NYC DATA SCIENCE ACADEMY

Introduction to Shiny 2

Outline

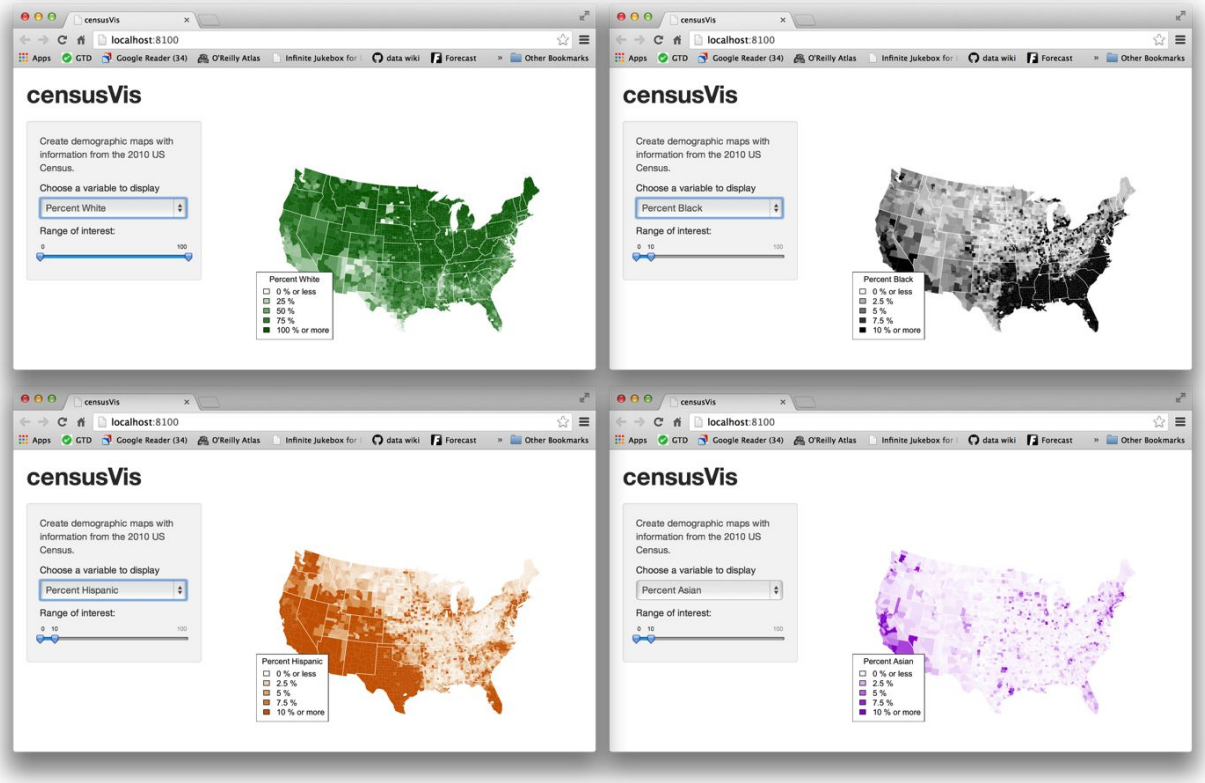
- Use R scripts, data and packages
- UI and server for the App
- Make your shiny perform quickly
- Use reactive expressions
- Share and deploy Shiny apps

Outline

- Use R scripts, data and packages
  - UI and server for the App
  - Make your shiny perform quickly
  - Use reactive expressions
  - Share and deploy Shiny apps

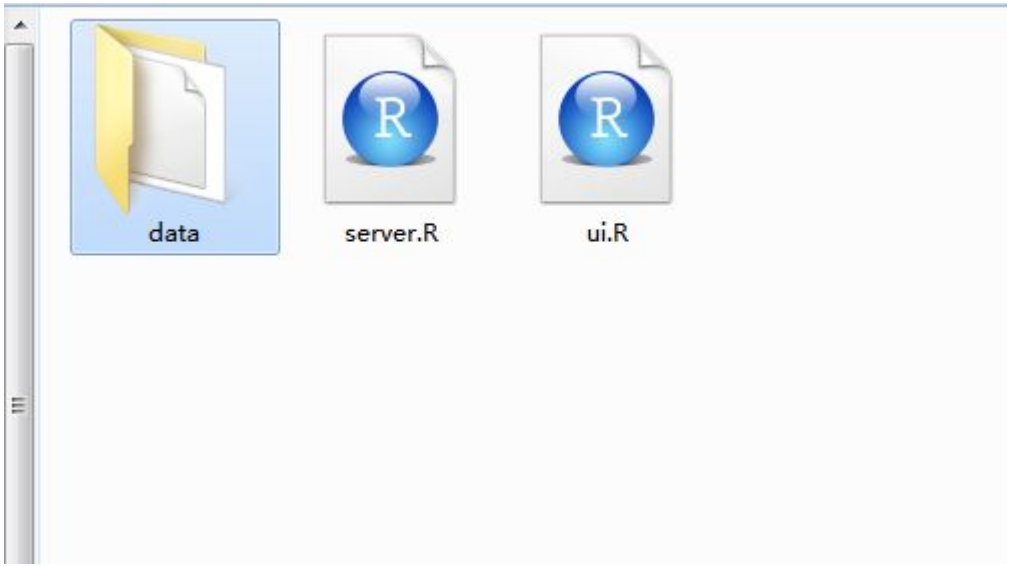
Use R Scripts, Data and Packages

- Including R scripts, data and packages to get shiny to visualize US Census data.



Use R Scripts, Data and Packages

- Load the data
  - Download the data here.
  - Create a folder called data in your app directory.
  - Put the data counties.rds in it.



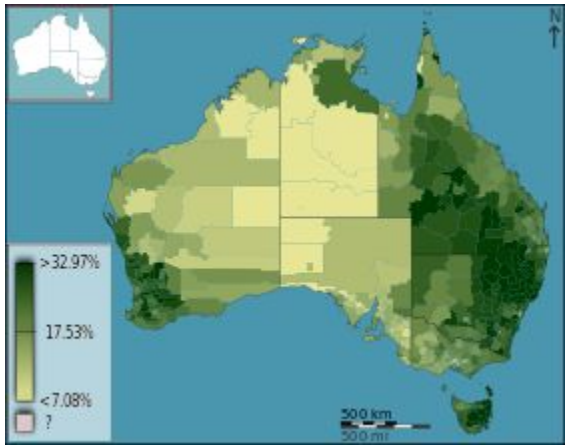
Use R Scripts, Data and Packages

- The data set in counties.rds contains:
  - The name of each county in the United States
  - The total population of the country
  - The percent of residents in the county who are white, black, hispanic, or asian

```
counties<-readRDS("case7/data/counties.rds")
head(counties)
##        name              total.pop     white     black     hispanic  asian
## 1     alabama,autauga    54571         77.2      19.3           2.4       0.9
## 2     alabama,baldwin    182265        83.5      10.9           4.4       0.7
## 3     alabama,barbour    27457         46.8      47.8           5.1       0.4
## 4     alabama,bibb       22915         75.0      22.9           1.8       0.1
## 5     alabama,blount     57322         88.9      2.5            8.1       0.2
## 6     alabama,bullock    10914         21.9      71.0           7.1       0.2
```

Choropleth Map

- A choropleth map is a map that uses color to display the regional variation of a variable.
- Areas are shaded or patterned in proportion to the measurement of the statistical variable.
- Example: population density or per-capita income.
  - wiki



Make a Choropleth Map in R

- Packages
  - Make sure you have maps and mapproj packages installed :

