Data Valorization: Shiny Application

Lionel Fillatre

fillatre@unice.fr

Topics

- Introduction
- First Application
- Build a user-interface
- Control Widgets
- Display reactive output
- Use R script and data
- Conclusion

1 Introduction

Motivation

- Why?
  - Want to develop R based “tiny” applications
  - Want to get R into web browsers

- How?
  - Shiny is open-sourced by RStudio on CRAN
  - New model for web-accessible R code
  - Able to generate basic web UIs
  - Built on a “Reactive Programming” model
  - Entirely extensible: custom inputs and outputs
  - Many information available at http://shiny.rstudio.com/

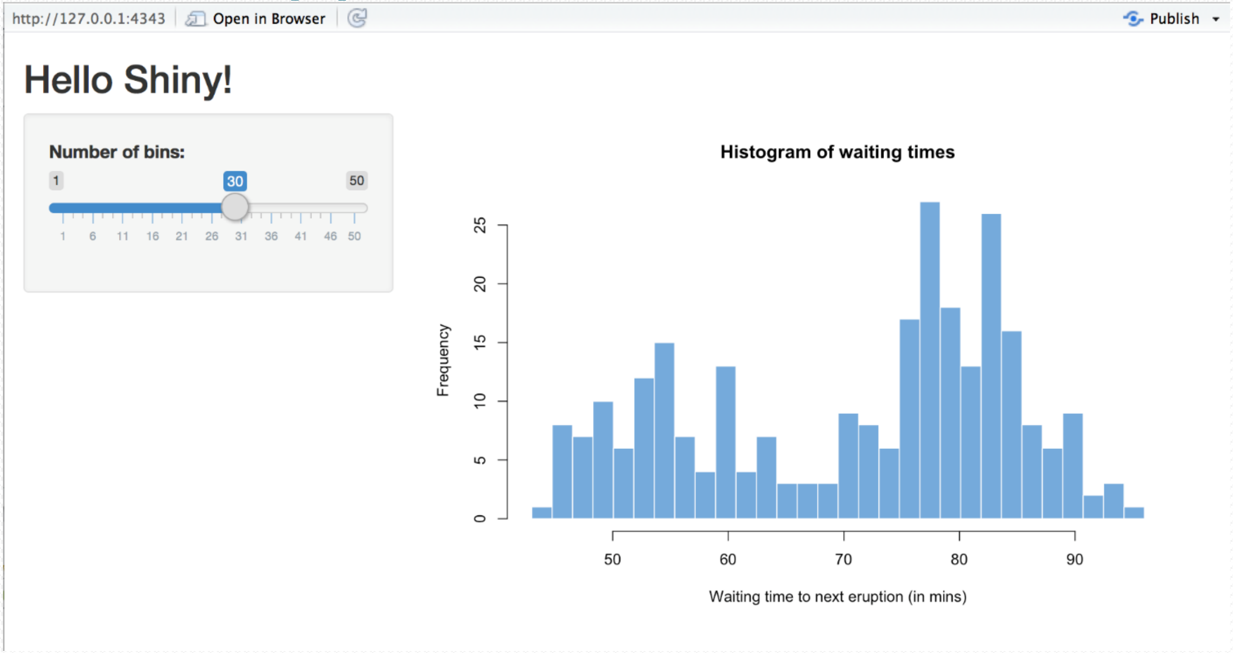
2 First Application

Try Shiny

- To install Shiny, type:
  - ➢ install.packages("shiny")

- To run Hello Shiny, type:
  - ➢ library(shiny)
  - ➢ runExample("01_hello")

First app


http://127.0.0.1:4343
Open in Browser
Publish
Hello Shiny!
Number of bins:
1
30
50
1 6 11 16 21 26 31 36 41 46 50
Histogram of waiting times
Frequency
0 5 10 15 20 25
50 60 70 80 90
Waiting time to next eruption (in mins)

Structure of a Shiny App

- Shiny apps have three components:
  - a user-interface script
  - a server script
  - a call to the shinyApp function

- The user-interface (ui) script controls the layout and appearance of your app.

- The server function contains the instructions that your computer needs to build your app

- The shinyApp function creates Shiny app objects from an explicit UI/server pair.

Ui

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

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

  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins", "Number of bins:", min = 1, max = 50, value = 30)
    ),

    # Show a plot of the generated distribution
    mainPanel(
        plotOutput(outputId = "distPlot")
    )
  )
)
```

Server

```
server <- function(input, output) {
# Histogram of the Old Faithful Geyser Data ----
# with requested number of bins
# 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({
   x <- faithful$waiting
   bins <- seq(min(x), max(x), length.out = input$bins + 1)
   hist(x, breaks = bins, col = "#75AADB", border = "white",
         xlab = "Waiting time to next eruption (in mins)",
         main = "Histogram of waiting times")
   })
}
```

App.R

```
library(shiny)

# See above for the definitions of ui and server
ui <- ...


server <- ...


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

Running a Shiny App

- Every Shiny app has the same structure: an app.R file that contains ui and server.

