the result is obsolete (because a widget has changed), the expression will recalculate the result. It then returns the new result and saves a new copy. The reactive expression will use this new copy until it too becomes out of date.

Let’s summarize this behavior

* A reactive expression saves its result the first time you run it.
* The next time the reactive expression is called, it checks if the saved value has become out of date (i.e., whether the widgets it depends on have changed).
* If the value is out of date, the reactive object will recalculate it (and then save the new result).
* If the value is up-to-date, the reactive expression will return the saved value without doing any computation.

You can use this behavior to prevent Shiny from re-running unnecessary code. Consider how a reactive expression will work in the new stockVis app below.

*# server.R*

library(quantmod)

source("helpers.R")

shinyServer(function(input, output) {

dataInput <- reactive({

getSymbols(input$symb, src = "yahoo",

from = input$dates[1],

to = input$dates[2],

auto.assign = FALSE)

})

output$plot <- renderPlot({

chartSeries(dataInput(), theme = chartTheme("white"),

type = "line", log.scale = input$log, TA = NULL)

})

})

When you click “Plot y axis on the log scale”, input$log will change and renderPlot will re-execute. Now

1. renderPlot will call dataInput()
2. dataInput will check that the dates and symb widgets have not changed
3. dataInput will return its saved data set of stock prices *without re-fetching data from Yahoo*
4. renderPlot will re-draw the chart with the correct axis.

Dependencies

What if your user changes the stock symbol in the symb widget?

This will make the plot drawn by renderPlot out of date, but renderPlot no longer calls input$symb. Will Shiny know that input$symb has made plot out of date?

Yes, Shiny will know and will redraw the plot. Shiny keeps track of which reactive expressions an outputobject depends on, as well as which widget inputs. Shiny will automatically re-build an object if

* an input value in the objects’s render\* function changes, or
* a reactive expression in the objects’s render\* function becomes obsolete

Think of reactive expressions as links in a chain that connect input values to output objects. The objects in output will respond to changes made anywhere downstream in the chain. (You can fashion a long chain because reactive expressions can call other reactive expressions).

Only call a reactive expression from within a reactive or a render\*function. Why? Only these R functions are equipped to deal with reactive output, which can change without warning. In fact, Shiny will prevent you from calling reactive expressions outside of these functions.

Warm up

Time to fix the broken check box for “Adjust prices for inflation.” Your user should be able to toggle between prices adjusted for inflation and prices that have not been adjusted.

The adjust function in helpers.R uses the [Consumer Price Index](http://research.stlouisfed.org/fred2/series/CPIAUCNS) data provided by the Federal Reserve Bank of St. Louis to transform historical prices into present day values. But how can you implement this in the app?

Here’s one solution below, but it is not ideal. Can you spot why? Once again it has to do with input$log.

*# server.R*

library(quantmod)

source("helpers.R")

shinyServer(function(input, output) {

dataInput <- reactive({

getSymbols(input$symb, src = "yahoo",

from = input$dates[1],

to = input$dates[2],

auto.assign = FALSE)

})

output$plot <- renderPlot({

data <- dataInput()

if (input$adjust) data <- adjust(dataInput())

chartSeries(data, theme = chartTheme("white"),

type = "line", log.scale = input$log, TA = NULL)

})

})

adjust is called *inside* renderPlot. If the adjust box is checked, the app will readjust all of the prices each time you switch from a normal y scale to a logged y scale. This readjustment is unnecessary work.

Your Turn

Fix this problem by adding a new reactive expression to the app. The reactive expression should take the value of dataInput and return an adjusted (or not adjusted) copy of the data.

When you think you have it, compare your solution to the model answer below. Make sure you understand what calculations will happen and what calculations will not happen in your app when your user clicks “Plot y axis on the log scale”.

*# server.R*

library(quantmod)

source("helpers.R")

shinyServer(function(input, output) {

dataInput <- reactive({

getSymbols(input$symb, src = "yahoo",

from = input$dates[1],

to = input$dates[2],

auto.assign = FALSE)

})

finalInput <- reactive({

if (!input$adjust) return(dataInput())

adjust(dataInput())

})

output$plot <- renderPlot({

chartSeries(finalInput(), theme = chartTheme("white"),

type = "line", log.scale = input$log, TA = NULL)

})

})

Now you have isolated each input in its own reactive expression or render\* function. If an input changes, only out of date expressions will re-run.

