

# Dashboards &



# Apps

# with



Mine Çetinkaya-Rundel @



@minebocek 

mine-cetinkaya-rundel 

mine@rstudio.com 

# Workshop materials

OPTION 1



[bit.ly/shiny-rlphl](https://bit.ly/shiny-rlphl)

log in and sit tight

OPTION 2



[bit.ly/shiny-rlphl-git](https://bit.ly/shiny-rlphl-git)

1. clone or download

2. launch rladies-phl-shiny.Rproj

A screenshot of a GitHub repository page for 'mine-cetinkaya-rundel'. At the top, there are buttons for 'Create new file', 'Upload files', 'Find file', and a green 'Clone or download' button. Below this, there are fields for 'Clone with SSH' (containing 'git@github.com:mine-cetinkaya-rundel') and 'Use HTTPS' (containing 'https://github.com/mine-cetinkaya-rundel'). At the bottom, there are 'Open in Desktop' and 'Download ZIP' buttons.



# Want just the slides?

[bit.ly/shiny-rlphl-slides](https://bit.ly/shiny-rlphl-slides)



# Meet & greet



Mine Çetinkaya-Rundel

Associate Professor, Duke Statistical Science

Data Scientist & Professional Educator, RStudio

# Overview

- 01 - Building dashboards with flexdashboard
- 02 - Getting started with shiny
- 03 - Understanding reactivity
- Lots of info!
- Lots of “your turn” breaks

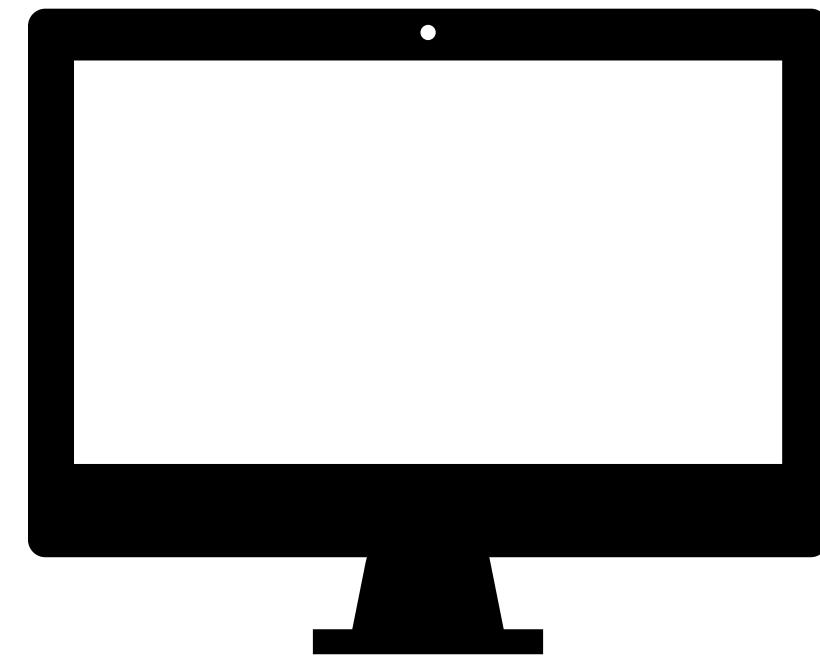


01

# Building dashboards with flexdashboard



gallery.shinyapps.io/un-women-dash



**DEMO**



# Dashboards

- Built in layouts and UI elements
- Good venue for displaying automatically updating data
- May or may not be interactive



# UI

- Static:
  - R code runs once and generates an HTML page
  - Generation of this HTML can be scheduled
- Dynamic:
  - Client web browser connects to an R session running on server
  - User input causes server to do things and send information back to client
  - Interactivity can be on client and server
  - Can update data in real time
- User potentially can do anything that R can do



# Building a dashboard



# 1. Set up the YAML

```
---
```

```
title: "UN Women Stats Explorer"
```

```
output:
```

```
  flexdashboard::flex_dashboard:
```

```
    orientation: rows
```

```
    social: menu
```

```
    source_code: https://github.com/mine-cetinkaya-rundel/r ladies-phl-shiny/blob/master/01-flexdash/un-
```

```
women-dash.Rmd
```

```
runtime: shiny
```

```
--
```

UN Gender Stats Explorer

Dashboard Data

  Source Code



## 2. Pick a layout

```
1 ---  
2 title: "Row Orientation"  
3 output:  
4   flexdashboard::flex_dashboard:  
5     orientation: rows  
6 ---  
7  
8 Row  
9 -----  
10  
11 ### Chart 1  
12  
13 `~~{r}  
14  
15 `~~  
16  
17 Row  
18 -----  
19  
20 ### Chart 2  
21  
22 `~~{r}  
23  
24 `~~  
25  
26 ### Chart 3  
27  
28 `~~{r}  
29  
30 `~~  
31
```

Chart 1

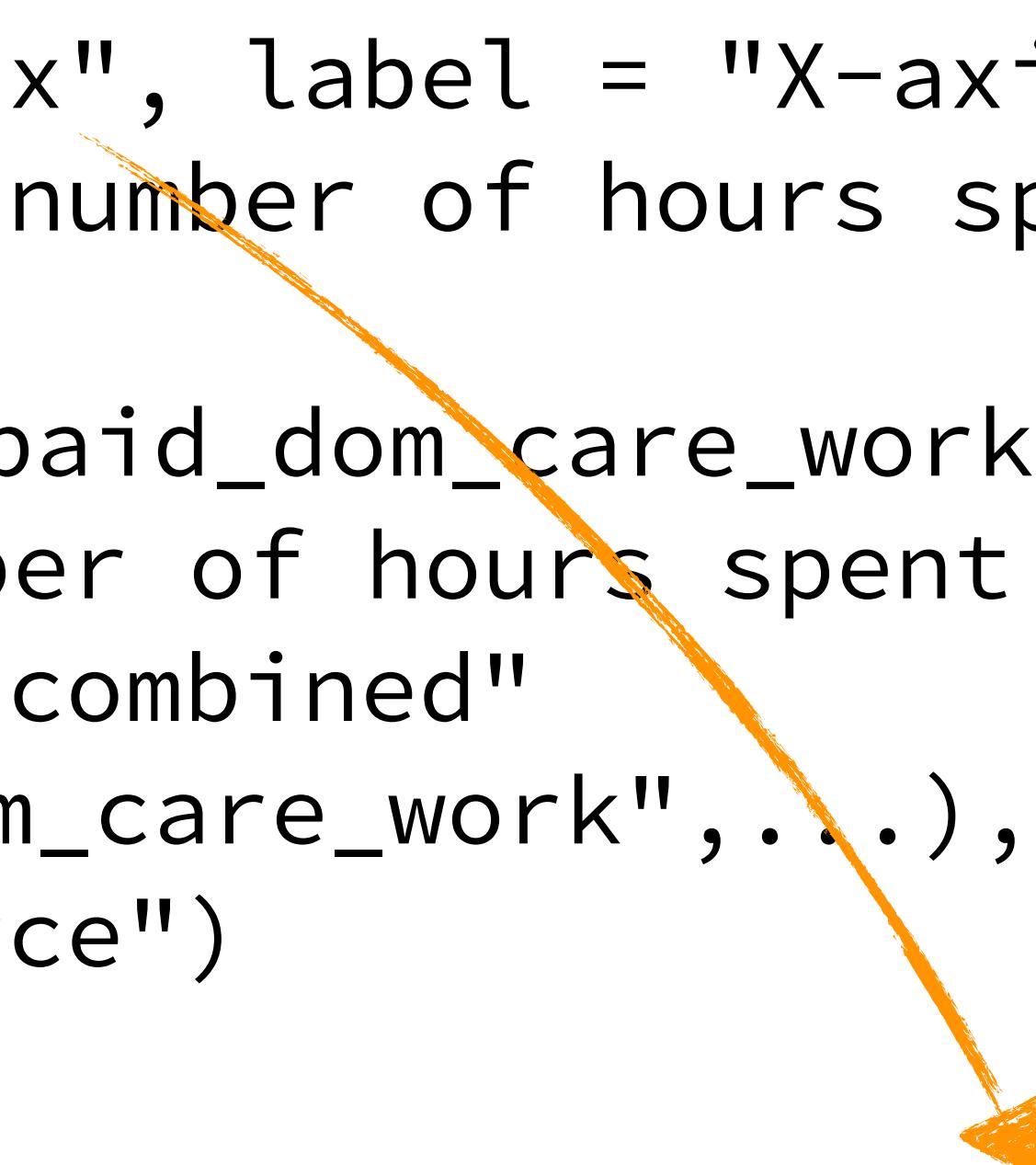
Chart 2

Chart 3



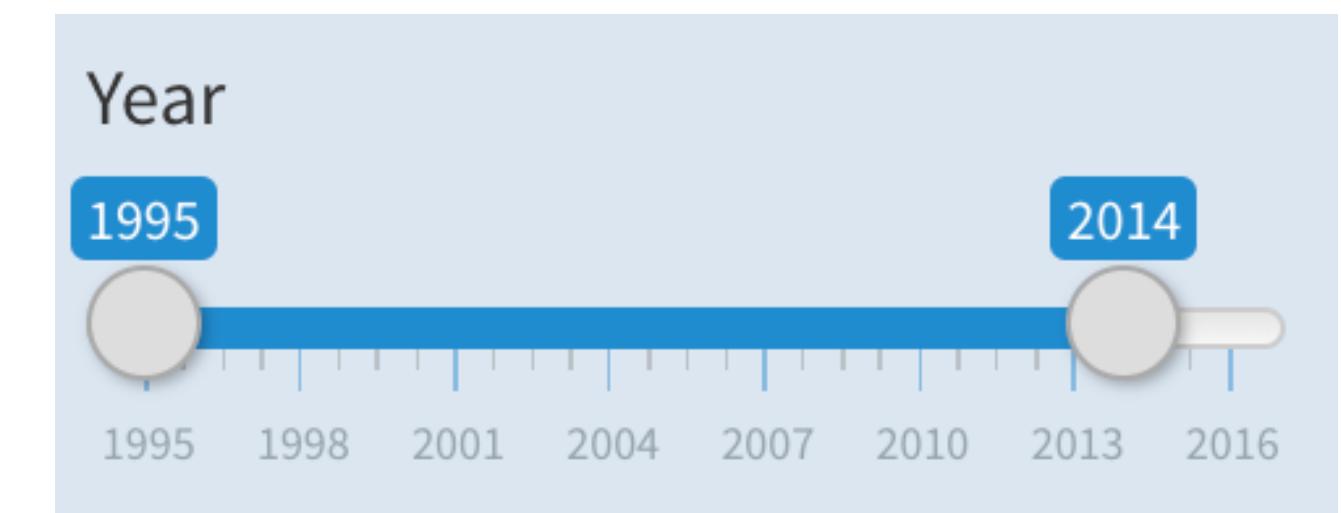
### 3. Use R Markdown and/or Shiny code to add components

```
selectInput(inputId = "x", label = "X-axis",
  choices = c("Average number of hours spent on unpaid domestic
and care work"
            = "hrs_unpaid_dom_care_work",
            "Average number of hours spent on paid and unpaid
domestic and care work combined"
            = "hrs_dom_care_work", ...),
  selected = "labor_force")  
  
renderPlot({  
  ggplot(data = sel_data(),  
         mapping = aes_string(x = input$x, y = input$y, color = "region")  
    geom_point(size = 2, alpha = 0.8) +  
    theme_minimal() +  
    labs(x = xlab(), y = ylab(), color = "Region")  
})
```