- You can create a Shiny app by making a new directory and saving an app.R file inside it. It is recommended that each app will live in its own unique directory

- You can run a Shiny app by giving the name of its directory to the function runApp. For example if your Shiny app is in a directory called my_app, run it with the following code:

```
> library(shiny)
> runApp(‘my_app’)
```

3 Build a user-interface

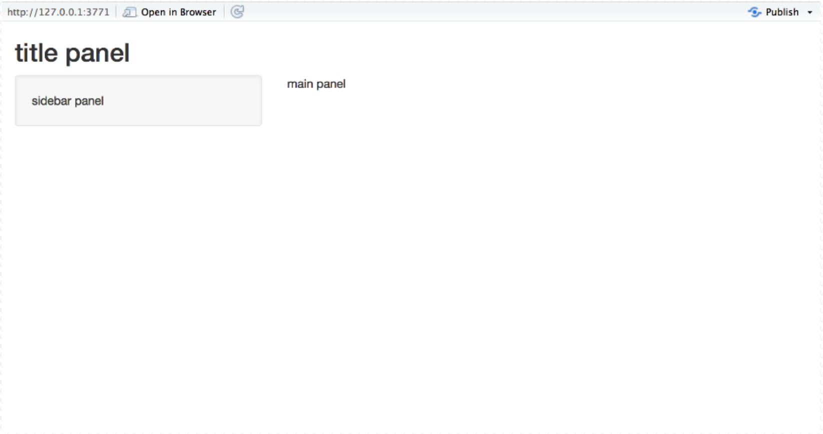
Minimal Application

```
library(shiny)

# Define UI ----
ui <- fluidPage(
)

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

# Run the app ----
shinyApp(ui = ui, server = server)
```

- This code is the bare minimum needed to create a Shiny app. The result is an empty app with a blank user-interface.

Layout

```
ui <- fluidPage(
 titlePanel("title panel"),

 sidebarLayout(
  sidebarPanel("sidebar panel"),
  mainPanel("main panel")
 )
)
```



- The function *fluidPage* creates a display that automatically adjusts to the dimensions of your user’s browser window.

HTML

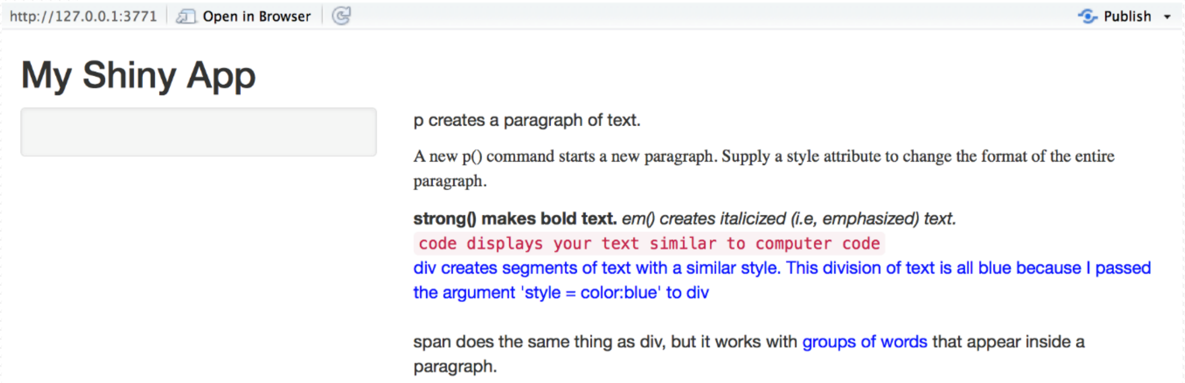
- To add more advanced content, use one of Shiny’s HTML tag functions. These functions parallel common HTML5 tags.
- Some functions:

shiny function	HTML5 equivalent	creates
p	<p>	A paragraph of text
h1	<h1>	A first level header
h2	<h2>	A second level header
h3	<h3>	A third level header
h4	<h4>	A fourth level header
h5	<h5>	A fifth level header
h6	<h6>	A sixth level header
a	<a>	A hyper link
br	<br>	A line break (e.g. a blank line)
div	<div>	A division of text with a uniform style
span	<span>	An in-line division of text with a uniform style
pre	<pre>	Text ‘as is’ in a fixed width font
code	<code>	A formatted block of code
img	<img>	An image
strong	<strong>	Bold text
em	<em>	Italicized text
HTML		Directly passes a character string as HTML code

Example: ui

```
ui <- fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      p("p creates a paragraph of text."),
      p("A new p() command starts a new paragraph. Supply a style attribute to change the format of 
the entire paragraph.", style = "font-family: 'times'; font-si16pt"),
      strong("strong() makes bold text."),
      em("em() creates italicized (i.e, emphasized) text."),
      br(),
      code("code displays your text similar to computer code"),
      div("div creates segments of text with a similar style. This division of text is all blue because I 
passed the argument 'style = color:blue' to div", style = "color:blue"),
      br(),
      p("span does the same thing as div, but it works with",
        span("groups of words", style = "color:blue"),
        "that appear inside a paragraph.")
    )
  )
)
```

Result



My Shiny App

p creates a paragraph of text.

