Developing Data Products

May, 2023

Table of Contents

# About this Course

A data product is the production output from a statistical analysis. Data products automate complex analysis tasks or use technology to expand the utility of a data informed model, algorithm or inference. This course covers the basics of creating data products using Shiny, R packages, and interactive graphics. The course will focus on the statistical fundamentals of creating a data product that can be used to tell a story about data to a mass audience.

# 1 First week

In this overview module, we’ll go over some information and resources to help you get started and succeed in the course.

## 1.1 Introduction

### 1.1.1 Welcome to Developing Data Products

Hello everyone, my name is Bryan Caffo and I am excited to welcome you to the Developing Data Products course. This course is designed to provide you with the necessary tools for improving the data analysis process and making data-driven decisions. Shaun Cross and I, both from the Department of Biostatistics at the Johns Hopkins Bloomberg School of Public Health, have co-developed this course. We will present the common practice techniques for building in-demand data tools in R. Throughout this course, we will cover a range of topics including R markdown and Quarto, which allow you to create websites, PDFs, presentations, and even e-books from a single file in R. We will also teach you the latest features of the Shiny package, which you can use to create interactive web applications in R. In addition, we will cover interactive graphics using Plotly and Leaflet, which enables you to create beautiful maps that you can share online. We’ve also added a section on how to use swirl and swirlify to design courses in R so that practice your understanding. We believe that the key word in Data Science is “science”. Our specialization is focused on providing you with three things: (1) an introduction to the key ideas behind working with data in a scientific way that will produce new and reproducible insight, (2) an introduction to the tools that will allow you to execute on a data analytic strategy, from raw data in a database to a completed report with interactive graphics, and (3) on giving you plenty of hands on practice so you can learn the techniques for yourself. This course represents the final cog in a data science application, creating an end-usable data product.

We are excited about the opportunity to attempt to scale Data Science education. We intend for the courses to be self-contained, fast-paced, and interactive. Thank you for joining us, and we look forward to seeing you in class!

### 1.1.2 Syllabus

In this section you will get to learn a bit more about your instructors, the course, the policies and the course objectives.

#### 1.1.2.1 Course Instructor(s)

The primary instructor of this class is [Brian Caffo](http://www.bcaffo.com/) Brian is a professor at Johns Hopkins Biostatistics and co-directs the [SMART working group](http://www.smart-stats.org/) This class is co-taught by Roger Peng and Jeff Leek.

#### 1.1.2.2 Course Description

A data product is the production output from a statistical analysis. Data products automate complex analysis tasks or use technology to expand the utility of a data informed model, algorithm or inference. This course covers the basics of creating data products using Shiny, R packages, and interactive graphics. The course will focus on the fundamentals of creating a data product that can be used to tell a story about data to a mass audience.

In this class students will learn a variety of core tools for creating data products in R and R Studio in specific. Students will be evaluated via quizzes and a culminating project.

Course Content

The lectures will be taught over four weeks with the third week dedicated to creating R packages.

The weeks are organized as follows

1. Shiny
2. Quarto
3. Creating and deploying projects
4. Creating R packages, classes and methods

#### 1.1.2.3 Github repository

This course is hosted on GitHub at <https://github.com/B7M/Developing-Data-Products>

Please issue pull requests so that we may improve the materials.

#### 1.1.2.4 YouTube

To keep the course materials as up to date as possible, we’ve decided to eliminate the video contents; however, if you think watching videos could help you have a grasp of the information faster we suggest you watch the videos on YouTube, most of them can be found here:

<https://www.youtube.com/playlist?list=PLpl-gQkQivXhr9PyOWSA3aOHf4ZNTrs90>

#### 1.1.2.5 Book: Developing Data Products in R

This book introduces the topic of Developing Data Products in R. A data product is the ideal output of a Data Science experiment. This book is based on the Coursera Class “Developing Data Products” as part of the Data Science Specialization. Particular emphasis is paid to developing Shiny apps and interactive graphics.

The book is available here: <https://leanpub.com/ddp>

It’s variable pricing, including free! It also includes some content that was not covered in the class and omits some other. It’s a little rough, but as I work on it you’ll get all of the updates.

#### 1.1.2.6 Weekly quizzes

* There are three weekly quizzes.
* You must earn a grade of at least 80% to pass a quiz
* You may attempt each quiz up to 3 times in 8 hours.
* The score from your most successful attempt will count toward your final grade.

#### 1.1.2.7 Course Project

The Course Project is an opportunity to demonstrate the skills you have learned during the course. It is graded through peer assessment. You must earn a grade of at least 80% to pass the course project.

#### 1.1.2.8 Grading policy

You must score at least 80% on all assignments (Quizzes & Project) to pass the course.

Your final grade will be calculated as follows:

* Quiz 1 = 20%
* Quiz 2 = 20%
* Quiz 3 = 20%
* Course project = 40%

#### 1.1.2.9 Differences of opinion

Keep in mind that currently data analysis is as much art as it is science - so we may have a difference of opinion - and that is ok! Please refrain from angry, sarcastic, or abusive comments on the message boards. Our goal is to create a supportive community that helps the learning of all students, from the most advanced to those who are just seeing this material for the first time.

#### 1.1.2.10 Plagiarism

Johns Hopkins University defines plagiarism as “…taking for one’s own use the words, ideas, concepts or data of another without proper attribution. Plagiarism includes both direct use or paraphrasing of the words, thoughts, or concepts of another without proper attribution.” We take plagiarism very seriously, as does Johns Hopkins University.

We recognize that many students may not have a clear understanding of what plagiarism is or why it is wrong. Please see the JHU referencing guide for more information on plagiarism.

It is critically important that you give people/sources credit when you use their words or ideas. If you do not give proper credit – particularly when quoting directly from a source – you violate the trust of your fellow students.

### 1.1.3 Some Basics

A couple of first week housekeeping items. First, make sure that you’ve completed R Programming and the Data Scientist’s Toolbox course. Reproducible Research would be helpful, but is not mandatory. At a minimum you must know: very basic git, basic R and very basic knitr.

You can clone the whole repo with (http)

git clone <https://github.com/B7M/Developing-Data-Products.git> or (ssh) git clone [git@github.com](mailto:git@github.com):B7M/Developing-Data-Products.git

Going through the R code is the best way to familiarize yourself with the lecture materials.

The lecture material for this class is largely front-loaded. This is because the latter time of the class is devoted to developing your data application. Thus the class should be doable in about a 1 month’s time or maybe less. Though make sure you’re keeping up with the classes at the beginning so that you have some space in your schedule later on for app development!

If you’d like to keep up with the instructors I’m ([**bcaffo?**](#ref-bcaffo)) on twitter, Roger is ([**rdpeng?**](#ref-rdpeng)) and Jeff is ([**jtleek?**](#ref-jtleek)). The Department of Biostat here is ([**jhubiostat?**](#ref-jhubiostat)).

<https://datasciencespecialization.github.io/Developing_Data_Products/welcome.html>

#### 1.1.3.1 Community Site - [Read on GitHub](https://datasciencespecialization.github.io/Developing_Data_Products/community.html)

Since the beginning of the Data Science Specialization, we’ve noticed the unbelievable passion students have about our courses and the generosity they show toward each other on the course forums. A couple students have created quality content around the subjects we discuss, and many of these materials are so good we feel that they should be shared with all of our students.

We’re excited to announce that we’ve created a site using GitHub Pages: <http://datasciencespecialization.github.io/> to serve as a directory for content that the community has created. If you’ve created materials relating to any of the courses in the Data Science Specialization, please send us a pull request so we can add a link to your content on our site. You can find out more about contributing here: <https://github.com/DataScienceSpecialization/DataScienceSpecialization.github.io#contributing>

We can’t wait to see what you’ve created and where the community can take this site!

#### 1.1.3.2 R and RStudio Links & Tutorials

[Practice with Lab Sandbox](https://www.coursera.org/learn/data-products/lab-sandbox?path=%2F)

If you haven’t yet installed R and RStudio, you’ll need to do so now. Here are some links and video tutorials.

Links

[Rstudio](https://posit.co) [R project](https://www.r-project.org)

Tutorials

[Installing R for Windows](https://www.youtube.com/watch?v=LII6of-5Odw) [Installing R for Mac](https://www.youtube.com/watch?v=xokJUwn0mis) [Installing RStudio Mac](https://www.youtube.com/watch?v=JbTMvQ-SbvQ)

## 1.2 Shiny Part 1

### 1.2.1 Shiny

**Remember to read the book chapters on Shiny.**

Shiny is an important enough topic to devote a lot of time to it. Shiny is a product by RStudio and it is described by RStudio as “A web application framework for R”. They further add “Turn your analyses into interactive web applications with no HTML, CSS, or JavaScript knowledge required”. This is mostly true, though a little HTML at least would be useful for understanding some of the concepts. We’ll proceed as if your html knowledge is very basic and no more advanced than understanding heading levels for fonts. It is important to distinguish between a Shiny applications (app) and a Shiny server. A Shiny server is required to host a shiny app for the world. Otherwise, only those who have have shiny installed and have access to your code could run your web page, which is really defeating the purpose of making a web page in the first place.

In this class, we won’t cover creating a shiny server, as that requires understanding a little linux server administration. Instead, we’ll run our apps locally and use RStudio’s service for hosting shiny apps on a platform called [shinyapps.io](https://www.shinyapps.io). In other words, RStudio does the server work for your so that all you need to worry about is building your app. [Shinyapps.io](https://www.shinyapps.io) is free up to a point in that you can only run 5 apps for a certain amount of time per month. This will be fine for our purposes, but if you’re really going to get into making Shiny apps, you’ll have to spring for a paid plan or run your own server.

### 1.2.2 Shinyapps.io Project

**Shinyapps.io Project**

Some people in this session let us know that they are concerned about running up against the 25-hour per month limit on the free tier of shinyapps.io.

Should you hit the limit on the free plan, RStudio will send you a message. If you receive the message and are more than a few days from getting a fresh 25 hours on your monthly renewal, please send an email to [shinyapps-support](shinyapps-support@rstudio.com) with the email address you use on the service and the account name you are using (the first part of the URL). RStudio will then increase your limit so you can continue working on your project.

Since there are a lot of folks in the class we’d appreciate if you only emailed RStudio after you get the message and only if you feel you’ll need more time.

### 1.2.3 Shiny 1.1

Here we will introduce Shiny, which is the cornerstone of this course and we will be using it to develop data products. Shiny is a web development framework that is based on R, so you only need to know R to use it. With Shiny, you can use R for both the back end and front end of the development process. This means you don’t need to be familiar with standard tools like JavaScript, CSS and HTML. However, we will briefly talk about these later in the course. So, let’s talk about Shiny and its use in data science. Shiny has been a game-changer for data scientists who specialize in R and focus on developing algorithms and analysis. However, if you come from a web development background, you probably have a different set of tools at your disposal, and Shiny may not be your first choice. Shiny is a great tool for those who want to prototype potential data products, especially if you are a smaller organization. For larger organizations, Shiny can be used to create a prototype, which can then be converted by other skilled professionals into a more polished and visually appealing product using compiled languages, JavaScript, and HTML.

With Shiny we create the GUI in R, the algorithms and everything else that are powering it. That means your time to create a data product is completely minimized. As we mentioned Shiny is a product created by R Studio and it’s free; however, hosting Shiny on a server, could potentially cost you money because you need to actually have a web server that is not only going to serve up the web pages but is also going to call R in the back end to run your algorithms or models. Surprisingly, RStudio has a free hosting service, and that’s what we’re going to use for the class. If you choose to host the application on your own server, you could run off of Amazon AWS or something like that. If you only need portability at a local level, you can create and test Shiny apps locally and share them with others who use RStudio. However, the real benefit of Shiny is the ability to create a web server that anyone can access to display your data analysis or prediction algorithm. In this class, we will use shinyapps.io as our free server, and we will cover how to use it later on. Now, we will start from scratch and build a working web server with R as the back end.

Before we dive into using Shiny, there are a few prerequisites to keep in mind.

* Although everything is done in R, some basic knowledge of HTML is recommended. While it’s not necessary to know CSS or JavaScript, familiarity with some HTML commands can help you better understand what the R commands are doing. However, we won’t be teaching HTML here as there are plenty of online tutorials available to learn from.
* Shiny incorporates the Bootstrap package, which is not related to the statistics bootstrap, but is instead a web design framework developed by Twitter. Bootstrap provides a set of buttons and other design elements that look good and resize well on various devices, from small phones to large screens. The creators of Bootstrap have put a lot of thought into making it user-friendly. If you are not familiar with web development, it’s recommended to stick with the Bootstrap themes in Shiny.
* To learn about the three core components of the applications we’ll be developing, you can refer to some tutorials on HTML, CSS, and JavaScript. Briefly, CSS provides styling, JavaScript is responsible for interactivity, and HTML is the backbone of the front end. It’s crucial to familiarize yourself with HTML, so be sure to check out some HTML tutorials.

To get started with Shiny, you’ll need to install it using the command {r, echo=T, eval=F} install.packages('shiny') and then ensure it’s loaded by typing {r, echo=T, eval=F} library(shiny). Before beginning, it’s worth considering the official Shiny tutorial at RStudio, which this lecture will largely be following.

Alright, let’s quickly go over the basics of a Shiny project before we jump into some code demonstrations. To create a Shiny project, you need two files: ui.R and server.R. The ui.R file is responsible for creating the user interface and determining how your app will look, while the server.R file controls the app’s behavior and functionality. It’s time to transition into some code demonstrations, so please stay tuned.

### 1.2.4 Shiny 1.2

To create our first app, you’ll need a relatively recent version of RStudio. The easiest way to start is by clicking Alt+F (on Windows) or File (on Mac), then selecting New File, and then Shiny Web App. Name your app whatever you want, such as “myApp” Next, you can choose whether to have the files in a **single** file or in **two separate files**, a ui.R and a server.R file. We recommend doing it with two files. It’s preferred to work with many small files rather than one big file containing multiple functions. However, it’s important to note that the specific named functions within the ui.R and server.R files are what’s required, not just the files themselves. Now go ahead and create a directory and the two files, namely ui.R and server.R, by copying the following snippet codes into the pre-populated Shiny files.

#ui.R  
library(shiny)  
shinyUI(fluidPage(  
 titlePanel("Data science FTW!"),  
 sidebarLayout(  
 sidebarPanel(  
 h3("Sidebar Text")  
 ),  
 mainPanel(  
 h3("Main Panel Text")  
 )  
 )  
))

# server.R  
  
library(shiny)  
shinyServer(function(input, output) {  
  
})

The ShinyUI function controls the user interface, and the shinyServer function is required in the server.R file. Although our server doesn’t do anything in this point. Let’s take a closer look at the ShinyUI function. The **fluidPage** function specifies the type of page used to create the user interface, which is typically the default option. The **titlePanel** function sets the title of the GUI element to *Data science FTW!* The fluidPage function contains a main panel and a side panel by default, so we use the **sidebarLayout** function to specify the sidebar’s contents. The sidebarPanel function is used to specify the sidebar’s contents, and the **h3** function sets the third-level heading for the *slidebar Text* and *Main panel text*. You can run the Shiny app in two ways. Firstly, if you have a recent version of RStudio, you can click **Run App** to run the app. Secondly, you can switch to the directory where the ui.R and server.R files are located and type **runApp()** in the console. You can also specify the app’s path within the **runApp** function. RStudio creates a local web server, which can be accessed using RStudio’s browser. While it’s possible to simply share the code with others so they can run it, that’s not our ultimate goal. Our goal is to have the app running on a webpage. We’ve taken the first step, so now let’s continue and work on adding functionality to make the app more useful.

### 1.2.5 Shiny 1.3

When we’re constructing our Shiny app using the default workflow, it’s important to remember that we’ll be utilizing R to insert our HTML elements. Fortunately, Shiny incorporates all of the typical HTML tags, such as the various heading tags (**h1** through **h6**), **hyperlinks**, **divs**, **spans**, and other commonly used ones. To view a comprehensive list of all the HTML tags available, simply type **?builder** in the R console. In the following sections, we’ll demonstrate how you can use your own HTML file to build your Shiny app, giving you the freedom to create a more intricate page. However, for now, we’ll be using the pre-built functions. The following code example will display the various heading styles and other elements.

library(shiny)  
shinyUI(fluidPage(  
 titlePanel("HTML Tags"),  
 sidebarLayout(  
 sidebarPanel(  
 h1("H1 Text"),  
 h3("H3 Text"),  
 em("Emphasized Text")  
 ),  
 mainPanel(  
 h3("Main Panel Text"),  
 code("Some Code!")  
 )  
 )  
))

Let’s go over the various HTML tags that we can utilize in our Shiny app, or at least a subset of them. Specifically, we will be covering the formatting tags. In the UI.R function, we have added some headings including h1, h2, h3, which are all located in the sidebar layout. Additionally, we have included some emphasized text and code text, which is displayed in a mono space format. It’s always best to display code in this format, just like when editing code. Once you run the app, you can see the different heading values, main panel, and code displayed in mono space format. While the app is running if you click on **Open in Browser**, it will open the app in a web browser. Now in the browser, you can use *Ctrl+U* on Windows or *Option+Command+U* on Mac to view the R code that generates the webpage. This will allow you to see how the app uses JavaScript libraries to run Shiny. You may not need to worry about the head section unless you have some familiarity with JavaScript.

Most of the elements you see here are style sheets from Twitter Bootstrap, which provides the formatting for all the buttons and styles that Shiny uses. However, this section also contains some of the input and output components that Shiny requires. The majority of the content we display is in the body, where you can observe how the function creates the HTML commands by translating your ui.R file. This is a valuable exercise, particularly when creating named buttons and other similar components. It can also help you become more comfortable with coding in HTML if you’re familiar with Shiny but not with HTML. Eventually, if you intend to develop advanced web pages that call R in the back end, you’ll likely need to create your own pages rather than relying solely on ui.R. Therefore, playing around with HTML tags and designing a simple non-interactive webpage can help you become more familiar with them. In our next session, we’ll start adding buttons, sliders, and graphs, calling R in the back end, and exploring some of Shiny’s essential features.

### 1.2.6 Shiny 1.4

Our Shiny app needs to take input values, perform R calculations, and display the resulting output. To start, we are going to create a slider in the user interface for input values. The slider’s selected values will be passed to the server function and then returned to the user interface function for display. This is a preliminary exercise where we’re not using the input values for any calculations. Let’s proceed to create a slider in the sidebar panel of the shinyUI function. The slider’s name is “slider2” and it goes from 0 to 100, with a starting value of 0, and a prompt for the user like “Slide Me!”. The slider’s selected value is given the name “slider2” and displayed to the user. The shiny server function takes input and output as arguments within curly braces to define the function.

