STA553 E-Pack: Data Visualization

Cheng Peng

West Chester University

Contents

1 Introduction **7**
- 1.1 Data or Information Visualization? . . . 7
- 1.2 Aesthetic Considerations . . . 8
- 1.3 Topic Coverage . . . 10

2 Getting Started With RMarkdown **11**
- 2.1 What is RMD . . . 11
- 2.2 Code Chunks . . . 13
- 2.3 Inserting Graphics . . . 13
- 2.4 Inserting Tables . . . 16
- 2.5 Creating PDF . . . 18
- 2.6 Miscellaneous . . . 18

3 Open-Source Tools for Data Viz **21**
- 3.1 DataViz . . . 21
- 3.2 R & RStudio . . . 22
- 3.3 Tableau Public . . . 23
- 3.4 Shiny Server . . . 23
- 3.5 RPubs . . . 24
- 3.6 Github . . . 25
- 3.7 SAS OnDemand . . . 26
- 3.8 R Viz Libraries . . . 27

4 R Functions and Flow Controls **29**
- 4.1 Control flow . . . 29
- 4.2 Functions . . . 33

5 Getting Started with Base R Graphics **41**
- 5.1 Base Graphics . . . 41
- 5.2 Simple Base Graphics . . . 42
- 5.3 Some Important Base Graphics Parameters . . . 44
- 5.4 Base Plotting Functions . . . 45
- 5.5 Base Plot with Regression Line . . . 48

5.6 Multiple Base Plots . . . 49
5.7 Controlling Point Size and Transparency . . . 52
5.8 Annotations . . . 53

6 Foundations of Data Visualization 55

7 Ethics in Data Visualization 73

8 Data Processing with Tidyverse 85
8.1 Data Cleaning and Preparation for Visualization . . . 86
8.2 Basic Data Management: Merging Data Sets . . . 86
8.3 Basic Data Management: Subsetting Data . . . 91
8.4 Importing/Exporting Data . . . 101
8.5 Importing Dara . . . 101
8.6 **Overview of `Tidyverse` (Optional)** . . . 102
8.7 Data Management with `dplyr` (Optional) . . . 103

9 Introduction to Ggplot 111
9.1 Basics of `ggplot()` . . . 111
9.2 Structure of `ggplot()` . . . 112
9.3 Geoms . . . 113
9.4 Mapping Data to Plot . . . 113
9.5 Adding Annotations to Graphics . . . 124
9.6 Aminated Graph with `gganimate()` . . . 130
9.7 Ridgetline Plot with `ggridges` Library . . . 132
9.8 Other Extensions to ggplot . . . 135
9.9 Save `ggplot` Images . . . 136
9.10 Save ggplot with `ggsave()` . . . 137

10 Interactive Statistics Graphics 139
10.1 Plotly . . . 139
10.2 ScatterPlot . . . 140
10.3 Barplot . . . 149
10.4 Histogram . . . 150
10.5 Boxplot . . . 151
10.6 Pie Chart . . . 152
10.7 Density Curve . . . 153
10.8 Serial Plot . . . 154
10.9 Plotly Maps . . . 155

11 Interactive Maps 157
11.1 Map Types . . . 157
11.2 Leaflet Maps . . . 159
11.3 Choropleth Maps . . . 164
11.4 Plotly Map . . . 170
11.5 Mapview Maps . . . 177
11.6 Thematic Maps . . . 184

11.7 Tableau Maps 188
11.8 R Color Palettes 191

12 A `mapview` Map Demo **199**

13 Creating Maps Using Shapefiles **201**
13.1 Existing Base Map File and New Information 201
13.2 Creating Shapefiles from Dataframes 202
13.3 Thematics Map with Created and Built-in Shapefiles 203

14 Introduction to Tableau **207**
14.1 Data Loading 207
14.2 Opening Work Sheet 209
14.3 Basic Statistical Charts with Existing Variables 209
14.4 Basic Charts with Derived Variables 222
14.5 Tableau Dashboards 222
14.6 Some Youtube Tutorials on Tableau 226

15 Getting Started with R Shiny **229**
15.1 The IDE of RShiny Apps 230
15.2 Embedding Shiny Apps in RMarkdown 230
15.3 Components of An Shiny Applications 230
15.4 The Anatomy of a Shiny Application 231
15.5 How Shiny Apps Work? 233
15.6 Some Built-in Demonstrations of Shiny Apps 233
15.7 UI Layout Designs - A Glance 234
15.8 Some Basic Input Widgets 237
15.9 Case Study I - Density of Normal Distribution 239
15.10Case Study 2 - Central Limit Theorem 240
15.11More Effective Code 243

16 Shiny UI and Server Design **247**
16.1 Naming Conventions Revisited 247
16.2 Working Data Set 248
16.3 UI Module: Front-end Design 248
16.4 Server Module: Back-end Coding 250
16.5 File Uploading 254

17 Shiny Dashboard and Storyboard **257**
17.1 **flexdashboard** 257
17.2 **shinydashboard** 258
17.3 Three Web Application Terms 258
17.4 Using Flexdashboard 258
17.5 Storyboards 264

Chapter 1

Introduction

The term **data visualization** has been used for a long time. It is still an evolving field due to the continuing advancement of computing technology.

> **Data visualization** is the presentation of data in a pictorial or graphical format, and a data visualization tool is the software that generates this presentation. Data visualization provides users with intuitive means to interactively explore and analyze data, enabling them to effectively identify interesting patterns, infer correlations and causalities, and support sense-making activities.

1.1 Data or Information Visualization?

Sometimes, data visualization is also called information visualization. Can these two terms be used interchangeably? To answer this question, we need to know what is data and what is information.

There are different versions of definitions for data and information.

- **Data** are facts, figures, observations, or recordings that can take the form of images, sound, text, or physical measurements. Data can come from many sources and it can be split into two groups based on the form it takes: structured data and unstructured data.
  - **Structured data** - typically categorized as quantitative data - is highly organized and easily decipherable by machine learning algorithms.
  - **Unstructured data** - typically categorized as qualitative data, cannot be processed and analyzed via conventional data tools and methods.

- **Information** is a collection of data that has been processed, organized, or structured in a meaningful way to convey knowledge, ideas, or instructions. It can be communicated through various mediums, such as text, images, audio, or video, and can be accessed and shared through multiple channels, such as books, websites, and social media.
- **Relationship between Data and Information**: Data is meaningless and has no significance. Information is processed data and has meaning and significance. Information is dependent on data.

Based on the above definitions of data and information, it is more appropriate to call *data visualization* **information visualization**.

Another concept related to what we will do in this course is scientific visualization. According to the definition

> Scientific visualization: the representation of data graphically as a means of obtaining comprehension and insight into the scientific data. It can also refer to visual data analysis.

1.2 Aesthetic Considerations

Data visualization is both an art and a science. Aesthetically designed visualization makes the visual representation of data and information more effective. With the advances in the development of graphical software, aesthetic features have been increasingly used in various visual designs including data and information visualization. The challenge is how to create aesthetically attractive visualizations without misleading and distorting information.

The basic elements of data aesthetics are shape, size, color, position, orientation, font type, font size, and many others.

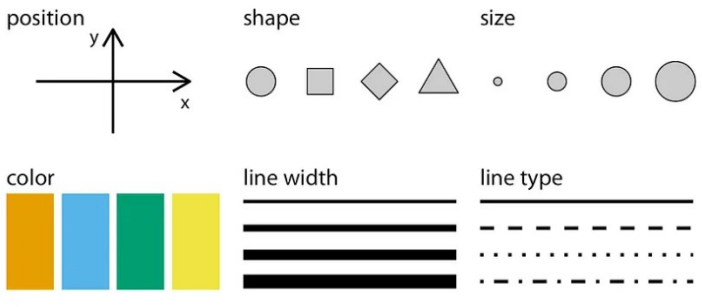


Image by Wilke in Fundamentals of Data Visualization.

Figure 1.1: Data aesthetic building blocks.

The key to creating an aesthetically effective and persuasive data visualization

is how choose the right tools for the right data/information. Keep visualization creative but simple! The following are a few examples that illustrate the right visual tools for the right types of data.

- Visualizing the magnitude of data

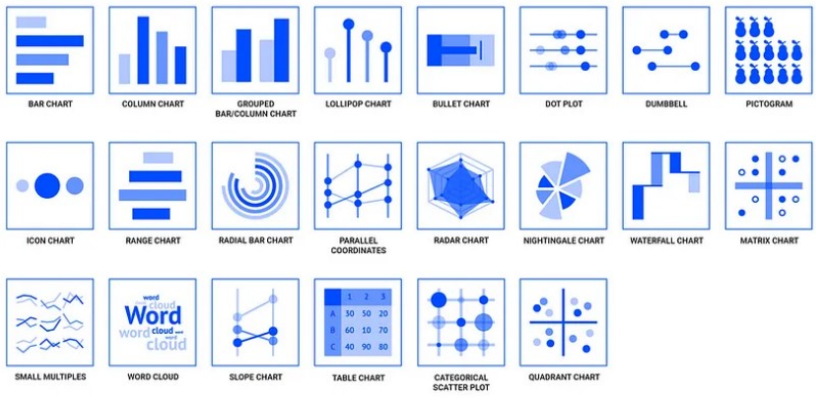


Image by Datylon

Figure 1.2: Visual comparison of numerical quantities.

- Visualizing the proportion of data

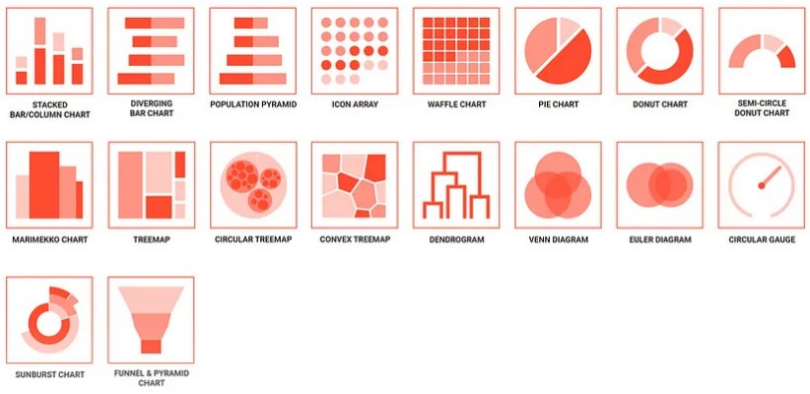


Image by Datylon.

Figure 1.3: Visualizing proportion and hiearchical relationships

- Visualizing the distribution of data

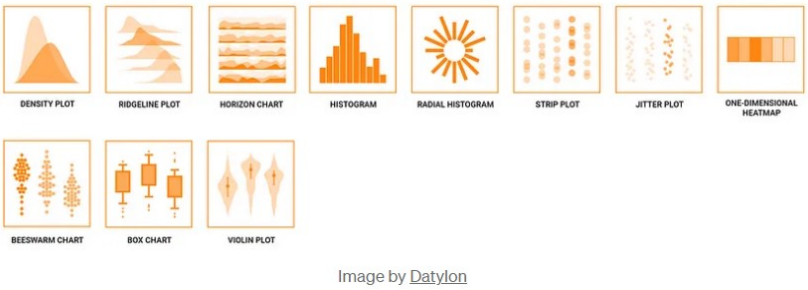


Figure 1.4: Visualizing the distribution of a variable.

1.3 Topic Coverage

This course focuses on both static and interactive data visualization using programmatic and non-programmatic approaches using R and Tableau. We will also briefly discuss the basic principles of visual design and its application in data visualization. We will cover a wide range of topics from technical tools and platforms commonly used in visualization, basic statistical plots, and interactive graphics, to dynamic dashboards.

Chapter 2

Getting Started With RMarkdown

2.1 What is RMD

R Markdown is a file format for making dynamic documents with R. An R Markdown document is written in markdown (an easy-to-write plain text format) and contains chunks of embedded R code, like the document below.

RMarkdown makes use of Markdown syntax. Markdown is a very simple 'markup' language that provides methods for creating documents with headers, images, links, etc. from plain text files while keeping the original plain text file easy to read.

R Markdown files are the source code for rich, reproducible documents. We can transform an R Markdown file in two ways.

- **knit** - The rmarkdown package will call the `knitr` package. `knitr` will run each chunk of R code in the document and append the results of the code to the document next to the code chunk.

- **convert** - The rmarkdown package will use the pandoc program to transform the file into a new format such as HTML, PDF, or Microsoft Word file. RMarkdown will preserve the text, code results, and formatting contained in the original.Rmd file.

The term `render` refers to the above two-step process of knitting and converting an R Markdown file.

2.1.1 Create an RMarkdown file

To create a new RMarkdown file (`.Rmd`), select `File -> New File -> R Markdown`in RStudio, then choose one of the file types we want to create.

The newly created.Rmd file comes with an RMarkdown template with basic instructions.

2.1.2 The YAML Header

At the top of any RMarkdown script is a YAML (Yet Another Markup Language) header section enclosed by `---`. By default, this includes a title, author, date and the file type you want to output to.

Many other options are available for different functions and formatting, see here for .html options and here for .pdf options. Rules in the header section will alter the whole document.

The following is the simplest YMAL template.

```
---
title: "Introduction to RMarkdown"
author: "Cheng Peng"
date: "12/25/2021"
output:
  html_document: default
  html_notebook: default
editor_options:
  chunk_output_type: inline
---
```

Note that YAML is automatically generated that reflects the choice of the file type and related file specifications. Therefore, we don't need to know YAML unless we want to modify it manually.

2.1.3 Markdown Syntax

- **Formatting Text**
  - *Italic* - single asterisk -> italic.
  - **bold** - double asterisks -> bold face
  - `code` - code in text
- `Header`
  - `# Header 1`: single `#` - level 1 header
  - `## Header 2`: double `##`- level 2 header
  - `### Header 3`: triple `###` - level 3 header
- **List**
  - `* Unordered list` - single asterisk **+ space** -> unordered list
  - `1. Ordered list` - numbered list
- **Hyperlink**

  - `< url >` - Example: https://www.wcupa.edu/
  - `[link-name](url)` - Example: WCUPA
- **LaTex Equations Syntax**
  - `$ LaTex commands $` - Example: $y = \beta_0 + \beta_1 x$

2.2 Code Chunks

Code that is included in .Rmd document should be enclosed by three `backward apostrophes ``` (grave accents!)`. These are known as code chunks. An executable code chunk looks like the following.

```
# code goes here
```

The first line:```{r chunk-name-with-no-spaces} contains the language (`r`) in this case, and the name of the chunk, and other optional code chunk options. Next to the {r}, there is a chunk name. The chunk name is not necessarily required however, it is good practice to give each chunk a `unique` name to support more advanced knitting approaches.