A new p() command starts a new paragraph. Supply a style attribute to change the format of the entire paragraph.

strong() makes bold text. *em() creates italicized (i.e, emphasized) text.*
`code displays your text similar to computer code`
div creates segments of text with a similar style. This division of text is all blue because I passed the argument 'style = color:blue' to div

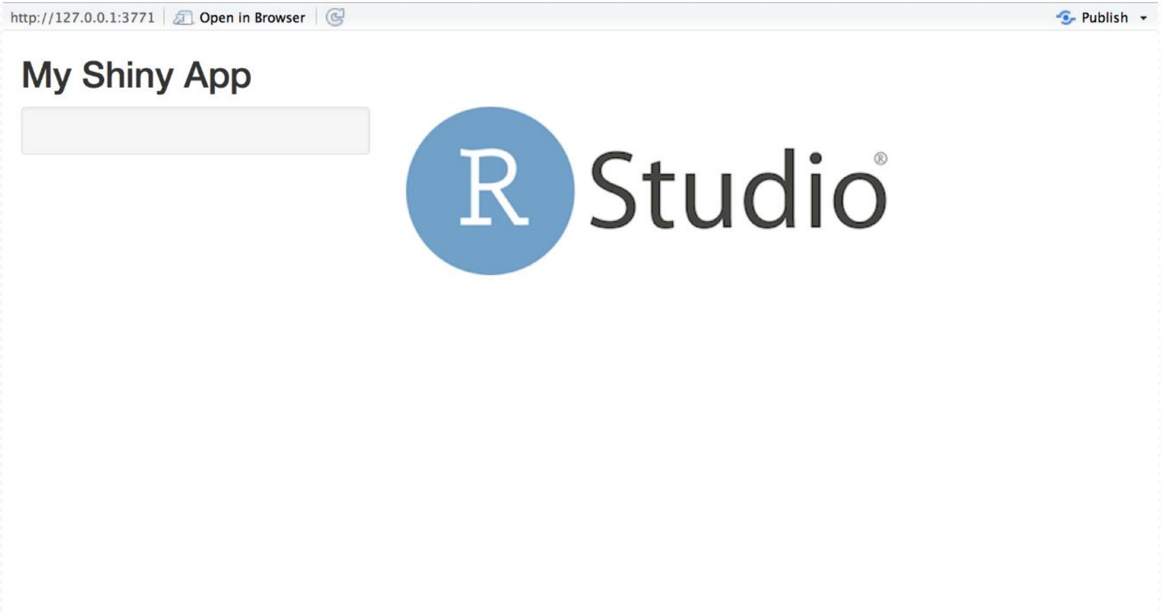
span does the same thing as div, but it works with groups of words that appear inside a paragraph.

Put an image

```
ui <- fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      img(src = "rstudio.png", height = 140, width = 400)
    )
  )
)
```

- The images must be stored into a subdirectory called « www »
  - Example: my_app/www/ rstudio.png

Result



My Shiny App

Grid Layout

- The familiar sidebarLayout() described above makes use of Shiny's lower-level grid layout functions.

- Rows are created by the fluidRow() function and include columns defined by the column() function.

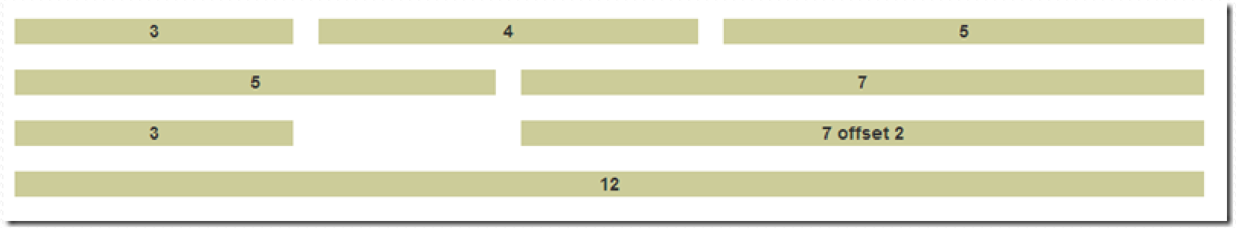
- Column widths are based on the Bootstrap 12-wide grid system (html), so should add up to 12 columns within a fluidRow() container.

- To illustrate, the sidebar layout implemented using the fluidRow(), column() and wellPanel() functions is given in next slides

- The first parameter to the column() function is it's width (out of a total of 12 columns).

- It's also possible to offset the position of columns to achieve more precise control over the location of UI elements. You can move columns to the right by adding the offset parameter to the column() function. Each unit of offset increases the left-margin of a column by a whole column.