# ui.R  
library(shiny)  
shinyUI(fluidPage(  
 titlePanel("Slider App"),  
 sidebarLayout(  
 sidebarPanel(  
 h1("Move the Slider!"),  
 sliderInput("slider1", "Slide Me!", 0, 100, 0)  
 ),  
 mainPanel(  
 h3("Slider Value:"),  
 textOutput("text")  
 )  
 )  
))

and

library(shiny)  
shinyServer(function(input, output) {  
 output$text <- renderText(input$slider1)  
})

In Shiny, a function needs to have curly braces and it needs to be closed with both curly braces and the shiny server parenthesis. The function takes input from the UI, which is the shiny UI function. The function takes in $slider and renders it, displaying the text. We can name the output text. In the UI.R, in the main Panel, we specify that the slider value is displayed and the text output is text. This ensures that the same text we labeled from the output in the server function will be displayed in the name panels. When we run the code, we can move the slider and see the slider value displayed.

It’s important to note that there’s nothing special about having the slider in the side bar panel and the text in the main panel. The order isn’t significant either. The code for the slider and the output could be in different panels and still work fine. This is an important concept to keep in mind when working with Shiny and reactive expressions. You don’t want to think about it running linearly like a regular R program because the server is running reactively and constantly going back and forth. To be comfortable with Shiny your mindset needs to change a bit from regular R programming. It’s still a program, but it’s more interactive. One thing to note is that if we were to label the text output $text1 instead of $text, it wouldn’t display anything. This is because the UI is looking for text when it says textOutput. Similarly, if we were to label the slider $slider1 instead of $slider, it wouldn’t display anything because the render text from the server function is looking for $slider1, but the UI hasn’t put out anything labeled slider1. It’s important to remember that your labels need to match up for everything to work properly.

### 1.2.7 Shiny 1.5

We’ve reached a point where we can start utilizing Shiny. One of the key ways in which Shiny proves helpful is in creating interactive graphics. R, in its default form, doesn’t possess a lot of capacity for interactive graphics. With Shiny; however, the scope is broadened, and it provides an efficient way to distribute your results as they’re embedded in a web page that you can share broadly. Now, we’re going to develop an interactive plot that requires a bit more code in the server.R functions. Once you go through this example, you should be able to use Shiny in a more practical way that can assist you with your everyday data science needs. Now, let’s proceed to the main task at hand. We start with the initial code and run it to check out the app before we delve into its internal workings.

# ui.R  
library(shiny)  
shinyUI(fluidPage(  
 titlePanel("Plot Random Numbers"),  
 sidebarLayout(  
 sidebarPanel(numericInput("numeric", "How many random numbers should be plotted?",  
 value = 500, min = 1, max = 1000, step = 1),  
 sliderInput("sliderX", "Pick minimum and maximum X values",  
 -100, 100, value = c(-50, 50)),  
 sliderInput("sliderY", "Pick minimum and maximum Y values",  
 -100, 100, value = c(-50, 50)),  
 checkboxInput("show\_xlab", "Show/Hide X axis label", value = TRUE),  
 checkboxInput("show\_ylab", "Show/Hide Y axis label", value = TRUE),  
 checkboxInput("show\_title", "Show/Hide Title")  
 ),  
 mainPanel(  
 h3("Graph of Random Points"),  
 plotOutput("plot1")  
 )  
 )  
))

# server.R  
library(shiny)  
shinyServer(function(input, output) {  
 output$plot1 <- renderPlot({  
 set.seed(2016-05-25)  
 number\_of\_points <- input$numeric  
 minX <- input$sliderX[1]  
 maxX <- input$sliderX[2]  
 minY <- input$sliderY[1]  
 maxY <- input$sliderY[2]  
 dataX <- runif(number\_of\_points, minX, maxX)  
 dataY <- runif(number\_of\_points, minY, maxY)  
 xlab <- ifelse(input$show\_xlab, "X Axis", "")  
 ylab <- ifelse(input$show\_ylab, "Y Axis", "")  
 main <- ifelse(input$show\_title, "Title", "")  
 plot(dataX, dataY, xlab = xlab, ylab = ylab, main=main,  
 xlim = c(-100, 100), ylim = c(-100, 100))  
 })  
})

Essentially, it generates two sets of random uniforms and plots them. You can input the number of uniforms in the text box that also has an increment and decrement operator in the buttons on the right-hand side. The sliders enable you to alter the range of the uniforms generated on both the X and Y axes. There are also buttons available that enable you to Show/Hide the X and Y axis labels and Show/Hide the Title. So, we have two sliders, three checkboxes, and a numeric input, as shown. Observe how the code is executed interactively. Whenever there is a change in the values, the entire code is re-run. It’s important to get accustomed to the server calculations and reactive programming with Shiny, which, functions somewhat differently than standard R programming, which follows a highly linear pattern.

Let’s dive into the code and see how everything was achieved. To begin with, let’s start by analyzing the ui.r function, which is the Shiny UI function. Firstly, we have the title pane, which is self-explanatory. Then we have the numeric input, which is labeled as “numeric”. It includes a title, starting value, minimum, and maximum values, so that users cannot input values beyond the specified range. Additionally, there is a step sizer that controls the increment and decrement of values when the buttons on the right-hand side are pressed. Moving on to the two sliders, they are similar to ones we have seen before. The first slider, sliderX, has two values, which are used to determine the two points of the slider. The same applies to the Y slider. Lastly, we have three check boxes: show\_xlab, show\_ylab, and show\_title. As for the sliders, the first slider is labeled as “sliderX” and the second one as “sliderY”. The labels for the check boxes are straightforward and need no explanation. Moving on to the server function, we will generate the plot output. In the function, we define output$plot1 so that when we display it on the user interface, it will show as plot1. Since it is a plot, we use the **renderPlot** statement, which uses reactive expressions (i.e., calculations that interact with the server). Reactive statements are enclosed in curly braces next to the renderPlot statement. After setting the random number seed, we assign the number of points to the input$numeric for convenience and to make the code more readable. The minimum and maximum values for X and Y are derived from the sliders, which are returned as a list, with the X slider’s values in a vector and the Y slider’s values in another vector. We then assign these values to variables for ease of reuse.

Our data is a list of random uniform numbers generated using the runif function with the number of points and range between the minimum and maximum X values as arguments. Similarly, we generate the dataY vector using the number of points, minimum and maximum Y values. We use an ifelse statement to define the labels for the X axis, Y axis, and title. The check box input$show\_xlab determines whether or not the X axis is labeled, and its corresponding label is either “X Axis” or nothing. The same goes for the Y axis and title. Finally, we create our plot, using the X and Y data generated earlier. The xlab, ylab, and title are all dependent on the values of their corresponding labels, and the x and y limits are set to predetermined values. We then return the plot. Running the code now that we understand its workings will allow us to view the output. Let’s review what’s happening when we change a value. The Shiny server takes the input value for the server, which includes slider X value one, slider X value two, slider Y value one, slider Y value two, and show X lab and show Y lab, which can be true or false. As we input these values, the Shiny server continuously checks for new values, and every time it receives a new set of values, it re-runs and re-displays everything, communicating back and forth between the server and the ui. If your calculations are more complex, we can add a done button to prevent immediate re-running. This example is simple and fast, but later on, we’ll show you how to add a done button for more complex simulations. This is a useful tool that can be extended to many meaningful applications.

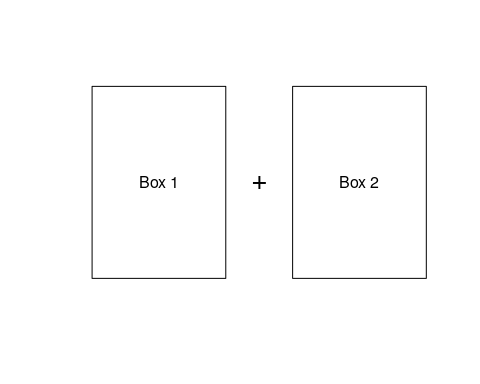
Your homework is to create an interactive graphic using Shiny and experiment with different inputs.

## 1.3 Shiny Part 2

### 1.3.1 Shiny 2.1

Welcome to the second lecture on Shiny. Today, we’ll delve into the fascinating world of reactive expressions. We’ll explore how they can be used to build more complex Shiny applications that perform sophisticated calculations, generate informative visualizations, and deliver insightful results. In the previous lecture, we learned the basics of creating interactive graphics using Shiny. However, to create more meaningful applications, we need to leverage the full power of Shiny. And that’s where reactivity comes in. A reactive expression is like a recipe that takes input from Shiny and returns a computed value. It is an essential concept in Shiny that enables us to build applications that respond dynamically to user input. Using reactive expressions may look daunting in the beginning, but with practice yo will get hang of it and utilize the additional functionality it provides. To use reactive expressions, we need to wrap the expression in a reactive function.

What exactly is a reactive expression? Essentially, it’s like a recipe that takes inputs from Shiny, manipulates them, and returns a value. However, due to the specific intricacies of Shiny, this requires some slightly unorthodox R coding. In particular, if you want isolated expressions that respond to input from your ui.r or ShinyUI function, they need to be wrapped as a reactive function. For instance, consider the scenario where we have two input boxes, labeled box one and box two, and we want to add the results from these boxes and assign the sum to a variable.



To ensure that this calculation is redone every time the input values change, we need to include it in a reactive statement.

calc\_sum <- reactive({  
 input$box1 + input$box2  
})  
# ...  
  
calc\_sum()

The output of this calculation, calc\_sum, is assigned as reactive, using a slightly unique notation that involves functional parentheses as well as curly braces containing the reactive statements. Once we add inputbox2 inside the reactive statement, the calc\_sum() function will adapt as we input new values into box1 and box2.

In the following section, we will deal with a relatively simple example that demonstrates how to create a reactive environment in Shiny. The example will provide you with a comprehensive tool set to start creating your own Shiny applications that can handle just about anything you need to do. For instance, if you have a machine learning algorithm, you can create an app that allows users to input parameters, generates predictions, and calculates prediction error on the fly.

### 1.3.2 Shiny 2.2

Here we’ll go through a code demonstration that heavily utilizes reactive expressions, and by the end of it, you’ll have a better understanding of how to incorporate these expressions into your own Shiny apps. In the following, we will first demonstrate the user interface and then the server calculations. Our objective is to showcase reactive expressions. Take a look at the code and run it to see how it works then we’ll go through it step by step.

# ui.R  
library(shiny)  
shinyUI(fluidPage(  
 titlePanel("Predict Horsepower from MPG"),  
 sidebarLayout(  
 sidebarPanel(  
 sliderInput("sliderMPG", "What is the MPG of the car?",10, 35, value=20),  
 checkboxInput("showModel1", "Show/Hide Model 1", value = TRUE),  
 checkboxInput("showModel2", "Show/Hide Model 2", value = TRUE)  
 ),  
 mainPanel(  
 h3("Plot of Predicted Horsepower"),  
 plotOutput("plot1"),  
 h3("Predicted Horsepower for Model 1:"),  
 textOutput("pred1"),  
 h3("Predicted Horsepower for Model 2:"),  
 textOutput("pred2")  
 )  
 )))

library(shiny)  
shinyServer(function(input, output) {  
 mtcars$mpgsp <- ifelse(mtcars$mpg - 20 > 0, mtcars$mpg-20,0)  
 model1 <- lm(hp ~ mpg, data = mtcars)  
 model2 <- lm(hp ~ mpgsp + mpg, data = mtcars)  
  
 model1pred <- reactive({  
 mpgInput <- input$sliderMPG  
 predict(model1, newdata = data.frame(mpg = mpgInput))  
 })  
 model2pred <- reactive({  
 mpgInput <- input$sliderMPG  
 predict(model2, newdata = data.frame(mpg = mpgInput,  
 mpgsp = ifelse(mpgInput - 20>0, mpgInput-20, 0)))  
 })  
  
output$plot1 <- renderPlot({  
 mpgInput <- input$sliderMPG  
  
 plot(mtcars$mpg, mtcars$hp, xlab = "Miles Per Gall",  
 ylab = "Horsepower", bty = "n", pch = 16,  
 xlim = c(10, 35), ylim = c(50, 350))  
  
 if(input$showModel1){  
 abline(model1, col = "red", lwd = 2)  
 }  
 if(input$showModel2){  
 model2lines <- predict(model2,  
 newdata = data.frame(mpg = 10:35, mpgsp = ifelse(10:35 - 20 > 0, 10:35-20, 0)))  
 lines(10:35, model2lines, col = "blue", lwd = 2)  
 }  
 legend(25, 250, c("Model 1 Prediction", "Model 2 Prediction"),pch=16,  
 col = c("red", "blue"), bty = "n", cex = 1.2)  
 points(mpgInput, model1pred(), col = "red", pch =16, cex = 2)  
 points(mpgInput, model2pred(), col = "blue", pch = 16, cex = 2)  
 })  
 output$pred1 <- renderText({  
 model1pred()  
 })  
output$pred2 <- renderText({  
 model2pred()  
 })  
})

We start with the shiny library statement, followed by the shinyUI function, which uses fluidPage. This page will have a title and a sidebarLayout containing a sidebarPanel and a mainPanel. The sidebarPanel will include a sliderInput labeled as sliderMPG to input the car’s miles per gallon, and a checkbox to indicate whether certain model values should be displayed in the mainPanel. The default value for the sliderInput is 20, with a range of 10 to 35 miles per gallon. Our goal is to create an output that displays the prediction for the horsepower. In the mainPanel, the inputs are showModel1 and showModel2.

The main panel will display the plot labeled plot1, which is defined in our server function to be displayed in the main panel. We will also have two text outputs labeled pred1 and pred2 that will be displayed in the main panel. In the user interface, we have a slider input labeled sliderMPG, which allows users to input the MPG of their car, and a checkbox labeled showModel1 and showModel2 to determine which model values to display in the output. The slider input has a default value of 20 and a range from 10 miles per gallon to 35 miles per gallon.

The first step in the process is to load the default “mtcars” dataset. This dataset is commonly used in R as an example dataset and is used to create a spline term. If the mpg of the car is greater than 20, the spline term returns the mpg value. Otherwise, it returns 0. This is useful when feeding a model that has a “broken stick” feed - a line with a breakpoint at 20, followed by another line that meets continuously at the breakpoint. The code fits two models, one with just mpg and another with mpg at the breakpoint in the middle. These two models are used to form the prediction. Next, we need to calculate predictions based on input values from the user interface, so it needs to be in a reactive statement. The reactive statement uses a parenthesis and a curly brace notation. We relabel the input as a regular R variable, mpgInput, to avoid typing input$ repeatedly. We use the predict statement with model1 to predict the new value from the slider. Similarly, we create model2 with an extra spline term, mpgsp, that returns the input value if it is greater than 20, otherwise returning 0. The two reactive assigned variables are model1 pred and model2 pred. The first output that we create is output$plot1, which our user interface expects to be a plot labeled plot1. We use the renderPlot notation for a plot, using the mpgInput value obtained from the slider.

Next, a plot will be created. You should be familiar with R plotting, so you can review how it’s working. However, pay attention to the conditional values for whether or not the lines for model1 and model2 will be included, which are controlled by showModel1 and showModel2. The plot also has a nice legend. Additionally, we want to include the result of our reactive expression, model1pred. However, remember that you need to access that number as a function because it was from a reactive expression elsewhere in the code. If the parentheses are not included, the function will be returned, which cannot be plotted by points because it is a function. Therefore, to obtain the value, the parentheses must be included. Another point is that the output of this function is a plot that will be returned to the user interface. It will include the two fitted lines if the check boxes are checked, the data used to fit the lines, and specific points at the prediction values. Additionally, the code is looking for pred1 and pred2, which will be text. These will be rendered using the function renderText() with the reactive expressions model1pred() and model2pred(), respectively. In both cases, the parentheses must be included for the reactive expression. Now if you run the code again, you see the plot is now generated. As we adjust the slider, you will notice the two prediction points appearing. The red line represents the model without the spline term, while the blue line has the spline term at 20, creating a broken stick-like pattern. We have successfully created our two models and added their respective predictions below the plot. Additionally, we can toggle the visibility of either of the two lines using the checkboxes.

Once you have gone through this example, you should feel confident in working with Shiny and utilizing it for more complex tasks. We encourage you to give this example a try, and we are confident that after the next lecture, you’ll have a strong understanding of working with shiny apps and be able to create useful applications for yourself and others to use on the internet.

### 1.3.3 Shiny 2.3

As we promised we are going to show you how to make the app reactive in such a way that the user has to press a button to execute the command. This conditionally reactive approach is useful for algorithms that take a lot of time to execute. If the server functions in your Shiny app involve small, quick calculations, then it’s not a problem. However, if your app requires computationally intensive tasks like a large Monte Carlo simulation or Bootstrap, you might want to have the values settled before clicking a submit button. To do this, it only requires one extra line of code in sidebarPanel.

library(shiny)  
shinyUI(fluidPage(  
 titlePanel("Predict Horsepower from MPG"),  
 sidebarLayout(  
 sidebarPanel(  
 sliderInput("sliderMPG", "What is the MPG of the car?",10, 35, value=20),  
 checkboxInput("showModel1", "Show/Hide Model 1", value = TRUE),  
 checkboxInput("showModel2", "Show/Hide Model 2", value = TRUE),  
 submitButton("Submit!") # This is the new line of code  
 ),  
 mainPanel(  
 h3("Plot of Predicted Horsepower"),  
 plotOutput("plot1"),  
 h3("Predicted Horsepower for Model 1:"),  
 textOutput("pred1"),  
 h3("Predicted Horsepower for Model 2:"),  
 textOutput("pred2")  
 )  
 )))

The button’s value is Submit, and it provides a labeled input value called Submit that you can use in your server functions, for example, if the submitButton is true.

Forgetting to include commas after functional arguments is a common mistake in Shiny app development. It’s important to remember that these commas are necessary for the app to run properly. RStudio will usually indicate an error with a little “x” symbol. In the current example, we have added a submit button using the submitButton command, which takes a labeled input value called “Submit”. This button allows us to avoid rerunning the code every time we change the slider value. Instead, we can click the submit button to run the code after we have settled on a value. This can be useful for computationally intensive tasks like Monte Carlo simulations or Bootstrap. Remember that the submit button can be used as a label button in your server functions, so you can use it to control calculations or other aspects of your app. Adding a submit button is a quick and handy way to improve the functionality of your Shiny tools.