If the language is not specified in the curly bracket, the chunk is simply a verbatim environment that keeps the content in the knitted document. For example, the content in the following chunk will not be executed.

```
The context will not be processed when converting this document to HTML,
PDF, or other document formats.
```

2.2.1 Code Chunk Options

There are a few options that can be used to control the graphical output generated from the code in the code chunk. The following table lists a few of them.

2.3 Inserting Graphics

There are two different ways to include images into RMD: Generated by the code in the code chunk and external source of images.

- **R Generated Graphics**

Including R-generated graphics into RMD is straightforward. The following example shows how to embed a graphic with some code chunk options to layout the graphic.

```
data(iris)
pairs(iris[, -5])
```

- **External Images**

There are several ways to include an external image into RMD documents.

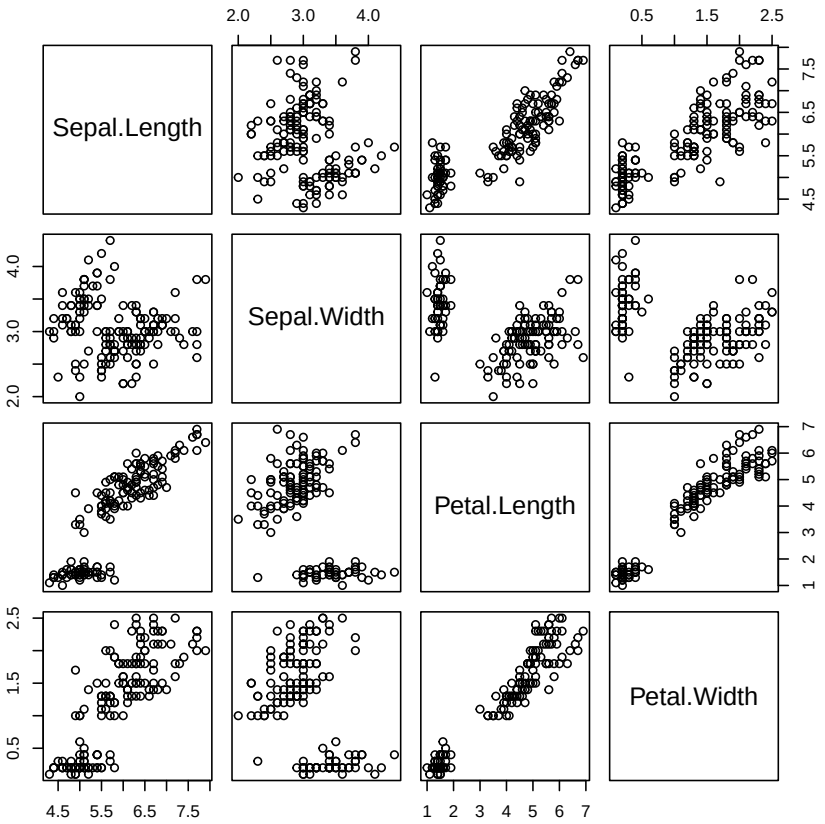


Figure 2.1: Pairwise scatter plot of iris data set

1. Using HTML `img` Tag

```
<br>
<center><img src="https://github.com/pengdsci/sta553/blob/main/image/boats.jpg?raw=true" alt="Por
<br>
```

Rule	Example (default)	Function
eval	eval=TRUE	Is the code run and the results included in the output?
include	include=TRUE	Are the code and the results included in the output?
echo	echo=TRUE	Is the code displayed alongside the results?
warning	warning=TRUE	Are warning messages displayed?
error	error=FALSE	Are error messages displayed?
message	message=TRUE	Are messages displayed?
tidy	tidy=FALSE	Is the code reformatted to make it look "tidy"?
results	results="markup"	**How are results treated?** "hide" = no results "asis" = results without formatting "hold" = results only compiled at end of chunk (use if many commands act on one object)
cache	cache=FALSE	Are the results cached for future renders?
comment	comment="##"	What character are comments prefaced with?
fig.width, fig.height	fig.width=7	What width/height (in inches) are the plots?
fig.align	fig.align="left"	"left" "right" "center"

Figure 2.2: RMarkdown Code Chunk Options

2. Using `knitr` Function

```
Chunk options: fig.align='center', echo=FALSE, fig.cap="White Mountain", out.width = '50%'
knitr::include_graphics("img01/w01-WhiteMountain.jpg)
```

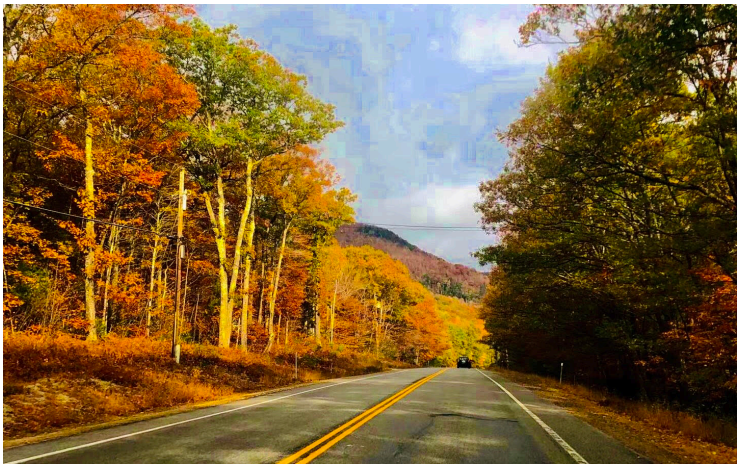


Figure 2.3: White Mountain

2.4 Inserting Tables

A quality table is also an important visualization tool. We introduce three methods for inserting nice-looking tables into the RMD document.

- **Extracting R Output Matrix Using `kable()` Function**

For example, we fit a linear regression model using iris data and extract the statistics of model fit to define a table.

```
data(iris)
iris.model = lm(Sepal.Length~., data = iris)  # fit a linear model
mod.stats = coef(summary(iris.model))         # inferential stats of coefficients
kable(mod.stats)   #
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	2.1712663	0.2797942	7.760228	0.0000000
Sepal.Width	0.4958889	0.0860699	5.761466	0.0000000
Petal.Length	0.8292439	0.0685276	12.100867	0.0000000
Petal.Width	-0.3151552	0.1511958	-2.084418	0.0388883
Speciesversicolor	-0.7235620	0.2401689	-3.012721	0.0030596
Speciesvirginica	-1.0234978	0.3337263	-3.066878	0.0025843

- **Markdown Table**

We can manually create a Markdown table. See the following example.

```
| Plant | Temp. | Growth |
|:------|:-----:|-------:|
| A     | 20    | 0.65   |
| B     | 20    | 0.95   |
| C     | 20    | 0.15   |
```

The resulting Markdown has the following

Plant	Temp.	Growth
A	20	0.65
B	20	0.95
C	20	0.15

The Markdown table syntax is explained in the following:

- `:----::` Centre
- `:-----:` Left
- `-----::` Right
- `------:` Auto

- **HTML Table**

```
<table border = 1>
  <tr>
    <th>Company</th>
    <th>Contact</th>
    <th>Country</th>
  </tr>
  <tr>
    <td>Alfreds Futterkiste</td>
    <td>Maria Anders</td>
    <td>Germany</td>
  </tr>
  <tr>
    <td>Centro comercial Moctezuma</td>
    <td>Francisco Chang</td>
    <td>Mexico</td>
  </tr>
</table>
```

A basic HTML table

Company

Contact

Country

Alfreds Futterkiste

Maria Anders

Germany

Centro comercial Moctezuma

Francisco Chang

Mexico

- **LaTex Table**

This will not work if we knit the RMD to HTML since it is a LaTex table.

```
\begin{table}
\centering
\begin{tabular}{llll}
1 & 2 & 3 & 4  \\
1 & 3 & 4 & 6  \\
3 & 4 & 1 & 8  \\
```

```
5 & 2 & 0 & 1
\end{tabular}
\end{table}
```

2.5 Creating PDF

Creating `.pdf` documents for printing in A4 requires a bit more fiddling around. RStudio uses another document compiling system called LaTeX to make `.pdf` documents.

The easiest way to use LaTeX is to install the `TinyTex` distribution from within RStudio. First, restart the R session (Session -> Restart R), then run these lines in the console:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

Becoming familiar with LaTeX will give you a lot more options to make your R Markdown .pdf look pretty, as LaTeX commands are mostly compatible with R Markdown, though some googling is often required.

To install a full-functioning LaTex, MikTex is recommended. The official website for MikTex is https://miktex.org/download.

We can also choose different options to make a nicer PDF by clicking the arrow on the gear sign () and then selection `Output Options...`. Then select **Output Format:** as `PDF`. Then we can choose appropriate options.

2.6 Miscellaneous

We have introduced the basics of R Markdown. Several good features are worth mentioning.

2.6.1 Markdown Presentations

R Markdown can prepare presentations in HTML slides, PDF Beamer, and Microsoft PowerPoint Presentation. Some of the lecture notes will be prepared in PDF Beamer.

2.6.2 Shiny Apps and RMarkdown

A recent development is the ability to put Shiny elements into an RMarkdown document.

These documents, again, need a Shiny server to run, but take advantage of the easy formatting of RMarkdown to present the user interface - server and UI elements sit in the same document.

- **RMarkdown** - supplies the HTML instead of a `ui.R` file.
- **Shiny** - supplied reactive components within your RMarkdown

We will prepare shiny lecture notes using RMarkdown in this class.

2.6.3 Working Directory

Sometimes we may want to use a specific directory as the working directory to store relevant files associated with the same analysis. The usual way to change the working directory is `setwd()`, **but** please note that `setwd()` is not persistent in R Markdown (or other types of knitr source documents), which means `setwd()` only works for the current code chunk, and the working directory will be restored after this code chunk has been evaluated.

It is not encouraged to use `setwd()` in RMarkdown documents.

2.6.4 CSS Style

We can include a CSS-style file to format HTML. The style file used in this RMD is given below.

```
<style type="text/css">
div#TOC li {
    list-style:none;
    background-image:none;
    background-repeat:none;
    background-position:0;
}
h1.title {
  font-size: 24px;
  color: darkRed;
  text-align: center;
}
h4.author { /* Header 4 - and the author and data headers use this too  */
    font-size: 18px;
  font-family: "Times New Roman", Times, serif;
  color: DarkRed;
  text-align: center;
}
h4.date { /* Header 4 - and the author and data headers use this too  */
  font-size: 18px;
  font-family: "Times New Roman", Times, serif;
  color: DarkBlue;
  text-align: center;
}
h1 { /* Header 3 - and the author and data headers use this too  */
    font-size: 22px;
```

```
    font-family: "Times New Roman", Times, serif;
    color: darkred;
    text-align: center;
}
h2 { /* Header 3 - and the author and data headers use this too  */
    font-size: 18px;
    font-family: "Times New Roman", Times, serif;
    color: navy;
    text-align: left;
}
h3 { /* Header 3 - and the author and data headers use this too  */
    font-size: 15px;
    font-family: "Times New Roman", Times, serif;
    color: darkred;
    font-face: bold;
    text-align: left;
}
h4 { /* Header 4 - and the author and data headers use this too  */
    font-size: 18px;
    font-family: "Times New Roman", Times, serif;
    color: darkred;
    text-align: left;
}
</style>
```

Chapter 3

Open-Source Tools for Data Viz

This note introduces software programs and platforms to be used for this data visualization course.

3.1 DataViz