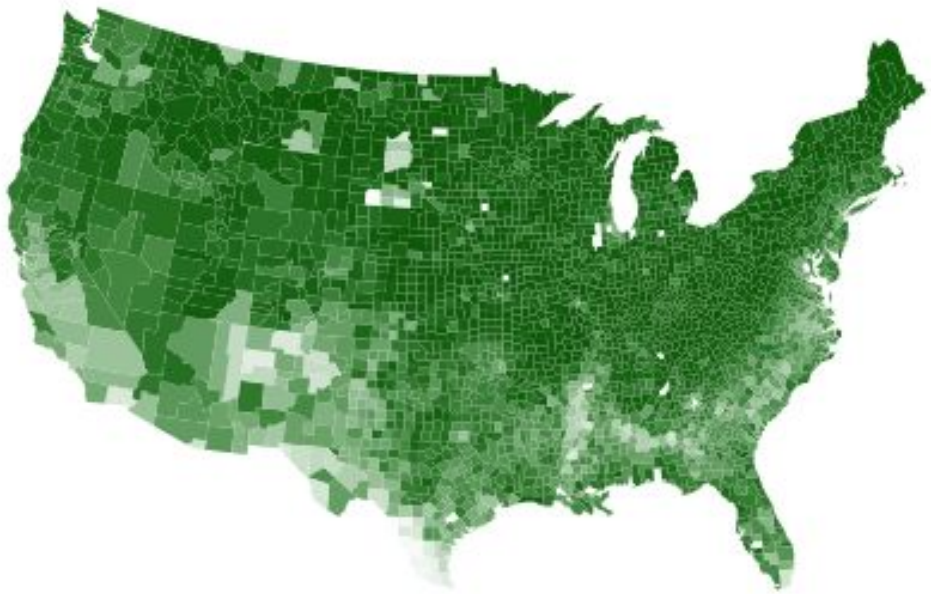
```
install.packages(c("maps","mapproj"))
```

- Map code

```
library(mapproj)
percents <- as.integer(counties$white)
shades <- colorRampPalette(c("white", "darkgreen"))(100)
fills <- shades[percents]
# plot choropleth map
map("county", fill = TRUE, col=fills,  resolution = 0, lty = 0,
    projection = "polyconic", myborder = 0, mar = c(0,0,0,0))
```

Make a Choropleth Map in R

- Map results
  - Not great. Let's make it better!



Make a Choropleth Map in R

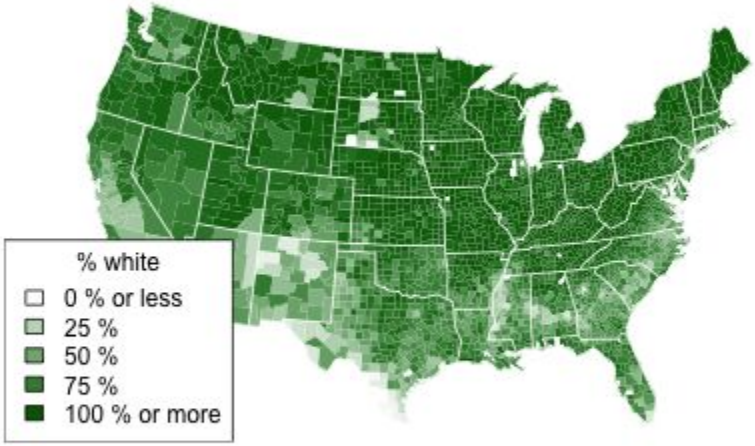
- helpers.R
  - helpers.R is an R script that can help you make choropleth maps.
  - helpers.R includes a function which can be used to draw a much more wonderful map.
  - The percent_map function :

ARGUMENT	INPUT
var	a column vector from the counties.rds dataset
color	any character string you see in the output of colors()
legend.title	A character string to use as the title of the plot's legend
max	A parameter for controlling shade range (defaults to 100)
min	A parameter for controlling shade range (defaults to 0)

Make a Choropleth Map in R

- Example for percent_map

```
library(maps)
source("case7/helpers.R")
counties <- readRDS("case7/data/counties.rds")
percent_map(counties$white, "darkgreen", "% white")
```



Code in helpers.R

- Step 1 Constrain gradient to percents
- Step 2 Plot choropleth map
- Step 3 Overlay state borders
- Step 4 Add the legend
- All code are in

```
percent_map<-function(var,color,legend.title,min=0,max=100) {
  ...
  }
```

Code in helpers.R

- Step 1: Constrain gradient to percents

```
# generate vector of fill colors for map
shades <- colorRampPalette(c("white", color))(100)

# constrain gradient to percents that occur between min and max
var <- pmax(var, min)
var <- pmin(var, max)
percents <- as.integer(cut(var, 100,
  include.lowest = TRUE, ordered = TRUE))
fills <- shades[percents]
```

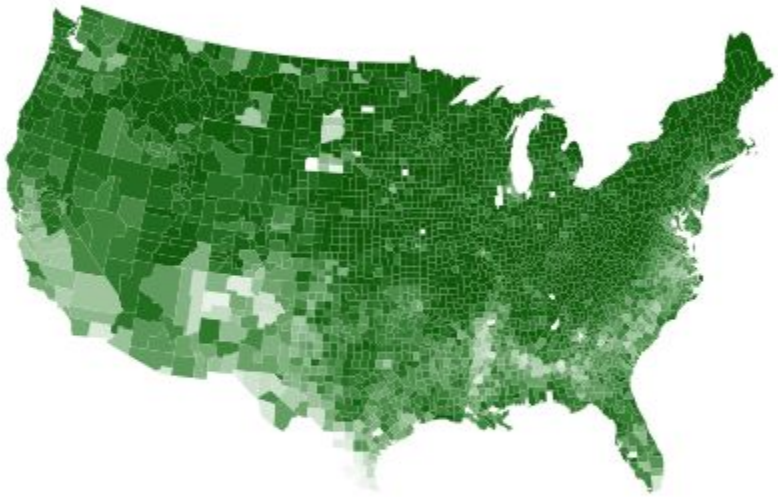
Code in helpers.R

- Step 2: Plot choropleth map

```
# plot choropleth map
map("county", fill = TRUE, col = fills, resolution = 0,
    lty = 0, projection = "polyconic",
    myborder = 0, mar = c(0,0,0,0))
```

Code in helpers.R

- Fig for Step 2



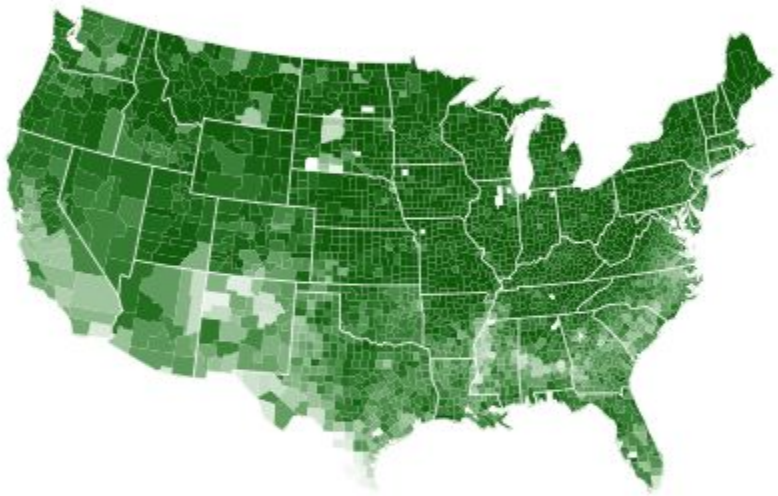
Code in helpers.R

- Step 3: Overlay state borders

```
# overlay state borders
map("state", col = "white", fill = FALSE, add = TRUE,
  lty = 1, lwd = 1, projection = "polyconic",
  myborder = 0, mar = c(0,0,0,0))
```

Code in helpers.R

- Fig for Step 3
  - Choropleth map with state borders



Code in helpers.R

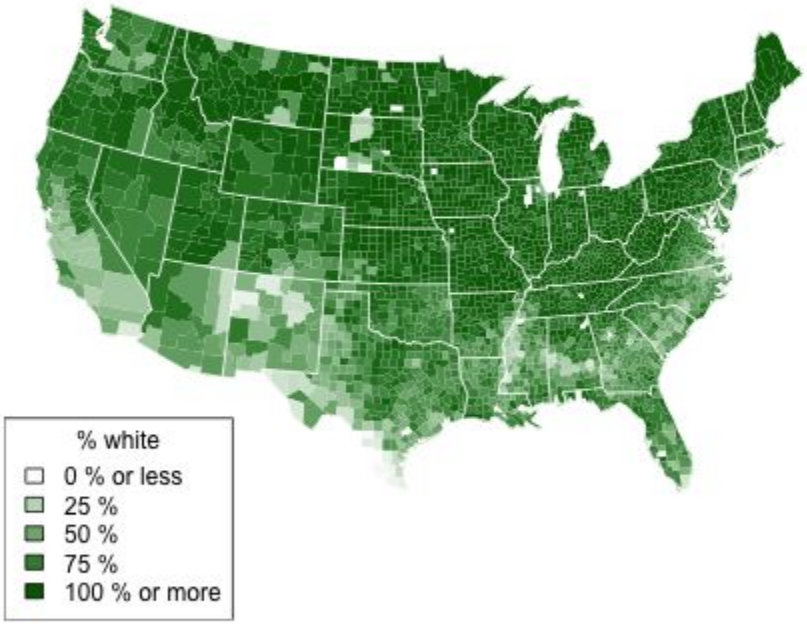
- Step 4: Add the legend

```
# add a legend
inc <- (max - min) / 4
legend.text <- c(paste0(min, " % or less"),
  paste0(min + inc, " %"),
  paste0(min + 2 * inc, " %"),
  paste0(min + 3 * inc, " %"),
  paste0(max, " % or more"))

legend("bottomleft",
  legend = legend.text,
  fill = shades[c(1, 25, 50, 75, 100)],
  title = legend.title)
```

