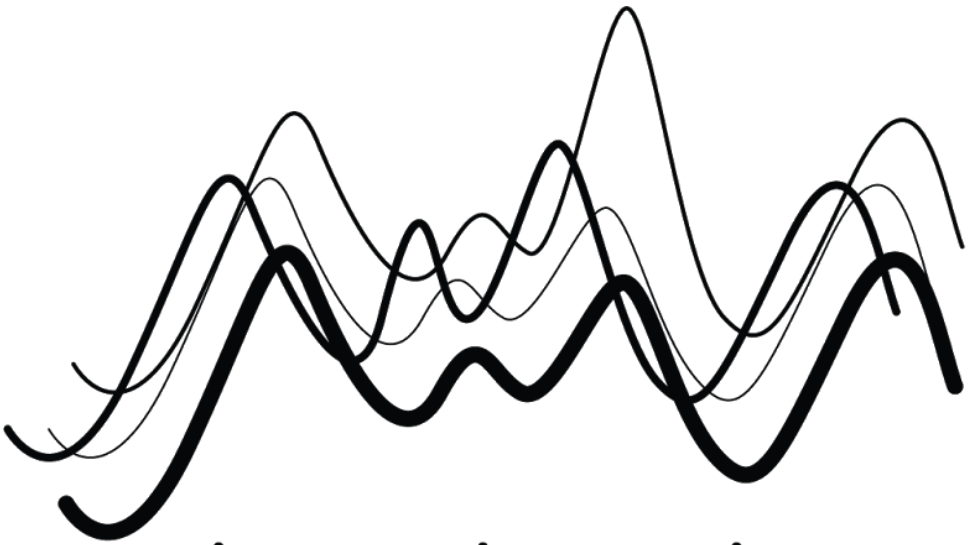
INTERACTIVE GRAPHICS WITH SHINY



jumping rivers

Contents

1
Shiny

Shiny is a R package that provides a web application framework for R[1]. It is a bit of a hot topic within the R community at the moment as it allows fast and simple development for analyses, graphics and documents that have an interactive browser based front end. This makes a nice interface when sharing results, reports, learning tools, dynamic and interactive presentations and so on. I won't include too much information here as the online tutorials are an excellent place to start. This section is intended to make you aware of shiny and the sorts of things that can be achieved with it[2]

[1] https://goo.gl/tzZs3T

[2] Further tutorials can be found at shiny.rstudio.com/tutorial/

Examples of some of the things that is possible to do with shiny can be found at the following websites with new things being added all the time.

- http://shiny.rstudio.com/gallery/
- http://www.showmeshiny.com/
- http://shiny.rstudio.com/tutorial/
- http://shiny.rstudio.com/articles/

Every shiny app is maintained by a computer running R. This computer could be your own laptop/desktop machine, or a machine anywhere in the world.

Exercise: Complete exercise 1 in the `chapter4` vignette.

```
vignette("chapter4", package="nclRshiny")
```

We can create a shiny app within an Rmarkdown file. Just create an `.Rmd` file with the header

```
---
title: "First app"
runtime: shiny
---
```

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

Table 1.1: Standard shiny widgets.

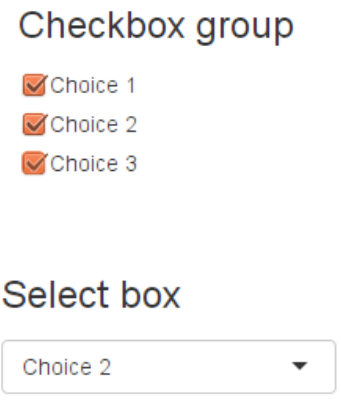


Figure 1.1: The checkbox and select box widgets.

Input: control widgets

Shiny comes with a collection of pre-built widgets[3]. A widget is one route that makes your application interactive. Each widget is a separate R function. Standard widgets include check-boxes, text inputs and radio buttons - see table 2.2 for a full list.