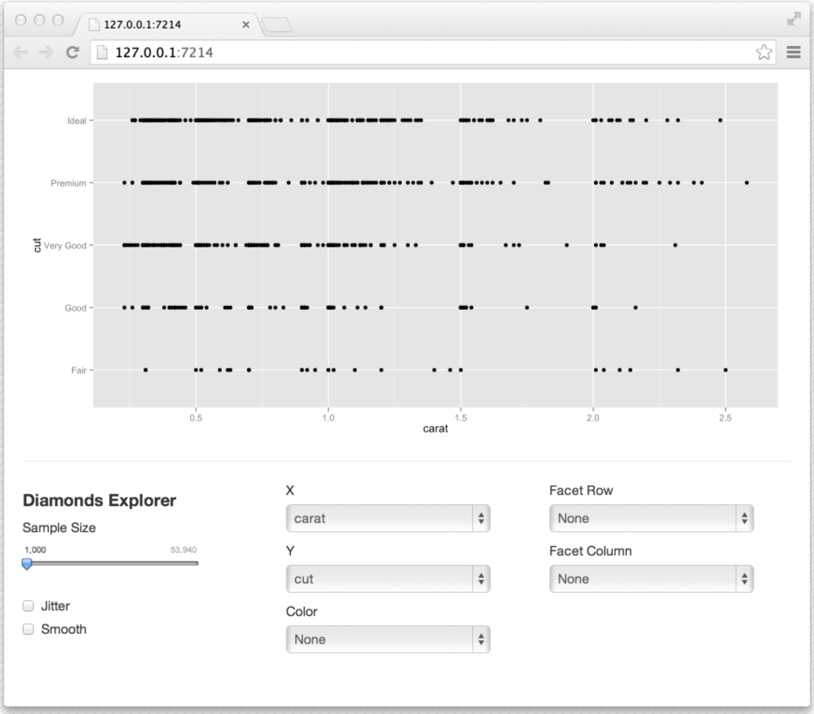
12-wide grid system



Example

```
library(ggplot2)
dataset <- diamonds
ui <- fluidPage(
 title = "Diamonds Explorer",
 plotOutput('plot'),
 hr(),
 fluidRow(
  column(3,
   h4("Diamonds Explorer"),
   sliderInput('sampleSize', 'Sample Size', min=1, max=nrow(dataset), value=min(1000, nrow(dataset)), step=500, round=0),
   br(),
   checkboxInput('jitter', 'Jitter'),
   checkboxInput('smooth', 'Smooth')
  ),
  column(4, offset = 1,
   selectInput('x', 'X', names(dataset)),
   selectInput('y', 'Y', names(dataset), names(dataset)[[2]]),
   selectInput('color', 'Color', c('None', names(dataset)))
  ),
  column(4,
   selectInput('facet_row', 'Facet Row', c(None='.', names(dataset))),
   selectInput('facet_col', 'Facet Column', c(None='.', names(dataset)))
  )
 )
)
```

