Break This App: an Ecologist's Intro to Shiny Part 1

Lyndsie S. Wszola

4/17/2020

Break this app: an ecologist's intro to Shiny

Hello and welcome to my first Shiny tutorial! Shiny is a framework for building interactive webpages using the R programming language. That means you can build powerful, customizable online tools to engage stakeholders in research communication using tools and analyses you've already developed. In this tutorial, we'll learn how to build a Shiny app that shows temporal trends in Stellar sealion pup counts at a user-specified count site over time. This tutorial is meant to be an absolute basic intro to Shiny, so if you're already making Shiny apps it might be a bit too basic for you. If you're getting ready to build your first app, you're in the right place!

What do I need to make a Shiny app?

To get started, you'll need to have R and RStudio installed on your computer. The main coding prerequisite to building our sealion Shiny app is a basic understanding of the Tidyverse, a collection of R packages for data science. If you're new to the Tidyverse, I highly recommend the book R for Data Science, as well as the unofficial solutions manual. If you haven't used Tidyverse before, no worries! We'll go over each function we use in detail.

Ready to get started? Awesome! Let's open RStudio and make sure our required packages are installed if you don't have them already:

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

Getting started

To start a new shiny app, we would make a new directory, then click File>New File>Shiny Web App. Since we already have an app to work with, we'll instead download or clone this repo if you haven't already. We'll follow along with the "app.R file" in the app folder. When you open app.R, you'll see some code at the top that sets up our workspace. We put these lines outside the main app code because that way they're run and loaded into the environment just once, which reduces memory needs and run time. The first lines load our required package libraries, shiny and tidyverse:

```r
library(shiny)
library(tidyverse)
```

In the next lines, we set up a custom ggplot theme, which we'll call "simpleTheme" so we can easily control the way our plot looks later:

```
simpleTheme <- theme_bw()+
    theme(panel.grid = element_blank(),
          axis.text = element_text(size = 20,
                       color = "black"),
          axis.title = element_text(size = 24,
                        color = "black"))
```

Now we're ready to load our data! The purpose of the app is to show trends in pup count data at different sites, so we need a list of sites. The following line loads an R data object containing a named list of the different count sites.

```
siteChoices <- read_rds("data/siteChoices.rds")
```

A note: sometimes directories can be a little tricky. You'll notice that the lines above use the path "data/" because a published Shiny app needs all the data to be in a folder called "data," which it automatically looks for in the app directory. However, if you download and modify this app, sometimes your paths may end up a little different. When you're running the app locally on your computer, it may be easier to use the commented out lines, which will help the app find the right paths to run on your computer:

```
# appPath <- dirname(rstudioapi::getSourceEditorContext()$path)
# siteChoices <- read_rds(paste0(appPath, "/data/siteChoices.rds"))
```

Basic app structure

Shiny apps need fundamental elements: the user interface and server. The user interface (ui), as the name suggests, creates all the elements that the user engages with. It contains all the static text and defines the appearence of interactive elements. We define the user interface using the `shinyUI` function:

```
ui <- shinyUI()
```

The second fundamental element, the server, does all the background work to run the app. The "magic" of a Shiny app is reactivity, the feedback between server and ui: the user manipulates the interactive elements in the ui, then the server process userinput and returns dynamic data and graphics elements. We define the server using the `shinyServer()` function:

```
server <- shinyServer(function(input, output, session){

})
```

Shiny assembles the server and ui into a cohesive unit using the `shinyApp()` function:

```
shinyApp(ui = ui, server = server)
```

Now that we have a handle on the basic structure of the app, let's dive into our sealion visualization!

UI

The first thing we need to decide for our new app's ui is how we want the text and visualization to be organized on the pages. Remember that Shiny apps make websites, so we have to plan out both the overall structure of the app's pages and how they relate, as well as the layout and organization of each page. The organization of the user interface code is consequently hierarchical, with each page's elements nested in a page layout, which is in turn nested in the whole app's layout.

We'll start out with an app layout that creates clickable tabs across the top, called a "navbarPage." I like navbarPage because it looks clean and professional, but there are lots of other options. we define this structure using the `navbarPage()` function. We'll define the title of our navbarPage, which will show up in the top left, by typing it in quotes: "Stellar Sealion Pup Count Trends." So we've updated our empty shinyUI function from above to read:

```r
ui <- shinyUI(navbarPage("Stellar Sealion Pup Count Trends"))
```

Now we need to choose the layout of our first page, which we initiate using the `tabPanel()` function and assign it a title that will show up in the navbar at the top of the page. We're going to use a sidebarLayout layout, which creates a shaded a side panel and a main panel. The side panel and main panel can both hold text, plots, maps, user input widgets, videos, and whatever else we need them to hold. The advantage of using a sidebarPanel() is that it draws the user's eye, so it's a great structure for communicating instructions and introductory material. Our sidepanel will hold a brief welcome for the user, made large using the `h2()` function, and a user input widget that prompts users to select a pup count site from a dropdown menu.