Control widgets are just functions that generate `HTML` code. For example, this widget[4]

```
selectInput("movie_type", # unique id
            label = "Movie genre", # Text for web
            c("Romance", "Action", "Animation"))
```

generates the rather unpleasant looking

```
<div class="form-group shiny-input-container">
 <label class="control-label" for="movie_type">Movie genre</label>
  <div>
   <select id="movie_type">
    <option value="Romance" selected>Romance</option>
    <option value="Action">Action</option>
    <option value="Animation">Animation</option></select>
   <script type="application/json" data-for="movie_type"
                                    data-nonempty="">{}</script>
 </div>
</div>
```

All shiny widget widgets have a similar format. The first argument is `inputId` and must be a unique id. The second argument is the text that will be display in the app. The other arguments are widget specific.

Exercise: Complete exercise 2 in the `chapter4` vignette.

[4] We'll use the IMDB data set to build an example shiny app in this chapter.

[3] `http://shiny.rstudio.com/gallery/widget-gallery.html`

It's worth highlighting key words in the `selectInput` function an matching them in the `HTML`.

Rendered outputs

The next step is to use the values in the shiny widgets to dynamically change plots and tables. When we create a shiny widget, the value of the widget is bound to `input$inputId`[5]. This object is a *reactive* value[6].

- What code will the server run? Code that builds something out of *reactive* values
- When will the code run? When the *reactive* value changes.

[5] In the example above, it would be bound to `input$movie_type`

[6] You cannot call a reactive value within a normal R session.

EXERCISE: Complete exercise 3 in the `chapter4` vignette.

Rendered (outputs) respond whenever a *reactive* value changes. For example, the scatter plot when we were exploring the occurrence of names. The Shiny package includes a wide variety of render function (See 2.2).

Any htmlwidgets you include also need to be wrapped in an appropriate `render*` function, e.g. for `plotly` widgets, there is `renderPlotly`.

Function	Output type
`renderPlot`	R graphics output
`renderPrint`	R printed output
`renderTable`	Data frame, matrix
`renderText`	Character vectors

Table 1.2: Key render objects.

CONTINUING WITH THE `movies` example, we could display the number of movies

```
renderText({
        type = movies[,input$movie_type] == 1
        nrow(movies[type,])
})
```

Or create a histogram of movie lengths

```
renderPlot({
        type = movies[,input$movie_type] == 1
        hist(movies[type,]$length)
})
```

EXERCISE: Complete exercise 4 in the `chapter4` vignette.

Reactive programming

When we use the `render*` functions, we are carrying out reactive programming. If you think about a standard R session, when we change a value of an object, print and plot statements are not rerun, i.e.

```
x = 1
print(x)

## [1] 1

x = 2 ## The print statement above is not updated.
```

Shiny gives the illustration that `print(x)` is updated. A naive way of doing this is to constantly[7] check if any objects have changed. However for a serious application, the number of checks required quickly grows. Instead, shiny just checks key objects. If these objects have changed, the necessary changes are propagated through.

When we include `input$X` within a `render*` function, we are telling shiny that we will need to rerun the `render*` function if the value of `input$X` ever changes. Hence, we control what gets re-run with reactive expressions[8].

> A reactive expression is an R expression that uses widget input and returns a value.

A reactive expression will update this value whenever the original widget changes.

In the example above, we calculated the variable `type` twice. This isn't particularly efficient (and when dealing with web pages, we need calculations to be as quick as possible). Shiny allows us to create our own reactive expressions using the `reactive` function. For example, we could create a new `reactive` variable[9]

```
sub_movies  = reactive(movies[movies[input$movie_type]==1,])
```

The first argument in `reactive` is an expression. If the expression has more than one line, we need to enclose it with { brackets, i.e.

```
sub_movies = reactive({
    type = movies[,input$movie_type] == 1
    movies[type,]
})
```

Calling `reactive` object builds a reactive expression; that is a reactive object made from reactive values. To access the object, we treat `sub_movies` as a function, i.e.

```
renderText(nrow(sub_movies()))
renderPlot(hist(sub_movies()$length))
```

Exercise: Complete exercise 5 in the `chapter4` vignette[10].