Visual Result



Diamonds Explorer

Sample Size

1,000 53,940

Jitter

Smooth

X: carat

Y: cut

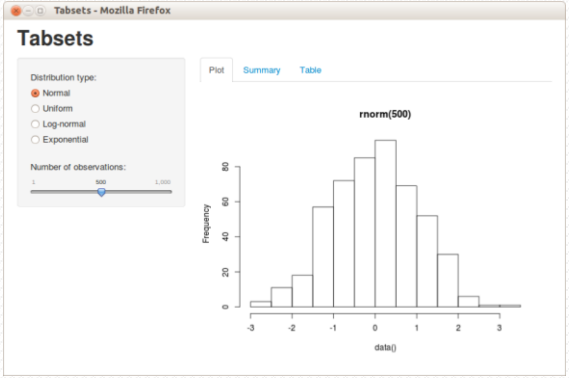
Color: None

Facet Row: None

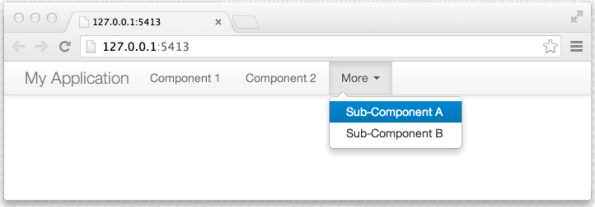
Facet Column: None

Other possibilities

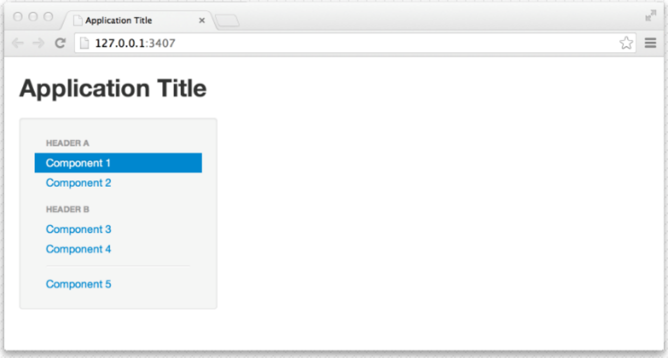
Tabsets



Navigation bars



Navigation lists



4 Control Widgets

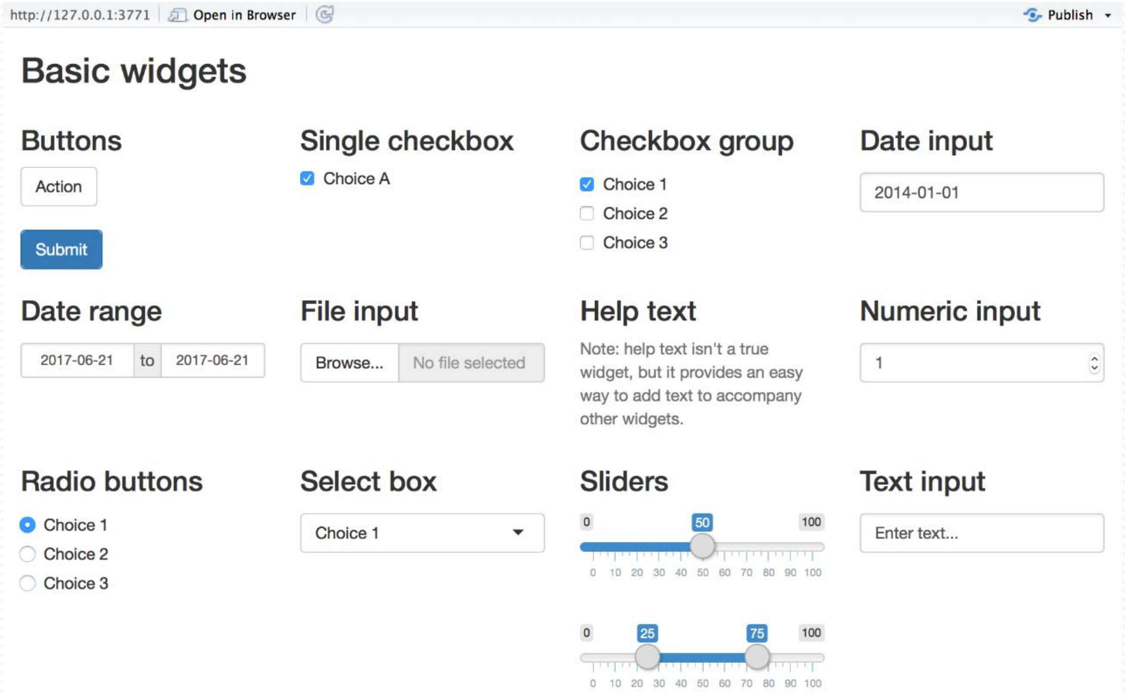
Standard Widgets

- Shiny comes with a family of pre-built widgets, each created with a transparently named R function

- The standard Shiny widgets are:

function	**widget**
actionButton	Action Button
checkboxGroupInput	A group of check boxes
checkboxInput	A single check box
dateInput	A calendar to aid date selection
dateRangeInput	A pair of calendars for selecting a date range
fileInput	A file upload control wizard
helpText	Help text that can be added to an input form
numericInput	A field to enter numbers
radioButtons	A set of radio buttons
selectInput	A box with choices to select from
sliderInput	A slider bar
submitButton	A submit button
textInput	A field to enter text.

Visual Results



Basic widgets

Buttons

Action

Submit

Single checkbox

[✓] Choice A

Checkbox group

[✓] Choice 1
[ ] Choice 2
[ ] Choice 3

Date input

2014-01-01

Date range

2017-06-21 to 2017-06-21

File input

Browse... No file selected

Help text

Note: help text isn't a true widget, but it provides an easy way to add text to accompany other widgets.

Numeric input

1

Radio buttons

● Choice 1
○ Choice 2
○ Choice 3

Select box

Choice 1

Sliders

0 50 100

0 10 20 30 40 50 60 70 80 90 100

0 25 75 100

0 10 20 30 40 50 60 70 80 90 100

Text input

Enter text...

Example (Basic Widgets)

```
ui <- fluidPage(
 titlePanel("Basic widgets"),
  fluidRow(
  column(3,
        h3("Buttons"),
        actionButton("action", "Action"),
        br(),
        br(),
        submitButton("Submit")),
  column(3,
        h3("Single checkbox"),
        checkboxInput("checkbox", "Choice A", value = TRUE)),
  column(3,
        checkboxGroupInput("checkGroup",
                     h3("Checkbox group"),
                     choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3),
                     selected = 1)),
  column(3,
        dateInput("date", h3("Date input"), value = "2014-01-01"))
 ),
 fluidRow(
   column(3, dateRangeInput("dates", h3("Date range"))),
   column(3, fileInput("file", h3("File input"))),
  column(3,
        h3("Help text"),
        helpText("Note: help text isn't a true widget,",
                 "but it provides an easy way to add text to",
                 "accompany other widgets.")),
  column(3, numericInput("num", h3("Numeric input"), value = 1))
 ),

 fluidRow(
  column(3,
        radioButtons("radio", h3("Radio buttons"),
                     choices = list("Choice 1" = 1, "Choice 2" = 2,
                                    "Choice 3" = 3),selected = 1)),
 column(3,
        selectInput("select", h3("Select box"),
                    choices = list("Choice 1" = 1, "Choice 2" = 2,
                                   "Choice 3" = 3), selected = 1)),
 column(3,
        sliderInput("slider1", h3("Sliders"),
                    min = 0, max = 100, value = 50),
        sliderInput("slider2", "",
                    min = 0, max = 100, value = c(25, 75))
  ),
  column(3, textInput("text", h3("Text input"), value = "Enter text..."))
 )
)
```

Adding widgets

- You can add widgets to your web page in the same way that you added other types of HTML content

- To add a widget to your app, place a widget function in sidebarPanel or mainPanel in your ui object.

- Each widget function requires several arguments. The first two arguments for each widget are:
  - A **name** for the widget. The user will not see this name, but you can use it to access the widget's value. The name should be a character string.
  - A **label**. This label will appear with the widget in your app. It should be a character string, but it can be an empty string "".

- The remaining arguments vary from widget to widget, depending on what the widget needs to do its job. They include things the widget needs to do its job, like initial values, ranges, and increments.