We define the dropdown menu using the `selectInput()` function, which allows users to select one option. We give the drop-down widget a name, "site," which will pass to the server, and give it some human-friendly text to display to the reader. We then specify the choices to display to the reader, which again include an index for our code and a vector of human readable names to display to users. The choices are the siteChoices list we loaded above, and we will set the default selection to 1, the first index, which is a summary of all the sites.

In addition, we'll add a main panel, using the `mainPanel()` function, to the right of our sidebarPanel to hold our dynamically updating graph of pup trends. Inside the mainPanel, we just have our sealion plot, which we print using the `plotOutput()` function and the name of the plot, "trendPlot" (more on that later). So altogether, our user interface now looks like:

```r
ui <- shinyUI(navbarPage("Stellar Sealion Pup Count trends", # page title
              tabPanel("Graphing pup counts", # Tab panel title
               sidebarLayout( # give us a nice data selection side bar!
                sidebarPanel(
                 h2("Welcome to the Stellar Sea Lion vizualization app! "),
                 # Some nice welcoming text for users
                 br(), br(),

                 selectInput("site", # Make a drop-down menu
                  "Choose a site to display the
                  time series of mean pup counts at that site.",
                 # data selection prompt for the users
                  choices = siteChoices,
                 # referencing the siteChoices list we loaded above
                  selected = 1)
                 # Make a default selection, the first item in the list, "All" sites
                   ),

                 mainPanel( # The big panel next to the side panel

                 plotOutput("trendPlot") # Print the plot we make below.

                  )))
                ))
```

The data we're using in our app comes from NOAA Fisheries, which means that we need to cite it properly. We'll build a "Citations" page using a tabPanel as we did for the "Graphing pup counts" tab. The citations page will contain a wellPanel, a shaded offset panel, telling users who collected the data (NOAA fisheries) and who made the app (me), and provide guidance on appropriate use. The wellPanel will also link users to the data source using a hyperlink written in html and a string that describes what to show the user the wellPanel. This ability to mix syntaxes and borrow elements from html and other languages is one of the coolest things about shiny. With the addition of our citation page, we now have our completed user interface!

```
ui <- shinyUI(navbarPage("Stellar Sealion Pup Count trends", # page title
               tabPanel("Graphing pup counts", # Tab panel title
                sidebarLayout( # give us a nice data selection side bar!
                 sidebarPanel(
                  h2("Welcome to the Stellar Sea Lion vizualizationapp!"),
                  # Some nice welcoming text for users
                  br(), br(),

                  selectInput("site", # Make a drop-down menu
                   "Choose a site to display the time
                    series of mean pup counts at that site.",
                    # data selection prompt for the users
                    choices = siteChoices,
                    # referencing the siteChoices list we loaded above
                    selected = 1)
                    # Make a default selection, the first item in the list, "All" sites
                    ),

                   mainPanel( # The big panel next to the side panel

                   plotOutput("trendPlot") # Print the plot we make below.

                   ))),
               tabPanel("Citations",
                # Must cite the data collectors! In a nice new tab panel!

                wellPanel("This app was created by Lyndsie S. Wszola
                           at the University of Nebraska for instructional
                           purposes only. The Stellar sealion pupcount
                           dataset was collected by The National Marine
                           Fisheries Service. More information on
                           the original dataset can be found",


                           tags$a(href="https://www.fisheries.noaa.gov/resource/data/counts-alaska
                           -steller-sea-lion-pups-conducted-rookeries-alaska-1961-2015", "here. "),

                          "The sealion pupcount data has been slightly
                           summarized and aggregated for this app. The
                           data visualizaions contained in the app are
                           therefore purely for educational purposes
                           and do not represent any
                           official finding or policy.")
                           # We must have a disclaimer so that people
                           #don't misuse/misinterpret anything here.
                )

                ))
```

Server

Now that we have our user interface set up, we need some cool data products for it to display! We'll start out by loading the sealion pup count data as we did for the named list of sites above (the note about file paths applies here too).

```
server <- shinyServer(function(input, output, session) {

            # seaLions <- read_rds(paste0(appPath, "/data/seaLions.rdata"))

            seaLions <- read_rds("data/seaLions.rds")

        })
```

Now we need to build the links between the user interface and the data wrangling and plotting functions we're about to describe. If you've ever built functions that depend on other functions, the structure of a Shiny server is very similar. If you haven't, no worries! We'll walk through it step by step.

