Tutorial 10: Interactive Graphics

Leah Brooks

April 16, 2021

Contents

A. Get things ready **1**
A.1. Prepping R . . . 1
A.2. Additional resources . . . 2

B. Look at some examples **2**

C. Structure of a Shiny App **4**

D. Make a Tiny Shiny App **4**
D.1. Prepare the Arlington property data . . . 4
D.2. Make a Shiny App . . . 6
D.3. Run the Shiny App . . . 7

E. A Slightly More Sophisticated User Interface **8**

F. Adding to the output **14**

G. Homework **18**

Welcome to the final tutorial for Data Visualization Using R. Today we are taking the static graphic skills we've already learned and extending them into interactive graphics.

Both in policy work and otherwise, if you are arguing for a specific point, the static graphic is your best bet. It allows you, rather than the user, to determine what it is shown and how the evidence is framed.

However, sometimes your goal is not structured communication, but rather data dissemination. In those cases, interactive graphics can be useful in letting readers find data on their own, or draw their own conclusions from underlying data.

In this tutorial, we learn how to create interactive graphics with a package called Shiny. You'll learn how to build a webpage where users can input information and the graphic changes in response. In this class we'll learn how to build this Shiny app. If you want to learn how to deploy it, please see this page.

In addition to the R Shiny commands, we also learn the helpful R commands `readRDS()` and `saveRDS()`.

A. Get things ready

A.1. Prepping R

For today's tutorial, you'll need to install the "shiny" package. Do this ONCE only (do not put it in your R script):

```r
install.packages("shiny", dependencies = TRUE)
```

A word of warning: pay very careful attention when the tutorial tells you to create a new directory for files. This is *not a suggestion*, it is a *requirement*. Shiny is very particular about how files are named. Things will go totally haywire if you do not create new directories as needed.

Finally, you'll need to load the `shiny` package and the `tidyverse` package.

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

```
## -- Attaching packages ------------------------------------------------------------------------------

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------------------------ tidy
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

A.2. Additional resources

Today we learn the very minimal basics of Shiny. In writing today's tutorial, I referred to at least three helpful additional tutorials or pages:

- Basics from RStudio
- Advice from Shiny Expert and surfer
- Step-by-step tutorial from RStudio

If you would like to learn more, I would start with these.

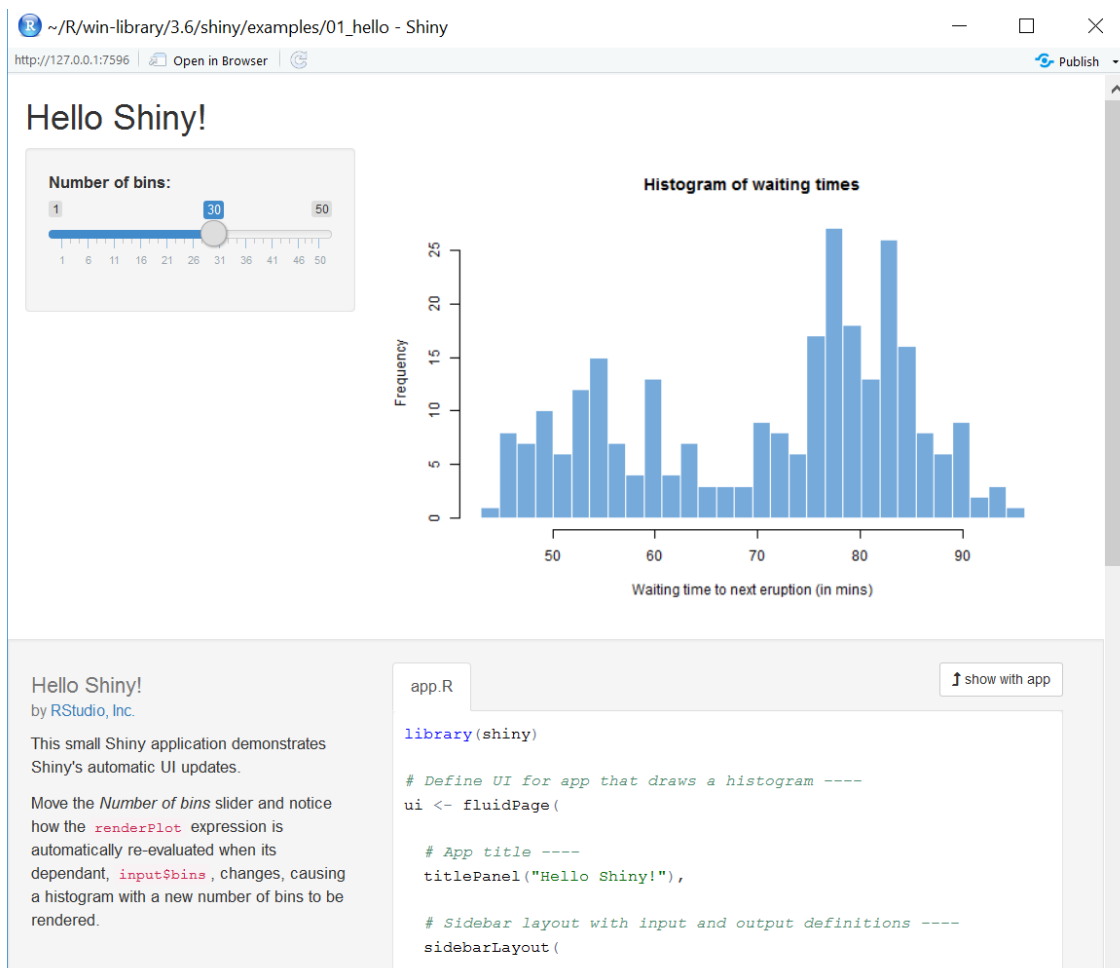
B. Look at some examples

Before we get started, it's helpful to see where we are heading. You should have already installed the Shiny package and called it using `library()`. Once you've installed and loaded the package, you have access to a set of examples that RStudio created to show you what Shiny does.

There are 11 examples in total. To look at the first one, type