The `eventReactive` function

In many shiny applications we might not wish to redraw or recalculate a value after **every** slider change or menu selection. Instead we might want to make multiple selections and only evaluate on a button press. In shiny this is achieved by using the special `actionButton` and `actionLink` widgets in conjunction with the `eventReactive` function. We construct our UI as normal

[7] For a human being, every few microseconds appears to be constant.

[8] The `renderText` and `renderPlot` example above are reactive statements.

[9] It's reactive because it depends on `input$movie_type`.

[10] In the above piece of code, try removing `reactive`. Does the code still work?

UI: User interface.

```
selectInput("movie_type", label = "Movie genre",
            c("Romance", "Action", "Animation"))
actionButton("plot", "Plot it now!!")
```

But we create our reactive values slightly differently

```
sub_movies = eventReactive(input$plot,{
    type = movies[,input$movie_type] == 1
    movies[type,]
})
```

The object `sub_movies` is still a reactive expression, **but** will only be re-evaluated when the `plot` button is pressed. The `renderPlot` and `renderText` function calls are unchanged.

Exercise: Complete exercise 6 in the `chapter4` vignette.

The `observeEvent` function

We are starting to get a bit more technical in this section. Don't worry if you don't get the concepts straight away; these ideas are more important for larger apps.

The reactive programming framework within shiny is primarily designed for calculated values (reactive expressions) and side-effect-causing actions (observers) that respond to any of their inputs changing. Typically that's what we want. But sometimes you want to wait for a specific action, such as clicking the `actionButton`, before calculating an expression or taking an action. When we only have a single `action` button, then `eventReactive` is often suitable. However suppose we have multiple buttons

```
actionButton("romance", "Romance")
actionButton("action", "Action")
```

The `eventReactive` approach doesn't work[11]. Instead, we need to use the `observeEvent` functions to monitor for changes and `reactiveValues` to pass variables.

[11] Try and think of a solution if you don't believe me.

In the movies example, we create an initial data set and create a reactive list object

```
rvs = reactiveValues(data=movies)
```

Next we monitor the buttons for changes, and update `rvs` as needed

```
observeEvent(input$romance, rvs$data = {
    type = movies[,input$movie_type] == 1
    movies[type,]
})
observeEvent(input$action, rvs$data = {
    type = movies[,input$movie_type] == 1
    movies[type,]
})
```

Then render the plot as usual

```
renderPlot(hist(rvs$data[, "length"]))
```

The `renderPlot` object depends on the *reactive* object `rvs$data`. This object will change whenever the `romance` or `action` buttons are clicked.

EXERCISE: Complete exercise 7 in the `chapter4` vignette.

The `isolate` function

Suppose the underlying data set is very large. Instead of plotting all the data, we could have a slider to indicate the number of rows to display

```
sliderInput("n", "Sample size", 10, 500, 100)
actionButton("romance", "Romance")
actionButton("action", "Action")
```

Initially, this would seem straightforward to incorporate this feature into our app

```
observeEvent(input$romance, {
    m =  movies[movies[, "Romance"]==1,]
    rows = sample(1:nrow(m), input$n)
    rvs$data = movies[rows,]
})
# Similar for Action
```

The `renderPlot` function would also be similar

```
renderPlot(hist(rvs$data[, "length"],
                main=paste("Sample size:", input$n)))
```

However, we've introduced a new reactive dependency. Whenever we change `input$n` the plot is redrawn with a new title, **but** the underlying data hasn't changed, since the `observeEvent` doesn't depend on `input$n`. This is where the `isolate`[12] function is handy. We simply place `input$n` inside an `isolate` function call

```
renderPlot(hist(rvs$data[, "length"],
                main=paste("Sample size:", isolate(input$n))))
```

[12] http://shiny.rstudio.com/articles/isolation.html

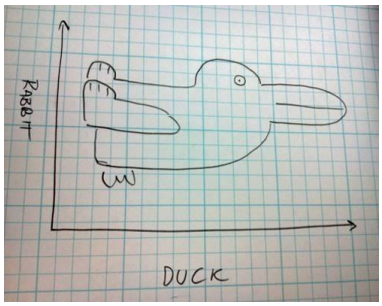


Figure 1.2: Source unknown.

