Post 2: Building Shiny App, a different perspective

Sudarshan Srirangapatanam

November 22, 2017

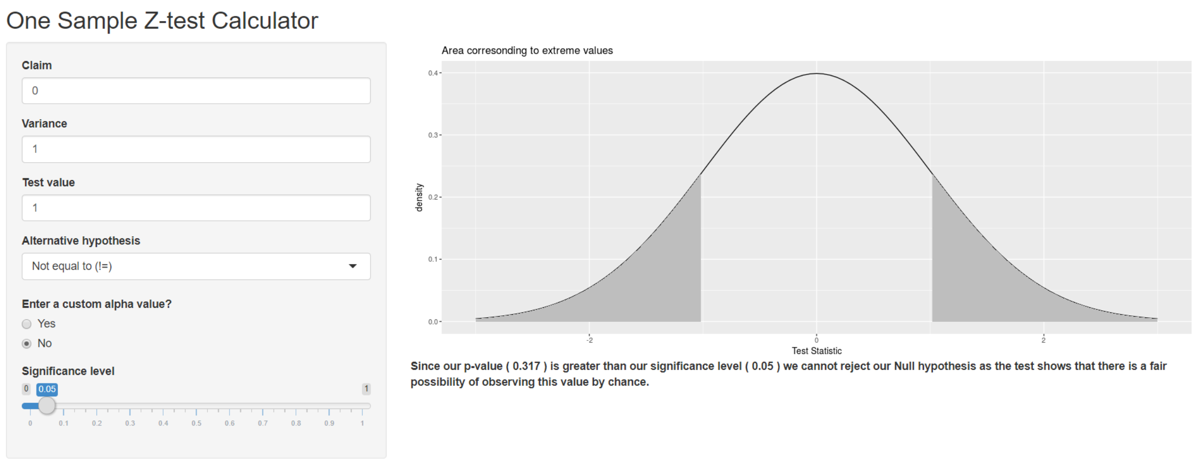
Introduction

Shiny apps are a way to take static content from R such as summary, plots, data tables and convert them to dynamic content. Shiny is a great tool for developers and analysts for their potential to automate many tedious tasks. For people just getting started with R and data analysis, shiny apps could be a fun project to stay in the game without being lost in lines of code and complex logic.

In this post, we will be building a shiny app from scratch, and we will be building it in terms of blocks.

We will **NOT** be copying any code from the popular Shiny Gallery. Instead, we will be focusing on the concepts that make shiny apps more extensible. We do not want to limit ourselves to the gallery page.

As a sneak peak, our final product will be One Sample Z-test Calculator I built for Post01.



One Sample Z-test Calculator

Claim: 0

Variance: 1

Test value: 1

Alternative hypothesis: Not equal to (!=)

Enter a custom alpha value? Yes [✓ No]

Significance level: 0.05

Since our p-value (0.317) is greater than our significance level (0.05) we cannot reject our Null hypothesis as the test shows that there is a fair possibility of observing this value by chance.

Here are the versions of R and packages I am using in the post, in case you need it.

- R: 3.4.2
- shiny: 1.0.5
- ggplot2: 2.2.1

Basic Shiny Components

Before we begin building the logic of the app we will have to understand the items sufficient to run a shiny app since we will be working in a bottom-up fashion.

```r
# Just follow this post from here on out and run any code you see in a code chunk (like this) in an R script file
```

Library

We need to tell R to load our tool `shiny`, so we do the following:

```r
library(shiny) # load shiny library

# if you need to install the package run:
install.packages("shiny")
```

Initialize App

If you run your file you'll realize nothing happened, but internally R now knows new functions it would not have known otherwise.

A Shiny app needs the following to run in the browser:

1. a `server()` function, and
2. `shinyApp()` function call

```r
# server function
server <- function(){

}

shinyApp(server = server) # shonyApp() call to run in the browser
```

Basic Structure

Now you have an app that is running in the browser but it doesn't necessarily display anything. This is because, All we have done so far is, given R a way to look for shiny functions, tell R to spin up a browser session and give it a server function to do the back-end logic.