```r
runExample("01_hello")
```

This will pop up a new window that should look like the picture below:



The top part – the bin choosing slider and the graph – are what you'll see when you write your Shiny code. For purposes of this example, the developers also show you the code that makes this example at the bottom. I recommend that you look this over as well, but don't worry too much about it. We'll go through code more carefully later. First, I want to make sure you know what Shiny does.

To that end, play with the bar at the left that tells R the number of bins to use for the histogram. See how the graph changes? This is at the heart of what Shiny does: it takes a user determined input and makes an output based on it.

Or, as this page says similarly: "The Shiny web framework is fundamentally about making it easy to wire up input values from a web page, making them easily available to you in R, and have the results of your R code be written as output values back out to the web page." User input in: graph or calculations out.

Now that you've looked at the first example, I recommend that you browse at least one or two more. Here is the complete list.

```
runExample("01_hello")       # a histogram
runExample("02_text")        # tables and data frames
runExample("03_reactivity") # a reactive expression
runExample("04_mpg")         # global variables
runExample("05_sliders")     # slider bars
runExample("06_tabsets")     # tabbed panels
runExample("07_widgets")     # help text and submit buttons
runExample("08_html")        # Shiny app built from HTML
runExample("09_upload")      # file upload wizard
```

```
runExample("10_download")    # file download wizard
runExample("11_timer")       # an automated timer
```

If you're up for more, there are lots of examples on this page. There's one on COVID that is probably the best match for the types of things we do in this class.

C. Structure of a Shiny App

Now that you know what a Shiny App can do, we'll start work on building one of our own.

Each Shiny app that you create has two key parts: a `ui` function and a `server` function. The `ui` part determines what the user sees. The `server` part creates outputs (for what the user sees) from the user-given inputs.

All Shiny apps have to be saved with the name "app.R". This means that if you're making multiple apps, as we will in today's class *you need to save each of them in its own directory.* This is so important that I will repeat! Each app you create

- must be named app.R and
- be saved in its own directory

The very simplest Shiny app – and this one does nothing – has four lines, as below. The first line loads the Shiny package. The second line defines the user interface (`ui`). The third line tells the computer what output to produce from the user input (`server`) and the final line calls the two parts you just defined.

Do not try to run this program in the console window. This is just to give you a sense of how the app gets set up.

```
# 1. load package
library(shiny)
# 2. define user interface
ui.def <- fluidPage()
# 3. define computed output to user interface
server.def <- function(input, output){}
# 4. call the app
shinyApp(ui = ui.def,
         server = server.def)
```

This is the basic outline. In the next step, we'll make a minimal example.

D. Make a Tiny Shiny App

You're now ready to make a very small Shiny app to get a sense of how things work. We will do this in two steps. First we'll prepare the Arlington property data from last tutorial to make the app building easier and faster. Then we'll write a tiny Shiny app and run it.

D.1. Prepare the Arlington property data

Our goal here is to create a small dataframe to make the data load faster when we run the Shiny app.

Read the Arlington properties data file from Tutorial 9. If you don't have this file, you can download it from here.

We want to do a few things with these data to prepare them

- clean up the year built variable to a plausible range
- create a variable that is the log of assessed value
- limit the variables in the dataframe

- save the dataframe as a R dataset

Hopefully the first three items on this list are things you already know how to do. I list these steps below.

```
# prep the arlington property data for this exercise
arl.p <- read.csv("H:/pppa_data_viz/2019/tutorial_data/lecture11/2019-04-19_arlington_2019_assessment_da
names(arl.p)
```