### 1.3.4 Shiny 2.4

Let’s review a few ways to customize the Shiny user interface. By default, you have a sidebar panel theme, but we’ll explore an alternative option using tabs. Once you understand these two examples, you’ll be able to adjust the interface using various settings. Additionally, later we’ll briefly touch on using a custom HTML function for Shiny. Let’s start with the advanced UI example and wrap up with some final thoughts.

Alright, let’s go over a few ways we can modify the UI for Shiny. We will be working on the following code.

# ui.R  
library(shiny)  
shinyUI(fluidPage(  
 titlePanel("Tabs!"),  
 sidebarLayout(  
 sidebarPanel(  
 textInput("box1", "Enter Tab 1 Text:", value = "Tab 1!"),  
 textInput("box2", "Enter Tab 2 Text:", value = "Tab 2!"),  
 textInput("box3", "Enter Tab 3 Text:", value = "Tab 3!")  
 ),  
 mainPanel(  
 tabsetPanel(type = "tabs",  
 tabPanel("Tab 1", br(), textOutput("out1")),  
 tabPanel("Tab 2", br(), textOutput("out2")),  
 tabPanel("Tab 3", br(), textOutput("out3"))  
 )  
 )  
 )  
))

# server.R  
library(shiny)  
shinyServer(function(input, output) {  
 output$out1 <- renderText(input$box1)  
 output$out2 <- renderText(input$box2)  
 output$out3 <- renderText(input$box3)  
})

You might already be familiar with the first few lines of code, as we’ll still be using a sidebar layout. However, we can also create tabs within a single panel, or have a panel with tabs but no sidebar. And if you’re feeling adventurous, we can even customize the UI with custom HTML. For our sidebar panel, we’ll have three text input boxes labeled “box1”, “box2”, and “box3”. The prompts for the user will be “Enter Tab 1 text”, “Enter Tab 2 text”, and “Enter Tab 3 text”, respectively, with starting values of “Tab 1”, “Tab 2”, and “Tab 3”. Moving on to the main panel, we’ll be using tabsetPanel function, which is a nested user interface structure in Shiny. We’ll use the mainPanel function with a tabsetPanel inside, specifying type = "tabs". Within that, we’ll have three tabPanel functions, each with a label indicating “Tab 1”, “Tab 2”, or “Tab 3”, and a textOutput that will look for the corresponding output label from the server.R file. The server.R function will be a simple one, using shinyServer with function(input, output) arguments. We’ll use renderText to display the text entered in each of the three input boxes. You can run the app and see how it works.

Be sure to avoid errors in your R code while making UI changes, as RStudio will show you X’s in the code if you forget to close parentheses or add commas. Proper indentation and using a nice environment like RStudio will make the task easier.

figure xxx

Your homework is to create a tabbed user interface for Shiny. Finally, we’ll go over some last comments about Shiny.

### 1.3.5 Shiny 2.5

We have repeatedly promised to show you how to use your own HTML, and we have a quick way to do that. We took our last example, saved the HTML, named it index.html and placed it in a subdirectory named www. You should also make sure you have the server.R function, in the same directory as www. The index.html file contains a lot of unnecessary preamble from the bootstrap style it gets from that package. So, you can remove most of that and have the style be whatever you want. If you’re not familiar with web development, you can start with a simple shiny page, output the HTML, and use that as a starting point to build from. But if you’re comfortable with web development, you can start with a blank HTML file and work from there. The inputs are the same way as regular HTML inputs.

<!DOCTYPE html>  
<html>  
  
<head>  
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />  
 <script type="application/shiny-singletons"></script>  
 <script  
 type="application/html-dependencies">jquery[3.6.0];shiny-css[1.7.4];shiny-javascript[1.7.4];bootstrap[3.4.1]</script>  
 <script src="jquery-3.6.0/jquery.min.js"></script>  
 <link href="shiny-css-1.7.4/shiny.min.css" rel="stylesheet" />  
 <script src="shiny-javascript-1.7.4/shiny.min.js"></script>  
 <meta name="viewport" content="width=device-width, initial-scale=1" />  
 <link href="bootstrap-3.4.1/css/bootstrap.min.css" rel="stylesheet" />  
 <link href="bootstrap-3.4.1/accessibility/css/bootstrap-accessibility.min.css" rel="stylesheet" />  
 <script src="bootstrap-3.4.1/js/bootstrap.min.js"></script>  
 <script src="bootstrap-3.4.1/accessibility/js/bootstrap-accessibility.min.js"></script>  
 <title>Tabs!</title>  
</head>  
  
<body>  
 <div class="container-fluid">  
 <h2>Tabs!</h2>  
 <div class="row">  
 <div class="col-sm-4">  
 <form class="well" role="complementary">  
 <div class="form-group shiny-input-container">  
 <label class="control-label" id="box1-label" for="box1">Enter Tab 1 Text:</label>  
 <input id="box1" type="text" class="form-control" value="Tab 1!" />  
 </div>  
 <div class="form-group shiny-input-container">  
 <label class="control-label" id="box2-label" for="box2">Enter Tab 2 Text:</label>  
 <input id="box2" type="text" class="form-control" value="Tab 2!" />  
 </div>  
 <div class="form-group shiny-input-container">  
 <label class="control-label" id="box3-label" for="box3">Enter Tab 3 Text:</label>  
 <input id="box3" type="text" class="form-control" value="Tab 3!" />  
 </div>  
 </form>  
 </div>  
 <div class="col-sm-8" role="main">  
 <div class="tabbable">  
 <ul class="nav nav-tabs" data-tabsetid="4040">  
 <li class="active">  
 <a href="#tab-4040-1" data-toggle="tab" data-bs-toggle="tab" data-value="Tab 1">Tab 1</a>  
 </li>  
 <li>  
 <a href="#tab-4040-2" data-toggle="tab" data-bs-toggle="tab" data-value="Tab 2">Tab 2</a>  
 </li>  
 <li>  
 <a href="#tab-4040-3" data-toggle="tab" data-bs-toggle="tab" data-value="Tab 3">Tab 3</a>  
 </li>  
 </ul>  
 <div class="tab-content" data-tabsetid="4040">  
 <div class="tab-pane active" data-value="Tab 1" id="tab-4040-1">  
 <br />  
 <div id="out1" class="shiny-text-output"></div>  
 </div>  
 <div class="tab-pane" data-value="Tab 2" id="tab-4040-2">  
 <br />  
 <div id="out2" class="shiny-text-output"></div>  
 </div>  
 <div class="tab-pane" data-value="Tab 3" id="tab-4040-3">  
 <br />  
 <div id="out3" class="shiny-text-output"></div>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
</body>  
  
</html>

In this example, we have three boxes labeled box1, box2, and box3. So, the input ID for Box 1 is “box1”, for Box 2 it’s “box2”, and for Box 3 it’s “box3”. The div statements are used because the Bootstrap style used by Shiny uses these blocky structures to create the default style.

Moving on to the output, you can see that it references “out2” and has the class “shiny-text-output”. If you’re familiar with web development, you may have recognized the output class “shiny-text-output” already. Otherwise, building a Shiny prototype app and grabbing its HTML would give you enough information to work with. If you build a prototype and save the HTML, you can use it directly in your Shiny app by placing it in a file named index.html in a subdirectory named www. This method will be very useful for those who are experienced with web development and find using R as an intermediary to be a hassle. By using a straight HTML file, you can avoid having to insert individual HTML elements in R. However, you’ll still need to use R to create the server-side logic.

### 1.3.6 Shiny 2.6

In this example, we’ll demonstrate how to create an interactive graphic using the brush argument in the UI and brushedPoints argument in the server. The purpose is to highlight points in a scatter plot and show the fitted line just for those points. Interactive graphics are a key functionality of Shiny and one of its primary uses for analysis. Let’s take a look at the code. We’ll go through the various lines, but we encourage you to try it out yourself by copying and running the code. You’ll likely gain a better understanding by doing so.

We start with the standard layout, including a sidebar. The sidebar displays the slope and intercept using textOutput (intOut) and textOutput (slopeOut), respectively. We need to make sure to include both labels in the server function, otherwise they won’t display correctly. Additionally, we create a plot in the server labeled plot1 and make sure to include plotOutput for it in the UI. The UI also includes an id of brush1 and the brushOpts for the brush options. We’ll need to use the label brush1 on the server side.

Next, let’s examine the server function. The input and output lists are passed into the Shiny server. A reactive model is created that responds to changes in the data input from the user interface. The notation for reactive functions is inside curly braces and parentheses. The variable brushed\_data is shorthand for retrieving the data from brush1 in the input. The brush points come from the trees dataset with Girth as the x-variable and Volume as the y-variable. If there are fewer than two points, it returns null, otherwise, it fits a linear model with Volume ~ Girth for brushed data. The resulting linear model is output without parentheses because it’s inside the reactive statement. In the Shiny server, we check if the model is fit or if the label output$slopeOut is present. We use the renderText function to let Shiny know that text will be displayed in the user interface. Inside the curly braces, we check if the slope is null, and if so, it will display “No Model Found”. Otherwise, it will display the slope term. Now, again go ahead and run the app and see how it looks.

We have run the application and observed that when no model is fit, the text output for slope and intercept correctly shows “No Model Found”. If a cluster of points is highlighted, the slope and intercept are calculated and displayed as a line. If no points are selected, the text output again shows “No Model Found”. A line is also displayed when all the data points are selected. This example shows the use of Shiny for creating interactive graphs. We encourage you to try to recreate this example and also create your own interactive graph using one of R’s built-in datasets. Shiny has many uses, including input and output of data frames, as well as embedding RGL and RS 3D graphics library in a Shiny webpage using WebGL. Although these advanced features may be more specialized, you should now have enough understanding to get started and incorporate Shiny into your toolkit.

## 1.4 Shiny Part 3

### 1.4.1 ShinyUiEditor

The Shiny UI Editor is a user-friendly, visual tool designed to simplify the process of creating the user interface (UI) for Shiny applications. Its primary objective is to enable users to build the UI of their Shiny apps without the need to write code. This editor is particularly useful for individuals who may not be familiar with Shiny’s UI functions that require HTML-style coding or who prefer a hassle-free approach for achieving proper layout without manually adjusting sizes. By utilizing the Shiny Ui Editor, users can effortlessly generate clean and easily understandable code for their Shiny app’s UI, streamlining the development process. **Note:** According to the developers’s [website](https://rstudio.github.io/shinyuieditor/) ShinyUiEditor currently is in the Alpha phase of development, which means it may be unstable, or some users may experience bugs while installing the package. However, we think it is still a useful tool for creating Shiny apps, and we encourage you to try it out. These problems will be resolved as the package is further developed. For more information on the Shiny Ui Editor, please visit the following link: <https://rstudio.github.io/shinyuieditor/>. This link will also provide you with instructions on how to install the Shiny Ui Editor package.

#### 1.4.1.1 Your first app in ShinyUiEditor

In this example, we’ll demonstrate how to create a simple Shiny app using the Shiny Ui Editor. We’ll start by creating a new Shiny app and then proceed to build the UI using the Shiny Ui Editor. Once we’re done, we’ll run the app to see how it looks. Let’s get started! In the Rstudio call ‘library(shinyuieditor)’ and run the launch\_editor(app\_loc = “newApp”). You must provide a string as your app name to variable app\_loc. This will create a folder in the directory with the string name you provided and open the Shiny Ui Editor in a new window in your browser and asks for you to choose some options to create a new Shiny app, steps are shown in the image below(xxx). Any update that you make in the Ui Editor will automatically update the app code itself. To follow the updates in UiEditor you can simply check the app in App Preview tab located on the bottom right of the Ui Editor window. (xxx) If you don’t like the layout you can delete the elements and redesign the layout to your liking. You can customize the size of each grid with your mouse or providing values in pixel or the relative units by hovering mouse to the left side of the desired grid. To delete an element click on the element and select the delete button on the properties window on the right side of the windows. In the R code file you can import or generate your own data and use that to make plots instead of the default data. The beauty of the Ui Editor is that you have all the common elements that you can add to your app directly from the Ui Editor window. This will make your life easier and you can focus on the data analysis part of your app.

### 1.4.2 Layouts

A web layout refers to the arrangement and structure of visual elements on a web page. It defines how different components such as text, images, menus, forms, and other interactive elements are positioned and organized within the overall design of a website. The layout plays a crucial role in determining the visual hierarchy, flow, and user experience of a web page.

Web layouts can vary widely in their structure and design, depending on the purpose of the website and the desired aesthetic. Common types of web layouts include:

1. Fixed Layout: In this type of layout, the content is set to a specific width and remains fixed regardless of the screen size or resolution. It often uses a centered design with a static width.
2. Fluid Layout: Fluid layouts adapt to the available space and adjust their width proportionally to the screen size or browser window. This ensures that the content expands or contracts based on the user’s viewing device or browser window size.
3. Responsive Layout: Responsive layouts are designed to provide an optimal viewing experience across various devices and screen sizes. They use CSS media queries to adapt the layout, content, and design elements dynamically based on the user’s device, whether it’s a desktop, tablet, or mobile phone.
4. Grid Layout: Grid layouts utilize a system of horizontal and vertical lines to create a structured framework for organizing content. They often involve dividing the page into multiple columns and rows, providing a consistent and balanced arrangement for different elements.
5. Single-Page Layout: Single-page layouts present all the content on a single scrolling page, eliminating the need for navigating to different pages. This layout is commonly used for portfolios, landing pages, or websites with a concise amount of content.

Web layouts are essential for achieving an aesthetically pleasing design, ensuring effective information presentation, and enhancing the overall user experience of a website.

The Shiny base package provides some common functions such as fluidPage(), splitLayout(), and fixedPage() to generate layouts. These build-in functions are useful for creating simple layouts, but they can be limiting when it comes to more complex designs. Fortunately, there are several packages available that provide more advanced layout options for Shiny apps. Here we introduce some of the most popular packages for creating layouts in Shiny.

#### 1.4.2.1 Bootstrap layouts

Bootstrap has been around for more than 10 years now. It is a versatile and powerful framework for building responsive, mobile-first websites and web applications. Bootstrap is one of the most popular front-end frameworks for web development, and it is widely used by developers around the world. It provides a 12-column grid system, buttons, forms, tables, navigation bars, and more. bslib is an R package that provides an alternative method for customizing the appearance of Bootstrap-themed user interfaces. With bslib, you can define your theme’s color palette, typography, spacing, and other visual attributes using Sass (Syntactically Awesome Style Sheets), a CSS preprocessor. This allows for more advanced customization and flexibility compared to the default theming options provided by Bootstrap.

Bootstrap 3 relies on the grid layout system, which allows to efficiently organize the content in rows and columns. For instance the fluidRow() and column() Shiny functions leverage the Bootstrap grid to control how elements are displayed on a page.

##### 1.4.2.1.1 bslib

The bslib R package offers a contemporary UI toolkit for Shiny and R Markdown, leveraging the Bootstrap framework. This package enhances the visual design and user interface capabilities of Shiny and R Markdown by incorporating modern styling and layout options. Working with bslib is straightforward and intuitive, and it provides a wide range of customization options for creating beautiful and responsive web applications.

##### 1.4.2.1.2 gridlayout

The gridLayout function in Shiny is a versatile layout option that allows you to create grid-based arrangements for your user interface components. With gridLayout, you can define the number of rows and columns in the grid and specify the placement of UI elements within specific cells. This layout could be extremely handy while using the Shiny Ui Editor.

##### 1.4.2.1.3 dashboardPage

The dashboardPage layout in Shiny provides a structured and organized framework for creating interactive dashboards. It consists of different sections, such as the header, sidebar, and body, that can be customized to build a visually appealing and functional dashboard. Here’s an overview of the different components of the dashboardPage layout:

1. dashboardHeader: This component represents the header section of the dashboard and typically includes the title or logo of the application. You can customize the appearance of the header by adding titles, logos, navigation menus, or other elements.
2. dashboardSidebar: The sidebar is a vertical panel located on the left side of the dashboard. It provides space for navigation menus, input controls, or additional information. You can define the sidebar content using functions like sidebarMenu, menuItem, or renderMenu.
3. dashboardBody: The body section of the dashboard occupies the remaining space after the header and sidebar. It is where you place the main content of your dashboard, such as graphs, tables, text, or other visualizations. You can structure the body using different layouts like fluidRow, column, or tabItems.
4. fluidRow and column: These layout functions allow you to create rows and columns within the dashboard body. They provide a responsive grid system that helps in arranging and positioning the UI elements. You can define the number of columns and their relative widths using the column function within a fluidRow.
5. tabItems and tabItem: If you want to organize your dashboard content into multiple tabs, you can use the tabItems and tabItem functions. Each tabItem represents a single tab within the dashboard, and you can place specific content within each tab, such as graphs, tables, or other UI elements.

By combining these components and customizing their content, you can create a personalized dashboard layout in Shiny. The dashboardPage layout provides a convenient structure for organizing and presenting data, visualizations, and interactive elements within a single web application.

##### 1.4.2.1.4 imola

The imola is one of the recently developed layouts for Shiny, which is based on the CSS Grid Layout Module. It provides a simple and intuitive way to create responsive layouts for Shiny apps. The imola package offers a wide range of customization options for creating beautiful and responsive web applications. It also provides a set of predefined themes that can be used to quickly change the appearance of your app. The imola package is still in the early stages of development, but it shows a lot of promise for the future of Shiny layouts. There are multiple [predefined layouts](https://sparktuga.shinyapps.io/imolatemplates/) with imola. You can checkout the website and use the example code to create your app.

There some other packages that you can use as themes for your Shiny app to reach a unique look in your apps. Here we introduce some of them for the interested readers. <https://github.com/RinteRface/shinydashboardPlus> <https://github.com/RinteRface/shinyMobile/> <https://pkgs.rstudio.com/flexdashboard/> <https://github.com/RinteRface/argonDash> <https://github.com/ericrayanderson/shinymaterial> <https://github.com/Appsilon/shiny.semantic> <https://rstudio.github.io/shinydashboard/> <https://github.com/nik01010/dashboardthemes>

## 1.5 Shiny Gadgets

### 1.5.1 Shiny Gadgets 1.1

In this lecture, we’ll be discussing Shiny Gadgets. I want to acknowledge Shaun Cross from the Data Science Lab for helping with this content. Previously, we covered a package called Manipulate, which was created by our studio for easy and simple interactive data analysis. However, it seems that development on Manipulate has stopped and we have a hunch why. Shiny provides a superior way of achieving the same goal, with Shiny Gadgets, you can do everything that Manipulate did, but within the RStudio program.

