Intermediate Programming in R Part II

Justin Post

August 16-17, 2018

What do we want to be able to do?

- Communicate findings effectively
- Document findings
- Make process reproducible
- Share process

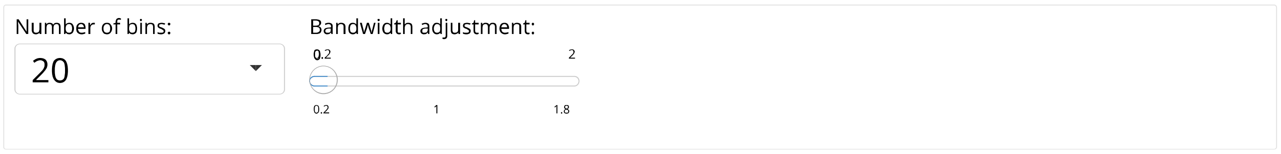
Where do we start?

- Review of Key Concepts
- R Markdown Basics
  - Code Chunks
  - Images/Equations/Misc.
- R Markdown Options
  - Documents: PDF, HTML
  - Presentations: Slides
  - Interactive Components
- R Shiny Applications/Presentations

What is R Shiny?

- R Shiny Package (http://shiny.rstudio.com/) allows for creation of interactive "web" applications in R
- Developed by RStudio
- Basically a folder with 2 R scripts:
  - ui.R (User Interface)
  - server.R (R functions that run/respond to UI)
- Requires no HTML, CSS, or JavaScript!

Example App



Why use R Shiny?

- If you know R, not too bad to learn
- Can be great way to
  - Share data analysis results
  - Allow user to explore data
  - Explain statistical concepts/teach

Ex: Multiple Linear Regression Idea

Explanatory Variable (x)

Response Variable (y)

☐ Fit Regression Equation?

Fitted Regression Equation

How to Develop R Shiny Apps

- Explore online repositories/resources for existing apps!
- Create a basic app
- Customize apps
- Learn solutions to common issues when creating apps
- Deploy the app

Available Apps

- Many available resources!!
- Apps I've created (http://www4.stat.ncsu.edu/~post/ShinyWorkshop/apps.html)
- Plenty of good examples
  - Shiny Showcase (https://www.rstudio.com/products/shiny/shiny-user-showcase/)
  - Shiny Gallery (https://shiny.rstudio.com/gallery/)
  - Stat Concepts (https://github.com/gastonstat/shiny-introstats/)
  - More Stat Concepts (https://www.researchgate.net/publication/298786680_Web_Application_Teaching_Tools_for_Statistics_Using_R_and_Shiny
  - Cal Poly (http://www.statistics.calpoly.edu/shiny)
- Take a few minutes to explore some apps!

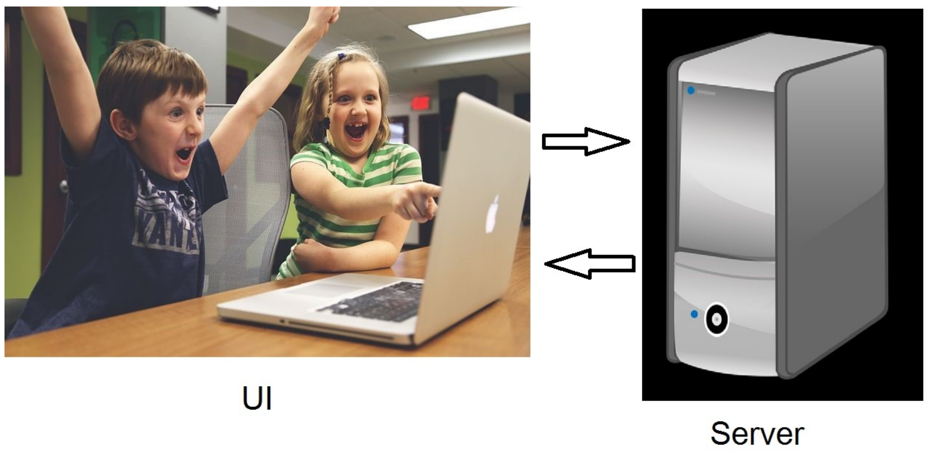
Getting Started - Basic Needs

- R, R studio
- `shiny` package
- Recommended other packages
  - `shinydashboard` (create dashboards)
  - `DT` (Nice Tables)

Elements of an App

- Each app has two things
  - User Interface (UI)
  - Server

App Duties



Elements of an App

- Each app has two things
  - User Interface (UI)
  - Server
- UI determines **layout** of app
  - Sets up widgets (items users can interact with)

Elements of an App

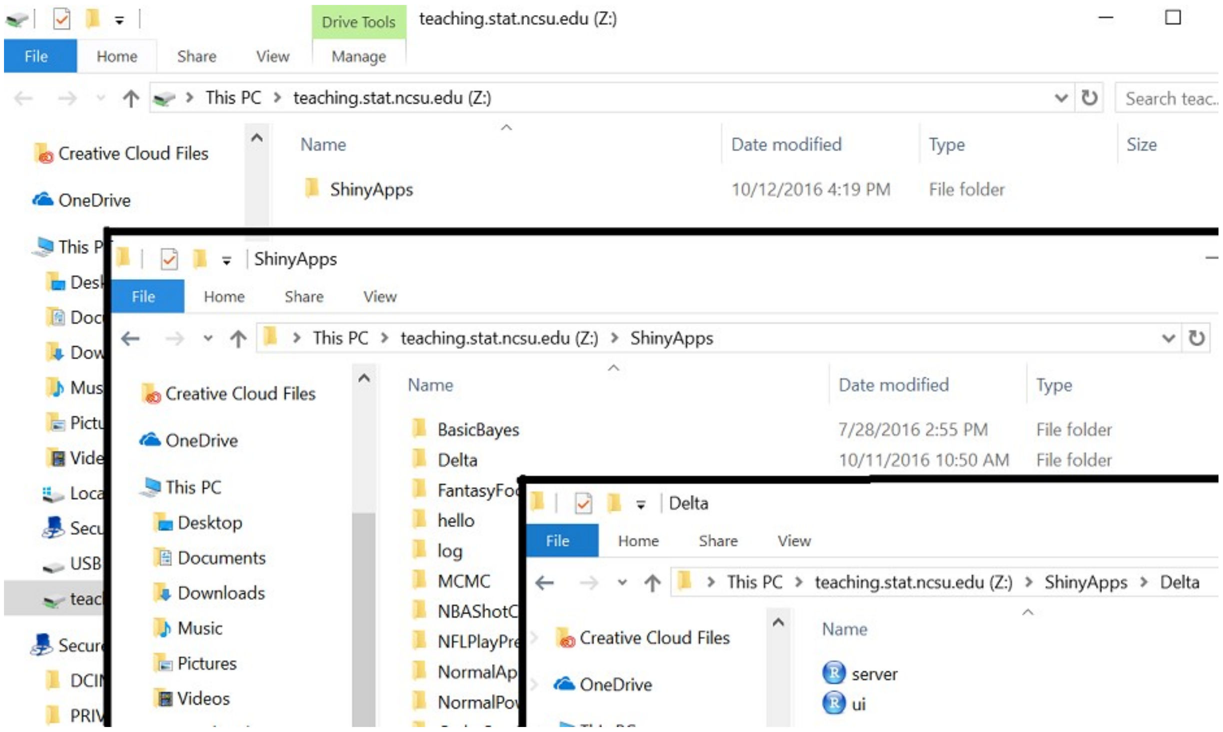
- Each app has two things
  - User Interface (UI)
  - Server
- UI determines **layout** of app
  - Sets up widgets (items users can interact with)
- Server contains R code to **run for the app**
  - Can include plots, model fitting, any R code really…

