# How to create a game like "Is It Normal?"

This game was created in RShiny. It is a very simple statistics game and it is intended to teach players a little bit about how sample size is used to determine whether something is or is not a normal distribution.

RShiny is a language and app deployment software that is available for free and uses the statistics computing language R. It is often used for things like visualizations and dashboards, as it can create a reactive interface, which makes it functional for game design of a relatively limited scope.

This toolkit and walk-through will help show you how to create an RShiny game of your own. In this toolkit I do assume that you have use R before, though I do provide installation instructions just because I know people forget if they haven't used in a while!

## Part 1 – install the software

In order to build an RShiny app on your computer you will need to first download and install:

* the computer language R (<https://www.r-project.org/>)
* the software RStudio (<https://posit.co/download/rstudio-desktop/>)

once these are installed, open RStudio. In the console (the thing in the bottom left corner) you will type the following code or copy paste it and then hit enter. This code will install all of the packages you need which are sort of like expansion packs for R.

install.packages("shiny")

install.packages("beepr")

install.packages("tuneR")

install.packages("ggplot2")

install.packages("shinyjs")

## Part 2 – open up the app

you can download the "is it normal" game from the location you presumably got this walk-through. This app is made of three parts:

* a file called server.R, which contains information on how to do all of the necessary game calculations in the background
* a file called UI.R, which contains information on how to display the game to the player including things like buttons and graphics
* a folder called www, which contains a number of audio files that are played when the player gets something wrong or right.

The first two files are things that you can open up in RStudio. So you should do that!

## Part 3 – Details in the UI

The UI file tells the computer what should be displayed. Inside are some functions that you should be aware of, as these are common features you might change if you wanted to edit this game. Anytime you see something that is green and has a #in front of it, that is a note and not code – sometimes these are meant to be reminders for myself, sometimes they are turning off chunks of code that in the end didn't work.

At the very top of the UI script you will see some library codes. R doesn't automatically load these "expansion packs" every time you run it - otherwise you would end up with way too many of them. The library code reminds R which ones you are working with. If you remove these, the game won't work. You will also see these in the server file.

Inside you can see some functions called things like titlePanel() and sidebarPanel(). These are instructions on how many columns there should be. Things are in the title panel go all the way across the top of the screen, things that are in the sidebar panel are restricted to certain columns.

You can also see a few "output" and "input" options. Output would be something where it is calculated in the server area and then displayed here. Input is something that the player gets to choose. So for example this script has both a radio button and an action button to submit the answer or to create a new plot if the player doesn't want to answer for that particular plot. These are both inputs.

Below is a table of the inputs and output codes from the UI script.

|  |  |  |
| --- | --- | --- |
| **Code** | **Input/output** | **What it does** |
| textOutput("PointsMaybe") | Output | Tells the player how many points they have, which is calculated in the server. It is of higher than the other chunks of output code because it is displayed in the main panel at the top and not in the side panel. |
| radioButtons("pvalguess", label = h3("Is it from a normal distribution?"),  choices = list("Probably" = 1, "Unlikely" = 2),   selected = 1) | Input | This function creates the radio button's which allow the player to choose either probably or unlikely. It specifies to the server that this value will be saved with the name **pvalguess,** and will have either a 1 for probably or 2 for unlikely. With radio buttons you always have to have one of them automatically selected, so the last chunk of code tells you that it starts with 1 or probably selected. There is also a chunk of code in here to set a label, this is just text that is displayed to the player. |
| actionButton("guess", "Submit Answer") | Input | I found that radio buttons by themselves didn't work very well with some of the reactive code in the server. So I added a submit button, so that once the player had made their choice they had to hit submit. This is really just so that everything actually worked. In the server, it now runs when this button is clicked and it refers to it as **guess.** |
| actionButton("new", "New Plot") | Input | I also wanted to give players the option to make a new plot if they thought that one was too difficult for them. This action button creates a new plot without changing the point total. It is referred to as **new** in the server code. |
| textOutput("Sass") | output | this outputs the response depending on whether the player was correct or not. That is referred to as **Sass** in the server code. If the player has guessed correctly then they will get a more positive response than if they have not. I decided to add the sass in as an additional extrinsic reward system. |
| plotOutput("distPlot") | Output | This is the function that outputs the plot that everyone is looking at. It also says that this is called **distPlot** in the server code. You have to have a distinct name for these things that matches between both the UI and the server. |
| textOutput("Answer") | Output | This text output provides the answer text that is underneath the plot, which goes something like "correct, this P value was .98 so is probably from a normal distribution as far as we can tell." |

## Part 4 – Details of the Server

The server function is the part of this app that calculates what the distribution looks like, calculates whether or not the answer is correct, provides some sass and point response, and tells the UI what to display. It uses the same names for inputs and outputs as were specified in the UI. This is especially pertinent down in the reactive section (it has a lot of hashtags and the word REACTIVES to make it clear where it is). For example, in observeEvent you can see something that says input$new. Remember that new was the name of the button that you click for a new plot – so this says that when someone clicks that button all of this stuff happens.