The focus of this lecture is on Shiny Gadgets, which offer an easier and faster way to use Shiny’s interactivity in data analysis. Unlike Shiny, which is suitable for creating forward-facing apps for a wider audience, Shiny Gadgets are primarily designed for use by us or a small group of people working on data analysis. To display the app on a small screen within our RStudio program, we’ll use the mini UI package, which helps create interfaces optimized for smaller screens. The core of a Shiny Gadget is a function that launches a small, single-page Shiny application in the RStudio viewer pane. Furthermore, since the Shiny Gadgets will be displayed in the RStudio viewer pane, the miniUI package is useful for creating interfaces that fit well on smaller screens. Now, let’s switch over to a code demonstration.

To begin, we need to ensure that both Shiny and miniUI are loaded, if you don’t have these packages make sure you install them. The gadget is a function with a user interface and a server function, just like a Shiny app. The UI here uses miniPage, which is a layout from the miniUI library. We create a title bar element using the gadgetTitleBar command and call it My First Gadget. The server function takes input, output, and session arguments. For now, it checks the input for the done variable and uses the observeEvent function to stop the app when it is true. It is important to note that the syntax for Shiny Gadgets is a bit different from regular R programming, but the rules of reactivity still apply. Finally, we use the runGadget function to run the gadget with the UI and server functions. Once we run the function, we see “My First Gadget” displayed in the viewer pane, and clicking “done” takes us back to the RPrompt. This is a good first step to becoming familiar with Shiny Gadgets, and more complex concepts will come naturally as we progress.

### 1.5.2 Shiny Gadgets 1.2

The previous example we looked at was rather trivial, as it merely had a button to click and exit. It even had a built-in cancel button to stop the gadget and return to the R prompt. Now, we’ll demonstrate how to add arguments to your Shiny Gadget. We’ll keep it simple and show basic manipulations so that you can understand the coding process and apply it to your own application. To proceed with this section, please copy and run the following code.

Let’s examine the code in detail. The code is divided into two parts due to space limitations. The first part contains the UI, which is a mini page and the second part contains the server. Our gadget’s title will be Multiply Two Numbers. The content panel will be a mini content panel, which will be the main body of the pane. The user interface will have a selector input, which will be a dropdown box with two variables labeled num1 and num2. We will use these names to reference them later. The labels and choices come from the two arguments in the function. Moving on to the server part, we will have input, output, and session. We will use the input$done variable in the observe event to let Shiny know when it’s time to quit. We must use curly braces in the correct place to avoid the most common mistake when coding Shiny applications. We will name num1 and num2 as the input values to make them easier to work with. We will then multiply them together, and the stopApp function which will tell Shiny to stop and display the result of num1 multiplied by num2. Remember to include the runGadget function, which specifies the UI and server. Try running the function with inputs like multiplyNumbers(1:10,1:10). Suppose we want to check our multiplication tables, if we choose 4 and 7 we will get the result of 28. Once we click “Done”, we will return to the R pane where we should see the result of 28. This is a simple example, but it demonstrates how to use Shiny Gadgets to create interactive applications. Now, let’s look at some other viewers that we can use to display our gadgets.

1. paneViewer: This is the default viewer, which displays the gadget in the viewer pane. runGadget(ui, server, viewer = paneViewer(minHeight = 500))
2. dialogViewer: This is the default viewer, which displays the gadget in a dialog box. runGadget(ui, server, viewer = dialogViewer("ggbrush"))
3. browserViewer: This is the default viewer, which displays the gadget in the RStudio browser. runGadget(ui, server, viewer = browserViewer())

### 1.5.3 Shiny Gadgets 1.3

One of the most valuable features of Shiny gadgets is the ability to create interactive plots. As many of you are already familiar with Shiny syntax from our previous modules, we won’t spend too much time revisiting that in the context of Shiny gadgets. Instead, we’ll present one example, and with your understanding of Shiny and a little extra syntax for Shiny gadgets, you’ll be able to construct and customize your own gadgets to meet your needs.

In this example, our goal is to create an interactive plot of the trees dataset, specifically plotting the girth by volume. We want to have a pair of crosshairs that will allow us to select a rectangle on the plot, and then have our function output a data frame of the trees within that rectangle. As always, we need to load the Shiny and mini libraries. Our Shiny gadget is a function that requires a user interface (UI). For the UI, we will use the miniPage function from the mini package. We will have a gadgetTitleBar with the message **Select points by dragging your mouse**  and a miniContentPanel containing a plot with the name plot. We want the height to be 100% and we will name the crosshairs brush.

Moving on to the server function, we will use the names plot and brush that we defined earlier. The output named plot will be created using the renderPlot function. We will plot the girth versus volume data from the trees dataset and label the plot with the title trees and axis labels girth and volume. Next, we use observeEvent to collect the brush points and stop the app after a rectangle is selected from the trees dataset. The input will be input$brush, which refers to the named input we defined earlier. The variables xvar and yvar need to be named correctly and matched to the corresponding names in the trees dataset. Finally, we call the runGadget function with viewer = dialogViewer("ggbrush") to display the plot in a popup window. By running this code, we can create a useful tool for selecting data points from a plot. Now we can source the function and call it. The function doesn’t require any arguments. After We’ve made our selection, we can move the box around if we need to. Once satisfied with the selection, we click the “Done” button, and it returns the selected data points. To save the selected data points, we can assign them to a variable like treesIPicked, and then we can retrieve them later by calling the variable. This feature can be useful if you need to select specific data points, for example, to identify outliers in a data analysis. Note that the original data frame’s row numbers are retained, enabling you to refer back to the data frame itself.

We hope these tools provide you with a good foundation to build your own Shiny gadgets. Learning how to build them is definitely worth it, especially for creating interactive graphics which can be extremely helpful. R has not historically excelled in this area, so having the add-on provided by RStudio is a valuable resource to have in your workflow. For this reason alone, it’s worth taking the time to learn how to build and incorporate shiny gadgets into your work.

### 1.5.4 Shiny Gadgets 1.4

In this section we will introduce some functions which can provide you with more design options towards a sophisticated gadget.

#### 1.5.4.1 miniTabstripPanel

If your gadget is generating multiple outputs such as graphs, tables, maps, etc. you can use the miniTabstripPanel function to create a tabbed panel. This function allows you to create a tabbed panel with multiple tabs, each containing a different output. The miniTabstripPanel function takes a list of tab names and a list of tab contents as arguments. The tab names are displayed as the labels for each tab, and the tab contents are the UI elements that will be displayed within each tab. The miniTabstripPanel function is useful for organizing and presenting multiple outputs within a single gadget. Here is an example of a gadget that uses the miniTabstripPanel function to create a tabbed panel with two tabs. The first tab contains a plot, and the second tab contains a table.

library(shiny)  
library(miniUI)  
  
multipanel<-function(){  
 ui <- miniPage(  
 gadgetTitleBar("My Shiny Gadget"),  
 miniTabstripPanel(  
 miniTabPanel("Plot", icon = icon("area-chart"),  
 miniContentPanel(  
 plotOutput("plot")),  
 ),  
 miniTabPanel("Data", icon = icon("table"),  
 miniContentPanel(  
 dataTableOutput("table")  
 )  
 )  
 )  
 )  
  
server <- function(input, output, session) {  
 output$plot <- renderPlot({  
 plot(trees$Girth, trees$Volume, main = "Trees!",  
 xlab = "Girth", ylab = "Volume")  
 })  
 output$table <- renderDataTable({  
 diamonds[1:50,]  
 })  
 observeEvent(input$done, {  
 stopApp(TRUE)  
 })  
 }  
runGadget(ui, server,viewer = dialogViewer("ggbrush"))  
}

#### 1.5.4.2 fillRow/fillCol

These functions are beneficial for partitioning an area into rows and columns that dynamically expand and contract in proportion to fill the available space. Unlike fluidRow and col functions, which do not adjust, fillRow and fillCol offer growth and shrinkage capabilities.You can get creative with these functions and use them to create a variety of layouts for your Shiny app or Shiny gadget.

library(shiny)  
library(miniUI)  
  
multipanel<-function(){  
 ui <- miniPage(fillRow(  
 plotOutput("plotLeft", height = "100%"),  
 fillCol(  
 plotOutput("plotTopRight", height = "100%"),  
 plotOutput("plotBottomRight", height = "100%")  
 )  
 ))  
  
 server <- function(input, output, session) {  
 cut\_graph = qplot(cut, data = diamonds) +  
 ggtitle("Quality of cut")  
 output$plotLeft <- renderPlot(qplot(x=carat, y=price, facets=.~cut, data=diamonds))  
 output$plotTopRight <- renderPlot(cut\_graph)  
 output$plotBottomRight <- renderPlot(hist(diamonds$price))  
 }  
  
 runGadget(ui, server,viewer = dialogViewer("ggbrush"))  
}

The code above will generate two columns the left one with single row and the right one with two rows.

# 2 Second week

During this module, we’ll learn how to create R Markdown files and embed R code in an Rmd. We’ll also explore Leaflet and use it to create interactive annotated maps. Learning Objectives Create an R Markdown file Employ R code in an Rmd Create a map using Leaflet Use Leaflet to add legends, markers, circles, and rectangles to your map

## 2.1 Quarto

Quarto is a freely available publishing system designed for scientific and technical content. It is based on the popular Pandoc tool. Quarto is a publishing system that allows you to create and publish content to Posit Connect, a publishing platform. With Quarto, you can create and publish your content to Posit Connect, a publishing platform. Before utilizing Quarto, it is necessary to have it installed and properly configured in Posit Connect. In case you don’t have Quarto support, reach out to your Posit Connect administrator. Detailed instructions on setting up Quarto in Posit Connect can be found in the Admin Guide’s Quarto chapter.

Quarto content supports executable code in multiple languages, and supports interactivity through a number of technologies.

Posit Connect supports most of Quarto’s capabilities, including:

Rendering R, Python, and Observable JS code. Running interactives written with Shiny, ObservableJS, and R htmlwidgets. Rendering Quarto projects, documents, websites, and books. Posit Connect doesn’t support the following Quarto capabilities:

Rendering Julia code chunks. Running interactives written with Jupyter Widgets. <https://docs.posit.co/connect/user/quarto/>

## 2.2 Publishing Quarto Content

There are multiple ways to publish Quarto content to Posit Connect. See the linked documentation for each method for more on how to use it to publish Quarto content. (rewrite)

## 2.3 R Markdown

### 2.3.1 R Markdown 1.1

This lecture will discuss how to create reproducible presentations using R Markdown. In previous versions of the class, we covered several ways to do this, including R Studio Presenter and Slidify. However, in this latest version, we will focus solely on Markdown. R Markdown is already covered in the data science specialization’s class on reproducible research, but we will still touch on why it’s important to create reproducible documents and presentations. R Markdown is a feature of R Studio that enables the creation of documents, such as HTML documents, directly from within R. What makes it particularly useful is that it embeds code, results, and plots from the analysis into the document, making it a self-contained and reproducible unit. In contrast, traditional workflows typically involve separate documents and code files, with manual copying and pasting or saving and importing files, which can create issues with version control and reproducibility. R Markdown provides a consistent structure for generating presentations and documents, making it a better option for those concerned with reproducibility. For instance, in a corporate setting where a report needs to be presented routinely, having a script that generates the presentation with the latest data in a version-controlled way would be more efficient and reliable than manually copying and pasting each time. R Markdown can be useful for creating recurring or automated presentations, but it has many other uses as well. One important benefit is its ability to improve the reproducibility of documents. In the academic community, there have been many instances where papers have been published and subsequent groups have tried to reproduce the results, only to find that it’s often not possible. This can lead to disputes and uncertainty over who is right. However, embedding the code in a document using R Markdown greatly improves its reproducibility. While there may still be some external factors that impact the results, overall it goes a long way towards creating more reliable and reproducible documents. Let’s hope that you’ve already covered some of these concepts in your reproducible research class as part of the data science specialization. However, if you’re taking this class in Ovo, we’ll guide you through a few examples with a focus on creating reproducible presentations. This class emphasizes the creation of products, and your presentation serves as your product pitch. Therefore, we want your pitch to be in one of these reproducible formats. So, let’s move on to some examples.

### 2.3.2 R Markdown 1.2

Alright, we have reached this slide in our presentation. From here on, we will be demonstrating the code to create reproducible presentations using R Markdown. So, instead of going through the slides one by one, we will switch to R Studio and show you how to do this practically. It’s important to highlight that we follow the same approach we’re teaching, and we have actually created the slides for this class using R Markdown. However, for the purpose of this demonstration, we will create a new presentation. To do so, we need to go to File > New File > R Markdown, which can also be accessed through Alt+F > F > R Markdown, that is M. Once there, we will see a “Presentation Title” field, where we can name our presentation “My Fancy Presentation” and add our name as the author. There are various output options available, but for now, let’s stick with the ioslides format as the default option. However, if you prefer a PDF output, it requires the installation of MiKTeX on Windows, MacTeX on OS X, or TeX Live on Linux. These installations are easy to set up but can be quite large. We’ll cover this in more detail later in the lecture. The ioslides format was created based on one of the Google presentations. Okay, so the file name for the R Markdown commands is currently “Untitled1”. Let’s compile it once to see what it looks like. You can either click on “Knit the current document” or use the shortcut Ctrl+Shift+K. I’m going to name my file “briansSlides” instead of the default name. And there you go, it’s running and the output is “My Fancy Presentation”. Your homework is to create a new presentation using the template, compile it once, and then we’ll move on to adding content.

### 2.3.3 R Markdown 1.3

Let’s take a look at the preamble section surrounded by three dashes. You can change the title, author, and date from here. You can also change the format if you want it to be a different type of output, such as “slidy” instead of “ioslides”. The knitr options specify that code included in the document won’t be echoed by default. To create a new slide, add two pound signs and then the slide title, such as “My New Slide”. Let’s knit the document to see the changes. Alright, now we have a slide with some content and a slide without a title. As you can see, we used two pound signs to create a slide with a title, and three asterisks to create a slide without a title. Let’s knit it again to see how it looks. Let’s take a look at our new slides with some content, and the slide without a title. Now, let’s add a subheading. We’ll use two pound signs for “My title”, three pound signs for a top-level subheading, and four pound signs for a second-level subheading. Let’s knit it to see how it looks. Remember, you can also use Ctrl+Shift+K to knit the document. Now, let’s add some more content by creating a list with two bullet points. Okay, let’s take a look at the YAML preamble first. The YAML section is surrounded by three dashes, and it contains information such as the title, author, date, and output format. You can easily change these options at this point.

To create a new slide, simply add two pound signs and the title of the slide. For example, “## My New Slide”. To create a slide without a title, use three asterisks like this: “\*\*\*“.

To add subheadings, use three or four pound signs depending on the level of the subheading. For example, “### Top Level Subheading” and “#### Second Level Subheading”.

To create a bulleted list, use two dashes to start each item. For an ordered list, use numbers followed by a period. For example, “1. Item a” and “2. Item b”.

Once you have added your content, you can knit the document to see the final output. You can use the shortcut Ctrl+Shift+K to knit the document.

Now, it’s time to create some slides with your own content. Add titles, headings, subheadings, bulleted and ordered lists to your slides.

### 2.3.4 R Markdown 1.4

Let’s take a look at some formatting options. For instance, if I write plain text, *italicized text*, and **bold text**, you can see that it’s automatically syntax highlighted. Additionally, I can add some code, such as 1:10, by using single right-facing quotation marks. Remember that to use italics, you need to have one asterisk next to the leading character, and to use bold, you need two asterisks next to the leading character. It’s important to note that the asterisks must be next to the leading character for it to work correctly. Let’s run it now and see the results.

### 2.3.5 R Markdown 1.5

Alright, let’s take a look at some formatting options. If I just type in plain text, *italicized text*, and **bold text**, notice how the formatting automatically adjusts. Also, here’s an example of some code: 1:10. To indicate that it’s R code, we surround it with single quotation marks and specify the language as “r” in the curly braces at the beginning. You’ll notice that the code is highlighted differently than the rest of the text. Let’s add a bit more code: head(mtcars). This will show the first few rows of the dataset “mtcars”. You can actually run the code in RStudio and see the output down below. However, to generate the entire document, hit Ctrl+Shift+K and there you have it - a new slide. Okay, let’s experiment with the options available. I’ve deleted some of the slides, so we can focus on this simple code example. I noticed that I don’t like the comment characters in front of the output, so I’ll use the comment option to remove them. Let’s try it out with comment = "". Great! Now, let’s play around with some more options. I deleted some slides so we can focus on this simple code example. Initially, I didn’t like the comment characters in front of the output, so I tried using the “comment” option to get rid of them. I set it to nothing, and it looks better now.

By default, the “echo” option was set to false, which means that it didn’t show the code that generated the output, just the output itself. When typing in Knitr options, you can see a list of available options once you add a comment and start typing.

If I set “echo” to true and regenerate the presentation, you’ll see that it now shows both the output and the code that generated it. The default option is “echo=true”, which means that you don’t have to specify it explicitly. I could have also just removed the Knitr options at the beginning and it would have behaved the same way, since “echo=true” is the default.

Let’s try this instead:

Now, let’s experiment with some Knitr options. By default, the echo option is set to false, which means that it doesn’t show the code used to generate the output. If we want to see the code, we can set echo to true. For example, if we use echo = TRUE, we will see both the code and the output.

If we want to show the code but not evaluate it, we can set eval to FALSE. On the other hand, if we want to hide the code and only show the output, we can set echo to FALSE.

There are many other options available in Knitr that can be used to customize the output of our R code. To change the comment character, for example, we can use comment = "".

Now, it’s your turn to try it out. Create a slide with some R code and experiment with different Knitr options to see how they affect the output. In the next section, we’ll cover how to include figures in our presentations.

### 2.3.6 R Markdown 1.6

Alright, let me show you a plot that I created using the code: plot(empty\_cars$weight, empty\_cars$mpg, xlab="Weight", ylab="MPG"). Here’s what it looks like:

[My Awesome Figure]

As you can see, I also used some of the figure options in knitr, such as echo = FALSE to not show the code, fig.align = "center" to center the figure, and fig.cap = "My AWESOME figure" to add a caption. There are many other figure options that you can explore in knitr. Let me summarize what you just explained. To show a plot without the code, you can set the “echo” option to FALSE when calling the plot function. You can also use other figure options such as “fig.align” and “fig.cap” to control the alignment and add a caption to the figure. To publish your slides, you can use R markdown to create an HTML file, which can be opened in a browser. You can also use R pubs, Drop Box, or GitHub to publish it online. GitHub is one of the most popular ways to share code and collaborate with others. Let me demonstrate how to publish your slides using different methods. First, let’s review some of the options for publishing your slides in R Markdown. When you knit your R Markdown document using knit HTML, it creates an HTML file in your directory. For example, if your R Markdown file is named briansSlides.Rmd, it creates a briansSlides.html file. You can open this file in a web browser to view your presentation.

