"Intro to Shiny" Workshop

Chris Payne

May 3, 2017

Shiny

Shiny is an R package that allows you to create interactive components (e.g., plots) that can be displayed as an html app. Shiny offers two unique features:

- Contains commands for taking in user input from an HTML user interface
- Updates plots automatically when user input changes.

Shiny apps have 2 main components:

1. User-interface (**UI**) script
2. **Server** script

The **User-interface script** (which can be saved as ui.R) controls the layout and appearance of the app/web-page.

- It will include Shiny-specific commands such as `sidebarLayout()`, `mainPanel()`, etc.

The **Server script** (server.R) has the R code needed to actually *build* the app.

Note: Although earlier versions of Shiny required that the UI and Server scripts were separate .R files, current versions allow for all code to be in a single .R file.

- For convenience sake, however, I will still refer to these "sections" as **ui.R** and **server.R**.

Getting Started

The first step is to install the Shiny package.

```r
install.packages("shiny", repos="http://cran.rstudio.com/")
library(shiny)
```

If you didn't believe me before, here is the simplest working Shiny app:

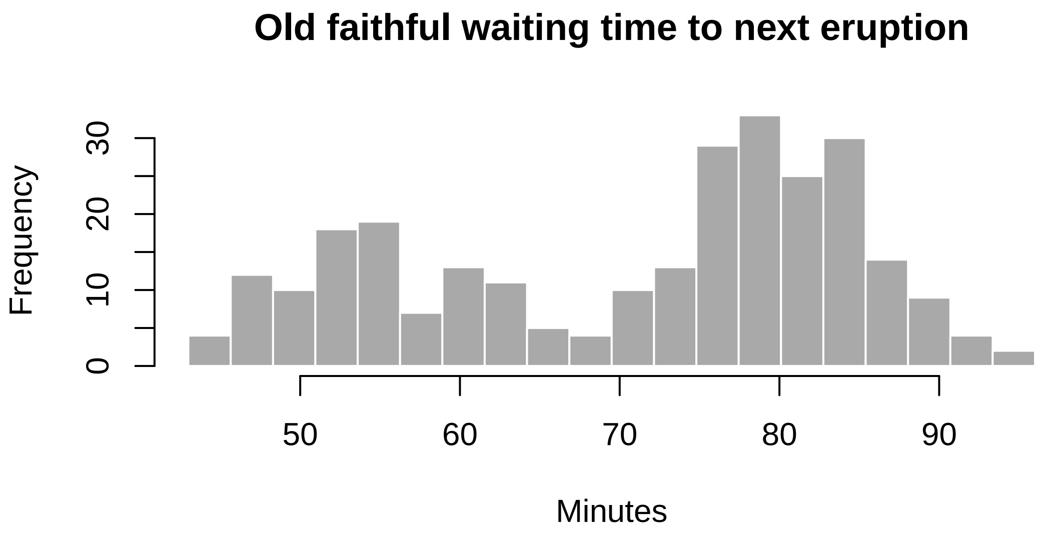
```r
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

Note: it's completely blank (but error free!). We can **use it as a template to build additional apps**.

Simple Example

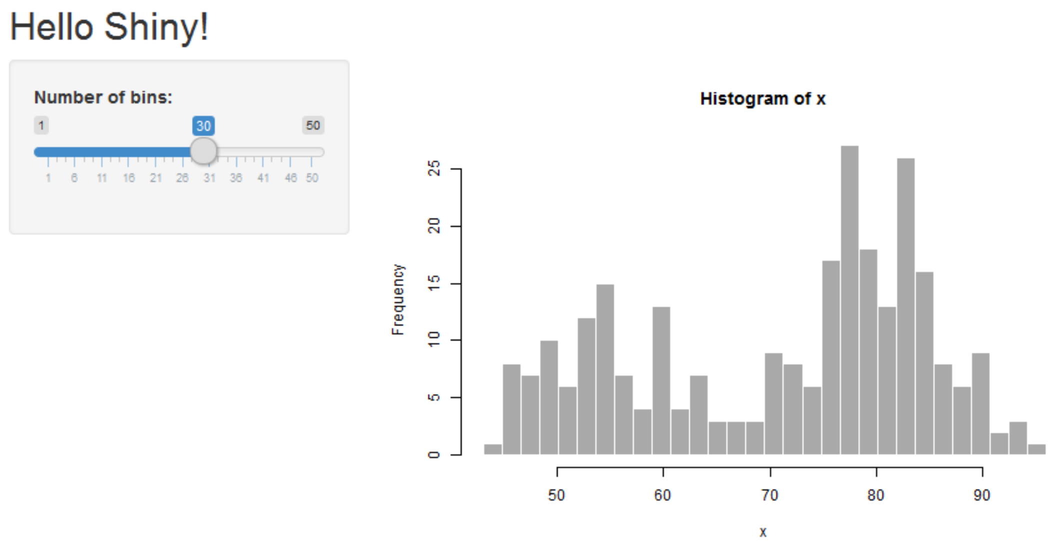
Shiny has a few built-in examples to show you its capabilities. We'll start by looking at a histogram that's made interactive by allowing you to change the number of bins. Let's review the code to generate a histogram.

```r
x <- faithful$waiting ## waiting times using internal dataset "faithful"
bins <- seq(min(x), max(x), length.out=20+1)
hist(x, breaks=bins, col="darkgray", border="white",
     main="Old faithful waiting time to next eruption", xlab="Minutes")
```



We can change the bins manually by increasing/decreasing the number of breaks, but Shiny allows the user to change the number of bins in *real-time*. The following code runs/opens this interactive example (which also conveniently provides its source code for demonstration).