# Your turn

```
sliderInput(inputId = "year", label = "Year",  
            min = min_year, max = max_year,  
            value = c(2001, max_year), step = 1, sep = "")
```



- Open un-women-dash.Rmd
- Change the default selection of years to the `min_year` to 2014
- Run the app
- Select view mode in the drop down menu next to Run App to Preview in Viewer Pane
- Rerun the app

3m 00s



02

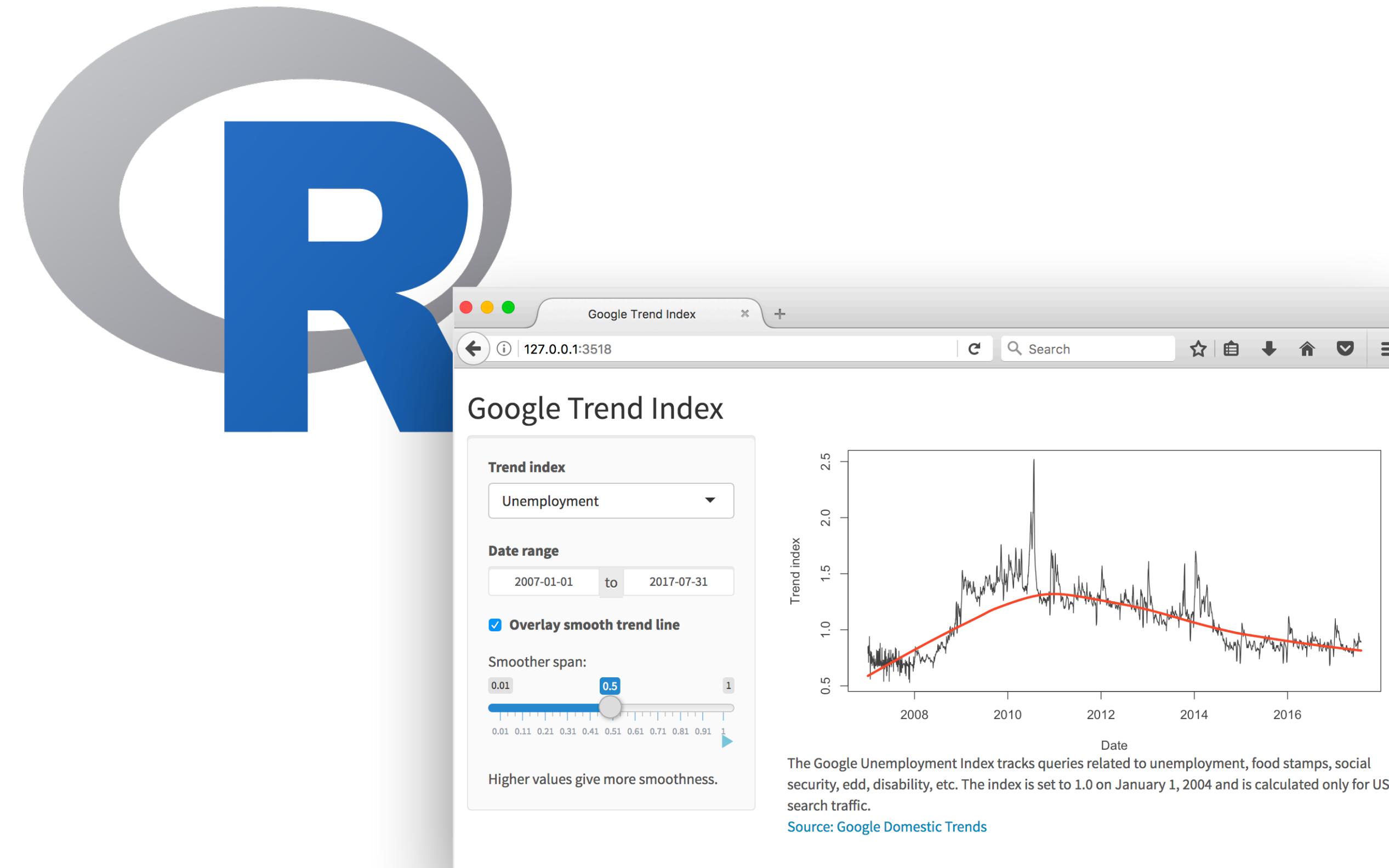
# Getting started with shiny



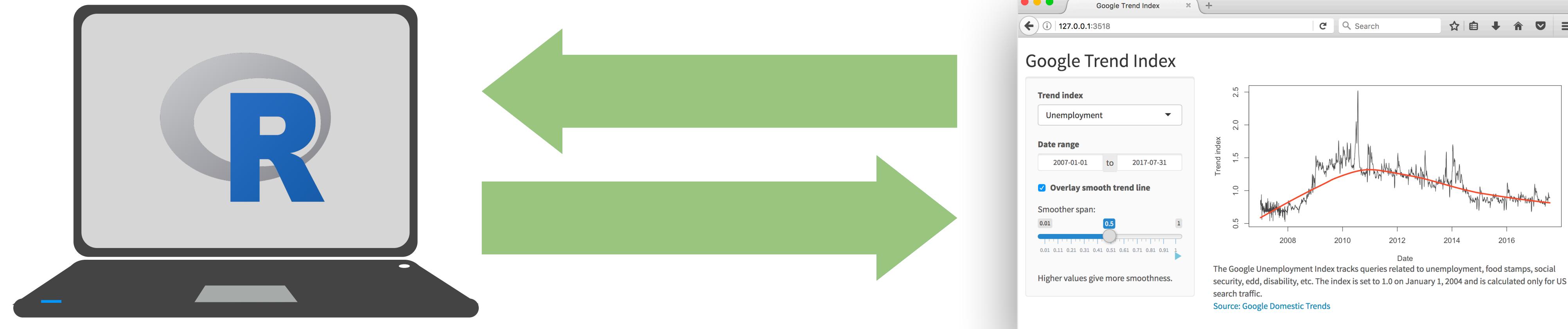
# High level view



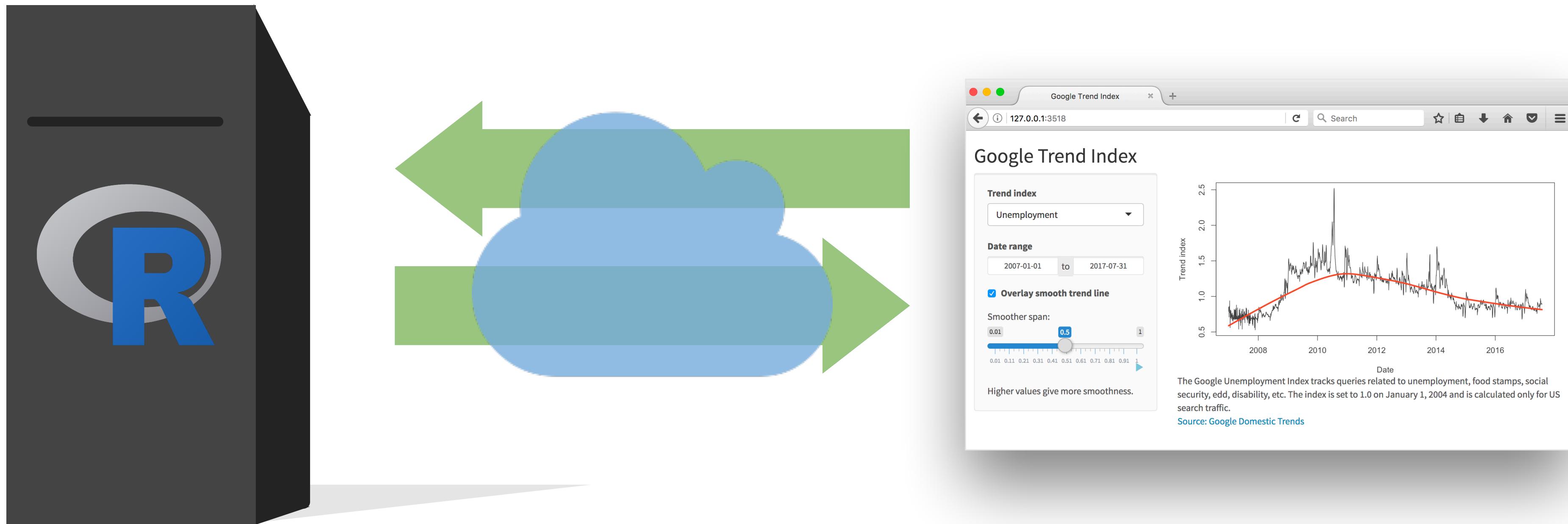
Every Shiny app has a webpage that the user visits,  
and behind this webpage there is a computer  
that serves this webpage by running R.