5 Display reactive output

Some examples

- Reactive output automatically responds when your user toggles a widget.

- You can create reactive output with a two step process.
  - Add an R object to your user-interface.
  - Tell Shiny how to build the object in the server function. The object will be reactive if the code that builds it calls a widget value.

Step 1: Add an R object to the UI

- Shiny provides a family of functions that turn R objects into output for your user-interface. Each function creates a specific type of output.

Output function	**creates**
dataTableOutput	Data Table
htmlOutput	raw HTML
imageOutput	image
plotOutput	plot
tableOutput	table
textOutput	text
uiOutput	raw HTML
verbatimTextOutput	text

- You can add output to the user-interface in the same way that you added HTML elements and widgets. Place the output function inside sidebarPanel or mainPanel in the ui.

Example

```
ui <- fluidPage(
  titlePanel("censusVis"),

  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with information from the 2010 US Census."),

      selectInput("var",
        label = "Choose a variable to display",
        choices = c("Percent White", "Percent Black", "Percent Hispanic", "Percent Asian"),
        selected = "Percent White"),

      sliderInput("range", label = "Range of interest:", min = 0, max = 100, value = c(0, 100))
    ),

    mainPanel(
      # The object in the following line is not yet defined!
      textOutput("selected_var")
    ) ) )
```

Step 2: Provide R code to build the object.

- Placing a function in ui tells Shiny where to display your object. Next, you need to tell Shiny how to build the object.

- Do this by providing R code that builds the object in the server function.

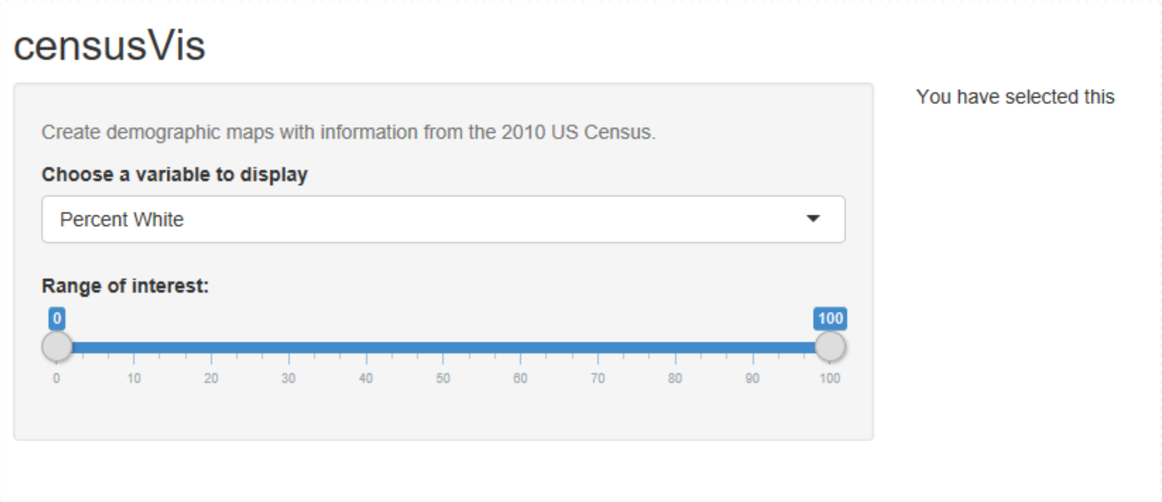
- The server function plays a special role in the Shiny process; it builds a list-like object named **output** that contains all of the code needed to update the R objects in your app. Each R object needs to have its own entry in the list.

- You can create an entry by defining a new element for output within the server function, like in the next slide. The element name should match the name of the reactive element that you created in the ui.

Example

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

  output$selected_var <- renderText({
    "You have selected this"
  })

}
```

Visual Result



censusVis

Create demographic maps with information from the 2010 US Census.

Choose a variable to display

Percent White

Range of interest:

You have selected this

Render Function

- You do not need to explicitly state in the server function to return output in its last line of code. R will automatically update **output**.

- Each entry to output should contain the output of one of Shiny's render* functions. These functions capture an R expression and do some light pre-processing on the expression. Use the render* function that corrresponds to the type of reactive object you are making.

render function	**creates**
renderDataTable	DataTable
renderImage	images (saved as a link to a source file)
renderPlot	plots
renderPrint	any printed output
renderTable	data frame, matrix, other table like structures
renderText	character strings
renderUI	a Shiny tag object or HTML

Render Function

- Each render* function takes a single argument: an R expression surrounded by braces, {}. The expression can be one simple line of text, or it can involve many lines of code, as if it were a complicated function call.

- Think of this R expression as a set of instructions that you give Shiny to store for later. Shiny will run the instructions when you first launch your app, and then Shiny will re-run the instructions every time it needs to update your object.

- For this to work, your expression should return the object you have in mind (a piece of text, a plot, a data frame, etc). You will get an error if the expression does not return an object, or if it returns the wrong type of object.

Use widget values

- If you run the server function script given in previous slides, the Shiny app will display "You have selected this" in the main panel. However, the text will not be reactive: it will not change even if you manipulate the widgets of your app.

- You can make the text reactive by asking Shiny to call a widget value when it builds the text.

- The server function mentions two arguments, **input** and **output.** You already saw that **output** is a list-like object that stores instructions for building the R objects in your app, **input** is a second list-like object. It stores the current values of all of the widgets in your app. These values will be saved under the names that you gave the widgets in the ui.

- Our example app has two widgets, one named "var" and one named "range". The values of "var" and "range" will be saved in input as input$var and input$range. Since the slider widget has two values (a min and a max), input$range will contain a vector of length two.

Reactive Object

- Shiny will automatically make an object reactive if the object uses an input value. For example, the server function below creates a reactive line of text by calling the value of the select box widget to build the text.

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

  output$selected_var <- renderText({
    paste("You have selected", input$var)
  })
}
```

