# Interactive Web Apps in R with shiny

Clay Ford

Spring 2019

# What is shiny?

- shiny is a package that provides functions for converting R code into interactive web applications
- The applications it creates can be run in RStudio; they don't have to be hosted on a web site
- Developed by RStudio
- Doesn't require web development skills but does require a good grasp of R

# Why use shiny?

Create interactive apps. . .

- for teaching statistical concepts: Example
- for using and visualizing statistical models: Example
- for exploring data or creating reports: Example

See more at the shiny Gallery: https://shiny.rstudio.com/gallery/

# Agenda

- Examine the RStudio shiny app template
- Build a basic shiny app
- Build a more advanced shiny app to explore a linear model of the Albemarle County Homes Data

# Installing shiny

shiny is an R package, so install as you would any other package:

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

At the beginning of any shiny app you will need to load shiny.

```
library(shiny)
```

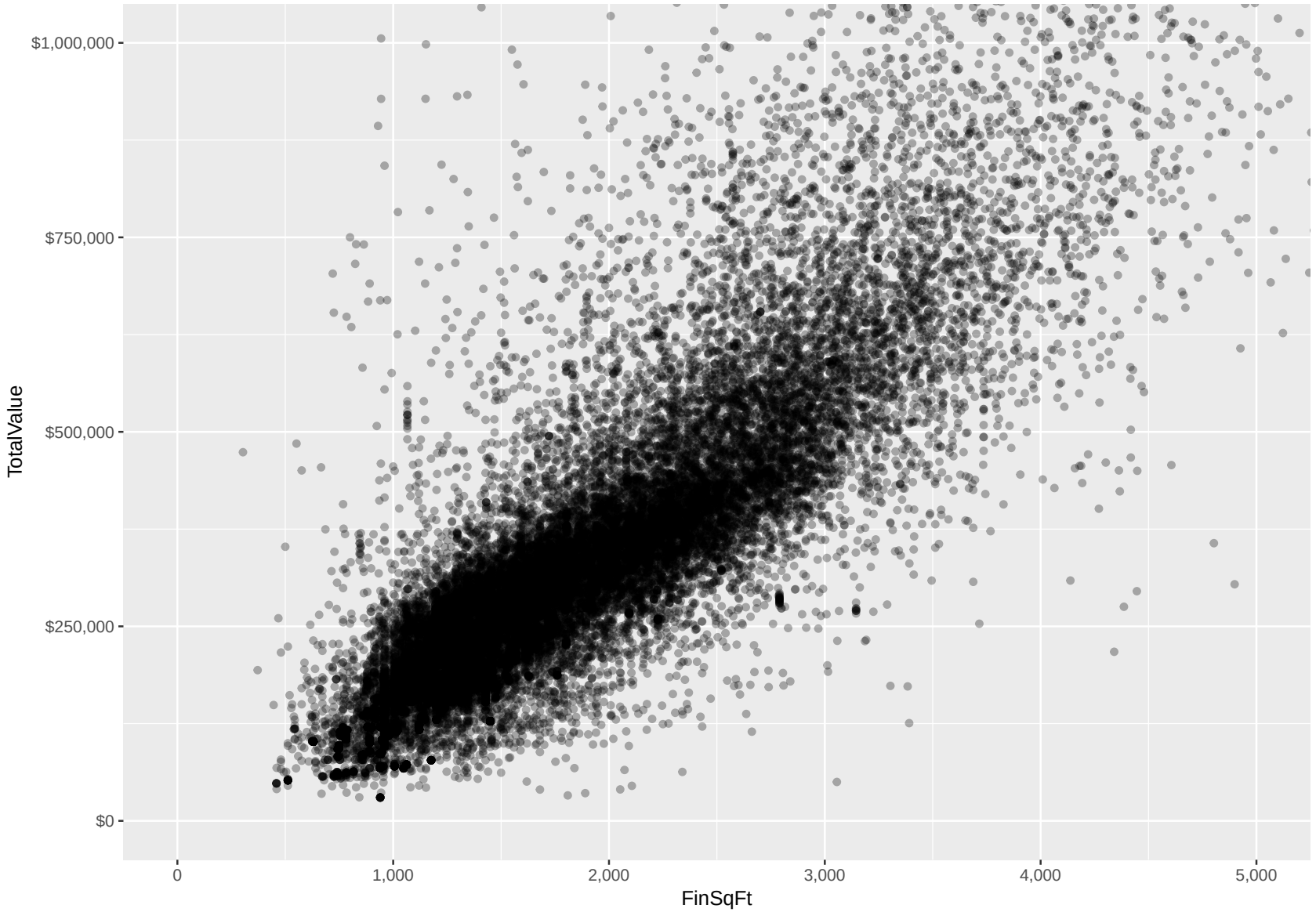
RStudio provides a shiny template to help us get started building a shiny app.