2
A shiny app

Introduction

So far, we have relied on markdown and associated tools to help with the layout[1]. When we create a full shiny app, we have to worry more about the layout. Of course, this means we have finer control[2].

Shiny apps typically take the form of a pair of R scripts.

- The `server.R` file – controls the server side logic, creating graphics, manipulating input etc.
- The `ui.R`[3] file – controls the front end interface, layout, output renders etc.

Most Shiny apps have the same structure, these two R scripts saved together in a directory[4].

[1] Trust me, if you can get by with `flexdashboard`, then your life is much easier and happier.

[2] But also bracket madness

[3] `ui` - user interface.

[4] For simple apps we can have the necessary functions in a single file

The `ui.R` file

The `ui.R` file contains all of the code for the front end of the application, this file typically takes a similar format for each app[5].

Here is an example of a basic `ui.R` file.

[5] In shiny < 0.1.0 you had to write the function in the `shinyUI` function. Now you just need to make sure the `fluidPage` is the last thing evaluated.

```
library("shiny")
fluidPage(
    titlePanel("Shiny happy people"), #title
    ## Sidebar with a slider input for no. of points
    sidebarLayout(
        sidebarPanel(
            sliderInput("n", "Number of points:",
                        min = 1, max = 50, value = 30)
            ),
        ## Show a plot of the generated distribution
        mainPanel(plotOutput("scatter"))
        )
    )
```

The above creates a simple layout; a title, side bar and a single panel - see 2.1. Shiny comes with a variety of easy to use design. A standard design is the `fluidPage`.

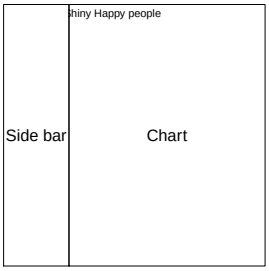


Figure 2.1: Simple shiny app layout.

The `server.R` file

The `server.R` file contains all of the logic for calculations that the user won't see. To complement the `ui.R` file, we have a server file that generates a scatter plot using samples from the Normal distribution. The number of samples is obtained via user input.

```
library("shiny")
# Function always has input & output
function(input, output) {
  # Expression that generates a plot.
  # A call to renderPlot indicates that:
  #  1) It is "reactive" and therefore should
  #    re-execute automatically when inputs change
  #  2) Its output type is a plot
  output$scatter = renderPlot({plot(rnorm(input$n))})
}
```

The `output$scatter` links to the `plotOutput('scatter')` plot in the `mainPanel` function.

Exercise: Complete exercise 1 in the `chapter5` vignette.

Running the app

Typically when we create a Shiny app, we save the files `ui.R` and `server.R` in a single directory. You can not have more than one app per directory. To run the app, use the `runApp` function. Assuming you have saved your files in a directory called `first_app`, then you can launch the app via

```
library("shiny")
runApp("my_app")
```

Assuming that the directory containing your app is in in your current working directory[6] the first argument of `runApp` is the directories name[7].

[6] Use `getwd()` to get your working directory.

[7] Alternatively, you can give the full path to `runApp`

Output objects

The above app is very similar to the flexdashboard. However there is one new addition, `*Output` functions. In the above code, `plotOutput` takes the plot function and creates an image. Likewise `tableOutput` is a quick way of displaying a table. The `textOutput` outputs some text. Each function in table 2.1 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

Table 2.1: Shiny output functions.