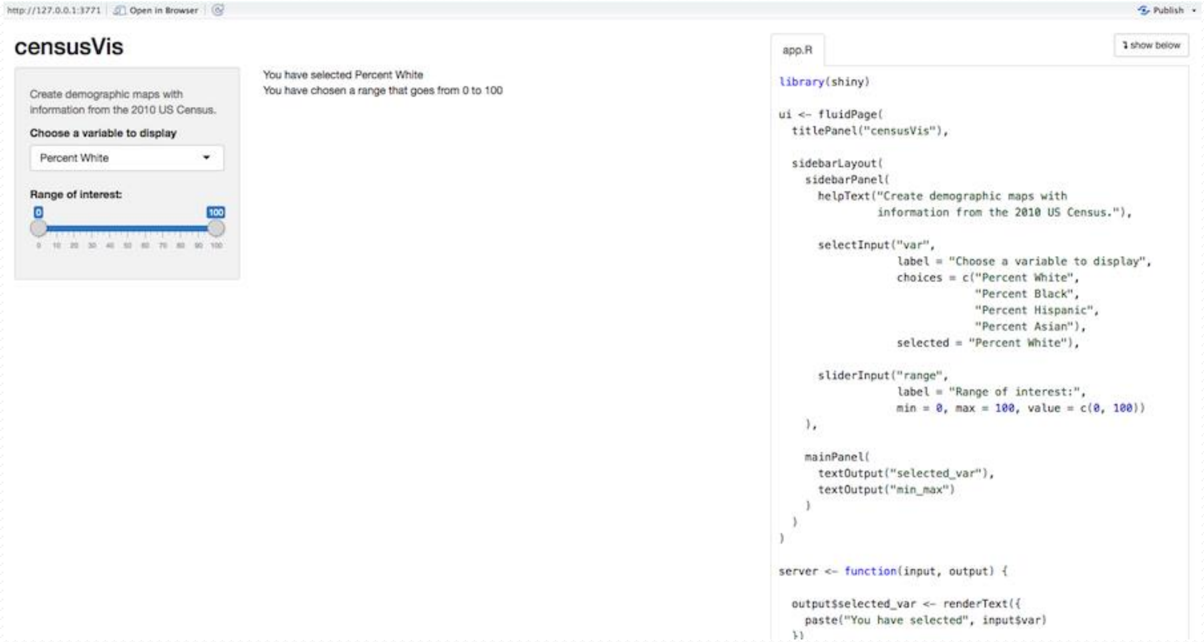
- Shiny tracks which outputs depend on which widgets. When a user changes a widget, Shiny will rebuild all of the outputs that depend on the widget, using the new value of the widget as it goes. As a result, the rebuilt objects will be completely up-to-date.

- This is how you create reactivity with Shiny, by connecting the values of input to the objects in output. Shiny takes care of all of the other details.

See the reactive output

- Launch your app and see the reactive output
- When you are ready, update your server and ui functions to match those above. Then launch your Shiny app by running runApp(‘myApp’, display.mode = "showcase") at the command line.
- Your app should look like the app with the scripts, and your statement should update instantly as you change the select box widget.
- Watch the server function. When Shiny rebuilds an output, it highlights the code it is running. This temporary highlighting can help you see how Shiny generates reactive output.

Reative App with Scripts



censusVis

Create demographic maps with information from the 2010 US Census.

Choose a variable to display

Percent White

Range of interest:

0 100

You have selected Percent White
You have chosen a range that goes from 0 to 100

```r
library(shiny)

ui <- fluidPage(
  titlePanel("censusVis"),

  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with
               information from the 2010 US Census."),

      selectInput("var",
                  label = "Choose a variable to display",
                  choices = c("Percent White",
                              "Percent Black",
                              "Percent Hispanic",
                              "Percent Asian"),
                  selected = "Percent White"),

      sliderInput("range",
                  label = "Range of interest:",
                  min = 0, max = 100, value = c(0, 100))
    ),

    mainPanel(
      textOutput("selected_var"),
      textOutput("min_max")
    )
  )
)

server <- function(input, output) {

  output$selected_var <- renderText({
    paste("You have selected", input$var)
  })
```

6 Use R scripts and data

Use R scripts and data

- How to load data, R Scripts, and packages to use in your Shiny apps?
- We will build a sophisticated app that visualizes US Census data: a code example is better than long explanations!

US Counties Example

- counties.rds is a dataset of demographic data for each county in the United States, collected with the UScensus2010 R package.

- Create a new folder named **data** in your “myApp” directory.

- Move counties.rds into the data folder.