```r
library(shiny)
runExample("01_hello")
```



This example app uses the input from the user (number of bins), and the `renderPlot()` function to allow the plot to update automatically.

Let's Walk Through a More Thorough Example:

The Data:

We'll be using the **CO2** data set that comes default with R. (note: use `data()` to see a list of all available datasets).

- The CO2 data set (84 rows, 5 columns) comes from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*.
- Experiment: CO_2 uptake of 12 plants (6 each from Quebec and Mississippi) was measured at several levels of ambient CO_2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.
- Let's examine the structure of the data:

```r
str(CO2)
```

I've chosen this data set because it's complex enough to use as a thorough example but simple enough for everyone to (hopefully) follow along.

For convenience sake, I'm going to rename the CO2 data set as `dat` while simultaneously creating a numeric ID# for each unique plant:

```r
dat <- data.frame(ID = as.numeric(CO2$Plant),CO2)
```

Start with a Basic App:

We'll start with a scatter plot showing CO_2 uptake as a function of CO_2 concentration. Our app will be able to control which points are included in the plot based on the plants' `Type` argument.

UI.R:

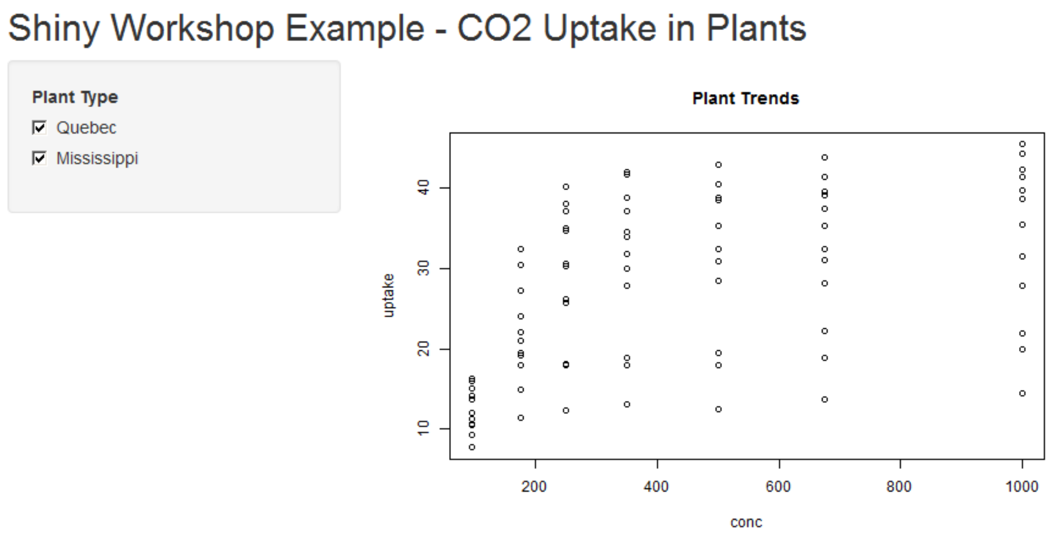
```r
ui <- fluidPage(  ## creates display that auto adjusts to user's device dimensions
  # Main title
  titlePanel("Shiny Workshop Example - CO2 Uptake in Plants"),
  # Establish sidebar layout:
  sidebarLayout(
    # Create sidebar (which we'll fill with input controls)
    sidebarPanel(
      # Create a check box input control allowing multiple items to be checked
      checkboxGroupInput(inputId = "type", label = "Plant Type", choices = levels(dat$Type),
                         selected = levels(dat$Type))
                         ## Note: levels() shows all unique "names" of a class = factor object
    ),
    # Spot for the plot
    mainPanel(
      plotOutput(outputId = "scatter.plot")
      ## Note: "scatter.plot" output object is created in server.R
    )
))
```

Server.R:

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

  # renderPlot indicates that the function is "reactive"
  # It will automatically re-execute when input changes
  output$scatter.plot <- renderPlot({

    # Render the plot:
    ##Create a stable plot size using all data before adding [changing] subsets using points().
    plot(uptake ~ conc, data = dat, type = "n")  ## type = "n" causes no points to be drawn.
    points(uptake ~ conc, data = dat[dat$Type  %in% c(input$type),])
    title(main = "Plant Trends")
    })
}
```

Now **combine the `UI` code and the `Server` code** into a working app:

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



In this example, we can choose to select which "plant type"" we view in the plot.

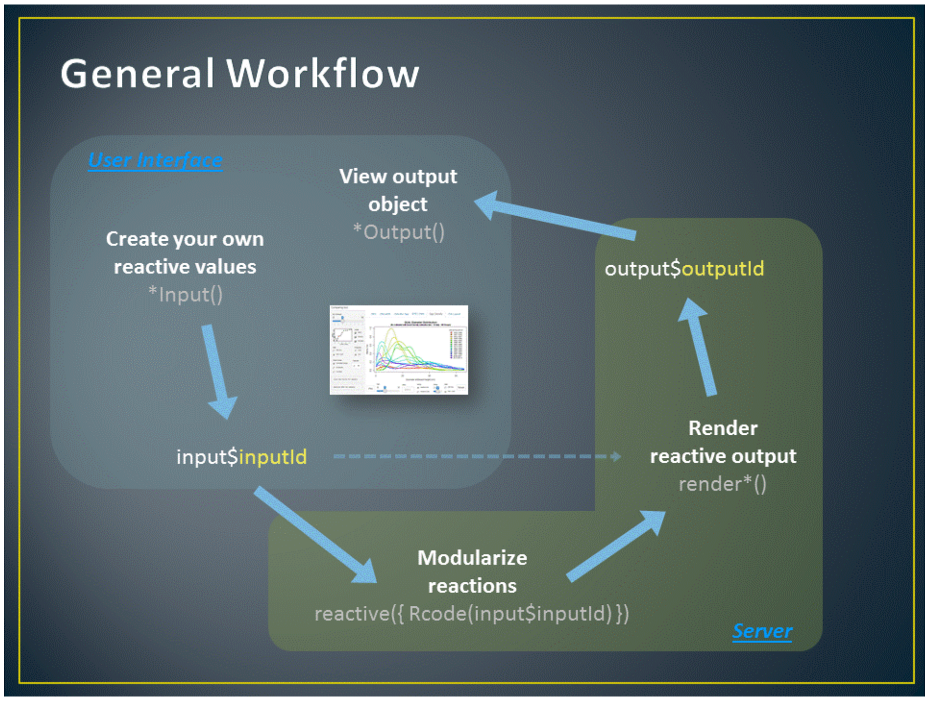
- This is accomplished by sub-setting `dat` by rows in which `dat$Type` equals whichever type we have checked in our sidepanel (provided by `input$type`).