Layout

Shiny comes with tags. For example, the `em` tag indicates that text should be displayed as italics, and the `h1` tag is indicates a level 1

Function	HTML	Description	Function	HTML	Description
`p`	`<p>`	A paragraph of text	`a`	`<a>`	A hyper link
`hX`	`<hX>`	An `X` level header where `X` is $1, \ldots 6$	`br`	` `	A line break
`div`	`<div>`	A division of text with	`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>`	Italicised text	`HTML`		Directly passes a character string as HTML code

Table 2.2: Available shiny HTML functions.

heading. Most of the markdown tags we have encountered have a corresponding HTML tag.

The `shiny` package gives a number of functions for creating standard tags, e.g.

```
em("Some text")
```

For a complete list see table 2.2.

LAYOUT IS made easy[8] using the `fluidPage` function in the `ui.R` file. This function just generates HTML

[8] But not as easy as flexdashboard!

```
fluidPage()
```

This function automatically adjusts the display to the dimensions of the browser's window. The easiest interface is to have is to have the sidebar layout – a sidebar panel and a main panel:[9]

[9] If you are familiar with HTML, run the `fluidPage` code and inspect output.

```
fluidPage(
    titlePanel("Title panel"),# Title
    ## Sidebar style
    sidebarLayout(
        sidebarPanel("The sidebar"),
        mainPanel("Main panel")
        )
    )
```

The `sidebarLayout` function also has a position argument if we want to swap the side of the side bar, i.e.

```
sidebarLayout(position="right",
              sidebarPanel("The sidebar"),
              mainPanel("Main panel")
              )
```

To add content, just place it inside the `*Panel` function. For example

```
sidebarLayout(
    sidebarPanel("The sidebar",
                 p("Choose an option")),
    mainPanel("Main panel")
    )
```

The `sidebarLayout` function uses Shiny's lower-level grid layout functions. Columns are defined by the `column` function and rows by the `fluidRow` function. A page contains 12 columns.

Rows are created by the `fluidRow` function and include columns defined by the `column` function. Column widths are based on the Bootstrap 12-wide grid system, so should add up to 12 within a `fluidRow` container.

`wellPanel` just provides some additional formatting.

```
ui = fluidPage(
  titlePanel("I love movies"),
  fluidRow(
    column(4,
           wellPanel(
             selectInput("movie_type",
             label = "Movie genre",
                         c("Romance", "Action", "Animation"))
           )
    ),
    column(8, plotOutput("scatter"))
  )
)
```

Other layouts

We have only considered basic layouts in this chapter. Other more advanced layouts include

- Tab Sets. These are similar to the tabbed panels we used in the `flexdashboard` example. They are created using `tabSetPanel` function[10].
- Navbar Pages. There are similar to the pages in the `flexdashboard` example. They are created using the `navbarPage` function.
- Dashboards. Similar to the `flexdashboard`, but more powerful[11].

[10] `http://shiny.rstudio.com/articles/layout-guide.html`

[11] https://rstudio.github.io/shinydashboard/get_started.

Tab sets

We might use tabsets to have multiple pages or tabs within a single main panel. To create a tabset panel within the main panel we can use the `tabsetPanel` and `tabPanel` functions. For example

```
mainPanel(
      tabsetPanel(type = "tabs",
```

```
        tabPanel("Plot", plotOutput("plot")),
        tabPanel("Summary", verbatimTextOutput("summary")),
        tabPanel("Table", tableOutput("table"))
      )
)
```

Navbar layout

With tabsets we can only use the `tabsetPanel` function within `mainPanel`. Consequently, if we were using a side bar layout, we have a single side bar common to all tabs, and the overall structure is constant. With navbar pages, we can have completely unique tabbed pages, each tab having it's own layout.

EXERCISE: Complete exercise "Layouts" in the `chapter5` vignette.

Dashboards

The package `shinydashboard` provides an extension to shiny via a set of utility functions to facilitate easier creation of shiny dashboards. The end product is similar in feel to the sorts of things we could create with `flexdashboard`.

EXERCISE: Complete exercise "Dashboards" in the `chapter5` vignette.

Interactive graphics with shiny

Since `shiny` version[12] `0.12.0`, Shiny has built-in support for interacting with static plots, generated by base graphics or by `ggplot2`. To be clear, this is a shiny feature and not javascript. This feature allows you to select points and regions as well as zooming in/out of images.