```
##  [1] "IvwPropertyAssessmentHistoryKey"   "ProvalLrsnId"
##  [3] "RealEstatePropertyCode"            "MasterRealEstatePropertyCode"
##  [5] "ReasPropertyStatusCode"            "PropertyClassTypeCode"
##  [7] "PropertyClassTypeDsc"              "PropertyStreetNbrNameText"
##  [9] "PropertyStreetNbr"                 "PropertyStreetNbrSuffixCode"
## [11] "PropertyStreetDirectionPrefixCode" "PropertyStreetName"
## [13] "PropertyStreetTypeCode"            "PropertyStreetDirectionSuffixCode"
## [15] "PropertyUnitNbr"                   "PropertyCityName"
## [17] "PropertyZipCode"                   "ZoningDescListText"
## [19] "TradeName"                         "PropertyYearBuilt"
## [21] "GrossFloorAreaSquareFeetQty"       "EffectiveAgeYearDate"
## [23] "NumberOfUnitsCnt"                  "StoryHeightCnt"
## [25] "ValuationYearDate"                 "CommercialPropertyTypeDsc"
## [27] "CondoModelName"                    "CondoStyleName"
## [29] "FinishedStorageAreaSquareFeetQty"  "StorageAreaSquareFeetQty"
## [31] "UnitNbr"                           "PropertyKey"
## [33] "ReasPropertyOwnerKey"              "ArlingtonStreetKey"
## [35] "PropertyExpiredInd"                "CommercialInd"
## [37] "DistrictNbr"                       "TaxExemptionTypeDsc"
## [39] "CondominiumProjectName"            "TaxBalanceAmt"
## [41] "AssessedYearDate"                  "TotalAssessedAmt"
## [43] "AssessmentDate"                    "AssessmentChangeReasonTypeDsc"
## [45] "ImprovementValueAmt"               "LandValueAmt"
```

```
# clean up year built variable
arl.p.toshiny <- filter(.data = arl.p, PropertyYearBuilt > 1875)

# make log of assessed property value for use later
arl.p.toshiny$ln.TotalAssessedAmt <- log(arl.p.toshiny$TotalAssessedAmt)

# limit to just these variables to make it smaller
arl.p.toshiny <- select(.data = arl.p.toshiny,
                        c("RealEstatePropertyCode","PropertyYearBuilt",
                          "TotalAssessedAmt","ln.TotalAssessedAmt"))

# check size
dim(arl.p.toshiny)
```

```
## [1] 62239     4
```

Now that we have a smaller dataframe, the last step is to save it. We save it as a R dataset so we can quickly load it. We save R datasets with the `saveRDS()` command which takes (at least) two inputs: the object (dataframe) you want to save and the location to which you want to save that object (`file =`).

```
saveRDS(object = arl.p.toshiny,
        file = "H:/pppa_data_viz/2020/tutorial_data/lecture12/small_arlington_2019_20200409.rds")
```

When you want to retrieve this dataframe – as we will below – you use the complementary `readRDS()` command.

D.2. Make a Shiny App

Now that the Arlington data are ready, we are ready to write our first Shiny app. Before writing the app, *create a new directory in which to save the app*. Recall that this app should be a directory with *no other apps*. Because each app needs its own directory, we will create multiple directories (and apps) throughout this tutorial.

My new directory is called `H:\pppa_data_viz\2020\tutorials\tutorial_12\first_shiny_app`

Now, open a new R window. Save this new file as "app.R" in the new folder that you just created.

Now you have an empty file called "app.R" that you need to fill up with commands. For your first file, type what I have below.

Let me describe the parts. First we load the `shiny` and `tidyverse` packages. We need the `shiny` package to make a Shiny app. We use the `tidyverse` package to get access to `ggplot` and `dplyr` and many other useful packages.

Under 2 below, we define the user interface. This basic user interface has three parts. The first is a title, which is defined by `titlePanel()`. All title text goes inside this command with quotes around it.

The second element of the user interface is the function to create a slider that takes user input. The function `sliderInput()` is pre-set by the Shiny package. The first element of this function is the inputID element. This is the name that you define (that you can use elsewhere in this program) for the input that the user gives with the slider. So, for example, if the user slides the little bar on the slider to 20, `input$bins` is equal to 20.

The remaining parts of the slider definition are the label you put on top of the slider (`label=`), the minimum and maximum number of bins allowed by the slider (`min` and `max`) and the default value at which the slider starts (`value`).

The final element of the user interface is a plot. Confusingly, this is the plot we create in step 3 (the server function). We name the plot we create in step 3 "distPlot" and then we refer to it here to put it into the user interface.

And that brings us to step three in the code below, or the server function. Inside the server function, there is a helper function called `renderPlot`. To show output based on the user-defined input you need to "render" something. You can render a plot, a table, or probably many other things about which I don't know.

Here, in the server function we

- load the Arlington data, using `readRDS()` and referring to the path in which we saved it earlier
- create a histogram of the year the structure on each property was built

Note that in this second step, the number of bins is set equal to `input$bins`, which the user defined in the user interface. Apart from this, everything about the historgram stays constant in each drawing.

Finally, we tell R that we have assembled the parts of the app by writing the final line

```
shinyApp(ui = ui.stuff.that.shows, server = server.stuff.that.processes)
```

So, to recap, type the code below in a new directory (the name of which you need to remember), and save it as `app.R`.[1].

