CREATING DYNAMIC WEB APPS WITH R SHINY

Business Administration, Analytics and Information Systems

LECTURE 1

Introduction | Your first Shiny app

Outline

- What is R Shiny?
- Features of R Shiny
- How is Shiny different from traditional applications?
- Installing R Shiny
- Structure of a Shiny app.
- Your first Shiny App

What is R Shiny?

- Shiny is an R package that allows users to build interactive web apps. This tool creates an HTML equivalent web app from Shiny code. We integrate native HTML and CSS code with R Shiny functions to make application presentable. Shiny combines the computational power of R with the interactivity of the modern web. Shiny creates web apps that are deployed on the web using your server or R Shiny's hosting services.

Features of R Shiny:

- Create easy applications with basic or no knowledge of web tools
- Integrate Shiny with native web tools to improve flexibility and productivity
- Pre-built I/O and render functions
- Easy rendering of the application content without multiple reloads
- Feature to add computed (or processed) outputs from R scripts
- Add live reports and visualizations.

That brings us to the question:

How is Shiny different from traditional applications?

Let's us take an example of a weather application, whenever the user refreshes/loads the page or change any input, it should update the whole page or part of the page using JS. This adds load to the server-side for processing. Shiny allows the user to isolate or render(or reload) elements in the app which reduces server load. Scrolling through pages was easy in traditional web applications but was difficult with Shiny apps. The structure of the code plays the main role in understanding and debugging the code. This feature is crucial for shiny apps with respect to other applications.

Let’s move on to the next topic in R Shiny tutorial, installing the R Shiny package.

Installing R Shiny

Installing Shiny is like installing any other package in R. Go to R Console and run the below command to install the Shiny package.

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

```
> install.packages("shiny")
WARNING: Rtools is required to build R packages but is not currently installed. Please download and install t
he appropriate version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing package into 'C:/Users/Cherukuri_Sindhu/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.6/shiny_1.3.2.zip'
Content type 'application/zip' length 4694770 bytes (4.5 MB)
downloaded 4.5 MB

package 'shiny' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Cherukuri_Sindhu\AppData\Local\Temp\RtmpiEPjec\downloaded_packages
```

Installing R Shiny

Once you have installed, load the Shiny package to create Shiny apps.

```
library(shiny)
```

Before we move any further in this R shiny tutorial, let’s see and understand the structure of a Shiny application.

Structure of a Shiny app

Shiny consists of 3 components:

1. User Interface
2. Server
3. ShinyApp

Structure of a Shiny app

1. User Interface Function

User Interface (UI) function defines the layout and appearance of the app. You can add CSS and HTML tags within the app to make the app more presentable. The function contains all inputs and outputs to be displayed in the app. Each element (division or tab or menu) inside the app is defined using functions. These are accessed using a unique id, like HTML elements.

Structure of a Shiny app

2. Server Function

Server function defines the server-side logic of the Shiny app. It involves creating functions and outputs that use inputs to produce various kinds of output. Each client (web browser) calls the server function when it first loads the Shiny app. Each output stores the return value from the render functions.

These functions capture an R expression and do calculations and pre-processing on the expression. Use the render* function that corresponds to the output you are defining. We access input widgets using **input$[widget-id]**. These input variables are reactive values. Any intermediate variables created using input variables need to be made reactive using **reactive({ })**. Access the variables using ( ).

Structure of a Shiny app

2. Server Function

render* functions perform the computation inside the server function and store in the output variables. The output needs to be saved with **output$[output variable name]**. Each **render*** function takes a single argument i.e, an R expression surrounded by braces, { }.

Structure of a Shiny app

3. ShinyApp Function

shinyApp() function is the heart of the app which calls UI and server functions to create a Shiny App.

The below image shows the outline of the Shiny app.

Your first Shiny app

There are several ways to create a Shiny app. The simplest is to create a new directory for your app, and put a single file called app.R in it. This app.R file will be used to tell Shiny both how your app should look, and how it should behave.

Try it out by creating a new directory, and adding an app.R file that looks like this:

```
library(shiny)
ui <- fluidPage(
  "Hello, world!"
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

Your first Shiny app

This is a complete, if trivial, Shiny app! Looking closely at the code above, our app.R does four things:

1. It calls `library(shiny)` to load the shiny package.
2. It defines the user interface, the HTML webpage that humans interact with. In this case, it's a page containing the words "Hello, world!".
3. It specifies the behaviour of our app by defining a `server` function. It's currently empty, so our app doesn't do anything, but we'll be back to revisit this shortly.
4. It executes `shinyApp(ui, server)` to construct and start a Shiny application from UI and server.

Your first Shiny app

RStudio Tip: There are two convenient ways to create a new app in RStudio:

- Create a new directory and an `app.R` file containing a basic app in one step by clicking **File | New Project**, then selecting **New Directory** and **Shiny Web Application**.
- If you've already created the `app.R` file, you can quickly add the app boilerplate by typing "shinyapp" and pressing Shift+Tab.

Your first Shiny app

Running and stopping

There are a few ways you can run this app:

- Click the Run App (Figure 1.1) button in the document toolbar.

Figure 1.1: The Run App button can be found at the top-right of the source pane.

- Use a keyboard shortcut: Cmd/Ctrl + Shift + Enter.

- If you're not using RStudio, you can (source()) the whole document, or call shiny::runApp() with the path to the directory containing app.R.

Your first Shiny app

Running and stopping

Pick one of these options, and check that you see the same app as in Figure 1.2. Congratulations! You've made your first Shiny app.

Figure 1.2: The very basic shiny app you'll see when you run the code above

Your first Shiny app

Adding UI controls

Next, we'll add some inputs and outputs to our UI so it's not quite so minimal. We're going to make a very simple app that shows you all the built-in data frames included in the datasets package.

Replace your ui with this code:

```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices = ls("package:datasets")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)
```

Your first Shiny app

Adding UI controls

This example uses four new functions:

- `fluidPage()` is a layout function that sets up the basic visual structure of the page. You'll learn more about them later in this course.

- `selectInput()` is an input control that lets the user interact with the app by providing a value. In this case, it's a select box with the label "Dataset" and lets you choose one of the built-in datasets that come with R. You'll learn more about inputs later in this course.

- `verbatimTextOutput()` and `tableOutput()` are output controls that tell Shiny where to put rendered output (we'll get into the how in a moment). `verbatimTextOutput()` displays code and `tableOutput()` displays tables. You'll learn more about outputs later in this course.

Your first Shiny app

Adding UI controls

Layout functions, inputs, and outputs have different uses, but they are fundamentally the same under the covers: they're all just fancy ways to generate HTML, and if you call any of them outside of a Shiny app, you'll see HTML printed out at the console. Don't be afraid to poke around to see how these various layouts and controls work under the hood.

Go ahead and run the app again. You'll now see Figure 1.3, a page containing a select box. We only see the input, not the two outputs, because we haven't yet told Shiny how the input and outputs are related.

Your first Shiny app

Adding UI controls

Figure 1.3: The datasets app with UI

Your first Shiny app

Adding behaviour

Next, we'll bring the outputs to life by defining them in the server function.

Shiny uses reactive programming to make apps interactive. You'll learn more about reactive programming later in this course, but for now, just be aware that it involves telling Shiny *how* to perform a computation, not ordering Shiny to actually go *do it*. It's like the difference between giving someone a recipe versus demanding that they go make you a sandwich.

We'll tell Shiny how to fill in the summary and table outputs in the sample app by providing the "recipes" for those outputs. Replace your empty server function with this:

Your first Shiny app

Adding UI controls

```
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })

  output$table <- renderTable({
    dataset <- get(input$dataset, "package:datasets")
    dataset
  })
}
```

Your first Shiny app

Adding behaviour

The left-hand side of the assignment operator (<-), output$ID, indicates that you're providing the recipe for the Shiny output with that ID. The right-hand side of the assignment uses a specific **render function** to wrap some code that you provide. Each render{Type} function is designed to produce a particular type of output (e.g. text, tables, and plots), and is often paired with a {type}Output function. For example, in this app, renderPrint() is paired with verbatimTextOutput() to display a statistical summary with fixed-width (verbatim) text, and renderTable() is paired with tableOutput() to show the input data in a table.

Run the app again and play around, watching what happens to the output when you change an input. Figure 1.4 shows what you should see when you open the app.

Your first Shiny app

Adding behaviour

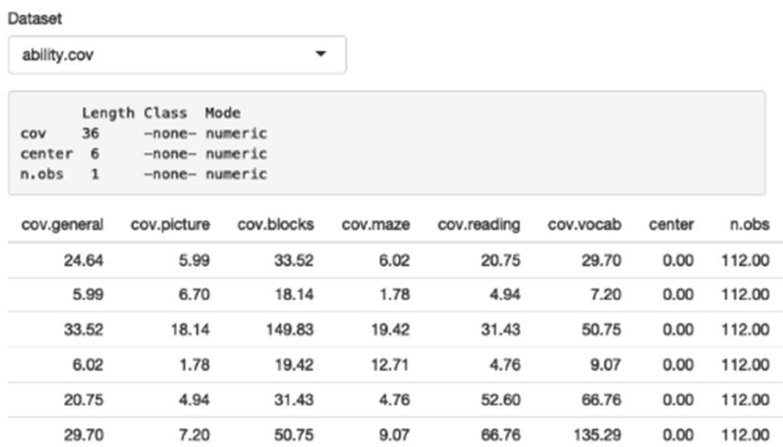


Figure 1.4: Now that we've provided a server function that connects outputs and inputs, we have a fully functional app

Notice that the summary and table update whenever you change the input dataset. This dependency is created implicitly because we've referred to input$dataset within the output functions. input$dataset is populated with the current value of the UI component with id dataset, and will cause the outputs to automatically update whenever that value changes. This is the essence of **reactivity**: outputs automatically *react* (recalculate) when their inputs change.

Your first Shiny app

Reducing duplication with reactive expressions

Even in this simple example, we have some code that's duplicated: the following line is present in both outputs.