# Suggested steps for building a shiny app

1. Start work in a normal R script developing code that does what you want; identify arguments or settings that you would like to make interactive
2. (optional) Draw on paper a rough sketch of what the app might look like
3. Start a shiny app using RStudio template: File. . . New File. . . Shiny Web App. . .
4. Copy in your code from step 1 and modify with shiny code to convert to interactive app

# Simple motivating example

Zoom in on scatter plot of TotalValue vs FinSqFt and adjust alpha setting (transparency of points).

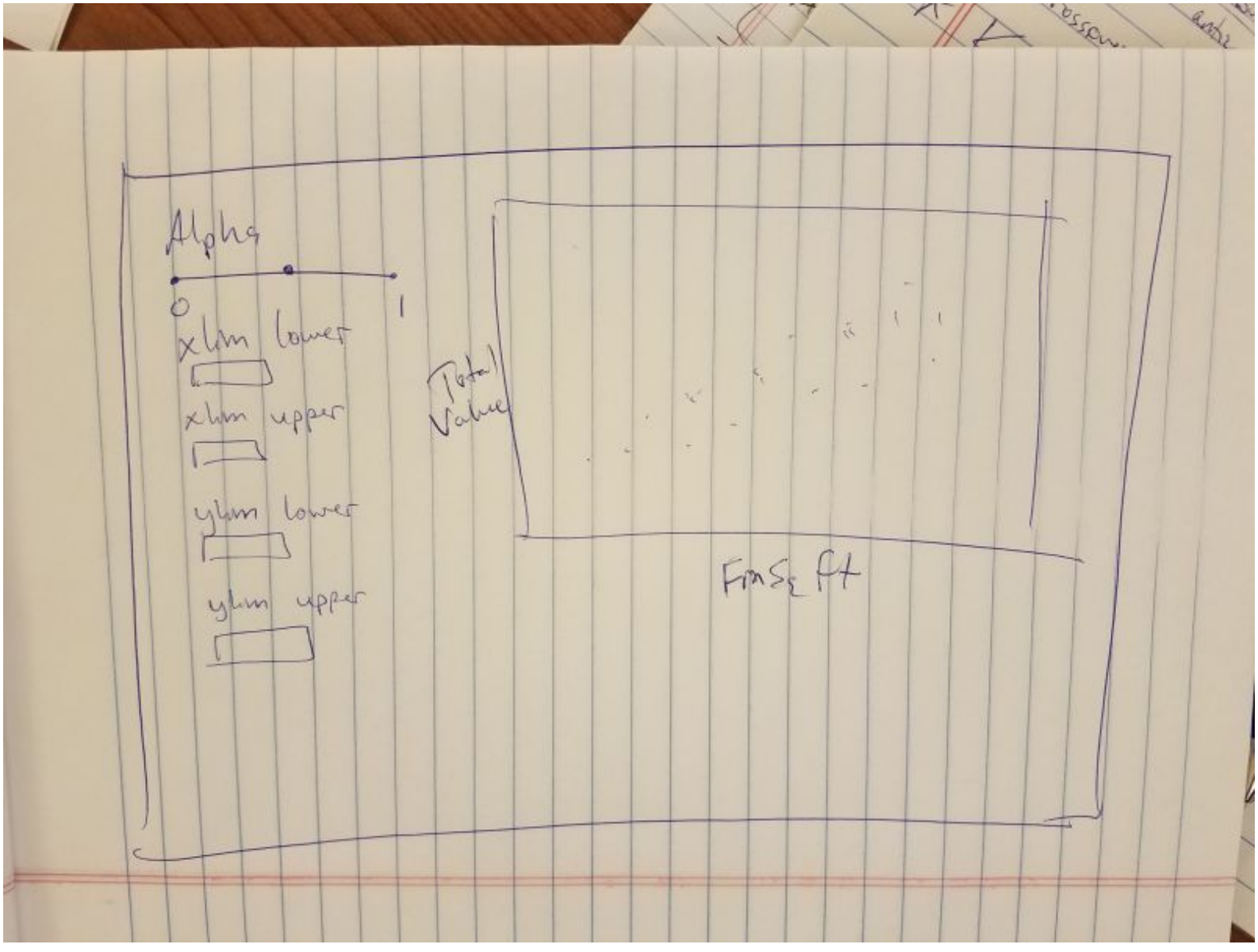


# The R code

```
ggplot(homes, aes(x = FinSqFt, y = TotalValue)) +
  geom_point(alpha = 0.3) +
  scale_x_continuous(labels = scales::comma) +
  scale_y_continuous(labels = scales::dollar) +
  coord_cartesian(xlim = c(0,5000),ylim = c(0,1e6))
```

It would be nice to interact with the alpha, xlim and ylim arguments.