Reactivity

Reactivity is the ability of Shiny apps to receive user inputs and perform operations that are then returned to the user as updating visuals and values. We pass user input from the user interface to the server using reactive values. It's generally good practice to keep our reactive values simple and discrete, so that when (not if) our code breaks, we will have an easier time identifying the problem.

our first reactive value, siteSelection, will extract the name of the site specified by the user in the drop-down menu, which is why we used a named list.

```
            siteSelection <- reactive({

                site <- names(siteChoices)[as.numeric(input$site)]
                # The user's selection from the drop-down

                })
```

Once we user has selected a site, we need to filter our data to include only observations from the site the user selected. We will accomplish our data filtering using our siteSelection() variable from above (notice how it's written like a function) and the `filter()` function from the dplyr package. We start our dplyr statement by specifing the dataframe we want to use, seaLions. We then add a pipe symbol (%>%), which basically tells r "take everything that has happened up to this point in this code block, and continue it down to the next line." we then filter our dataframe to include only those observations whose "site" variable matches exactly the "site" reactive variable defined above.

Notice that we use `dplyr::filter()` with the ::, which indicates that we want to use the `filter()` function found in the dplyr package. I did this because it's a good habit to specify packages in large coding projects you might share with other people, but also because we might want to add some spatial components in a future tutorial. Several spatial packages also have a `filter()` function, so specifying the package now saves us some future headaches.

```
            data <- reactive({ # Subsetting the data based on user input
                    site <- siteSelection()
                    # referencing site-selection from above

                    siteDat <- seaLions %>%
                               # Keep only data from the user-specified site.
                               dplyr::filter(sitename == site)

            })
```

Plotting

Now that we have our data reactive values set up, it's time to make our graph! We bring our user-specified filter data into the scope of the plot using the `data()` reactive value and assign it to an object called "siteDat." We will create a line plot with points on top of the lines using the `renderPlot({})` function, which we will designate as an output object by calling it `output$trendPlot`. If you refer back to the UI code above, you will notice that the mainPanel on the "Graphing pup counts" page displays this graph by calling `plotOutput(trendPlot)`.

We will build the pup count trend plot using the `ggplot()` function from the ggplot2 package, which we loaded as part of the tidyverse along with dplyr. ggplot graphics are built in layers, so each line will visually appear on top of the one before it. That means that each time we add a new graphics argument, it will inherit data and aestheteics (the x and y specification) unless we choose to overwrite it.

We start our pupcount trend plot with a line graph using the `geom_line()` function that creates a continuous line whose x coordinates are the years, specified as numbers just in chase, and whose y coordinates are the mean pups counted in each year. We will then add points on top of the line using the `geom_point()` function, with the same x and y coordinates. The default point size is just a little smaller than I want it to be, so we'll make it bigger by specifying size = 3. Next, we specify the x and y limits of the plot. The `coord_cartesian()` function allows us to specify x and or y limits. We'll set the y axis to adjust with the data. By specifying htat `ylim = c(0, max(siteDat$meanPups + 5))`, we tell ggplot to limit the y axis to between 0 and 5 pups above our maximum number of mean observed pups. Next, we set the limits of our x axis. We'll keep the years constant so the user can see the different time spans of data collection. We specify the limits and breaks, where the axis ticks should go, `using the scale_x_continuous()` function. Finally, we add some axis labels usin the `labs()` function and add the theme we set up at the beginning to make the graph layout nice and simple.

```
output$trendPlot <- renderPlot({

siteDat <- data() # Source our data from above

ggplot(siteDat) + # Make a beautiful (simple!) plot
      geom_line(aes(x = as.numeric(year),
                    y = meanPups)) +
      geom_point(aes(x = as.numeric(year),
                     y = meanPups), size = 3) +
      coord_cartesian(ylim = c(0, max(siteDat$meanPups + 5))) +
      scale_x_continuous(limits = c(1960, 2015),
                        breaks = seq(1960, 2020, 10)) +
      labs(x = "Year", y = "Mean pups per observation") +
      simpleTheme

            })
```

At this point, we should have a fully functional Shiny app that shows sealion pup count data at a site of the user's choosing. We can and will add more later, but it's a good start. We can run our new app using the "Run App" button that should have appeared in the upper right hand corner of your RStudio window. It will launch the app in a browser window.

Other Shiny resources

The Shiny app gallery

Mastering Shiny book

Why break this app?

I called this tutorial "Break this app" in the hopes that you would do just that: take the example app and break it. Then fix it, add your own elements to it, and adapt the basic structure for your project. The best way to learn Shiny is to simply change one or a few things at a time and observe the effects. Just pick something and change it. You can start simple: text color, size, or content, graph style, and then work up to things like changing page layouts and adding output. If it works, great! If it doesn't, you still learned something. Good luck and happy coding!