Wondering how something works? Shiny Apps will show you!

Irlanda Ayon-Moreno

11/1/2017

Have you ever sat in a Statistics class and listen to the pressor go on about a theorem that you just can't wrap your head around? I know this happens to me all the time! Sometimes I just can not believe these theorems to be accurate. However, sometimes it really takes seeing to believe it. Shiny Apps facilitates this process through reactivity! To demonstrate how Shiny Apps uses reactivity to help people visualize results, we will look at an example in which Shiny apps is used to demonstrate the Central Limit Theorem (CLT).

Reactivity is at the core of what makes Shiny Apps so special. Reactivity allows the user to make changes to the widgets and to see the results of CLT unfold almost instantly. Therefore a user can make specific changes to examine how a certain aspect affects the result. For example, a user of the CTL app may change the sample size to see how its size relative to the population affects the distribution of averages. The user may also change the number of times that a sample is drawn, or the size of the sampling population. Shiny Apps saves you the time you would spend attempting to figure out what multiple changes would result in, and simply gives you results!

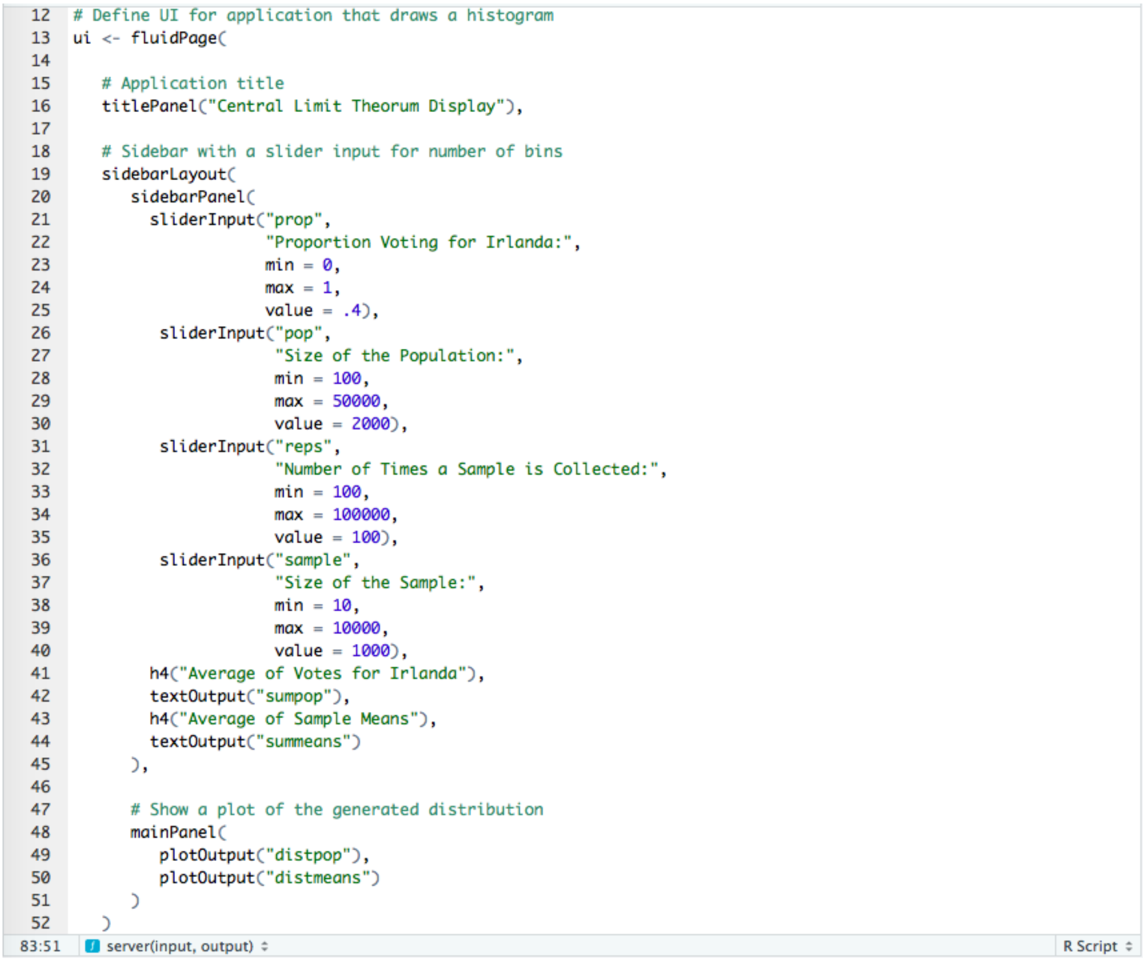
Using a Shiny App is great, but it's even better to be able to create your own, so that you can simulate situations that you want more information on. To do this, it is important to understand reactivity and objects in reactive programming. The three main objects to know are: reactive values (reactive sources), reactive observers (reactive endpoints), and reactive expressions (reactive conductors).

