TRUMAN STATE UNIVERSITY

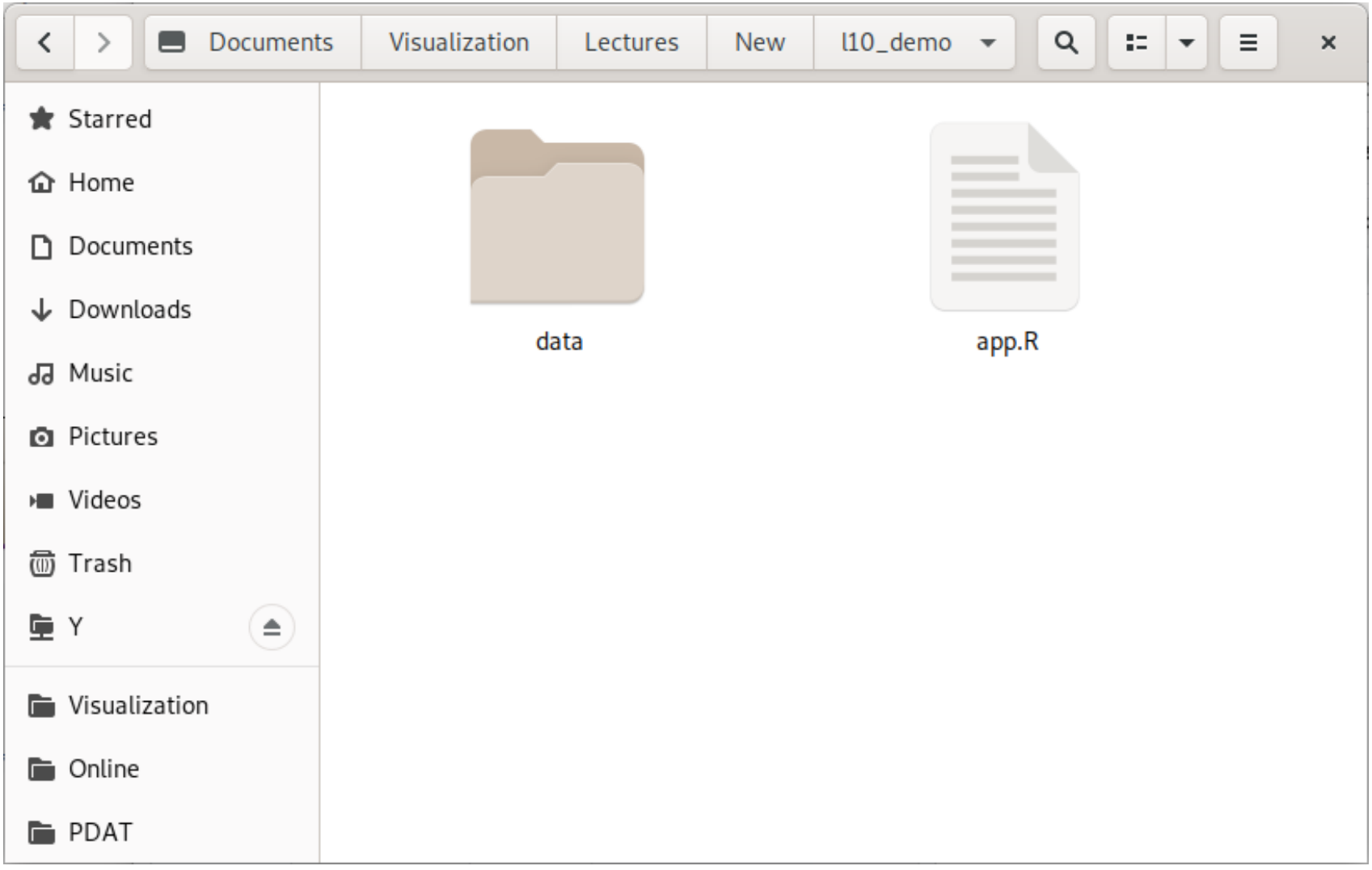
Department of STATISTICS

Introduction to Shiny Apps

DISTINCT BY DESIGN

Basic Structure of a Shiny App

Shiny apps are stored in a file folder as app.R and ancillary files.