As before, we create a ui component

[12] Running `packageVersion("shiny")` will give you the current version.

```
library("shiny")
## Basic layout. Two regions.
ui = basicPage(
  plotOutput("scatter", click = "plot_click"),
  verbatimTextOutput("info")
)
```

The key addition is in the `click` argument in the `plotOutput` function. This will capture where we click on the plot. The corresponding server part[13]

[13] `\n` in the `paste0` function tells R to print a line break. `\t` would print a tab break.

```
## Simulate data
x = signif(rnorm(10), 3); y = signif(rnorm(10), 3);
server = function(input, output) {
    output$scatter = renderPlot(plot(x, y))
    output$info = renderText({
```

```
        paste0("x=", input$plot_click$x,
               "\ny=", input$plot_click$y)
    })
}
```

uses the variable `input$plot_click` which contains the x-, y- coordinates of the nearest pixel.

EXERCISE: Complete exercise 1 in the `chapter5` vignette.

The session argument

The `server` function has an optional third argument `session`. This object is an R environment containing information and functionality relating to the session. .

The `session` environment contains things like client side data, allows functions to be called if the page is bookmarked etc., see https://shiny.rstudio.com/reference/shiny/latest/sessio

Whilst there are lots of things that you can do with this object, I find myself using it most often in a couple of scenarios.

- Sending messages to client's browser,
- Invalidating reactive objects.

Suppose we wanted to poll a file for changes and update our graphics accordingly, or perhaps supply a new set of random samples for some calculations. We could use the `session` argument, together with a function `invalidateLater` to acheive this.

```
server = function(input,output,session){
    data = reactive({
        invalidateLater(1000, session)
        runif(10)
    })
}
```

The server side logic here will invalidate the reactive context every 1000 milliseconds, causing the re–evaluation of the expression and it's dependencies. The consequence in this case is to generate a new set of random numbers each time.

EXERCISE: Complete exercise 5 in the `chapter5` vignette.

Deployment

If your document does not contain any shiny components, then you can just email the html file. However, if there are any shiny elements, there are two options.

1. Set up your own shiny server[14]. There is an open-source version and a paid version.
2. Host your app in the cloud with `shinyapps.io`. Again there are free and paid for versions.

[14] `https://www.rstudio.com/products/shiny/shiny-server/`

Full details on uploading your app are at

http://shiny.rstudio.com/articles/shinyapps.html

Once you have created a shiny apps account, to deploy your app, just run

```
library("rsconnect")
deployApp("name_of_app.Rmd")
```

When I deployed the dashboard from this course, the upload process took around twenty minutes, but subsequent uploads have been quicker.

Amazon web services

If you want to set up your own instance of shiny server, but not deal with the hassle of maintaining your own servers you could use something like Amazon web services for deploying your shiny apps and web pages.

Once you have signed up to an account, you can choose between free and paid computing instances. Once set up you can install your own shiny server, as well as any other software packages you might want.

The potential advantages of something like amazon is that there is always room to scale up if your organisation or web demand grows. You also have control over what is installed and can integrate more easily with other tools (as compared to shinyapps.io rather than local server).

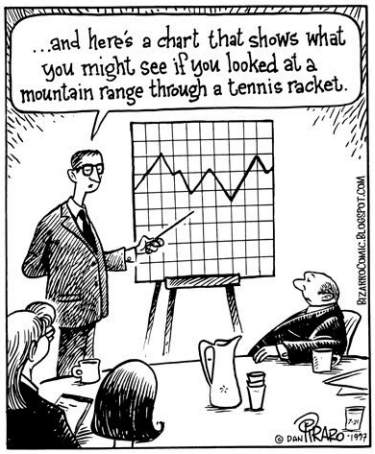


Figure 2.2: http://www.bizarrocomics.com/

https://aws.amazon.com to get an account.

http://www.louisaslett.com provides some amazon AMI for Rstudio and shiny server to make everything a bit easier.