Three Things to Note:

1. Code structuring in both scripts
   - Each object of code in ui.R acts as an argument of `fluidPage`. As a result, each of these arguments *needs* to be **separated by a comma**.
   - Code in server.R functions just like normal R code (with the addition of reactive objects), and therefore separate chunks of code do not need to be separated in any unique way.
2. The use of `{ }` in server.R
   - Just like when creating custom functions or for-loops in R, we use `{ }` inside reactive functions to denote that everything within those brackets *can* be changed based on changing inputs.

- Like functions and loops, you can put endless code within the brackets with the final object being the "main" output of the reactive function.
- Note that not all code within these reactive functions (i.e., within the curly brackets) needs to be directly impacted by inputs.

3. Inputs and Outputs:
   - All manipulative inputs (i.e., the results of `input()` functions) are assigned an **inputID** in ui.R, which are referenced in server.R as `input$inputId`.
   - All outputs we wish to view in the App are assigned as `output$outputId` using a `render*()` function in server.R. These outputs are referenced with their **outputId** in `output()` functions in ui.R.



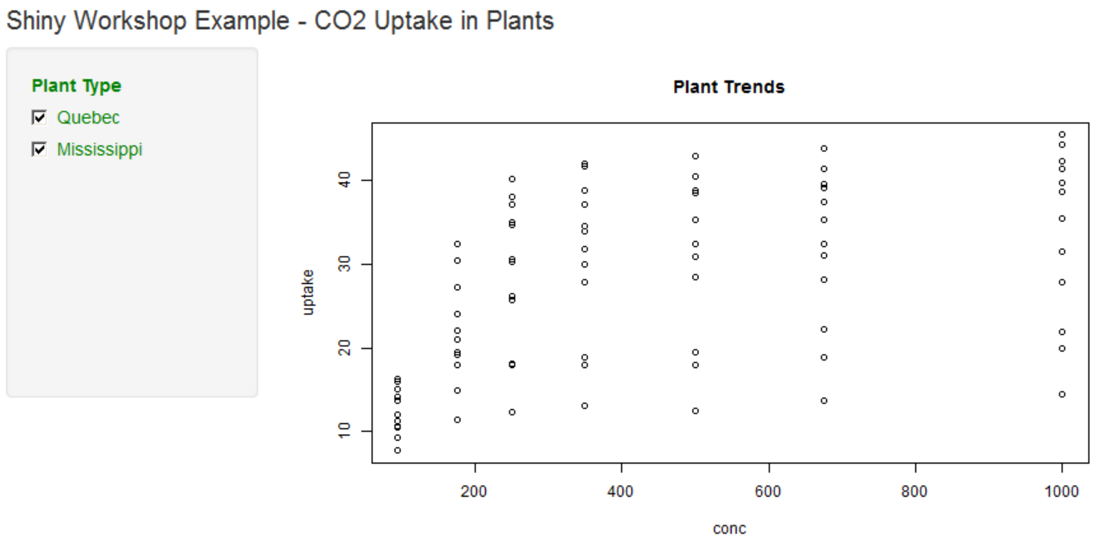
We'll Progressively Make This App More Complex...

First, Let's Add Some Styling:

HTML styling occurs as arguments within tags (or tag helper functions) and takes the form of: `style = "property:value;"`