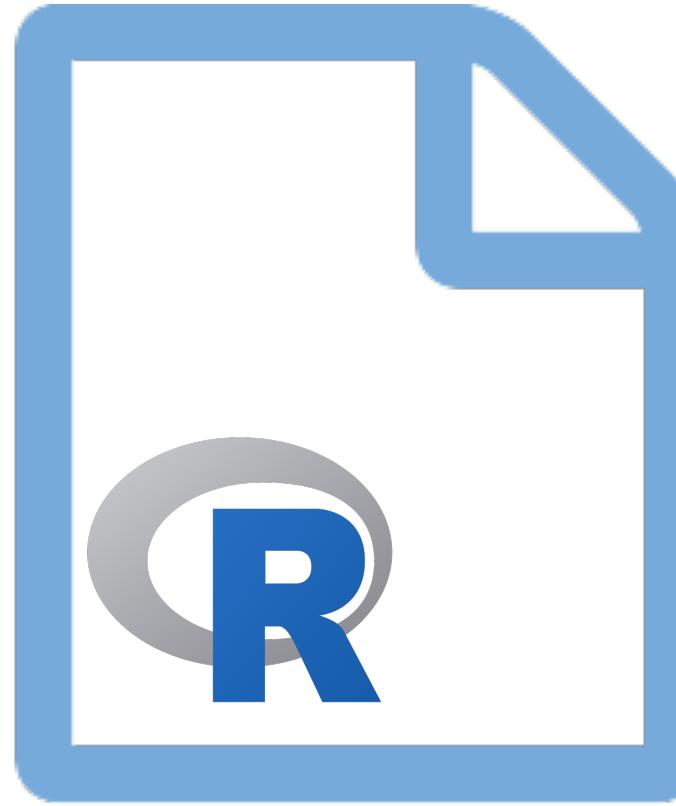
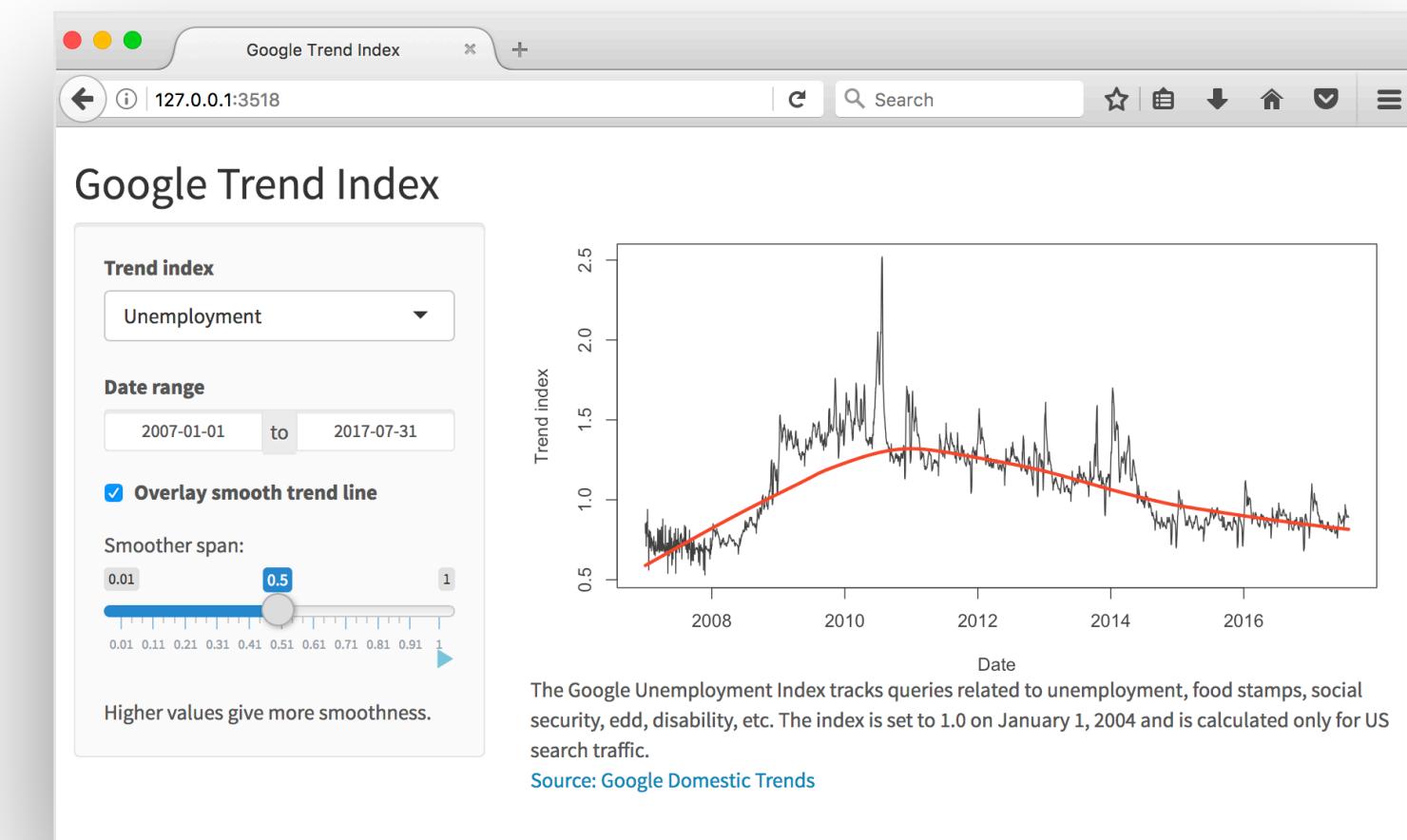


# When running your app locally, the computer serving your app is your computer.



When your app is deployed,  
the computer serving your app is a web server.





Server instructions



User interface



# Anatomy of a Shiny app



# What's in an app?

```
library(shiny)  
ui <- fluidPage()
```

## User interface

controls the layout and appearance of app

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

## Server function

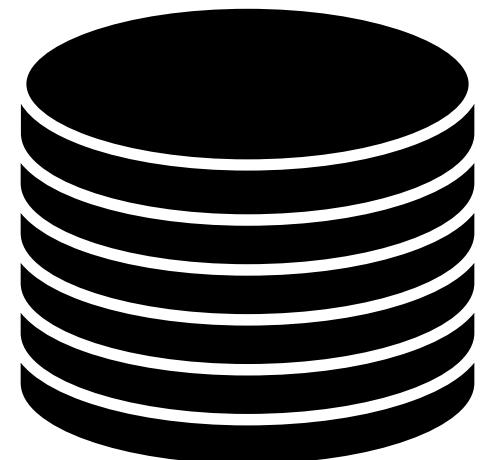
contains instructions needed to build app

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





## National Health and Nutrition Examination Survey



NHANES::NHANES

Data from the 2009 - 2010 and 2011 - 2012 surveys on  
10,000 participants and 76 variables collected on them

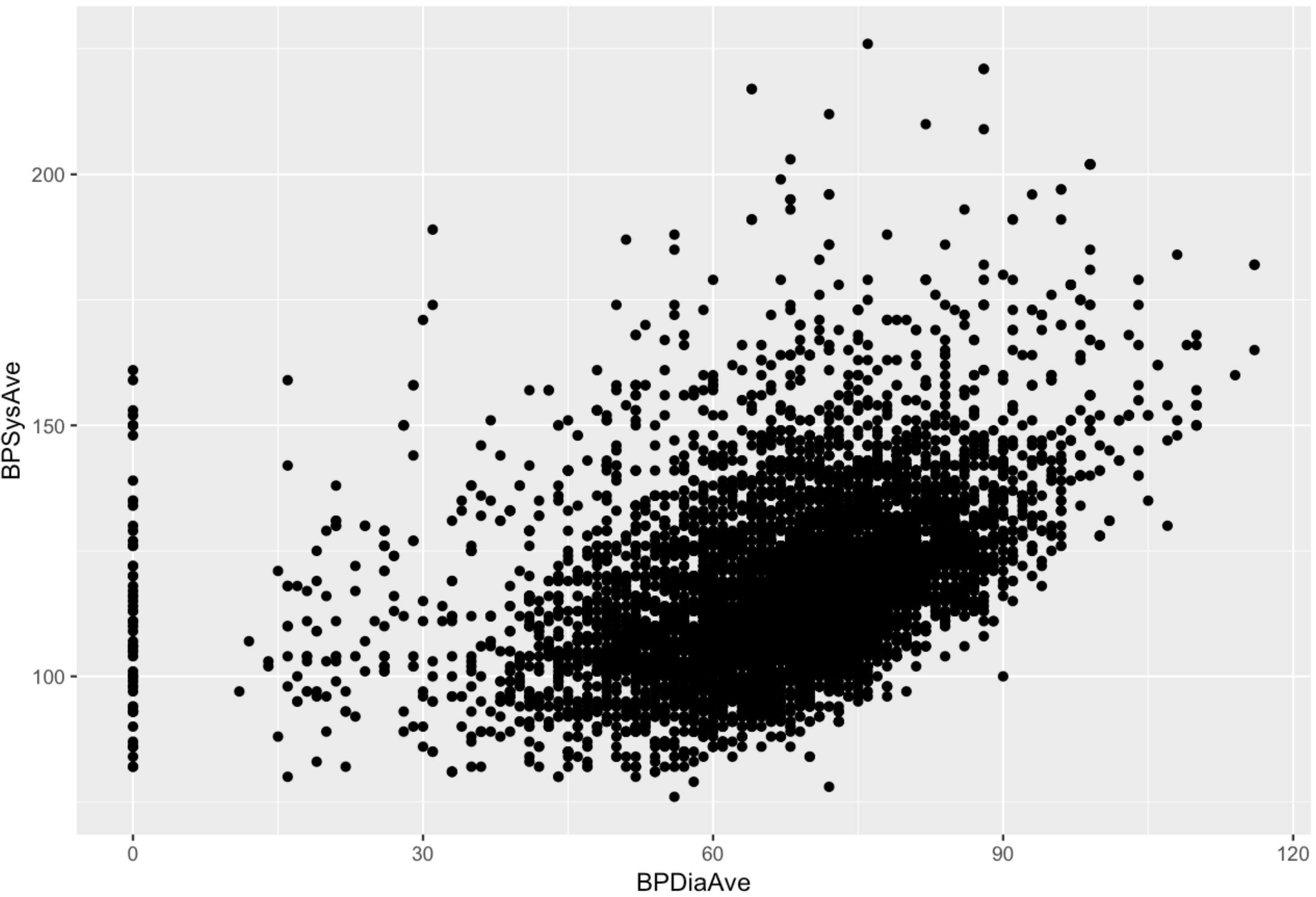


**Y-axis:**

BPSysAve

**X-axis:**

BPDiaAve



# App template

```
library(shiny)  
library(tidyverse)  
library(NHANES)  
  
ui <- fluidPage()  
  
server <- function(input, output) {}  
  
shinyApp(ui = ui, server = server)
```



# User interface



```
# Define UI
ui <- fluidPage(

  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPSysAve"),
                  selected = "BPSysAve"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPSysAve"),
                  selected = "BPDiaAve")
    ),

    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```