```
dataset <- get(input$dataset, "package:datasets")
```

In every kind of programming, it's poor practice to have duplicated code; it can be computationally wasteful, and more importantly, it increases the difficulty of maintaining or debugging the code. It's not that important here, but I wanted to illustrate the basic idea in a very simple context.

In traditional R scripting, we use two techniques to deal with duplicated code: either we capture the value using a variable, or capture the computation with a function. Unfortunately, neither of these approaches work here, for reasons you'll learn later in this course, and we need a new mechanism: **reactive expressions**.

Your first Shiny app

Reducing duplication with reactive expressions

You create a reactive expression by wrapping a block of code in `reactive({...})` and assigning it to a variable, and you use a reactive expression by calling it like a function. But while it looks like you're calling a function, a reactive expression has an important difference: it only runs the first time it is called and then it caches its result until it needs to be updated.

We can update our `server()` to use reactive expressions, as shown below. The app behaves identically, but works a little more efficiently because it only needs to retrieve the dataset once, not twice.

Your first Shiny app

Reducing duplication with reactive expressions

```
server <- function(input, output, session) {
  # Create a reactive expression
  dataset <- reactive({
    get(input$dataset, "package:datasets")
  })

  output$summary <- renderPrint({
    # Use a reactive expression by calling it like a function
    summary(dataset())
  })

  output$table <- renderTable({
    dataset()
  })
}
```

We'll come back to reactive programming multiple times, but even armed with a cursory knowledge of inputs, outputs, and reactive expressions, it's possible to build quite useful Shiny apps!

Your first Shiny app

Summary

In these sections you've created a simple app — it's not very exciting or useful, but seen how easy it is to construct a web app using your existing R knowledge. In the next two sections, you'll learn more about user interfaces and reactive programming, the two basic building blocks of Shiny. Now is a great time to grab a copy of the Shiny cheatsheet. This is a great resource to help jog your memory of the main components of a Shiny app.

https://shiny.posit.co/r/articles/start/cheatsheet/

LECTURE 2

Basic UI

Basic UI

Inputs

As we saw in the previous section, you use functions like sliderInput(), selectInput(), textInput(), and numericInput() to insert input controls into your UI specification. Now we'll discuss the common structure that underlies all input functions and give a quick overview of the inputs built into Shiny.

The inputId has two constraints:

- It must be a simple string that contains only letters, numbers, and underscores (no spaces, dashes, periods, or other special characters allowed!). Name it like you would name a variable in R.

- It must be unique. If it's not unique, you'll have no way to refer to this control in your server function!

Basic UI

Inputs

Most input functions have a second parameter called label. This is used to create a human-readable label for the control. Shiny doesn't place any restrictions on this string, but you'll need to carefully think about it to make sure that your app is usable by humans! The third parameter is typically value, which, where possible, lets you set the default value. The remaining parameters are unique to the control.

When creating an input, I recommend supplying the inputId and label arguments by position, and all other arguments by name:

```
sliderInput("min", "Limit (minimum)", value = 50, min = 0, max = 100)
```

Basic UI

Inputs

The following sections describe the inputs built into Shiny, loosely grouped according to the type of control they create. The goal is to give you a rapid overview of your options, not to exhaustively describe all the arguments. I'll show the most important parameters for each control below, but you'll need to read the documentation to get the full details.

Basic UI

Inputs – Free text

Collect small amounts of text with `textInput()`, passwords with `passwordInput()`, and paragraphs of text with `textAreaInput()`.

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "What's your password?"),
  textAreaInput("story", "Tell me about yourself", rows = 3)
)
```

Basic UI

Inputs – Free text

If you want to ensure that the text has certain properties you can use validate(), which we'll come back to later in this course.