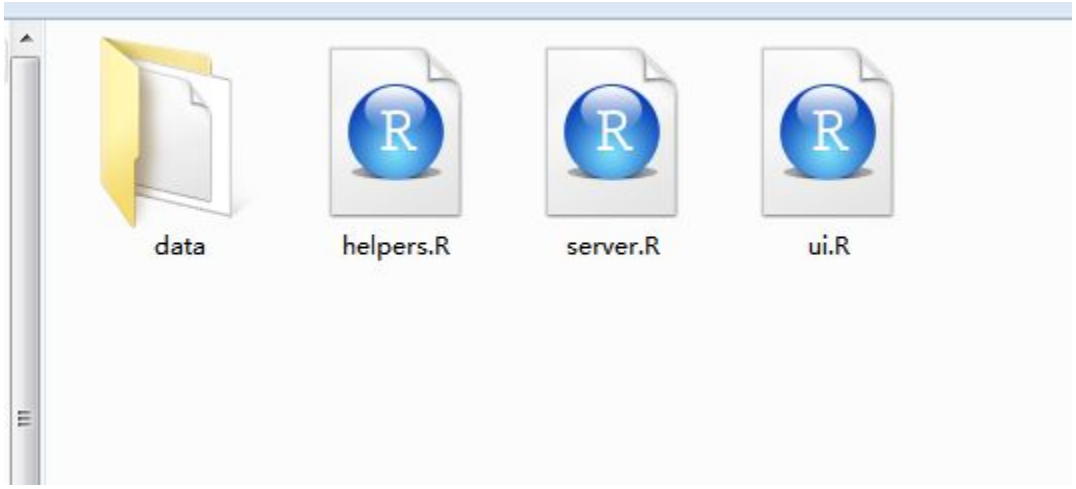
Code in helpers.R

- Final map



Summary for helpers.R

- Allows us to just call the percent_map function in our app.
- Put helpers.R in the same app directory with ui.R, server.R.



Outline

- Use R scripts, data and packages
- **UI and server for the App**
- Make your shiny perform quickly
- Use reactive expressions
- Share and deploy Shiny apps

UI

- There are two parts to input.
  - var as choose what kind of residents:
    - "Percent White"
    - "Percent Black"
    - "Percent Hispanic"
    - "Percent Asian"
  - range as a range for plotting (vector)

ui.R

```
shinyUI(fluidPage(
    titlePanel("censusVis"),   # main title
    sidebarLayout(
      sidebarPanel(
        helpText("Create demographic maps with
                 information from the 2010 US Census."), ## subtitle
        selectInput("var",  # choose the residents
                    label = "Choose a variable to display",
                    choices = c("Percent White", "Percent Black",
                                "Percent Hispanic", "Percent Asian"),
                    selected = "Percent White"),
        sliderInput("range",  # choose the range
                    label = "Range of interest:",
                    min = 0, max = 100, value = c(0, 100))
        ),
      mainPanel(plotOutput("map"))
    )
  ))
```

Server

- server.R including three parts:
  - Part1: Load the packages, scripts and the data.
  - Part2: Use switch function to build the args variables.
  - Part3: Use percent_map in helpers.R to plot the fig.
- You need renderPlot to output in server.R.

Server

- Part 1 : Load the packages, scripts and the data

```
# server.R

library(maps)
library(mapproj)
counties <- readRDS("case7/data/counties.rds")
source("case7/helpers.R")
```

Server

- Part 2 : Use switch function to build the args variable

```
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x))
}
x <- rnorm(10)
centre(x, "mean")
```

```
## [1] -0.1602302
```

```
centre(x, "median")
```

```
## [1] -0.2480295
```

Server

- Part 2 : Use switch function to build the args variable

```
args <- switch(input$var,
               "Percent White" = list(counties$white, "darkgreen", "% White"),
               "Percent Black" = list(counties$black, "black", "% Black"),
               "Percent Hispanic" = list(counties$hispanic, "darkorange", "%
Hispanic"),
               "Percent Asian" = list(counties$asian, "darkviolet", "% Asian"))

args$min <- input$range[1]
args$max <- input$range[2]
```

  - args include var, color, legend.title, min, max
  - It will be easy to use percernt_map() for mapping.

Server

- Part 3 : Use percent_map in helpers.R to plot the fig.
  - Function do.call(function,Args) is used for saved argument in list.
  - Constructs and executes a function call from function and a list of arguments (Args) to be passed to it.

```
do.call(percent_map, args)
## plot the map
```

Server

- Put part 2 and part 3 into shinyServer.

```
shinyServer(
  function(input, output) {
    output$map <- renderPlot({

      args <- switch(input$var,
                     "Percent White" = list(counties$white, "darkgreen", "%
White"),
                     "Percent Black" = list(counties$black, "black", "% Black"),
                     "Percent Hispanic" = list(counties$hispanic, "darkorange", "%
Hispanic"),
                     "Percent Asian" = list(counties$asian, "darkviolet", "%
Asian"))

      args$min <- input$range[1]
      args$max <- input$range[2]
      do.call(percent_map, args)
    })
  }
)
```

Final Shiny App

censusVis

Create demographic maps with information from the 2010 US Census.

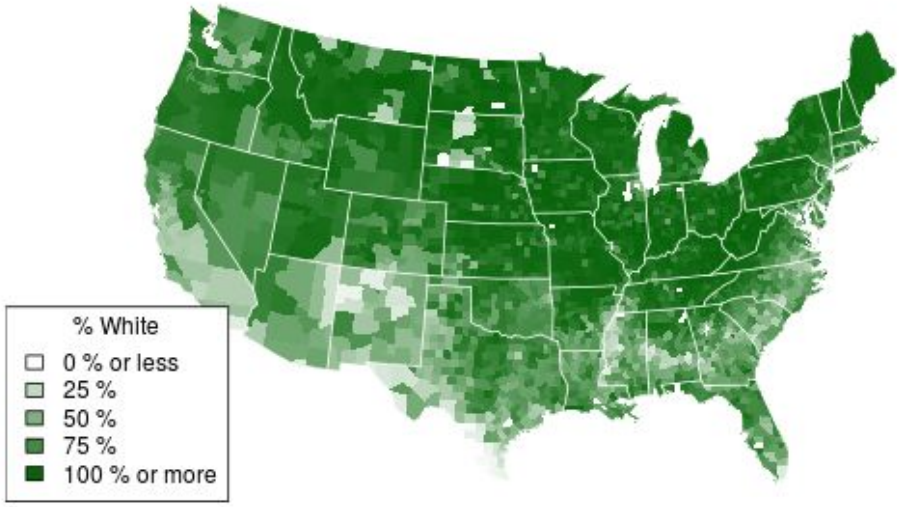
Choose a variable to display

Percent White

Range of interest:

0 100

0 10 20 30 40 50 60 70 80 90 100



Outline

- Use R scripts, data and packages
- UI and server for the App
- **Make your shiny perform quickly**
- Use reactive expressions
- Share and deploy Shiny apps