```
# Define UI
ui <- fluidPage(                                     Create fluid page layout
  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPSysAve"),
                  selected = "BPSysAve"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPDiaAve"),
                  selected = "BPDiaAve")
    ),
    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```



```
# Define UI
ui <- fluidPage(
  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPSysAve"),
                  selected = "BPSysAve"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPDiaAve"),
                  selected = "BPDiaAve")
    ),
    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```

Create a layout with a sidebar and main area

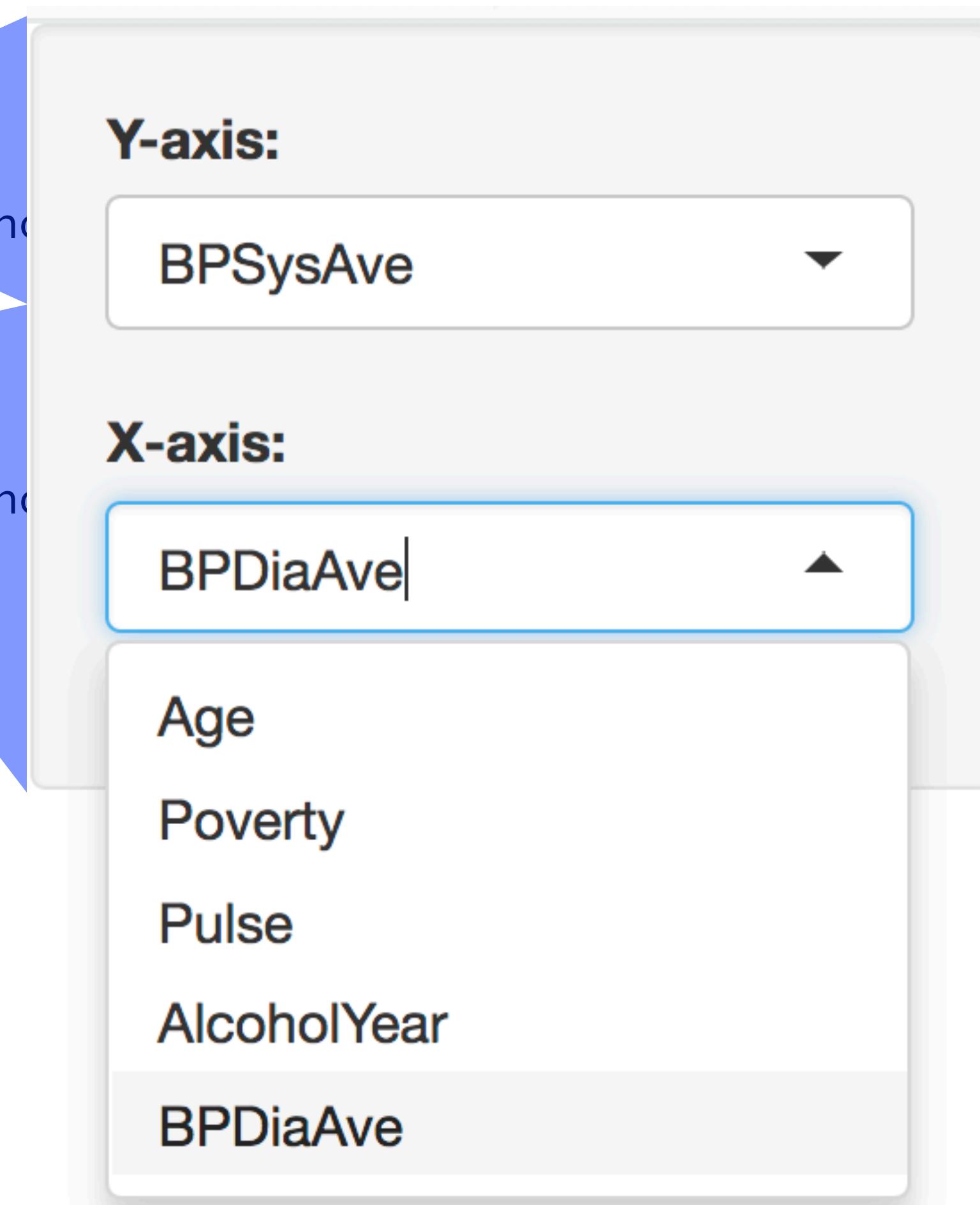


```
# Define UI
ui <- fluidPage(
  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPSysAve"),
                  selected = "BPSysAve"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPDiaAve"),
                  selected = "BPDiaAve")
    ),
    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```

Create a sidebar panel containing **input** controls that can in turn be passed to **sidebarLayout**



```
# Define UI
ui <- fluidPage(
  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear",
                             "BPDiaAve", "BPSysAve"),
                  selected = "BPSysAve"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear",
                             "BPDiaAve", "BPSysAve"),
                  selected = "BPDiaAve")
    ),
    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```



```
# Define UI
ui <- fluidPage(
  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPSysAve"),
                  selected = "BPSysAve"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("Age", "Poverty", "Pulse", "AlcoholYear", "BPDiaAve"),
                  selected = "BPDiaAve")
    ),
    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```

Create a main panel containing **output** elements that get created in the server function can in turn be passed to **sidebarLayout**



# Server

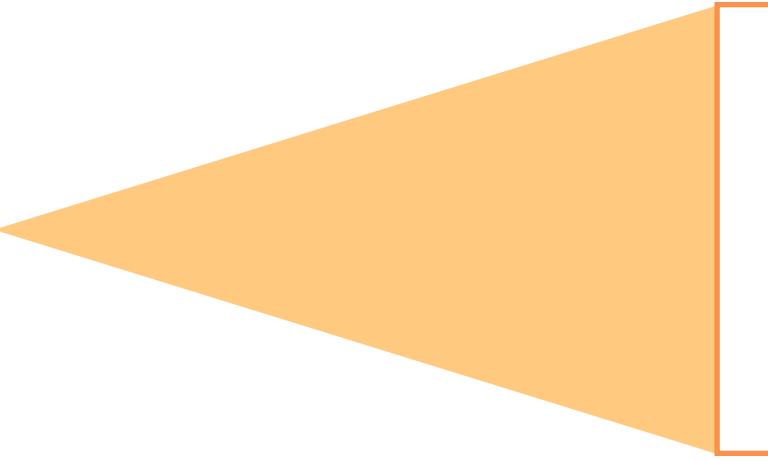


```
# Define server function
server <- function(input, output) {

  # Create the scatterplot object the plotOutput function is expecting
  output$scatterplot <- renderPlot({
    ggplot(data = NHANES, aes_string(x = input$x, y = input$y)) +
      geom_point()
  })
}
```



```
# Define server function
server <- function(input, output) {
  # Create the scatterplot object the plotOutput function is expecting
  output$scatterplot <- renderPlot({
    ggplot(data = NHANES, aes_string(x = input$x, y = input$y)) +
      geom_point()
  })
}
```



Contains instructions  
needed to build app



```
# Define server function  
server <- function(input, output) {  
  
  # Create the scatterplot object the plotOutput fun  
  output$scatterplot <- renderPlot({  
    ggplot(data = NHANES, aes_string(x = input$x, y  
      geom_point()  
  })  
}
```

plotOutput

Renders a **reactive** plot that is suitable for assigning to an output slot



```
# Define server function  
server <- function(input, output) {  
  
  # Create the scatterplot object the plotOutput function is expecting  
  output$scatterplot <- renderPlot({  
    ggplot(data = NHANES, aes_string(x = input$x, y = input$y)) +  
    geom_point()  
  })  
}  
}
```

Good ol' ggplot2 code,  
with **inputs** from UI



# **UI + Server**

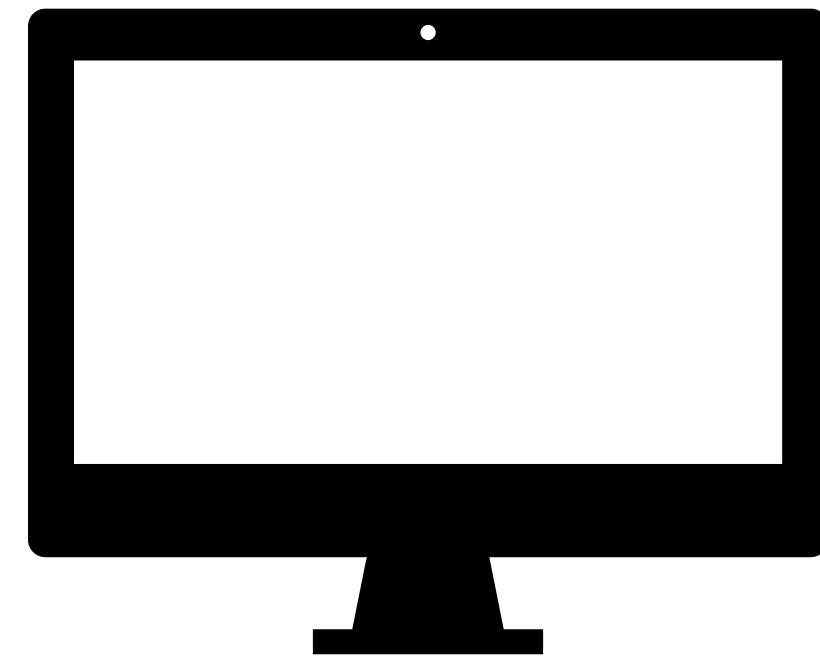


```
# Create the Shiny app object  
shinyApp(ui = ui, server = server)
```



Putting it all together...

nhanes-01.R



**DEMO**

# Your turn

- Add new select menu to color the points by
  - `inputId = "z"`
  - `label = "Color by:"`
  - `choices = c("Gender", "Depressed", "SleepTrouble", "SmokeNow", "Marijuana")`
  - `selected = "SleepTrouble"`
- Use this variable in the aesthetics of the `ggplot` function as the `color` argument to color the points by
- Run the app in the Viewer Pane
- Compare your code / output with the person sitting next to / nearby you

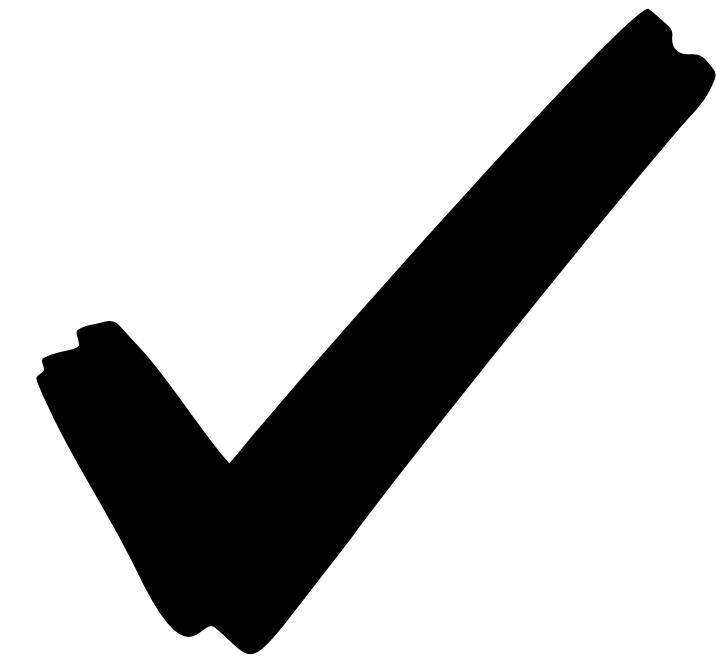


5m 00s



Solution to the previous exercise

nhanes-02.R



**SOLUTION**



# Inputs

**Action**

**actionButton(inputId, label, icon, ...)**

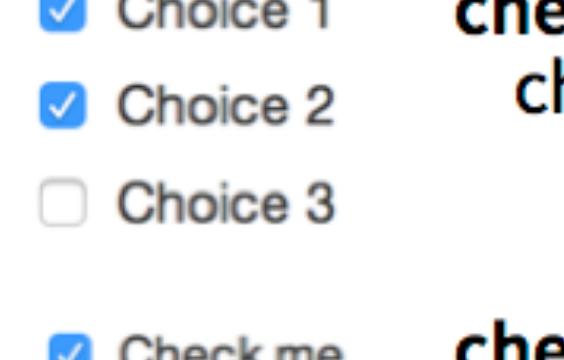

  