3
Advanced shiny

Dynamic user interfaces

Elements of the user interface can also be created and rendered via server logic. This allows elements such as tabs in a tabset or navbar to be created or shown dependent on some reactive context.

Using the `shiny` package we can create any of the standard user interface elements contained within that package using `renderUI` and `uiOutput`.
For example we could change the type of input required like the code below:

```
library(shiny)

ui = fluidPage(fluidRow(
  column(4,wellPanel(radioButtons("dist", label = "Distribution",
                                  choices = c("Normal","Poisson")))),
  column(4,wellPanel(numericInput("n", label = "N points",
                                  value = 10, min = 1))),
  column(4,uiOutput("parchoice"))
  ),
  fluidRow(plotOutput("hist"))
)

server = function(input,output){
  output$parchoice = renderUI({
    switch(input$dist,
           Normal = wellPanel(numericInput("mean",
                              label = HTML("&mu;:"), value = 0),
                              numericInput("sd",
                              label = HTML("&sigma;:"), value = 1)),
           Poisson = wellPanel(numericInput("rate",
                            label = HTML("&lambda;:"), value = 10))
    )
  })
  dat = reactive({
    switch(input$dist,
           Normal = rnorm(input$n, input$mean, input$sd),
```

```
           Poisson = rpois(input$n, input$rate)
           )
  })
  output$hist = renderPlot({
    switch(input$dist,
           Normal = hist(dat()),
           Poisson = barplot(table(dat()))
    )
  })

}

runApp(list(ui = ui, server = server))
```

Here, both the inputs required and the outputs depend on what the chosen distribution is. This is a simple example of a reactive user interface, but in theory full interfaces could be created dependant on user input.

Some of the packages which extend shiny also allow the rendering of UI components. For example `shinydashboard` has the following functions

```
## [1] "renderMenu"         "dropdownMenuOutput" "renderInfoBox"
## [4] "renderValueBox"     "sidebarMenuOutput"  "infoBoxOutput"
## [7] "renderDropdownMenu" "valueBoxOutput"     "menuItemOutput"
```

Shinyjs

`shinyjs` is a package which allows you to run common useful JavaScript operations without having to know any JavaScript. If you do know Javascript you might also use `shinyjs` to call your own custom JavaScript functions from R.

```
devtools::install_github("daattali/shinyjs")
```

This package has some neat functionality such as `onevent`. This can be used to detect any valid JQuery event, such as keyboard presses or mouse clicks and run code accordingly. Other particularly neat functions are shown in table 3.1. To use the `shinyjs` functions you need to have a call to `useShinyjs()` within your UI.

The following url is a shiny app that demos some of the functionality of the package. The package also has a number of helpful vignettes.

https://daattali.com/shiny/shinyjs-demo/

Tags

If you are familiar with html, CSS and javascript you can use shiny tags to fully customise your shiny apps. `shiny::tags` is a list of 110 functions for creating html tags that match the html equivalents. For example `tags$h1("My header")` creates

Function:	Description:
`useShinyjs`	Necessary for running other `shinyjs` functions
`onevent`	run an R expression when an event on an element is triggered
`runcodeUI`	a construct that allows you to run R code live in a shiny app
`runcodeServer`	server side function needed for `runcodeUI`
`alert`	A handy wrapper function for sending browser popups

Table 3.1: Some useful `shinyjs` functions

```
<h1>My header</h1>
```

`tags$div(id='myDiv', class='simpleDiv','Here is a div with some attributes.')` gives

```
<div id="myDiv" class="simpleDiv">Here is a div with some attributes.</div>
```

We could then use these tags as though we were writing a webpage. Through them we can also use bespoke CSS and Javascript code using the `head`, `style` and `script` tags.

If you want to have full CSS and javascript source files outside of the R script create a directory called `www` within your app to house them. See the shiny documentation for further information. You can learn html, CSS and JavaScript at `http://www.w3schools.com/`

See `vignette("chapter6", package = "nclRshiny")` for more example code.