Basic UI

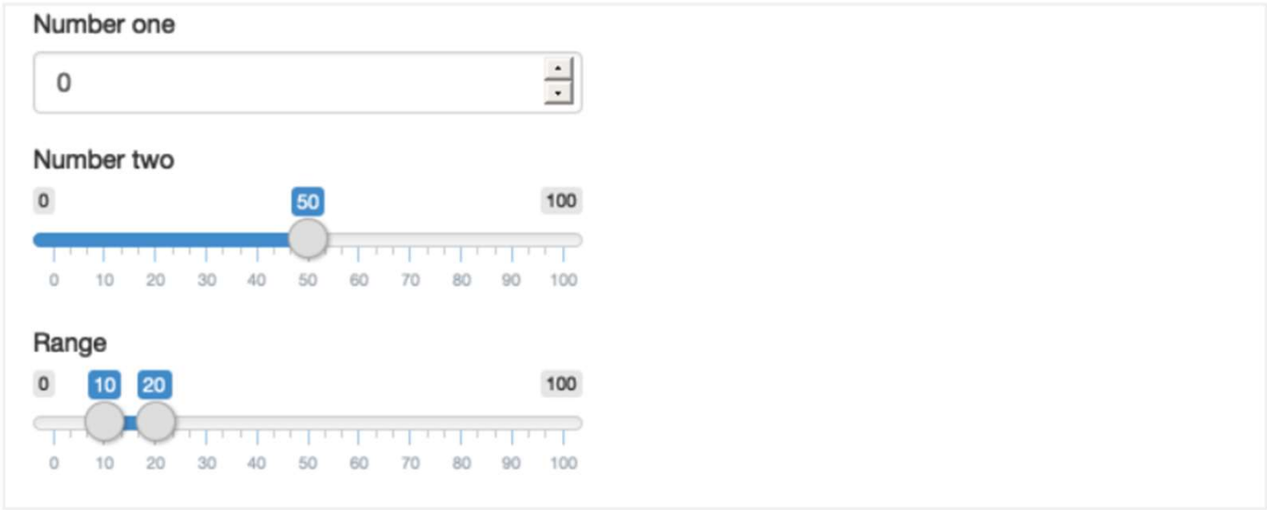
Inputs – Numeric inputs

To collect numeric values, create a constrained text box with numericInput() or a slider with sliderInput(). If you supply a length-2 numeric vector for the default value of sliderInput(), you get a “range” slider with two ends.

```
ui <- fluidPage(
  numericInput("num", "Number one", value = 0, min = 0, max = 100),
  sliderInput("num2", "Number two", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```

Basic UI

Inputs – Numeric inputs



Generally, It is recommended only using sliders for small ranges, or cases where the precise value is not so important. Attempting to precisely select a number on a small slider is an exercise in frustration!

Sliders are extremely customisable and there are many ways to tweak their appearance. See ?sliderInput and https://shiny.rstudio.com/articles/sliders.html for more details.

Basic UI

Inputs – Dates

Collect a single day with dateInput() or a range of two days with dateRangeInput(). These provide a convenient calendar picker, and additional arguments like datesdisabled and daysofweekdisabled allow you to restrict the set of valid inputs.

```
ui <- fluidPage(
  dateInput("dob", "When were you born?"),
  dateRangeInput("holiday", "When do you want to go on vacation next?")
)
```

Basic UI

Inputs – Dates

Date format, language, and the day on which the week starts defaults to US standards. If you are creating an app with an international audience, set format, language, and weekstart so that the dates are natural to your users.

Basic UI

Inputs – Limited choices

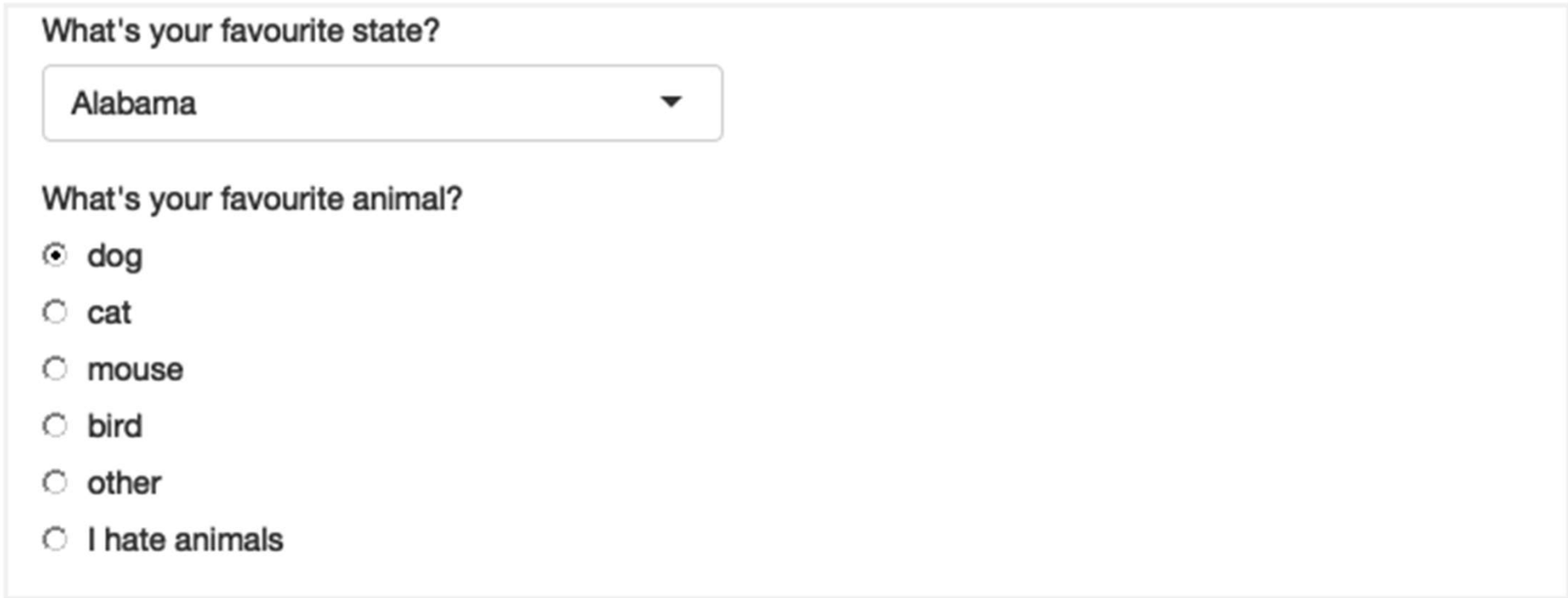
There are two different approaches to allow the user to choose from a prespecified set of options: selectInput() and radioButtons().