```
# 1. load packages
library(shiny)
library(tidyverse)

# 2. UI
# Define UI for app that draws a histogram ----
```

[1] This code is, with some small edits, taken from the R Studio Shiny (tutorial page)[https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/)]

```r
ui.stuff.that.shows <- fluidPage(

  # App title ----
  titlePanel("Hello Shiny!"),

  # Input: Slider for the number of bins ----
  sliderInput(inputId = "bins",
              label = "Number of bins:",
              min = 1,
              max = 50,
              value = 30),

  # Output: Histogram ----
  plotOutput(outputId = "distPlot")

)

# 3. server
#Define server logic required to draw a histogram ----
server.stuff.that.processes <- function(input, output) {

    # This expression that generates a histogram is wrapped in a call
    # to renderPlot to indicate that:
    #
    # 1. It is "reactive" and therefore should be automatically
    #    re-executed when inputs (input$bins) change
    # 2. Its output type is a plot
    output$distPlot <- renderPlot({

      dats <- readRDS("H:/pppa_data_viz/2020/tutorial_data/lecture12/small_arlington_2019_20200409.rds")

      ggplot() +
        geom_histogram(data = dats,
                       mapping = aes(x = PropertyYearBuilt),
                       bins = input$bins) +
        labs(title = "Number of Properties by Year Built",
             x = "Year Structure on Property was First Constructed",
             y = "Number of Properties")


    })

  }

# 4. Define app
shinyApp(ui = ui.stuff.that.shows, server = server.stuff.that.processes)
```

D.3. Run the Shiny App

Having just created the app, you're now ready to run it. To run it, type

```r
runApp("[directory where you saved app.R]")
```

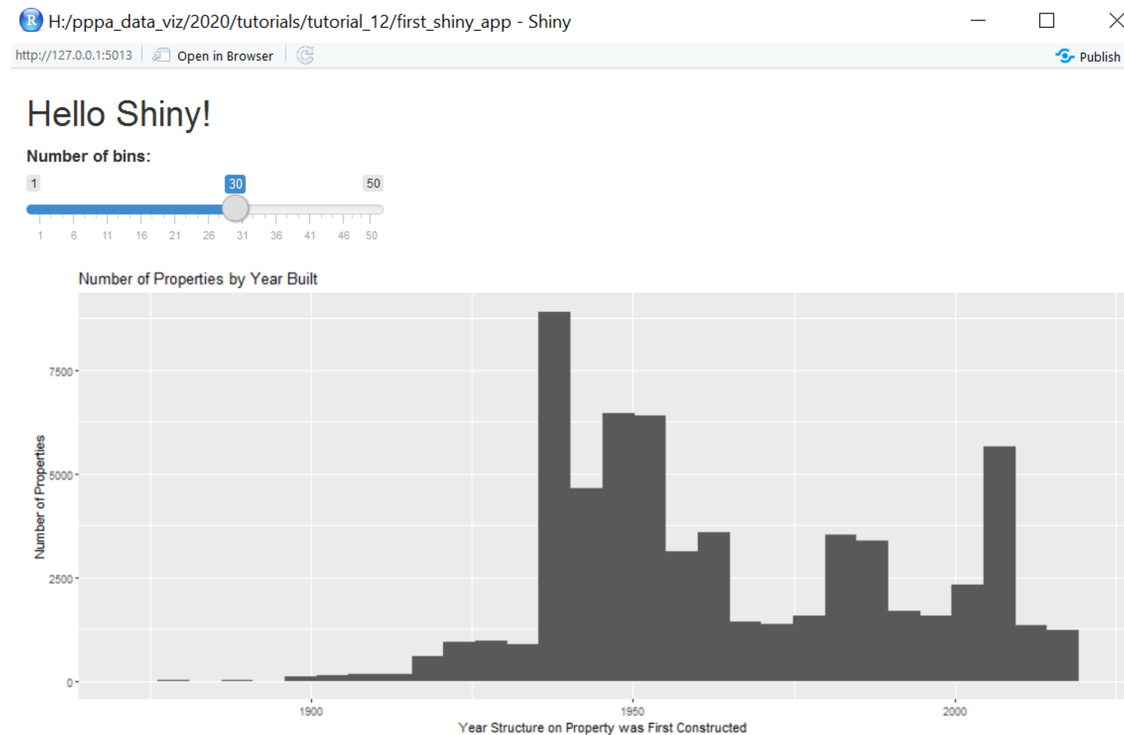
You can do this either in a separate R script (probably easiest so you know what it is you've done), or by

typing directly at the console. However, to run the Shiny app, **do not run your entire R script. Instead, highlight this particular line of code and run just this line.** Run just this line by going to the Code menu and choosing "Run Selected Line(s)."

For my part, I typed the below in my R program and ran just this line:

```
runApp("H:/pppa_data_viz/2020/tutorials/tutorial_12/first_shiny_app")
```

Then a screen pops up that looks like the picture below. When I change the value of the slider, the number of bins on the histogram changes.



Note that your console is busy while you're running this app, so you can't run any other programs. Quit out of the Shiny window to get back to a prompt in the console.

E. A Slightly More Sophisticated User Interface

In step D, we created the bones of a Shiny app. Now, in step E, we are going to add to the user interface. In section F we will add to the server output.

In this section, we will first add text to the user interface in E.1. and then add another input widget in E.2.

Here and for the rest of the tutorial, you may find the R Shiny cheatsheet helpful.

E.1. Add text and sections

We start by adding text to the user interface. This is non-interactive text – text that gives general context to the user and explains what you are asking them to input.

We are also going to add a grey box, in a separate section, to the interface. We'll use the grey box, called a sidebar, to separate out where the user changes values.

All the text that you want to show up for users – either in the sidebar or elsewhere – on your Shiny app needs to be inside some kind of command. If you know HTML, these commands parallel HTML commands.

If you don't know HTML, don't worry about it – just take a look at how the commands are structured, and you'll get a sense of how the input is set up.

Here are two commands that are helpful in entering text. There are zillions more, but I think this is enough for today!

- `p()` surrounds each paragraph
- `br()` is a line break

For the millions of other commands, see the UI section of the cheatsheet linked above.

One way to make text easier for the reader is to separate the widget where the user enters. In the "Layouts" section of the cheatsheet, you can see the `sidebarLayout()` option that we'll use today, along with the many other options that you could choose. In essence, you tell R that you're going to use a sidebar layout and then you need to tell R, through the two subparts, what goes in the sidebar panel (`sidebarPanel()`) and what goes in the main panel (`mainPanel()`).

With all that said, now let's revise our app.R program to have this updated user interface. The first thing to do is to create a new directory for our new program (so we preserve the old one). I call my new subdirectory `improved_ui`.

The updated `app.R` file reads as below. We are changing text *only* in the user interface section.

```
library(shiny)
library(tidyverse)

# Define UI for app that draws a histogram ----
ui.stuff.that.shows <- fluidPage(

  # App title ----
  titlePanel("Analysis of Arlington Properties"),

  # you cant just put random text in this part. it needs to be inside some kind of directions
  sidebarLayout(

    # this where we put the sidebar text
    sidebarPanel(

      p("This is the sidebar panel, where we have chosen to gather inputs."),

      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),

    # this is where we put the text for the main panel
    mainPanel(

      p("This is the main panel."),
      br(),
      p("Here we gather some input from you, dear reader, about what you'd like to learn
        about Arlington County."),
      br(),
      p("And we also put the inputs into a little box called a sidebar."),
```

```r
      br(),
      p("Look at the sidebar layout bit on the cheat sheet to see what it's doing.
        It has a sidebar and a main part. We use both."),
      br(),
      p("We can also write code here with special tags: "),
        code("# program R with comments"),

      # Output: Histogram ----
      plotOutput(outputId = "distPlot")

    )
  )
)


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

    # This expression that generates a histogram is wrapped in a call
    # to renderPlot to indicate that:
    #
    # 1. It is "reactive" and therefore should be automatically
    #    re-executed when inputs (input$bins) change
    # 2. Its output type is a plot
    output$distPlot <- renderPlot({

      dats <- readRDS("H:/pppa_data_viz/2020/tutorial_data/lecture12/small_arlington_2019_20200409.rds")

      ggplot() +
        geom_histogram(data = dats,
                       mapping = aes(x = PropertyYearBuilt),
                       bins = input$bins) +
        labs(title = "Number of Properties by Year Built",
             x = "Year Structure on Property was First Constructed",
             y = "Number of Properties")


    })

  }


shinyApp(ui = ui.stuff.that.shows, server = server.stuff.that.processes)
```

Hopefully you can look at the text above and see the sidebar panel, along with the text inside the main panel (inside `p()` markers) and the slider input widget inside the sidebar panel.

To run this file we type

```r
runApp("H:/pppa_data_viz/2020/tutorials/tutorial_12/improved_ui")
```

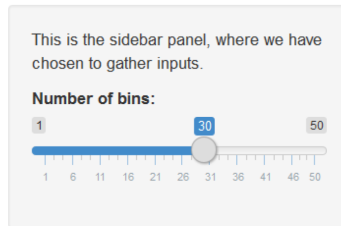
in a separate program or in the console. If in a separate program, run **just this line**.

My version looks as below:

H:/pppa_data_viz/2020/tutorials/tutorial_12/improved_ui - Shiny

http://127.0.0.1:5013 | Open in Browser | Publish

Analysis of Arlington Properties



This is the main panel.

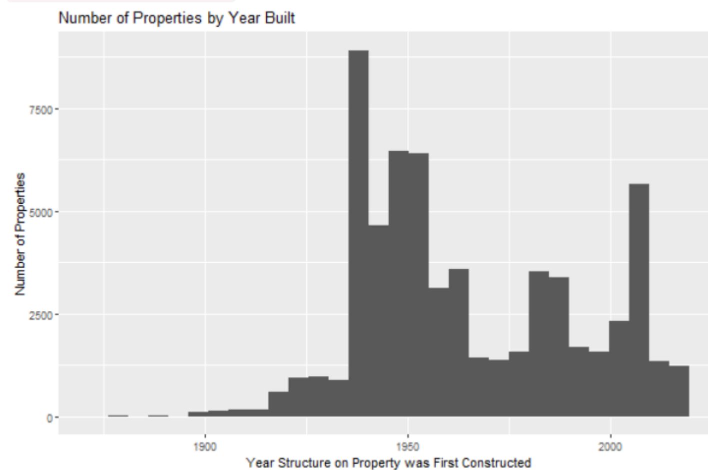
Here we gather some input from you, dear reader, about what you'd like to learn about Arlington County.

And we also put the inputs into a little box called a sidebar.

Look at the sidebar layout bit on the cheat sheet to see what it's doing. It has a sidebar and a main part. We use both.

We can also write code here with special tags:

```
# program R with comments
```



E.2. Add another data input part

Before we move to the server side, we will add one more element to the user input: another widget that allows users to input information.

Shiny has a variety of widgets that you can use to gather data from users – and these are (I think) the only way to gather data from users. You can see all the available widgets here.

We already have a slider. The goal of the new widget is for the user to tell us whether our next plot should be versus log of property assessed value, or just plain old assessed value (no log). For this kind of restricted choice, you could use radio buttons, a checkbox, or a list. We're going to use the list version, which is the "select box" widget.

Click on the button that says "code" below this widget and a new window opens up, showing the `server` and `ui` code. You'll need to click the `ui` tab and then you should see sample code, as below

server.R | ui.R

show with app

```
fluidPage(

  # Copy the line below to make a select box
  selectInput("select", label = h3("Select box"),
    choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3),
    selected = 1),

  hr(),
  fluidRow(column(3, verbatimTextOutput("value")))

)
```

Code license: MIT

The first (unhelpfully unlabled) element inside the `selectInput` function should have a `inputId =` associated with it, to let know what that that is what the output of this selection is going to be called. The `label` element lets you give a heading for the box, and the choices list is where you specify the choices available to the user. The left side of each pair is the what the user sees. The right side of each pair is how you'd like to refer to this variable. Finally, the `selected` option sets the default selection when the app loads.

To add this widget to the app.R code, I create yet another new sub-directory called "add_widget" and save the code below in that sub-directory.

```
library(shiny)
library(tidyverse)

# Define UI for app that draws a histogram ----
ui.stuff.that.shows <- fluidPage(

  # App title ----
  titlePanel("Analysis of Arlington Properties"),

  # you cant just put random text in this part. it needs to be inside some kind of directions
  sidebarLayout(

    sidebarPanel(

      p("This is the sidebar panel, where we have chosen to gather inputs."),

      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30),

      # Input: Select boxes for levels or logs of property values
      selectInput(inputId = "level.or.log",
                  label = "Property Value in Levels or Logs?",
                  choices = list("Levels" = "TotalAssessedAmt",
                                 "Logs" = "ln.TotalAssessedAmt"),
                  selected = 1)

    ),
    mainPanel(
```

```
      p("This is the main panel."),
      br(),
      p("Here we gather some input from you, dear reader, about what you'd like to learn
        about Arlington County."),
      br(),
      p("And we also put the inputs into a little box called a sidebar."),
      br(),
      p("Look at the sidebar layout bit on the cheat sheet to see what it's doing.
        It has a sidebar and a main part. We use both."),
      br(),
      p("We can also write code here with special tags: "),
        code("# program R with comments")

    )
  )
)


      # Output: Histogram ----
      plotOutput(outputId = "distPlot")


    )
  )
)


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

    # This expression that generates a histogram is wrapped in a call
    # to renderPlot to indicate that:
    #
    # 1. It is "reactive" and therefore should be automatically
    #    re-executed when inputs (input$bins) change
    # 2. Its output type is a plot
    output$distPlot <- renderPlot({

      dats <- readRDS("H:/pppa_data_viz/2020/tutorial_data/lecture12/small_arlington_2019_20200409.rds")

      ggplot() +
        geom_histogram(data = dats,
                       mapping = aes(x = PropertyYearBuilt),
                       bins = input$bins) +
        labs(title = "Number of Properties by Year Built",
             x = "Year Structure on Property was First Constructed",
             y = "Number of Properties")


    })

  }
```

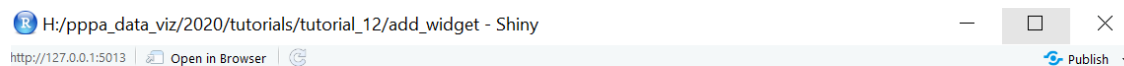
```
shinyApp(ui = ui.stuff.that.shows, server = server.stuff.that.processes)
```

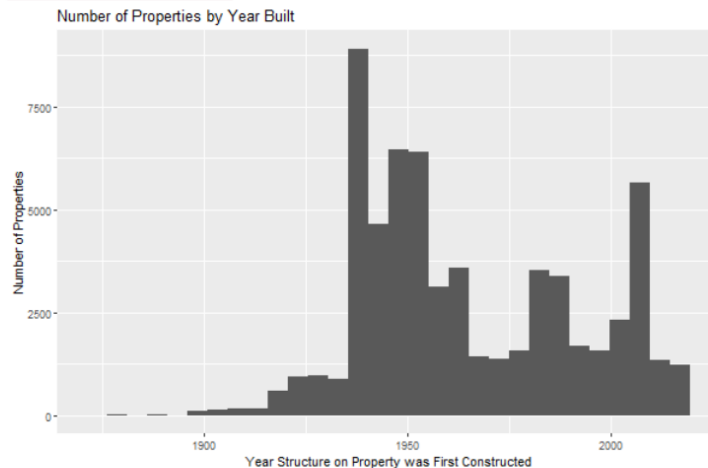
The only difference between this file and the previous is the additional widget that we've added inside the sidebar. Nothing changes in the output when you change this element (test it!) since we haven't linked the user input to a rendered output (that's section F). We can refer to this user-generated input as `input$level.or.log`, since that is what we named it in the `InputId` element.

To run this app, type