Execution

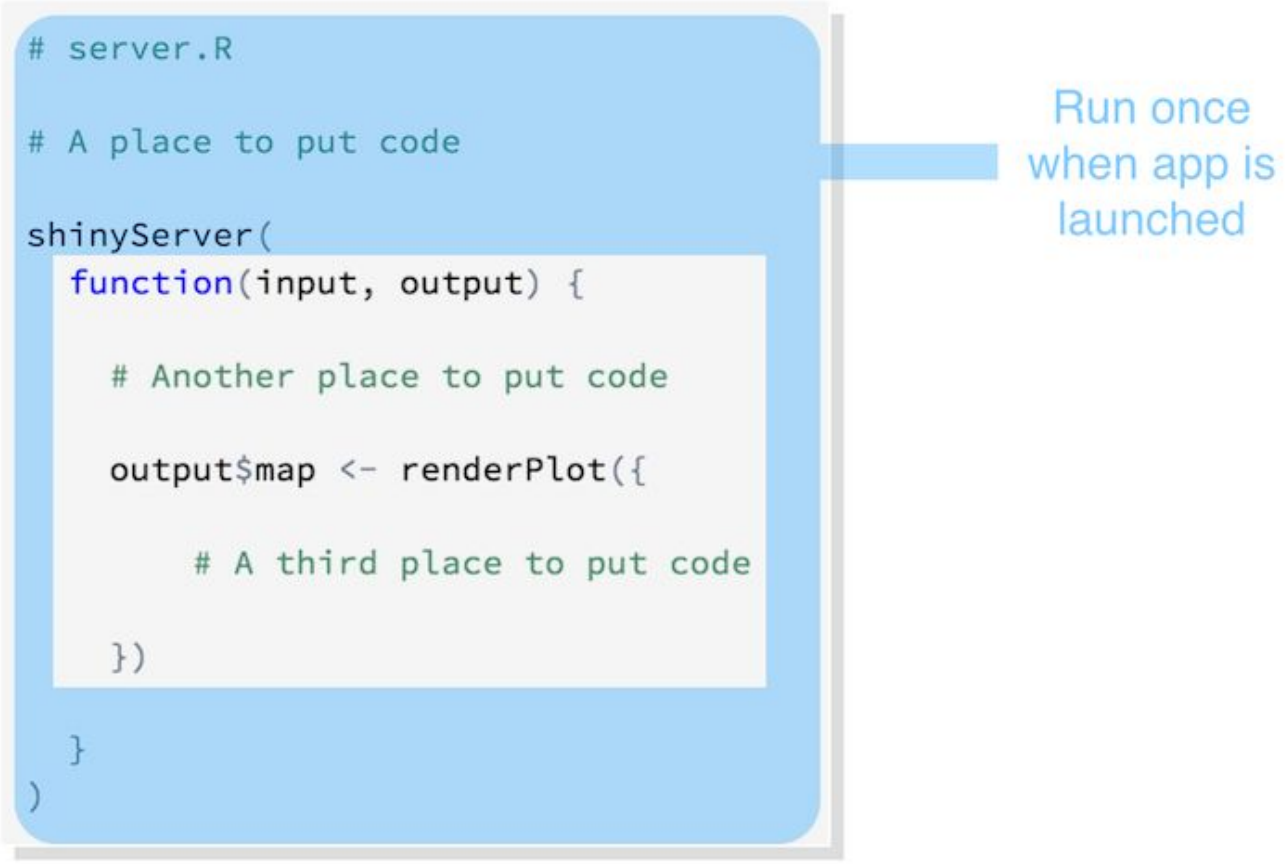
- Shiny will execute all commands in server.R. script.
- The commands where you place them in server.R determine how many times they are run (or re-run), which will in turn affect the performance of your app.
- Shiny will run some sections of server.R more often than others.

Execution

- When server.R is executed
  - When shiny is launched
  - When someone visits shiny
  - When the widget is changed

When Shiny is Launched

- Shiny will run the whole script the first time you call runApp.




```
# server.R

# A place to put code

shinyServer(
  function(input, output) {

    # Another place to put code

    output$map <- renderPlot({

        # A third place to put code

    })

  }
)
```

When Someone Visits Shiny

- Each time a new user visits your app, Shiny runs the unnamed function again, one time.

```
# server.R

# A place to put code

shinyServer(
  function(input, output) {

    # Another place to put code

    output$map <- renderPlot({

        # A third place to put code

    })

  }
)
```

Run once each time a user visits the app

When A Widget Is Changed

- As users change widgets, Shiny will re-run the R expressions assigned to each reactive object.

```
# server.R

# A place to put code

shinyServer(
  function(input, output) {

    # Another place to put code

    output$map <- renderPlot({

      # A third place to put code

    })

  }
)
```

Run each time a user changes a widget that output$map relies on

Summary

- The server.R script is run once,
  - when you launch your app
- The unnamed function inside shinyServer is run once
  - each time a user visits your app
- The R expressions inside render* functions are run many times.
  - each time a user changes a widget

What Should We Do?

- Source scripts, load libraries, and read data sets at the beginning of server.R outside of the shinyServer function.
- Define user specific objects inside shinyServer's unnamed function, but outside of any render* calls.
- Only place code that Shiny must rerun to build an object inside of a render* function.
- Avoid placing code inside a render* that does not need to be there .

Summary

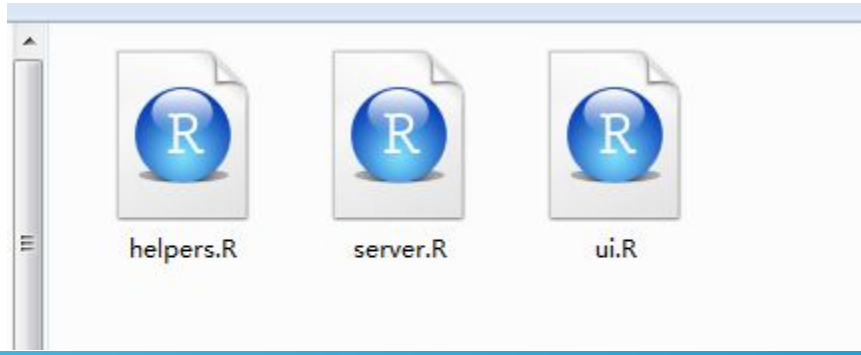
- You can create more complicated Shiny apps by loading R scripts, packages, and data sets.
- The directory that server.R appears in will become the working directory of the Shiny app.
- Shiny will run code placed at the start of server.R, before shinyServer, only once during the life of the app.
- Shiny will run code placed inside shinyServer multiple times, which can slow the app down.
- switch is a useful companion to multiple choice Shiny widgets.
- As your apps become more complex, they can become inefficient and slow.

Outline

- Use R scripts, data and packages
- UI and server for the App
- Make your shiny perform quickly
- **Use reactive expressions**
- Share and deploy Shiny apps