# Sketch of a possible app

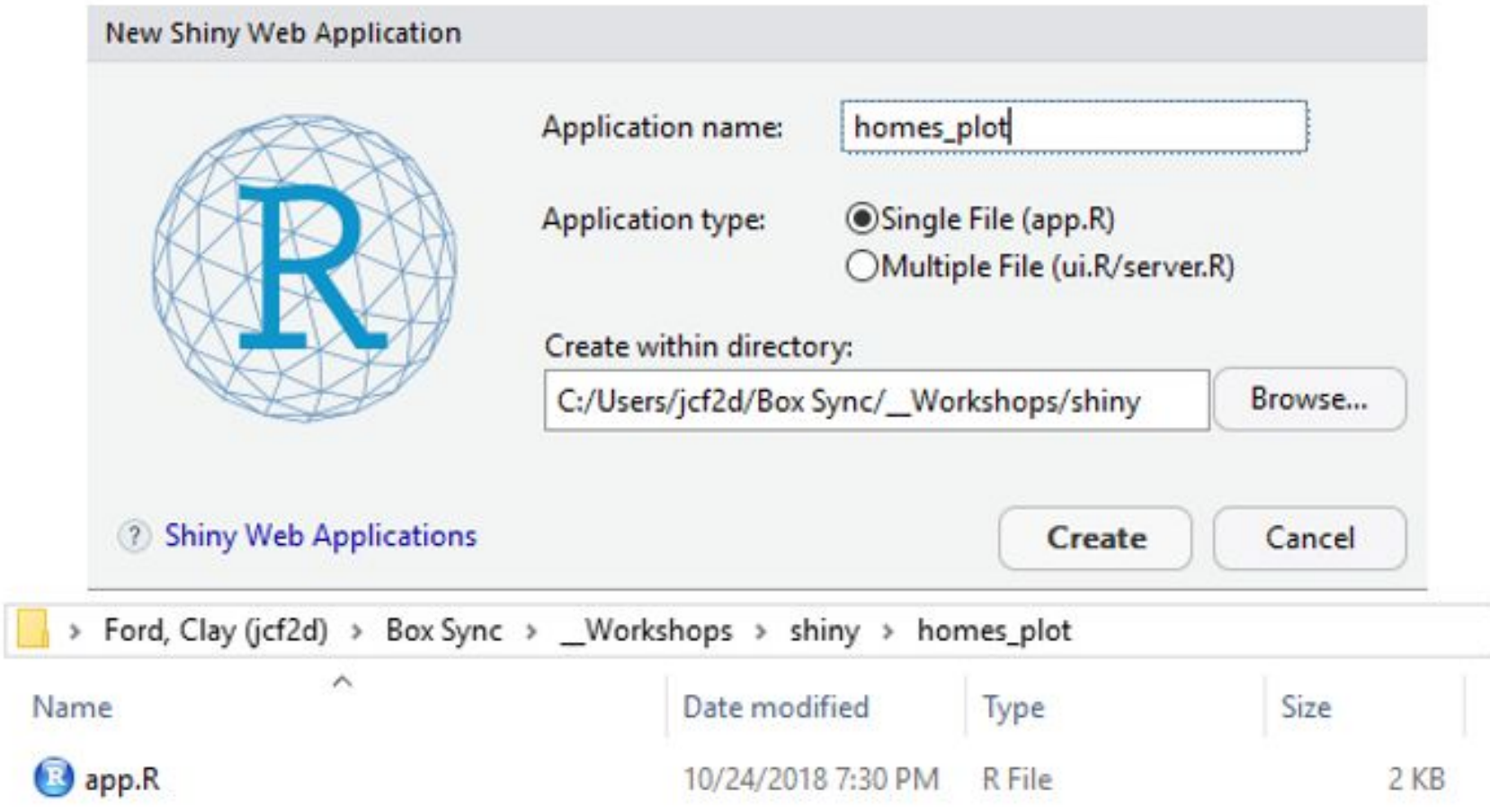


# Let's build that shiny app!

Here's how we get started:

1. In RStudio, go to File... New File... Shiny Web App...
2. In the Application Name field, enter a name for your application. (Example: "homes_plot") This will become the *name of the folder* where your shiny app is saved.
3. Leave Application Type set to "Single File (app.R)"
4. Browse to where you want to save your shiny app. In that location a folder will be created with the name of your application. In that folder will be a file called "app.R". **Do not change the name of that file!**

# New Shiny Web App dialog



# Shiny Web App template

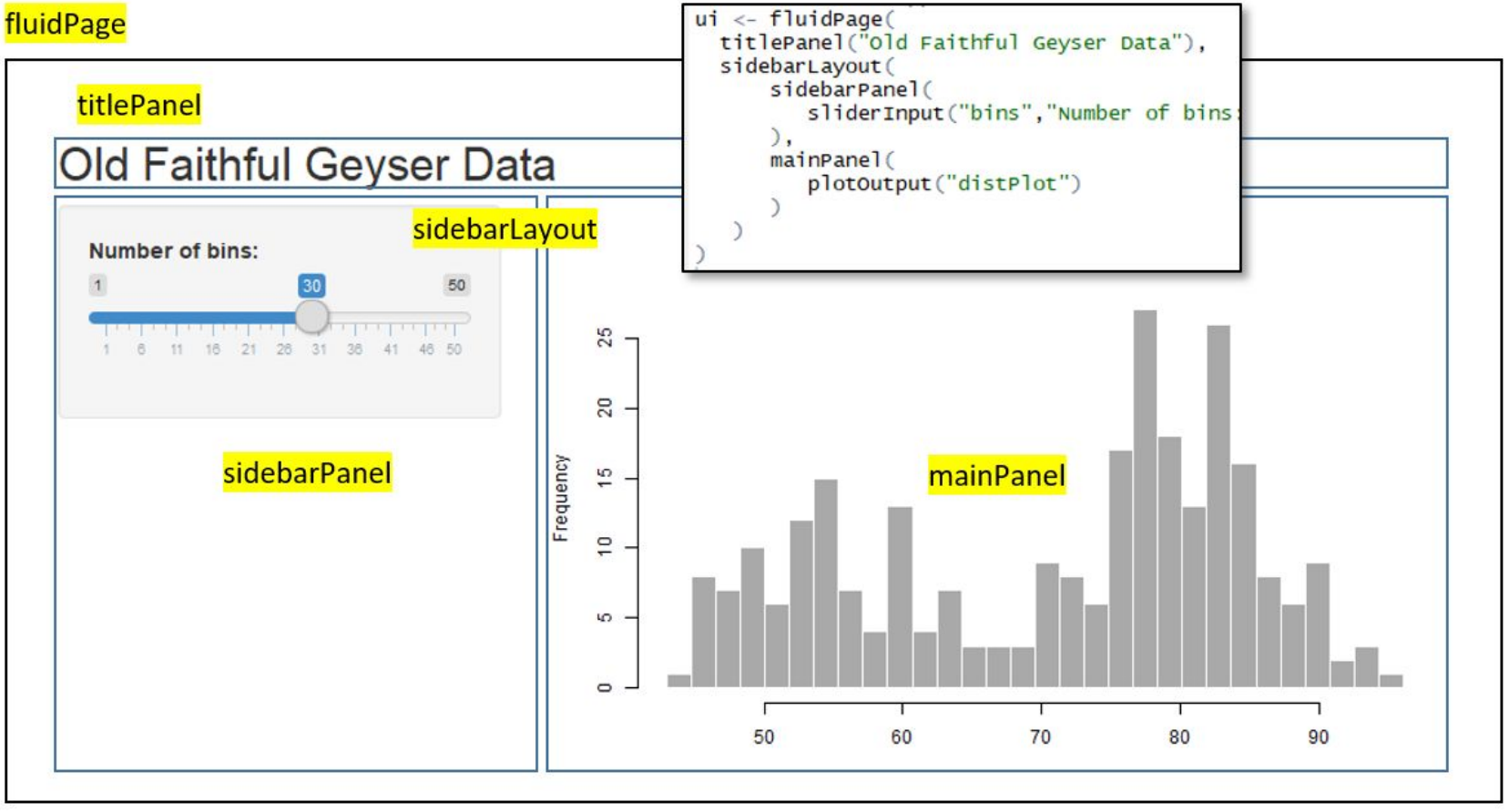
When you start a new shiny web app you are a provided a working app as a template to help get your started. Feel free to run it and see what it does by clicking the Run App button.

It's a simple app, but it provides the core shiny components we need to start building our own shiny app.

1. the user interface (an object called `ui`)
2. the server logic (an object called `server`; the R code of our app)
3. the function call `shinyApp(ui = ui, server = server)`

Building your shiny app means modifying the user interface, server, and comments.

