Data Visualization 2

Ryan Womack

2025-10-01

Data Visualization 2

interactive visualizations with Shiny, including hosting a Shiny server

1 Overview

Let's start by looking at the basic documentation and gallery examples at https://shiny.posit.co/ and https://shiny.posit.co/r/gallery/. Shiny allows us to build interactive data visualizations. It now supports Python visualizations as well, although this session will use R examples.

A few examples of Shiny in action:

- Explore ggplot2 with Shiny
- College Scorecard Data
- BTAA librarians

1.1 *Mastering Shiny*, by Hadley Wickham

We'll refer frequently to Mastering Shiny by Hadley Wickham, master R developer, which provide a complete presentation of working with Shiny beyond this introductory workshop. The open version of the text is linked above, but we also have access through Rutgers to the O'Reilly collection which contains the officially published version, along with other useful Shiny learning materials (and much more for all kinds of tech!).

2 Setup

2.1 Shiny and rsconnect

We will use the *pak* package for installation as a more complete approach to package management. Replace the pkg commands with *install.packages()* versions if you prefer.

The *shiny* package encapsulates most of what you need to run *shiny* (of course you'd need the packages for whatever kinds of graphics you'd like to display as well). The *rsconnect* package

is used to publish Shiny apps to the web, as we'll see later. Install *pak* and *tidyverse* if you don't already have them on your system. We will not run Python code in this session, but keep in mind that it is now possible to use Python in Shiny.

```
install.packages("pak", dependencies=TRUE)
library(pak)
pkg_install("shiny")
pkg_install("rsconnect")
```

Now let's load the *tidyverse*, *shiny*, and *rsconnect*.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 -
-
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.1     v stringr   1.5.2
v ggplot2   4.0.0     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.1.0
-- Conflicts ------------------------------------------ tidyverse_conflicts() -
-
x purrr::%||%()   masks base::%||%()
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become err
```

```
library(shiny)
library(rsconnect)
```

```
Attaching package: 'rsconnect'

The following object is masked from 'package:shiny':

    serverInfo
```

3 Shiny basics

We'll walk through some of the Shiny gallery examples to understand the fundamentals of Shiny.

3.1 ui and server

The *ui* section of the code controls the user interface, and often passes variables to the server side. The ui side has a number of Shiny-specific functions to control the interface (e.g., "sliderInput") and allow us to quickly build interactivity via these building blocks.

The *server* section of the code controls the generation of data, tables, and visualization, creating output that is passed to the ui side. While the output generated by the server may be encapsulated by a Shiny fuction, the core output is typically generated by standard R functions (or Python if you choose).

We should use the default labeling of "input" and "output" to render our code clear and consistent, although in theory these names are arbitrarily assigned.

3.2 Conventions and app.R

We put our Shiny files in a single directory, including supplementary files like data (unless using direct file location references). While all Shiny visualizations will have a *ui* and *server* component, we have two options on how to set this up:

1. Create two files, one called *ui.R* and the other called *server.R* with code describing the user interface and server commands respectively. We can also (optionally) use a third file called *global.R* to take care of some settings and globally applicable commands (like loading packages). This approach has the advantage of making the separate sides of the Shiny app easy to view and edit.
2. Create a single file, called *app.R*. This approach can be simpler to deploy, since it is obvious that everything happens in the single file, just one thing to update and upload. Inside the app.R file, we have a *ui* section and a *server* section. We then run the app by running *shinyApp(ui, server)* .

Here's a very basic single file *app.R*, as an illustration.

```
library(shiny)

ui <- basicPage(
  plotOutput("plot1", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(mtcars$wt, mtcars$mpg)
  })

  output$info <- renderText({
    paste0("x=", input$plot_click$x, "\ny=", input$plot_click$y)
  })
}
```

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

In the live workshop (and recording), we walk through creating the files in RStudio at this point.

3.3 Reactivity

The concept of *reactivity* is fundamental to the interactivity of the Shiny display. Reactive expressions dynamically update as inputs change.

The code wrapper *reactive({...})* is used to turn any function into a reactive expression. Then one can call it like a function. The difference between it and a function is that the reactive expression runs once and caches its result, unless and until its inputs are updated.

We can modify the default Shiny server.R to make define its reactive as follows:

```
function(input, output, session) {

  my_input <- reactive(input$bins)

  output$distPlot <- renderPlot({

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = my_input() + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white',
         xlab = 'Waiting time to next eruption (in mins)',
         main = 'Histogram of waiting times',
         ylab = paste(my_input(), ' bins'))