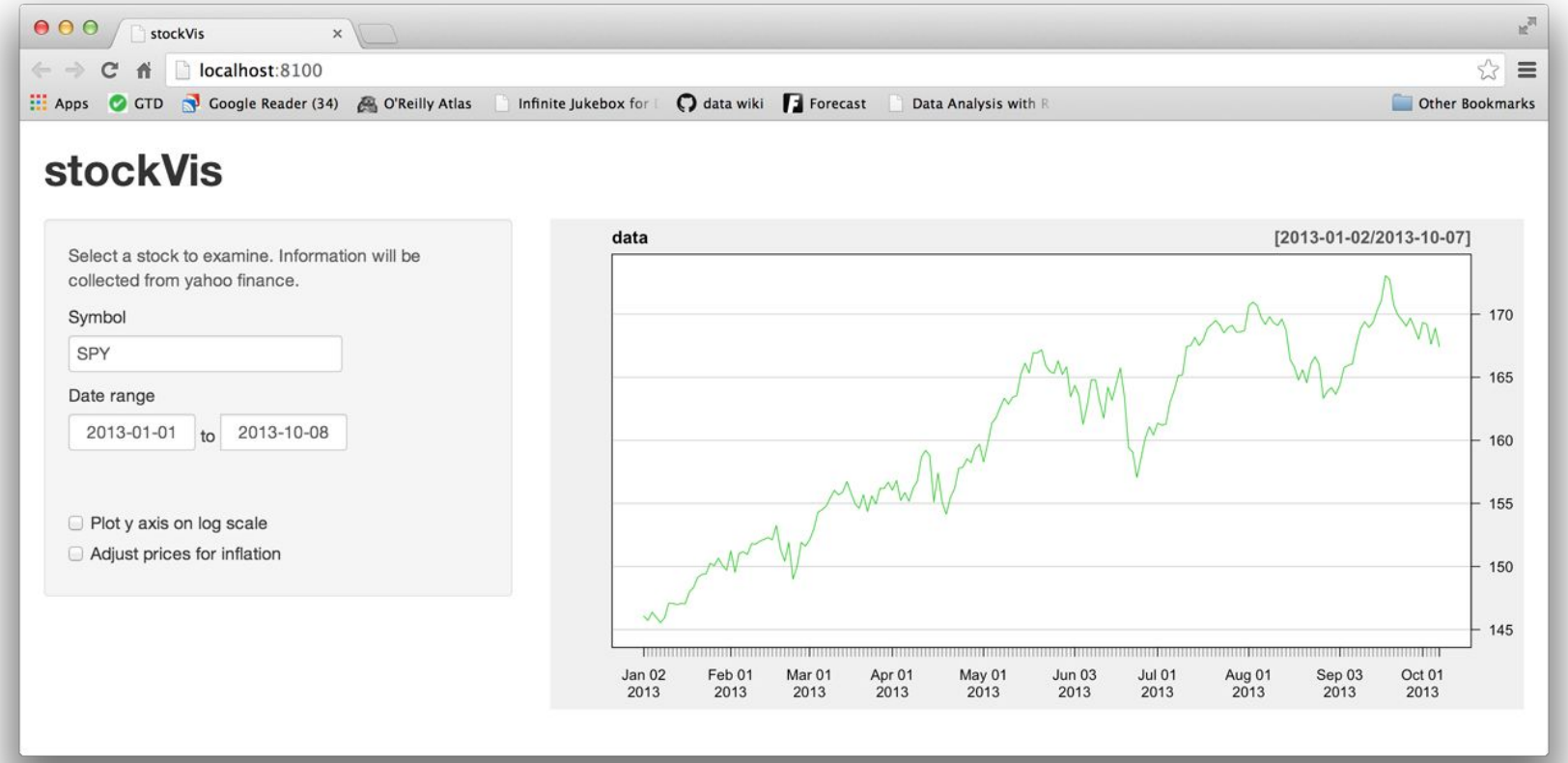
A New App: stockVis

- To get started:
  - Create a new folder named stockVis in your working directory.
  - Download the following files and place them inside stockVis:
    - ui.R
    - server.R
    - helpers.R
  - Launch the app with runApp("stockVis")
- **make sure the quantmod package is installed



A New App: stockVis

- Plot stock prices using shiny

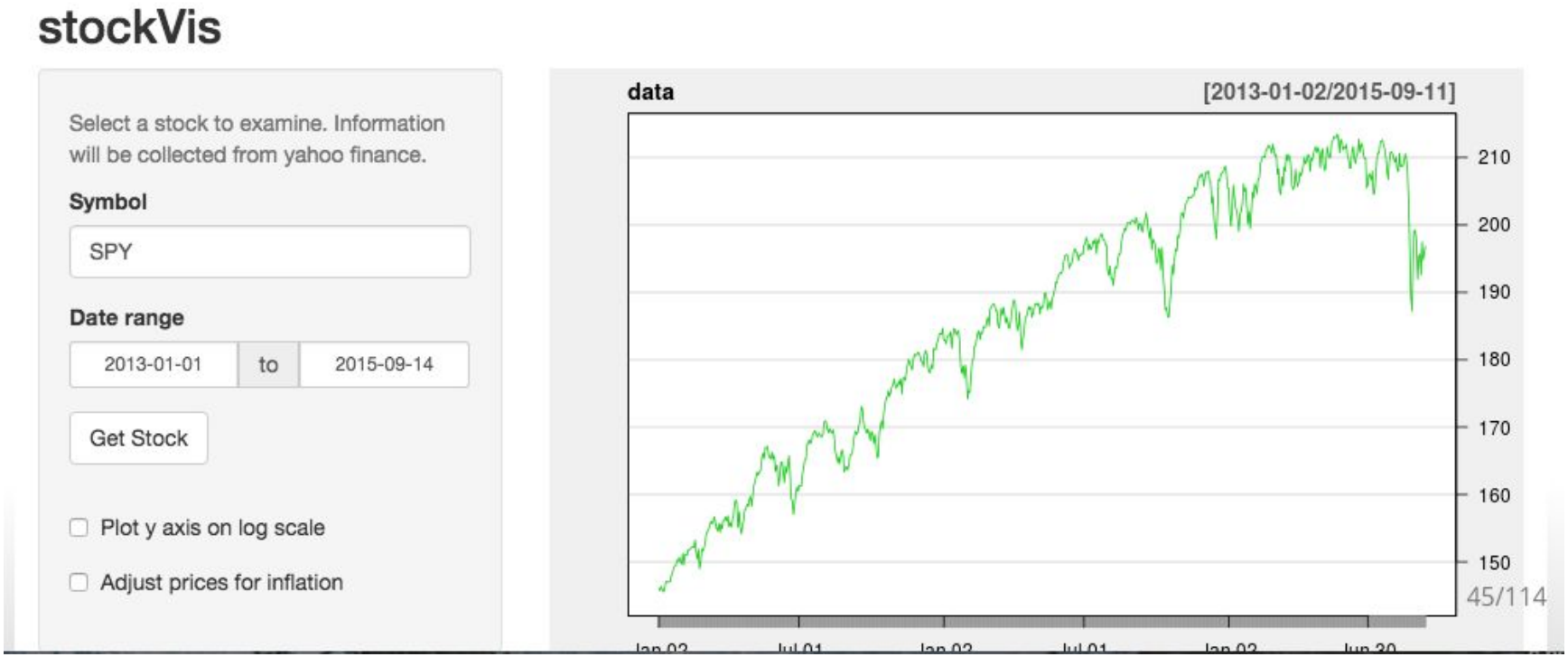


A New App: stockVis

- What can shiny do?
  - Select a stock to examine
  - Pick a range of dates to review
  - Choose whether to plot stock prices or the log of the stock prices on the y axis
  - Decide whether or not to correct prices for inflation.

A New App: stockVis

- Final app
- Note that the “Adjust prices for inflation” check box doesn’t work yet. One of our tasks in this section is to fix this check box.