```
w <- ggplot(gapminder, aes(gdpPercap, lifeExp,
                  size = pop, colour = country)) +
           geom_point(alpha = 0.7, show.legend = FALSE) +
           scale_colour_manual(values = country_colors) +
           scale_size(range = c(2, 12)) +
           scale_x_log10() +
           # break down the previous single plot by continent
            facet_wrap(~continent) +
           # Here comes the gganimate specific bits
           labs(title = 'Year: {frame_time}',
                x = 'GDP per capita', y = 'life expectancy') +
           transition_time(year) +
           ease_aes('linear')
###
animate(w, renderer = gifski_renderer())
```

https://github.com/pengdsci/sta553/blob/main/image/lifeExpContinent02.gif?raw=true

3.2 R & RStudio

3.2.1 What is R?

R is a language and environment for statistical computing and graphics. It is a GNU project that is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

R is an integrated suite of software facilities for data manipulation, calculation, and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple, and effective programming language that includes conditionals, loops, user-defined recursive functions, and input and output facilities.

– https://www.r-project.org/about.html

3.2.2 RStudio

RStudio is an integrated development environment (IDE) for R. It includes a console and syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging, and workspace management.

There are two versions of RStudio: RStudio Desktop and RStudio Server. Both versions have free open-source and commercial editions. We use the free open-source edition of RStudio Desktop that has the following features:

- Access RStudio locally
- Syntax highlighting, code completion, and smart indentation
- Execute R code directly from the source editor
- Quickly jump to function definitions
- View content changes in real time with the Visual Markdown Editor
- Easily manage multiple working directories using projects
- Integrated R help and documentation
- Interactive debugger to diagnose and fix errors
- Extensive package development tools

3.2.3 The Relationship between R and RStudio

R and RStudio are two distinctly different applications that serve different purposes. R is a programming language used for statistical computing while RStudio uses the R language to develop statistical programs.

R and RStudio are not separate versions of the same program, and cannot be substituted for one another. R may be used without RStudio, but RStudio may not be used without R.

3.3 Tableau Public

3.3.1 About Tableau

Tableau is a powerful and fastest-growing data visualization tool used in the Business Intelligence Industry. The great thing about Tableau software is that it doesn't require any technical or any kind of programming skills to operate. Tableau suite has different products that confuse new users.

- **Tableau Desktop** has a rich feature set and allows you to code and customize reports. It is not free (actually pretty expensive)!
- **Tableau Public** creates workbooks that cannot be saved locally, in turn, it should be saved to Tableau's `public server` in the cloud which can be viewed and accessed by anyone. You need to download and install it on your computer to design workbooks offline and then save them to Tableau's `public server`. It is totally free!
- **Tableau Server** is specifically used to share the workbooks, and visualizations that are created in the Tableau Desktop application across the organization. It is NOT free! . However, the public server is free.
- **Tableau Online** has all the similar functionalities of the Tableau Public, but the data is stored on servers hosted in the cloud which are maintained by the Tableau group. That means you design workbooks on the Tableau's public server. It is also free!
- **Tableau Reader** is a free tool that allows you to view the workbooks and visualizations created using Tableau Desktop or Tableau Public.

3.4 Shiny Server

3.4.1 What is shiny.io?

`Shinyapps.io` is an easy-to-use, scalable place to put `Shiny applications` so that other people can use them over the web without having to set up a Shiny Server.

`Shinyapps.io` is not completely free but has a free version with limited. We can register a free account for practice purposes. The following is brief information about `Shinyapps.io`.

3.4.2 Register An Account with shiny.io

If you don't have an account with `shinyapps.io`, you need to register an account. Otherwise, log into the account to publish your ShinyApps. The following two hyperlink buttons provide the links to the appropriate web pages.

You can also find the official web page at https://www.shinyapps.io/

3.4.3 Requirements

Your deed to install a recent version of You'll need `R` itself, `RStudio`, and the `knitr` package on your machine.

3.4.4 Steps for Publishing ShinyApps

- In RStudio, create a new R ShinyApps (e.g., `myshinyapps.R`) document by choosing `File` | `New File` | `Shiny Web Apps`.
- Click button in top right toolbar to preview the shinyapp.
- In the preview window, click button.

3.5 RPubs

3.5.1 What is RPubs?

3.5.2 Register An Account With RPubs

First of all, you need to sign up for an account with RPubs if you don't have one. Otherwise, sign in to your existing RPubs account. The following two hyperlink buttons will bring you to the appropriate website.

3.5.3 Requirements

Your deed to install a recent version of You'll need `R` itself, `RStudio`, and the `knitr` package on your machine.

3.5.4 Steps for Publishing on RPubs

- In RStudio, create a new R Markdown document by choosing `File` | `New` | `R Markdown`.
- Click the `Knit HTML` button in the doc toolbar to preview your document.
- In the preview window, click button.

3.6 Github

3.6.1 What is Github?

GitHub is a social networking site for programmers to share their code. Many companies and organizations use it to facilitate project management and collaboration. It is the most prominent source code host, with over 60 millions of new repositories.

Most importantly, it is free. We can also use this resource to host web pages. Many images and data sets that I used are stored on GitHub.

3.6.2 Register A Github Account

You can use the following two buttons to sign up an an account with Github or sign in to an existing Github account.

3.6.3 Getting Started With GitHub

We will use screenshots to demonstrate how to create repositories, folders, and files.

1. After you log into your account, you click the "continue for free" button located in the bottom of the following page (screenshot)

2. Now you see your GitHub front page. Click the green button "create repository" on the left panel. Our first repository is called "sta553"

3. To organize the file in the repository `sta553`, We want folders for different files. To create a folder under `sta553`, click the hyperlink `creating a new file`

4. The first folder to create is called `data` folder which will be used to store data files. After typing "data/", a new box appears under "data" folder, type the first file name - readme, and the content of the file (see the screenshot). In the end, click the green button "commit new file" to complete the creation of the first folder in the repository `data`.

5. To load the data file to the `data` folder, we click drop-down menu on the top right corner and select `upload files`

6. To create other folders under `sta553`, we click `Creating New File`, we can create a new folder `image` similarly.

7. To create a new repository, Click the drop-down menu on the top right corner and select `New repository` to create a new repository.

3.7 SAS OnDemand

3.7.1 What is SAS OnDemand (SAS Studio)

SAS OnDemand provides **free** data management and data analysis tools. The advantage of SAS OnDemand is that it does not require any installation and it runs on the cloud via the Internet and process data by connecting to the SAS server in the cloud. In other words, your computer is only used as a monitor since it does not use any resources (memory and CPU) of your computer.

Click `Access` to enter into the SAS OnDemand login page.

3.7.2 Sign-in / Sign-up

If you have already created your SAS Profile, use the email or user ID and the password to log into the SAS OnDemand page.

3.7.3 Create An SAS Profile

If you don't have a SAS profile, click the link ' Don't have a SAS profile?`, you will have the following pop-up dialogue box. Click`Create profile', then you will see a pop-up sign-up page. You then follow the direction to create your SAS profile.

3.7.4 Log Into SAS Academic OnDemand

Provide your profile information to log into the OnDemand page, you will see the link to the SAS Studio user interface and your account information as well.

Once you create a SAS profile, you will have 5 GB of free storage.

3.7.5 SAS Studio User Interfacce

In the `Applications` tab, click `SAS Studio`, and you see the SAS Studio user interface on a separate page (it may take a little bit of time to initialize your account if you use it for the first time).

The above screenshot was taken from my SAS course webpage. For those who learned SAS using the classical SAS, you will see SAS Studio is much more convenient and easier to use.

3.7.6 A Cautionary Note On Data Security

SAS Studio (Academic OnDemand) is installed on SAS servers hosted in the Microsoft Azure Cloud. Although SAS claims that your assigned storage is private and secured, it is suggested to avoid uploading `sensitive data` to your private storage on the SAS server since SAS does not release the level of security for the storage.

3.8 R Viz Libraries

The following libraries will be used throughout this class.

3.8.1 Tidyverse

3.8.2 ggplot2

Ggplot2 is a system for creating charts based on the Grammar of Graphics. It proved to be one of the most powerful R libraries for visualization.

3.8.3 plotly

`plotly` is an online platform for data visualization in R (also available in Python). This package creates interactive web-based plots using `plotly.js` library. Plotly gives users an opportunity to interact with graphs, change their scale and point out the necessary record. The library also supports graph hovering. Moreover, one can easily add Plotly in knitr/R Markdown or Shiny apps.

3.8.4 leaflet

Leaflet is a well-known package based on JavaScript libraries for interactive maps. It is widely used for mapping and working with the customization and design of interactive maps. Besides, Leaflet provides an opportunity to make these maps mobile-friendly.

3.8.5 mapview

3.8.6 tmap

3.8.7 Other infrequently used packages

`ggmap`, `map`, `dygraph`,

Chapter 4

R Functions and Flow Controls

This chapter reviews the fundamentals of R programming: Functions and flow controls.

4.1 Control flow

There are two primary tools of control flow: choices and loops.

Choices, like if statements and `switch()` calls, allow us to run different code depending on the input.

Loops, like for and while, allow us to repeatedly run code, typically with changing options.

What is the difference between **if** and **ifelse()**?

4.1.1 Choices

The basic form of an if statement in R is as follows:

if (condition) true_action if (condition) true_action else false_action If the condition is TRUE, true_action is evaluated; if the condition is FALSE, the optional false_action is evaluated.

Typically the actions are compound statements contained within {:

```
grade <- function(x) {
  if (x > 90) {
    "A"
  } else if (x > 80) {
```

```
    "B"
  } else if (x > 50) {
    "C"
  } else {
    "F"
  }
}
```

`if` returns a value so that you can assign the results:

```
x1 <- if (TRUE) 1 else 2
x2 <- if (FALSE) 1 else 2
c(x1, x2)
```

```
## [1] 1 2
```

When we use the single argument form without an else statement, it invisibly returns NULL if the condition is FALSE. Since functions like `c()` and `paste()` drop NULL inputs, this allows for a compact expression of certain idioms:

```
greet <- function(name, birthday = FALSE) {
  paste0(
    "Hi ", name,
    if (birthday) " and HAPPY BIRTHDAY"
  )
}
greet("Maria", FALSE)
```

```
## [1] "Hi Maria"
```

```
greet("Jaime", TRUE)
```

```
## [1] "Hi Jaime and HAPPY BIRTHDAY"
```

Invalid inputs

The condition should evaluate to a single TRUE or FALSE. Most other inputs will generate an error:

```
#if ("x") 1
#> Error in if ("x") 1: argument is not interpretable as logical
#if (NA) 1
#> Error in if (NA) 1: missing value where TRUE/FALSE needed
```

Vectorized if

Given that it only works with a single TRUE or FALSE, you might wonder what to do if you have a vector of logical values. Handling vectors of values is the job of `ifelse()`: a vectorized function with test, yes, and no vectors (that will be recycled to the same length):

```
x <- 1:10
ifelse(x %% 5 == 0, "XXX", as.character(x))
```

```
##  [1] "1"   "2"   "3"   "4"   "XXX" "6"   "7"   "8"   "9"   "XXX"
```

```
ifelse(x %% 2 == 0, "even", "odd")
```

```
##  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

Note that missing values will be propagated into the output.

Another vectorized equivalent is the more general *dplyr::case_when()*. It uses a special syntax to allow any number of condition-vector pairs:

```
dplyr::case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  is.na(x) ~ "???",
  TRUE ~ as.character(x)
)
```

```
##  [1] "1"    "2"    "3"    "4"    "fizz" "6"    "buzz" "8"    "9"    "fizz"
```

```
9 %/% 5     ## quotient
```

```
## [1] 1
```

```
9%%5     ## remainder
```

```
## [1] 4
```

switch() statement

Closely related to it is the switch()-statement. It's a compact, special-purpose equivalent that lets you replace code like:

```
x_option <- function(x) {
  if (x == "a") {
    "option 1"
  } else if (x == "b") {
    "option 2"
  } else if (x == "c") {
    "option 3"
  } else {
    stop("Invalid `x` value")
  }
}
```

with the more succinct:

```
x_option <- function(x) {
  switch(x,
    a = "option 1",
    b = "option 2",
    c = "option 3",
    stop("Invalid `x` value")
  )
}
```

The last component of a `switch()` should always throw an error, otherwise, unmatched inputs will invisibly return NULL:

```
(switch("c", a = 1, b = 2))
```

```
## NULL
```

If multiple inputs have the same output, you can leave the right-hand side of = empty and the input will "fall through" to the next value. This mimics the behavior of C's switch statement:

```
legs <- function(x) {
  switch(x,
    cow = ,
    horse = ,
    dog = 4,
    human = ,
    chicken = 2,
    plant = 0,
    stop("Unknown input")
  )
}
legs("cow")
```

```
## [1] 4
```

```
legs("dog")
```

```
## [1] 4
```

It is also possible to use `switch()` with a numeric x, but is harder to read, and has undesirable failure modes if x is not a whole number.

4.1.2 Loops

`for loops` are used to iterate over items in a vector. They have the following basic forms:

```
for (item in vector) perform_action
```

For each item in the vector, perform_action is called once; updating the value of the item each time.

```
for (i in 1:3) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

`for` assigns the item to the current environment, overwriting any existing variable with the same name:

```
i <- 100
for (i in 1:3) {}
i
```

```
## [1] 3
```

There are two ways to terminate a `for loop` early:

- `next` exits the current iteration.
- `break` exits the entire for a loop.

```
for (i in 1:10) {
  if (i < 3)
    next

  print(i)

  if (i >= 5)
    break
}
```

```
## [1] 3
## [1] 4
## [1] 5
```

4.2 Functions

We have already created R functions and know how to use them to reduce duplication in our code. In this note, we'll learn how to turn that informal, working knowledge into a more rigorous understanding of R functions.

4.2.1 Function fundamentals

To understand functions in R we need to internalize two important ideas:

Functions can be broken down into three components: arguments, body, and environment. Functions are objects, just as vectors are objects.

4.2.1.1 Function components

A function has three parts:

- `The formals()`, the list of arguments that control how you call the function.
- `The body()`, the code inside the function.
- `The environment()`, the data structure that determines how the function finds the values associated with the names.

While the formals and body are specified explicitly when you create a function, the environment is specified implicitly, based on where you defined the function. The function environment always exists, but it is only printed when the function isn't defined in the global environment.

```
f02 <- function(x, y) {
  x + y
}
```

```
formals(f02)
```

```
## $x
##
##
## $y
```

```
body(f02)
```

```
## {
##     x + y
## }
```

```
environment(f02)
```

```
## <environment: R_GlobalEnv>
```

4.2.1.2 First-class functions

It's very important to understand that R functions are objects in their own right, a language property often called "first-class functions". Unlike in many other languages, there is no special syntax for defining and naming a function: we simply create a function object (with function) and bind it to a name with <-:

```
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```

While you almost always create a function and then bind it to a name, the binding step is not compulsory. If you choose not to give a function a name, you get an anonymous function. This is useful when it's not worth the effort to figure out a name:

```
lapply(mtcars, function(x) length(unique(x)))
```

```
## $mpg
## [1] 25
##
## $cyl
## [1] 3
##
## $disp
## [1] 27
##
## $hp
## [1] 22
##
## $drat
## [1] 22
##
## $wt
## [1] 29
##
## $qsec
## [1] 30
##
## $vs
## [1] 2
##
## $am
## [1] 2
##
## $gear
## [1] 3
##
## $carb
## [1] 6
```

```
Filter(function(x) !is.numeric(x), mtcars)
```

```
## data frame with 0 columns and 32 rows
```

```
integrate(function(x) sin(x) ^ 2, 0, pi)$value
```

```
## [1] 1.570796
```

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
iris$Species
```

```
##   [1] setosa     setosa     setosa     setosa     setosa     setosa     setosa     s
##  [11] setosa     setosa     setosa     setosa     setosa     setosa     setosa     s
##  [21] setosa     setosa     setosa     setosa     setosa     setosa     setosa     s
##  [31] setosa     setosa     setosa     setosa     setosa     setosa     setosa     s
##  [41] setosa     setosa     setosa     setosa     setosa     setosa     setosa     s
##  [51] versicolor versicolor versicolor versicolor versicolor versicolor versicolor v
##  [61] versicolor versicolor versicolor versicolor versicolor versicolor versicolor v
##  [71] versicolor versicolor versicolor versicolor versicolor versicolor versicolor v
##  [81] versicolor versicolor versicolor versicolor versicolor versicolor versicolor v
##  [91] versicolor versicolor versicolor versicolor versicolor versicolor versicolor v
## [101] virginica  virginica  virginica  virginica  virginica  virginica  virginica  v
## [111] virginica  virginica  virginica  virginica  virginica  virginica  virginica  v
## [121] virginica  virginica  virginica  virginica  virginica  virginica  virginica  v
## [131] virginica  virginica  virginica  virginica  virginica  virginica  virginica  v
## [141] virginica  virginica  virginica  virginica  virginica  virginica  virginica  v
## Levels: setosa versicolor virginica
```

A final option is to put functions in a list:

```
funs <- list(
  half = function(x) x / 2,
  double = function(x) x * 2
)

funs$double(10)
```

```
## [1] 20
```

```
funs$half(10)
```

```
## [1] 5
```

4.2.2 Function Composition

Base R provides two ways to compose multiple function calls. For example, imagine we want to compute the population standard deviation using `sqrt()` and `mean()` as building blocks:

```
square <- function(x) x^2
square(7)
```

```
## [1] 49
```

```
sqrt0 = function(x){
  x^2
}
##
sqrt0(7)
```

```
## [1] 49
```

```
deviation <- function(x) x - mean(x)
```

You either nest the function calls:

```
x <- runif(100)
sqrt(mean(square(deviation(x))))
```

```
## [1] 0.2936577
```

Or we save the intermediate results as variables:

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
```

```
## [1] 0.2936577
```

The `magrittr` package provides a third option: the binary operator `%>%`, which is called the pipe and is pronounced as `"and then"`.

```
library(magrittr)
square <- function(x) x^2
deviation <- function(x) x - mean(x)
x <- 1:50
##
x %>%
  deviation() %>%  # deviation () is defined function above
  square() %>%
  mean() %>%
  sqrt()
```

```
## [1] 14.43087
```

pipe operator %>%:`x %>% f()` is equivalent to f(x); `x %>% f(y)` is equivalent to f(x, y).