# Default user interface: sidebar layout

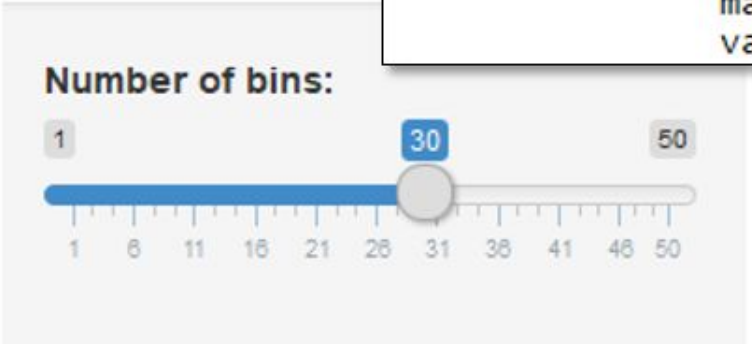



This is the layout we'll use in the workshop. It's very functional. See the shiny cheat sheet for several other layouts. (RStudio: Help. . . Cheatsheets. . . )

# Inputs

- Inputs allow us to collect values from the user
- These can be numeric fields, radio buttons, pull down lists, slider bars, check boxes, action buttons, text fields, etc
- The shiny template provides us with a slider bar, created with the `sliderInput` function

```
sidebarPanel(
   sliderInput("bins",
               "Number of bins:",
               min = 1,
               max = 50,
               value = 30) # default value when app starts
```



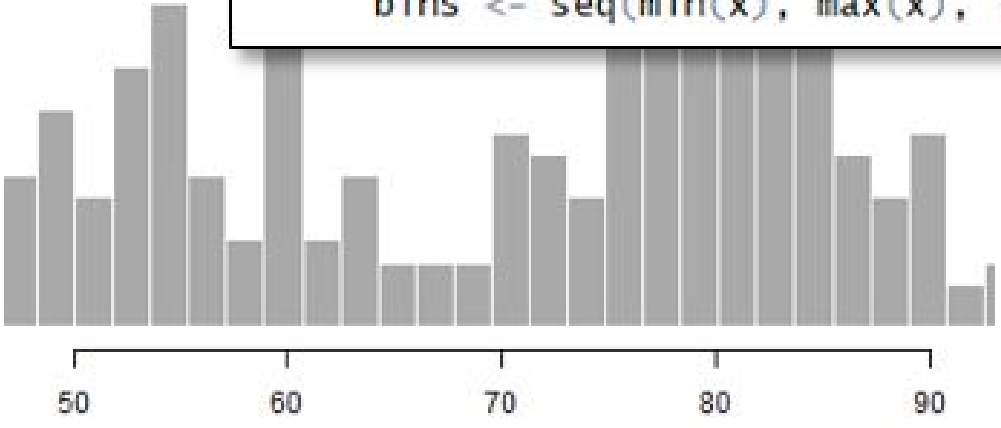

- The value collected is input into our R code as `input$bins`

# Outputs

- The user interface not only collects values from the user but also displays R *output*, such as plots, tables and statistical summaries.
- The shiny template provides us with plot output, created with the `plotOutput` function
- `plotOutput` is called on `"distPlot"`, which is created in the server portion of the app using a `renderPlot` function.
- `"distPlot"` is a user-generated name; it could have been called `"histogram"`
- In general, output that is presented in the user interface will be created in the server portion using a `render*` function

# render and Output functions work together

```
        mainPanel(
           plotOutput("distPlot")
        )
    )
)

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

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



# The server

The "server" is a function where the application's R code is executed. The following process roughly summarizes how a shiny app works:

1. user provides input in the user interface (*eg*, *number of bins*)
2. the input is passed to the server where the R code is processed (*eg*, *create histogram with specified number of bins*)
3. the output is passed back to the user interface to be displayed (*eg*, *plot histogram*)

# The server template

Your R code typically goes inside a `render*` function that creates output to be displayed in the user interface.

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

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

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white')
    })
}
```

Sample R code

Input from user interface

# Updating the UI controls for our example

We want to create 1 slider input and 4 numeric inputs.

```
sidebarLayout(
   sidebarPanel(
      sliderInput("alpha", "Alpha:",
                  min = 0, max = 1,
                  value = 0.5, step = 0.1),
      numericInput("xlim1", "xlim lower:",
                  min = 0, max = max(homes$FinSqFt),
                  value = 0, step = 1e2),
      numericInput("xlim2", "xlim upper:",
                  min = 0, max = max(homes$FinSqFt),
                  value = max(homes$FinSqFt), step = 1e2),
      numericInput("ylim1", "ylim lower:",
                  min = 0, max = max(homes$TotalValue),
                  value = 0, step = 1e3),
      numericInput("ylim2", "ylim upper:",
                  min = 0, max = max(homes$TotalValue),
                  value = max(homes$TotalValue), step = 1e3)
   ),
```

# Updating the server for our example

We copy-and-paste our code into the `renderPlot` function and update the alpha, xlim and ylim arguments. Notice I chose to save the result as `output$scatterPlot`.