Elements of an App

- Each app has two things
  - User Interface (UI)
  - Server
- UI determines **layout** of app
  - Sets up widgets (items users can interact with)
- Server contains R code to **run for the app**
  - Can include plots, model fitting, etc.
- Can do with single file (`app.R`) but we'll use a separate file (`ui.R` and `server.R`)

Two File Approach (Recommended)

- Create folder for each App
- Each App's folder should have `ui.R` and `server.R` files
- If single file, `app.R`


Drive Tools
teaching.stat.ncsu.edu (Z:)
File Home Share View Manage
This PC > teaching.stat.ncsu.edu (Z:)
Search teac...
Creative Cloud Files
OneDrive
Name Date modified Type Size
ShinyApps 10/12/2016 4:19 PM File folder
ShinyApps
This PC > teaching.stat.ncsu.edu (Z:) > ShinyApps
This PC
Desktop
Documents
Downloads
Music
Pictures
Videos
Name Date modified Type
BasicBayes 7/28/2016 2:55 PM File folder
Delta 10/11/2016 10:50 AM File folder
hello
log
MCMC
Delta
This PC > teaching.stat.ncsu.edu (Z:) > ShinyApps > Delta
Name
server
ui

`ui.R` Basic Layout

```
library(shiny)

ui <- fluidPage(
    titlePanel(),

  sidebarLayout(
    sidebarPanel(#usually widgets
      ),
    mainPanel(#usually output
      )
  )
)
```

UI Common Layout

title panel

```
sidebar panel
|
|
|
|
|
|
V
Usually User Inputs
```

```
main panel content-------------------->
|
|
|
| Usually Output
| that Reacts to User Input
|
V
```

`server.R` Basic File

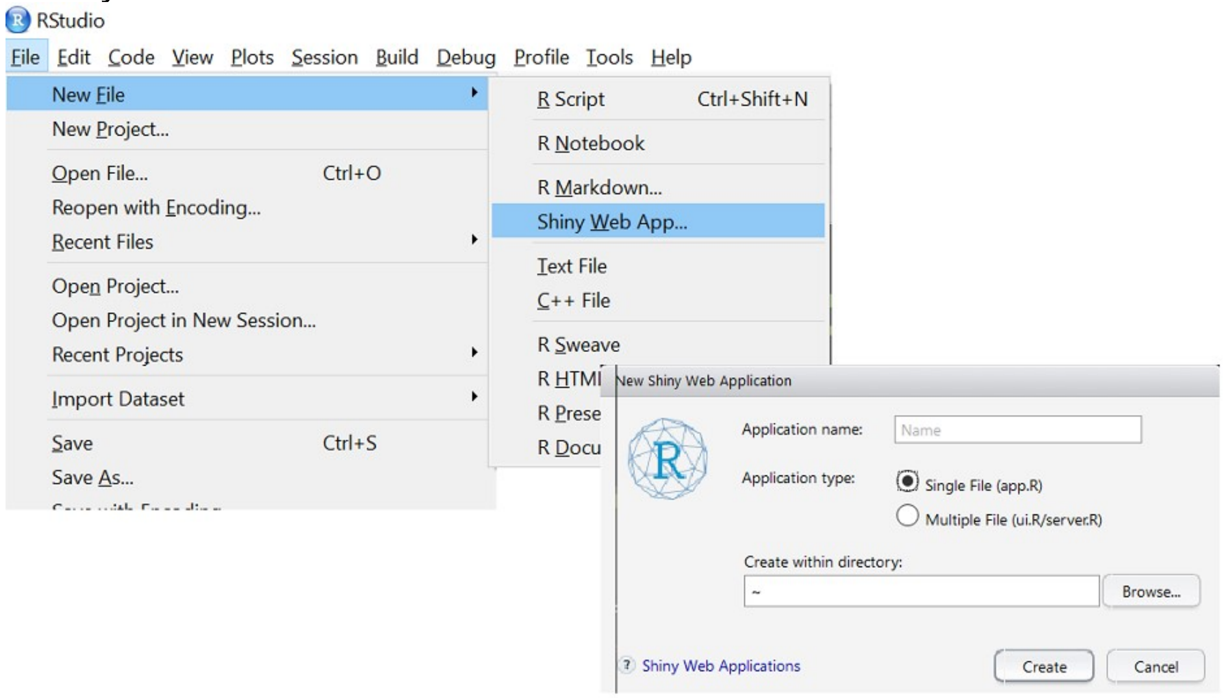
```
library(shiny)

shinyServer(function(input, output, session) {

})
```

Shiny Templates

Readily available in R studio



New Shiny Web Application

Application name: Name

Application type: [✓ Single File (app.R)] Multiple File (ui.R/server.R)

Create within directory: ~

21/107

Two File Template

```r
library(shiny)
ui <- fluidPage(
   # Application title
   titlePanel("Old Faithful Geyser Data"),
   # Sidebar with a slider input for 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("distPlot")
      )
   )
)
```