```
animals <- c("dog", "cat", "mouse", "bird", "other", "I hate animals")

ui <- fluidPage(
  selectInput("state", "What's your favourite state?", state.name),
  radioButtons("animal", "What's your favourite animal?", animals)
)
```

Basic UI

Inputs – Limited choices



Basic UI

Inputs – Limited choices

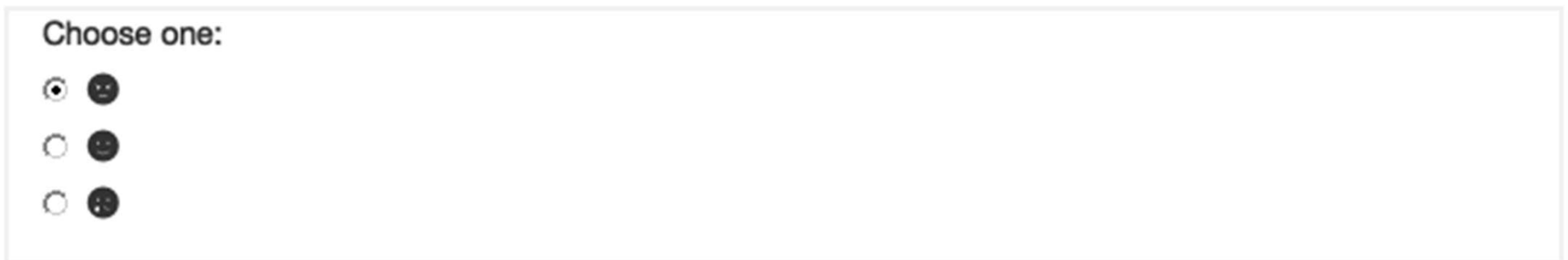
Radio buttons have two nice features: they show all possible options, making them suitable for short lists, and via the choiceNames/choiceValues arguments, they can display options other than plain text. choiceNames determines what is shown to the user; choiceValues determines what is returned in your server function.

```
ui <- fluidPage(
  radioButtons("rb", "Choose one:",
    choiceNames = list(
      icon("angry"),
      icon("smile"),
      icon("sad-tear")
    ),
    choiceValues = list("angry", "happy", "sad")
  )
)
```

Basic UI

Inputs – Limited choices

```
#> This Font Awesome icon ('angry') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`
#> This Font Awesome icon ('smile') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`
#> This Font Awesome icon ('sad-tear') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`
```

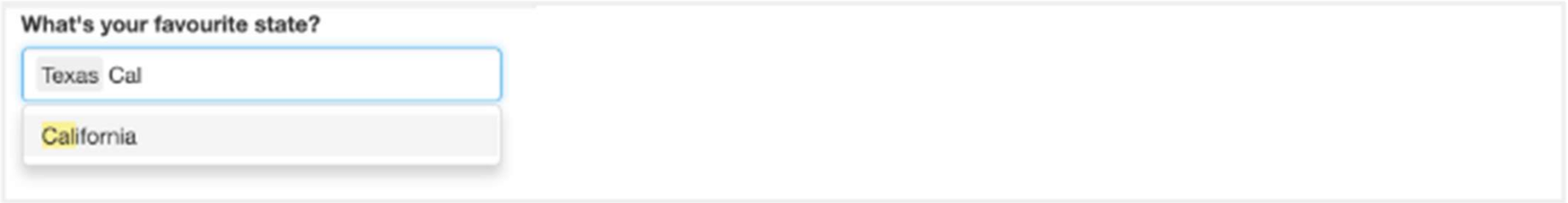


Basic UI

Inputs – Limited choices

Dropdowns created with `selectInput()` take up the same amount of space, regardless of the number of options, making them more suitable for longer options. You can also set `multiple = TRUE` to allow the user to select multiple elements.