You can also publish your slides online using several platforms. One option is R Pubs, where you can share your presentation with others. Another option is to put it in a Dropbox public folder, or you can publish it on GitHub.

To publish your slides on GitHub, you can create a repository and push your R Markdown and HTML files to it. For example, Sean Cross’ Developing Data Products course has a page for slides in his GitHub repository. You can see that it is just an R Markdown file and the associated HTML file pushed up into GitHub.

When you click on the R Markdown file in GitHub, it does a lot of the formatting for you. It recognizes that it is an R Markdown document and does a lot of formatting ahead for you. You can see what the bullets and various points are going to look like. When you click on the HTML file, it shows you the raw HTML, not the rendered presentation. Okay, let’s discuss how to render the HTML file. There are different ways to do it, such as using a Chrome or Firefox add-on, but Sean published his slides through GitHub Pages. He created a branch called “gh-pages,” which is required for GitHub Pages to work. Usually, a Node.js file is also needed in the root directory, but it seems Sean didn’t have to use it. To access the rendered HTML, you just need to slightly modify the URL by replacing “github.com/seankross” with “seankross.github.io” and removing “slides/blob/gh-pages” from the path. Then, you just need to navigate to the subdirectories where the HTML file is located. For example, in Sean’s GitHub repository, the file is in “slides/Developing Data Products/R Markdown/R Markdown.html.” These slides are now hosted on GitHub, which is a convenient option for data scientists who are already using GitHub to manage their projects. Personally, I push my course materials to a public GitHub repository and my private research to a private repository. Okay, now it’s time to create a figure in your presentation and experiment with the different figure options available. I also want you to try publishing your presentation to GitHub. If you encounter any issues, read up on GitHub pages or try publishing to Rpubs instead. You can easily publish to Rpubs by clicking on the publish button in RStudio. Once you’ve published your presentation, make sure to verify that the HTML file gets created in your directory. You can also host it on your computer and view it offline, which is useful if you’re working in an environment without internet access.

Hosting your presentation on GitHub can be technically challenging, but it’s worth trying since it’s a convenient way to publicly host your repositories. Once you’ve developed your project, you’ll create a pitch in R Markdown and hopefully host it on GitHub or another platform. This way, we can all view your project and appreciate how awesome it is.

That’s all the information about R Markdown we need for this class. Thank you for listening, and we’ll see you in the next lesson.

## 2.4 A few Ways to Share R Markdown Products

If you are dealing with a small group who have access to R you can simply share the R code and the R Markdown file or just RMarkdown rendered as HTML file and simply email it to a colleague. If they don’t have access to R, or you don’t wat to share your code you can share the HTML file. Furthermore, if you wish to share the HTML file with a wider audience, you can host it on a website. There are several options for doing this. We recommend the following websites for sharing the HTML files that you produce from R Markdown documents. If you’ve taken The Data Scientist’s Toolbox then you should be familiar with [GitHub Pages](https://pages.github.com). If you’re familiar with Git and you haven’t used it before their documentation for creating a site is very straightforward. GitHub also offers an anonymous option for hosting a single HTML file, called [Gist](https://gist.github.com).

If you’re not familiar with Git and you’re using RStudio you should take advantage of [RPubs](https://rpubs.com), which is perhaps the easiest way to share an HTML document from RStudio. You can find simple instructions for using RPubs [here](https://rpubs.com/about/getting-started). You can also take advantage of [ShinyApps.io](https://www.shinyapps.io) which is an alternative of RPubs for Shiny applications. If you’re looking for a site that will host any HTML file for free, including HTML files that are generated by R Markdown, you should check out [NeoCities](https://neocities.org). You’ll need to provide links to HTML files you’ve created with R Makrdown in upcoming peer assessments, so make sure to familiarize yourself with one of these sites. ## Plotly

### 2.4.1 Plotly 1.1

In this lecture, we will be discussing Plotly - a handy web application and R library. Although we had previously recorded a video on it during its early development for R, there have been a lot of developements since then. Plotly is a great tool for creating interactive web graphics in R, and it has the added benefit of integrating with multiple languages. Essentially, Plotly is a web application that enables the creation and sharing of visualizations. Plotly is a versatile web application and library for R that allows users to create interactive web graphics. It can work with several programming languages including MATLAB, Python, and JavaScript. In addition to its integration with different languages, Plotly offers a web interface for users who prefer to upload csv files and create plots with point-and-click interactions. Plotly is particularly easy to use with R and RStudio. To install Plotly, users can simply type “install.packages(”plotly”)“. To use it, they can load it with”library” or “require (”plotly”)“. Plotly plots can be shared through its web interface, which offers free and paid options for private displays and user authentication. Alternatively, users can create HTML graphics that can be shared on their own websites with the Plotly R package. The Plotly web interface offers tools for interactively changing axis labels, a feature that is not available in R. In this lecture, code demonstrations will be used to showcase different types of Plotly plots. It is worth noting that Plotly graphics can be embedded in R Markdown documents, including Slidify, R Studio Presenter, and Slidy documents.

### 2.4.2 Plotly 1.2

Here in the R markdown document associated with the Plotly lecture, I will be using the library(plotly) function, assuming that I have already installed it. The plot\_ly() command is straightforward: plot\_ly(mtcars, x = “weight”, y = “miles per gallon”, mode = “markers”). The mtcars dataset is stored as a dataframe and serves as the data source for this command. The x-axis represents weight, while the y-axis represents miles per gallon, and the mode is set to markers, creating a scatter plot. This code can be replicated for any dataframe to generate a scatter plot. The resulting scatter plot is interactive and can be viewed in the RStudio viewer pane. As you move your cursor over the points, their corresponding x and y values are displayed. In addition, there are various controls at the top of the viewer pane that enable you to download the plot as a PNG, adjust the scale, reset the axes, and modify the hover-over behavior, among other things. The Publish button allows you to publish the plot to RPubs with a single click, generating a web graphic. However, we’ll demonstrate other methods for publishing the plot shortly. Another easy way to publish the plot is to click on Export and choose to Save as Web Page. This creates a Web page with the plot and a file with the specified name, which can be referenced and embedded into other projects. Plotly is incredibly user-friendly and straightforward. Your first assignment is to create a simple scatter plot with a different dataset, save it as a webpage, and create graphics in PNG and PDF formats. You can also try publishing the plot to RPubs.

### 2.4.3 Plotly 1.3

Let’s explore some options for scatter plotting in plot\_ly, as scatter plots are a common type of visualization. One option is to change the color of the points. For example, we can use the “cyl” variable from the mtcars dataset as a factor variable for coloring the points. To do this, we convert the “cyl” variable to a factor using the function as.factor(cyl). Then, we plot the scatter plot using plot\_ly(mtcars, x = weight, y = mpg, color = as.factor(cyl), mode = “markers”). This will create a scatter plot where the points are colored based on the number of cylinders in the car, which is effectively a categorical variable. By default, the legend will display the three values of “cyl”: 4, 6, and 8.

Another option is to use a continuous variable for coloring the points. For example, we can use the “disp” variable from the mtcars dataset, which is a continuous variable. To do this, we plot the scatter plot using plot\_ly(mtcars, x = weight, y = mpg, color = disp, mode = “markers”). plot\_ly will recognize that “disp” is a continuous variable and use a continuous color gradient to display the points. The key to the gradient will be displayed on the side of the plot.

Let’s explore the option of changing the point size in scatter plots using plot\_ly. Currently, the points in the scatter plot appear small, almost invisible when outputted to an HTML file. We can change the size of the points using a continuous variable, in this case, horsepower. Each point will have a different size based on its horsepower. Color is already being used to represent the cylinder. By changing the size of the points, we can show four dimensions in the same plot: miles per gallon, weight, cylinders, and horsepower. However, since we’re limited to a 2D scatter plot, color and size are the only two dimensions we can display along with the x and y coordinates. Let’s look into a 3D scatterplot. We will run the code and display the plot, and then we will explain what it’s doing. This is another way to add a dimension to a scatter plot. The plot is created using web GL, which is a web-based version of the open GL graphics library. It allows you to embed interactive 3D graphics in webpages. So when you output it as a webpage, it will be interactive just like this. We have three variables in this example: temperature, pressure, and dtime. To create the dataset, we generated 100 random normal values for temperature and pressure, and dtime is just the numbers 1 to 100. We use the same “plot\_ly” command as before, specifying the x, y, and z variables and setting type to “scatter3d” and mode to “markers”. To add a color gradient to the plot, we use the “color = temp” parameter, which displays a key to the gradient on the side of the plot. This creates a nice interactive 3D scatter plot. For homework, I would like you to create a variety of 2D scatter plots using different plotting attributes, such as color and point shape, to visualize other dimensions. You can also create a 3D scatter plot and output it as a web page to become familiar with working with these types of plots. You may consider publishing your scatter plots on platforms like rpubs to have a public hosting. Give it a try and explore the possibilities. Next, we will discuss some even more exciting applications of plot\_ly.

### 2.4.4 Plotly 1.4

Let’s discuss a different type of chart, the line graph. To illustrate this, let’s use the “airmiles” dataset, which is a time series with values corresponding to each year. We can confirm this by typing “data(”airmiles”)” and then “time(airmiles)” to see the associated times.

To create the line graph using Plotly, we simply use the “plot\_ly” command with the “x” argument set to the time variable and the “y” argument set to “airmiles”. Since these variables are in our environment, we don’t need to specify a data frame. Once we plot the data, we get a time series chart with the added feature of showing the corresponding x and y values when the mouse hovers over the plot. This type of chart is often used to display market indices, such as the S&P 500. Let’s talk about creating a line graph using time series data. We can use the “airmiles” dataset as an example. It is a time series with values corresponding to each year. To create the line graph, we can use the “plot\_ly” command, with “x” as the time variable and “y” as the airmiles variable. The resulting plot will display the values over time, and when we hover over the plot, it will display the corresponding x and y values. A similar plot is often used to display market indices like the S&P 500.

Now, let’s create a multi-line graph using stock market data. We’ll need to use the “tidyr” and “dplyr” libraries for data manipulation. The “EuStockMarkets” dataset contains market indices from Germany, Switzerland, France, and Britain. To create the plot, we first convert the dataset into a data frame using the “as.data.frame” command. Then, we use the “gather” command to convert the data into a format suitable for plotting. The resulting data frame can be used with the “plot\_ly” command to create a multi-line graph. Note that the original data EuStockMarkets is not a data frame, as shown by the output of is.data.frame(EuStockMarkets). However, after converting it to a data frame using the command as.data.frame(EuStockMarkets) and then applying the gather command, we have transformed the data from a short format to a long format. In the long format, there is a variable called index which corresponds to the column name of the data points in the original data. Although this format may not be as convenient for display purposes, it is necessary for plot\_ly to display multiple lines, as it requires a single data frame with a factor variable that demarcates the different lines. Let’s take a look at how we can create a multi-line graph using stock market data. To begin with, we need to convert the data into a data frame. EuStockMarkets is not a data frame, as shown by the is.data.frame() function. Therefore, we first need to convert it into a data frame using the command “stocks <- as.data.frame(EuStockMarkets)”. We then use the gather() function from tidyr to convert the data from a short format to a long format, where each index type is now in a single column called “index”. The final step is to create the time variable using the command “stocks$time <- as.Date(rownames(stocks))”. Once the data is formatted correctly, we can create the plot using plot\_ly, where x is time, y is price, and color is the index type. The resulting plot will display the different index types in different colors and allow for interactive hovering over specific points to see the index value at that time. It’s worth noting that using tidyr and dplyr for data manipulation is essential and highly recommended. Give it a try using this data or some other stock market data, and see if you can create your own multi-line stock market graph to display on a webpage.

### 2.4.5 Plotly 1.5

The following examples are straightforward, so I won’t provide any code demonstrations. Instead, I’ll show you the resulting output. To create a histogram in plot\_ly, you only need to specify the x variable and set the type to “histogram.” The output will show the height of the bars as you move the mouse along. Additionally, you can customize the colors, create side-by-side histograms, and perform a variety of other actions in plot\_ly.

For a boxplot, you’ll need to provide the plot\_ly data frame (in this example, it’s the iris dataset), set the y variable to petal length, specify the color as species, and set the type to “box.” This will generate separate boxplots for each species. When you hover over the boxplot, it’ll display values such as the 25th percentile, 75th percentile, and median. The whiskers extend out to 1.5 times the distance between these two values, as per Tukey’s idea, which refers back to the normal distribution and the 1.5 times the inter-quartile range. If there’s an outlier, plot\_ly will display a little point to indicate it.

The following examples are relatively straightforward, so I won’t provide code demonstrations. For a histogram, simply specify type=“histogram” and the x variable. Here is an example:

[Output shown]

You can change the colors, display side-by-side histograms, and perform other modifications using plot\_ly. The process is intuitive. For a box plot, use the plot\_ly data frame, specify y=petal length, color=species, and type=“box”. The result will be separate box plots for each species. Here is an example:

[Output shown]

Hovering over the box provides information about the box plot, such as the median, 25th and 75th percentiles, and whiskers that extend 1.5 times the distance between these values. A heat map is essentially a graphical representation of an image. To create one using plot\_ly, generate a matrix with random normal values and specify z=matrix, type=“heatmap”. The result will be an interactive heat map with a color key on the right-hand side. As you move your mouse around, plot\_ly will display x and y values and the corresponding intensity values. We can create a similar dataset as before, but with different random normal values. Next, instead of plotting the intensity values as colors, we can plot them in a third dimension as a surface. To do this, we set the type as “surface” and specify the z values as the intensity values. The row and column values are assigned integers for the x and y values. The resulting plot is a 3D surface. It’s worth noting that when plotting a 3D surface, there is some smoothing involved, so it’s important to understand what this smoothing is doing if you plan on using this for real.

### 2.4.6 Plotly 1.6

Okay, let’s create a map with some interactive hover-over effects. The code in this example is a bit long, so I won’t go over every detail. First, let’s create the dataset and then display it. Here we show you the first few rows of this dataset. As you can see, there are 50 rows, each representing a state. For example, the first row is for Alabama and includes its population in millions and some hover-over text. The hover-over text includes the state abbreviation, “AL”, a line break, and the population in 1975, which was 3,615,000. We can also add red borders as an option, although it’s not necessary. Additionally, the map options will be a list and we will set the scope to the USA. You can experiment with different options to see what works best. To dramatically alter this map for a different region or country, you may have to improvise a bit, but many of the options are self-explanatory. The “showlakes = TRUE” option simply means that lakes will be displayed, and in this case, the lake color was set to white to avoid blending in with the blue population color. To break down the code we start by selecting the first couple of rows of the dataset and displaying them to get an idea of what it looks like. It contains the population of each state in millions and some hover-over text which includes the state abbreviation and population in 1975. Next, we set some map options such as showing the lakes, setting the lake color to white, and selecting the USA as the scope. Then we use the plot\_ly command to create the choropleth map. We use the state\_pop dataframe, set the population column as the color variable, and use the state column for locations. We set the type to choropleth and location mode to USA-states. We use a blue color palette and specify that we want to draw markers on the borders. Finally, we pass the output of plot\_ly to layout, which sets a title and includes the map options we previously set.

We can go ahead and run this code now. In our case, it was a bit complicated and the map wouldn’t display in the viewer. So, we saved it as a web page and overwrote the template web page that we’ve been using for this example. Now, when if we hover over a state, we can see its population. For example, when we can hover over Alabama and see its population of 3,615,000 in 1975.

If you want to create your own map, start by using this data frame, which has more columns than just population, and try creating different maps with various options. As you experiment with different options, you’ll get a better understanding of how they work. To begin with, you can replicate the map we just created and then attempt to create another map using different variables, where the color represents something else. This can be a helpful technique to create interactive maps.

### 2.4.7 Plotly 1.7

If you’re a fan of ggplot, then you might like this code from the Plotly website. I’m using the diamond dataset to create a ggplot of carat vs. price. What’s cool is that Plotly has a function called ggplotly that allows you to easily convert a ggplot into an interactive HTML graphic. The top window shows the ggplot in the RStudio plotting pane, while the bottom window displays it as an HTML graphic in the RStudio viewer pane, just like it would appear in a web browser.

### 2.4.8 Plotly 1.8

To directly post a Plotly graph we will follow the previous video. We created a GG plot example, labeled GG, which can be displayed in the viewer pane by typing GG in the R prompt. However, if we type plotly\_POST GG, it will post it directly to the Plotly website. You can then use their web-based graphical user interface to edit the graph by clicking on the edit option. Plotly has many tutorials and videos to help you understand how to use it. Before using Plotly, you must set two environment variables - plotly username and plotly API key. You can find your API key by going to the settings in Plotly and copying and pasting it from there. If you set these two environment variables in your .Rprofile, you won’t be prompted for your credentials every time you start R, making the process seamless.

## 2.5 Leaflet

### 2.5.1 Leaflet 1.1

Creating interactive maps is crucial in developing data products. One widely used JavaScript library for this is Leaflet, which also has an associated R package that enables the creation of interactive maps in the R environment. Using Leaflet within RStudio is especially convenient as it opens up in the RStudio window. While there are other ways to create interactive maps, such as GoogleVis, Leaflet is popular among the R community. However, you may also want to explore creating maps using Google Maps in the GoogleVis lecture. To get started with Leaflet, you should first install the package by running the command install.packages(“leaflet”). Let’s now look at a code example that demonstrates how to create your first Leaflet map in R. The R Markdown document for the following slides contains two sections - one that displays the code and another that evaluates it. To create the map, first load the library by running library(leaflet), and then execute the three commands provided. You can then zoom in as much as you like on the resulting map. In order to develop data products, creating interactive maps is an important aspect. The Leaflet package for R makes use of the Leaflet JavaScript library to create such maps within the R environment, and is particularly convenient to use within RStudio. We will begin by exploring some Leaflet examples to get you started, and it won’t take long for you to become familiar with it. While there are several other ways to create interactive maps, such as using GoogleVis, Leaflet is widely accepted and widely used within the R community. To get started, you need to install the package by running install.packages(“leaflet”). The code for creating your first Leaflet map in R is shown in the R markdown document for the next few slides, and it includes piping notation which we will use heavily in these lectures. The first command, leaflet(), generates the map, while the addTiles() command adds the first set of content. We will discuss how to add more useful features, such as markers, later on. For now, try generating a world map and zooming around to ensure that you have successfully installed the Leaflet library.