- Where different `property:value` combos are separated by semicolons and all combos are held within quotes.
- Styles can be added directly to .R script in numerous ways:
  1. Inline (within tag or tag helper function): `p("Some Text", style = "font-size: 15px; color:blue;")`
  2. Applied to whole sections: `div(style = "color:green; padding:10px 0px 10px 0px;",...code to apply to...)`

```
ui <- fluidPage(  ## creates display that auto adjusts to user's device dimensions
  # Main title
  titlePanel(
    ## paragraph of size-20 text:
    p("Shiny Workshop Example - CO2 Uptake in Plants", style = "font-size: 20px")),
  # Establish Sidebar layout:
  sidebarLayout(
    # create sidebar (which we'll fill with input controls)
    sidebarPanel(width = 3,
      div(style = "color:green; padding:0px 0px 150px 0px;", #padding: top, right, bottom, left
        checkboxGroupInput(inputId = "type", label = "Plant Type", choices = levels(dat$Type),
                           selected = levels(dat$Type))
        ) ## everything held within this div() tag helper function will take on the styles
          ##   assigned to it (Unless overridden internally) -- see next code example
      ),
    # Spot for the plot
    mainPanel(width = 9,
      plotOutput(outputId = "scatter.plot")
    )
))

server <- function(input, output) {

  # renderPlot indicates that the function is "reactive"...
  # ...it will automatically re-execute when input changes
  output$scatter.plot <- renderPlot({

    # render the plot:
    ##Create a stable plot size using all data before adding [changing] subsets using points().
    plot(uptake ~ conc, data = dat, type = "n")  ##type = "n" causes no points to be drawn.
    points(uptake ~ conc, data = dat[dat$Type  %in% c(input$type),])
    title(main = "Plant Trends")
    })
}

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


Shiny Workshop Example - CO2 Uptake in Plants

Plant Type

☑ Quebec

☑ Mississippi

We've reduced the font size of the title, changed the color of the text in the sidebarPanel and extended the lower edge of the sidepanel by 150 pixels. Note that all of these changes occurred in the ui code only, and the server code remains unchanged.

You'll also notice we added `width` arguments to the two panels in our sidebarLayout. This caused the widths of our side panel and main panel to change (shrink and grow, respectively).

FluidPage windows are split width-wise into 12 units. These are essentially unit-less and instead represent relative proportions of the flexible screen width. (Remember, fluidPage allows the app to fit numerous screen sizes).

- The arbitrary 12-unit width applies to all sub-setted layers as well. So even though the whole screen can be assigned 1 to 12 units of width, each panel can likewise itself be split-up based on a 12-unit scale.
  - If we wanted to subset either panel further, we would use `fluidRow` and `column` functions with their own assignments of `widths` (as well as `offsets` for spaces).

Next, Let's Add More Input Controls!

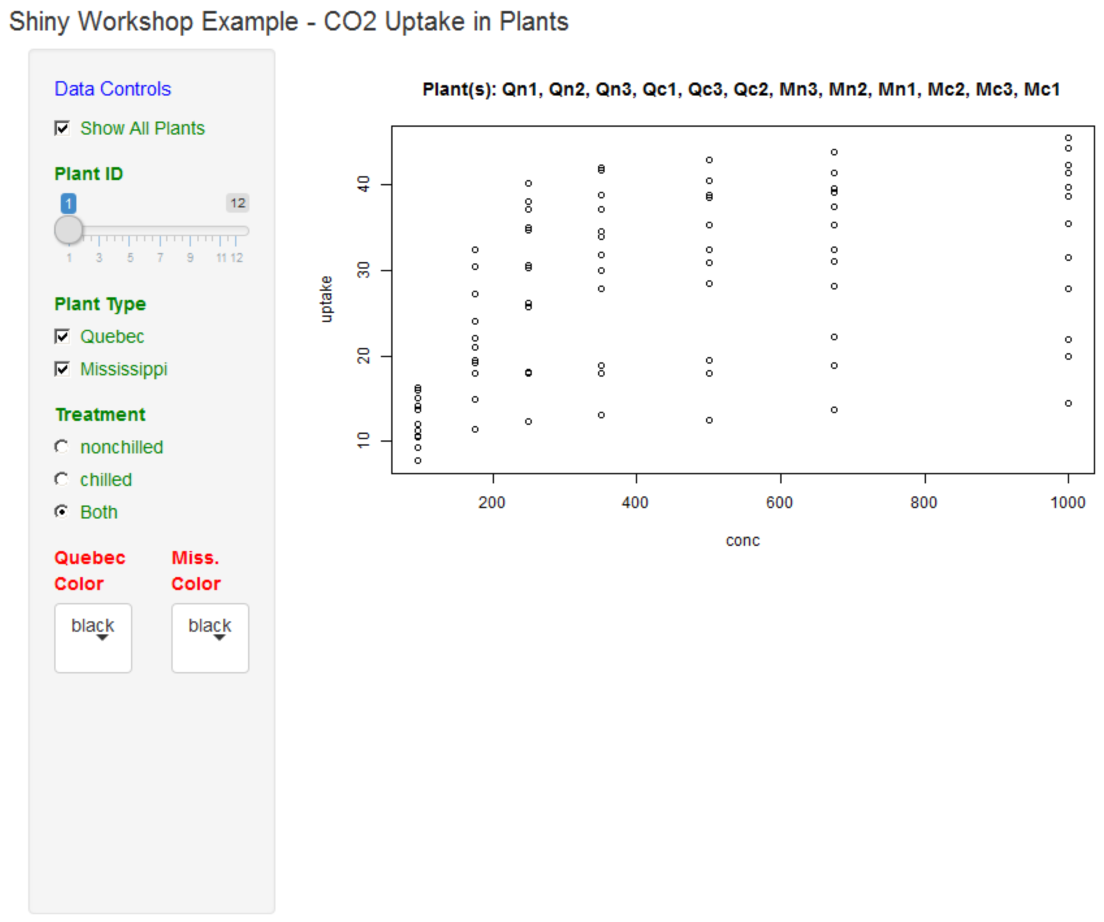
Next, we'll add a number of other input controls that can each modify our plot. These will help to demonstrate [only some of] the myriad of input widgets available in shiny.

We'll simultaneously demonstrate **how one can split up an app's panel using `fluidRow` and `column` arguments**.

- Specifically, `fluidRow` must be called anytime you want to isolate adjustment to a given row as well any time you want to add objects side by side. This latter case is when you apply the `column` function.