**Link**

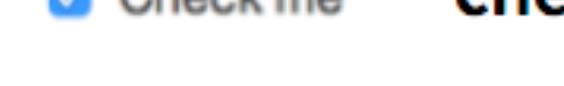
**actionLink(inputId, label, icon, ...)**

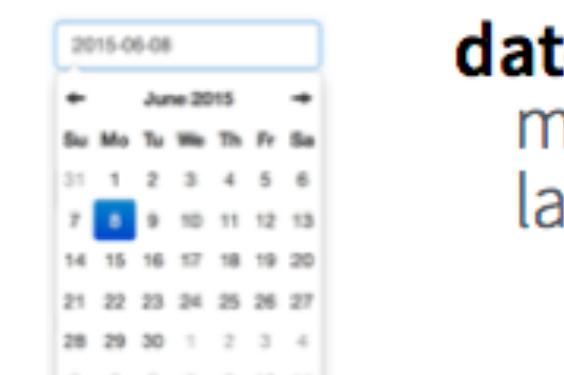
**checkboxGroupInput(inputId, label, choices, selected, inline)**

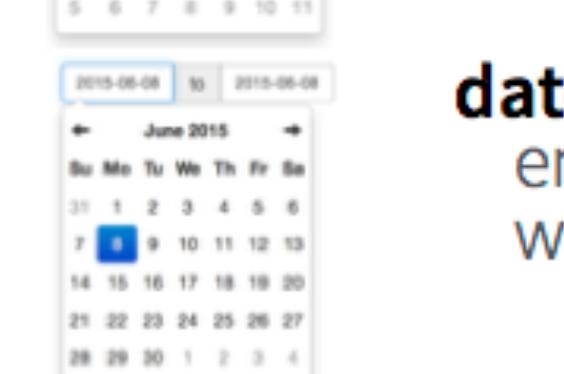
**checkboxInput(inputId, label, value)**

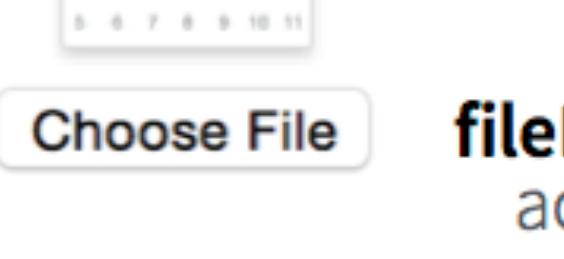
**dateInput(inputId, label, value, min, max, format, startview, weekstart, language)**

**dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)**

**fileInput(inputId, label, multiple, accept)**



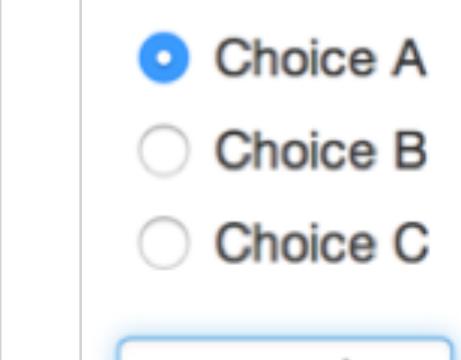
**numericInput(inputId, label, value, min, max, step)**

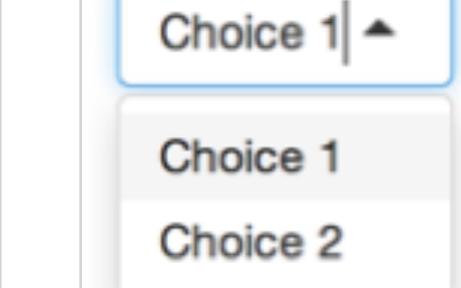
**passwordInput(inputId, label, value)**

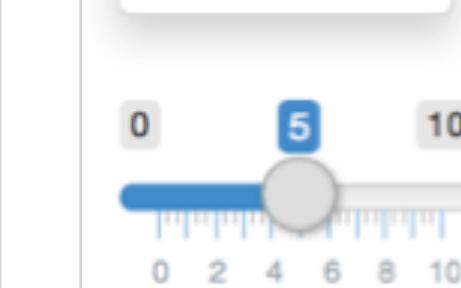
**radioButtons(inputId, label, choices, selected, inline)**

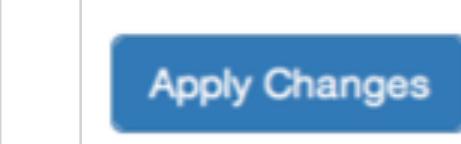
**selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())**

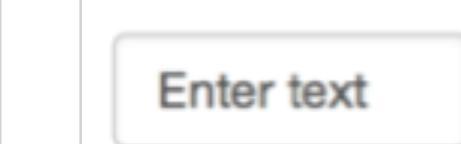
**sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)**

**submitButton(text, icon)**  
(Prevents reactions across entire app)

**textInput(inputId, label, value)**



# Your turn

- Add new input variable to control the alpha level of the points
  - This should be a `sliderInput`
    - See [shiny.rstudio.com/reference/shiny/latest/](https://shiny.rstudio.com/reference/shiny/latest/) for help
  - Values should range from 0 to 1
  - Set a default value that looks good
- Use this variable in the geom of the `ggplot` function as the alpha argument
- Run the app in a new window
- Compare your code / output with the person sitting next to / nearby you