### 2.5.2 Leaflet 1.2

We were able to quickly generate a JavaScript map widget without any knowledge of JavaScript using the Leaflet function, which creates a background layer. The addTiles function adds mapping data from Open Street Map to the background. After adding this content, we can zoom in and explore the map. However, to add markers and direct interactivity, we need to use the addMarkers function, which is very easy to use. We can specify the longitude and latitude of a location and add a label to the marker. In the example, we added a marker for Johns Hopkins hospital with the label “Jeff Leek’s office”. The piping notation was used, and although it can be a little strange at first, it’s a cleaner and easier way to read the code. We can click on the marker to see Jeff Leek’s office and zoom in to see the Bloomberg School of Public Health. To recap, we were able to generate a map without knowing any JavaScript by using the leaflet function, which creates a background for the map. The addTiles function adds mapping data from OpenStreetMap to the background, and this allows us to zoom in and out and look around. However, to add specific content, like markers, we need to use additional functions. The addMarkers function is used to add markers to the map at specific longitude and latitude points, and we can also give them labels.

Now, let’s go through the code. To explain the piping notation, I will show you the equivalent code. First, let’s reset my\_map to its initial state with no markers. Then, to add the marker, we use the addMarkers function and pipe the output to my\_map using the piping notation. This is equivalent to writing my\_map = addMarkers(my\_map, …), and the code we used adds a marker at a specific location with a label. I will copy this code, as it is easier than retyping it.

Additionally, it is worth noting that the Bloomberg School of Public Health is the oldest School of Public Health in the country. Okay, let’s break down what’s happening here. The lines of code that we just wrote to add a marker are equivalent to this one line of code. We’re using the pipe operator to pass my\_map as the first argument to the addMarkers() function. Then, we assign the result back to my\_map.

Now, you might be wondering why we’re doing all of this when we could have just kept the one-liner. Well, the reason is that piping is a powerful tool that you’ll want to get used to, especially when you’re doing chained operations in tools like dplyr or in mapping contexts like this. When you’re adding lots of layers and markers, it can become tedious to keep retyping the same set of commands. The first line is only necessary because we separated it from the set of lines up above. Let me show you another way of writing this code, which is what you would likely do. Instead of using the piping operator, you would assign my\_map to the initial map and then add the marker using addMarkers(). This set of commands is equivalent to the previous one. The reason why you might prefer the chaining approach is that it simplifies the code and avoids having to reassign the variable every time you add a new layer or marker. This is particularly useful when working with complex maps with lots of layers and markers. From now on, we will use the chaining approach to demonstrate Leaflet commands.

To add a marker, you just need to provide the addMarkers() function with the my\_map variable as the first argument, followed by the longitude and latitude coordinates, and the text for the popup. For example, in the code we just saw, the longitude and latitude coordinates are for Jeff Leek’s office and the popup text says “Jeff Leek’s office”. If you have a map with many markers, you can use this function to add each one with its own set of coordinates and popup text. We will cover more examples in the following sections.

### 2.5.3 Leaflet 1.3

To add a lot of markers, you can simply put the collection of markers in a data frame. Here’s an example. First, let’s set the seed so that we get the same results every time we run the code. Then, we’ll create a data frame with some random longitude and latitudes. The data frame has 20 rows, and the columns are named “lat” and “lng”. We’ll pass this data frame as an argument to leaflet to create our map, which will initially be blank. If we run the code up to this point, we’ll get a blank map. Next, we’ll pass the output of the leaflet argument evaluated at the data frame to the addTiles function. This will add the mapping data to the map, but won’t plot anything yet. Finally, we’ll pass the map with the mapping data to the addMarkers function to add the markers. You can run the entire code at once instead of in pieces. The resulting map will have multiple markers. Give it a try by creating your own data frame with random longitude and latitudes and adding it to a map with markers.

### 2.5.4 Leaflet 1.4

In the following code, we will cover two topics: adding custom markers and separate popups for each marker. We want to achieve this result, where each marker displays the Hopkins logo and has a separate popup. Here is an example of what we’re aiming for. We’ll first define our Hopkins icon by specifying its URL, width, height, and anchor points. Then we’ll create a series of latitude and longitude values for buildings at Hopkins. We’ll pass these values to leaflet, which will generate a map with markers using the addTiles and addMarkers functions. However, we also need to set the icon for each marker to the Hopkins icon. Once we have the markers displaying the Hopkins logo, we want to add links to each marker. We’ll create a series of sites, which are text vectors containing HTML commands that create a hyperlink for each building. These sites correspond to the latitude and longitude values in the same order. Finally, we’ll pass the latitude and longitude values to leaflet, addTiles, and addMarkers functions, along with the Hopkins icon. We’ll set the popup to display the collection of hopkinsSites. Once we run the entire code, we will have a map with Hopkins logos at each marker, and separate popups with links for each building.

### 2.5.5 Leaflet 1.5

Leaflet offers a great solution for situations when multiple points appear too close together to differentiate. This solution is clustering, where the points are grouped together and will break apart into individual points when the map is zoomed in. To demonstrate this feature, we can define a data frame with 500 latitude and longitude points that are close together and pass it as an argument to leaflet. Then, we can add tiles and markers with default cluster options. The clusters will show the number of points they contain. As we zoom in, the clusters break apart into individual points. Additionally, we can add circle markers instead of standard pop-ups or clusters by using the add circle markers command. We can create a couple of 20 markers and add circle markers to them. This will display the markers as little circles.

### 2.5.6 Leaflet 1.6

Now, let’s discuss how to digitally draw shapes on your map. This can be accomplished using commands that allow you to draw circles and rectangles, and even add a legend to your map. These commands will enable you to annotate and draw on your map in useful ways. We will demonstrate how to draw a circle in a way that you might actually want to use. For example, we have a dataset of cities such as Baltimore, Frederick, Rockville, and Gaithersburg, along with their population, latitude, and longitude. We will pass this dataset to leaflet, add tiles to it, and then add circles to it using the addCircles command. We will also provide some important options such as scaling the radius relative to the population of the cities. This is a more reasonable use of adding circles than just having them as markers. Let’s go ahead and run the code to see the result. Alright, now we have the circles representing the population of the cities and we can see the relative sizes of the circles. Let’s move on to drawing rectangles. We can use the addRectangles command to do that. Here’s an example where we’re going to draw a rectangle around Baltimore city. So we’ll define a bounding box that contains Baltimore and then use that to draw the rectangle. Finally, we’ll add a legend to the map to explain what the circles and the rectangle represent. So let’s get started.

We’ll define a bounding box that contains Baltimore using the leaflet boundingBox function. Then we’ll pass that to the addRectangle function along with some options to define the color and opacity of the rectangle. Finally, we’ll add a legend to the map using the addLegend function. This will display a key explaining what the circles and rectangle represent. So let’s run the code and see the final map. Let’s move on to adding a rectangle to the map to show you what it looks like. We’ll use the same process as before, starting with leaflet, adding tiles, and then adding the rectangle. For a rectangle, you just need to provide the coordinates of its two opposite corners. In this example, we’re creating a rectangle around the Mountain View Corporate Center. Next, let’s cover how to add a legend to the map. This is a common task when you want to differentiate between different annotations, such as circles and rectangles, using different colors and symbols, and want to provide a key for users to understand what each one means. Here, we have a data frame containing 20 points. Let’s try creating a rectangle to demonstrate how to add it to the map. First, we’ll add tiles to the Leaflet map. Then, we’ll add a rectangle using the coordinates of its two opposite corners. Here’s the rectangle that we created, which happens to be the Mountain View Corporate Center. Finally, we’ll cover how to add a legend. When you add markers to your map, you may want to color them differently and add a legend to help the user understand what each color represents. In this example, we have a data frame with 20 points, colored red, blue, or green. We pass the data frame to Leaflet and add circle markers colored according to the random color we generated. Of course, in a real application, you would want to use colors that have a specific meaning. To add a legend, we use the same syntax as the legend command in base R plotting. We define labels, in this case just LETTERS[1:3], and colors, which are blue, red, and green. When we run the code, the legend appears at the top of the map. This is the last lecture on Leaflet. We hope you found it useful. Leaflet is a powerful tool, and when used in conjunction with Shiny, it can create amazing applications. It’s also one of the easiest interactive plotting utilities available in R, and definitely worth learning. Thanks for joining us, and we look forward to seeing you in the next lab.

## 2.6 Quiz

## 2.7 Assignment

# 3 Third week

In this module, we’ll deal with the world of creating R packages and practice developing an R Markdown presentation that includes a data visualization built using Plotly.

## 3.1 R Packages

An R Package is a form of data product. In creating an R package, you will create polished, documented software for your users. This is a big step over distributing functions or code that perform a task. In this brief set of lectures, we cover the basics of creating R packages.

### 3.1.1 R Packages part 1

In this lecture we will be discussing how to build R packages, which is an essential tool for implementing your own statistical or other methodologies and distributing them to other R users through centralized platforms and repositories. We will cover the basic aspects and components involved in building an R package, so that you have a clear understanding of the process. R packages are a means of extending the basic functionality of R. While the base R distribution includes packages like graphics and GR devices, there are countless other types of functionality that are not included. The package system enables developers to write their own functions and other types of functionality that can be added on to the base installation. An R package typically consists of a collection of R functions and data objects, along with documentation and other elements that are organized in a systematic way. This allows users to access help files and other resources in a consistent manner. To find R packages you might visit a centralized repository like the Comprehensive R Archive Network or CRAN. Another major repository is called Bioconductor, which houses numerous tools for bioinformatics and genomics. Alternatively, many packages are hosted on source code sharing websites like GitHub, Bitbucket, and Gitorious. Installing packages from CRAN or Bioconductor is simple with the install.packages() function, where you specify the package name and repository. If you’re installing packages from GitHub, you can use the install\_github() function from the devtools package. Putting your package on a central repository to distribute makes it easy for people to find and install your package.

What is the purpose of creating an R package and why is it advantageous over just sending a code file? There are several benefits to creating an R package. Firstly, it provides a structured format that other R users can understand, especially if they are already familiar with other R packages. Secondly, documentation is required for an R package, which forces the creator to write documentation that explains how the functions in the package should be used. This includes documenting the arguments and return values. Thirdly, creating an R package allows the creator to define a well-defined API, which specifies what functions the users should call and what functions they should not call. This allows the creator to hide the implementation details from the user and to change the details in the future without disrupting the public interface. Fourthly, maintaining an R package is typically easier because of the structure and documentation required. If the creator cannot or does not want to maintain the package anymore, they can pass it off to someone else, and the structure is already in place for the new maintainer. Finally, there are some standards for reliability and robustness associated with packages hosted on central repositories like CRAN, which can help users feel confident that the package will load properly and not cause problems on their system.

The general process for developing an R package typically involves several steps. Initially, you write the R code in a script or .R file, testing and debugging it as necessary. Eventually, you may decide that the code is useful and should be made available to others. At this point, you’ll incorporate the R code into the R package structure. The next step involves writing documentation for the user functions, which are the functions accessible to the public. Additionally, you may include other materials such as examples, demos, datasets, tutorials, or vignettes. Once everything is complete, the package can be compiled and packaged, and you’ll have your R package ready for distribution. After creating an R package, you have the option to keep it for personal use, email it to a friend, or upload it to a repository such as CRAN or Bioconductor. You can also share the source code on sites like GitHub for others to view and modify. Once you make the package public, two scenarios may arise. First, users may encounter issues and notify you to fix them. Second, users may identify problems and offer solutions, which you can incorporate into your package. Having other users review to improve your code is beneficial. Once you incorporate these changes, you can release a new version of the package.

### 3.1.2 R Packages part 2

An R package consists of several essential elements and optional elements. To create an R package, the first step is to create a directory on the file system. The directory is typically named after the package name, such as “my package”. The next essential element is a description file, which contains metadata about the package include the name of the package, a title, a longer description, version number, and the name of the author. The version number typically follows a format of major number dot minor number dash patch level. The package also needs an R sub-directory containing R code and a man/sub-directory with documentation for the R code. The author may not necessarily be the person who created the package code, but it could be you. Additional optional elements include a NAMESPACE file, which is commonly used and highly recommended. For a full list of requirements, you can refer to the [Writing R Extensions](https://cran.r-project.org/doc/manuals/R-exts.html) document available on the R website. The maintainer plays a crucial role in the package’s maintenance and must include their name and email address. They are responsible for addressing any issues that arise with the package. Another essential element is the license, which outlines the terms under which the source code is released. Common licenses for R code include the GNU public general license, BSD license, and MIT license. Other optional fields in the description file are “Depends,” which lists the R packages that the package relies on, and “Suggests,” which lists non-essential packages. The release date and URL for the package’s home page can also be included. Additionally, you can add other fields for personal use, but they will be ignored by R.

Here is an example of a description file from the gpclib package.

Package: gpclib  
Title: General Polygon Clipping Library for R  
Description: General polygon clipping routines for R based on Alan Murta's C library.  
Version: 1.6-1  
Author: Roger D. Peng <rpeng@jhsph.edu> with contributions from Duncan Murdoch and Barry Rowlingson; GPC library by Alan Murta  
Maintainer: Roger D. Peng <rpeng@jhsph.edu>  
License: MIT + file LICENSE  
Depends: R (>= 3.0.0), methods  
Imports: graphics  
URL: https://github.com/rickbrew/GeneralPolygonClipper, https://github.com/rdpeng/gpclib

The package name is gpclib, and the title is “General Polygon Clipping Library for R”. The description field provides a brief explanation of what the package does, which is to provide general polygon clipping routines for R based on the C library by Alan Murta. The version number is 1.6-1. The author is the same as the maintainer and provides an email address for contact. The package has a special license, which is explained in the license files. It depends on R version 3.0.0 or later, as well as the methods package and the graphics package for importing functions. The URL for the website of the underlying C library and the GitHub repository is also provided. After writing the description file, the next step is to copy your R code into the R sub-directory. The number of files in this directory can vary and there is no need to have one file per function. However, it is recommended to organize the files into logical groups of functions, such as those for reading data or fitting models. All R code should be included in this directory and should not be included in any other part of the package.

The NAMESPACE file is important as it defines the API for your package and lists all the dependencies, including code from other packages. There are two types of things to indicate in this file: exports and imports. Exports are the functions that are available to the user and are considered public functions. Non-exported functions are not listed in this file and cannot be called directly by the user, unless they use some special functions to access the code. This allows you to hide implementation details from users and create a cleaner package interface. Imports allow your package to use functions from other packages without making those packages visible to the user. This is useful for avoiding loading unnecessary packages onto the search list and allows you to use the functionality of other packages without making them directly available to the user. Importing a function loads the package NAMESPACE but does not attach it to the search list. There are a few key directives for the NAMESPACE file, including export, which allows you to export a function, import, which is used for importing a package, and import from, which is used for importing a specific function from a package. These directives allow you to describe your package’s dependencies and public API in a specific and clear way. Here is the NAMESPACE file for the gpclib package.

useDynLib(gpclib, .registration = TRUE, .fixes = "C\_")  
  
importFrom(graphics, plot)  
import(methods)  
  
exportClasses("gpc.poly", "gpc.poly.nohole")  
  
exportMethods("show", "get.bbox", "plot", "intersect", "union",  
 "setdiff", "[", "append.poly", "scale.poly", "area.poly",  
 "get.pts", "coerce", "tristrip", "triangulate")  
  
export("read.polyfile", "write.polyfile")

If you’re using s4 classes and methods, the export classes and export methods directives are important. It’s worth noting that while these directives look like R functions, they’re not exactly the same. Here’s an example of a NAMESPACE file from the mvtspolt package, which only has one function, mvtsplot. This function is the only one that users can call, and it imports the access function from the graphics package and all the functionality of the splines package.

# Generated by roxygen2: do not edit by hand  
  
export(mvtsplot)  
importFrom(RColorBrewer,brewer.pal)  
importFrom(grDevices,colorRampPalette)  
importFrom(grDevices,gray)  
importFrom(graphics,Axis)  
importFrom(graphics,abline)  
importFrom(graphics,axis)  
importFrom(graphics,box)  
importFrom(graphics,image)  
importFrom(graphics,layout)  
importFrom(graphics,lines)  
importFrom(graphics,par)  
importFrom(graphics,plot)  
importFrom(graphics,points)  
importFrom(graphics,segments)  
importFrom(graphics,strwidth)  
importFrom(graphics,text)  
importFrom(splines,ns)  
importFrom(stats,complete.cases)  
importFrom(stats,lm)  
importFrom(stats,na.exclude)  
importFrom(stats,predict)  
importFrom(stats,quantile)

After creating the necessary files, documentation is crucial to help users understand how to use your function. Documentation files typically have a .Rd extension. The package documentation is important for users to understand how to use functions properly. Documentation files are written in a specific markup language and must be located in the “man” subdirectory of the package directory structure. A documentation file is required for every exported function, which means limiting the number of exported functions is advisable to reduce the workload of documentation. Documentation can also cover data sets, basic concepts, or provide a package overview.

For instance, the base R Line function has a simple help file that illustrates the components of a typical documentation file.

\name{line}  
\alias{line}  
\alias{residuals.tukeyline}  
\title{Robust Line Fitting}  
\description{  
 Fit a line robustly as recommended in \emph{Exploratory Data Analysis}.  
}  
\usage{  
line(x, y)  
}  
\arguments{  
 \item{x, y}{the arguments can be any way of specifying x-y pairs. See  
 \code{\link{xy.coords}}.}  
}  
\details{  
 Cases with missing values are omitted.  
 Long vectors are not supported.  
}  
\value{  
 An object of class \code{"tukeyline"}.  
 Methods are available for the generic functions \code{coef},  
 \code{residuals}, \code{fitted}, and \code{print}.  
}  
\references{  
 Tukey, J. W. (1977).  
 \emph{Exploratory Data Analysis},  
 Reading Massachusetts: Addison-Wesley.  
}

The file contains a name and an alias to call up the documentation file. A title and description provide context for the function, followed by a usage section that shows how the function is called with its arguments. Arguments need to be described; for example, the Line function only has two arguments, x and y, which can be specified in various ways. After describing the arguments, the detailed section typically goes into more depth about various aspects of the function. In this example, there are only a few additional notes. Lastly, there is a value section which describes what is returned by the function. Some functions do not necessarily return anything useful, particularly plotting functions which often just have the side effect of creating a plot. In this case, the line function returns an object of class tukeyline. Finally, at the end of the help file, there may be references. For example, if you are implementing a statistical method and it has a reference such as a paper or book, you can include those references in the help file so that people know where to look if they want to explore the topic further.