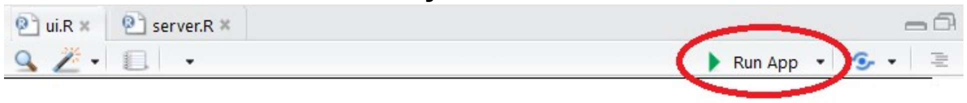
Two File Template

```
# Define server logic required to draw a histogram
server <- function(input, output) {
   output$distPlot <- renderPlot({
      # generate bins based on input$bins from ui.R
      x    <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)

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

Running an App

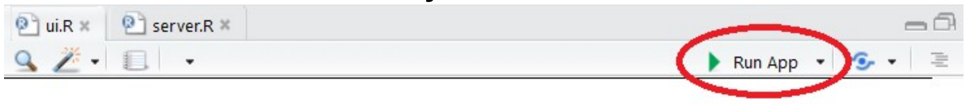
- While `ui.R` or `server.R` is your active window, click the **Run App** button



24/107

Running an App

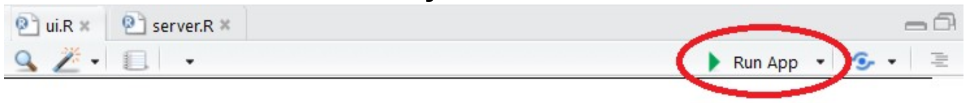
- While `ui.R` or `server.R` is your active window, click the **Run App** button



- Set working directory to ShinyApps folder
- Load `shiny` package
- Use `runApp()` function
    - ex: `runApp("normalPower")`

Running an App

- While `ui.R` or `server.R` is your active window, click the **Run App** button

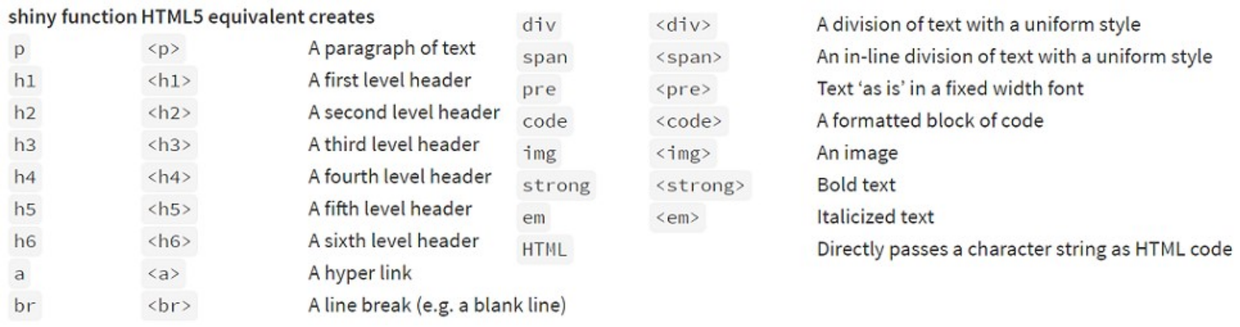


- Set working directory to ShinyApps folder
- Load `shiny` package
- Use `runApp()` function
  - ex: `runApp("normalPower")`
- Running App will tie up R console!
- End by hitting Esc or closing shiny app
- Take a minute and run the template app

Adding to the UI

Using a comma to separate items, you can add

- Any plain strings
- Widgets
- Formatted text (using HTML type 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

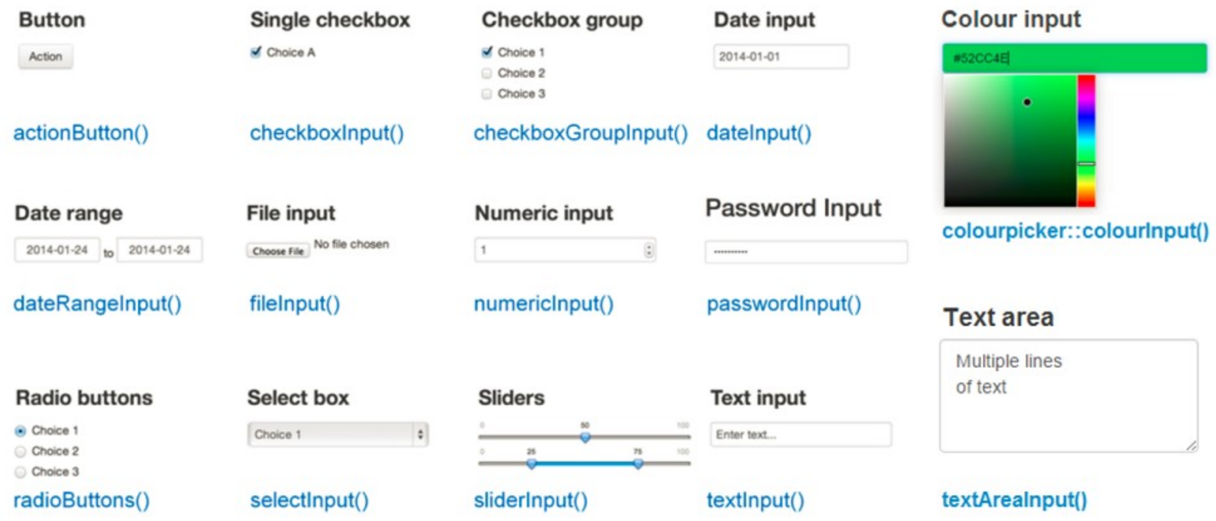
- Output from things created in the `server.R` file

Adding to the UI - Widgets

- Widgets can be added using their `*Input` functions
- Separate widgets (and other items) by commas in ui.R file

Adding to the UI - Widgets

- Widgets can be added using their `*Input` functions
- Separate widgets (and other items) by commas in ui.R file



Shiny Widgets for the UI

Widget

Button

Click Here!

Help Information for Widget

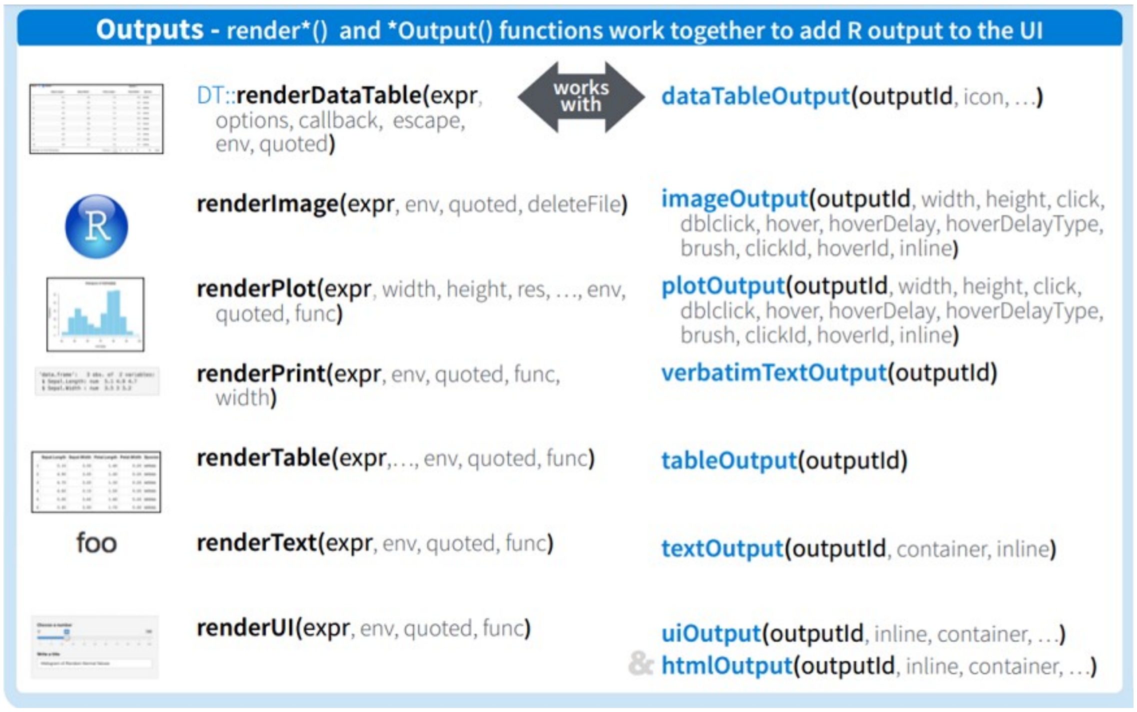
Code Used for Widget Above

What does Shiny return for use?

Sharing Between Server and UI

- Widgets are used to take input from the user
- Use their values in `server.R` (has your analysis or vis code!)
- Functions in `server.R` will create output to go in the `ui.R`

Sharing Between Server and UI

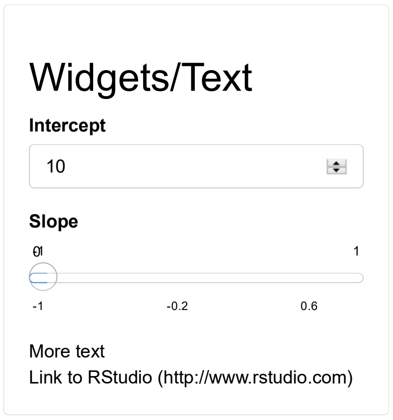


Outputs - render*() and *Output() functions work together to add R output to the UI

DT::**renderDataTable**(**expr**, options, callback, escape, env, quoted)	**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)	**uiOutput**(**outputId**, inline, container, ...) & **htmlOutput**(**outputId**, inline, container, ...)

Adding to the UI - Example Syntax