Similarly below that you can see something called output$distPlot. That name matches the plot output function in the UI, so this is the section that creates that plot.

There are a number of custom functions in here – meaning, functions that I have made. These just make it significantly easier to rerun the same code. The following sections have some detail on those.

## Part 5 – calculating distributions

The very first custom function is at the top and it looks like this:

compute\_data <- function(){

samplesize <- sample(c(25:175), size = 1)

rnorm0 <- rnorm(n = samplesize, mean = 60, sd = 3)

lskew <- 69\*rbeta(samplesize,25,3)

rskew <- 10+69\*rbeta(samplesize,3,8)

bimod <- c(rnorm(n = samplesize/2, mean = 55, sd = 3), rnorm(n = samplesize/2, mean = 65, sd = 3))

unlist(sample(list(rnorm0, lskew, rskew, bimod), 1)) #randomly take one of these samples

}

When this function is used later on in the script, it re-computes a whole bunch of different types of distributions. So in the first line it randomly takes a number between 25 and 175. That random number is used as the sample size in four different functions that create distributions of data. The first makes a normal distribution with an average of 60, a standard deviation of 3, and whatever sample size was specified. The next to create skewed distributions, either to the right or to the left. The fourth one creates a bimodal distribution by combining two different normal distributions.

The last chunk of code here randomly takes only one of these distributions. I found it more effective to have them calculated all at the same time and then just randomly take one. So even though every time you press the button it calculates four distributions, the player will only ever see one.

## Part 6 – making the plot look pretty (myplot and newcolors)

the next custom function is called **myplot.** It uses code from the package ggplot2 to create the actual plot. It makes a histogram and it colors it according to the function **newcolors.** This function just randomly pulls one of those colors in that list, I just wanted to be able to present something with a different color.

You'll notice that both of these are specifying the function, they're not actually running it in the top part of the script. That happens down below.

## Part 7 – score calculator and sass

the final two functions before you get into the section where the code actually does something (up until now the code has just been making new functions that we can use later) are the sassafras and pstuff functions. The first randomly pulls either a good or a bad chunk of sass from the enormous list of created up above. This is the feedback the player gets depending on if they were correct or not. The piece of code calculates whether or not their guess was correct. It does a Shapiro test on whatever is the data, which test for normality, and then basically says whether or not the input for the radio button should be 1 or 2 (which corresponds to probably or unlikely on the radio buttons).

## Part 8 – Reactives

when I initially began creating this game I ran into some problems because there is a multistep process in the game itself. Essentially, gameplay goes like this:

the computer creates a plot which might be a normal distribution or not

the computer determines if that is a normal distribution or not using a Shapiro test (pstuff) and figures out what the answer should be for the player to give it (probably or unlikely)

* The computer displays that plot to the player
* the player selects a radio button and hit submit or asks for a new plot
* if the player has selected correctly, the computer adds a point, picks a sound, and chooses the sass.
* If the player has selected incorrectly, the computer subtracts a point, picks a sound, and chooses the sass
* the computer then displays the correct point total, plays the actual sound, and displays the actual sass

When I initially created this game I ran into an issue where the computer would calculate things correctly, but would not display the correct point total, play the sound, or display the sass until the player had asked for a new plot. This is why I ended up using reactive values and the code observeEvent(), as these helped to dictate when these things should happen. If you go and look up RShiny there are other options you can use but these are the ones you should use if you need to very what is happening on the user interface (e.g., display a different point total).

All of these functions use what is known as reactive programming. I don't fully understand what that means except that it does mean things get calculated at the right time!

The reactiveValues() function was especially important because it created values that I could refer to outside of the observed event code. I found that if I was using observeEvent and I tried to calculate things like the P value outside of that function, I was running into issues. This function fixed that.

This is where some of the functions from above are actually being run. So for example, you see something like: **rnorm1 <- reactiveValues(x = compute\_data())** which is where it is actually using that function to create and select the distribution for that plot. Again, this was the solution to display the point at the correct time and to use the observeEvent functions.