Dataset Description

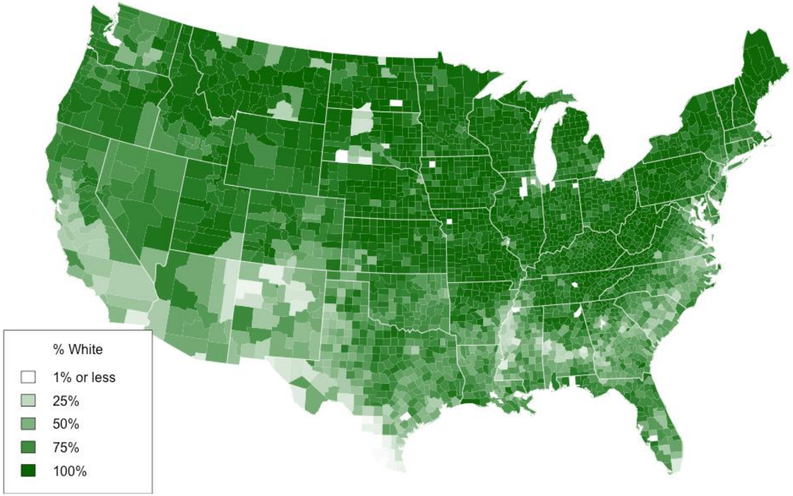
- The dataset in counties.rds contains
  - the name of each county in the United States
  - the total population of the county
  - the percent of residents in the county who are white, black, hispanic, or asian

```
> counties <- readRDS("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
```

Helpers.R

- helpers.R is an R script that can help you make choropleth maps.
- A choropleth map is a map that uses color to display the regional variation of a variable.
- In our case, helpers.R will create percent_map, a function designed to map the data in counties.rds.
- helpers.R uses the *maps* and *mapproj* packages in R. If you’ve never installed these packages before, you’ll need to do so before you make this application run.

Choropleth map



Helpers.R

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

  # 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]

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

  # 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))

  # 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)
}
```

Percent_map function

- The percent_map function in helpers.R takes five arguments:

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)

- You can use *percent_map* at the command line to plot the counties data as a choropleth map

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

How to call helpers.R

- You will need to ask Shiny to call the same functions before it uses *percent_map* in your app. Both *source* and *readRDS* require a file path, and file paths do not behave the same way in a Shiny app as they do at the command line.

- When Shiny runs the commands in App.R, it will treat all file paths as if they begin in the same directory as App.R. In other words, the directory that you save App.R in will become the working directory of your Shiny app.

- Since you saved helpers.R in the same directory as App.R, you can ask Shiny to load it with *source("helpers.R")*

- Since you saved *counties.rds* in a sub-directory (named data) of the directory that *App.R* is in, you can load it with: *counties <- readRDS("data/counties.rds")*

- You can load the *maps* and *mapproj* packages in the normal way with

  library(maps)

  library(mapproj)

Ui function

```
ui <- fluidPage(
  titlePanel("censusVis"),

  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with information from the 2010 US Census."),

      selectInput("var",
                  label = "Choose a variable to display",
                  choices = c("Percent White", "Percent Black",
                              "Percent Hispanic", "Percent Asian"),
                  selected = "Percent White"),

      sliderInput("range",
                  label = "Range of interest:", min = 0, max = 100, value = c(0, 100))
      ),

    mainPanel(plotOutput("map"))
  ) )
```

Server function

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

  output$map <- renderPlot({

      data <- switch(input$var,
                     "Percent White" = counties$white,
                     "Percent Black" = counties$black,
                     "Percent Hispanic" = counties$hispanic,
                     "Percent Asian" = counties$asian)

      color <- switch(input$var,
                      "Percent White" = "darkgreen",
                      "Percent Black" = "black",
                      "Percent Hispanic" = "darkorange",
                      "Percent Asian" = "darkviolet")

      legend <- switch(input$var,
                       "Percent White" = "% White",
                       "Percent Black" = "% Black",
                       "Percent Hispanic" = "% Hispanic",
                       "Percent Asian" = "% Asian")

      percent_map(var = data, color = color, legend.title = legend, max = input$range[2], min =
input$range[1])
    })
  }
```

Execution

- Shiny will execute all of these commands if you place them in your App.R. However, where you place them in the App.R will determine how many times they are run (or re-run).

- Shiny will run some sections of App.R more often than others.

- Shiny will run the whole script the first time you call runApp. This causes Shiny to execute the server function.

- Shiny saves the server function until a new user arrives. Each time a new user visits your app, Shiny runs the server function again, one time. The function helps Shiny build a distinct set of reactive objects for each user.

- As users change widgets, Shiny will re-run the R expressions assigned to each reactive object. If your user is very active, these expressions may be re-run many, many times a second.

```
ui <- fluidPage(

)

# A first place to put code

server <- function(input, output) {

    # A second place to put code

    output$map <- renderPlot({

        # A third place to put code

    })
}
```

7 Conclusion

Conclusion

- Shiny: very relevant and easy tool to create “tiny” and reactive applications in R
- Many possibilities are offered: reactive expression, style with CSS, etc.
- Customization is encouraged!
- Many ways to share the app
  - Transmitting the R files
  - Publishing as a web page.