4.2.2.1 Infix functions

Infix functions get their name from the fact the function name comes in between its arguments and hence has two arguments. R comes with a number of built-in infix operators:

- `:` - an operator that generates a patterned sequence. It is also used to indicate an `interaction` of two variables.
- `::` - an operator to access an object in a known package. For example, `stats::sd`.
- `:::` - an operator to access an object in a package - it is rarely used.,

- `$` - extracts elements by name from a named list.
- `^` - exponential operator (to-the-power-of)

- `*` - multiplication (operator)
- `/` - division (operator)
- `+` - addition (operator)
- `-` - subtraction (operator)
- `>` - logical and numerical GREATER THAN

- `>=` - logical and numerical GREATER OR EQUAL TO
- `<` - logical and numerical EQUAL TO
- `<=` - logical and numerical LESS THAN OR EQUAL TO
- `==` - logical EQUAL TO
- `!=` - logical and numerical NOT EQUAL TO
- `!` - logical negation (NOT)
- `&` - logical AND (element-wise).
- `&&` - logical AND.
- `|` - logical OR (element-wise).
- `||` - logical OR.
- `~` - operator used in the formation of a model
- `<-` - leftwards assignment
- `<<-`- leftwards assignment (used for assigning to variables in the parent environments)
- `->` - rightwards assignment
- `->>` - rightwards assignment (used for assigning to variables in the parent environments)
- `%%` - modulus (Remainder from division)
- `%/%` - integer Division

We can also create your own infix functions that start and end with %. Base R uses this pattern to define %%, %*%, %/%, %in%, %o%, and %x%.

Defining our own infix function is simple. We create a two-argument function and bind it to a name that starts and ends with `%`. For example,

```
`%+%` <- function(a, b) paste0(a, b)
"new " %+% "string"
```

```
## [1] "new string"
```

The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters except for %. You will need to escape any special characters in the string used to define the function, but not when you call it:

```
`% %` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)
```

```
"a" % % "b"
```

```
## [1] "a b"
```

```
"a" %/\% "b"
```

```
## [1] "a b"
```

R's default precedence rules mean that infix operators are composed left to right.

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
```

```
## [1] "((a %-% b) %-% c)"
```

Chapter 5

Getting Started with Base R Graphics

The two core plotting and graphics packages in the base R are:

- **graphics**: contains plotting functions for the "base" graphing systems, including plot, hist, boxplot, and many others.
- **grDevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

The `grDevices` package contains the functionality for sending plots to various output devices. The `graphics` package contains the code for actually constructing and annotating plots.

In this note, we focus on using the base plotting system to create graphics on the screen device.

5.1 Base Graphics

Base graphics are used most commonly and are a very powerful system for creating data graphics. There are two phases to creating a base plot:

- Initializing a new plot.
- Annotating (adding to) an existing plot.

For example, calling the base plot functions `plot(x, y)` or `hist(x)` will launch a graphics device (if one is not already open) and draw a new plot on the device. The based plot function has many arguments, letting us set the title, x-axis label, y-axis label, etc.

The base graphics system has many global parameters that can be set and tweaked. These parameters are documented in `?par` and are used to control the global behavior of plots, such as the margins, axis orientation, and other details.

5.2 Simple Base Graphics

This section explains how to use the base plotting functions to make basic statistical graphics.

5.2.1 Histogram

Here is an example of a simple histogram made using the `hist()` function in the graphics package. If we run this code and our graphics window is not already open, it should open once you call the `hist()` function.

```
## Draw a new plot on the screen device
hist(iris$Sepal.Length,
     xlab = "Sepal Length",
     ylab = "Counts",
     main = "Frequency distribution of sepal length")
```

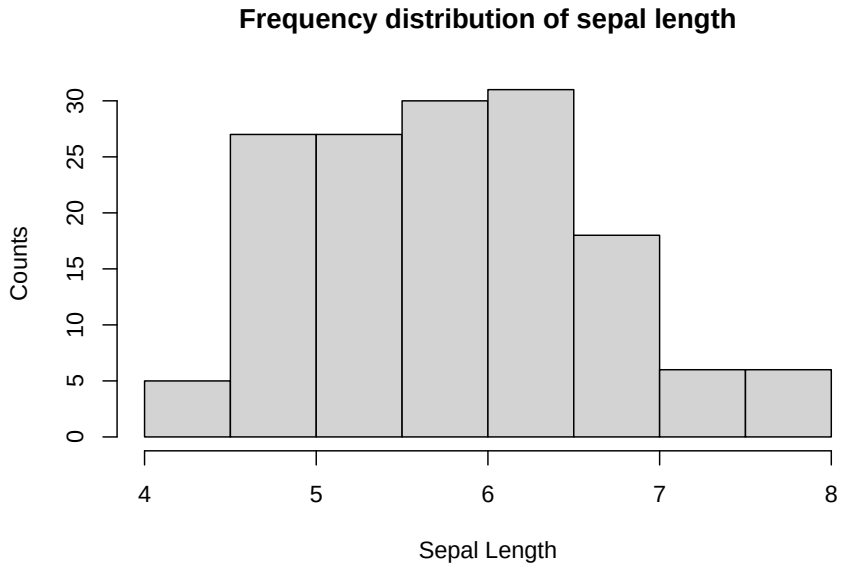


Figure 5.1: Figure 1. Sepal width of iris

5.2.2 Boxplot

Boxplots can be made in R using the `boxplot()` function, which takes as its first argument a formula. The formula has a form of `y-axis ~ x-axis`. Anytime you see a `~` in R, it's a formula. Here, we are plotting the sepal length of iris flowers and the right-hand side of `~` the Species.

```
boxplot(Sepal.Length ~ Species, data = iris, xlab = "Species", ylab = "Sepal Length")
```

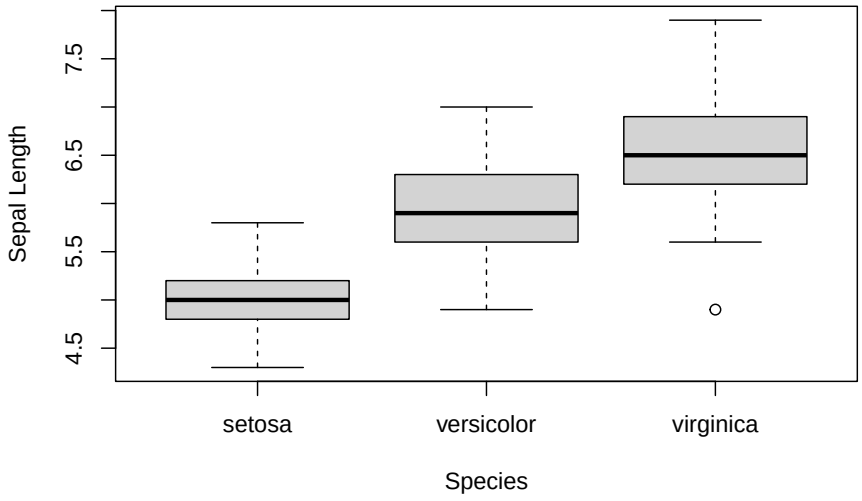


Figure 5.2: Box plot of sepal length by species

Each boxplot shows the median, 25th, and 75th percentiles of the data (the "box"), as well as +/- 1.5 times the interquartile range (IQR) of the data (the "whiskers"). Any data points beyond 1.5 times the IQR of the data are indicated separately with circles.

We can see that Virginica has an outlier.

5.2.3 Scatterplot

Here is a simple scatter plot made with the `plot()` function.

```
plot(iris$Sepal.Length, iris$Sepal.Width,
     xlab = "sepal length",
     ylab = "sepal width",
     main = " ")
```

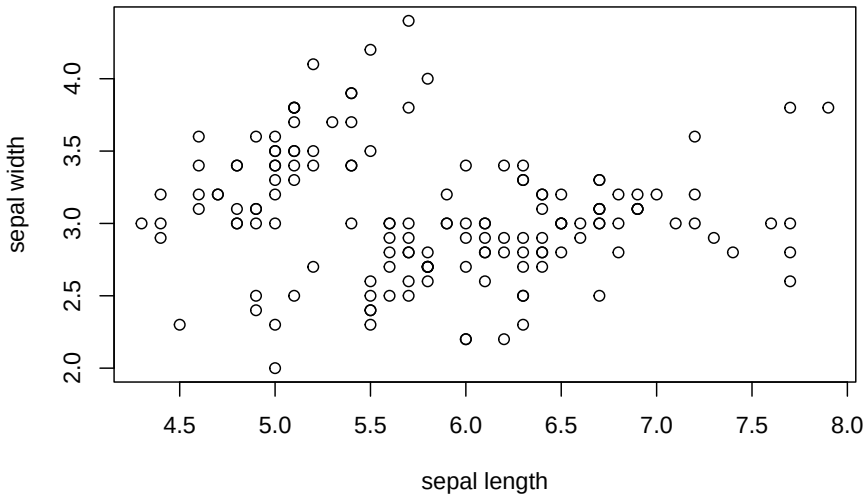


Figure 5.3: Scatter plot of sepal length and sepal width

Generally, the `plot()` function takes two vectors of numbers: one for the x-axis coordinates and one for the y-axis coordinates. However, `plot()` is a generic function in R, which means its behavior can change depending on what kinds of data are passed to the function.

One thing to note here is that we have provided labels for the x- and the y-axis and title as well. If they were not specified, the plot function will provide this information automatically using the names of the variables.

5.3 Some Important Base Graphics Parameters

Many base plotting functions share a set of global parameters. Here are a few key ones.

- `pch`: the plotting symbol (default is an open circle).
- `lty`: the line type (default is a solid line), can be dashed, dotted, etc.
- `lwd`: the line width, specified as an integer multiple.
- `col`: the plotting color, specified as a number, string, or hex code; the colors() function gives you a vector of colors by name.
- `xlab`: character string for the x-axis label.
- `ylab`: character string for the y-axis label.

The `par()` function is used to specify the global graphics parameters that affect

all plots in an R session. These parameters can be overridden when they are specified as arguments to specific plotting functions.

- `las`: the orientation of the axis labels on the plot.
- `bg`: the background color.
- `mar`: the margin size.
- `oma`: the outer margin size (default is 0 for all sides).
- `mfrow`: number of plots per row, column (plots are filled row-wise).
- `mfcol`: number of plots per row, column (plots are filled column-wise).

We can see the default values for global graphics parameters by calling the `par()` function and passing the name of the parameter in quotes.

```
par("lty")
```

```
## [1] "solid"
```

```
par("col")
```

```
## [1] "black"
```

```
par("pch")
```

```
## [1] 1
```

Here are some more default values for global graphics parameters.

```
par("bg")
```

```
## [1] "transparent"
```

```
par("mar")
```

```
## [1] 5.1 4.1 4.1 2.1
```

```
par("mfrow")
```

```
## [1] 1 1
```

For the most part, we usually don't have to modify these when making quick plots. However, we might need to tweak them to finalize the plots.

5.4 Base Plotting Functions

The most basic base plotting function is `plot()`. The `plot()` function makes a scatter-plot, or other types of plot depending on the class of the object being plotted. Calling `plot()` will draw a plot on the screen device (and open the screen device if not already open). After that, annotation functions can be called to add to the already-made plot.

Some key annotation functions are

- `lines()`: add lines to a plot, given a vector of x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots
- `points()`: add points to a plot
- `text()`: add text labels to a plot using specified x, y coordinates
- `title()`: add annotations to x, y-axis labels, title, subtitle, outer margin
- `mtext()`: add arbitrary text to the margins (inner or outer) of the plot
- `axis()`: adding axis ticks/labels

Here's an example of creating a base plot and adding some annotation. First, we make the plot with the `plot()` function and then add a title to the top of the plot with the `title()` function.