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

  output$scatterPlot <- renderPlot({
    ggplot(homes, aes(x = FinSqFt, y = TotalValue)) +
      geom_point(alpha = input$alpha) +
      scale_x_continuous(labels = scales::comma) +
      scale_y_continuous(labels = scales::dollar) +
      coord_cartesian(xlim = c(input$xlim1, input$xlim2),
                      ylim = c(input$ylim1, input$ylim2))
  })
```

# Add data for the app

We also need to include the R code that reads in and prepares the homes data for plotting. Insert under `library(shiny)` but before we define the UI.

```
library(shiny)
library(tidyverse)
homes <- readRDS(url("http://people.virginia.edu/~jcf2d/data/albemarle_homes.rds"))
vars <- c("CHARLOTTESVILLE", "CROZET", "EARLYSVILLE",
          "KESWICK", "SCOTTSVILLE", "NORTH GARDEN")
homes <- homes %>%
  filter(city %in% vars & Bedroom > 0 & FullBath > 0)
homes$city <- droplevels(homes$city)

# Define UI for application that draws a histogram
ui <- fluidPage(
```

# Updating the UI output for our example

Two last steps:

1. in the UI section we update the `plotOutput` function to use `"scatterPlot"`, which we named our plot in the server section.

```
# Show plot
mainPanel(
   plotOutput("scatterPlot")
)
```

2. Update the plot title as follows: `titlePanel("TotalValue vs FinSqFt")`

Try out the app by clicking the Run App button.

# The finished app

TotalValue vs FinSqFt

Alpha:

0 0.5 1

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

xlim lower:

0

xlim upper:

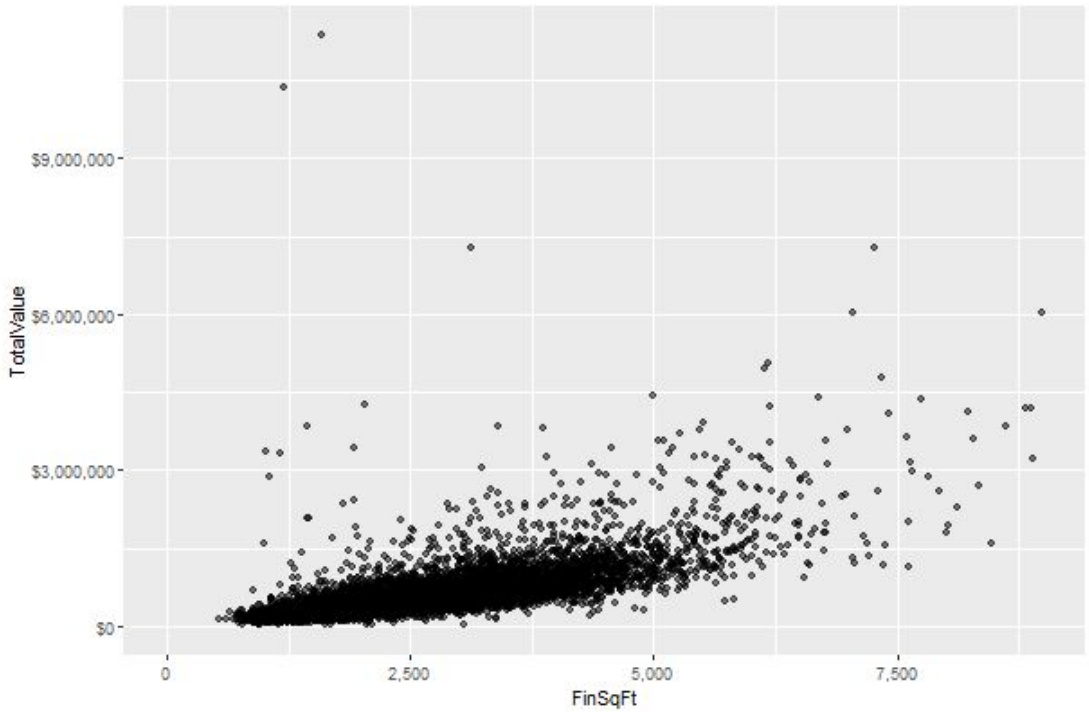
8982

ylim lower:

0

ylim upper:

11352300



# Congratulations on your first shiny app!

That's how the development of your first few shiny apps will proceed:

1. recognize R code that could be presented as an interactive app
2. start a new shiny web app using RStudio
