A Tutorial on Shiny and Leaflet (plus a little plotly!)

Zachary Hoylman

11/12/2024

Montana Climate Office

zachary.hoylman@umontana.edu

Set up and download required packages

Before we begin, let's install all the packages we will use throughout this lecture.

```
libraries = c("knitr", "shiny", "leaflet","ncdf4", "lubridate", "dplyr",
              "zoo", "ggplot2", "scales", "leaflet.extras",
              "mapview", "sf", "terra", 'plotly')

install.packages(libraries)

lapply(libraries, library, character.only = TRUE)
```

Introduction

In today's lecture, we will delve into the dynamic realms of data visualization and interactivity in R by exploring two powerful libraries: Shiny and Leaflet. Shiny empowers R users to construct interactive web applications seamlessly, while Leaflet, integrated through the 'leaflet' R package, enhances spatial data exploration by enabling the creation of customizable and interactive maps. Throughout this lecture, we'll navigate the intricacies of Shiny, understanding how it facilitates the development of user-friendly interfaces and responsive dashboards, making data analysis a more engaging and accessible experience. Additionally, we'll delve into Leaflet's capabilities, exploring how it harnesses the power of JavaScript to seamlessly integrate interactive maps into Shiny applications, providing a dynamic platform for spatial data representation and exploration.

Throughout an academic career, the traditional avenue for sharing research is typically through scientific publications, a crucial method for communicating findings to fellow researchers within a specific field. While this approach is essential for advancing knowledge within the scientific community (and something that is focused upon in graduate school), it often targets a relatively narrow audience of fellow scientists and academics. However, the broader impact of scientific research extends beyond the academic realm. To bridge the gap between scientific discovery and practical application, researchers must engage in additional strategies for dissemination. Communicating research findings to a wider audience, including policymakers, managers, and the general public, is essential for translating scientific knowledge into real-world applications. This is where where the world of apps and interactive mapping become so important!

Ok, with that in mind, let's get started... We will dive into Shiny first.

Shiny: Introduction and example

Shiny is an R package that enables the creation of interactive web applications directly from R scripts. Shiny apps consist of a user interface (UI) that defines the layout and appearance of the app and a server script that defines the app's behavior. The UI is built using a variety of pre-built and customizable widgets, allowing users to interact with and visualize data dynamically. The server script processes user inputs, runs scientific code and generates reactive outputs (such as plots), creating a seamless and responsive user experience. Shiny apps are widely used in data analysis, visualization, and reporting, providing a way to share R-powered analyses and insights with a broader audience through web interfaces. Importantly, shiny apps are relatively easy to develop and deploy locally, which can be extremely useful for data visualization, exploration and analysis dissemination (for example to co-authors or advisors).

While Shiny apps are powerful for creating interactive web applications in R, they do have some important limitations to be aware of. For example:

Performance: Shiny apps may face performance issues when dealing with large datasets or complex computations. Efficient coding practices and optimization techniques are crucial for maintaining responsive applications.

Scalability: Scaling Shiny apps to accommodate a large number of users can be challenging. As the user base grows, server resources need to be managed carefully to ensure consistent performance.

Despite these limitations, Shiny remains a valuable tool for interactive data visualization and analysis within the R ecosystem. It's important to weigh these considerations against the specific requirements and goals of a given project.

First lets run a shiny app example that comes with shiny to understand what apps are (in a very very very simple form).

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

Shiny layout

Like I introduced above, there are two main components to a shiny app, a User Interface (UI) and a server. The UI is used to pass arguments to the server script which conducts the operations. This concept is the underpinning of most apps, regardless of language (python, R, JavaScript, etc), so it is important to understand why this structure exists in Shiny apps. Typically we would describe this as front-end (UI) vs back-end (server) functions. These two parts of the app can either be contained in two separate scripts ("ui.R" and "server.R") or can be combined in a single script that has both components, typically "app.R". Throughout this lecture, we are going to keep them combined.

```
shinyApp(ui = defines the user interface,
         server = function(input,output) {
            define the operations being conducted and does things like plotting
            }
         )
```

Simple script example

The following script recreates the app shown above to familiarize ourselves with how shiny apps operate. We are going to be using a base R data set for this example, "faithful", which is a data set of Old Faithful geyser eruptions.