Here’s an example of the flow:

* A user clicks “Plot y axis on the log scale.”
* renderPlot re-runs.
* renderPlot calls finalInput.
* finalInput checks with dataInput and input$adjust.
* If neither has changed, finalInput returns its saved value.
* If either has changed, finalInput calculates a new value with the current inputs. It will pass the new value to renderPlot and store the new value for future queries.

Recap

You can make your apps faster by modularizing your code with reactive expressions.

* A reactive expression takes input values, or values from other reactive expressions, and returns a new value
* Reactive expressions save their results, and will only re-calculate if their input has changed
* Create reactive expressions with reactive({ })
* Call reactive expressions with the name of the expression followed by parentheses ()
* Only call reactive expressions from within other reactive expressions or render\* functions

You can now create sophisticated, streamlined Shiny apps. The final lesson in this tutorial will show you how to share your apps with others.

[Continue to lesson 7](http://shiny.rstudio.com/tutorial/lesson7)

**LESSON 7**  
Share your apps

You can now build a useful Shiny app, but can you share it with others? This lesson will show you several ways to share your Shiny apps.

When it comes to sharing Shiny apps, you have two basic options:

1. **Share your Shiny app as two files: server.R and ui.R.** This is the simplest way to share an app, but it works only if your users have R on their own computer (and know how to use it). Users can use these scripts to launch the app from their own R session, just like you’ve been launching the apps.
2. **Share your Shiny app as a web page.** This is definitely the most user friendly way to share a Shiny app. Your users can navigate to your app through the internet with a web browser. They will find your app fully rendered, up to date, and ready to go.

Share as two R files

Anyone with R can run your Shiny app. They will need a copy of your server.R and ui.R files, as well as any supplementary materials used in your app (e.g., www folders or helpers.R files).

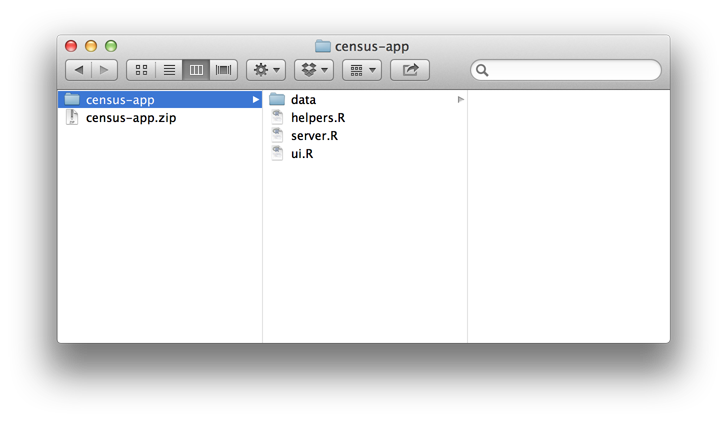
To send your files to another user, email the files (perhaps in a zip file) or host the files online.

Your user can place the files into an app directory in their working directory. They can launch the app in R with the same commands you used on your computer.

*# install.packages("shiny")*

library(shiny)

runApp("census-app")



Shiny has three built in commands that make it easy to use files that are hosted online: runUrl,runGitHub, and runGist.

runUrl

runUrl will download and launch a Shiny app straight from a weblink.

To use runURL:

* Save your Shiny app’s directory as a zip file
* Host that zip file at its own link on a web page. Anyone with access to the link can launch the app from inside R by running:

library(shiny)

runUrl( "<the weblink>")

runGitHub

If you don’t have your own web page to host the files at, you can host your the files for free at[www.github.com](http://www.github.com/).

Github is a popular project hosting site for R developers since it does more than just host files. Github provides many features to support collaboration, such as issue trackers, wikis, and close integration with the[git](http://git-scm.com/) version control system. To use Github, you’ll need to sign up (it’s free) and choose a user name.

To share an app through Github, create a project repository on Github. Then store your server.R andui.R files in the repository, along with any supplementary files that the app uses.