```
runApp("H:/pppa_data_viz/2020/tutorials/tutorial_12/add_widget")
```

My app looks like





F. Adding to the output

Our last step for this tutorial is to take the new user input we just solicited and create an output from it. To do this, we'll make one final new app.R file in yet another new sub-directory. I call mine "add_prop_vals."

Our goal here is to add a second plot to our Shiny app, showing the relationship between the year a property's structure was built and the assessed value of that property. We already have the user input from our modifications in E.2, and it's called `input$level.or.log`. Now we need to add a scatter with this input on the y axis.

For brevity, I am just putting the "UI" and the "server" portions of the app here, which are the only parts of

the code you should update. But *you need all four parts of the app* to have the whole thing work.

The UI part looks as below. The key addition is the additional `plotOutput` in the main panel (and the comma after the original output) to tell R that it should draw a second graph.

```
# Define UI for app that draws a histogram ----
ui.stuff.that.shows <- fluidPage(

  # App title ----
  titlePanel("Analysis of Arlington Properties"),

  # you cant just put random text in this part. it needs to be inside some kind of directions
  sidebarLayout(

    sidebarPanel(

      p("This is the sidebar panel, where we have chosen to gather inputs."),

      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30),

      # Input: Select boxes for levels or logs of property values
      selectInput(inputId = "level.or.log",
                  label = "Property Value in Levels or Logs?",
                  choices = list("Levels" = "TotalAssessedAmt",
                                 "Logs" = "ln.TotalAssessedAmt"),
                  selected = 1)

    ),
    mainPanel(

      p("Less text so maybe graphs will fit."),

      # Output: Histogram ----
      plotOutput(outputId = "distPlot"),


      # Output: Scatter ----
      plotOutput(outputId = "scatter.vals")

    )
  )
)
```

We define the second graph in the server portion of the code. The server portion looks like this:

```
#Define server logic required to draw a histogram ----
server.stuff.that.processes <- function(input, output) {

    # This expression that generates a histogram is wrapped in a call
    # to renderPlot to indicate that:
    #
```

```
  # 1. It is "reactive" and therefore should be automatically
  #    re-executed when inputs (input$bins) change
  # 2. Its output type is a plot
  output$distPlot <- renderPlot({

    dats <- readRDS("H:/pppa_data_viz/2020/tutorial_data/lecture12/small_arlington_2019_20200409.rds")

    ggplot() +
      geom_histogram(data = dats,
                     mapping = aes(x = PropertyYearBuilt),
                     bins = input$bins) +
      labs(title = "Number of Properties by Year Built",
           x = "Year Structure on Property was First Constructed",
           y = "Number of Properties")
  })

  # This expression that generates a scatter is wrapped in a call
  # to renderPlot to indicate that:
  #
  # 1. It is "reactive" and therefore should be automatically
  #    re-executed when inputs (input$bins) change
  # 2. Its output type is a plot
  output$scatter.vals <- renderPlot({

    dats <- readRDS("H:/pppa_data_viz/2020/tutorial_data/lecture12/small_arlington_2019_20200409.rds")

    # this is text to fix up the labels below
    # there may be a more elegant way to do this, but I don't know it!
    desc.value <- ifelse(input$level.or.log == "ln.TotalAssessedAmt", "Log of ", "")

    # make name for variable in input
    dats$dv <- ifelse(test = input$level.or.log == "ln.TotalAssessedAmt",
                      yes = dats$ln.TotalAssessedAmt,
                      no = dats$TotalAssessedAmt)

    # this code sets up a new variable that takes the value of the input (level or log variable)
    dats$ll <- input$level.or.log

    # this makes a new variable called -dv- that is either the level or log
    dats$dv <- ifelse(test = dats$ll == "ln.TotalAssessedAmt",
                    yes = dats$ln.TotalAssessedAmt,
                    no = dats$TotalAssessedAmt)