```
library(shiny)
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      h2("Widgets/Text"),
      numericInput("NI",label="Intercept",value=10),
      sliderInput("SI",label="Slope",min=-1,max=1,value=0,step=0.1),
      "More text",
      br(),
      a(href="http://www.rstudio.com",target="_blank","Link to RStudio")
    ),
    mainPanel(plotOutput("dataPlot"), #dataPlot is name of "plot" object in server
              textOutput("dataInfo"), #dataInfo is name of "text" object in server
              dataTableOutput("dataTable") #dataTable is name of "data" object in server
    )
  )
)
```


Widgets/Text

Intercept

10

Slope

More text
Link to RStudio (http://www.rstudio.com)

Summary So Far

`ui.r`

- Controls layout of app
- Basic layout uses a sidebar panel and a main panel
- Use strings, formatted (html style) text, widgets (`input*` functions), and output from `server.r` (`*Output` functions)
- Separate items with commas

Activity

- **UI Set-up Activity** instructions (http://www4.stat.ncsu.edu/~post/IntermediateR/UISetUpActivity.html) available on web
- Work in small groups
- Ask questions! TAs and I will float about the room
- Feel free to ask questions about anything you didn't understand as well!

Server file

`server.r` also called the 'back-end' because it works behind-the-scenes; its actions are not directly visible

```
## set up server
shinyServer(function(input, output, session) {
  # add stuff
})
```

Server file

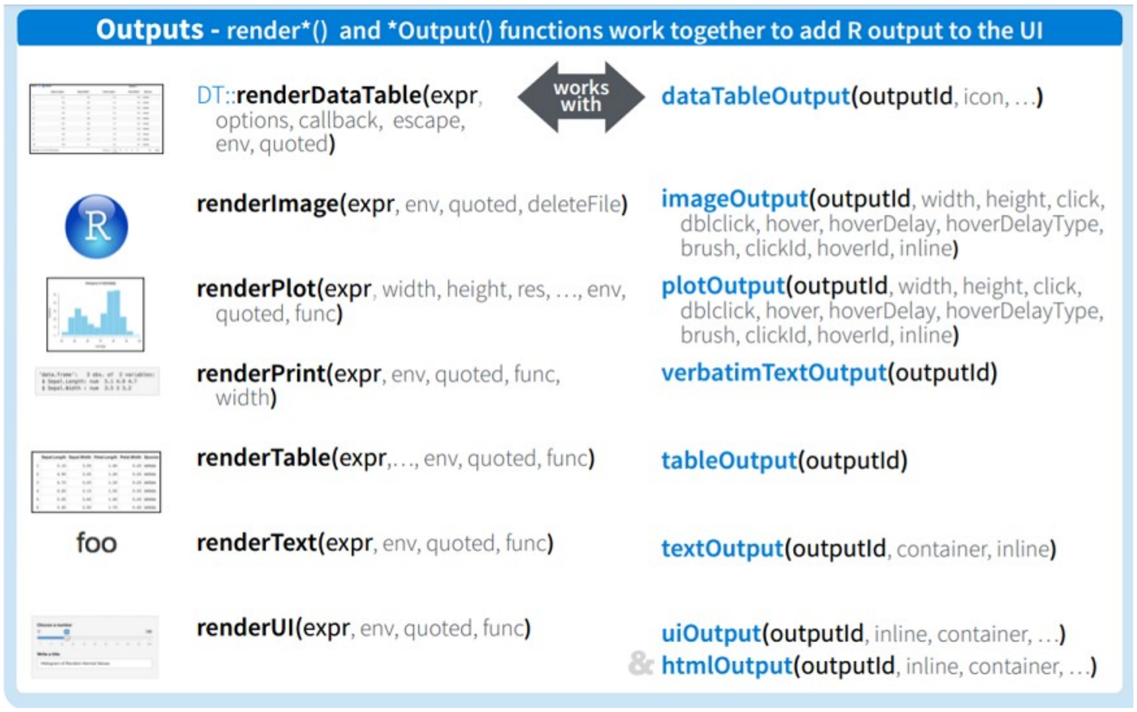
`server.r` also called the 'back-end' because it works behind-the-scenes