To make the app somewhat more complete, we need to give a front-end to this app and this is done by a `ui` object.

```r
# ui object
ui <- "This is the Front-End"

shinyApp(server = server, ui = ui) # shonyApp() call to run in the browser
```

Full Code

With all of the above we have the following code:

```r
library(shiny)

ui <- "This is the Front-End"

server <- function(input, output){

}

shinyApp(ui, server)
```

Note: `shinyApp()` call is quite different here, this is just a more tidy version and all we are doing here is telling the browser to use `ui` as our front-end and `server()` as our back-end.

You'll also notice that the server function is using **input** and **output**, this is because the back-end logic requires a proxy to hold any inputs and outputs, which are appropriately named.

App Objects

With basic structure ready, we can start building our app. As noted earlier shiny is a tool used to convert static content to a dynamic one. We also know that our `ui` object holds all of the front-end, and `server()` function is responsible for all back-end work.

App layout

In a shiny app, we need a front-end layout and there are plenty of layout options available. In this post, we will be using `fluidPage()`. This also gives us access to `sidebarLayout()` which in turn gives us access to `sidebarPanel()` and `mainPanel()`.

As you may have guessed since this is the front-end layout we will be using this in `ui`, so our `ui` will now look like:

```r
# ui object with frontend layout
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

Outputs

A simple rule of thumb in shiny is that anything that has to do with calculations should go in the `server` function.

But `server` is a function and anything inside is local. How do we get around this?

Remember the **input** and **output** proxies inside our `server()` function??

We can use outputs from inside server to pass objects into `ui`. And use the output functions below to hold the output objects in our ui layout (like a placeholder).

There are many types of outputs available, and few of the most common ones are:

- htmlOutput()
- plotOutput()
- textOutput()
- verbatimTextOutput()

For a more comprehensive list of output functions available and for information regarding how to use them, see Shiny Reference and navigate to "Function reference" of your version, and scroll to "UI Outputs".

Here is the link to function reference for version 1.0.5

The output we will be using is `plotOutput()` like below. And we'll use name *plot* as our object name

```
mainPanel(
  plotOutput("plot") # Output for plot
)
```

Note: the outputs can go in the sidebarPanel and the app would work just fine, but we decided to put them in mainPanel.

Inputs

Inputs are cornerstones of shiny apps because all users interact with these and nothing else.

All inputs are basically front-end objects that store their values inside the **input** object.

Like outputs, we also have many types of the inputs available, following are just a few of the inputs:

- selectInput()
- dateRangeInput()
- sliderInput()
- radioButtons()
- numericInput()

For a more comprehensive list of input functions available and for information regarding how to use them, see Shiny Reference and navigate to "Function reference" of your version, and scroll to "UI Inputs".

Here is the link to function reference for version 1.0.5

```
sidebarPanel(
  # All inputs go here
)
```

Note: as before the inputs could go in the mainPanel and the app would work just fine, but we decided to put them in sidebarPanel.

Static plots

We will be using `ggplot2` from tidyverse for our plot so let's go ahead and add this library to our file:

```
library(ggplot2) # load ggplot2 library

# if you need to install the package run:
install.packages("ggplo2")
```

From here will want to generate a density plot using normal distribution and to do that we need to add the code for the plot. Below is the code, I will not be discussing `ggplot2` since it is not within the scope of this post.

```
dat <- data.frame(range = c(-3, 3)) # create data frame with x-axis range

# density value function
density.fxn <- function(x) {
  dnorm(x, mean = 0, sd = 1)
}

# shading function
fill.fxn <- function(x) {
  d_value <- density.fxn(x)
  d_value[x < 0] <- NA

  return(d_value)
}

# plot object
```

All we are doing here is:

1. Creating a data frame with our x-axis range, because ggplot2 only takes data frames as data input
2. Creating a density function so that we can draw a density curve
3. Creating a fill function to tell ggplot to shade only certain values
4. Creating ggplot object to create our plot

But where do we add it?

As you can see, anything that has to do with calculation should go in `server` and `ui` should only have outputs, so we can add the above inside our server function. And to mark the plot as a special output (because it is), we need to wrap our plot object in `renderPlot({})`

Full Code

Our code now looks like this:

```r
library(shiny)
library(ggplot2)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
    # All inputs go here
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)

server <- function(input, output){
  dat <- data.frame(range = c(-3, 3))
```

Dynamic Objects

We wouldn't be building shiny apps if it was not for the dynamic aspect of shiny, so let's learn how to make our app dynamic.

User Interaction

Right now, our app doesn't have a way for users to interact, and to make our app interact-able we have to use inputs from before.

For our app we need the following inputs:

1. Claim
2. Variance
3. Test value
4. Alternative hypothesis
5. Enter a custom alpha value?
6. Significance level (numeric)
7. Significance level (slider)

We already know where to add them and also know where to find how to use them (Function Reference). Let's add them to our app.

Note: In most cases, the first argument is the name of the input object, accessible at input$NAME. The second is the title, and the third (and possibly fourth) argument is specific to input UI. You'll also have access to one other argument to set default values.

Claim

```r
numericInput(
  "null",
  "Claim",
  value = 0
)
```

Variance

```r
numericInput(
  "population_var",
  "Variance",
  min = 0,
  value = 1,
  step = .01
)
```

Test value

```r
numericInput(
  "alternative",
  "Test value",
  value = 2
)
```

Alternative hypothesis

```r
selectInput(
  "sidedness",
  "Alternative hypothesis",
  c("Not equal to (!=)", "Less than or equal to (<=)", "Greater than or equal to (>=)")
)
```

Enter a custom alpha value?

```r
radioButtons(
  "manual_alpha",
  "Enter a custom alpha value?",
  c("Yes", "No"),
  selected = "No"
)
```

Significance level (numeric)

```r
sliderInput(
  "alpha_slider",
  "Significance level",
  min = 0,
  max = 1,
  value = .05,
  step = .01
)
```

Significance level (slider)

```r
numericInput(
  "alpha_numeric",
  "Significance level",
  min = 0,
  max = 1,
  value = .05,
  step = .001
)
```

Using Inputs

As noted earlier, our inputs are accessible using input$NAME inside our `server` function. But to be able to dynamically update calculations we have to understand the term reactive.

In shiny, all of the UI outputs are marked special. Like our `renderPlot()` function. This is because we need to tell the server to update these objects every time a user interacts with an object, otherwise, our app wouldn't be dynamic.

All of the render functions in Function Reference are reactive, and if you need to create your own objects that are reactive, simply use `reactive({})` wrapper. And to use the object, you have to use it like a function i.e. `OBJECTNAME()`.

Here is the link to function reference for version 1.0.5

To make our plot dynamic, all we have to do is replace static content with `input$NAME`.

Note: we can only use our inputs inside a reactive context i.e. UI output functions or `reactive()`.

```r
# Old
dat <- data.frame(range = c(-3, 3))
# New
dat <- reactive({
  data.frame(range = c(
  input$null - 3*sqrt(input$population_var),
  input$null + 3*sqrt(input$population_var)
  ))
})
```

```r
# Old
density.fxn <- function(x) {
  dnorm(x, mean = 0, sd = 1)
}
# New
density.fxn <- reactive({
  density.fxn <- function(x) {
   dnorm(x, mean = input$null, sd = sqrt(input$population_var))
  }

  return(density.fxn)
})
```

```r
# Old
fill.fxn <- function(x) {
  d_value <- density.fxn(x)
  d_value[x < 0] <- NA

  return(d_value)
}
# New
fill.fxn <- reactive({
  fill.fxn <- function(x) {
    d_value <- density.fxn()(x)
    d_value[x < input$alternative] <- NA

    return(d_value)
  }
```

```r
# Old
output$plot <- renderPlot({
  ggplot(dat, aes(x = range))+
  labs(x = "Test Statistic", y = "density", title = "Area corresonding to extreme values")+
  stat_function(fun = density.fxn)+
  stat_function(fun = fill.fxn, geom = "area", fill = "grey")
})
# New
output$plot <- renderPlot({
  ggplot(dat(), aes(x = range))+
  labs(x = "Test Statistic", y = "density", title = "Area corresonding to extreme values")+
  stat_function(fun = density.fxn())+
  stat_function(fun = fill.fxn(), geom = "area", fill = "grey")
})
```

Dynamic UI Components

One of many cool things about shiny is that the front-end can also be dynamic. We can have certain inputs controlling other inputs. Or a more advanced version is one ui element controlling another ui element.

Right now in our app, we want to only show one significance level input based on the answer to a previous question. Let's see how to do this the advanced way, which is also more robust.

Two main functions we'll be using to achieve this is `renderUI()` and `uiOutput()`. As the names suggest, `renderUI` goes inside `server` and `uiOutput` goes inside `ui`.

To do this we'll have to modify our code a little in the `ui` as well as `server`. We'll strip some ui input functions and replace them with one placeholder ui like this:

```r
# Old
sliderInput(
  "alpha_slider",
  "Significance level",
  min = 0,
  max = 1,
  value = .05,
  step = .01
),
numericInput(
  "alpha_numeric",
  "Significance level",
  min = 0,
  max = 1,
  value = .05,
  step = .001
```

and inside `server` we'll add this:

```r
# Check answer to the question before and conditionally output a UI
output$alpha_display <- renderUI({
  if (input$manual_alpha == "Yes") {
    numericInput(
      "alpha_numeric",
      "Significance level",
      min = 0,
      max = 1,
      value = .05,
      step = .001
    )
  } else {
    sliderInput(
      "alpha_slider",
      "Significance level",
      min = 0,
```

As I said earlier, the logic here is simple. We replace our original inputs with a placeholder and treat the placeholder as output. Then we do some logical calculation in server to display only one of the inputs to the placeholder.

The same effect can be achieved using `conditionalPanel()`, see Conditional Panel in shiny reference for function reference.

Here is the link to conditionalPanel() for version 1.0.5

Example using `conditionalPanel()`:

```
# Old
sliderInput(
  "alpha_slider",
  "Significance level",
  min = 0,
  max = 1,
  value = .05,
  step = .01
),
numericInput(
  "alpha_numeric",
  "Significance level",
  min = 0,
  max = 1,
  value = .05,
  step = .001
```

Note: This involves calculations in `ui` object, which is not ideal since all calculations are better off when reserved for `server` function.

Full Code

Our code so far looks like this:

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      numericInput(
        "null",
        "Claim",
        value = 0
      ),
      numericInput(
        "population_var",
        "Variance",
        min = 0,
        value = 1,
```

Note: The above code uses a different `fill.fxn()` and this is because of the functionality of shading that is required based on the z-tests, see Post01 for more details regarding z-tests and hypothesis test paradigm.

More Practice

If you have come this far, kudos to you!

You are now a pro at building shiny apps and with the bottom-up approach above, you can take on any shiny app project.

If you look at the app you created using the code chunks above and the final product, there are two things missing:

1. Big title at the top of the page
2. Small result text below the graph

Questions

If you can answer these questions regarding the missing objects, then I am 100% confident that you can build them on your own.

1. Where in the app layout should they go?
2. Are they dynamic?
3. What shiny wrapper/function should they use?
4. Do they need any server logic, if they do what is the shiny wrapper/function?

Answers

1. Where in the app layout should they go?
    1. Title should go before `sidebarLayout()`
    2. Text should go inside `mainPanel()` after `renderPlot()`
2. Are they dynamic?
    1. No
    2. Yes
3. What shiny wrapper/function should they use?
    1. titlePanel()
    2. textOutput()
4. Do they need any server logic, if they do what is the shiny wrapper/function?
    1. No logic needed
    2. Needs a logic inside `renderText({})` or any other related function

Full Code

Here is the full code for the app, including the above logic.

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  titlePanel("One Sample Z-test Calculator"),
  sidebarLayout(
    sidebarPanel(
      numericInput(
        "null",
        "Claim",
        value = 0
      ),
      numericInput(
        "population_var",
        "Variance",
        min = 0,
```

Note: Major changes lie in the `server` after plot output. For the general paradigm of hypothesis tests and the logic behind the text, see Post01.

Take home message

As you can see above, the app you just built is no simple task. And this app has lots of moving parts, but you were able to do it with a simple bottom-up approach.

Yes, you could have built the same app by using the code chunks from Shiny Gallery and I do encourage you to take a look at the site if you haven't already. But remember, it should be your source of inspiration, not limitation.

The approach laid out above helps you understand every single aspect of shiny apps and how the `shiny` package itself works.

If you think about it, the first code chunk has nothing but an empty `server` function and an empty `ui` object. And if you take the same approach You can successfully build any shiny app, even the complex ones like Movie explorer

References

1. Post 01 - Intro to hypothesis tests and One Sample Z-tests
2. One Sample Z-test Calculator
3. Shiny Gallery
4. Movie Explorer
5. ggplot2 Reference
6. shiny Reference
7. Shiny Function Reference v. 1.0.5
8. Conditional Panel Reference v. 1.0.5