The most simple Shiny App will always have a reactive value and an observer. The reactive values are determined by the widgets that a user changes. The observers are what the user will see displayed on the window as the result of the change. You may think of these as inputs and outputs of regular R expressions. In R, when an input changes, the output does not change unless the expression is recomputed. The user of the app may be able to re-run the expression to account for the change in the input, but it is labor intensive. The server may also re-run all the code, but that might cause the program to slow down. As a solution to these problems, Shiny Apps lets the server know which expressions to run. It will not recompute all the code. Instead, it will only compute code for values that are out of date, due to one of its inputs changing.

When an observer uses a reactive value, the observer will pull the value of the reactive value. As it does this, it registers a "reactive context" which can be thought of as a letter, which carries a message. The message in the letter is called a "callback". In the letter, the observer is asking the reactive value to let it know when it changes. The callback is a command to re-run the observer. As a result, the reactive value holds on to these letters until it changes. Since the observer cannot re-run itself, the callback is sent to the server. This is how the server only re-runs code for values that are out of date. When the observer is updated, it leaves another context which restarts the cycle.

Now that we have explained how reactivity happens with reactive values and observers, we can talk about the middle man, reactive expressions. A Shiny App may not always contain reactive expressions, but they are useful for cutting down on the time it takes for the server to re-run code. Reactive expressions reduce the need for duplicating code and they are great when the code is computationally intensive and runs slowly. The reason reactive expressions are efficient is because they save their values, so if a value that was previously computed is needed again, no code needs to be run because the value was saved and ready to use. When there is a reactive expression involved, the observer ay call on it and then the reactive expression calls on the reactive value which receives the context and the cycle continues.

Now that we have covered how reactivity occurs in Shiny Apps, we can look at the code used to generate the CLT app and examine the three components mentioned.