```
## set up server
shinyServer(function(input, output, session) {
  # add stuff
})
```

The arguments for the server are `input`, `output`, and `session`. Allow us to

1. Take in inputs from the UI
2. Run functions on them
3. Create outputs to send back

Creating Output to Send to UI



Outputs - render*() and *Output() functions work together to add R output to the UI

	works with	
DT::**renderDataTable(expr**, options, callback, escape, env, quoted**)**		**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**)**		**uiOutput(outputId**, inline, container, …**)** & **htmlOutput(outputId**, inline, container, …**)**

Creating Output to Send to UI

Example syntax

```
shinyServer(function(input,output){
  output$nameOfOutputObject<-renderPlot(
    #code that will return a plot
  )

  output$otherOutput<-renderText(
    #code that will return something that R can coerce to a string
  )
})

#in ui.r file, reference would look like
plotOutput("nameOfOutputObject")
textOutput("otherOutput")
```

Accessing Input Values in server.R

- Every input object has an `inputID`

Accessing Input Values in server.R

- Every input object has an `inputID`
- In `server.r`, reference input value by

```
input$inputID
```

Accessing Input Values in server.R

- Every input object has an `inputID`
- In `server.r`, reference input value by

```
input$inputID
```

- Example

```
#input widget code from ui.r file
sliderInput(inputID = "slide",label = "Select the Range Here",min = 0,max = 1,
            value = c(0,1))
#reference in server.r might look like
output$userPlot<-renderPlot({
  range<-input$slide
  #create plot that changes based on user input
  plot(data,xlim=range)
})
```

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them
- Notice how we name our output objects

```
output$nameOfOutputObject<-renderPlot(...))
```

Input and Output

- `input` and `output` objects are kind of like **lists**
- Shiny passes the information back and forth through them
- Notice how we name our output objects

```
output$nameOfOutputObject<-renderPlot(...))
```

- Notice how we access our inputs

```
output$nameOfOutputObject<-renderPlot(
   range<-input$slide
))
```

Quick Try

- Using the template app
- Add text output object in the `server.R` file (use `renderText`) that returns the current value of the input slider
- To do this, just reference the input (like an R function, it will return the last thing you do)
- Don't forget to add a `textOutput` in the `ui.R` file!

Reactivity

- Output objects do not have to depend on an input
- Those that don't will be static
- Any 'chunk' of code in `server.r` that references a user input must be **reactive**
- When a user changes the reference, the code **re-evaluates**

Example Reactivity

```
##code chunk "reacts" to and re-evaluates if
##input$sampleSize or input$otherInput changes

output$dataPlot<-renderPlot({

  n<-input$sampleSize
  input$otherInput #not used anywhere else, but entire
                   #renderPlot chunk still re-evaluates
                   #if changed

  hist(rbinom(n=1,size=n,prob=0.4))

})
```

- type `runApp("01_hello")` (load `shiny` library 1st: `library(shiny)`)

Reactivity

- Reactive variables (user inputs) can only be used in reactive contexts
- All `render*` functions are reactive contexts
- `server.r` can run any R code, but can't access inputs unless put into a reactive context

Error Using Reactive Variables

Following returns the error:

```
shinyApp(ui<-fluidPage(
                numericInput("NI","Give me a number",value = 10),
                textOutput("string")
                ),

         shinyServer(function(input,output){
                print(input$NI+10)
                output$string<-renderText(paste("value plus 10 is",input$NI+10))
         }
))
```

Warning: Error in .getReactiveEnvironment()$currentContext: Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)

Other Reactive Contexts

- `reactive({})` function allows for reactivity and creation of a new variable
- `observe({})` function allows for reactivity

```
shinyServer(function(input,output){

  #Creates a new reactive variable
  newVar<-reactive({
    value<-c(input$NI+10,input$NI*3)
  })

    #would now print to console
  observe({print(input$NI+10)})

  output$textString<-renderText({
    value<-newVar()  #access like a function!
    paste0("Input plus 10 is ",value[1]," and Input times 3 is ",value[2])
  })
}
```

More on `reactive({})`

- 'Wraps' a normal expression to create a reactive expression (code user can cause to change)
- Can read reactive values and call other reactive expressions
- Only re-evaluates *if necessary*
- Access object as though calling it as a function

More on `reactive({})`

- Access object as though calling it as a function

```
shinyServer(function(input,output){
  #Creates a new reactive variable
  newVar<-reactive({
    value<-c(input$NI+10,input$NI*3)
  })

  output$textString<-renderText({
    value<-newVar()  #access like a function!
    paste0("Input plus 10 is ",value[1]," and Input times 3 is ",value[2])
  })
}
```

More on `observe({})`

- Can read reactive values and call reactive expressions
- *Automatically* re-execute when dependencies change
- Doesn't yield a result
- Mostly used to update UI elements (more later)

```
shinyServer(function(input,output){
  #would now print to console
  observe({print(input$NI+10)})