```r
ui <- fluidPage(

  # Title Panel: ####
  titlePanel(
    p("Shiny Workshop Example - CO2 Uptake in Plants", style = "font-size: 20px")
  ),


  sidebarPanel(width = 3,
               div(style = "color:green; padding:0px 0px 150px 0px;",
               ## padding is top, right, bottom, left
                   p("Data Controls", style = "font-size: 15px; color:blue;"),

                   # Check box to give us the option whether to show all plants...
                   # ...or just the plant selected in the slider input below
                     checkboxInput(inputId = "all.plants", label = "Show All Plants",
                                   value = T),

                   # Slider bar allows us to select which plant ID to plot.
                   # This slider only works if the above checkboxInput is unchecked...
                   # ...(see server.R code).
                     sliderInput(inputId = "plant", label = "Plant ID", min = min(dat$ID),
                                 max = max(dat$ID), value = 1),  ## sets starting value at 1

                   # This group check box let's us narrow which plant Types are plotted.
                   # checkboxGroupInput differs from checkboxInput by allowing more than one...
                   # ...option to be checked.
                     checkboxGroupInput(inputId = "type", label = "Plant Type",
                                        choices = levels(dat$Type),
                                        selected = levels(dat$Type)),
                                          ## sets starting checks to all levels
                   # Radio buttons work like checkboxes, but result is in strings/#'s...
                   # ...instead of boolean T/F.
                   # They allow multiple options like checkboxGroupInputs...
                   # ...but only one selection.
                     radioButtons(inputId = "treat", label = "Treatment",
                                  choices = c(levels(dat$Treatment), "Both"),
                                  selected = "Both"),

                   fluidRow(
                     div(style = "color:red;",
                         ## since both columns below are assigned a width of 6 (with 0 offset),
                         ## they'll fill the tabpanel width equally.
                         column(width = 6, offset = 0,
                                selectInput(inputId = "colorQ", label = "Quebec Color",
                                            choices = palette(), selected = "black")
                         ),
                         column(width = 6, offset = 0,
                                selectInput(inputId = "colorM", label = "Miss. Color",
                                            choices = palette(), selected = "black")
                         )
                     )

                   )
               ) ## Everything held within this div tag helper function takes on the styles...
                 ##  ...assigned in div() (Unless overridden internally)
                 ## see the div() function assigned in fluidRow above.
  ),

  # Spot for the plot
  mainPanel(width = 9,
            plotOutput(outputId = "scatter.plot")
  )
)


server <- function(input, output) {

  # We need to Set-up various input values:

    ## Allows us to narrow data by one plant or to include all plants.
    ## This is necessary b/c checkboxInput simply generates T/F, and we must translate that...
    ## ...to plant ID #'s.
    plants.to.plot <- reactive({  # Note: Reactive objects take on () when used. (see below)
      if(input$all.plants == T) {
        sort(unique(dat$ID))
      } else { input$plant
      }
    })

    ## Allows us to narrow data by one treatment level or to include both.
    ## This is necessary because "both" is not actually a Treatment type, ...
    ## ...and therefore needs to be assigned something meaningful (i.e., relevant to the data)
    treats.to.plot <- reactive({
      if(input$treat == "Both") {
        levels(dat$Treatment)
      } else {input$treat
      }
    })

    ## provides vector of color options (corresponding to plant$Type) to apply to...
    ## ...coloring points in plot
    colors <- reactive({
      Colors <- rep(NA,length(dat2()$Type))
      Colors[which(dat2()$Type == "Quebec")] <- input$colorQ
      Colors[which(dat2()$Type == "Mississippi")] <- input$colorM
      Colors
    })

  # Create modified data set based on our numerous input selections:
  ## This will keep our plotting code neater...

  dat2 <- reactive({  dat[dat$ID %in% c(plants.to.plot()) & # Note: reactive objects use ().
                          dat$Type  %in% c(input$type) &
                          dat$Treatment %in% c(treats.to.plot()), ]
  })


  # Create plot to be rendered in mainpanel. Notice we now use dat2 as the data source.

  output$scatter.plot <- renderPlot({
    plot(uptake ~ conc, data = dat, type = "n")
    points(uptake ~ conc, data = dat2(),col = colors())
    title(main = paste0("Plant(s): ",
                        paste(levels(dat$Plant)[plants.to.plot()],collapse =", ")))
     ## Combines "Plant(s)" w/ the plant names of whichever plants are included by the inputs.
  })
}