```
ui <- fluidPage(
  selectInput(
    "state", "What's your favourite state?", state.name,
    multiple = TRUE
  )
)
```



Basic UI

Inputs – Limited choices

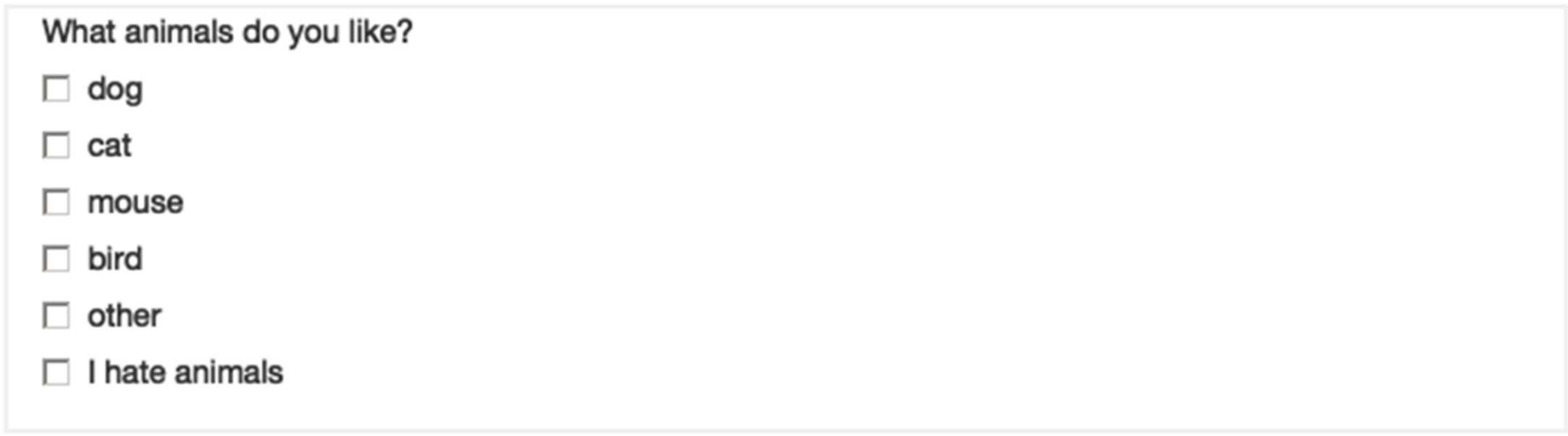
If you have a very large set of possible options, you may want to use “server-side” selectInput() so that the complete set of possible options are not embedded in the UI (which can make it slow to load), but instead sent as needed by the server. You can learn more about this advanced topic at https://shiny.rstudio.com/articles/selectize.html#server-side-selectize.

There’s no way to select multiple values with radio buttons, but there’s an alternative that’s conceptually similar: checkboxGroupInput().

Basic UI

Inputs – Limited choices

```
ui <- fluidPage(
  checkboxGroupInput("animal", "What animals do you like?", animals)
)
```

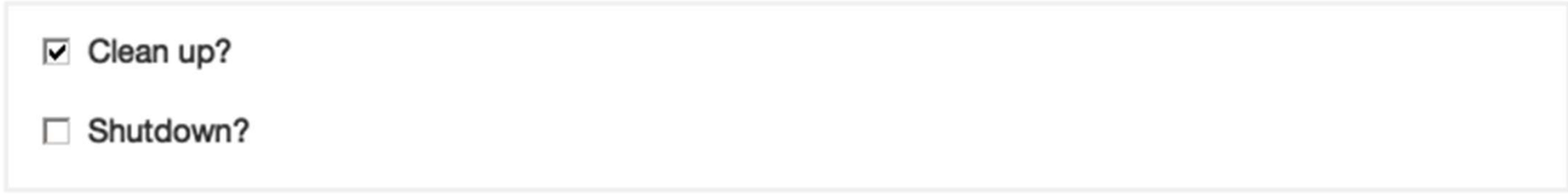


Basic UI

Inputs – Limited choices

If you want a single checkbox for a single yes/no question, use checkboxInput():

```
ui <- fluidPage(
  checkboxInput("cleanup", "Clean up?", value = TRUE),
  checkboxInput("shutdown", "Shutdown?")
)
```

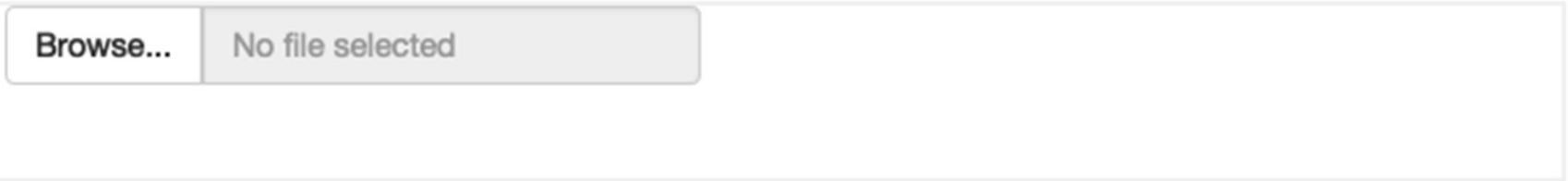


Basic UI

Inputs – File uploads

Allow the user to upload a file with fileInput():

```
ui <- fluidPage(
  fileInput("upload", NULL)
)
```



fileInput() requires special handling on the server side, and is discussed in detail later in this course.

Basic UI

Inputs – Action buttons

Let the user perform an action with `actionButton()` or `actionLink()`:

```
ui <- fluidPage(
  actionButton("click", "Click me!"),
  actionButton("drink", "Drink me!", icon = icon("cocktail"))
)
#> This Font Awesome icon ('cocktail') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`
```

Basic UI

Inputs – Action buttons

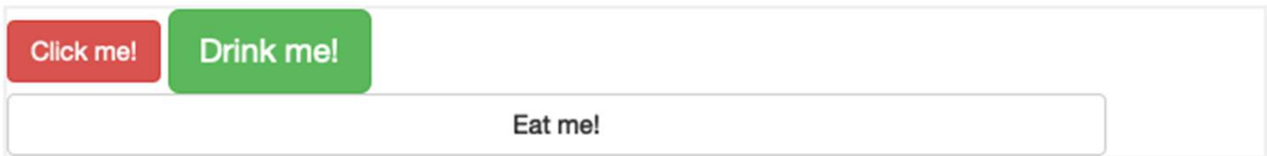
Actions links and buttons are most naturally paired with `observeEvent()` or `eventReactive()` in your server function. You haven't learned about these important functions yet, but we'll come back to them later in this course.

You can customise the appearance using the class argument by using one of `"btn-primary"`, `"btn-success"`, `"btn-info"`, `"btn-warning"`, or `"btn-danger"`. You can also change the size with `"btn-lg"`, `"btn-sm"`, `"btn-xs"`. Finally, you can make buttons span the entire width of the element they are embedded within using `"btn-block"`.

Basic UI

Inputs – Action buttons