Lets take a quick look at the data:

```
head(faithful)
```

```
##   eruptions waiting
## 1     3.600      79
## 2     1.800      54
## 3     3.333      74
## 4     2.283      62
## 5     4.533      85
## 6     2.883      55
```

Here, we can see faithful is a data.frame with 2 columns. "eruptions" represent the length of time of a single eruption, and "waiting" is the time in between eruptions. We are going to be using the second column in this data set. Let's go ahead and build an app to produce a histogram of waiting times between eruptions. The user in this app will be able to choose a bin size for the histogram.

Now lets look at the app.

```
shinyApp(
  # First lets build the user interface (UI).
  ui = fluidPage(
  # App title ----
  titlePanel("Old Faithful Eruptions"),
  # There are different lay out options we can use in shiny,
  # here we will be using the "sidebarLayout" option.
  sidebarLayout(
    # Now we add sidebar panel for inputs ----
    sidebarPanel(
      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    # Main panel for displaying outputs ----
    mainPanel(
      # Output: Histogram named distPlot.
      # distPlot is defined below.
      plotOutput(outputId = "distPlot")
    )
  )),
  #Now for the server function
  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({
      # Fist we define what data is being used in the plot.
```

```
      # For a histogram we want our variable of interest on the x
      # The user doesnt have an option to change this.
      x = faithful$waiting
      # This is the part that is modifiied by the user (input$bins).
      # Here the user is overwriting the "bins" variable
      bins = seq(min(x), max(x), length.out = input$bins + 1)
      #here is the plot
      hist(x, breaks = bins, col = "#75AADB", border = "white",
           xlab = "Time in between eruptions (Minutes)",
           main = "Histogram of waiting times")
      })
})
```

The same script without comments.

Try changing this code to instead show a histogram of the length of en eruption rather than waiting times in between and allow the user to choose bins from a more constricted range, say 10 - 20 with a starting value of 15.

```
shinyApp(
  ui = fluidPage(
  titlePanel("Old Faithful Eruptions"),
  sidebarLayout(
    sidebarPanel(
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput(outputId = "distPlot")
    )
  )
),
server = function(input, output) {
  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 = "Time in between eruptions (Minutes)",
         main = "Histogram of waiting times")
    })
})
```

Now you can see that there is a cyclic nature to these apps. The UI passes an input to the server, the server does computation and passes an output to the UI. UI -> input -> server -> output -> UI

Multivariate example

Great! Now we have a better idea of how these apps work. Lets now do something a bit more realistic. Say you have a lot of data that you want to visualize in different ways. Lets make an app that allows the user

to choose x and y variables from a list of variables, plots them and does a bit of stats. This would be an example of data exploration that could help you to consider different analysis options. We are going to use the dataset "iris" for this example. This dataset has information about the properties of flowers. Let's look at the first few rows:

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The app

This app is going to have 3 user inputs and output a plot. We are going to let the user choose an X dataset to plot against a Y dataset and then calculate a linear model to display the relationship between them. Finally we will allow the user to choose a polynomial degree to modify the linear model's shape.

```
shinyApp(ui = fluidPage(
  # First we will build the UI in the same fashion as before
  titlePanel("Flowers!"),
  # we are going to allow the user to choose a polynomial degree for the model
  numericInput(inputId = "poly_degree",
                 label = "Choose Polynomial Degree",
                 value = 1, min = 1, max = 5),
  # this time instead of a slider, we are going to define the
  # drop down options for selecting data from the iris dataset.
  # We are going to define the input ID, Label and Choices.
  # Choices are a name you decide, and then the respective Column
  # name in iris. (See print out above for more details)

  sidebarLayout(
    sidebarPanel(
      selectInput(inputId = "x",
                  label = "Choose an independent variable",
                  choices = c("Sepal Length" = "Sepal.Length",
                              "Sepal Width" = "Sepal.Width",
                              "Petal Length" = "Petal.Length",
                              "Petal Width" = "Petal.Width"))
    ),
    # repeate for y
    sidebarPanel(
      selectInput(inputId = "y",
                  label = "Choose an dependent variable",
                  choices = c("Sepal Length" = "Sepal.Length",
                              "Sepal Width" = "Sepal.Width",
                              "Petal Length" = "Petal.Length",
                              "Petal Width" = "Petal.Width"))
    ),
  ),
```

```
  # the output will be a plot, same as the previous example
  mainPanel(
    plotOutput(outputId = "Plot")

  )
), server = function(input, output) {
  # now we define what part of the app is "reactive" in the server section.
  # in this case it will be the user defined variables to use for the modeling
  # and plotting.
  output$Plot = renderPlot({
    # using the user inputs, we will build the dataset used
    data = data.frame(x = iris[,input$x],
                      y = iris[,input$y])
    # compute a linear model with  polynomial term
    model = with(data,lm(y ~ poly(x,input$poly_degree)))
    # predict data out for plotting
    predict_data = data.frame(x = seq(min(data$x), max(data$x), length.out = 1000))
    predicted.intervals = data.frame(predict(model, newdata = predict_data,
                                             interval = "confidence"))
    # extract some inforamtion from the lm for the plot
    summary = summary(model)
    # build the plot itself
    plot(data$x, data$y, xlab = input$x, ylab = input$y, col = iris$Species)
    # add the lm as an and confidence intervals as lines
    lines(predict_data$x, predicted.intervals$fit,col='green',lwd=3)
    lines(predict_data$x, predicted.intervals$lwr,col='black',lwd=1)
    lines(predict_data$x, predicted.intervals$upr,col='black',lwd=1)
    # add the r2 of the regression to the plot
    mtext(paste0("r2 = ", summary(model)$r.squared), side=3)
    # add a color scalling legend
    legend("topleft",legend=levels(iris$Species),col=1:3, pch=1)
  })
})
```

Bingo! We have a working multivariate app that allows for some data analysis. At this point you should start to see some of the utility of building apps. Apps allow the user to get a much more "data rich" experience by allowing them to explore the data. In other words, they can evaluate your data and analysis more on their own terms. In summary allowing for flexibility in data visualization and analysis can promote a much greater understanding of your research.

Again, with very minimal comments:

Now try to remove the UI option to choose a polynomial degree and fix the analysis to use just a first order linear function (hint: you will have to remove a component of the UI and modify the model section of the server). I want you to start hacking this code so you get more comfortable modifying the scripts.

```
shinyApp(ui = fluidPage(
  titlePanel("Flowers!"),
  numericInput(inputId = "poly_degree",
                  label = "Choose Polynomial Degree",
                  value = 1, min = 1, max = 5),
  sidebarLayout(
```

```
    sidebarPanel(
      selectInput(inputId = "x",
                  label = "Choose an independent variable",
                  choices = c("Sepal Length" = "Sepal.Length",
                              "Sepal Width" = "Sepal.Width",
                              "Petal Length" = "Petal.Length",
                              "Petal Width" = "Petal.Width"))
    ),
    sidebarPanel(
      selectInput(inputId = "y",
                  label = "Choose an dependent variable",
                  choices = c("Sepal Length" = "Sepal.Length",
                              "Sepal Width" = "Sepal.Width",
                              "Petal Length" = "Petal.Length",
                              "Petal Width" = "Petal.Width"))
    ),
  ),
  mainPanel(
    #Output "Plot"
    plotOutput(outputId = "Plot")

  )
), server = function(input, output) {
  #render a plot called "Plot"
  output$Plot = renderPlot({
    #Data
    data = data.frame(x = iris[,input$x],
                      y = iris[,input$y])
    #Model
    model = with(data,lm(y ~ poly(x,input$poly_degree)))
    predict_data = data.frame(x = seq(min(data$x), max(data$x), length.out = 1000))
    predicted.intervals = data.frame(predict(model, newdata = predict_data,
                                             interval = "confidence"))
    #plot
    plot(data$x, data$y, xlab = input$x, ylab = input$y, col = iris$Species)
    lines(predict_data$x, predicted.intervals$fit,col='green',lwd=3)
    lines(predict_data$x, predicted.intervals$lwr,col='black',lwd=1)
    lines(predict_data$x, predicted.intervals$upr,col='black',lwd=1)
    mtext(paste0("r2 = ", summary(model)$r.squared), side=3)
    legend("topleft",legend=levels(iris$Species),col=1:3, pch=1)
  })
})
```

REMEMBER!! When you start to build these apps on your own and get stuck.... GOOGLE!!! Stack Overflow, GPT (etc) is your BEST FRIEND!! There are so many resources out there for R users, take advantage of the community and start piecing together others code and insight until you feel comfortable. For example, check this out https://shiny.rstudio.com/gallery/ All of the code used to create these apps is available to you.