    #update UI
    observe({
        input$noPitch
        updateCheckboxGroupInput(session, "pitchTypeChoice", selected = c(""))
    })
}
```

Summary So Far

`ui.r`

- Controls layout of app
- Basic layout uses a sidebar panel and a main panel
- Use strings, formatted (html style) text, widgets (`input*` functions), and output from `server.r` (`*Output` functions)
- Separate items with commas

`server.r`

- Back-end for app
- Create outputs that react to inputs (`render*` functions)
- To respond to input, must be in a reactive context

Devloping an App

- **Highly Recommended:**

Draw out what you want the app to look like

- Write R code to complete your app in a static manner!
- Translate to appropriate Shiny output functions

Activity

- **First Full App Activity** instructions (http://www4.stat.ncsu.edu/~post/IntermediateR/FirstAppActivity.html) available on web
- Work in small groups
- Ask questions! TAs and I will float about the room
- Feel free to ask questions about anything you didn't understand as well!

What do we want to be able to do?

- Communicate findings effectively
- Document findings
- Make process reproducible
- Share process

Dynamic UI

- Often want to update UI based on user input!
- Recall: UI and Server basically pass lists back and forth
- Methods for updating UI
  - `update*` functions
  - `renderUI()`/`uiOutput()`
  - `conditionalPanel()`

Using update* Functions

- Every input widget has a corresponding update function
  - `updateActionButton()`
  - `updateCheckboxInput()`
  - `updateNumericInput()`
  - …

Using update* Functions

- Every input widget has a corresponding update function
  - `updateActionButton()`
  - `updateCheckboxInput()`
  - `updateNumericInput()`
  - …
- Require session argument on server() function

```
shinyServer(function(input, output,session) {
  ## do stuff
})
```

Using update* Functions

- Every input widget has a corresponding update function
  - `updateActionButton()`
  - `updateCheckboxInput()`
  - `updateNumericInput()`
  - …
- Require session argument on server() function

```
shinyServer(function(input, output,session) {
  ## do stuff
})
```

- After all observers (reactive things) evaluate, updater sends message back to client

Using update* Functions

- Syntax of `update*` functions similar to the functions that created the inputs

Example syntax:

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA,
  width = NULL)

updateNumericInput(session, inputId, label = NULL, value = NULL,
  min = NULL, max = NULL, step = NULL)
```

Using update* Functions

- Syntax of `update*` functions similar to the functions that created the inputs

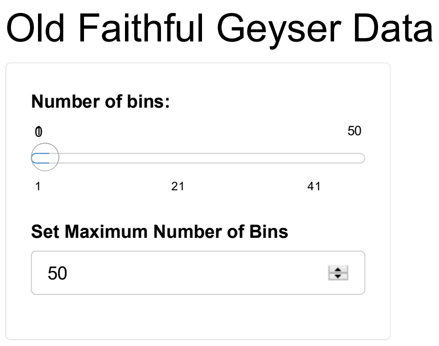
Example syntax:

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA,
  width = NULL)

updateNumericInput(session, inputId, label = NULL, value = NULL,
  min = NULL, max = NULL, step = NULL)
```

- Any arguments with `NULL` values ignored (i.e. will not result in any changes to the input object)
- For `radioButtons()`, `checkboxGroupInput()`, and `selectInput()`, the set of choices can be cleared by using `choices = character(0)` (similary for the set of selected)

Using `update*` Functions



Old Faithful Geyser Data

Number of bins:

Set Maximum Number of Bins

50

`updateSliderInput()` (First Attempt)

```
  ui <- fluidPage(
        ...
        sidebarPanel(
           sliderInput("bins","Number of bins:",min = 1,
                       max = 50,value = 30),
           numericInput("maxBins",label="Set Maximum Number of Bins",
                        value=50,min=1,max=100)
        ),
        ...
  ),
  server <- function(input, output,session) {
      ...
      updateSliderInput(session,"bins",max=input$maxBins)
  }
)
```

What is our issue?

`updateSliderInput()` (Fixed)

```
ui <- fluidPage(
      ...
      sidebarPanel(
         sliderInput("bins","Number of bins:",
                     min = 1,max = 50,value = 30),
         numericInput("maxBins",label="Set Maximum Number of Bins",
                      value=50,min=1,max=100)
      ),
      ...
)
server <- function(input, output,session) {
    ...
    observe({updateSliderInput(session,"bins",max=input$maxBins)})
}
```

68/107

`update*` UI Functions

- Use the template app
- Try to add a numeric input for the user to specify the largest value of the slider
- Use the `updateSliderInput` function to update the max of the slider
- Don't forget `observe`!

`renderUI()` and `uiOutput()`

- Alternatively, `renderUI()` and `uiOutput()` can be used

`renderUI()` and `uiOutput()`

- Alternatively, `renderUI()` and `uiOutput()` can be used
- Note: Shiny essentially writes HTML/JavaScript for us!

```
print(fluidPage(titlePanel(title="Hi"),
                sidebarLayout(sidebarPanel(),mainPanel())))
```

```
## <div class="container-fluid">
##   <h2>Hi</h2>
##   <div class="row">
##     <div class="col-sm-4">
##       <form class="well"></form>
##     </div>
##     <div class="col-sm-8"></div>
##   </div>
## </div>
```

`renderUI()` and `uiOutput()`

- Alternatively, `renderUI` and `uiOutput` can be used
- Note: Shiny essentially writes HTML/JavaScript for us!

```
print(numericInput("id","Label User Sees",value=10))
```

```
## <div class="form-group shiny-input-container">
##   <label for="id">Label User Sees</label>
##   <input id="id" type="number" class="form-control" value="10"/>
## </div>
```

`renderUI()` and `uiOutput()`

`renderUI()`

- Makes a **reactive version** of a function that generates HTML (like any widget)
- Have `renderUI()` return a shiny 'tag object,', HTML, or a list of these

- Use with `uiOutput()` in UI file
- Interprets the HTML and outputs appropriately (a `div` element)

`renderUI()` and `uiOutput()` (using widgets)

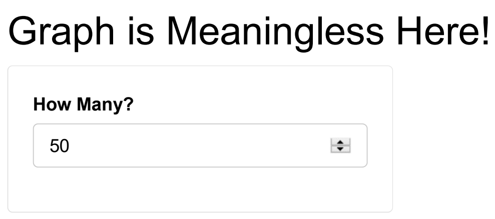
```
ui <- fluidPage(
      ...
      sidebarPanel(
          uiOutput("slider"),
          numericInput("maxBins",label="Set Maximum Number of Bins",
                       value=50,min=1,max=100)
      ),
      ...
),
server <- function(input, output,session) {
    ...
    output$slider<-renderUI({
        sliderInput("bins","Number of bins:",min = 1,
                    max = input$maxBins,value = 30)
        })
}
```

74/107

`renderUI()` and `uiOutput()` (using HTML)