# Note that we can save a shinyApp as an object that, when called, runs the app automatically
app <- shinyApp(ui = ui, server = server)
app
```

See Output Image on Next Page



One More Round of Complexity:

We'll update once more by including three more levels of complexity to our app. This will include introduction to two additional `render*()` and `*output()` functions.

1. I'll demonstrate that the **main panel can** likewise **contain multiple objects (of different types)**.
    - We'll add a data table below the scatter plot using `renderDataTable()` and `dataTableOutput()`
2. Similarly, I'll demonstrate that **output can be placed in the sidebarPanel**.
    - We'll add reactive printed text using `renderPrint()` and `verbatimTextOutput()` (more on that in a minute).
3. I'll demonstrate one of **many ways to add multiple "pages" (in this case *tabs*)** to your Shiny app.
    - Specifically, we'll add navigable tab panels using the `tabsetPanel()` function so as to extend our app to more pages.

Before we run this new code chunk, I want to introduce a cool functionality of `plotOutput()`. If you examine the help page for `plotOutput()` you'll see the arguments `click`, `dblclick`, `hover` etc. As you might be able to guess, these arguments **allow reactivity when clicking or hovering your mouse over plots** in the UI.

- One application of this is to click on a point in a plot and have that point's data reported to you. This can be done in a couple of ways, but I'll demonstrate the function **`nearPoints`**. From R documentation:

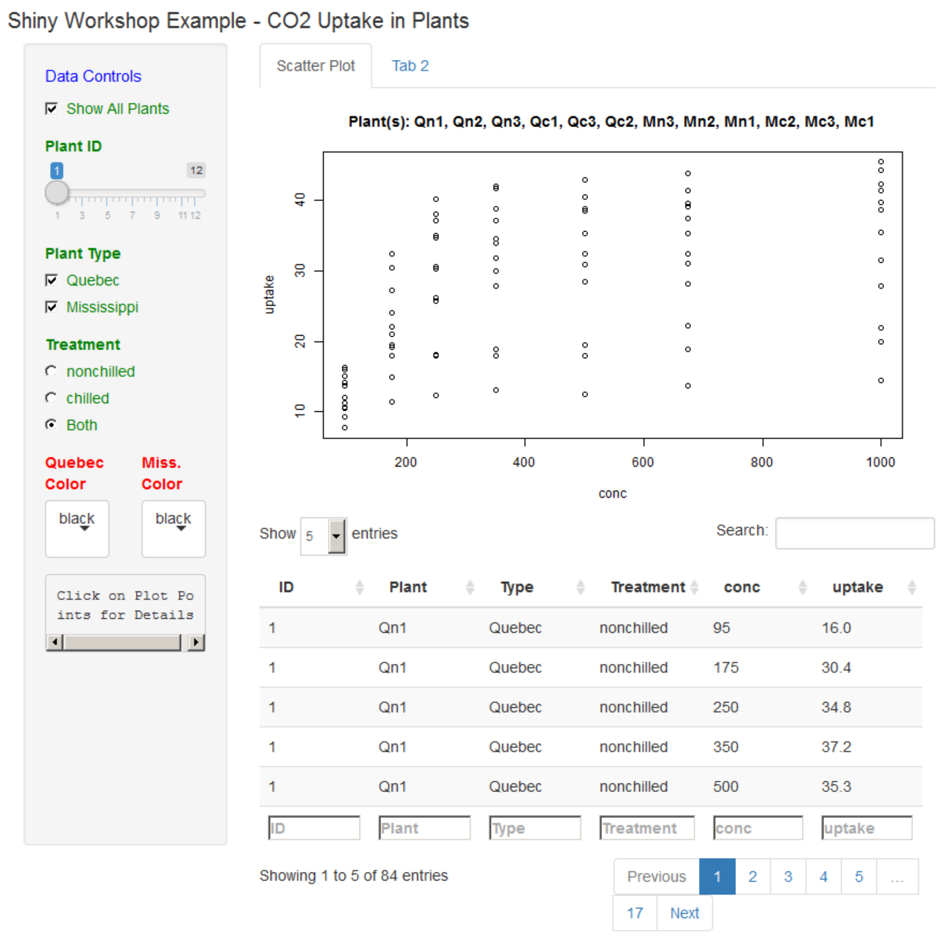
    > This function returns rows from a data frame which are near a click, hover, or double-click, when used with plotOutput. The rows will be sorted by their distance to the mouse event.

Ok, on to the next code chunk:

```r
ui <- fluidPage(

  # Title Panel: ####
  titlePanel(
    p("Shiny Workshop Example - CO2 Uptake in Plants", style = "font-size: 20px")
  ),

  sidebarPanel(width = 3,
               div(style = "color:green; padding:0px 0px 150px 0px;",
               ## padding is top, right, bottom, left
                   p("Data Controls", style = "font-size: 15px; color:blue;"),

                   # Check box to give us the option whether to show all plants...
                   # ...or just the plant selected in the slider input below
                     checkboxInput(inputId = "all.plants",label = "Show All Plants",value = T),

                   # Slider bar allows us to select which plant ID to plot.
                   # This slider only works if the above checkboxInput is unchecked...
                   # ...(see server.R code).
                     sliderInput(inputId = "plant", label = "Plant ID", min = min(dat$ID),
                                 max = max(dat$ID), value = 1),  ## sets starting value at 1

                   # This group check box let's us narrow which plant Types are plotted.
                   # checkboxGroupInput differs from checkboxInput by allowing more than one...
                   # ...option to be checked.
                     checkboxGroupInput(inputId = "type", label = "Plant Type",
                                        choices = levels(dat$Type),
                                        selected = levels(dat$Type)),
                                          ## sets starting checks to all levels

                   # Radio buttons work like checkboxes, but result is in strings/#'s...
                   # ...instead of boolean T/F.
                   # They allow multiple options like checkboxGroupInputs...
                   # ...but only one selection.
                     radioButtons(inputId = "treat", label = "Treatment", selected = "Both",
                                  choices = c(levels(dat$Treatment), "Both")),

                   fluidRow(
                     div(style = "color:red;",
                         ## since both columns below are assigned a width of 6 (with 0 offset),
                         ## they'll fill the tabpanel width equally.
                         column(width = 6, offset = 0,
                                selectInput(inputId = "colorQ", label = "Quebec Color",
                                            choices = palette(), selected = "black")
                         ),
                         column(width = 6, offset = 0,
                                selectInput(inputId = "colorM", label = "Miss. Color",
                                            choices = palette(), selected = "black")
                         )
                     )
                   ),

                   ## We'll insert printed R output generated from clicking the plot
                   ## Note: this text is created using nearPoints() in server.R
                   verbatimTextOutput(outputId = "plot.info")
```

```
              )
  ),

  mainPanel(width = 9,

            # Create tab layout for our app using tabsetPanel()
            ## Note b/c of the placement of this function, the side bar remains consistent...
            ## ...but we will have changing mainpanel content
            tabsetPanel(

              ## Everything held w/in this argument is placed in a tab w/ title "Scatter Plot"
              tabPanel(title = "Scatter Plot",

                       plotOutput(outputId = "scatter.plot",click = "plot.click"),
                       ## Note: we've assigned a new input object called "plot.click"