Now that we have gained a cursory understanding of Shiny and its capacity to create dynamic applications, let's explore how Leaflet seamlessly integrates with Shiny to enhance spatial data visualization. This integration not only adds a spatial dimension to the dynamic capabilities of Shiny but also provides a powerful tool for researchers to convey geographic information and engage their audience in a more immersive and

interactive manner.

Leaflet: Introduction and installation

Leaflet is a very powerful library that allows for interactive mapping. A lot of research in the natural sciences has a spatial component and often we rely on graphical [G] user interfaces [UIs] (GUIs) to produce visualizations of our data. Think ESRI ArcGIS. In some cases, making quick maps for visualizing can be a pain and sometimes you want a product that is interactive for the end user without them having to have access to a GIS platform.

With this I present Leaflet (and subsequently mapview, which we will touch on at the end of the lecture), an open-source JavaScript library for desktop and mobile-friendly interactive maps. But... Now we have a R library that brings this powerful mapping service to R users without having to code in JavaScript (except for customization). This is awesome because, for scientists like us, JavaScript can be a bit of a pain. However, in some cases, you will need to migrate to JavaScript to make complex apps, but for the most part, you can get a lot of cool things done in the R environment that suits most peoples purpose.

Quick Note!

In this next section I am going to be using an operator you might not be familiar with, the "pipe" operator ("%>%" will be used here, though now R has native pipes, "|>"). This is a very nifty tool that allows a user to avoid having tons of parentheses, making code much more difficult to read. Pipes also allow you to avoid redefining derivatives of the same data.

When you put a bunch of pipes together, complex data manipulation can be easy to read and understand by someone not familiar with your code. This is called a "method chain" and is very common in other languages, like JavaScript. Modern R methods mostly follow method chaining. The main jist of the pipe operator is that the data set from the previous line is passed to the next part of the "method chain".

Here is an example:

```
library(dplyr)

# traditional way of doing some wierd data manipulation (HARD TO READ!!!)
method_1 = round(exp(sin(log(iris$Sepal.Length))),2)

# the other non preferable option.
data = iris$Sepal.Length
log_data = log(data)
sin_log_data = sin(log_data)
exp_sin_log_data = exp(sin_log_data)
method_2 = round(exp_sin_log_data,2)

# with a pipe operator (much easier to read, also allows you to avoid redefining data)
method_3 = iris$Sepal.Length %>%
  log()%>%
  sin()%>%
  exp()%>%
  round(., 2)

all.equal(method_1, method_2, method_3, tolerance=0.00000001)
```

They all yield the same result, as shown by the all.equal == TRUE, they just do so in different ways. When you start to have large data sets the pipe chain method saves tons of RAM (compared to defining each derivative product) and mixed with packages that leverage C++ (like dplyr) make your code run much, much faster.

Ok, back to leaflet.

Simple Leaflet example

Let's say you want a simple ESRI like base map.

```
library(leaflet)
leaflet() %>%
  addProviderTiles(providers$Esri.NatGeoWorldMap)
```

DONE! How cool is that? Now we can make the map go straight to Missoula.

```
leaflet() %>%
  addProviderTiles(providers$Esri.NatGeoWorldMap) %>%
  setView(lat = 46.875676, lng = -113.991386, zoom = 12)
```

Now lets add some data you collected from the field

```
# define some data with a location and a name
# this is where you would substatute your own data from the field
data_from_the_field = data.frame(names = c("Site 1", "Site 2", "Site 3"),
                                 lat = c(46.875, 46.877, 46.887),
                                 long = c(-113.991, -113.994, -114))

leaflet() %>%
  addProviderTiles(providers$Esri.NatGeoWorldMap) %>%
  setView(lat = 46.875676, lng = -113.991386, zoom = 13) %>%
  addMarkers(data_from_the_field$long, data_from_the_field$lat,
             popup = data_from_the_field$names)
```