```
ui <- fluidPage(
      ...
      sidebarPanel(
           uiOutput("info"),
           numericInput("purchase",label="How Many?",
                        value=50,min=0,max=100)
      ),
      ...
),
server <- function(input, output,session) {
    ...
      output$info<-renderUI({
        text<-paste0("You have selected to buy ",input$purchase)
        h3(text)
      })
}
```

`renderUI()` and `uiOutput()` (using HTML)



Graph is Meaningless Here!

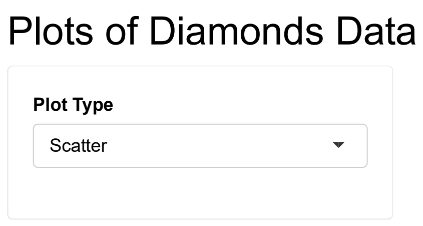
`renderUI()` and `uiOutput()` (using HTML)

- Use the template app
- Try to add some dynamic updating text to the UI

`conditionalPanel()`

- Create a 'panel' that is only visible if a condition is met
- Condition can depend on input or output value
- Accessed differently! (Use a '.' not a '$')

`conditionalPanel()`


Plots of Diamonds Data

Plot Type

Scatter

conditionalPanel()

```
...
sidebarPanel(
  selectInput("plotType", "Plot Type",
            c(Scatter = "scatter",Histogram = "hist")),

  # Only show this panel if the plot type is a histogram
  conditionalPanel(condition = "input.plotType == 'hist'",
          selectInput("breaks", "Breaks",
               c("Sturges","Scott","Freedman-Diaconis","[Custom]" = "custom")),

      # Secondary conditonalPanel, Only show this panel if Custom is selected
      conditionalPanel(
          condition = "input.breaks == 'custom'",
          sliderInput("breakCount", "Break Count", min=1, max=200, value=40)
      )
  )
)
```

`conditionalPanel()`

- Use the template app
- Try to add a new UI element if a condition of the slider is met

Dynamic UI Recap

- Often want to update UI based on user input!
- Recall: UI and server basically pass lists back and forth
- Methods for updating UI
  - `update*` functions
  - `renderUI()`/`uiOutput()`
  - `conditionalPanel()`

More Advanced UI Layout

- Contents of UI wrapped in `fluidPage()`
- Content can be wrapped in `fluidRow()`'s
- Columns can be created
- Should sum to 12 in total width!

Customized Layout

fluidRow with columns

2nd column

column widths in a given row should add to 12

2nd fluidRow below above row

Columns can contain their own fluidRow as well, allowing for a lot of customization of layouts!

subcol subcol

```
shinyUI(fluidPage(
  fluidRow(
    column(2,"fluidRow with columns--------...---------"),
    column(6,"2nd column------------...--------"),
    column(4,"column widths in a given row must add to 12------...---------")),
  fluidRow(tags$hr()),
  fluidRow(
    column(6,"2nd fluidRow below above row----...-----"),
    column(6,
           fluidRow("Columns can contain their own fluidRow as well, allowing for a lot o
           fluidRow(
             column(3,"subcol ----...-----"),
             column(9,"subcol ----...-----")
           ))
  )
))
```

Recap

`ui.r`

- Controls layout of app (can use standard layouts or customize)
- Use strings, formatted (html style) text, widgets (`input*` functions), and output from `server.r` (`*Output` functions)
- Separate items with commas
- Update inputs, render HTML reactively, conditionally show input

`server.r`

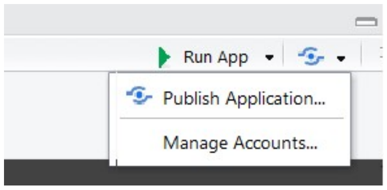
- Back-end for app
- Create outputs that react to inputs (`render*` functions)
- To respond to input, must be in a reactive context
- Code can be included prior to `shinyServer()`

Activity

- **Dynamic UI App Activity** instructions (http://www4.stat.ncsu.edu/~post/IntermediateR/DynamicUIActivity.html) available on web
- Work in small groups
- Ask questions! TAs and I will float about the room
- Feel free to ask questions about anything you didn't understand as well!

Sharing App

- Running App locally ties up your system
- Others can't access it!
- Can host apps on shinyapps.io (powered by RStudio)
  - Free, but number of connects and hours limited
  - Gives stats about usage
  - Integrated into R Studio



Sharing App

Another option:

- Host App on Shiny Server
- Costs money!
  - Free for academic use (I believe)
- Might need IT help to utilize

What do we want to be able to do?

- Communicate findings effectively
- Document findings
- Make process reproducible
- Share process

Where do we start?

- Review of Key Concepts
- R Markdown Basics
  - Code Chunks
  - Images/Equations/Misc.
- R Markdown Options
  - Documents: PDF, HTML
  - Presentations: Slides
  - Interactive Components
- R Shiny Applications/Presentations

Time Permitting: Miscellaneous Useful Things

Code can be placed prior to shinyServer

```
##Code here that you only need to evaluate once.
##This can include reading in data, creation of
##     functions common to all sessions, and
##     reading of other common r scripts.

shinyServer(function(input, output) {

##Code here that can be reactive.  Differs for
##     every instance of your app that runs.

})
```

Other Useful Things

Including Other Files

```
## top of server.R, output from here is common to all users

#data set only read in once
dat <- read_csv("dataset.csv")

#function created and not modified
helper <- function(item1, item2) {item1 + item2}

shinyServer(function(input, output) {
  ##reactive things, instance of app dependent
})
```

Other Useful Things

Including Other Files

If you have a lot of code, you can read in a separate script

Other Useful Things

Including Other Files

If you have a lot of code, you can read in a separate script

- If external script is `helpers.R` in same folder as app:

```
## top of server.R
source("helpers.R")

shinyServer(function(input, output) {
  ## do stuff
})
```

Other Useful Things

- Return `NULL` to remove errors when loading things
- Can use `isolate()` to improve code efficiency

```
observe({
  input$saveButton  # Do take a dependency on input$saveButton

  # isolate a whole block
  data <- isolate({
    a <- input$valueA   # No dependency on input$valueA or input$valueB
    b <- input$valueB
    c(a=a, b=b)
  })
  writeToDatabase(data)
})
```

Other Useful Things

- Improved data tables with `DT` package!
- DT example (http://shiny.stat.ncsu.edu/jbpost2/NBAShotChart/)

- Improved plots with `plotly` package!
- plotly example (http://shiny.stat.ncsu.edu/jbpost2/RegVis/)

Other Useful Things

- Can add in Latex easily!
- Include `withMathJax()` as a UI argument
- Calls in javascript that will replace Latex source code
- Must open in browser to render!