3. tweak the template to work with your R code

You are developing an application. Expect it to take time and patience.

# Going beyond the RStudio template

The RStudio template is a great foundation for creating basic shiny apps. However shiny offers much more functionality than what is offered in the template.

Let's look at a few:

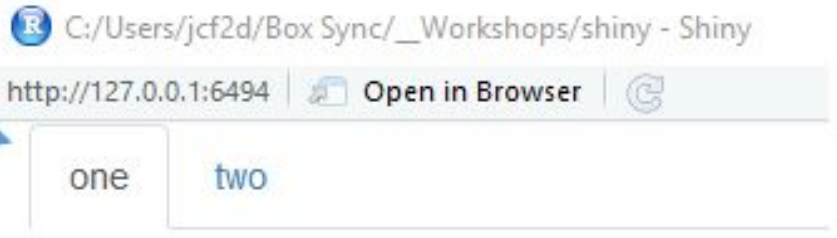
- use tabs so we can run multiple shiny apps in a single app
- use HTML to format the look of the app and/or add text
- modularize reactions so we can store values and reduce computation

# Multiple tabs

- Using `tabsetPanel` and `tabPanel` we can create multiple tabs, or pages, for our apps.
- Examples:
  - an app on one tab and instructions on another tab
  - multiple apps, each on their own tab
  - an app on one tab, data browser on another tab
- Today our goal is to build a shiny app with two tabs: one to display effect plots and another to calculate expected home values for given specifications.

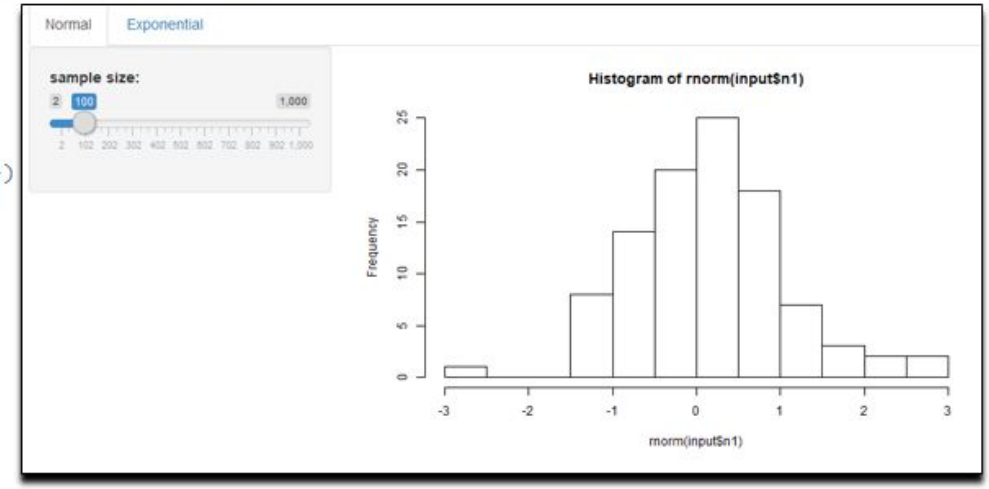
# Example: Minimal tab panel

```
# Minimal example
ui <- fluidPage(tabsetPanel(
  tabPanel("one"),
  tabPanel("two"))
  )

# Define server logic
server <- function(input, output){}

# Run the application
shinyApp(ui = ui, server = server)
```



# Example: Different apps on each tab

```
# Example with an app on each tab
ui <- fluidPage(tabsetPanel(
  tabPanel("Normal",
           sidebarLayout(
             sidebarPanel(
               sliderInput("n1",
                           "sample size:",min = 2,max = 1000,value = 100)),
             mainPanel(plotOutput("hist1")))
           ),
  tabPanel("Exponential",
           sidebarLayout(
             sidebarPanel(
               sliderInput("n2",
                           "sample size:",min = 2,max = 1000,value = 100)),
             mainPanel(plotOutput("hist2")))
           )
  )
  )