5m 00s



Solution to the previous exercise

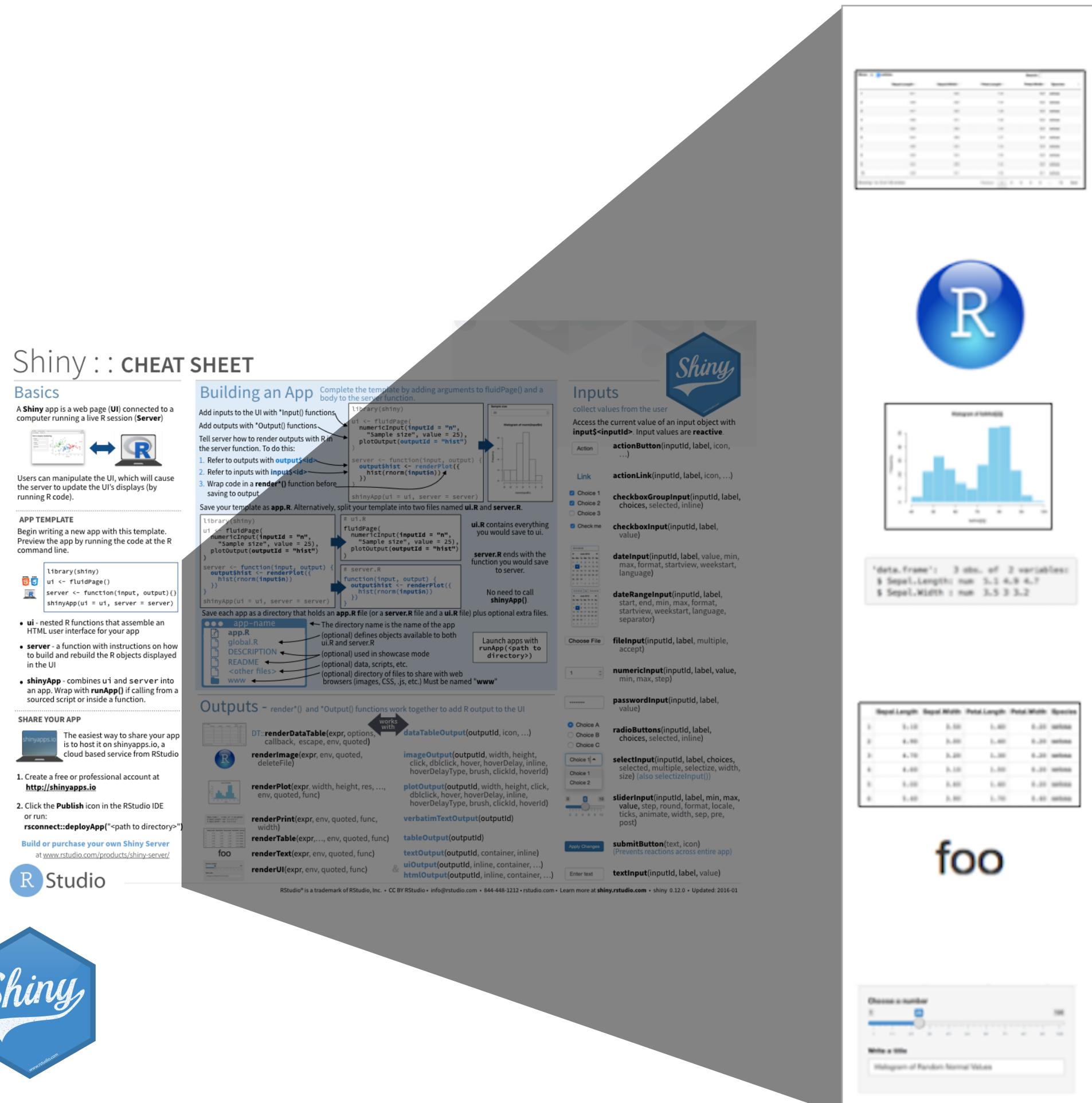
nhanes-03.R



**SOLUTION**



# Outputs



```
DT::renderDataTable(expr,  
options, callback, escape,  
env, quoted)
```

works  
with

**dataTableOutput(outputId, icon, ...)**

**renderImage(expr, env, quoted, deleteFile)**

**imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)**

**renderPlot(expr, width, height, res, ..., env, quoted, func)**

**plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)**

**renderPrint(expr, env, quoted, func, width)**

**verbatimTextOutput(outputId)**

**renderTable(expr,..., env, quoted, func)**

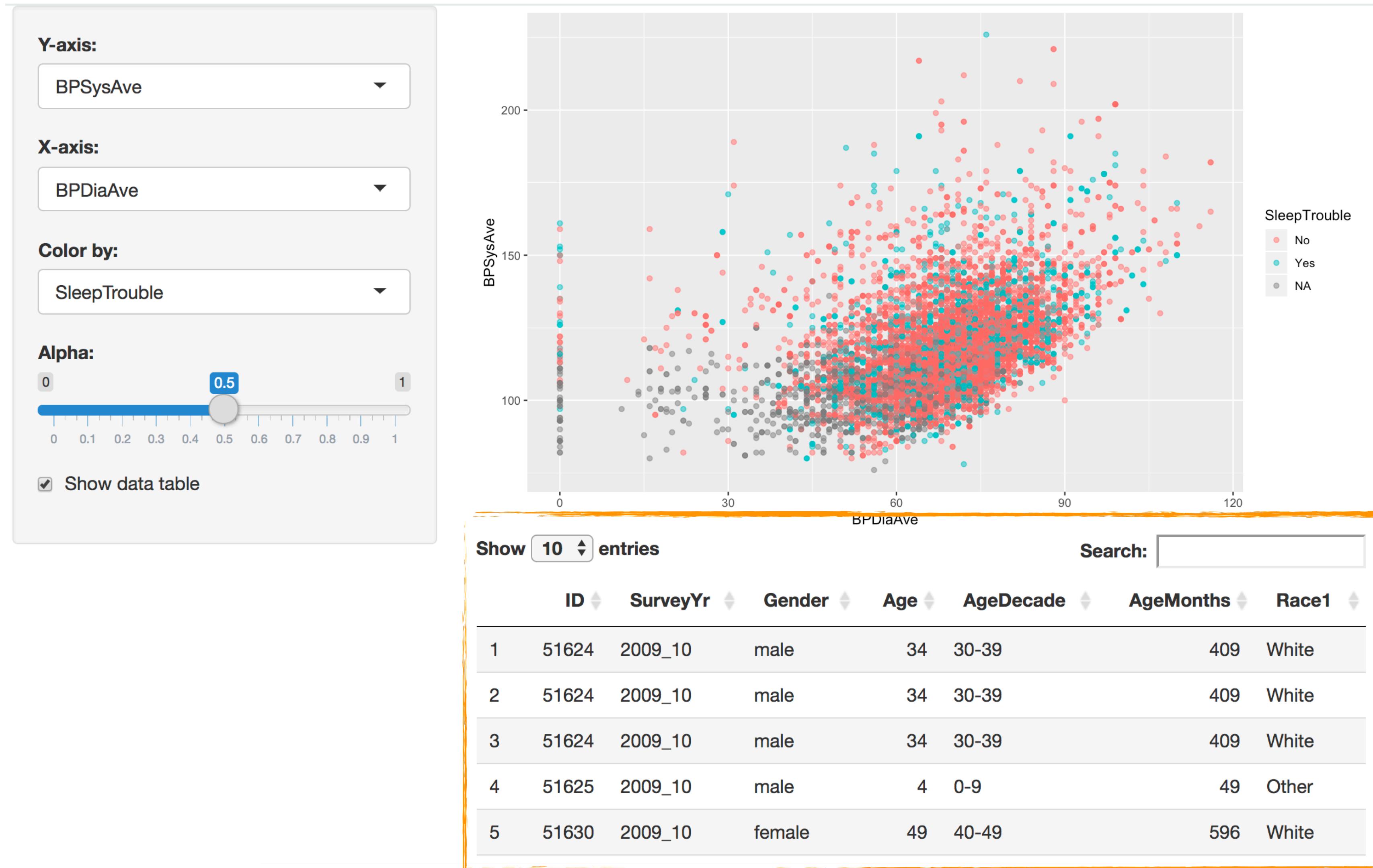
## tableOutput(outputId)

**renderText(expr, env, quoted, func)**

**textOutput(outputId, container, inline)**

**renderUI(expr, env, quoted, func)**

# Which render\* and \*Output function duo is used to add this table to the app?



```
library(shiny)
library(tidyverse)
library(NHANES)
ui <- fluidPage(
  DT::dataTableOutput()
)

server <- function(input, output) {
  DT::renderDataTable()
}

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



# Your turn

- Create a new output item using `DT::renderDataTable`.
- Show first seven columns of NHANES data, show 10 rows at a time, and hide row names, e.g.
  - `data = NHANES[, 1:7]`
  - `options = list(pageLength = 10)`
  - `rownames = FALSE`
- Add a `DT::dataTableOutput` to the main panel
- Run the app in a new Window
- Compare your code / output with the person sitting next to / nearby you
- **Stretch goal:** Make the number of columns visible in the table a user defined input

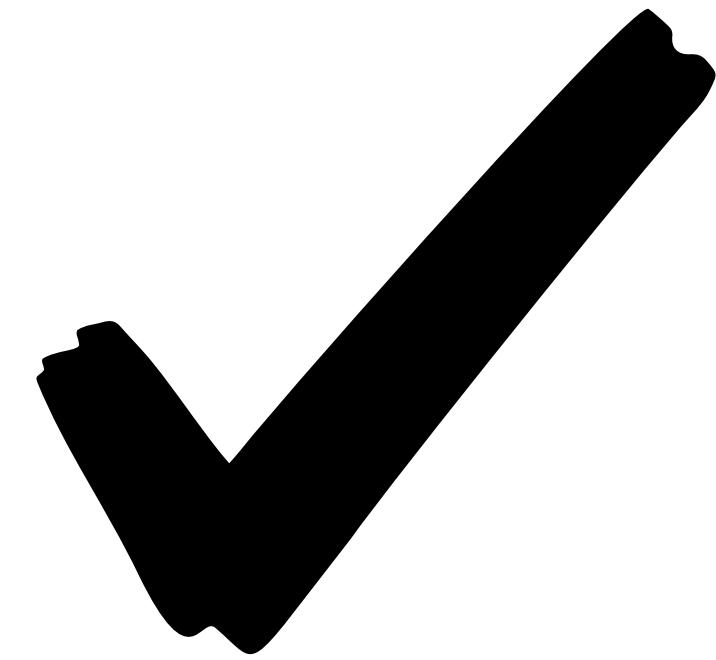


5m 00s



Solution to the previous exercise

nhanes-04.R



**SOLUTION**



# Execution



Where you place code in your app will determine how many times they are run (or re-run), which will in turn affect the performance of your app, since Shiny will run some sections of your app script more often than others.

```
library(shiny)  
library(tidyverse)  
library(NHANES)
```

```
ui <- fluidPage(  
  ...  
)  
  
server <- function(input, output) {  
  
  output$x <- renderPlot({  
    ...  
  })  
  
}  
  
shinyApp(ui = ui, server = server)
```

**Run once  
when app is  
launched**



```
library(shiny)
library(tidyverse)
library(NHANES)

ui <- fluidPage(
  ...
)

server <- function(input, output) {
  output$x <- renderPlot({
    ...
  })
}

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



Run once  
each time a user  
visits the app



```
library(shiny)
library(tidyverse)
library(NHANES)

ui <- fluidPage(
  ...
)

server <- function(input, output) {
  output$x <- renderPlot({
    ...
  })
}

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

Run once  
each time a user  
changes a widget that  
output\$x depends on



03

# Understanding reactivity



# Reactivity 101

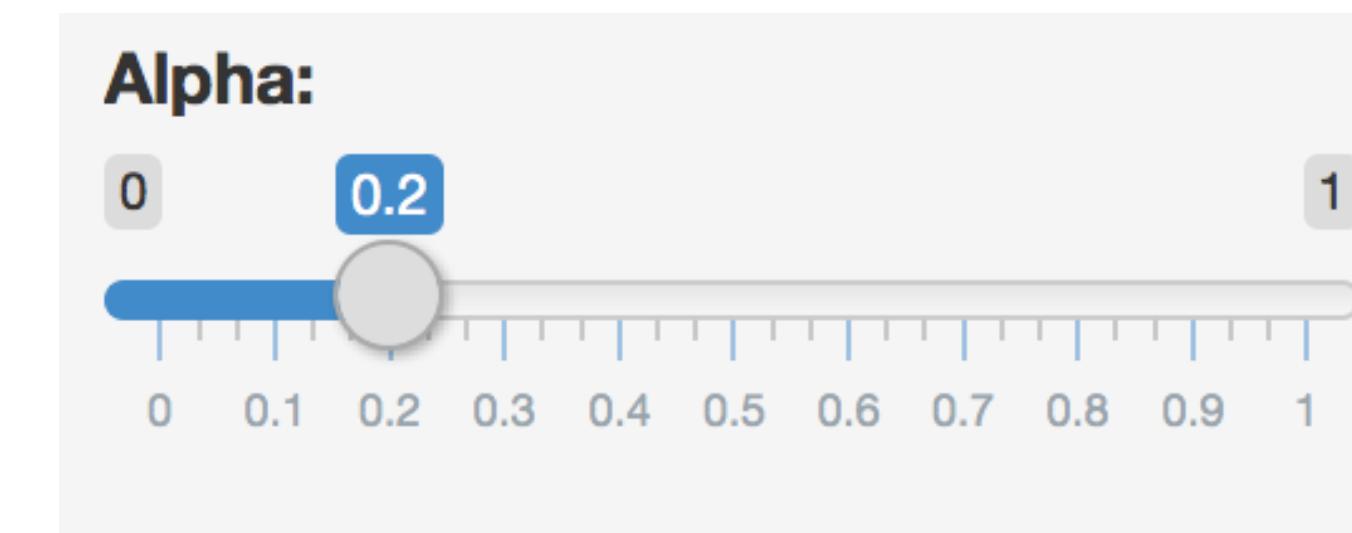


# Reactions

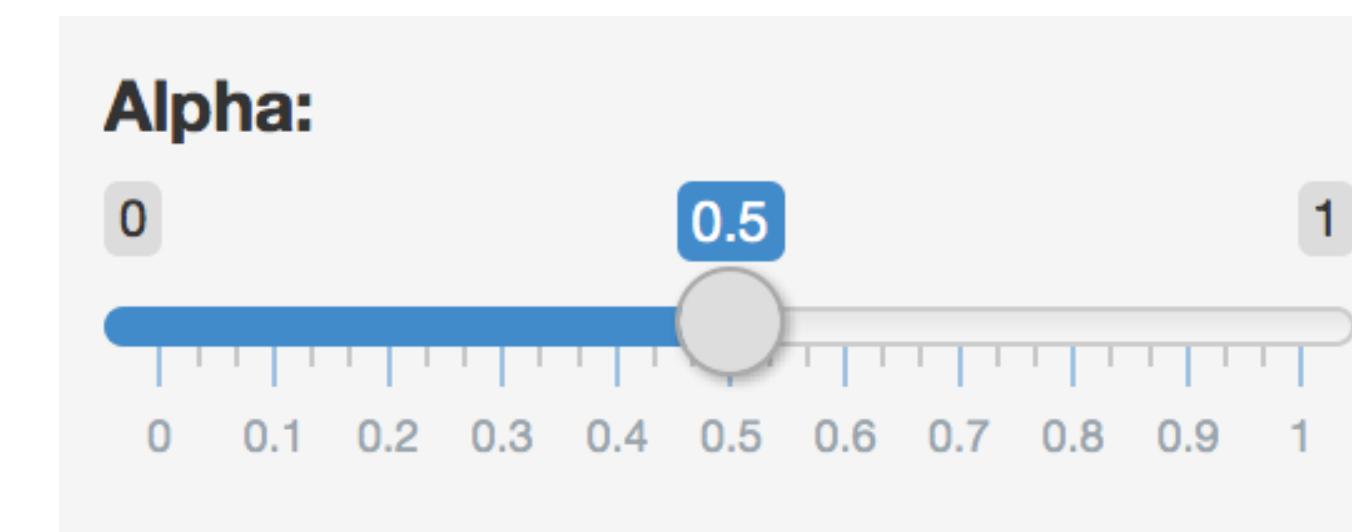
The **input\$** list stores the current value of each input object under its name.