A New App: stockVis

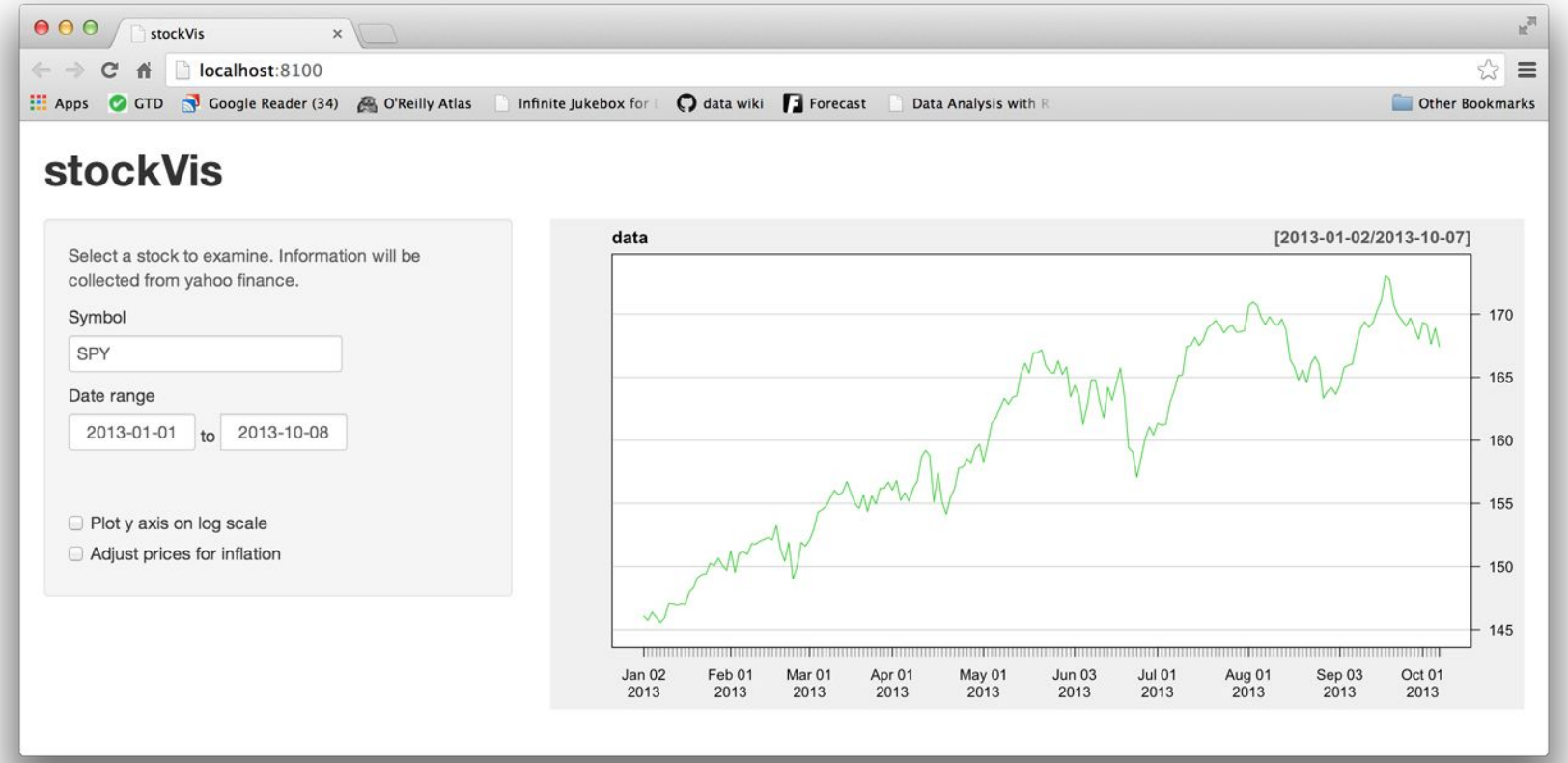
- The symbol
  - By default, stockVis displays the SPY ticker (an index of the entire S&P 500).
  - To look up a different stock, type in a stock symbol that Yahoo finance will recognize.
  - Some common symbols are GOOG (Google), AAPL (Apple), and GS (Goldman Sachs) etc.

A New App: stockVis

- To get the stock data, stockVis relies heavily on two functions from the quantmod package:
  - getSymbols: download financial data straight into R from websites like Yahoo finance and the Federal Reserve Bank of St. Louis.
  - chartSeries: display prices in an attractive chart.
  - stockVis also relies on an R script named helpers.R, which contains a function that adjusts stock prices for inflation.

Checkboxes and Date Ranges

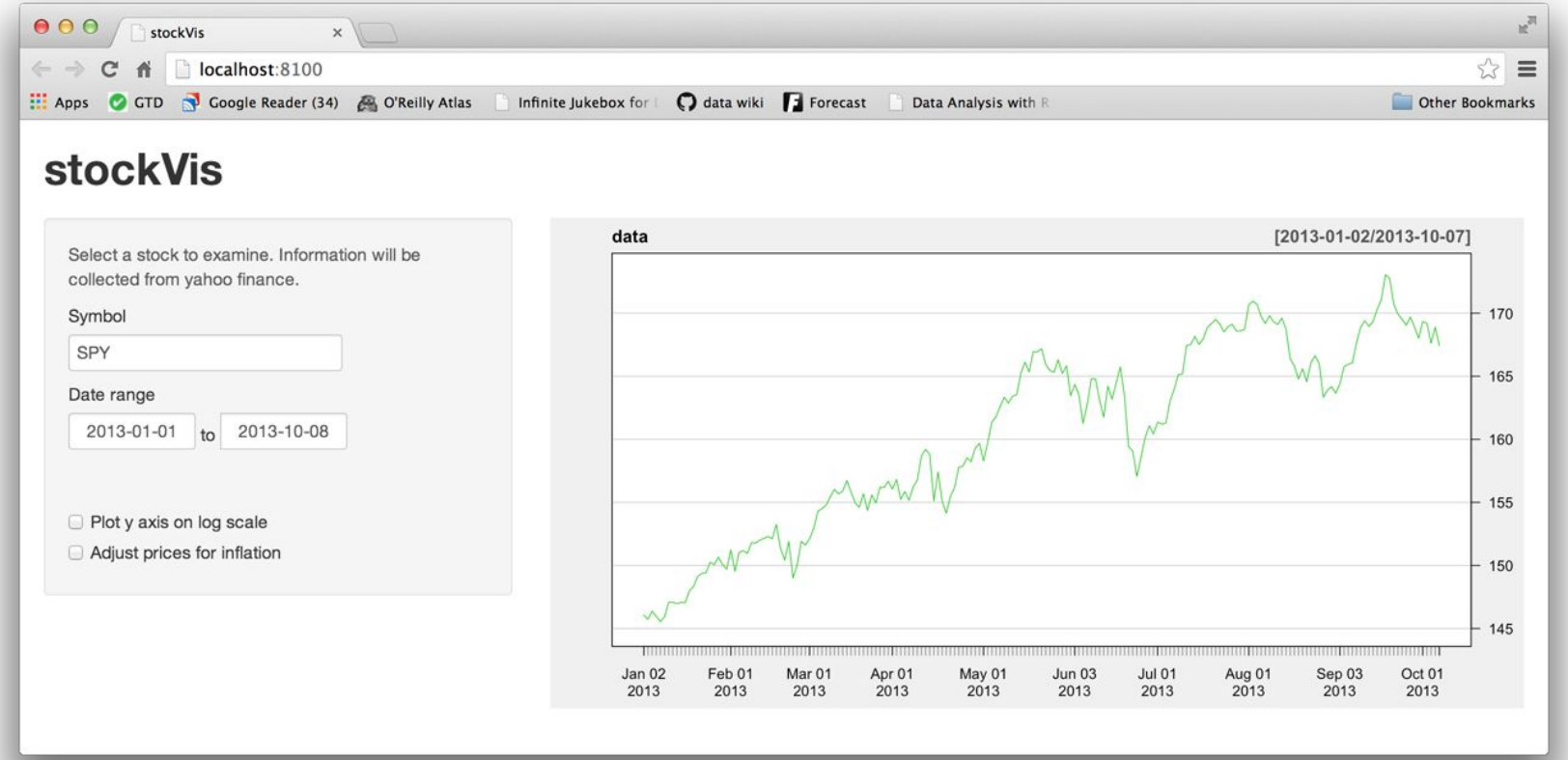
- The stockVis app uses a few widgets.
  - a date range selector, created with dateRangeInput



Checkboxes and Date Ranges

- A couple of check boxes made with checkboxInput.
- Checkbox widgets
  - return TRUE when the checkbox is checked.
  - return FALSE when the checkbox is not checked.
- The check boxes are named log and adjust in the ui.R script, which means you can look them up as input$log and input$adjust in the server.R script.

Checkboxes and Date Ranges



Checkboxes and Date Ranges

- ui.R

```
sidebarPanel(
   helpText("Select a stock to examine.
     Information will be collected from yahoo finance."),
   textInput("symb", "Symbol", "SPY"),
   dateRangeInput("dates",
     "Date range",
     start = "2013-01-01",
     end = as.character(Sys.Date())),
   actionButton("get", "Get Stock"),
   br(), # blank row
   br(),
   checkboxInput("log", "Plot y axis on log scale",
     value = FALSE),
   checkboxInput("adjust",
     "Adjust prices for inflation", value = FALSE)
 )
```

Checkboxes and Date Ranges

- ui.R
  - Put the sidebarPanel in it.

```
library(shiny)

shinyUI(fluidPage(
  titlePanel("stockVis"),

  sidebarLayout(
    sidebarPanel(.... ),

    mainPanel(plotOutput("plot"))
  )
))
```

Checkboxes and Date Ranges

- helpers.R
- Provide a function adjust to correct data for inflation
  - The adjust function in helpers.R uses the Consumer Price Index data provided by the Federal Reserve Bank of St. Louis to transform historical prices into present day values.

```
if (!exists(".inflation")) {
  .inflation <- getSymbols('CPIAUCNS', src = 'FRED',
     auto.assign = FALSE)
}
# adjusts yahoo finance data with the monthly consumer price
index values provided by
#the Federal Reserve of St. Louis historical prices are returned
in present values
```