```
fluidRow(
    #add in latex functionality if needed
    withMathJax(),
          ...
```

Other Useful Things

- Can add tabs to your apps!
- Create "dashboards" with `shinydashboard` package
- Tab and Dashboard example (http://shiny.stat.ncsu.edu/jbpost2/OrderStatsDist/)

- Use mouse over and click inputs!
- Click Input Example (http://shiny.stat.ncsu.edu/jbpost2/BasketballCharting)

- Include Shiny in your Markdown slides!
- Use ioslides and add `runtime: shiny`

Other Useful Things

- shinythemes (https://rstudio.github.io/shinythemes/) are available
- shinyjs package (https://github.com/daattali/shinyjs) adds more functionality
- Can grab apps from GitHub (https://github.com/rstudio/shiny_example)
- List of all functions (https://shiny.rstudio.com/reference/shiny/latest/) for the UI and server
- Lots of good tutorials!
  - Shiny tutorials (https://shiny.rstudio.com/tutorial/)
  - Dean Attali (http://deanattali.com/blog/building-shiny-apps-tutorial/)
  - Shiny Articles (http://shiny.rstudio.com/articles/)
- R Shiny Cheat Sheet (http://shiny.rstudio.com/images/shiny-cheatsheet.pdf)

Debugging

- Much harder in shiny!
- Recommendation: Get static working code, then transfer to shiny

Debugging

- Can use `observe({print(...)})`

```
shinyServer(function(input,output){

  #would now print to console
  observe({print(input$NI+10)})

}
```

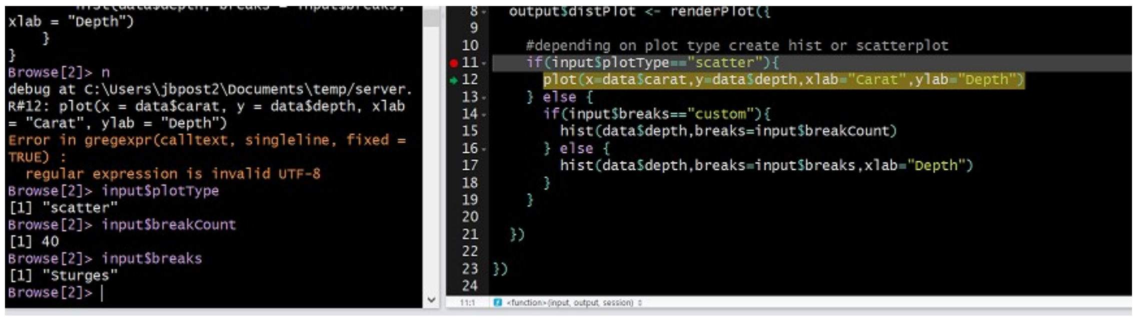
Debugging

R studio gives (http://shiny.rstudio.com/articles/debugging.html) three major approaches:

1. Breakpoints - Pausing execution of your program
2. Tracing - Collecting information as your program runs
3. Error handling - Finding the source of errors (both on the client and server side) and ascertaining their cause.

Breakpoints

- Can be used in `server.r`
- Click to the left of the line number



```
xlab = "Depth")
    }
}
Browse[2]> n
debug at C:\Users\jbpost2\Documents\temp/server.
R#12: plot(x = data$carat, y = data$depth, xlab
= "Carat", ylab = "Depth")
Error in gregexpr(calltext, singleline, fixed =
TRUE) :
  regular expression is invalid UTF-8
Browse[2]> input$plotType
[1] "scatter"
Browse[2]> input$breakCount
[1] 40
Browse[2]> input$breaks
[1] "Sturges"
Browse[2]> |
```

```
 8   output$distPlot <- renderPlot({
 9
10     #depending on plot type create hist or scatterplot
11     if(input$plotType=="scatter"){
12       plot(x=data$carat,y=data$depth,xlab="Carat",ylab="Depth")
13     } else {
14       if(input$breaks=="custom"){
15         hist(data$depth,breaks=input$breakCount)
16       } else {
17         hist(data$depth,breaks=input$breaks,xlab="Depth")
18       }
19     }
20
21   })
22
23 })
24
```

- Now can access values and step through program
- Can also use browser()

Tracing

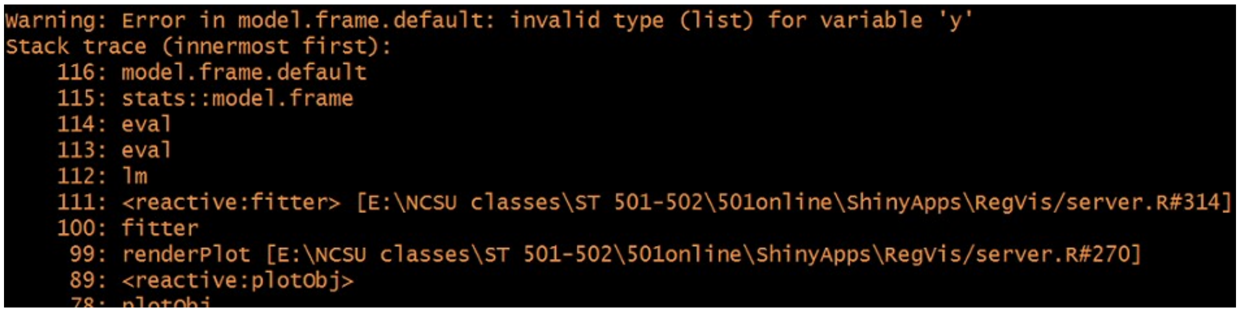
- Can run apps in 'showcase mode' (http://shiny.rstudio.com/gallery/kmeans-example.html)
- Invoke your app with the code below

```
shiny::runApp(display.mode="showcase")
```

- Also a reactive log that can be viewed

Error Handling

- Check stack trace shiny returns



```
Warning: Error in model.frame.default: invalid type (list) for variable 'y'
Stack trace (innermost first):
    116: model.frame.default
    115: stats::model.frame
    114: eval
    113: eval
    112: lm
    111: <reactive:fitter> [E:\NCSU classes\ST 501-502\501online\ShinyApps\RegVis/server.R#314]
    100: fitter
     99: renderPlot [E:\NCSU classes\ST 501-502\501online\ShinyApps\RegVis/server.R#270]
     89: <reactive:plotObj>
     78: plotObj
```

106/107

Enter Debug Mode on Error

- Can make Shiny enter the debugger when an error occurs by using the following statement:

```
options(shiny.error = browser)
```

- Overall, experience helps!