Mixing Leaflet and Shiny

Ok, now let's explore mixing Shiny and Leaflet together! Here we are going to pull in current earthquake data from the last 30 days from the USGS (updated every minute) and plot it spatially using Leaflet. Here we are going to access remote data directly via HTTPS (we will do this more below!). We will then use Shiny to allow the user to select a minimum magnitude to crop the data and have shiny re-render the Leaflet map. Notice how little code this is... pretty sweet!

```
library(dplyr)

# read in data from the USGS server
earthquakes = read.csv("https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_month.csv") %>%
  select(latitude, longitude, mag, depth)

#build the app
```

```
shinyApp(ui = bootstrapPage(
  titlePanel("Earthquakes!"),
  # Input Slider
  sliderInput(inputId = "min_mag",
              label = "Minimum Earthquake Magnitude",
              min = 2.5, max = max(earthquakes$mag),
              value = 2.5, step = 0.1),
  # Render the leaflet map
  leafletOutput("mymap", height = "600")
  ),
   # Define the server operations
   server = function(input, output){
    output$mymap = renderLeaflet({
    # Filter the type of earthquake based on magnitude
      quakes = earthquakes %>%
        filter(mag > input$min_mag)
      # reactively define color ramp
      pal = colorNumeric(palette = c("green", "yellow", "red"), domain = quakes$mag)
      # generate map with leaflet
      leaflet(data=quakes) %>%
        #add a base map
        addProviderTiles("CartoDB.Positron") %>%
        #add the quake points! notice the ~ is used to reference the data = quakes dataset
        addCircleMarkers(lng=~longitude, lat=~latitude, weight = 1, radius = 7, color = "black", fillCo
        #add a legend
        addLegend(position="bottomleft", pal=pal, values = ~mag, title = "Magnitude", opacity = 0.3)
  })
})
```

Challenge! Can you add another slider bar to the app to crop data to a minimum depth as well as the current magnitude cropping? Hint depth data is already present in the dataset.

Shiny, Leaflet and custom functions

This is where things get really cool. Coupling custom functions with geospatial information from leaflet and using the reactive capabilities of Shiny yield some seriously powerful tools. For example, a question I get a lot working in the climate office is "home much precipitation does [insert place here] get". I am going to share with you a tool that can answer that question for any location in the continental U.S. using gridMET data produced by the University of Idaho/ UC Merced (now). Here I am going to also be using the ncdf4 library to interact with NetCDF files. NetCDF, which stands for Network Common Data Form, is a file format and set of software tools widely used in the scientific community for storing multidimensional scientific data, such as climate and atmospheric data, oceanography, and geophysics. NetCDF files are self-describing, platform-independent, and allow for the efficient storage and retrieval of large datasets.

In R, the ncdf4 library provides a way to work with NetCDF files. This library enables users to read from and write to NetCDF files, extract data from specific variables, and manipulate multidimensional arrays efficiently. The ncdf4 package is particularly valuable for researchers and scientists working with complex datasets, as it facilitates the handling of structured, multidimensional data commonly encountered in fields such as environmental science and climate research. The core concept here is of the "hypercube". In NetCDF, a hypercube refers to a multidimensional array structure that allows the storage of gridded scientific data