Checkboxes and Date Ranges

- helpers.R

```
adjust <- function(data) {
      latestcpi <- last(.inflation)[[1]]
      inf.latest <- time(last(.inflation))
      months <- split(data)
      adjust_month <- function(month) {
        date <- substr(min(time(month[1]), inf.latest), 1, 7)
        coredata(month) * latestcpi / .inflation[date][[1]]
      }
      adjs <- lapply(months, adjust_month)
      adj <- do.call("rbind", adjs)
      axts <- xts(adj, order.by = time(data))
      axts[ , 5] <- Vo(data)
      axts
}
```

Reactive Expressions

- Use a reactive function to get stock data
- Use a reactive function to correct data for inflation
- Use a renderplot function to draw the plot

Reactive Expressions

- Limit what gets re-run with reactive expressions.
- reactive uses widget input and returns a value.
- To create a reactive expression use the reactive({})(just like the render* functions).

```
dataInput <- reactive({
    getSymbols(input$symb, src = "yahoo",
               from = input$dates[1],
               to = input$dates[2],
               auto.assign = FALSE)
  })
```

Why Use Reactive Functions

- If you use a renderPlot and put getSymbols in it
  - Re-runs every times when you change the widgets.
  - May let Yahoo finance cut you off if you re-fetch your data too often.
  - Most important, shiny will be very slow!

```
output$plot <- renderPlot({
  data <- getSymbols(input$symb, src = "yahoo",
    from = input$dates[1],
    to = input$dates[2],
    auto.assign = FALSE)

  chartSeries(data, theme = chartTheme("white"),
    type = "line", log.scale = input$log, TA = NULL)
})
```

Why Use Reactive Functions

- If you use a reactive expression
  - A reactive expression saves its result the first time you run it.
  - The next time the reactive expression is called, it checks if the saved value has become out of date (i.e., whether the widgets it depends on have changed).
  - If the value is out of date, the reactive object will recalculate it (and then save the new result).
  - If the value is up-to-date, the reactive expression will return the saved value without doing any computation.

Why Use Reactive Functions

```
 dataInput <- reactive({
    getSymbols(input$symb, src = "yahoo",
               from = input$dates[1],
               to = input$dates[2],
               auto.assign = FALSE)
  })
output$plot <- renderPlot({
  chartSeries(dataInput(), theme = chartTheme("white"),
    type = "line", log.scale = input$log, TA = NULL)
})
```

- renderPlot will call dataInput().
- dataInput will check that the dates and symb widgets have not changed.
- dataInput will return its saved data set of stock prices without re-fetching data from Yahoo.
- renderPlot will re-draw the chart with the correct axis.

Another Reactive

- It's important to use reactive to decide whether to adjust.

```
finalInput <- reactive({
    if (!input$adjust) return(dataInput())
    adjust(dataInput())
  })
```

Reactive Expressions

```
# server.R
library(quantmod)
source("stockVis/helpers.R")
shinyServer(function(input, output) {
  dataInput <- reactive({
    getSymbols(input$symb, src = "yahoo",
               from = input$dates[1],
               to = input$dates[2],
               auto.assign = FALSE)
  })
  finalInput <- reactive({
    if (!input$adjust) return(dataInput())
    adjust(dataInput())
  })
  output$plot <- renderPlot({
    chartSeries(finalInput(), theme = chartTheme("white"),
                type = "line", log.scale = input$log, TA = NULL)
  })
})
```

Summary

- To make shiny faster!
  - Reactive expressions save their results, and will only re-calculate if their input has changed.
  - Create reactive expressions with reactive({ }).
  - Only call reactive expressions from within other reactive expressions or render* functions.

Outline

- Use R scripts, data and packages
- UI and server for the App
- Make your shiny perform quickly
- Use reactive expressions
- **Share and deploy Shiny apps**

Share Your Shiny Apps

- Two basic methods to share your app
- 1.Share your Shiny app as two files: server.R and ui.R
  - The simplest way to share an app.
  - It works only if your users have R and shiny on their own computer.
- 2.Share your Shiny app as a web page
  - Your users can navigate to your app through the internet with a web browser.
  - They will find your app fully rendered, up to date, and ready to go.

Share Your Shiny Apps

- Share apps as two R files
- Three methods to share apps as two R files:
  - GitHub repository & GitHub Gist
  - Zip file delivered over the web or local copy
  - Package

Share Your App Using Github

- If your project is stored in a git repository on GitHub, then others can download and run your app directly.
- The following command will download and run the app:

```
shiny::runGitHub('shiny_example', 'rstudio')
```

- In this example, the GitHub account is 'rstudio' and the repository is 'shiny_example'.
- Just replace them with your own account and repository name.

Share Your App Using Github

- Share your app as a Gist
  - Put your code on gist.github.com
    - a pasteboard service for sharing files offered by GitHub.
  - Copy and paste your server.R and ui.R files to the Gist web page.
    - Example:
    - https://gist.github.com/jcheng5/3239667

Share Your App Using Github

- Share your app as a Gist
  - Launch your app using the following command:

```
shiny::runGist('<gist number>')
```

  - gist number appears at the end of your Gist's web address.
  - Or, you can use the entire URL of the gist.
  - e.g. 'https://gist.github.com/jcheng5/3239667'

Share Your App Using Github

- Pros
  - Version control
  - Easy to run (for R users)
  - Very easy to post and update if you already use GitHub for your project
  - GitHub users can clone and fork your repository
- Cons
  - The app code and data will be exposed to the public

Share Your App as an R Package

- If your Shiny app is useful to a broader audience, it might be worth the effort to turn it into an R package.
- Put your Shiny app directory under the package's inst directory, then create and export a function that contains something like this:

```
shiny::runApp(system.file('appdir', package='packagename'))
```

- where appdir is the name of your app's subdirectory in inst, and packagename is the name of your package.

Share Your App as an R Package

- Pros
  - Publishable on CRAN/Bioconductor
  - Easy to run (for R users)
- Cons
  - Requires knowing how to build an R package

Share Your App as a Zip File

- If you store a zip or tar file of your project on a web or FTP server, users can download and run it with a command like this:

```
shiny::runUrl('https://github.com/rstudio/shiny_example/archive/master.zip')
```

- The URL in this case is a zip file that happens to be stored on GitHub. Just replace it with the URL to your zip file.
- Another way is to simply zip up your project directory and send it to your recipient(s), where they can unzip the file and run it the same way you do.

```
shiny::runApp('~/unzipped/app/path/')
```

Share Your App as a Zip File

- Pros
  - Share apps using email, USB flash drive, or any other way you can transfer a file
- Cons
  - Updates to app must be sent manually

Share Your App as a Web Page

- Shinyapps.io: Turn your Shiny app into a web page
- RStudio's hosting service for Shiny apps.
- To get started with ShinyApps.io, you will need:
  - An R development environment, such as the RStudio IDE
  - (for Windows users only) RTools for building packages
  - (for Mac users only) XCode Command Line Tools for building packages
  - (for Linux users only) GCC
  - The devtools R package (version 1.4 or later)
  - The latest version of the shinyapps R package

Share Your App as a Web Page

- Install package
  - Make sure the packages devtools and shinyapps installed:

```
install.packages('devtools')
devtools::install_github('rstudio/shinyapps')
```