    # make the plot to appear
    ggplot() +
      geom_point(data = dats,
                 mapping = aes(x = PropertyYearBuilt, y = dv),
                 size = 0.2) +
      labs(title = paste0("Property Year Built vs ", desc.value, "Assessed Value"),
           x = "Year Structure on Property was First Constructed",
           y = paste0(desc.value,"Assessed Value"))
  })
}
```

What this code adds is the second `renderPlot()` command. Inside of this command we load the data (this might be unnecessary). We then need to use the user-input variable in the scatter and modify the y label so that it is comprehensible.

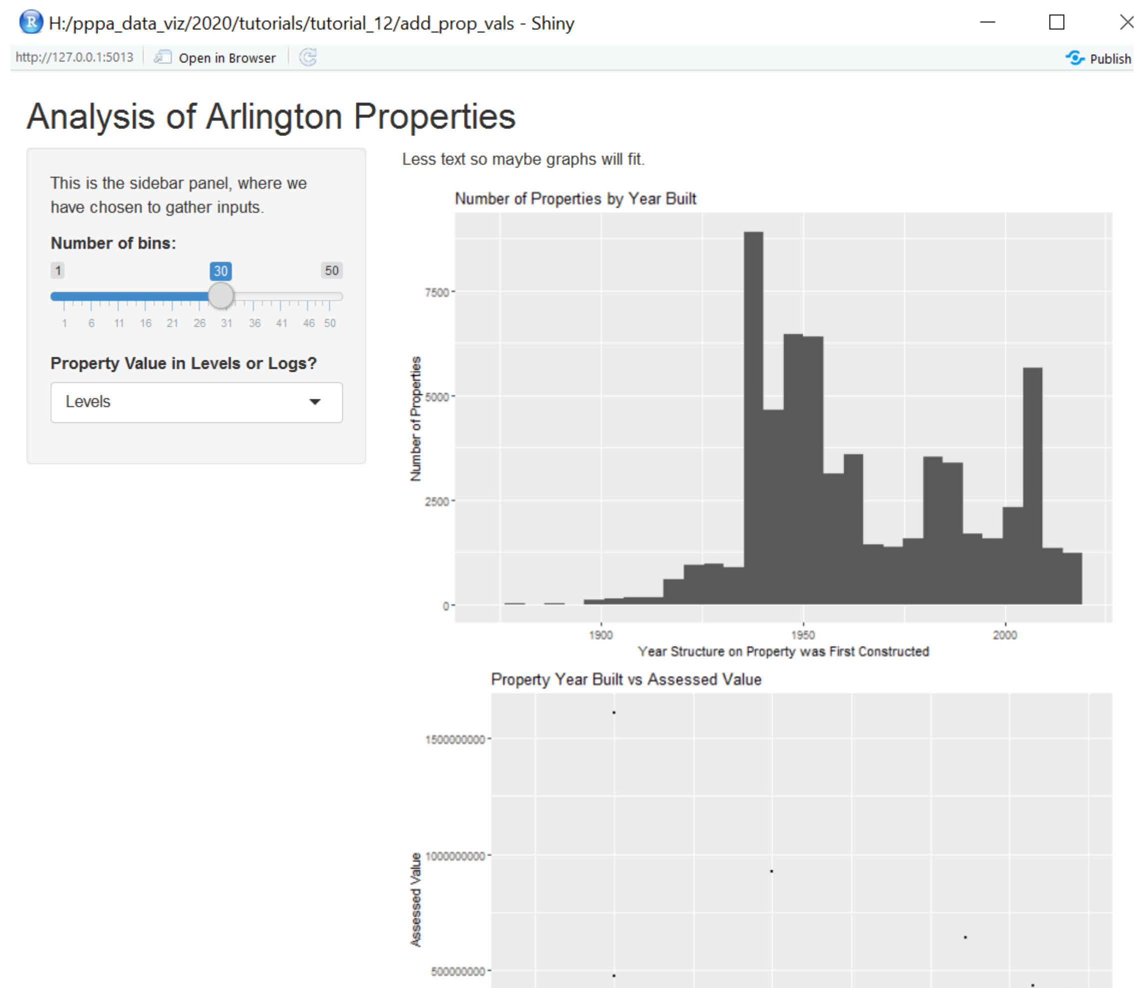
We use the user-input value of `input$level.or.log`, but with one small twist. The `aes()` command expects to see a variable in the dataframe after `x=` or `y=` – not an input like `input$level.or.log` that resolves to a variable. Due to this, we turn to using `aes_string()`, rather than plain old `aes()`, which takes input variables as strings. Note that this means we have to change how we refer to the `x` variable as well, putting that in quotes.

And finally, we need to do something for the y axis label, which would just show the variable name if we didn't make a fix. Generally, we need something that works for both variables – and the only difference between the two is that one of them is logged. So, above the `ggplot` command, we create a little stand alone variable caleld `desc.value`. We use an `iselse` function to set this variable to "Log of" if the user input is that we should use a log value; otherwise the value of this new variable is nothing (`""`). Note that we use this little variable in combination with `paste0` in two places: in the title and in the y label.

Now try to run this new Shiny app. I run mine by typing

```
runApp("H:/pppa_data_viz/2020/tutorials/tutorial_12/add_prop_vals")
```

Here's what mine (and hopefully yours) looks like:



There are literally thousands of additional things you could do for this Shiny app, and to make this output prettier. But we have now covered the basics, and hopefully you've gotten a good overview of what Shiny

can do. Now it's your turn to try with a new dataset.

G. Homework

Make a Shiny app with two inputs and two graphs using a different dataset.

For this homework, sumbit just your app.R file and the data on which the app relies.

Follow these instructions so I do not have to fiddle with each individual assignment. In your google folder, make a "tutorial 10" sub-folder. In this sub-folder, put your `app.R` program and any data to which the program refers.

Make sure that I can run your `app.R` without adjustments. In other words, for example, don't refer to data on your C drive. Instead, refer to data in the same folder as the app.

Imagine that your app is in `c:\brooks\app_loc`. In this case, this is a bad way to read data

```
data.for.app <- read.csv("c:\brooks\app_loc\mydata.csv")
```

since it won't work when I try to run your app.

But the below is ok, as long as `mydata.csv` is in the same folder as `app.R`.

```
data.for.app <- read.csv("mydata.csv")
```

Good luck.