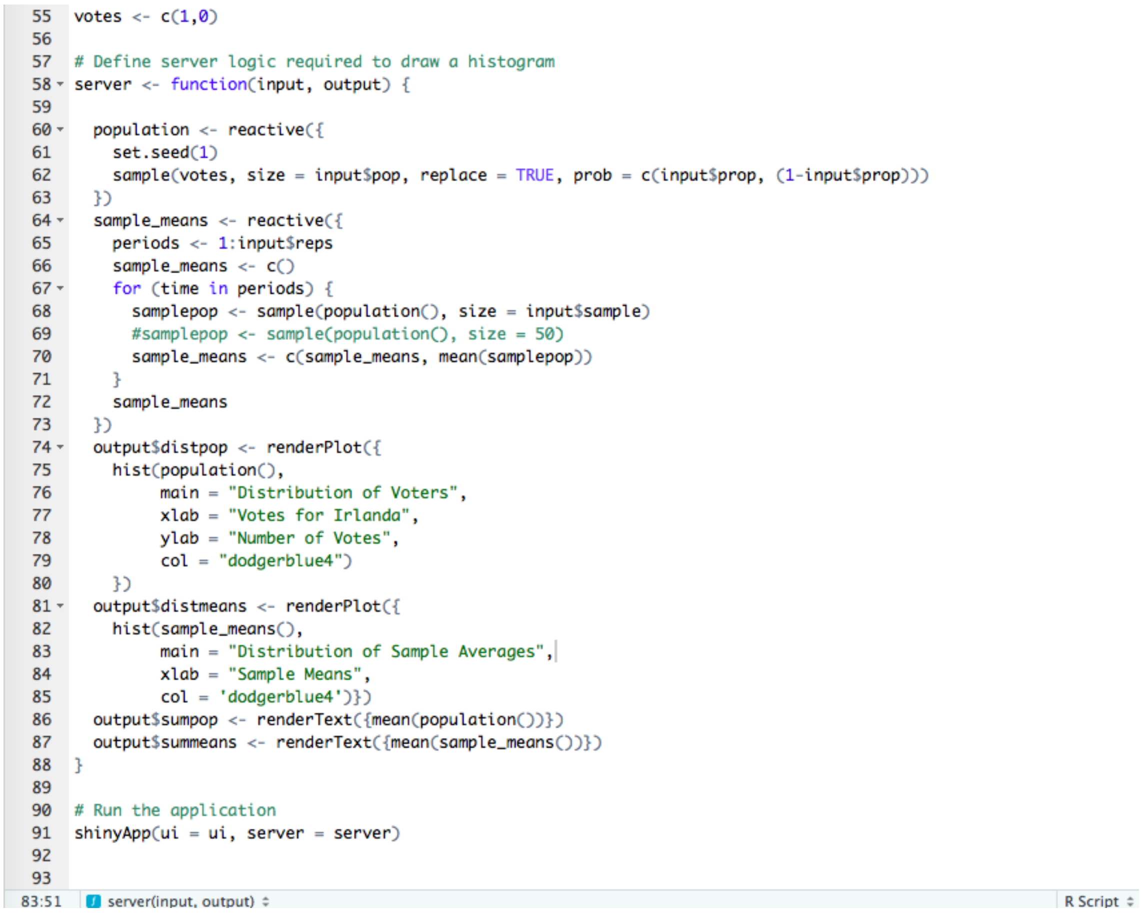


```r
# Define UI for application that draws a histogram
ui <- fluidPage(

   # Application title
   titlePanel("Central Limit Theorum Display"),

   # Sidebar with a slider input for number of bins
   sidebarLayout(
      sidebarPanel(
        sliderInput("prop",
                    "Proportion Voting for Irlanda:",
                    min = 0,
                    max = 1,
                    value = .4),
         sliderInput("pop",
                     "Size of the Population:",
                     min = 100,
                     max = 50000,
                     value = 2000),
         sliderInput("reps",
                     "Number of Times a Sample is Collected:",
                     min = 100,
                     max = 100000,
                     value = 100),
         sliderInput("sample",
                     "Size of the Sample:",
                     min = 10,
                     max = 10000,
                     value = 1000),
        h4("Average of Votes for Irlanda"),
        textOutput("sumpop"),
        h4("Average of Sample Means"),
        textOutput("summeans")
      ),

      # Show a plot of the generated distribution
      mainPanel(
         plotOutput("distpop"),
         plotOutput("distmeans")
      )
   )
```



```r
votes <- c(1,0)

# Define server logic required to draw a histogram
server <- function(input, output) {

  population <- reactive({
    set.seed(1)
    sample(votes, size = input$pop, replace = TRUE, prob = c(input$prop, (1-input$prop)))
  })
  sample_means <- reactive({
    periods <- 1:input$reps
    sample_means <- c()
    for (time in periods) {
      samplepop <- sample(population(), size = input$sample)
      #samplepop <- sample(population(), size = 50)
      sample_means <- c(sample_means, mean(samplepop))
    }
    sample_means
  })
  output$distpop <- renderPlot({
    hist(population(),
         main = "Distribution of Voters",
         xlab = "Votes for Irlanda",
         ylab = "Number of Votes",
         col = "dodgerblue4")
  })
  output$distmeans <- renderPlot({
    hist(sample_means(),
         main = "Distribution of Sample Averages",
         xlab = "Sample Means",
         col = 'dodgerblue4')})
  output$sumpop <- renderText({mean(population())})
  output$summeans <- renderText({mean(sample_means())})
}

# Run the application
shinyApp(ui = ui, server = server)
```

The reactive values are accessible through the input object with is like a list.Analogously, observers are accessible through the output object which is like a list. When creating a Shiny app, what one places inside the user interface (ui) will create the input object. It will also determine where the output is displayed. Then, the server creates the output object. In the server function, one must specify how to use the reactive values to build the object. The name given to inputs and outputs in the user interface will be the names used to create outputs. In this example, input$prop is a reactive value, mean(population()) is an observer, and population() is a reactive expression.

It is important to mention that reactive values may only exist in a reactive environment. That is the reason why R code is encapsulated by render functions which are reactive functions. Also, the function reactive() is used to create a reactive function that can hold reactive values. This is why reactive expressions are called with parenthesis, like functions. It is also important to mention that render functions are similar to other functions in that objects defined in its body cannot be called outside of the function. Therefore, if I had created population inside of renderPlot, I would not be able to use population in renderText. Using a reactive expression instead solves this problem.

Now that we have discussed the code, click on https://ayonmorenoi.shinyapps.io/post1/ to view the CLT in your web browser. You'll see that the simulation is successful. It shows the concepts of the CLT. One can see how the sampling population does not resemble a normal curve, but with sufficient samples and a large enough sample size, the distribution of means does in fact resemble a normal curve. Even more impressive, The average of the distribution of means in indeed the average of the population. The Central Limit Theorem is true! I hope that you can use this example and modify it to your needs so that you can show yourself the truth behind all your question.

Refrences

https://shiny.rstudio.com/articles/understanding-reactivity.html

https://shiny.rstudio.com/tutorial/written-tutorial/lesson4/

https://shiny.rstudio.com/articles/reactivity-overview.html

https://www.programiz.com/r-programming/function

https://shiny.rstudio.com/reference/shiny/1.0.5/