                       ##Let's add a data table to our main panel below the graph.
                       ## (Note: don't forget to add a comma after `plotOutput()`)
                       dataTableOutput(outputId = "data")

              ),

              tabPanel(title = "Tab 2",
                       p("We can add another graph here!!",style = "font-size:38px;" )

              )
            )
  )
)


server <- function(input, output) {

  # We need to Set-up various input values:

    ## Allows us to narrow data by one plant or to include all plants.
    ## This is necessary b/c checkboxInput simply generates T/F, and we must translate that...
    ## ...to plant ID #'s.
    plants.to.plot <- reactive({  # Note: Reactive objects take on () when used. (see below)
      if(input$all.plants == T) {
        sort(unique(dat$ID))
      } else { input$plant
      }
    })

    ## Allows us to narrow data by one treatment level or to include both.
    ## This is necessary because "both" is not actually a Treatment type, ...
    ## ...and therefore needs to be assigned something meaningful (i.e., relevant to the data)
    treats.to.plot <- reactive({
      if(input$treat == "Both") {
        levels(dat$Treatment)
      } else {input$treat
      }
    })

    ## provides vector of color options (corresponding to plant$Type) to apply to...
    ## ...coloring points in plot
    colors <- reactive({
      Colors <- rep(NA,length(dat2()$Type))
      Colors[which(dat2()$Type == "Quebec")] <- input$colorQ
      Colors[which(dat2()$Type == "Mississippi")] <- input$colorM
      Colors
    })

  # Create modified data set based on our numerous input selections:
  ## This will keep our plotting code neater...
  dat2 <- reactive({  dat[dat$ID %in% c(plants.to.plot()) & # Note: reactive objects use ().
                            dat$Type  %in% c(input$type) &
                            dat$Treatment %in% c(treats.to.plot()), ]
  })

  # Create plot to be rendered in mainpanel. Notice we now use dat2 as the data source.
  output$scatter.plot <- renderPlot({
    plot(uptake ~ conc, data = dat, type = "n")
    points(uptake ~ conc, data = dat2(),col = colors())
    title(main = paste0("Plant(s): ",
                        paste(levels(dat$Plant)[plants.to.plot()],collapse =", ")))
     ## Combines "Plant(s)" w/ the plant names of whichever plants are included by the inputs.
  })

  # Create data.frame showing data of only the plants (i.e. the rows) selected to be plotted
  output$data <- renderDataTable({ dat2()},
                                 options = list(lengthMenu = list(c(5, 10, -1),
                                                                  list("5", "10", "All")),
                                                pageLength = 5, ordering = T))
                                 ## Options allow for modification of the data table presented.
                                 ## This is a rare case in which UI formatting is assigned...
                                 ## ...in server.R code
                                   ## lengthMenu creates drop down menu giving you choice...
                                   ## ...in # of rows to show.
                                      ## (-1 = show all rows)
                                   ## pageLength assigns how many rows are shown when table...
                                   ## ...is 1st rendered
                                   ## ordering allows you to click on columns to change...
                                   ## ...ordering of the data.

    # Create data to be generated when the scatter plot is clicked

    ## Note: input$plot.click is generated by clicking on the scatter plot.
    ## This plot-click input object provides x & y coordinates of where you click

    ## We could use the nearPoints function to do this, but I don't like that it has a...
    ## ..."messy" NULL output

       # output$plot.info <- renderPrint({
       #   nearPoints(dat2()[,2:6], input$plot.click, xvar = "conc", yvar = "uptake",
       #   threshold=3)
       # })

    ## Instead, I'll make my own function that provides a custom string of text when...
    ## ...nothing is clicked.
```

```
       # Update to make plot clicker info look better:
       nearPoints2 <- function(coordinfo,blank.text = "Click on Plot Points for Details",
                               cols, ...) {
         if (is.null(coordinfo)) {
           cat(blank.text,"\n")   #"Click on Plot Points for Details"
           } else {
             points.info.table <- nearPoints(coordinfo = coordinfo,...)
             print(points.info.table[,cols], row.names = F)
             }
       }

       # Create table containing columns 2:6 of dat2 for the point clicked and assign to...
       # ...renderPrint output:
         output$plot.info <- renderPrint({
           nearPoints2(coordinfo = input$plot.click, cols = c(2:6), df = dat2(), xvar = "conc",
                       yvar = "uptake",threshold=3)
         })  ## Note: nearPoints (default or our custom nearPoints2) actually reports...
             ## ...all points (i.e., rows in your data.frame) that are within the threshold...
             ## So you might want to shrink `threshold` if your clicks result in too many data.
}

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



[✓ Show All Plants]

Plant Type: [✓ Quebec] [✓ Mississippi]

Treatment: nonchilled chilled [✓ Both]