```
## Make the initial plot
plot(iris$Sepal.Length, iris$Sepal.Width)
## Add a title
title(main = "Sepal length and width of iris")
```

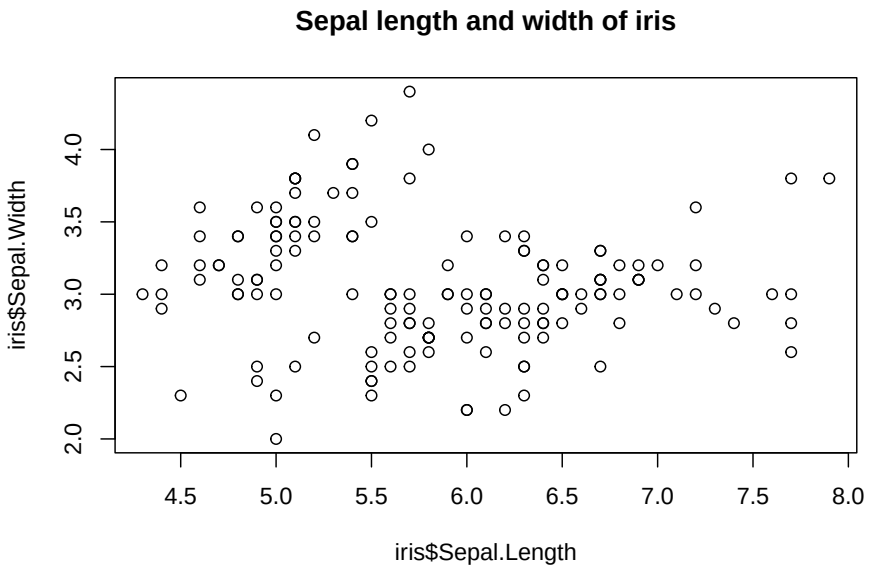
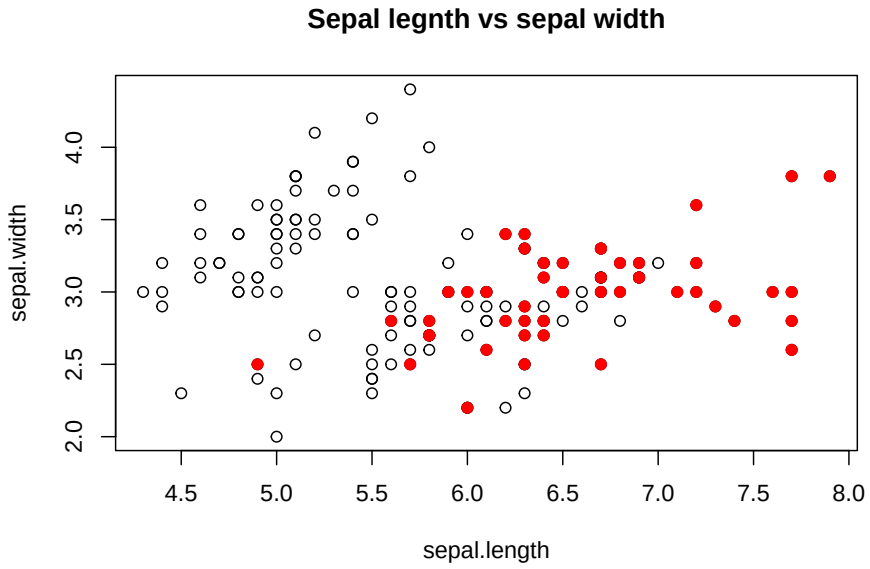


Figure 5.4: Base plot with annotation: Iris

Here, I start with the same plot as above (although I add the title right away using the main argument to `plot()`) and then annotate it by coloring the data points corresponding.

```
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
```

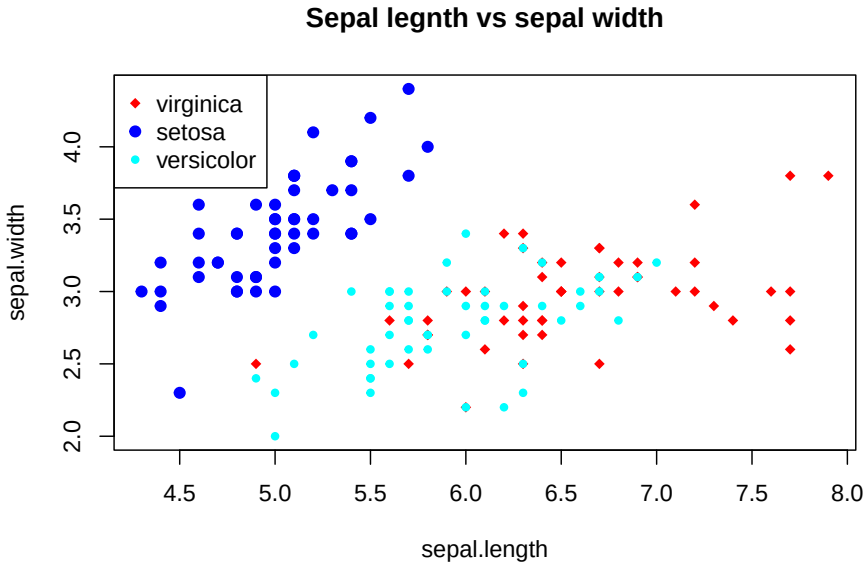
```
species = iris$Species
## identifying the ID of Virginica
virginca.id = which(species=="virginica")  # value are case sensitive!
## making scatter plot
plot(sepal.length, sepal.width, main = "Sepal legnth vs sepal width")
##
points(sepal.length[virginca.id], sepal.width[virginca.id], pch = 19, col = "red")
```



The following plot colors the data points with different colors based on species. `legend()` function explains the meaning of the different colors in the plot.

```
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
species = iris$Species
## identifying the ID of Virginica
virginca.id = which(species=="virginica")  # value are case sensitive!
setosa.id = which(species=="setosa")
versicolor.id = which(species=="versicolor")
## making an empty plot: type = "n" ==> no point
plot(sepal.length, sepal.width, main = "Sepal legnth vs sepal width", type = "n")
##
points(sepal.length[virginca.id], sepal.width[virginca.id], pch = 18, col = "red")
points(sepal.length[setosa.id], sepal.width[setosa.id], pch = 19, col = "blue")
points(sepal.length[versicolor.id], sepal.width[versicolor.id], pch = 20, col = "cyan")
legend("topleft", c("virginica", "setosa", "versicolor"),
```

```
                col=c("red", "blue", "cyan"),
                pch=c(18, 19, 20))
```



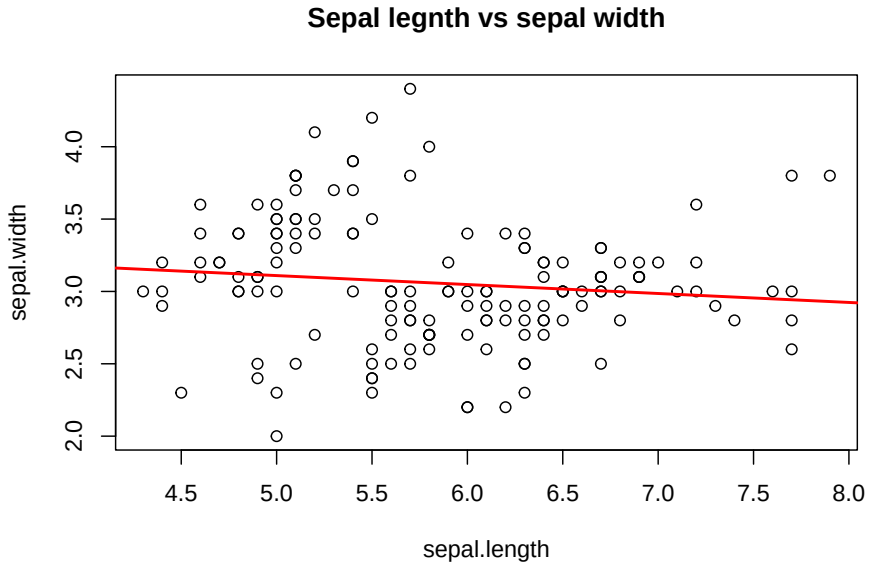
5.5 Base Plot with Regression Line

It's fairly common to make a scatterplot and then want to draw a simple linear regression line through the data. This can be done with the `abline()` function.

Below, we first make the plot (as above). Then we fit a simple linear regression model using the `lm()` function. Here, we try to model Ozone as a function of Wind. Then we take the output of `lm()` and pass it to the `abline()` function which automatically takes the information from the model object and calculates the corresponding regression line.

Note that in the call to `plot()` below, we set `pch = 20` to change the plotting symbol to a filled circle.

```
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
## making a plot
plot(sepal.length, sepal.width, main = "Sepal legnth vs sepal width", pch = 21)
##
 model <- lm( sepal.width ~ sepal.length)
 ## Draw regression line on plot
 abline(model, lwd = 2, col = "red")
```



5.6 Multiple Base Plots

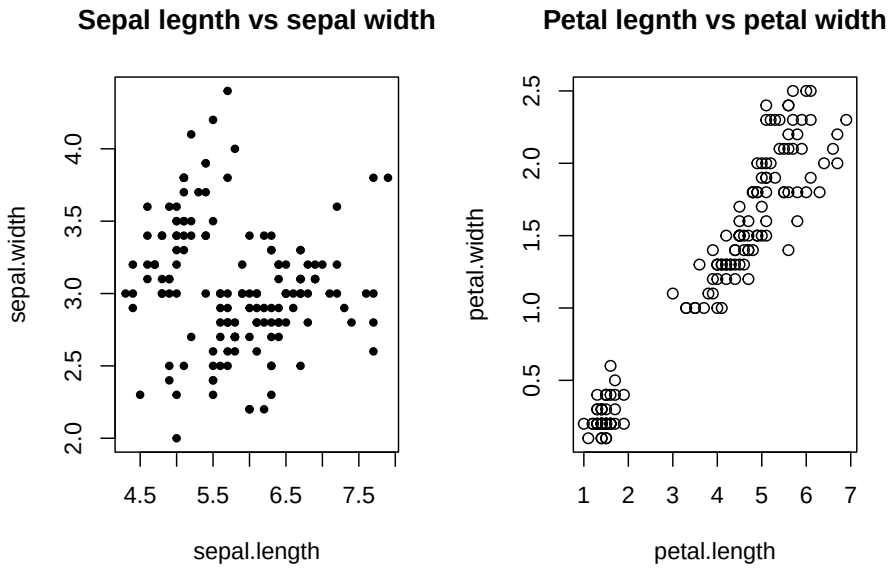
Making multiple plots side by side is a useful way to visualize many relationships between variables with static 2-D plots. Often the repetition of data across a single plot window can be a useful way to identify patterns in the data. In order to do this, the `mfrow` and `mfcol` parameters set by the `par()` function are critical.

Both the `mfrow` and `mfcol` parameters take two numbers: the number of rows of plots followed by the number of columns. The multiple plots will be arranged in a matrix-like pattern. The only difference between the two parameters is that if `mfrow` is set, then the plots will be drawn row-wise; if `mfcol` is set, the plots will be drawn column-wise.

In the example below, we make two plots: sepal length vs sepal width and petal length vs petal width. We set `par(mfrow = c(1, 2))`, which indicates that we have one row of plots and two columns of plots.

```
par(mfrow = c(1, 2))
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
petal.length = iris$Petal.Length
petal.width = iris$Petal.Width
## making a plot
plot(sepal.length, sepal.width, main = "Sepal legnth vs sepal width", pch = 20)
```

```
plot(petal.length, petal.width, main = "Petal legnth vs petal width", pch = 21)
```

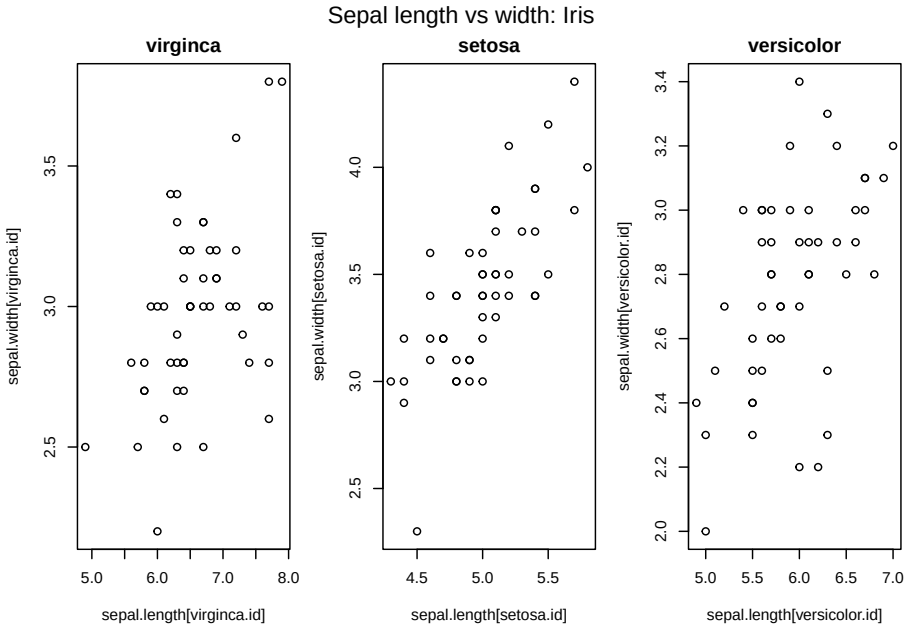


The example below creates three plots in a row by setting `par(mfrow = c(1, 3))`. Here we also change the plot margins with the mar parameter. The various margin parameters, like `mar`, are specified by setting a value for each side of the plot. `Side 1` is the bottom of the plot, `side 2` is the left-hand side, `side 3` is the top, and `side 4` is the right-hand side.

In the example below we also modify the outer margin via the `oma` parameter to create a little more space for the plots and to place them closer together.

```
# layout of the plot
par(mfrow = c(1, 3),
    mar = c(4, 4, 2, 1),
    oma = c(0, 0, 2, 0))
# extract variables from the data frame
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
species = iris$Species
## identifying the ID of Virginica
virginca.id = which(species=="virginica")  # value are case sensitive!
setosa.id = which(species=="setosa")
versicolor.id = which(species=="versicolor")
## making three plots
plot(sepal.length[virginca.id], sepal.width[virginca.id], main = "virginca")
plot(sepal.length[setosa.id], sepal.width[setosa.id], main = "setosa")
```

```
plot(sepal.length[versicolor.id], sepal.width[versicolor.id], main = "versicolor")
##
mtext("Sepal length vs width: Iris", outer = TRUE)
```



In the above example, the `mtext()` function was used to create an overall title for the panel of plots. Hence, each individual plot has a title, while the overall set of plots also has a summary title. The `mtext()` function is important for adding text annotations that aren't specific to a single plot.

Notice that the scales of the vertical axes of the three plots are not the same. To misleading visual comparison, we should make the scales of both axes the same.