```
ui <- fluidPage(
  fluidRow(
    actionButton("click", "Click me!", class = "btn-danger"),
    actionButton("drink", "Drink me!", class = "btn-lg btn-success")
  ),
  fluidRow(
    actionButton("eat", "Eat me!", class = "btn-block")
  )
)
```



The class argument works by setting the class attribute of the underlying HTML, which affects how the element is styled. To see other options, you can read the documentation for Bootstrap, the CSS design system used by Shiny: http://bootstrapdocs.com/v3.3.6/docs/css/#buttons.

Basic UI

Outputs

Outputs in the UI create placeholders that are later filled by the server function. Like inputs, outputs take a unique ID as their first argument: if your UI specification creates an output with ID "plot", you'll access it in the server function with output$plot.

Each output function on the front end is coupled with a render function in the back end. There are three main types of output, corresponding to the three things you usually include in a report: text, tables, and plots. The following sections show you the basics of the output functions on the front end, along with the corresponding render functions in the back end.

Basic UI

Outputs – Text

Output regular text with textOutput() and fixed code and console output with verbatimTextOutput().

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("code")
)
server <- function(input, output, session) {
  output$text <- renderText({
    "Hello friend!"
  })
  output$code <- renderPrint({
    summary(1:10)
  })
}
```

Basic UI

Outputs – Text

Hello friend!

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

Note that the {} are only required in render functions if need to run multiple lines of code. As you'll learn shortly, you should do as little computation in your render functions as possible, which means you can often omit them. Here's what the server function above would look like if written more compactly::

```
server <- function(input, output, session) {
  output$text <- renderText("Hello friend!")
  output$code <- renderPrint(summary(1:10))
}
```

Basic UI

Outputs – Text

Note that there are two render functions which behave slightly differently:

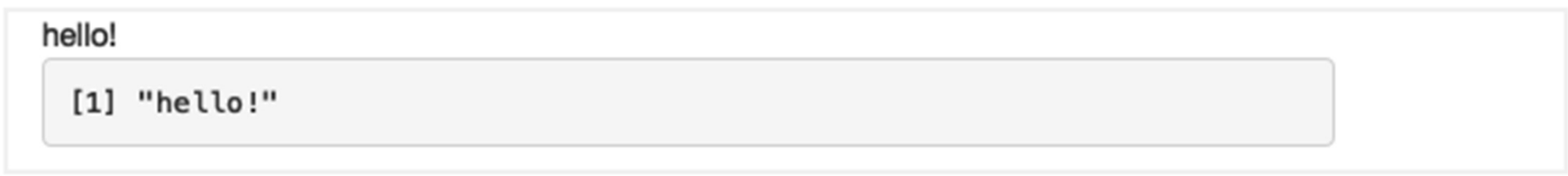
- `renderText()` combines the result into a single string, and is usually paired with `textOutput()`
- `renderPrint()` prints the result, as if you were in an R console, and is usually paired with `verbatimTextOutput()`.

We can see the difference with a toy app:

Basic UI

Outputs – Text

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("print")
)
server <- function(input, output, session) {
  output$text <- renderText("hello!")
  output$print <- renderPrint("hello!")
}
```



This is equivalent to the difference between `cat()` and `print()` in base R.

Basic UI

Outputs – Tables

There are two options for displaying data frames in tables:

- `tableOutput()` and `renderTable()` render a static table of data, showing all the data at once.
- `dataTableOutput()` and `renderDataTable()` render a dynamic table, showing a fixed number of rows along with controls to change which rows are visible.

`tableOutput()` is most useful for small, fixed summaries (e.g. model coefficients); `dataTableOutput()` is most appropriate if you want to expose a complete data frame to the user. If you want greater control over the output of `dataTableOutput()`, we highly recommend the reactable package by Greg Lin.

Basic UI

Outputs – Tables

```
ui <- fluidPage(
  tableOutput("static"),
  dataTableOutput("dynamic")
)
server <- function(input, output, session) {
  output$static <- renderTable(head(mtcars))
  output$dynamic <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

Basic UI

Outputs – Tables

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.00	6.00	160.00	110.00	3.90	2.62	16.46	0.00	1.00	4.00	4.00
21.00	6.00	160.00	110.00	3.90	2.88	17.02	0.00	1.00	4.00	4.00
22.80	4.00	108.00	93.00	3.85	2.32	18.61	1.00	1.00	4.00	1.00
21.40	6.00	258.00	110.00	3.08	3.21	19.44	1.00	0.00	3.00	1.00
18.70	8.00	360.00	175.00	3.15	3.44	17.02	0.00	0.00	3.00	2.00
18.10	6.00	225.00	105.00	2.76	3.46	20.22	1.00	0.00	3.00	1.00