Once you have your R code, R files, description, and NAMESPACE, you can begin building the package. There is a command line tool that comes with R called R CMD build which creates the package archive file with a .tar or .gz extension. Once you have built the package, you can run a series of tests on the package using the R CMD check command line program. This runs a set of tests to ensure that everything is consistent within the package structure, including documentation, exports, and imports. You can run R CMD build and R CMD check from the command line using a terminal or command shell type of application, or you can run them within R using the system function.

system("R CMD build newpackage")   
system("R CMD check newpackage")

As previously mentioned, **R Cmd check** performs various tests to ensure that your package is consistent and error-free. It checks for the existence of documentation, ensures that every exported function has documentation, and verifies that your code can be loaded. Additionally, it checks for common coding issues using a code checker in one of the tests. If your documentation contains examples, R Cmd check runs them to verify that they run properly. It also checks for mismatches between the argument list in your documentation and the code itself. Passing all of the tests in R Cmd check without errors or warnings is crucial if you intend to submit your package to CRAN.

To get started with creating an R package, you can use the package.skeleton() function. This function generates a directory structure, including the R directory, man directory, description file, and NAMESPACE file, based on the name of the function you pass as the first argument. It also creates documentation files in the man directory for any functions visible in your workspace. If you only want to include specific functions, ensure that they are the only functions in your workspace before running package.skeleton(). Once the package skeleton is created, you need to edit the documentation stubs in the man directory to customize them to your needs. Additionally, you must fill in all of the required information in the description file and namespace file.

In summary, R packages provide a standardized and systematic way to share R code with others. They are important for ensuring that packages have proper documentation and robustness. These packages can be obtained from repositories such as CRAN, Bioconductor, or GitHub. To create an R package, the basic checklist involves: - Creating a new directory - Adding R and man sub-directories (which can be done using package.skeleton()) - Writing a description file - Adding R code to the R sub-directory - Creating documentation files in the man sub-directory - Defining exports and imports in a NAMESPACE file - Building and checking the package to ensure it passes all tests. -  
This is the basic template for creating an R package.

### 3.1.3 Building R Packages Demo

In this lecture, we will demonstrate how to build an R package using R Studio. We will build a small package with only two functions that will create a prediction model for a high-dimensional dataset using the top ten predictors. The package will be called the Top Ten package. Let’s get started with building our R package in R Studio. The first thing we need to do is create a new project by clicking on “New Project” in the “Project” menu. We’ll select “New Directory” and then choose “R Package” as the project type. We’ll name our package “topten” and select our home directory as the subdirectory for the package files. After hitting “Create Project,” we’ll see that the package files have been created in the directory. The package files include a description file, a namespace file, and an R directory with a code file that’s currently empty. We’ll need to fill in the details for the description file, including the package title, version, date, author, maintainer, and description. Open the DESCRIPTION file and for title you can use “Building a Prediction Model from the Top 10 Features” with version 0.1.0. For author you should put your name, for maintainer you should put your name and more importantly provide your email address, for description you can put something like “Functions for building and prediction models from selecting the top 10 predictors in a dataset.”, and for license you can use one of the open source licenses like “GPL-3”.

Now start a new R script named “topten.R” and write your function.

# topten.R  
topten <- function(x, y) {  
 p <- ncol(x)  
 if(p < 10)  
 stop("there are less than 10 predictors")  
 pvalues <- numeric(p)  
 for(i in seq\_len(p)) {  
 fit <- lm(y ~ X[, i])  
 summ <- summary(fit)  
 pvalues [i] <- summ$coefficients [2, 4]  
 ord <- order(pvalues)  
 ord <- ord[1:10]  
 x10 <- x[, ord] fit <- Im(y ~ ×10) coef (fit)  
 }  
 predict10 <- function(X,b){  
 X<- cbind(1,X)  
 drop(X%\*%b)  
 }  
}

We begin by building a function called topten(x,y), which takes two arguments: x and y. The x argument represents the matrix of predictors, while y represents the vector of responses. We check to see if there are at least ten predictors by verifying if the number of columns, p, is greater than or equal to ten. If p is less than ten, we stop. Our prediction model works by fitting a univariate regression model for each predictor in the matrix of predictors, using the response vector y. We calculate the p-value associated with each predictor, sort them from smallest to largest, and choose the top ten predictors with the smallest p-values. We then use these predictors to build a final regression model.

To achieve this, we start by initializing a vector of p-values, which we set to be an empty vector of zeros. We then loop through each predictor and fit a univariate regression model with lm(). We extract the p-values using summary() and store them in our pvalues vector. After fitting the univariate regression models for each predictor, we will accumulate all the p-values for each model, which can be obtained by extracting the fourth column from the summary of the model fit. Then, we will sort the p-values in ascending order to get the indices of the smallest to the largest p-values. From these indices, we will select the top ten predictors and create a new dataset called x10 with only those predictors. This dataset will be used to fit the final linear model, from which we will extract the coefficients using the coefficients() function. The top ten function returns these coefficients. To make predictions using the coefficients and new data, we have the function called predict10. This function takes a matrix of predictors, which can have no more than ten columns, and a vector of coefficients called b. It returns the predicted response for each row in the predictor matrix. First, we need to add the intercepts to the predictor matrix. Then, we perform a matrix multiplication of the updated predictor matrix and the coefficients from the final fitted model. We will call this matrix multiplication “x times b”. Next, we need to drop the dimension of the resulting matrix to obtain a vector of predicted responses, which we will call drop. Our package consists of two functions, but to create a package, we also need to include documentation and specify the namespace file. We will use the **Roxygen2** package for documentation, which allows us to put the documentation in the code file itself. This has two advantages: it keeps us focused on one file and makes it easier to keep the documentation up to date since it is physically close to the code.

Let’s start documenting the top ten function using Roxygen2 package. First, we need to give it a title and a brief description of what it does. We’ll call it “Build a Model with Top Ten Features”, and the function develops a prediction algorithm based on the top ten features in x that are most predictive of y. Next, we’ll document the function arguments. The first argument is x, which is an n by p matrix of n observations and p predictors. The second argument is y, which is a vector of length n representing the response. The function returns a vector of coefficients from the final fitted model with top ten features.

#' Building a Model with Top Ten Features  
#'  
#' This function develops a prediction algorithm based on the top 10 features  
#' in 'x' that are most predictive of 'y'  
#'  
#' @param × a n × p matrix of n observations and p predictors  
#' @param y a vector of length n representing the response  
#' @return a vector of coefficients from the final fitted model with top 10 features  
#' @author Roger Peng  
#' @details  
#' This function runs a univariate regression of y on each predictor in × and  
#'calculates a p-value indicating the significance of the association. The  
# final set of 10 predictors is taken from the 10 smallest p-values.  
#' @seealso \code{lm}  
#' @export  
#' @importFrom stats lm  
  
topten <- function(x, y) {  
 p <- ncol(x)  
 if(p < 10)  
 stop("there are less than 10 predictors")  
 pvalues <- numeric(p)  
 for(i in seq\_len(p)) {  
 fit <- lm(y ~ X[, i])  
 summ <- summary(fit)  
 pvalues [i] <- summ$coefficients [2, 4]  
 }  
 ord <- order(pvalues)  
 ord <- ord[1:10]  
 x10 <- x[, ord]  
 fit <- lm(y ~ x10)  
 coef (fit)  
}  
  
#' Prediction with Top Ten Features  
#  
#' This function takes a set of coefficients produced by the \code{topten}  
#' function and makes a prediction for each of the values provided in the  
#' input 'X' matrix.  
#'  
#' @param X a n × 10 matrix containing n new observations  
#' @param b a vector of coefficients obtained from the \code{topten} function  
#' @return a numeric vector containing the predicted values  
#' @export!  
  
 predict10 <- function(X,b){  
 X<- cbind(1,X)  
 drop(X%\*%b)  
 }

Note that both functions need to be exported, so we’ll add an export directive for each. We also need to import the lm() function from the stats package in order to use it.

Now we can start building our package in R Studio. We’ll go to the build tab and build the package, then load it into R. You’ll notice that the documentation files haven’t been created by Roxygen2, so we’ll need to configure our build tools to generate them. So, the next step is to generate the documentation using Roxygen and create the namespace file from the documentation we wrote. We can do this by configuring the build tools. The package documentation file on the left is not required, so we can delete it. After building and reloading the R package again, we can see that the two documentation files for our code have been written. The topten.Rd file contains all the information extracted from the documentation we wrote in the R file, and there’s another set of documentation for the predict10.Rd. Let’s examine our **topten** package. We can load it using the library function and view its documentation using the help function with the package name as an argument. The description file we wrote is displayed, along with the exported functions. We can print out the code for the topten() function and view its documentation using the question mark followed by the function name. Similarly, we can view the documentation for the predict ten function. The R package is almost complete. We can check if it passes R command check by clicking the **check button** in the build tab. The tests run and if everything is okay, we will pass the test. We don’t need to upload this package to command, but if we wanted to, passing R command check would be a good indication that it is ready for upload.

We have now created a basic R package by writing two functions, but as you can see, R Studio provides a lot of useful tools for package creation and documentation generation. We encourage you to give it a try and create your own package using the functions you have written in your previous assignments. With the help of R Studio’s tools, the process can be completed quickly and easily.

### 3.1.4 R Classes and Methods Part 1

The focus of this lecture is on the creation of classes and methods in R. The main objective of this process is to introduce new types of data or data structures that are not natively supported by R. For instance, while R has built-in functionalities for lists, it may not be equipped to handle new types of data that may emerge, such as data from a new field. In such cases, a new structure needs to be developed to manage this data, along with a new set of functions to operate on it. This is where classes and methods come in, enabling the R system to adapt and support various data types and structures. Therefore, this lecture will delve into the details of writing classes and methods to facilitate the process of supporting diverse data and data structures. The classes and methods system in R is essentially an object-oriented programming system. What’s unique about R is that it was both an interactive system and supported a formal system for object orientation when it was first introduced. In contrast, other well-known object-oriented programming languages such as C++ and Java were not considered interactive languages. Instead, they were languages where you write programs, compile them, and then run them, rather than typing commands and executing them one at a time as in R. Much of the code for developing classes and methods in R was written by John Chambers, the creator of S and a prolific R programmer. The fundamental concepts are explained in his book [Programming with Data](https://link.springer.com/book/9780387985039).

The concept behind the class and methods system in R is to facilitate the transition from being a user to becoming a programmer. As we discussed in the R programming class, John Chambers initially designed the S language with the idea that users would gradually become programmers as they became more familiar with the language and attempted to customize it. This philosophy is reflected in R, where users can initially execute functions, and as their needs expand beyond what R offers, they can become programmers and create new things in R using classes and methods.

In R, there are two main styles of classes and methods, which are a result of its historical development. The older style is referred to as S3 classes and methods, which are informal, a bit clunky, and not very formally rigorous but they get the job done and work well. In fact, many basic functions in R are built on S3 classes and methods, making them quite useful and not going away anytime soon. The more formal and rigorous system for writing classes and methods is called the S4 system, which was introduced in R version 1.4.0 in December 2001. S4 classes and methods are sometimes called new-styled classes and methods and are encouraged for new projects where developers are creating new types of classes and methods. So, both the S3 and S4 systems will coexist in R for the foreseeable future due to the language’s history and built-in functionality. However, they are separate systems, and it’s not typical to mix them together when developing new projects. The S4 system is generally encouraged for new development, especially in projects like Bioconductor that use it extensively. While S3 classes and methods are sometimes useful for quick and dirty projects, we’ll focus on S4 development in this discussion. The implementation of object-oriented methods in the S4 classes and methods is done through the methods package, which is usually loaded by default in R, if not, you can load it using the library() function and calling library(<methods>). It’s worth noting that we will touch on the S3 system to familiarize you with it, but we will mostly discuss developing in the S4 system.

In object-oriented programming with R, we have classes and methods. A class is essentially a description of a new thing, such as a new data type or a new idea. You can define a class using the setClass() function in the methods package. Objects are instances of these classes and can be created using the new() function or other types of functions. A method is a function that operates on a specific class of objects. It’s important to note that methods are functions that are specific to certain classes. Finally, there are generic functions, which are R functions that dispatch methods. Generic functions encapsulate a generic concept and can dispatch methods depending on the data type. For example, the plot() function is a generic function that can dispatch different plotting methods based on the data type. Generic functions don’t perform any computation, but instead, their main job is to dispatch methods. A method is the implementation of a generic function for an object of a particular class. So, the main concepts in object-oriented programming in R are classes, objects, methods, and generic functions. As you progress in learning the system, you may find the help files in the methods package to be useful. The primary documentation is contained in the help files, which include ?Classes and ?Methods help pages. Additionally, you can refer to the help pages for ?setClass, ?setMethod, and . Some of this material can be quite technical, so you may want to skim through it initially. However, as you become more familiar with the system, it will make more sense.

Every object in R has a class, which can be determined using the class function. For example, if we take the number 1, it has a class of “numeric” by default since numbers in R are typically represented as real or double-precision numbers.

class(1)

## [1] "numeric"

Logical values, “TRUE” and “FALSE”, have their own class, “logical”.

class(TRUE)

## [1] "logical"

Randomly generated numbers would also have a class of “numeric”, as they form a numeric vector. Missing values, represented by “NA”, are by default logical, but you can also have missing values of other classes, such as missing integers. Character strings have a class of “character”. These are all examples of atomic classes.

class(rnorm(10))

## [1] "numeric"

class(NA)

## [1] "logical"

class("oops")

## [1] "character"

In R, you can go beyond the atomic classes and create new data types. For instance, if you fit a linear model, the output from the lm() function is an lm class, which is not a standard data type like numeric, integer, or character. Instead, it represents a set of statistical ideas that are encapsulated in the output of a linear model. Thus, it deserves its own data type or class, which we call the lm data type.

x<-rnorm(100)  
y<-x+rnorm(100)  
fit<-lm(y~x) #linear regression model  
class(fit)

## [1] "lm"

Generics and methods are functions that you write to implement functionality for certain types of classes. While S4 and S3 generic functions may look different, they play the same role conceptually. You can write new methods for an existing generic or create new generic functions for performing a new operation. To do this, you have to write both the generic and the method for a given data type. Help files in the methods package can provide documentation and guidance for learning and working with these concepts. In R, there are various generic functions such as mean and print.

mean

## function (x, ...)   
## UseMethod("mean")  
## <bytecode: 0x557f23238290>  
## <environment: namespace:base>

print

## function (x, ...)   
## UseMethod("print")  
## <bytecode: 0x557f24ccec60>  
## <environment: namespace:base>

These functions have almost no code associated with them except for a UseMethod() statement. The purpose of these generic functions is to find an appropriate method for a given data type. To see the available methods for a generic function, you can use the methods() function.

methods(mean)

## [1] mean.Date mean.default mean.difftime mean.POSIXct mean.POSIXlt   
## [6] mean.quosure\*  
## see '?methods' for accessing help and source code

Some packages may add new methods to the generic functions as you load them into R. Here is an example of S4 generic function.

show

## standardGeneric for "show" defined from package "methods"  
##   
## function (object)   
## standardGeneric("show")  
## <bytecode: 0x557f23ae91c8>  
## <environment: 0x557f23c12220>  
## Methods may be defined for arguments: object  
## Use showMethods("show") for currently available ones.  
## (This generic function excludes non-simple inheritance; see ?setIs)

The show() function from the methods package is an S4 generic function that is equivalent to print(). When you print out the body of the show() function, you will see more code than for the S3 generic functions.

showMethods(show)

## Function: show (package methods)  
## object="ANY"  
## object="classGeneratorFunction"  
## object="classRepresentation"  
## object="envRefClass"  
## object="externalRefMethod"  
## object="function"  
## (inherited from: object="ANY")  
## object="genericFunction"  
## object="genericFunctionWithTrace"  
## object="MethodDefinition"  
## object="MethodDefinitionWithTrace"  
## object="MethodSelectionReport"  
## object="MethodWithNext"  
## object="MethodWithNextWithTrace"  
## object="namedList"  
## object="ObjectsWithPackage"  
## object="oldClass"  
## object="refClassRepresentation"  
## object="refMethodDef"  
## object="refObjectGenerator"  
## object="signature"  
## object="sourceEnvironment"  
## object="standardGeneric"  
## (inherited from: object="genericFunction")  
## object="traceable"

The idea is the same, however, and the appropriate method is dispatched based on the class of the object being passed to it. Most of the time in R, objects are auto-printed at the command line, so you don’t usually need to call the show() or print() functions explicitly. However, various types of objects have their own special print or show methods that are associated with the show() or print() generic functions.

The interaction between generic functions and methods can be explained as follows. The first argument of any generic function is an object of a particular class. The generic function checks the class of the object and searches for a method specific to that class. If a method exists, it is called and the generic function’s job is done. If there is no method for that class, the generic function checks for a default method that always gets applied if there is no specific method. If there is no default method, an error occurs. It is important to note that it is generally not possible to print the code of an S3 or S4 method function by simply typing the name of the function in the command line. Instead, you can use the getS3method(<generic>, <class>) or getMethod(<generic>, <class>) function to view the code. The first argument of these functions is the name of the generic, and the second argument is the class of the object.

### 3.1.5 R Classes and Methods Part 2

In this example, we will demonstrate how a simple S3 class and method work. First, we generate a vector of random normal data and calculate its mean, which turns out to be -0.03069.

set.seed(2)  
x <- rnorm(100)   
mean(x)

## [1] -0.03069816

However, there’s a lot going on behind the scenes. The class of the vector x is numeric, so when we call the mean function on it, it will search for a specific method for numeric objects. However, since there is no such method, the default method for mean is called. This default method calculates the sum of all the elements in the vector and divides it by the length of the vector. You can inspect this default method using the getS3method() function, which reveals additional arguments like trim and na.rm.

head(getS3method("mean", "default"), 10)

##   
## 1 function (x, trim = 0, na.rm = FALSE, ...)   
## 2 {   
## 3 if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {   
## 4 warning("argument is not numeric or logical: returning NA")  
## 5 return(NA\_real\_)   
## 6 }   
## 7 if (na.rm)   
## 8 x <- x[!is.na(x)]   
## 9 if (!is.numeric(trim) || length(trim) != 1L)   
## 10 stop("'trim' must be numeric of length one")

In the function’s code, it checks a number of things, such as trimmed mean, before ultimately calling some internal C code that efficiently calculates the mean.