The basic structure of app.R consists of a user interface and a server component.

```r
library(shiny)

# Define UI for application
ui <- fluidPage(
  titlePanel("Page Title"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

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

}

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

The UI is specified as a series of nested page components, separated by commas, and translated to HTML.

```
library(shiny)

# Define UI for application
ui <- fluidPage(
  titlePanel("Page Title"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

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

}

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

The server side is specified by a code block inside of {}'s that defines the `server` function.

```
library(shiny)

# Define UI for application
ui <- fluidPage(
  titlePanel("Page Title"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

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

}

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

Things that should happen once when the app starts can go outside UI and server functions.

```
library(shiny)
library(tidyverse)
storm <-
  read.csv("data/storm_2011.csv") %>%
  filter(State=="MO", Longitude > 0)

# Define UI for application
ui <- fluidPage(
  titlePanel("Page Title"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

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

}

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

We can add static elements using functions that correspond to HTML tags, or the `tag` function.

```
library(shiny)
storm <-
  read.csv("data/storm_2011.csv") %>%
  filter(State=="MO", Longitude > 0)

# Define UI for application
ui <- fluidPage(
  titlePanel("Missouri Weather Events."),
  sidebarLayout(
    sidebarPanel(
      h2("Select Month")
    ),
    mainPanel(
      h1("Location of Weather Events")
    )
  )
)
```

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

}

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

`render` functions create output on the server side, and `Output` functions display it in the UI.

```
library(shiny)
library(tidyverse)
storm <-
  read.csv("data/storm_2011.csv") %>%
  filter(State=="MO", Longitude > 0)

# Define UI for application
ui <- fluidPage(
  titlePanel("Missouri Weather Events."),
  sidebarLayout(
    sidebarPanel(
      h2("Select Month")
    ),
    mainPanel(
      h1("Location of Weather Events"),
      plotOutput("stormplot")
    )
  )
)
```

```
# Define server logic
server <- function(input, output) {
  output$stormplot <- renderPlot(
    ggplot(storm) +
    geom_point(aes(x=-Longitude,
                   y=Latitude))
  )
}

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

Note: Output is rendered as an element of the output list, and the name is quoted on the UI side.

```
library(shiny)
library(tidyverse)
storm <-
  read.csv("data/storm_2011.csv") %>%
  filter(State=="MO", Longitude > 0)

# Define UI for application
ui <- fluidPage(
  titlePanel("Missouri Weather Events."),
  sidebarLayout(
    sidebarPanel(
      h2("Select Month")
    ),
    mainPanel(
      h1("Location of Weather Events"),
      plotOutput("stormplot")
    )
  )
)
```

```
# Define server logic
server <- function(input, output) {
  output$stormplot <- renderPlot(
    ggplot(storm) +
    geom_point(aes(x=-Longitude,
                   y=Latitude))
  )
}