Show 10 entries Search:

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.46	0	1	4	4
21	6	160	110	3.9	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb

Showing 1 to 5 of 32 entries Previous 1 2 3 4 5 6 7 Next

Basic UI

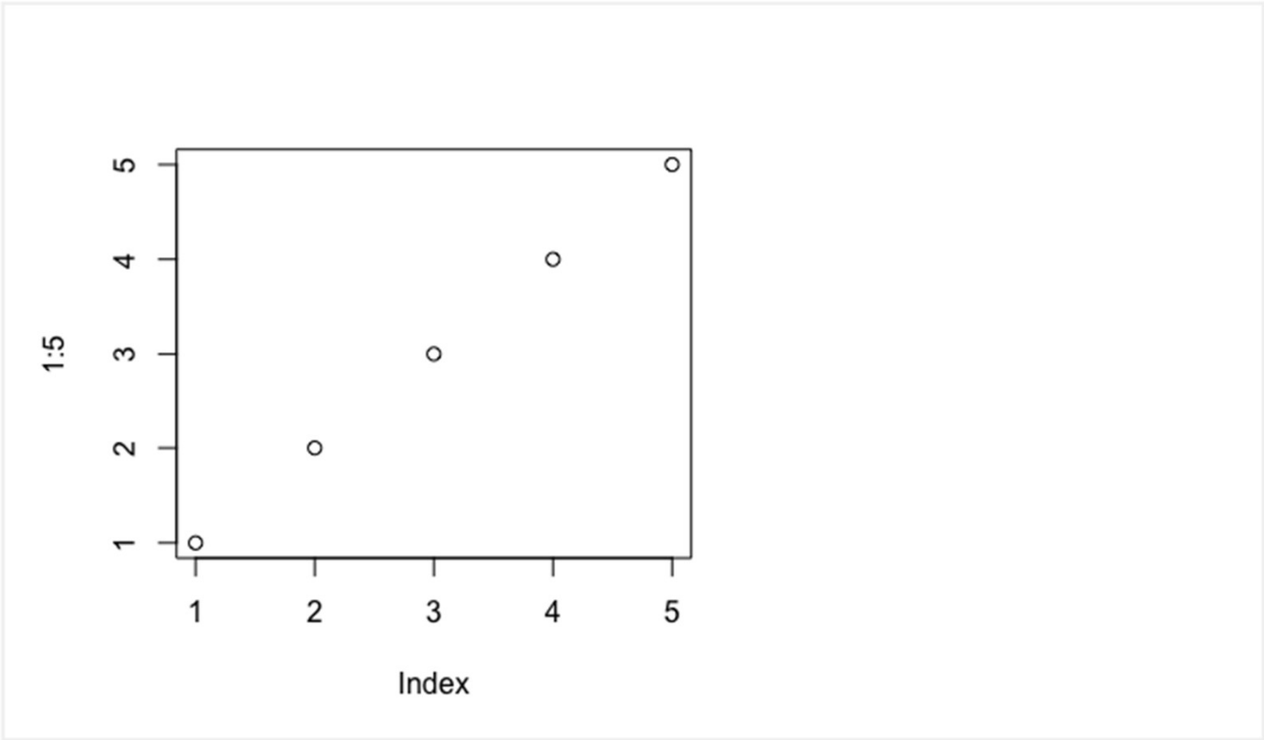
Outputs – Plots

You can display any type of R graphic (base, ggplot2, or otherwise) with plotOutput() and renderPlot():

```
ui <- fluidPage(
  plotOutput("plot", width = "400px")
)
server <- function(input, output, session) {
  output$plot <- renderPlot(plot(1:5), res = 96)
}
```

Basic UI

Outputs – Plots



By default, plotOutput() will take up the full width of its container (more on that shortly), and will be 400 pixels high. You can override these defaults with the height and width arguments. We recommend always setting res = 96 as that will make your Shiny plots match what you see in RStudio as closely as possible.

Basic UI

Outputs – Plots

Plots are special because they are outputs that can also act as inputs. `plotOutput()` has a number of arguments like `click`, `dblclick`, and `hover`. If you pass these a string, like `click = "plot_click"`, they'll create a reactive input (`input$plot_click`) that you can use to handle user interaction on the plot, e.g. clicking on the plot. We'll come back to interactive plots in Shiny later in this course.

Basic UI

Outputs – Downloads

You can let the user download a file with `downloadButton()` or `downloadLink()`. These require new techniques in the server function, so we'll come back to that later in this course.

Basic UI

Summary

This section has introduced you to the major input and output functions that make up the front end of a Shiny app. This was a big info dump, so don't expect to remember everything after a single read. Instead, come back to this chapter when you're looking for a specific component: you can quickly scan the figures, and then find the code you need.

In the next section, we'll move on to the back end of a Shiny app: the R code that makes your user interface come to life.