```
# layout of the plot
par(mfrow = c(1, 3),
    mar = c(4, 4, 2, 1),
    oma = c(0, 0, 2, 0))
# extract variables from the data frame
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
species = iris$Species
## identifying the ID of Virginica
virginca.id = which(species=="virginica")  # value are case sensitive!
setosa.id = which(species=="setosa")
versicolor.id = which(species=="versicolor")
## making three plots
```

```
plot(sepal.length[virginca.id], sepal.width[virginca.id], xlim=c(4,8), ylim=c(2, 4.5),
plot(sepal.length[setosa.id], sepal.width[setosa.id], xlim=c(4,8), ylim=c(2, 4.5),main
plot(sepal.length[versicolor.id], sepal.width[versicolor.id], xlim=c(4,8), ylim=c(2, 4
##
mtext("Sepal length vs width: Iris", outer = TRUE)
```

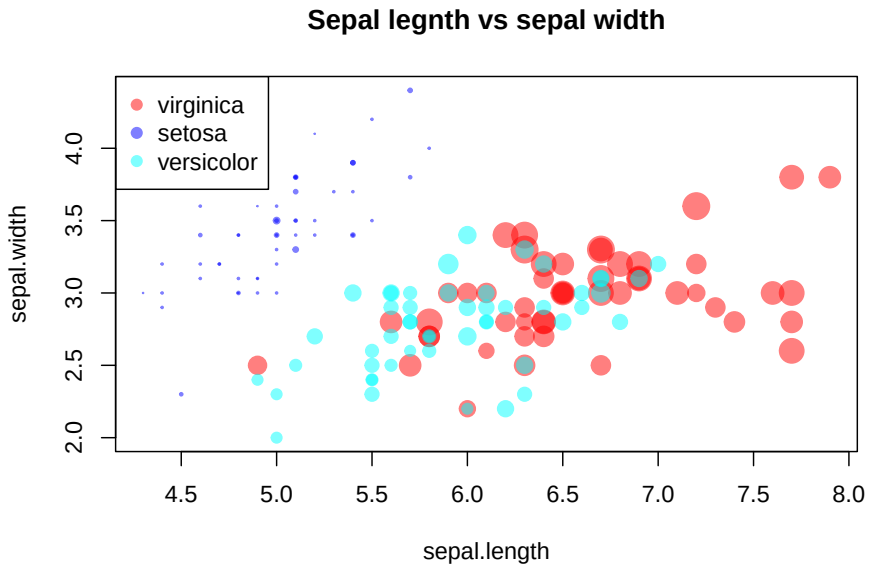
We now can visualize the mean sepal width and sepal length among the species.

5.7 Controlling Point Size and Transparency

we use `cex=` and `alpha=` to control the point size according to the value of a variable and the level of transparency of the point.

```
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
size.petal = iris$Petal.Width
species = iris$Species
## identifying the ID of Virginica
virginca.id = which(species=="virginica")  # value are case sensitive!
setosa.id = which(species=="setosa")
versicolor.id = which(species=="versicolor")
## color code
col.code = c(alpha("red",0.5),alpha("blue",0.5),alpha("cyan",0.5))
## making an empty plot: type = "n" ==> no point
plot(sepal.length, sepal.width, main = "Sepal legnth vs sepal width", type = "n")
```

```
## change the point size based on their average of sepal length and width
points(sepal.length[virginca.id], sepal.width[virginca.id],
       pch = 19, col = col.code[1], cex = size.petal[virginca.id])
points(sepal.length[setosa.id], sepal.width[setosa.id],
       pch = 19, col = col.code[2], cex = size.petal[setosa.id])
points(sepal.length[versicolor.id], sepal.width[versicolor.id],
       pch = 19, col = col.code[3], cex = size.petal[versicolor.id])
legend("topleft", c("virginica", "setosa", "versicolor"),
                  col=col.code,
                  pch=c(19, 19, 19))
```

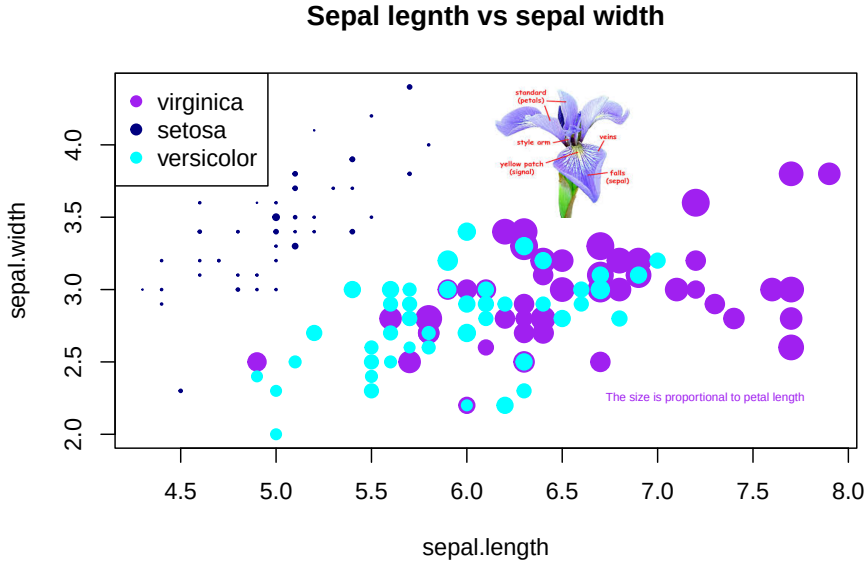


5.8 Annotations

We add annotations to the base R plot. The annotations could be plain texts, images, and mathematical expressions.

```
library(scales)
my_iris <- readPNG('img02/irisPNG.png')
raster.iris <- as.raster(my_iris)
# Use the code in the precious section
sepal.length = iris$Sepal.Length
sepal.width = iris$Sepal.Width
size.petal = iris$Petal.Width
species = iris$Species
```

```
## identifying the ID of Virginica
virginca.id = which(species=="virginica")  # value are case sensitive!
setosa.id = which(species=="setosa")
versicolor.id = which(species=="versicolor")
## color code
#  col.code = c(alpha("red",0.5),alpha("blue",0.5),alpha("cyan",0.5))
## making an empty plot: type = "n" ==> no point
plot(sepal.length, sepal.width, main = "Sepal legnth vs sepal width", type = "n")
## change the point size based on their average of sepal length and width
points(sepal.length[virginca.id], sepal.width[virginca.id],
       pch = 19, col = "purple", cex = size.petal[virginca.id], alpha = 0.6)
points(sepal.length[setosa.id], sepal.width[setosa.id],
       pch = 19, col = "navy", cex = size.petal[setosa.id], alpha = 0.6)
points(sepal.length[versicolor.id], sepal.width[versicolor.id],
       pch = 19, col = "cyan", cex = size.petal[versicolor.id], alpha = 0.6)
legend("topleft", c("virginica", "setosa", "versicolor"),
                  col=c("purple", "navy", "cyan"),
                  pch=c(19, 19, 19))
## various annotations
#specify the position of the image through bottom-left and top-right coords
rasterImage(raster.iris,6,3.5,7,4.4)
text(7.25, 2.25, "The size is proportional to petal length", col = "purple", cex = 0.5)
```