```
# Set alpha level  
sliderInput(inputId = "alpha",  
            label = "Alpha:",  
            min = 0, max = 1,  
            value = 0.5)
```

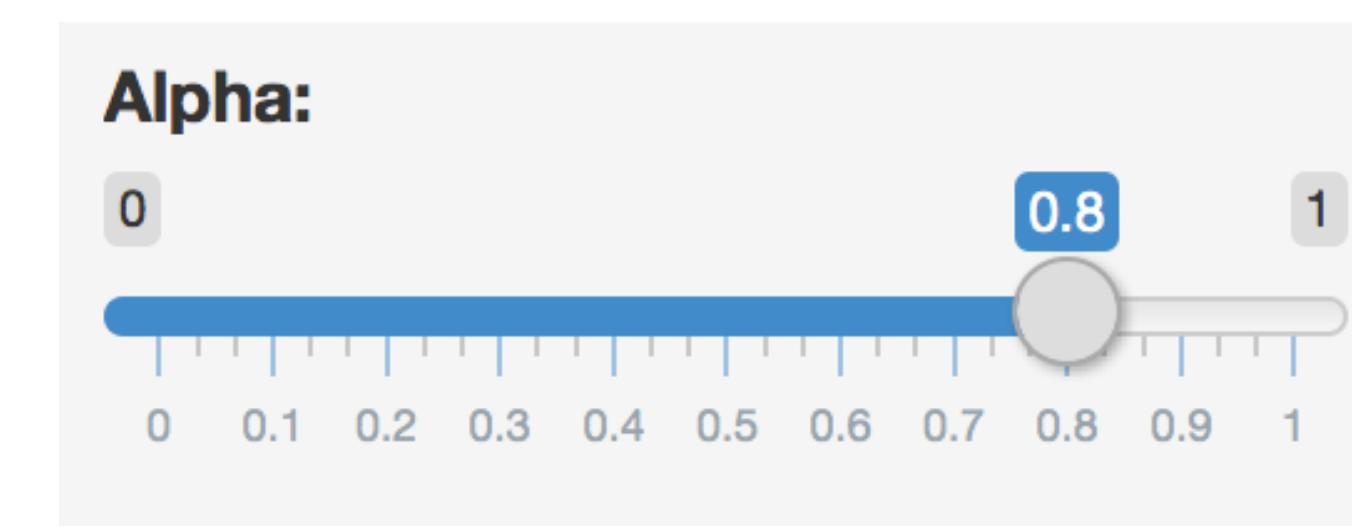
input\$alpha



input\$alpha = 0.2



input\$alpha = 0.5



input\$alpha = 0.8



# Reactivity 101

Reactivity automatically occurs when an input value  
is used to render an output object

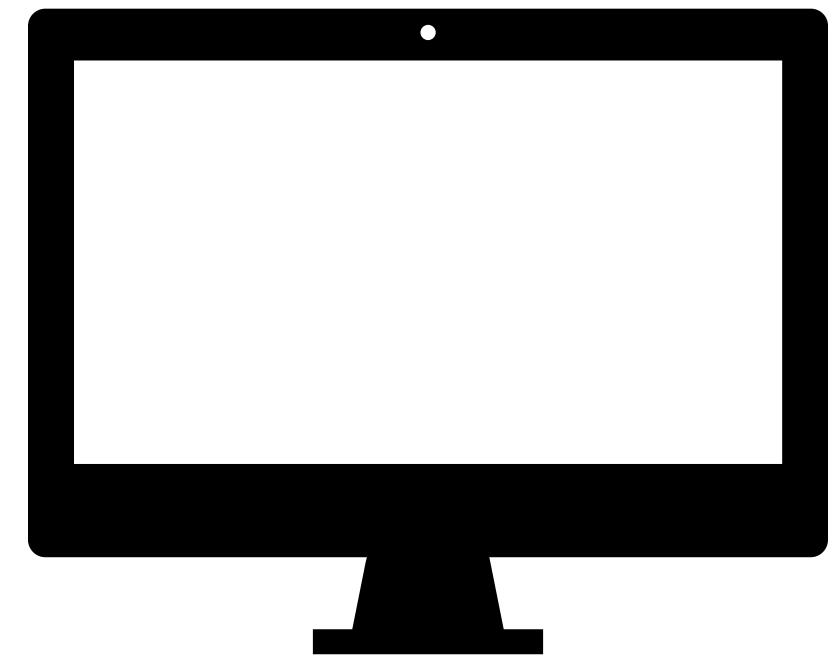
```
# Define server function required to create the scatterplot
server <- function(input, output) {
  # Create the scatterplot object the plotOutput function is expecting
  output$scatterplot <- renderPlot(
    ggplot(data = NHANES, aes_string(x = input$x, y = input$y,
                                      color = input$z)) +
    geom_point(alpha = input$alpha)
  )
}
```



# Reactive flow

Suppose you want the option to plot only certain education level(s) as well as report how many such participants are plotted:

1. Add a UI element for the user to select which education level(s) they want to plot
2. Filter for chosen education level(s) and save as a new (reactive) expression
3. Use new data frame (which is reactive) for plotting
4. Use new data frame (which is reactive) also for reporting number of observations



**DEMO**

1. Add a UI element for the user to select which education level(s) they want to plot

```
# Select which education level(s) to plot
checkboxGroupInput(inputId = "education",
  label = "Select education level(s):",
  choices = levels(NHANES$Education),
  selected = "College Grad")
```



2. Filter for chosen education level(s) and save as a new (reactive) expression

```
# Server  
# Create a subset of data filtering for chosen education levels  
NHANES_subset <- reactive({  
  req(input$education)  
  filter(NHANES, title_type %in% input$education)  
})
```

Creates a **cached expression** that knows it is out of date when input changes



### 3. Use new data frame (which is reactive) for plotting

```
# Create the scatterplot object the plotOutput function is expecting
output$scatterplot <- renderPlot({
  ggplot(data = NHANES_subset(), aes_string(x =
    col1, y = col2)) +
  geom_point(...) +
  ...
})
```

**Cached** - only re-run  
when inputs change



4. Use new data frame (which is reactive) also for printing number of observations

```
# UI
mainPanel(
  ...
  # Print number of obs plotted
  uiOutput(outputId = "n"),
  ...
)
# Server
output$n <- renderUI({
  types <- NHANES_subset$title_type %>%
    factor(levels = input$selected_type)
  counts <- table(types)

  HTML(paste("There are", counts, input$selected_type, "participants in this
dataset.<br>"))
})
```

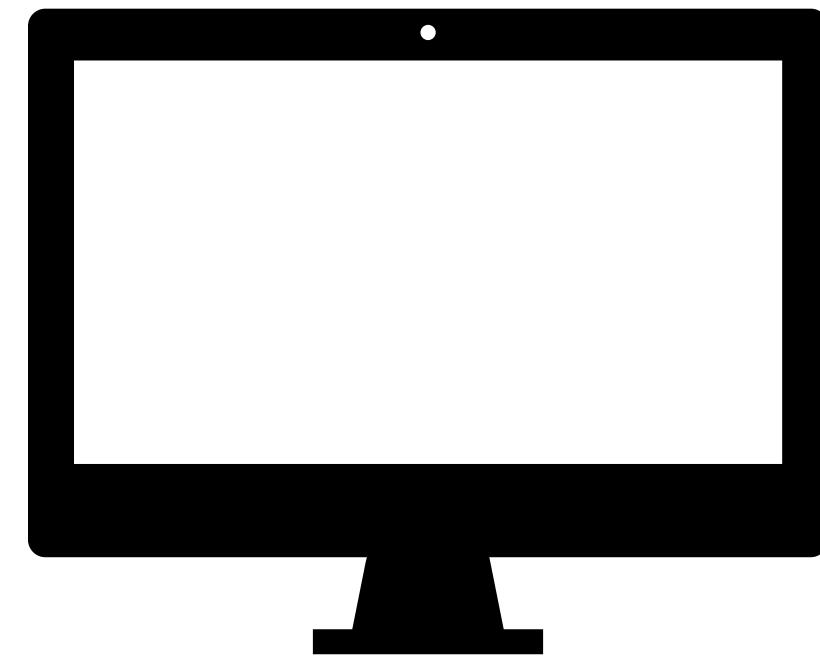


# Putting it altogether

`nhanes-05.R`

Also notice

- HTML tags for visual separation
- `req()`



## DEMO

# When to use reactives

- By using a reactive expression for the subsetted data frame, we were able to get away with subsetting once and then using the result twice
- In general, reactive conductors let you
  - not repeat yourself (i.e. avoid copy-and-paste code) which is a maintenance boon)
  - decompose large, complex (code-wise, not necessarily CPU-wise) calculations into smaller pieces to make them more understandable
- These benefits are similar to what happens when you decompose a large complex R script into a series of small functions that build on each other



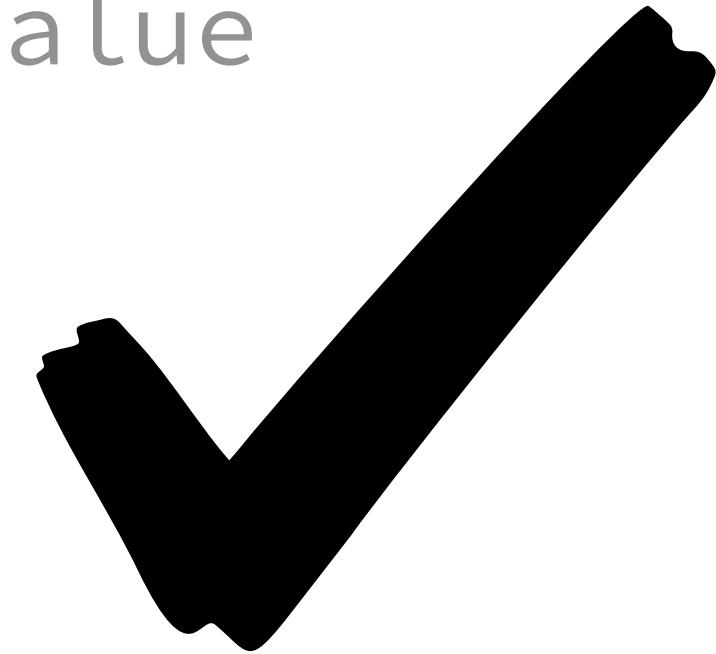
Suppose we want to plot only a random sample of participants, of size determined by the user. What is wrong with the following?