  })

}
```

```
function (input, output, session)
{
    my_input <- reactive(input$bins)
    output$distPlot <- renderPlot({
        x <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = my_input() +
            1)
        hist(x, breaks = bins, col = "darkgray", border = "white",
            xlab = "Waiting time to next eruption (in mins)",
            main = "Histogram of waiting times", ylab = paste(my_input(),
                " bins"))
    })
}
```

Note this is a very basic example that doesn't accomplish too much.

To continue the learning process, see these sections from *Mastering Shiny*:

- Shiny in Action
- Mastering Reactivity
- Best Practices

4 Publishing your Shiny app

In this section we walk through the process of using the RStudio interface to upload a Shiny app to shinyapps.io. This requires us to log in and perform a few steps (notably grabbing an authentication token), using the shinyapps instructions. Afterwards, we'll be able to publish to *my_account.shinyapps.io/my_project* (substituting your account name and your project folder name).

Shinyapps.io is free for 5 projects with limited bandwidth. Later we'll see how to completely take control of your own server. The publishing button used in RStudio for shinyapps.io might also be used for in house connection to an organization's own Shiny server (if you have that kind of tech support).

Note that shinylive is another option and approach to serving up your Shiny content.

5 Some extras

Like other widely used ecosystems, Shiny has evolved a variety of tools to extend its functionality and to make working with it easier.

One example is the *thematic* package which will automatically style your plots to match the customised style of your app, using the *thematic_shiny()* command.

There are other packages like shinydashboard or also flexdashboard which will help you quickly build dashboards.

A complete list of extensions is here.

6 Other topics

This is an introductory workshop, but the *Mastering Shiny* book and other sites gives many example of ways to improve your coding and make more powerful and complex Shiny apps, via topics like

- modularizing Shiny code
- scaling modules
- code tips such as the difference between data variables and environmental variables which is important in making reactive functions work properly, for example *diamonds %>% filter(.data[[var]] > .env$min)*
- debugging

- reactive graphs using the *reactlog* package which shows how the reactive graph evolves over time

7 Run your own Shiny server

Shiny gives some information on this at https://shiny.posit.co/r/deploy and also information on deploying Python on a Shiny server. *Posit Connect* is their commercial offering, but the open source *Shiny Server* remains free and open source.

7.1 Digital Ocean

To run one's own server, one has to set up a server someplace. This is a sketch of how that is done via DigitalOcean, which has a straightforward and easy-to-use infrastructure that I would recommend over the complexity and unreliability of AWS. This will cost at least $4 a month for a minimal setup.

7.1.1 Get a Droplet

First we set up an account with Digital Ocean, then set up a "Droplet", which in DigitalOcean's lingo is a self-contained server instance. You should be familiar with the basics of Linux and the command line to manage these steps.

Follow the instructions to install an Ubuntu or Debian server. Note that other distros are avaialble, but I prefer Debian for its long-term stability and commitment to open source.

7.1.2 Configure login with SSH

I highly recommend configuring login to the Droplet with SSH keys instead of password access.

These links may be helpful for that:

- Connect with OpenSSH
- Lost SSH key?
- Troubleshooting SSH
- add SSH identity

7.1.3 Install Nginx

We need a web server. The Nginx instructions are straightforward.

Then we should understand how to use the ufw firewall, and then install an SSL certificate. A modern server should use SSL, since most browsers have begun to reject unencrypted sites by default.

Our Nginx documents (for the home page or other non-Shiny html we want to host) will be put in

/var/www/html on the server, using DigitalOcean's tools, or *scp* or other transfer/editing methods.

7.1.4 Install Shiny

Now that the web server is up and running, we can follow the instructions to install Shiny. There are several steps here that should be followed carefully, in particular to use your own server names where appropriate.

The complete Shiny server guide is here.

7.1.5 Upload!

The Shiny apps (R files as well as data files) that are served up should be placed in

/srv/shiny-server/myprojectname

and your home page should be

/srv/shiny-server/index.html

I find it is easiest to just use *git clone* to pull the files in there. Note that, unlike the *rsconnect*/publish process, there is no generating of a bundle. The Shiny server can work directly with the R files in the directory.

7.1.6 Final steps

You may also have to configure your nameservers to point to the DigitalOcean droplet if it is just a component of your web services.

My little installation is at shiny.ryanwomack.com, with a demo project, CollegeSearchTool and actually uses Plotly for its graphics.

As always, *Enjoy R!*