```
detach(package:scales,unload=TRUE)
```

Chapter 6

Foundations of Data Visualization

Foundations of Data Visualization
(A Brief Overview)

Cheng Peng

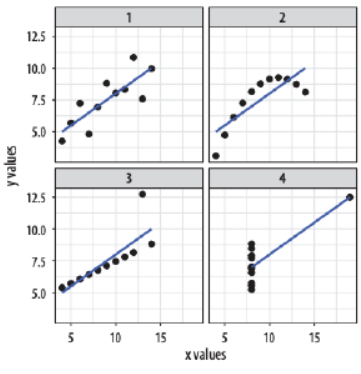
Department of Mathematics

WCU
WEST CHESTER
UNIVERSITY

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuasion
Steps for Good Charts

Why Visualization

- Plots of Anscombe's quartet.



Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuasion
Steps for Good Charts

Agenda

- Brief history of data viz.
- Overview of dataviz process
- Principles of dataviz
- Building blocks of dataviz.
- Color coding and usage guidelines
- A general framework for creating better charts
- Purpose and types of dataviz
- Specific visualization Skills

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Dataviz in past 50 Years

1970s - Foundation of modern dataviz
- John Tukey pioneers the use of visualization with computers (exploratory and confirmatory visualization).

1980s - The science of visualization
- Edward Tufte's work combines statistics with visual design principles
- Cleveland and McGill's work on measuring graphical perception
- Mackinlay's work carries visualization theories to digital age

1990s-2000s: The computer-driven scientific visualization thrives

2010s: The social internet, cheap and easy-to-use software, and massive volumes of data democratize the practice of visualization.
- Rensink and Harrison establish science around graphic perception

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Visualization Process

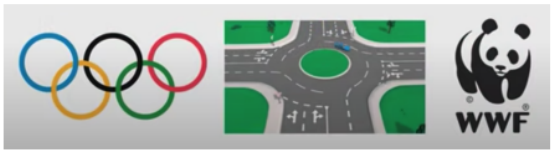
- Understand Visualization
  - Dataviz is a process
  - Dataviz is new language built on the science and art.
- Prepare Visualization
  - Manage and clean data
  - Talk and listening
  - Sketch and prototype
- Create Visualization
  - Is the information conceptual or data driven?
  - Am I declaring or exploring something?
  - Types of visualization
- Refine Visualization
  - to make impressive charts
  - to make persuasive charts
- Present and Practice Visualization
  - to persuade audience
  - to make better charts

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Gestalt Principles

The principles describe the various ways we tend to visually assemble individual objects into groups and are highly relevant to the design of charts and graphs.

- Objects will be perceived in their simplest form
- Humans naturally follow lines and curves
- The mind will attempt to fill in detail that isn't actually there.



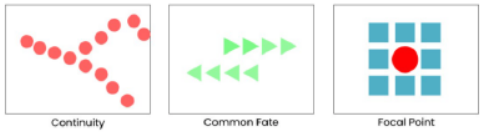
Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Gestalt Design Principles

- **Closure**: Elements are typically grouped together if they are a part of an entity
- **Proximity**: Elements are typically grouped together based on their immediacy
- **Similarity**: Elements similar to one another tend to be grouped together

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

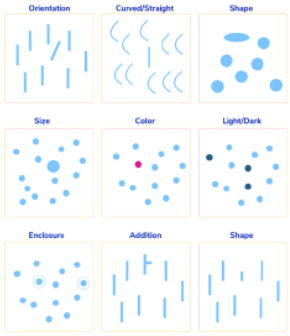
- **Continuity**: Elements that are arranged on a line or curve are perceived to be more related than elements not on the line or curve.
- **Common Fate**: When elements coordinate movement together, we tend to relate them to each other.
- **Focal point**: When an element or elements stands out visually, it captures and holds our attention first.



Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Pre-attentative Attributes

- A preattentive visual property is one which is processed in spatial memory without our conscious action.



- It takes less than 500 milliseconds for the eye and the brain to process a preattentive property

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Visual Encoding

- The visual encoding is the way in which data is mapped into visual structures made of marks and channels.
- Data visualization is the graphical representation of information and data built based on visual structures.
- Marks and Channels - Building blocks of visualization
  - **Marks** (geometric primitives) represent items or links - basic graphical element in an image.
  - **Channels** (aka channel variables) change appearance of marks based on attributes - independent of the dimensionality of the geometric primitive.
- By using marks and channels to create visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Marks

- Marks for items

Points Lines Areas

0D 1D 2D

- Marks for links

Containment Connection

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Channels

Control appearance proportional to or based on attributes.

Position
- Horizontal
- Vertical
- Both

Color

Shape

Tilt

Size
- Length
- Area
- Volume

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

Expressiveness & Effectiveness

Expressiveness

- visual encoding should express all of, and only, the information in the dataset attributes
- simple one - lie factor (the ratio of the information "in the chart" and the information "in data")

Effectiveness

- importance of the attribute should match the salience of the channel
- simple one - data-ink ratio (the ratio of "ink in data" and "ink in the chart")

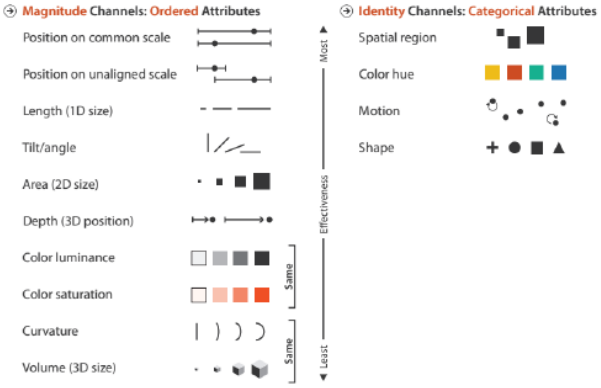
Chart Junk

- Unnecessary visual elements in charts that distracts the viewer from the information

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
13/31

More on Channels

More on Channels: Expressiveness Types and Efectiveness Ranks



Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
14/31

Color for Viz

- Color improves a chart's aesthetic quality, expressiveness, hence, its ability to effectively communicate about its data.
- **Categorical Pallet**: Categorical colors help users map non-numeric meaning to objects in a visualization.

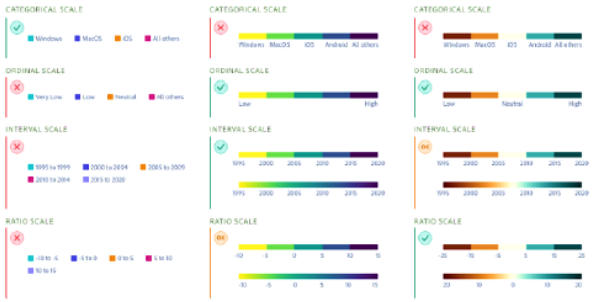
- **Sequential Pallet**: Sequential colors have numeric meaning.

- **Divergent Pallet**: Diverging colors also have numeric meaning.

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts

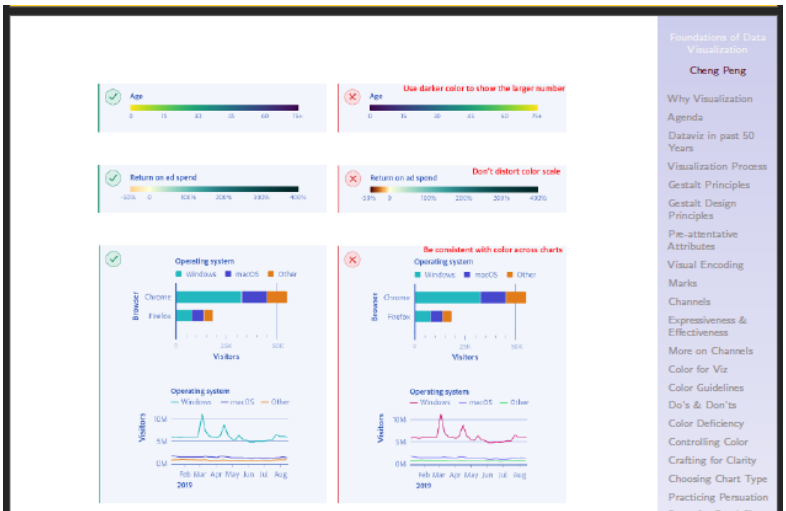
Color Guidelines

Guidelines of using colors with different types of data.



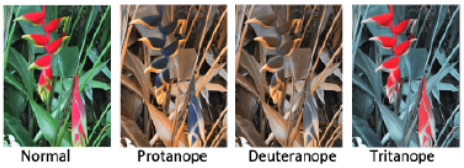
Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts


Do's & Don'ts
Use color to separate items into categories.
Don't use color to separate items.
More than 6 colors!
Should use categorical pallet
Should use sequential pallet
Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
17/31
18/31



Color Deficiency

- ABout 8% - 10 % of men and 1% of women have color vision deficiency.
- Red-green is common (deuteranope and protanope two subcategories).
- Blue-yellow is possible (tritanope is most common in this category)



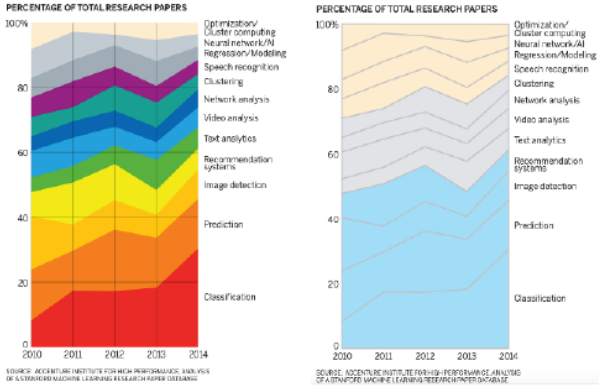
Controlling Color

Practical guidelines of using colors in dataviz.

- **Use less color** - keep the number of colors minimum.
- **Use gray** - It doesn't draw the eye the way stronger colors do and is the default color in software.
- **Complement / contrast** - When variables are inherently similar, use similar or complementary colors. When they are in opposition, use contrasting colors.
- **Stick to the variables** - using color for text decoration is distracting.
- **Think how, not which** - It is more important to think about how to use color than which color is used.
- **Consider the color-blind**.

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
21/31

Comparing poor chart and better chart.



Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
22/31

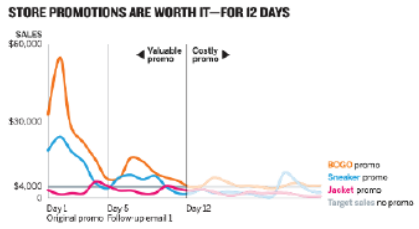
Crafting for Clarity

Some guidelines to achieve a clear design.

- **Take stuff away** - think about every mark on your chart and ask, Is this necessary to make your point?
- **Remove redundancy** - A headline that reads "Sales vs. Revenue" just repeats the axis labels.
- **Limit color and eye travel** - Color is powerful—and distracting.
- **Know how people think** - The brain works on heuristics. It is important to respect convention — and take advantage of it.
- **Describe ideas, not structure** - Use text, headlines, captions, and other visual markers to highlight ideas or insights rather than to describe the visualization's architecture.
- **Align everything** - This simple guideline is supremely effective at creating visual order.

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuasion
Steps for Good Charts

Comparing poor chart and better chart.



Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuasion
Steps for Good Charts

Choosing Chart Type

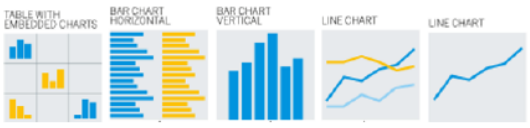
Guidelines for selecting appropriate chart type.

- **Know the basic categories** - The simplest way to begin is to understand your intent.
- **Listen to how you describe things** - Find someone to chat with about your data and the idea you want to convey.
- **Rely on your workhorses** - Understand that more specialized and unusual chart types will require more effort on the part of your viewers.
- **Don't forget tables** - Sometimes all the individual data points in a set matter more than a trend or what comprises them.
- **Good writers are great readers** - good chart makers are great chart consumers. It is important to find inspiration in others' visualizations to improve you visualizations.

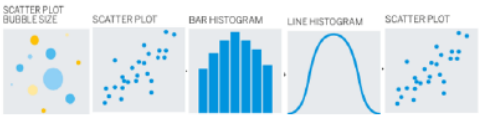
Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
25/31

According to the applications, there are four basic types of charts.

- **Comparisons** - Some keywords: before/after, categories compare, contrast over time, peaks, rank, trend, types, etc.

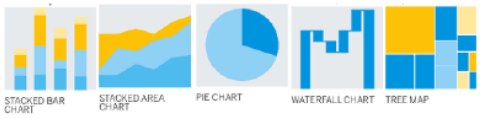


- **Comparisons and Distributions** - Some keywords: alluvial, cluster, distributed, from/to, plotted, points, spread, spread over, relative to, transfer, etc.



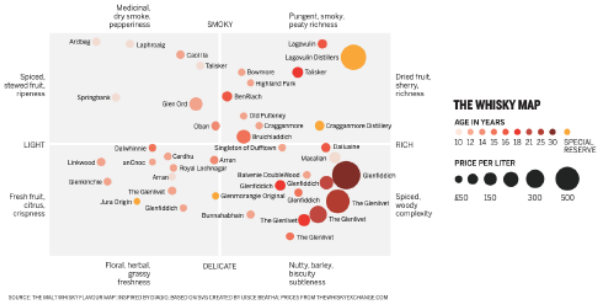
Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
26/31

- **Compositions** - Some Keywords: components, divvied up, group, makes up, of the whole, parts, percentage, pieces, portion, proportion, slices, subsections, total, etc.



- **Maps, Networks, and Logics** - Some keywords: cluster, complex connections, group, hierarchy, if/then, network, organize, paths, places , relationships, routes, structure, space, yes/no, etc.

Whisky data contains information of 42 records of whiskys brands. Six variables including age, cost, character, flavor, and region. Choose an appropriate chart to visualize the data.



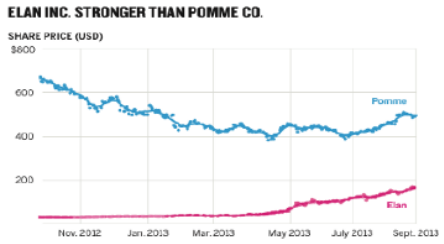
Data Source: https://raw.githubusercontent.com/pengdsci/sta553/main/DatavizPrinciple/whisky.csv

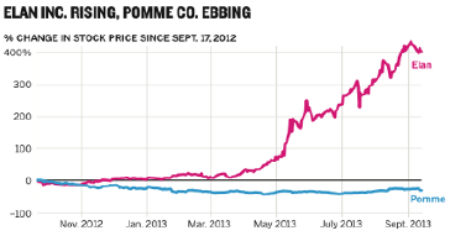
Practicing Persuation

The guidelines for building persuasion into your charts.

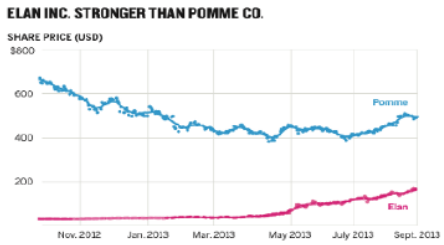
- **Shift the context question** - before making a chart, asking yourself what you try to say, to whom, and where.
- **Emphasize and isolate** - shine a bright light on the most salient information. Limit the number of places an audience can focus. Move their eyes to where you want them to go.
- **Consider your reference points** - The ultimate form of isolation is to remove any information that doesn't directly support your point. Try to avoid multiple interpretations.
- **Point things out** - It doesn't take much to move someone's eyes. Pointers, demarcations, and simple labels signal to an audience what matters.
- **Lure** - Upending expectations can be powerfully persuasive. Evidence to the contrary is challenging and will foster discussion: Here's what you think our data looks like; here's how it actually looks.

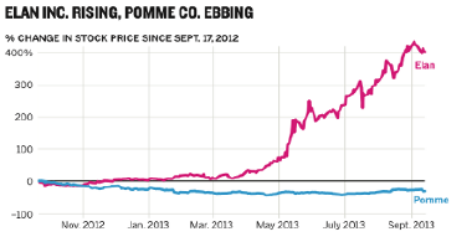
Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
29/31





Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