Your users can launch your app by running:

runGitHub( "<your repository name>", "<your user name>")

runGist

If you want an anonymous way to post files online, Github offers a pasteboard service for sharing files at[gist.github.com](http://gist.github.com/). You don’t need to sign up for Github to use this service. Even if you have a Github account, Gist can be a simple, quick way to share Shiny projects.

To share your app as a Gist:

* Copy and paste your server.R and ui.R files to the Gist web page.
* Note the URL that Github gives the Gist.

Once you’ve made a Gist, your users can launch the app with runGist("<gist number>") where"<gist number>" is the number that appears at the end of your Gist’s web address.

[Here](https://gist.github.com/jcheng5/3239667)’s an example of an app hosted as a Gist. You could launch this app with:

runGist("3239667")

Share as a web page

All of the above methods share the same limitation. They require your user to have R and Shiny installed on their computer.

However, Shiny creates the perfect opportunity to share output with people who do *not* have R (and have no intention of getting it). Your Shiny app happens to be one of the most widely used communication tools in the world: a web page. If you host the app at its own URL, users can visit the app (and not need to worry about code).

If you are familiar with web hosting or have access to an IT department, you can host your Shiny apps yourself.

If you’d prefer an easier experience or need support, RStudio offers three ways to host your Shiny app as a web page:

1. Shinyapps.io.
2. Shiny Server, and
3. Shiny Server Pro

Shinyapps.io

The easiest way to turn your Shiny app into a web page is to use [shinyapps.io](http://my.shinyapps.io/), RStudio’s hosting service for Shiny apps.

shinyapps.io lets you upload your app straight from your R session to a server hosted by RStudio. You have complete control over your app including server administration tools. You can find out more about shinyapps.io by visiting [shinyapps.io](http://shinyapps.io/).

Shiny Server

Shiny Server is a companion program to Shiny that builds a web server designed to host Shiny apps. It’s free, open source, and available from Github.

Shiny Server is a server program that Linux servers can run to host a Shiny app as a web page. To use Shiny Server, you’ll need a Linux server that has explicit support for Ubuntu 12.04 or greater (64 bit) and CentOS/RHEL 5 (64 bit). If you are not using an explicitly supported distribution, you can still use Shiny Server by building it from source.

You can host multiple Shiny applications on multiple web pages with the same Shiny Server, and you can deploy the apps from behind a firewall.

To see detailed instructions for installing and configuring a Shiny Server, visit the Shiny Server [guide](https://github.com/rstudio/shiny-server/blob/master/README.md).

Shiny Server Pro

Shiny Server will get your app to the web and take care of all of your Shiny publishing needs. However, if you use Shiny in a for-profit setting, you may want to give yourself the server tools that come with most paid server programs, such as

* Password authentification
* SSL support
* Administrator tools
* Priority support
* and more.

If so, check out [Shiny Server Pro](http://www.rstudio.com/shiny/server/), RStudio’s paid professional version of Shiny Server.

Recap

Shiny apps are easy to share. You can share your app as a couple of R scripts, or as a fully functioning web app with its own URL. Each method has its own advantages.

You learned:

* Anyone can launch your app as long as they have a copy of R, Shiny, and a copy of your app’s files.
* runUrl, runGitHub, and runGist make it simple to share and retrieve Shiny files from web links.
* You can turn your app into a live web app at its own URL with [shinyapps.io](http://shinyapps.io/).
* You can use the open source Shiny Server to build a Linux server that hosts Shiny apps.
* If you need closer control, or want to manage large volumes of traffic, you can purchase [Shiny Server Pro](http://www.rstudio.com/shiny/server/) from RStudio.

Congratulations. You’ve worked through the entire Shiny development process. You can build a sophisticated, reactive app, deploy it, and share it with others. Users can interact with your data and follow your stories in a new way.

The next step is to practice, and then explore the advanced features of Shiny.

The [Shiny Dev Center](http://shiny.rstudio.com/) can help you along the way. It hosts a [gallery](http://shiny.rstudio.com/gallery/) of inspiring apps, along with the code that makes the apps.

The Shiny Dev Center also includes an [articles](http://shiny.rstudio.com/articles/) section for continuing education. Each article examines an intermediate to advanced Shiny topic in depth.

You now know enough to build your own Shiny apps. See what you can do!