```
# Server  
# Create a new data frame that is a sample of n_samp  
# observations from NHANES  
NHANES_sample <- sample_n(NHANES_sample(), input$n_samp)  
  
# Plot the sampled participants  
output$scatterplot <- renderPlot({  
  ggplot(data = NHANES_sample,  
          aes_string(x = input$x, y = input$y,  
                      color = input$z)) +  
  geom_point(...)  
})
```



```
# Server
# Create a new data frame that is a sample of n_samp
# observations from NHANES
NHANES_sample <- reactive({
  req(input$n_samp)      # ensure availability of value
  sample_n(NHANES_sample(), input$n_samp)
})

# Plot the sampled participants
output$scatterplot <- renderPlot({
  ggplot(data = NHANES_sample(),
    aes_string(x = input$x,
               y = input$y,
               color = input$z)) +
  geom_point(...)
})
```



# SOLUTION



# Render functions



# Render functions

```
render*({ [code_chunk] })
```

- Provide a code chunk that describes how an output should be populated
- The output will update in response to changes in any reactive values or reactive expressions that are used in the code chunk

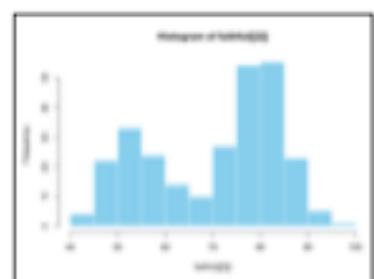


	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.40	0.20	versicolor
2	4.90	3.00	1.40	0.20	versicolor
3	4.70	3.20	1.30	0.20	versicolor
4	4.60	3.10	1.50	0.20	versicolor
5	5.00	3.40	1.50	0.20	versicolor
6	5.40	3.90	1.70	0.40	versicolor

**DT::renderDataTable(expr,  
options, callback, escape,  
env, quoted)**



**dataTableOutput(outputId, icon, ...)**



**renderImage(expr, env, quoted, deleteFile)**

**renderPlot(expr, width, height, res, ..., env,  
quoted, func)**

**renderPrint(expr, env, quoted, func,  
width)**

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.40	0.20	versicolor
2	4.90	3.00	1.40	0.20	versicolor
3	4.70	3.20	1.30	0.20	versicolor
4	4.60	3.10	1.50	0.20	versicolor
5	5.00	3.40	1.50	0.20	versicolor
6	5.40	3.90	1.70	0.40	versicolor

**renderTable(expr,..., env, quoted, func)**

**foo**

**renderText(expr, env, quoted, func)**

**renderUI(expr, env, quoted, func)**

**imageOutput(outputId, width, height, click,  
dblclick, hover, hoverDelay, hoverDelayType,  
brush, clickId, hoverId, inline)**

**plotOutput(outputId, width, height, click,  
dblclick, hover, hoverDelay, hoverDelayType,  
brush, clickId, hoverId, inline)**

**verbatimTextOutput(outputId)**

**tableOutput(outputId)**

**textOutput(outputId, container, inline)**

**uiOutput(outputId, inline, container, ...)**

**& htmlOutput(outputId, inline, container, ...)**

# Recap

```
render*({ [code_chunk] })
```

- These functions make objects to display
- Results should always be saved to output\$
- They make an observer object that has a block of code associated with it
- The object will rerun the entire code block to update itself whenever it is invalidated



# Implementation



# Implementation of reactives

- **Reactive values** – reactiveValues():
  - e.g. `input`: which looks like a list, and contains many individual reactive values that are set by input from the web browser
- **Reactive expressions** – reactive(): they depend on reactive values and observers depend on them
  - Can access reactive values or other reactive expressions, and they return a value
  - Useful for caching the results of any procedure that happens in response to user input
  - e.g. reactive data frame subsets we created earlier
- **Observers** – observe(): they depend on reactive expressions, but nothing else depends on them
  - Can access reactive sources and reactive expressions, but they don't return a value; they are used for their side effects
  - e.g. output object is a reactive observer, which also looks like a list, and contains many individual reactive observers that are created by using reactive values and expressions in reactive functions



# Reactive expressions vs. observers

- Similarities: Both store expressions that can be executed
- Differences:
  - Reactive expressions return values, but observers don't
  - Observers (and endpoints in general) eagerly respond to reactives, but reactive expressions (and conductors in general) do not
  - Reactive expressions must not have side effects, while observers are only useful for their side effects



# Your turn

Debug the following app scripts:

- review/whats-wrong.R
- review/mult-3.R
- review/add-2.R

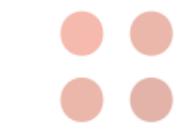


5m 00s



# Where to go next?





## rstudio::conf 2018

This category is for anything and everything related to rstudio::conf.

7 / week  
4 unread  
2 new



## tidyverse

This category is for anything and everything about the tidyverse.

15 / week  
6 new



## RStudio IDE

This category is for discussing the RStudio IDE, both desktop and server versions.

21 / week  
1 unread  
7 new



## Teaching

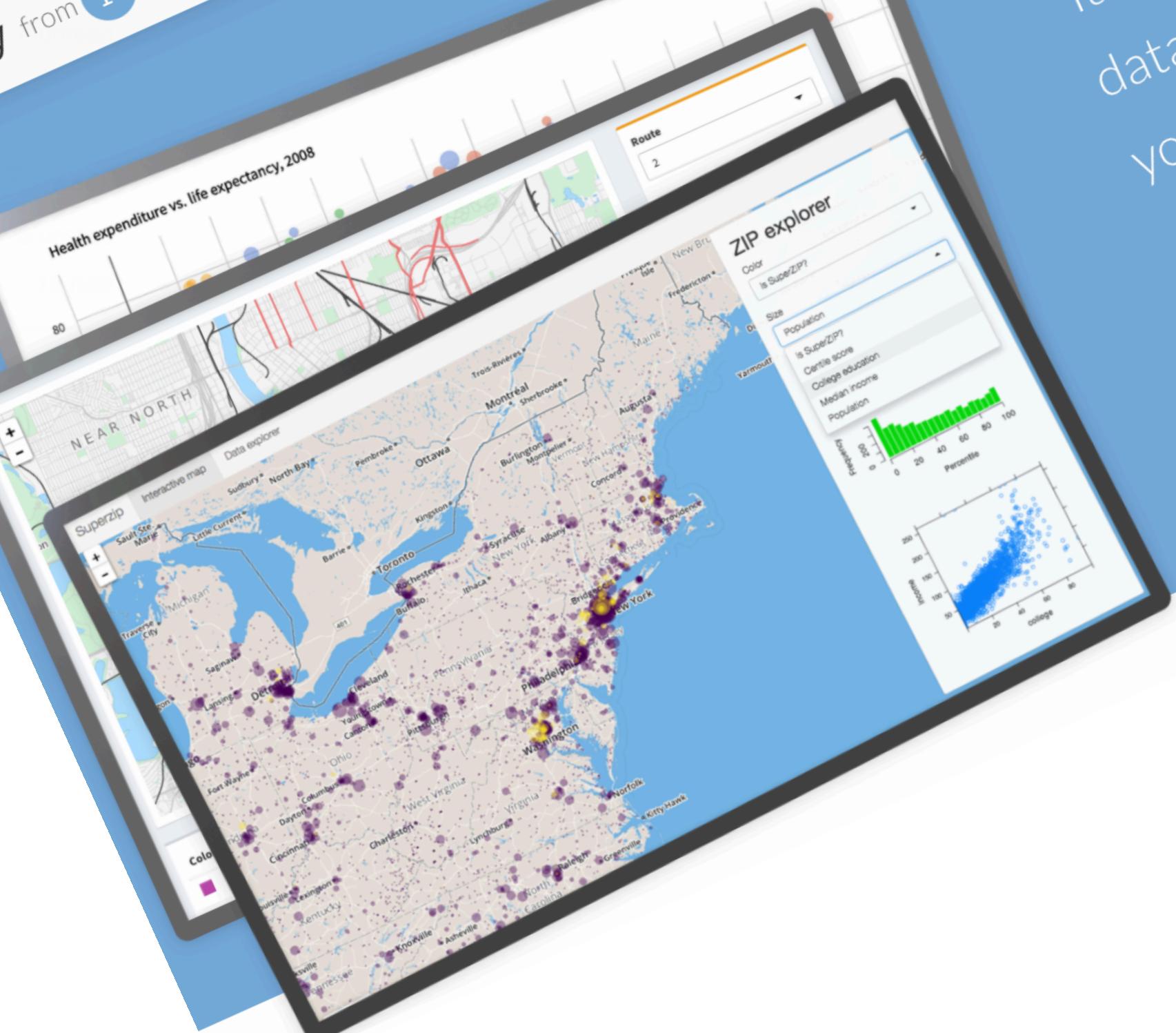
For discussions about teaching.

shiny from R Studio



## shiny

Please ask your questions about shiny here.



## R Markdown

Please ask your questions about R Markdown here.

Get Started

Gallery

Articles

Reference

Deploy

Help

Contribute



Interact. Analyze. Communicate.  
Take a fresh, interactive approach to telling your data story with Shiny. Let users interact with your data and your analysis. And do it all with R.

[bit.ly/shiny-rlphl-git](https://bit.ly/shiny-rlphl-git)



Mine Çetinkaya-Rundel

@minebocek

mine-cetinkaya-rundel

mine@rstudio.com