Quebec Color: black

Miss. Color: black

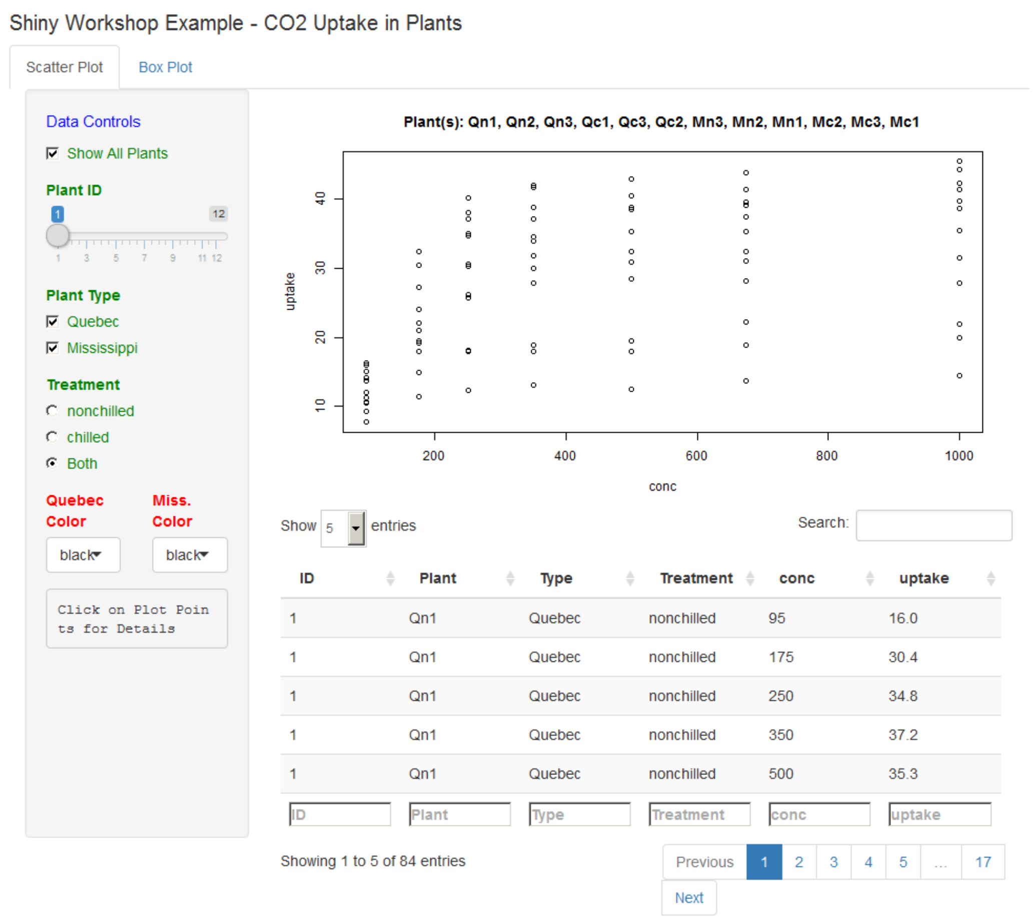
```
Click on Plot Po
ints for Details
```

ID	Plant	Type	Treatment	conc	uptake
1	Qn1	Quebec	nonchilled	95	16.0
1	Qn1	Quebec	nonchilled	175	30.4
1	Qn1	Quebec	nonchilled	250	34.8
1	Qn1	Quebec	nonchilled	350	37.2
1	Qn1	Quebec	nonchilled	500	35.3
ID	Plant	Type	Treatment	conc	uptake

Showing 1 to 5 of 84 entries

Now It's Your Turn!

Can you generate the app demonstrated below?



Shiny Workshop Example - CO2 Uptake in Plants

Data Controls

☑ Show All Plants

Plant ID: 1 (slider 1 to 12)

Plant Type

☑ Quebec

☑ Mississippi

Treatment

○ nonchilled

○ chilled

● Both

Quebec Color: black

Miss. Color: black

```
Click on Plot Poin
ts for Details
```

Show 5 entries   Search:

ID	Plant	Type	Treatment	conc	uptake
1	Qn1	Quebec	nonchilled	95	16.0
1	Qn1	Quebec	nonchilled	175	30.4
1	Qn1	Quebec	nonchilled	250	34.8
1	Qn1	Quebec	nonchilled	350	37.2
1	Qn1	Quebec	nonchilled	500	35.3
ID	Plant	Type	Treatment	conc	uptake

Showing 1 to 5 of 84 entries

Here's a Couple of Clues:

- You'll need to move/reorder the tabsetPanel functions.
- You need to add a new sub-title and 2 new input controls within the second tab
  - More information regarding input widgets can be found in the widget gallery or from the Shiny Cheat Sheet.
- You'll need to adjust the style of the panel sub-title.
  - More information regarding styling via HTML tags can be found in the tag glossary.
- You need to use the functions `boxplot` and `lines`.

`Look for the Follow-Up Handout for the Answer!`

Sharing Your Shiny App

Once you have completed an app, you can share it with your colleagues in several ways. The easiest way is to send them the code (server.R and ui.R file(s)). However, you can also host the code online so that it can be run from anyone using R without the files.

For example, Github can host the code. RStudio likewise provides options for hosting Shiny apps. They can host your app(s) in RStudio's dedicated Shiny Server, or they can help you host your app(s) on the cloud with ShinyApps.io.

- ShinyApps.io is free for the first 5 apps (and up to 25 user "active" hours per month).

Hosting Shiny apps is beyond the scope of this intro workshop, so I will leave learning more about that aspect of this process up to you. See here for more information.

Further Reading and Useful Resources

Learning Resources

1. Shiny Cheat Sheet: https://www.rstudio.com/wp-content/uploads/2016/01/shiny-cheatsheet.pdf
2. Widget Gallery: https://shiny.rstudio.com/gallery/widget-gallery.html
3. HTML Tag Glossary https://shiny.rstudio.com/articles/tag-glossary.html
4. Tutorials: https://shiny.rstudio.com/tutorial/
5. Example Apps: https://www.rstudio.com/products/shiny/shiny-user-showcase/

Shiny App Deployment & Sharing:

1. Deployment Guidance: https://shiny.rstudio.com/deploy/
2. Github Shiny Server: https://github.com/rstudio/shiny-server
3. RStudio's Shiny Server: https://www.rstudio.com/products/shiny/shiny-server/
4. ShinyApps.io: https://www.shinyapps.io/