# Define server logic
server <- function(input, output) {
  output$hist1 <- renderPlot({hist(rnorm(input$n1))})
  output$hist2 <- renderPlot({hist(rexp(input$n2))})
}
```

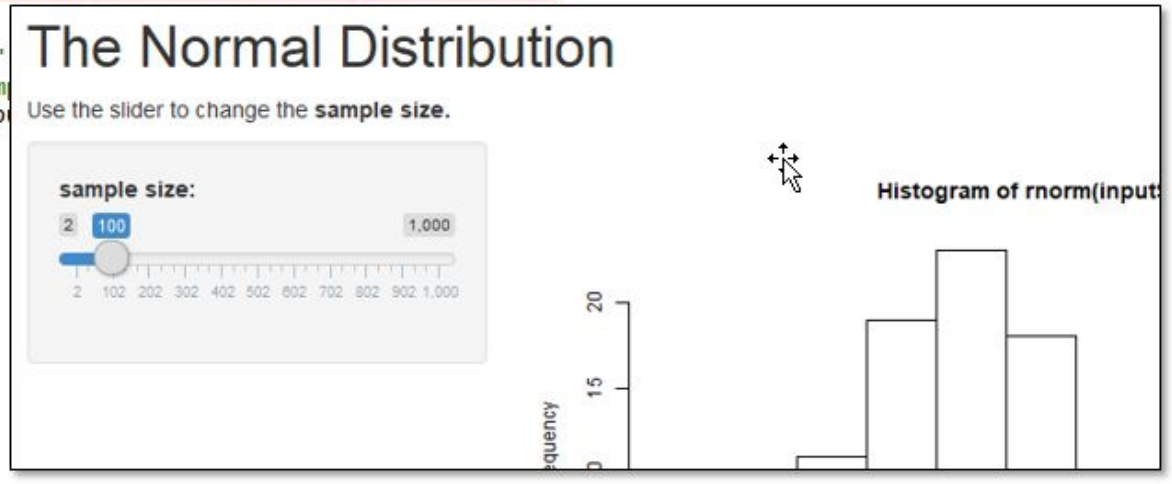


# Using HTML to add text and format look

- HTML is a markup language that web browsers use to render text
- Example: `this is <strong>bold</strong>` is rendered as "This is **bold**"
- We can add text to our apps and format the text with HTML
- Most common tags have wrapper functions
- Example: `This is strong("bold")` is rendered as "This is **bold**"

# Example: using HTML in shiny app

```
ui <- fluidPage(tabsetPanel(
  tabPanel("Normal",
           h1("The Normal Distribution"),
           p("Use the slider to change the", strong("sample size.")),
           sidebarLayout(
             sidebarPanel(
               sliderInput("n1"
                           "sam
             mainPanel(plotOutp
```



# Modular reactions

- Let's say we want to sample data from a Normal distribution, plot a histogram, and then try different bin widths.
- We could do this:

```
output$hist1 <- renderPlot({hist(rnorm(input$n1),
                                 breaks = input$bins)})
```

- But changing the bin widths (`breaks`) would cause `renderPlot` to execute again and generate a new sample.
- We would like to *modularize* the call to `rnorm` so it only runs when we change the sample size and not the bin width.

# Create modular reactions with the `reactive` function

- The `reactive` function creates a *reactive expression*
- It caches (or saves) its value and can be called by other code
- Example:

```
re <- reactive({n <- rnorm(input$n1)})
output$hist1 <- renderPlot({hist(re(),
                                 breaks = input$bins)})
```

- The reactive expression `re()` runs when the sample size changes and saves the result to `n`
- `renderPlot` uses the reactive expression to generate the histogram

# Our final shiny app

Let's apply what we learned to build a shiny app that allows us to explore a complex linear model based on the Albemarle County homes data.

Click here to access the workshop exercise.

Or go to https://at.virginia.edu/2SS695y

# Where to host Shiny apps

To put your Shiny app on the web, it needs to be hosted on a Shiny server. Three options are available:

1. Deploy to the cloud: http://www.shinyapps.io/ Free and paid options available
2. Deploy on-premises (open source): Shiny Server
3. Deploy on-premises (commercial): Shiny Server Pro

Options 2 and 3 require setting up your own server. There are some articles on the web about how to do it with Amazon Web Services (AWS).

# Resources

Written and video tutorials:
https://shiny.rstudio.com/tutorial/

Gallery of example shiny apps:
https://shiny.rstudio.com/gallery/

Building Shiny apps - an interactive tutorial
https://deanattali.com/blog/building-shiny-apps-tutorial/

# Thanks for coming today!

For help and advice with your statistical analysis:
statlab@virginia.edu

Sign up for more workshops or see past workshops:
http://data.library.virginia.edu/training/

Register for the Research Data Services newsletter to stay up-to-date on RDS events and resources:
http://data.library.virginia.edu/newsletters/