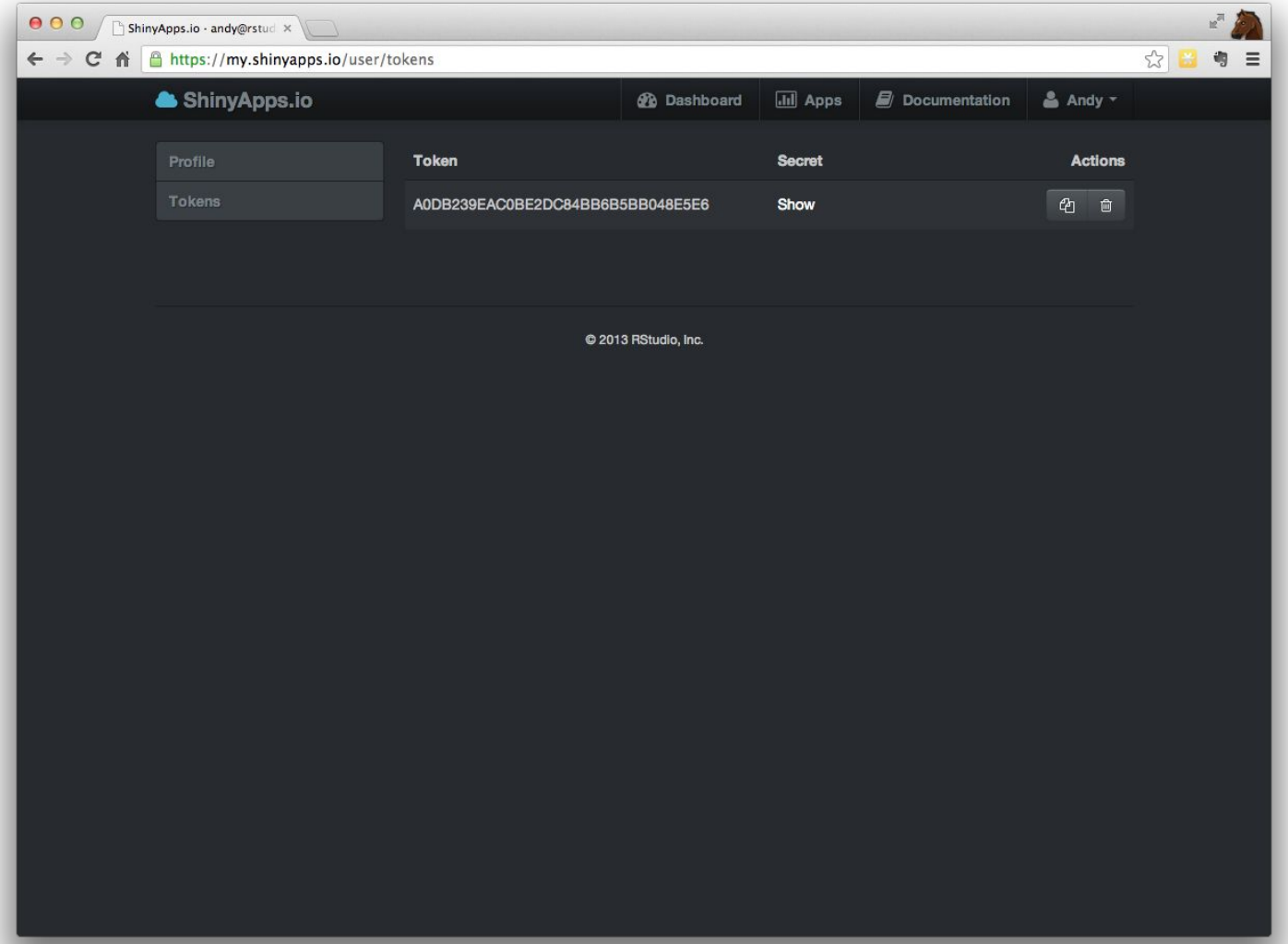
  - Once installed, load it into your R session:

```
library(shinyapps)
```

Share Your App as a Web Page

- Create account on ShinyApps.io
  - Go to shinyapps.io and click “Log In”
  - Set up your own account
  - shinyapps.io uses the account name as the domain name for all your apps.
- Configure shinyapps
  - shinyapps.io automatically generates a token and secret for you, which the shinyapps package can use it to access your account.
  - Retrieve your token from the ShinyApps.io dashboard.
  - Tokens are listed under Tokens in the menu at the top right of the ShinyApps dashboard.

Configure Shinyapps



Share Your App as a Web Page

- Configure shinyapps
- You can configure the shinyapps package to use your account with two methods:
  - Method 1
    - Click the copy button on the token page and paste the result into your R session.
  - Method 2
    - Run the setAccountInfo() function from the shinyapps package passing in the token and secret from the Profile / Tokens page.

```
shinyapps::setAccountInfo(name="<ACCOUNT>", token="<TOKEN>",
secret="<SECRET>")
```

Share Your App as a Web Page

- Deploy apps: test locally
  - Test your app works fine by running it locally.
  - Set your working directory to the app directory, and then run:

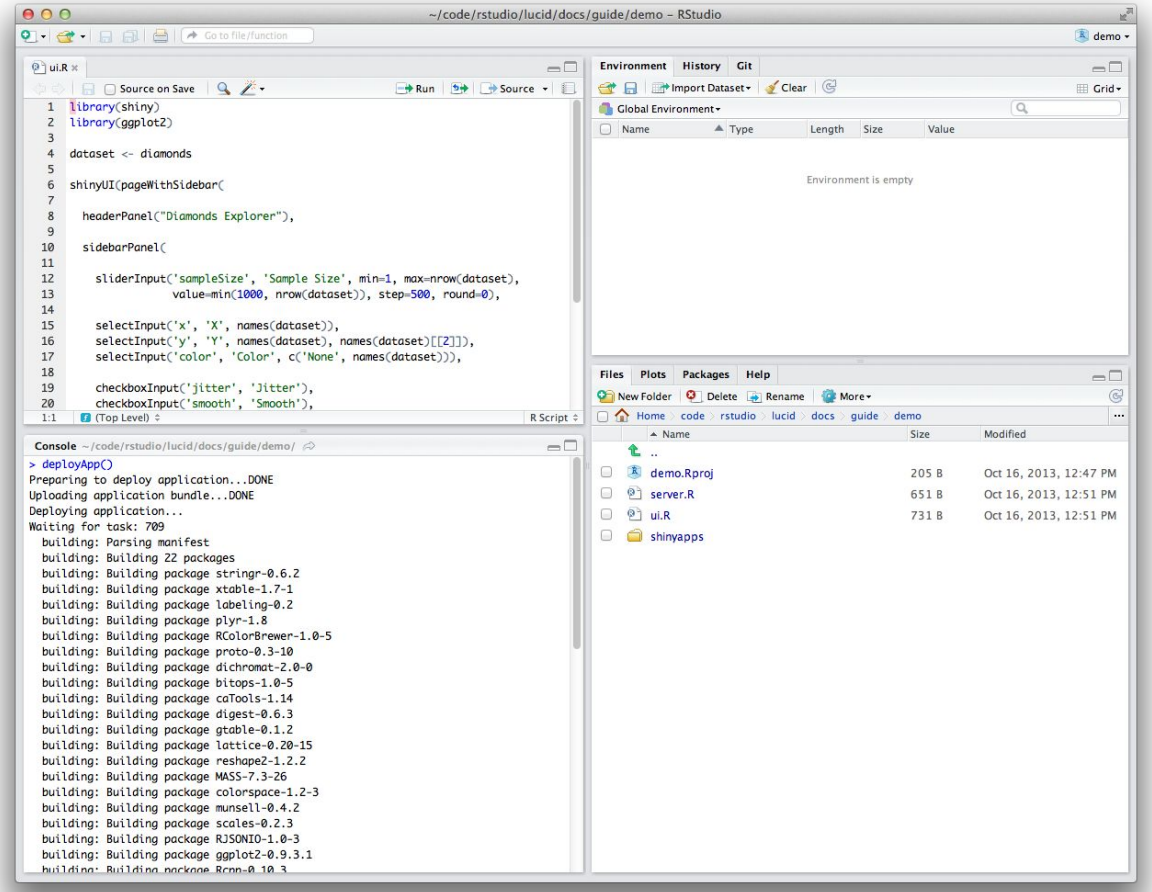
```
library(shiny)
runApp()
```

- Now that the application works, let's upload it to ShinyApps.io.
- Deploy apps
  - To deploy our Shiny app, use the deployApp() function from the shinyapps package.

```
library(shinyapps)
deployApp()
```

Share Your App as a Web Page

- Deploy apps



ShinyApps.io Resource Limitation

- ShinyApps.io limits the amount of system resources on an instance.
- The amount of resources available to an instance depends on its type.
- The table below outlines the various instance types and how much memory is allowed.

INSTANCE TYPE	MEMORY
small (default)	256 MB
medium	512 MB
large	1024 MB
xlarge	2048 MB
xxlarge	4096 MB

- By default, ShinyApps.io deploys all applications on 'small' instances, which are allowed to use 256 MB of memory.

ShinyApps.io Resource Limitation

- We can change the instance type used by an application with the configureApp function from the shinyapps package.
- To change the instance type of your application (here from small to medium), run:

```
library(shinyapps)
configureApp("appname", size="medium")
```

- This change will redeploy our app using the medium instance type.