To illustrate a slightly more complex example, consider the following example.

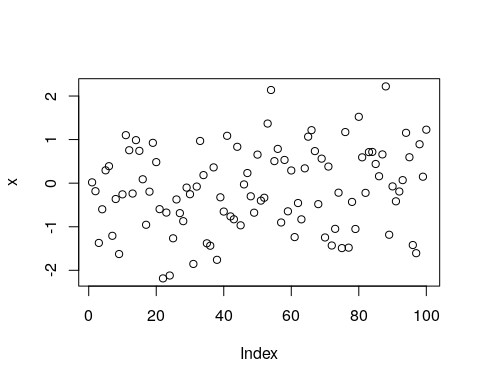
set.seed(3)  
df <- data.frame(x = rnorm(100), y = 1:100)   
sapply(df, mean)

## x y   
## 0.01103557 50.50000000

One important thing to note about data frames is that each column can potentially be of a different class. In this case, the first column is numeric, while the second column is integer. When we apply the mean function over the data frame using the sapply() function, the mean function will determine the class of each column and search for an appropriate method for that class. For the first column, which is numeric, there is no specific method, so the default method is called. For the second column, which is integer, there is also no specific method, so the default method is called again. If you have a large data frame with many columns of different classes, the mean function will check each column to find an appropriate method for that class. Sometimes it’s possible to directly call an S3method(), as some methods are visible to the user. For instance, you can call the mean.default function directly without invoking the generic function. However, it’s generally recommended to always call the generic function and let the appropriate method be dispatched automatically. This approach results in cleaner and more robust code, since the method name can change without affecting the underlying code. With the S4 system, this isn’t an issue because for the most part, you cannot call methods directly.

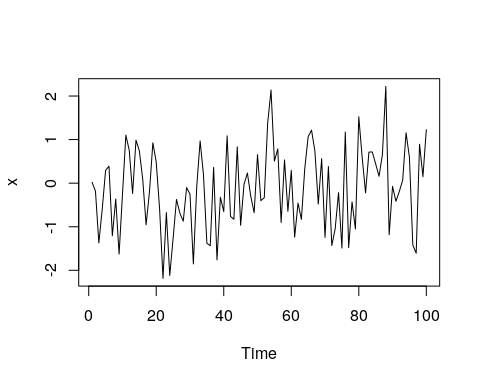
Here are two examples of an S3 Class/Method the plot function.

set.seed(10)  
x <- rnorm(100)   
plot(x)



Here we generate random normal data and call the plot() function. Since there’s no numeric method for plot(), it calls the default method and creates a scatter plot.

set.seed(10)  
x <- rnorm(100)  
x <- as.ts(x) ## Convert to a time series object   
plot(x)



However, in the second example, we convert the random data into a time series object using the as.ts() function and call plot(x) in the exact same way as before. This time, the plot looks different and has a label on the x-axis called **time**. This is because there’s a special plotting method for time series objects, and that method is being called here instead of the default method. If you frequently work with new data types, such as gene expression or spatial-temporal data, it may be necessary to create a new class and write methods for printing, showing, summarizing, or plotting the data.

There are two basic ways to extend R with the classes and methods system: you can create a new class and then write methods for existing generic functions like print or plot, or you can create new generic functions and methods for those generics for your new class. In this lecture, we will focus on S4 classes. The reason for creating a new class is that it allows you to work at a higher level of abstraction and represent more complex data types in a more compact manner. For instance, you might want to represent hierarchical or sparse matrices, which are not included in R’s basic data types. New concepts often emerge, such as point process models or mixed effects models, that haven’t been implemented in R yet, prompting the need for new classes. Additionally, you may want to hide implementation details from the user that would be exposed if you tried to represent a new data type using just lists and vectors. It’s worth noting that creating a new class doesn’t necessarily mean inventing a completely new data type; it simply means that R doesn’t have any special handling for that type. To create a new class, you can use the setClass() function, which requires at least the name of the new class. However, new classes often have associated data elements called **slots**. You can define methods for the new class using the setMethod() function. Information about a class definition can be obtained with the showClass() function. At a high level, a class can be thought of as a list with slots, where each slot contains an object of a specific class. Thus, you cannot store arbitrary data in any slot; you must use the specific data type for that slot.

Here we are creating a polygon class there is no data object called polygon in R.

library(methods)   
setClass("polygon",  
 representation(x = "numeric",  
 y = "numeric"))

To create a polygon class, one way is to represent it as a set of vertices with lines connecting them. This can be achieved by creating a class with two slots: x for the x coordinates of the vertices and y for the y coordinates of the vertices. It’s important to note that the data type for each slot must be specified, such as **numeric** in this case. Once a new class is created, methods can be defined using the setMethod() function. For example, a plot method can be created for the Polygon class. The signature specifies the set of classes that the method will operate on, which in this case is the Polygon class. The plot method takes an X and a Y argument, where X is the polygon object and Y is missing. Other arguments can be passed in using the @ symbol.

setMethod("plot", "polygon",   
 function(x, y, ...) {  
 plot(x@x, x@y, type = "n", ...)   
 xp <- c(x@x, x@x[1])  
 yp <- c(x@y, x@y[1])  
 lines(xp, yp)  
 }  
 )

Since plot already existed as a generic function, we did not need to create a new one. After calling setMethod the new plot method will be added to the list of methods for plot.

library(methods)   
showMethods("plot")

## Function: plot (package graphics)  
## x="ANY"  
## x="polygon"

In addition to “polygon” method, there is also a default method denoted by “ANY” for plot that takes an argument of any class. This method is called when no other method is available.

Here we are creating a new object of the polygon class and plotting it.

p <- new("polygon", x = c(1, 2, 3, 4), y = c(1, 2, 3, 1))  
plot(p)

To demonstrate the process of creating a new class and its methods, we have used the new() function to create an object of the polygon class. This is an example of how classes and methods can be used to extend the functionality of R. Classes define new data types and allow R to represent new types of data, while methods specify the behavior of generic functions on these new classes. Developing classes and methods provides an easier interface for users to interact with new types of data without getting bogged down in implementation details. R packages are a popular way to make new classes and methods available to users. Keep in mind that the best way to learn this stuff is to look at examples. There are quite a few examples on CRAN which use S4 classes/methods. You can usually tell if they use S4 classes/methods if the methods package is listed in the Depends: field. Another good resource is the Bioconductor project, which is a collection of R packages for analyzing genomic data. The [Bioconductor](http://www.bioconductor.org) project makes extensive use of S4 classes and methods. The stats4 package (comes with R) has a bunch of classes/methods for doing maximum likelihood analysis.

## 3.2 Quiz 3

## 3.3 Assignment 3

# 4 Fourth week

## 4.1 Swirl Courses

Hello, I’m Sean Cross your instructor for this week’s content on Swirl. Swirl is an interactive tool designed to teach R programming and data science concepts directly within the R console. With Swirl, you can learn at your own pace, interactively, and without needing to switch between different environments. Swirlify is an R package that allows you to create your own Swirl courses and lessons. In this lesson, we will demonstrate how to create Swirl courses and lessons using Swirlify.

### 4.1.1 Swirl 1.1

We will demonstrate how to create Swirl courses and lessons with Swirlify. Swirl is an R package that transforms the R console into an interactive learning environment. In case you’re new to Swirl, we will give you a quick overview. First, you need to load the Swirl package library(swirl) and then start Swirl by calling swirl() in R consol. As you can see, the console starts prompting with questions. We can select a Swirl course to begin, and then within that course, we can pick a lesson. Swirl provides text prompts in the console and asks you to enter specific R commands. For example, it would ask you to type in 5 + 7. If you entered the correct answer, Swirl would move on to the next question. If you made a mistake, Swirl would provide a hint to guide you in the right direction.

Swirl offers various types of questions, and we can exit a lesson at any time by pressing the Escape key. Now, let’s move on to creating Swirl lessons using Swirlify.

### 4.1.2 Swirl 1.2

Here we are going to demonstrate how to write your own swirl courses and lessons using the swirlify package. To begin, you will need to load the swirlify package library(swirlify), if it is not installed on your computer you can install it with install.packages("swirlify"). Swirlify is always aware of your current working directory, so make sure you set the working directory to your desired directory. Here, we will set the directory to a special folder on the desktop called “Courses,” where we wish to keep our swirl courses and lessons.

You can start writing a new swirl course and lesson using the following template.

setwd(file.path("~", "Desktop/Courses"))  
swirlify("Lesson 1", "My First Course")

So we set the working directory and name the lesson “Lesson 1” and the course “My First Course”. Upon executing this function, several things happen. It generates a few template files for you to create your course, then it will start the Shiny lesson where you can start authoring the app. All swirl lessons are written in YAML, which is a simple markup language. In YAML, each question is demarcated by a hyphen and is followed by key-value pairs that specify each type of question and its content. To give you a look at what we mean, let’s go to my “Courses” folder after executing the above code. Inside, you’ll see the course that was just created “My First Course” which has one lesson inside it named “Lesson 1”. Each course is its own folder, and each lesson is its own folder inside a course.

My\_New\_Course  
- My\_First\_Lesson  
 - lesson.yaml  
 - initLesson.R  
 - dependson.txt  
 - customTests.R

Alternatively, you can use the new\_lesson() function to create your course and lesson. With this approach, you should specify the directory where you want to create the course and lesson. Then run the new\_lesson() function and specify the name of the lesson and course.

setwd(file.path("~", "Desktop/Courses"))  
new\_lesson("My First Lesson", "My First Course")

The difference between this approach and the previous approach is that the new\_lesson() function gives you access to the YML file through R consol, whereas the swirlify() function starts a Shiny app which contains YML file and it is self-explanatory. Any changes you make in the Shiny app will be reflected in the YML file. There is also a great tutorial on the swirlify package that you can access in [Youtube](https://www.youtube.com/watch?v=UPL_W-Umgjs).

In the lesson folder, the file, “customTests.R”, is where you can specify your own functions for testing the correctness of swirl questions, [example](https://github.com/seankross/Test_Course/blob/master/Test_Lesson/customTests.R). The “dependson.txt” text file allows you to list the names of R packages that are on Cran, [example](https://github.com/seankross/Test_Course/blob/master/Test_Lesson/dependson.txt). This is where we check to ensure the R packages are installed before the lesson begins. The “initLesson.R” script is run before a lesson starts, allowing you to load data or create functions that will be used in the lesson, [example](https://github.com/seankross/Test_Course/blob/master/Test_Lesson/initLesson.R). Lastly, the “lesson.yaml” file is where we specify the questions for the lesson, [example](https://github.com/seankross/Test_Course/blob/master/Test_Lesson/lesson.yaml).

We recommend using RStudio for writing swirl lessons because it allows you to have a text editor and R console side by side. However, different people have different workflows. To add questions, you can use the wq\_ series of functions that come with swirl. To see the list of these questions you can just type in wq\_ in the consol and the different types will automatically show up. There are several types of questions you can include in a lesson, including message questions wq\_message() and command questions wq\_command(). A message question is a simple prompt that displays a message to the student. A command question prompts the student to enter R code into the console. You either just execute wq\_message() and then edit the put your text output here in the YML file Output: put your text output here or you can execute wq\_message("Welcome to lesson 1") and it will automatically add the message to the YML file.

When writing a command question, you specify the correct answer and a function to test the correctness of the student’s answer. The function has a format like this:

wq\_command(output = "explain what the user must do here",  
 correct\_answer = "EXPR or VAL",  
 answer\_tests = "omnitest(correctExpr='EXPR', correctVal=VAL)",  
 hint = "hint")

The output argument is the prompt that is displayed to the student. The correct\_answer argument is the correct answer to the question. The answer\_tests argument is a function that tests the correctness of the student’s answer.The omnitest function is recommended for most questions. The hint argument is a hint that is displayed to the student if they answer incorrectly. You can either fill in the arguments with the correct values or you can just execute the function and then edit the YML file. Here is an example of a command question:

- Class: cmd\_question  
 Output: Add 2 and 2 together using the addition operator.  
 CorrectAnswer: 2 + 2  
 AnswerTests: omnitest(correctExpr='2 + 2')  
 Hint: Just type 2 + 2.

Once you have written some questions, and you want to add the lesson to the course, you need to update the course manifest using the add to manifest function add\_to\_manifest(). This function creates a manifest file inside the course directory outside the lesson folder and it keeps track of the order of lessons in the course and prevents files that are not lessons from appearing in the lesson selection menu. To check the lesson’s formatting, use the test lesson function test\_lesson(). If no warnings or messages are returned, the lesson passed formatting checks. To preview the lesson, use the demo lesson function demo\_lesson(), which takes you directly to the lesson without going through swirl’s menu. Once you’re in the lesson, you can answer the questions and complete it. In this case, the first question is “Welcome to lesson 1,” and the second question is “Add 2 and 2 using a plus operator.”

### 4.1.3 Swirl 1.3

Now that we have created and tested our first lesson, let’s move on to some more complex question types that we can include in a swirl lesson. If we want to start a new lesson, we can simply close the current lesson.yaml file and create a new lesson within the same course. To confirm that we are working in the correct course, we can use the get\_current\_lesson() function, which will return information about the current lesson and course. In this case, our course is called “My First Course” and we will create Lesson 2 within it. It’s important to note that the first question generated in a lesson is the meta question, which provides information about the lesson’s author and organization. This question is not visible to the students. We can fill out the author and organization fields within the meta question to provide this information. Okay, let’s dive into some more advanced question types. For instance, to create a multiple-choice question, we can use the wq\_multiple() function and specify the question prompt and answer choices. Here is an example to add a multiple-choice question.

wq\_multiple(output = "What is the capital of Canada?",  
 answer\_choices = c("Toronto","Montreal","Ottawa","Vancouver"),  
 correct\_answer = "Ottawa",  
 answer\_tests = "omnitest(correctVal= 'Ottawa')",  
 hint = "This city contains the Rideau Canal.")  
>

Once we’ve created the question, we can add it to the course manifest and run the test\_lesson() and demo\_lesson() functions to ensure everything works properly.

To create a figure question, we can use the wq\_figure() function and specify the message to display to the student, such as “This is a simple graph.” Example:

wq\_figure(output = "This is a simple graph.",  
 figure = "sourcefile.R",  
 figure\_type = "new")

To start a new lesson in the “My First Course” course, we can close the current “lesson.yaml” file. You can use the get\_current\_lesson() function to confirm which course you are working on.

Let’s start a new lesson and name it “Lesson 2” and create a multiple-choice question using the wq\_multiple() function. We can provide the question prompt, answer choices separated by semi-colons, the correct answer, and a hint. After adding this lesson to the course manifest and running test\_lesson() and demo\_lesson() with no errors, we can move on to creating a figure type question. Using wq\_figure(), we add a new figure question template with the Figure and FigureType fields. The Figure field specifies an R script that will generate the figure, while the FigureType field specifies whether the question will build on an existing figure or create a new one. We create a simple plot and save it as “fig1.R” in the lesson directory, then use demo\_lesson() to display the plot.

wq\_figure(output = "simple graph", figure = "fig1.R",  
 figure\_type = "new")

# fig1.R  
plot(1:10)

We can then add another figure question to modify the plot by adding a line. We save the code for adding the line in “fig2.R” and set the FigureType to add. Finally, we use demo\_lesson() to display the modified plot with the added line.

wq\_figure(output = "I added a line",  
 figure = "fig2.R",  
 figure\_type = "add")

# fig2.R  
abline(h=5, col="red", lwd=2)

For more information on creating swirl lessons, you can visit the swirl [website](http://swirlstats.com/swirlify/). There you can the documentation. You can also find a link to the swirl course development guide, which contains more detailed information on creating swirl lessons.

### 4.1.4 Swirl 1.4

Swirlify makes sharing a swirl course easy. We recommend three different methods for sharing a swirl course.

#### 4.1.4.1 Sharing Your Course as a File

We’ve developed the .swc file type so that you can share your course as a single file. To create your .swc file follow these steps:

1. Set any lesson in the course you want to share as the current lesson using set\_lesson().
2. Create an .swc file using the pack\_course() function. Your .swc file will appear in the same directory as the directory that contains the course folder. You can then share the .swc file with others through email, file sharing services, etc. To install the .swc file, the user can use the install\_course() function and specify the path to the .swc file. The course will then be installed in the user’s home directory. The user can then use the swirl() function to start the course.

#### 4.1.4.2 Sharing Your Course on GitHub

We highly encourage you to develop your course on GitHub so that you can get better support if you have questions or need assistance while developing your course.In addition, developing your course on GitHub provides the added benefit that your course will be instantly ready to distribute. Students can install your course from swirl using the ‘install\_course\_github()’ function. Make sure that your course directory is the root folder of your git repository. For examples of courses that have been shared on GitHub you can browse some of the courses on the [Swirl Course Network](http://swirlstats.com/scn/).

#### 4.1.4.3 Sharing Your Course on the Swirl Course Network

The goal of the Swirl Course Network is to list and organize all publicly available swirl courses. Visit the [homepage](http://swirlstats.com/scn/) of the SCN for more information. After adding your course to the SCN students will be able to install your course using install\_course("[Name of Your Course]") in swirl.

Briefly, in order to add your course to the SCN:

1. Create an .swc file for your course.
2. Fork <https://github.com/swirldev/scn> on GitHub.
3. Add the .swc file to your fork.
4. Add an Rmd file to your fork like this one. You can include a description of your course, authors, a course website, and how to install your course.
5. Run rmarkdown::render\_site() when your current directory is set to your fork.
6. Add, commit, and push your changes to GitHub, then send us a Pull Request.

### 4.1.5 Course Project

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | [FirstName LastName](link%20to%20personal%20website) |
| Lecturer(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | Delivered the course in some way - video or audio |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2023-05-26   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2023-03-28 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.7 2023-02-24 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.1 2023-03-23 [1] CRAN (R 4.0.2)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.20 2023-01-17 [1] CRAN (R 4.0.2)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## htmltools 0.5.5 2023-03-23 [1] CRAN (R 4.0.2)   
## knitr 1.33 2023-03-28 [1] Github (yihui/knitr@a1052d1)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 1.1.0 2023-03-14 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2023-03-28 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.3 2022-04-02 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2023-03-28 [1] Github (R-lib/testthat@e99155a)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2023-03-28 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 5 References