with multiple dimensions, such as time, latitude, longitude, and variable, facilitating efficient representation and analysis of complex datasets. These can be 4+ dimensions!

```
library(ncdf4)
library(lubridate)
library(dplyr)
library(zoo)
library(ggplot2)
library(scales)

#here we will define a function called "get_precip" which will access remote meterological data from th
#netcdf Hypercube
get_precip = function(lat_in, lon_in){
  #Define URL to netcdf from opendap servers (hypercube)
  urltotal = "http://thredds.northwestknowledge.net:8080/thredds/dodsC/agg_met_pr_1979_CurrentYear_CONUS
  # OPEN THE FILE
  nc = nc_open(urltotal)
  # find length of time variable for extraction
  endcount = nc$var[[1]]$varsize[3]
  # Querry the lat lon matrix
  lon_matrix = nc$var[[1]]$dim[[1]]$vals
  lat_matrix = nc$var[[1]]$dim[[2]]$vals
  # find lat long that correspond to a lat lon of interest! This is just minimizing the
  # difference between a user selected point, and the centrod of the pixels.
  lon=which(abs(lon_matrix-lon_in)==min(abs(lon_matrix-lon_in)))
  lat=which(abs(lat_matrix-lat_in)==min(abs(lat_matrix-lat_in)))
  # define variable name
  var="precipitation_amount"
  # read data and time and extract useful time information
  data = data.frame(data = ncvar_get(nc, var, start=c(lon,lat,1),count=c(1,1,endcount))) %>%
    mutate(time = as.Date(ncvar_get(nc, "day", start=c(1),count=c(endcount)), origin="1900-01-01")) %>%
    mutate(day = yday(time)) %>%
    mutate(year = year(time)) %>%
    mutate(month = month(time, label = T, abbr = F))
  # close file
  nc_close(nc)

  #now that we have the raw daily data, lets calculate monthly summaries of the data
  monthly_data = data %>%
    group_by(month, year) %>%
    dplyr::summarise(sum = sum(data)) %>%
    mutate(time = as.POSIXct(as.Date(as.yearmon(paste(year, month, sep = "-"),
                                                '%Y-%b')))) %>%
    arrange(time)
  # define ggplot function to display 3 years of data
  plot_function = function(data){
    precip_plot = ggplot(data = data, aes(x = time, y = sum))+
      geom_bar(stat = 'identity', fill = "blue")+
      xlab("")+
      ylab("Precipitation (mm/month)")+
      theme_bw(base_size = 16)+
      ggtitle("")+
      theme(legend.position="none",
            axis.text.x = element_text(angle = 60, vjust = 0.5))+
```

```
      scale_x_datetime(breaks = date_breaks("3 month"), labels=date_format("%b / %Y"),
                       limits= as.POSIXct(c(data$time[length(data$time)-36],
                                            data$time[length(data$time)])))
    return(precip_plot)
  }
  # return a list of 3 things, the plot (using the function above), daily daya and monthly data
  return(list(final_plot = plot_function(monthly_data),
              daily_data = data.frame(time = data$time, precipitation_mm = data$data),
              monthly_data = data.frame(month = monthly_data$month,
                                        year = monthly_data$year,
                                        precipitation_mm = monthly_data$sum)))
}
```

Now using this custom function, leaflet and shiny, lets make a map that allows a user to click on a location and receive a precipitation plot and download 40+ years of data (both daily and monthly).

```
library(shiny)
library(leaflet)
library(leaflet.extras)
library(ncdf4)

shinyApp(ui = fluidPage(
  # build our UI defining that we want a vertical layout
  verticalLayout(),
  # first we want to display the map
  leafletOutput("mymap"),
  # add in a conditional message for when calculations are running.
  conditionalPanel(condition="$('html').hasClass('shiny-busy')",
                                  tags$div("Calculating Climatology...",
                                           id="loadmessage")),
  # display our precip plot
  plotOutput("plot", width = "100%", height = "300px"),
  # set up download buttons for the user to download data
  downloadButton("downloadDaily", "Download Daily Data (1979 - Present)"),
  downloadButton("downloadMonthly", "Download Monthly Data (1979 - Present)")
  ),
  # now on to the server
  server = function(input, output) {
  # this is our map that we will display
  output$mymap = renderLeaflet({
    leaflet() %>%
    # this is the base map
    leaflet::addProviderTiles("CartoDB.Positron") %>%
    # terrain tiles
    leaflet::addTiles("https://maps.tilehosting.com/data/hillshades/{z}/{x}/{y}.png?key=KZO7rAv96Alr8UV
    # set default viewing location and zoom
    leaflet::setView(lng = -97.307564, lat = 40.368971, zoom = 4) %>%
    # modify some parameters (what tools are displayed with the map)
      leaflet.extras::addDrawToolbar(markerOptions = drawMarkerOptions(),
                    polylineOptions = FALSE, polygonOptions = FALSE,
                    circleOptions = FALSE, rectangleOptions = FALSE,
                    circleMarkerOptions = FALSE, editOptions = FALSE,
                    singleFeature = FALSE, targetGroup='draw')
```

```
  })
  # Now for our reactive portion which is when the user drops a pin on the map
  observeEvent(input$mymap_draw_new_feature,{
      # create a variable "feature" that will be overwritten when pin drops
      feature = input$mymap_draw_new_feature
      # call our precip function and store the outputs as a variable
      function_out = get_precip(feature$geometry$coordinates[[2]],
                                feature$geometry$coordinates[[1]])
      # render the plot from our function output
      output$plot = renderPlot({
        function_out[[1]]
      })
      # render the daily data output from our function to a csv for download
      # with a reactive name (lat long)
      output$downloadDaily = downloadHandler(
        filename = function() {
          paste("daily_precip_",round(feature$geometry$coordinates[[2]],4),"_",
                round(feature$geometry$coordinates[[1]],4),".csv", sep = "")
        },
        content = function(file) {
          write.csv(function_out$daily_data, file, row.names = FALSE)
        }
      )
      # render the monthly data output again with a reactive name
      output$downloadMonthly = downloadHandler(
        filename = function() {
          paste("monthly_sum_precip_",round(feature$geometry$coordinates[[2]],4),"_",
                round(feature$geometry$coordinates[[1]],4),".csv", sep = "")
        },
        content = function(file) {
          write.csv(function_out$monthly_data, file, row.names = FALSE)
        }
      )
    })
})
```