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

Input functions create controls on the UI side, and their values are read as elements of the input list in the server.

```
library(shiny)
library(tidyverse)
storm <-
  read.csv("data/storm_2011.csv") %>%
  filter(State=="MO", Longitude > 0)

# Define UI for application
ui <- fluidPage(
  titlePanel("Missouri Weather Events."),
  sidebarLayout(
    sidebarPanel(
      h2("Select Month"),
      sliderInput("month", "Month",
        min=1, max=12, value=1)
    ),
    mainPanel(
      h1("Location of Weather Events"),
      plotOutput("stormplot")
    )
  )
)
```

```
# Define server logic
server <- function(input, output) {
  output$stormplot <- renderPlot(
    storm %>%
    filter(Month==input$month) %>%
    ggplot() +
    geom_point(aes(x=-Longitude,
                   y=Latitude))
  )
}

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

Shiny Widgets

Buttons

Action `actionButton`

Submit `submitButton`

Single checkbox

Choice A `checkboxInput`

Checkbox group

Choice 1
Choice 2
Choice 3

`checkboxGroupInput`

Date input

2014-01-01

`dateInput`

Date range

2017-06-21 to 2017-06-21

`dateRangeInput`

File input

Browse... No file selected

`fileInput`

Help text

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

`helpText`

Numeric input

1

`numericInput`

Radio buttons

Choice 1
Choice 2
Choice 3

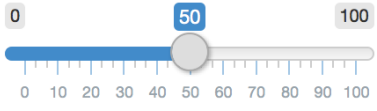
`radioButtons`

Select box

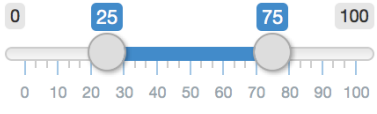
Choice 1

`selectInput`

Sliders



`sliderInput`



Text input

Enter text...

`textInput`

Original: https://shiny.rstudio.com/tutorial/written-tutorial/lesson3/

Modified from the RStudio Shiny tutorial (rstudio.com)

Library of basic `Output` and matching `render` functions.

Object	`render` Function	`Output` UI Command
Data Table (interactive)	`renderDataTable`	`dataTableOutput`
Data Frame, Matrix, etc.	`renderTable`	`tableOutput`
Images (as a link to file)	`renderImage`	`imageOutput`
Plot	`renderPlot`	`plotOutput`
Text (character strings)	`renderText`	`textOutput`
Printed R Output	`renderPrint`	`verbatimTextOutput`
User Interface (Shiny/HTML)	`renderUI`	`uiOutput/htmlOutput`

Adding More Widgets

- Date range
- Select box (State)
- Checkbox group (Storm Type)
- Select box (Other Storm Type)

Reactive expressions can be used to recalculate based on user actions.

- The `render` commands are reactive.
- The `reactive` command creates a new reactive value.
- When nothing has changed, a reactive value returns a cached value—a fast operation.
- Whenever a user input changes, any reactive value that depends on it is marked “invalidated.”
- Whenever a reactive value is invalidated, any other reactive elements are also marked “invalidated.”
- Reactive elements are recalculated only when “invalidated.”

Example: Filtering storm data only when state is changed.

- Assumptions:
  - Filtering data from all 50 states might be slow.
  - Users will change the state less often than date or storm type.
- Results:
  - `storm.st` is updated only when state changes.
  - Final `stormplot` is updated whenever any user input changes.

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

  storm.st <- reactive({
    storm %>%
      filter(State==input$state)
  })

  output$stormplot <- renderPlot({
    storm.st() %>%
      filter(Date >= ...) %>%
      filter(Date < ...) %>%
      filter(Evtype %in% ...) %>%
    ggplot() %>% ...
  })

}
```

Reactive calculations can change the user interface.

- We'd like to show the user the top 5 storm types for the selected state.
- `fct_lump_n` keeps the *n* most common factors, and recodes the rest as "Other".
- Use the `renderUI` and `uiOutput` pair to change the checkboxes.

```
ui <- fluidPage( ...
    sidebarPanel( ...
      uiOutput("evcheck") ...
    ), ...
)
```

```
server <- function(input, output) {
  ...
  storm.st <- reactive({
    storm %>%
      filter(State==input$state) %>%
      mutate(Evlump <-
        fct_lump_n(Evtype, 5))
  })
  output$evcheck <- renderUI(
    checkboxGroupInput(
      "events", "Choose Events",
      choices=levels(storm.st()$Evlump
    )
  )
  ...
}
```

Visit the RStudio Shiny Tutorial for more information!

Click here for the RStudio Shiny tutorial.
(shiny.rstudio.com/tutorial/)