Bingo! Hopefully now you can see how powerful these packages are, especially when combined. Another nice thing about both of these packages is that they are "web ready", meaning you can host shiny apps on the web with your own web server when combined with "shiny-server" or use an online system like https://www.shinyapps.io. Further, leaflet maps (when not combined with shiny) can be saved as HTML documents that are ready to host on a traditional web server like Apache or NGINX. You can also send HTML documents containing your interactive map within an email. If you have a shiny component to your leaflet map you will still need a shiny compatible web server.

Mapview (super-duper brief)

There is one last package I want to introduce, called "mapview". Mapview uses Leaflet to allow for ultra fast and easy spatial data visualization. It is especially useful for quick checks, making sure your data makes sense and that your analysis is, in fact, doing what you think it's doing, without going to an ESRI-esk system.

```
#read some remote data from remote sorce.
#Here we are going to be using the sf library to read in vector data
montana = st_read('https://eric.clst.org/assets/wiki/uploads/Stuff/gz_2010_us_040_00_20m.json') %>%
  filter(NAME == 'Montana')
mapview(montana)
```

Wow, that was pretty easy. What about "raster" data? I know we haven't covered raster data, but they are simply grids of data. Once you see it, you will know exactly what I mean by grids. Here we will use the "terra" library. In this final example, lets add some percent of normal precipitation data over the last 60 days to this map with the Montana outline! These data are computed daily by the Montana Climate Office.

```
#read some remote data from remote sorce.
#Here we are going to be using the terra library to read in raster data
percent_norm_precip = rast('https://data.climate.umt.edu/drought-indicators/precipitation/current_percer
#read some remote data from remote sorce.
#Here we are going to be using the sf library to read in vector data
montana = st_read('https://eric.clst.org/assets/wiki/uploads/Stuff/gz_2010_us_040_00_20m.json') %>%
  filter(NAME == 'Montana')

mapview(percent_norm_precip, na.color = 'transparent') +
  mapview(montana, alpha.regions = 0)
```

Pretty darn easy!

Ok, one more quick interactive adventure. Interactive plots! Introducing plotly! The coolest part about plotly is that is maps directly to ggplot grobs. Lets make a quick plot of data from the iris dataset, this time, in ggplot, then we will convert it to plotly.

```
#make a simple base plot in ggplot
plot = ggplot(data = iris, aes(x = Sepal.Width, y = Petal.Width, color = Species)) +
  geom_point()

# Now lets convert it to an interactive plot
ggplotly(plot)
```

ggplotly is a powerful function that instantly transforms static ggplot2 visualizations into interactive plots, making it easy to explore data with zooming, hovering, and tooltips. This interactive layer enhances user engagement and is particularly useful for sharing data insights in web-based dashboards or presentations. Even better, ggplotly integrates seamlessly into Shiny apps! You can simply add ggplotly() to your ggplot2 code within the Shiny renderPlotly() function, and it will display as an interactive plot in your app, allowing users to dynamically explore the data.

Ok, that wraps up this lecture. If you have any questions about creating your own apps or maps, please feel free to contact me. My email address is at the top of this document.