30/31





Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
30/31

Steps for Good Charts

Talk and Listen - Put aside your data and find someone in the domain to have a conversation to set your context. Address questions like

- Who is this for?
- What do you want them to do after seeing this?
- How will it be displayed?
- If you could show them only one thing, what would it be?
- Will it be surprising or affirming?

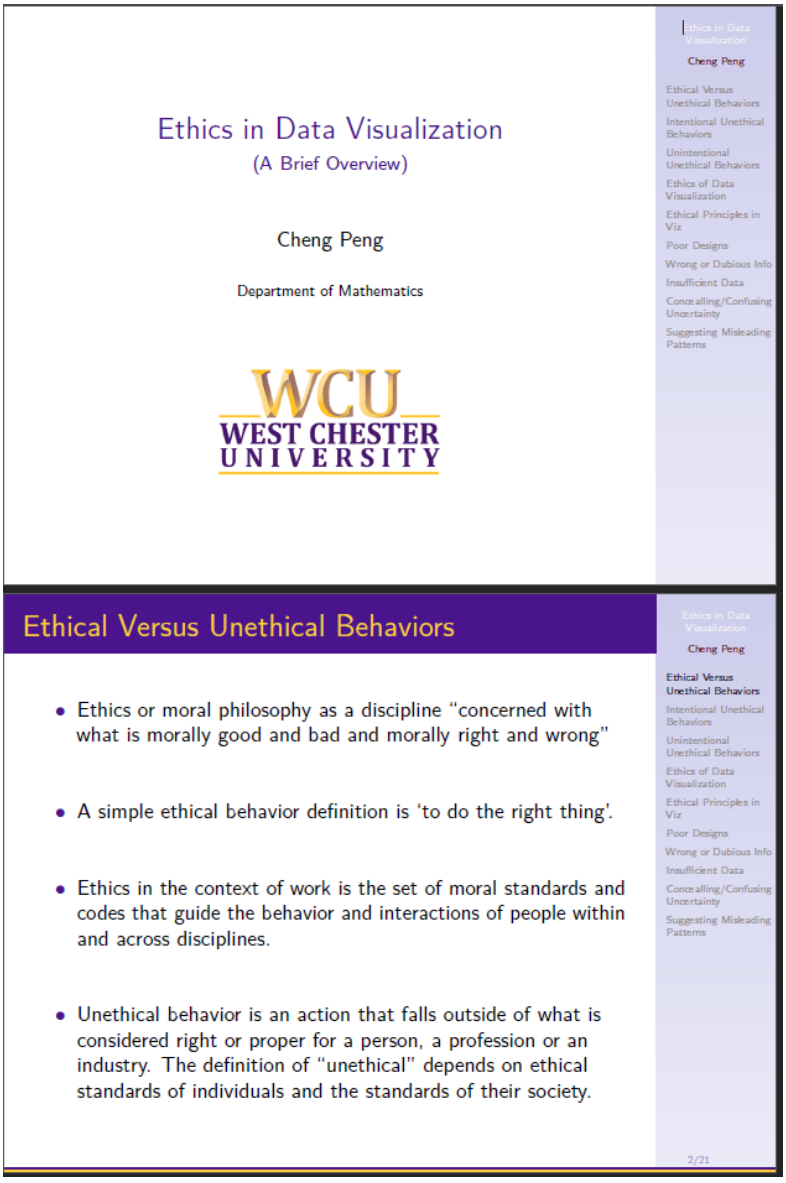
Sketch - As you're talking, start to sketch possible approaches. Go fast. The key is to keep moving. You want to be generative, creating ideas rapidly. Continue talking through the process, and as new ideas and visual words come up, jot them down.

Prototype - Whereas sketching is fast and open, prototyping is a bit slower and more deliberate. Use color purposefully. Sketching is generative; prototyping is iterative. Hone your chart until it approaches good.

Foundations of Data Visualization
Cheng Peng
Why Visualization
Agenda
Dataviz in past 50 Years
Visualization Process
Gestalt Principles
Gestalt Design Principles
Pre-attentative Attributes
Visual Encoding
Marks
Channels
Expressiveness & Effectiveness
More on Channels
Color for Viz
Color Guidelines
Do's & Don'ts
Color Deficiency
Controlling Color
Crafting for Clarity
Choosing Chart Type
Practicing Persuation
Steps for Good Charts
31/31

Chapter 7

Ethics in Data Visualization



Intentional Unethical Behaviors

- **Intentional unethical** behavior occurs when people engage in actions they know to be wrong but are unaware of the biases and forces affecting their judgments.
  1. An authority demands obedience rather than a person's character causes unethical behaviors.
  2. The more room a situation provides for people to be able to justify their behavior, the more likely they are to behave unethically.
  3. The physical features of an environment can produce profound changes in behavior surrounding ethical and social norms.
  4. An in-group member behaves unethically and the behavior is visible to others, people follow suit.

Ethics in Data Visualization
Cheng Peng
Ethical Versus Unethical Behaviors
Intentional Unethical Behaviors
Unintentional Unethical Behaviors
Ethics of Data Visualization
Ethical Principles in Viz
Poor Designs
Wrong or Dubious Info
Insufficient Data
Concealling/Confusing Uncertainty
Suggesting Misleading Patterns

Unintentional Unethical Behaviors

- **Unintentional unethical behavior** occurs when people engage in unethical action beyond their awareness. There are three sources of ethical blind spots.
  1. **Implicit Biases**: implicit attitudes, egocentric biases, over-discounting.
  2. **Temporal Distance from An Ethical Dilemma**: temporal inconsistencies prevent us from being as ethical as we desire to be.
  3. **Failure to Notice Others' Unethical Behavior**: conflicts of interest, outcome bias, and intermediaries, etc.

Ethics in Data Visualization
Cheng Peng
Ethical Versus Unethical Behaviors
Intentional Unethical Behaviors
Unintentional Unethical Behaviors
Ethics of Data Visualization
Ethical Principles in Viz
Poor Designs
Wrong or Dubious Info
Insufficient Data
Concealling/Confusing Uncertainty
Suggesting Misleading Patterns

Ethics of Data Visualization

- **Charts can be misleading**: Well-designed charts are empowering. However, they can also deceive viewers regardless of whether they are designed with ill-intent or not. Therefore, information and data visualization has ethical consequences.
- Ethical thinking is not only about intentions, but also about consequences.

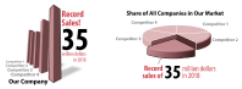
Ethics in Data Visualization
Cheng Peng
Ethical Versus Unethical Behaviors
Intentional Unethical Behaviors
Unintentional Unethical Behaviors
Ethics of Data Visualization
Ethical Principles in Viz
Poor Designs
Wrong or Dubious Info
Insufficient Data
Concealling/Confusing Uncertainty
Suggesting Misleading Patterns

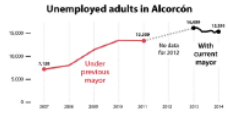
Ethical Principles in Viz

- **Three ethical principles** in creating and distributing visualizations: Transparency, Integrity, and Accountability.
- **Transparency** helps users effectively understand the data and glean new insights. A graphic that is confusing or misleading is unethical, regardless of intent, since it creates misunderstanding for the audience.
- **Integrity**: Act with honesty when creating and distributing visualizations and avoid bias in charts and graphs. Become an ethical creator and responsible purveyor of data visualizations.
- **Accountability**: Accepting responsibility for all decisions in creating data visualizations since a visualization has consequences.

Ethics in Data Visualization
Cheng Peng
Ethical Versus Unethical Behaviors
Intentional Unethical Behaviors
Unintentional Unethical Behaviors
Ethics of Data Visualization
Ethical Principles in Viz
Poor Designs
Wrong or Dubious Info
Insufficient Data
Concealling/Confusing Uncertainty
Suggesting Misleading Patterns

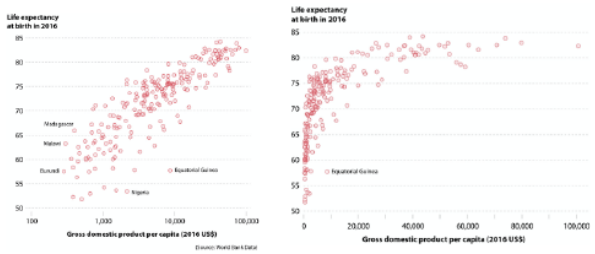
Poor Designs





- Display insufficient information or too much information: Too much data or too complex a chart can also cause misunderstanding.

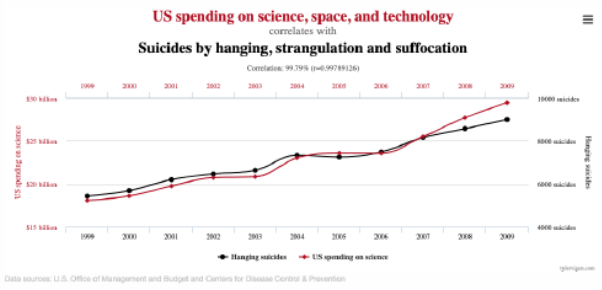
Wrong scale in the horizontal axis!



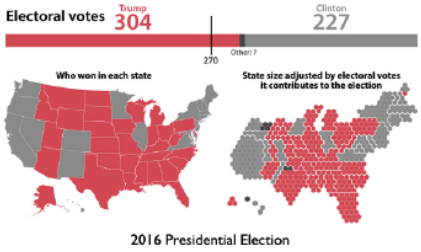
Wrong or Dubious Info

- "Garbage in, garbage out"
- An argument may sound very solid and convincing, but if its premise is wrong, then the argument is wrong.
- A chart may look pretty, intriguing, or surprising, but if it encodes faulty data, then it's a chart that lies.
- Graphs and charts might be automatically generated by the software, but we still have to make sure the output is right.
- How to spot the garbage before it gets in requires some technical understanding of the data and the content to visualize.

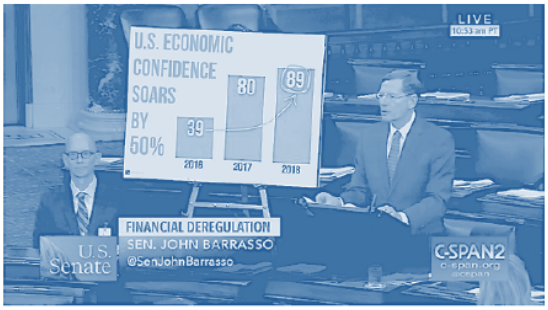
- Spurious correlation between two variables



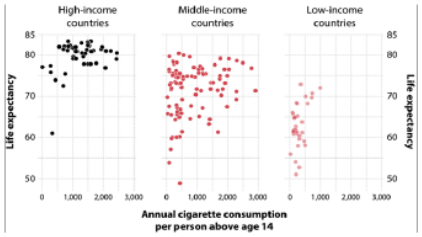
- In US, the result of presidential election is determined by the number of electoral votes won by each of the two candidates instead of the size of the territory or the number of voters.

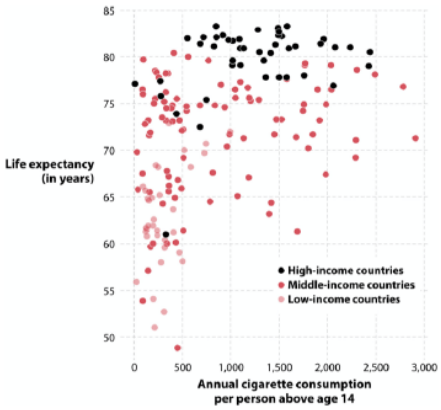


- What's the issue?



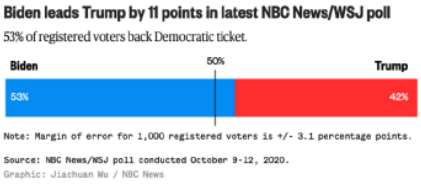
- The spurious positive association between cigarette consumption and life expectancy at a country-by-country level does not imply causation. Correlation is not equal to causation. Other factors such as affluence may be the cause. People in wealthier countries can afford to buy more cigarettes. They also tend to live longer due to access to better diets and health care.





Insufficient Data

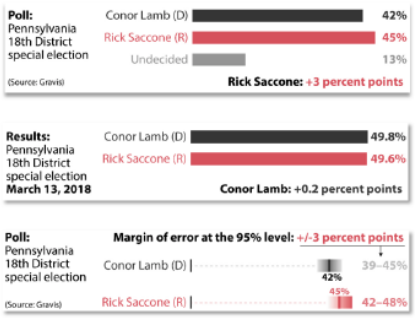
- The error margin in the above chart reflects the extent of uncertainty in the poll result. Typically, it is reported at 95% confidence level



Adjusted or unadjusted data?

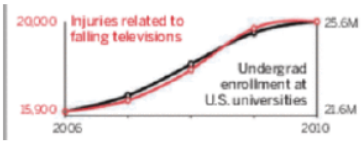
Concealling/Confusing Uncertainty

- To avoid lying, charts need to be precise; but sometimes too much precision is detrimental to understanding.
- Data is often uncertain, and this uncertainty should be disclosed. Ignoring it may lead to faulty reasoning.



Suggesting Misleading Patterns

- Good charts are useful because they untangle the complexity of numbers, making them more concrete and tangible.
- Charts can also lead us to spot patterns and trends that are dubious, spurious, or misleading, particularly when we pair them with the human brain's tendency to read too much into what we see and to always try to confirm what we already believe.



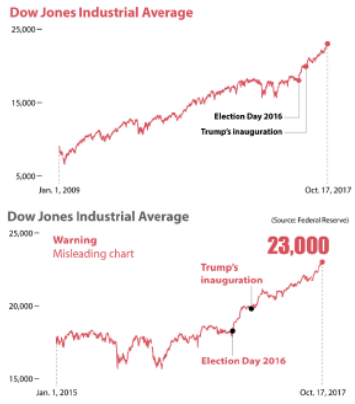
Advices for Reading Charts

- Good charts reveal realities that may otherwise go unnoticed.
- A chart shows only what it shows, and nothing else.
- Don't read too much into a chart—particularly if you're reading what you'd like to read.
- Different levels of thinking may require different levels of data aggregation.
- A good chart is a simplification of reality, and it reveals as much as it hides.
- The patterns or trends on this chart, on their own, should be sufficient to support the claims the author makes.

Advices for Reading Charts

- Good charts reveal realities that may otherwise go unnoticed.
- A chart shows only what it shows, and nothing else.
- Don't read too much into a chart—particularly if you're reading what you'd like to read.
- Different levels of thinking may require different levels of data aggregation.
- A good chart is a simplification of reality, and it reveals as much as it hides.
- The patterns or trends on this chart, on their own, should be sufficient to support the claims the author makes.

Insufficient information leads misleading charts.



Chapter 8

Data Processing with Tidyverse

Data visualization is a form of data analysis (also called visual analytics). This means we need to prepare data sets that are appropriate for visualizations. Recall the following work flow of data visualizations mentioned in earlier notes.

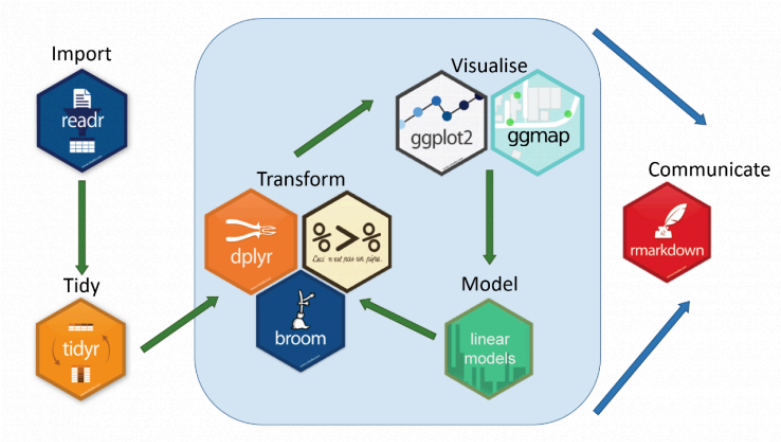


Figure 8.1: Tidyverse workflow.

The major data management tasks are data aggregation and extraction.

- **Information Aggregation** - combining information in different relational data sets to make an integrated single data set for data visualization.
- **Information Extraction** - subsetting a single data set to make small data sets that have specific information for creating a visualization.

The base R package has some powerful and easy-to-use functions to perform these types of data management.

8.1 Data Cleaning and Preparation for Visualization

8.1.1 Data Cleaning

Data cleaning refers to the process of making a data set possibly from different sources of `raw data` for modeling, visualization, and relevant analysis. The major tasks include:

- Removing unnecessary variables
- Deleting duplicate rows/observations
- Addressing outliers or invalid data
- Dealing with missing values
- Standardizing or categorizing values
- Correcting typographical errors

8.1.2 Data Preparation for Visualization

For a specific data analysis such as modeling or data visualization, we need to create an analytic data set based on clean data sets.

Formatting/Conversion

- Formatting columns appropriately (numbers are treated as numbers, dates as dates)
- Convert values into appropriate units

Filtering/Subsetting

- Filter your data to focus on the specific data that interests you.
- Group data and create aggregate values for groups (Counts, Min, Max, Mean, Median, Mode)
- Extract values from complex columns

Aggregation/Merging

- Combine variables to create new columns
- Merge different relational data sets

8.2 Basic Data Management: Merging Data Sets

There are different packages in R that have various functions capable of doing data management. In this note, we introduce the commonly used functions in

base R and `tidyverse`.

8.2.1 Merge Data Sets

It is very common that the information we are interested in resides in different data sources. In order to merge different data sets, there must be at least one variable "key" that links to different data sets.

There are several different operations in SQL to create different types of the merged data set. The following are the most commonly used ones.

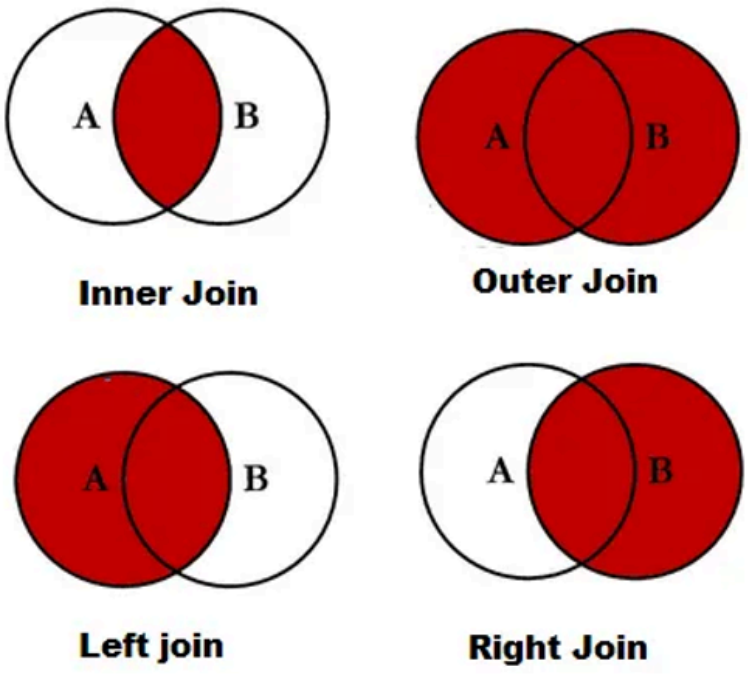


Figure 8.2: Table joins.

The next figure shows the basic operations with tiny tables illustrating the operations.

8.2.2 Merging Data in Base R

The base R function `merge()` can be used to perform different joins. To illustrate, we use the tiny toy data set in the above figure to show you how to use `merge()` function.

- Defining Data Frames

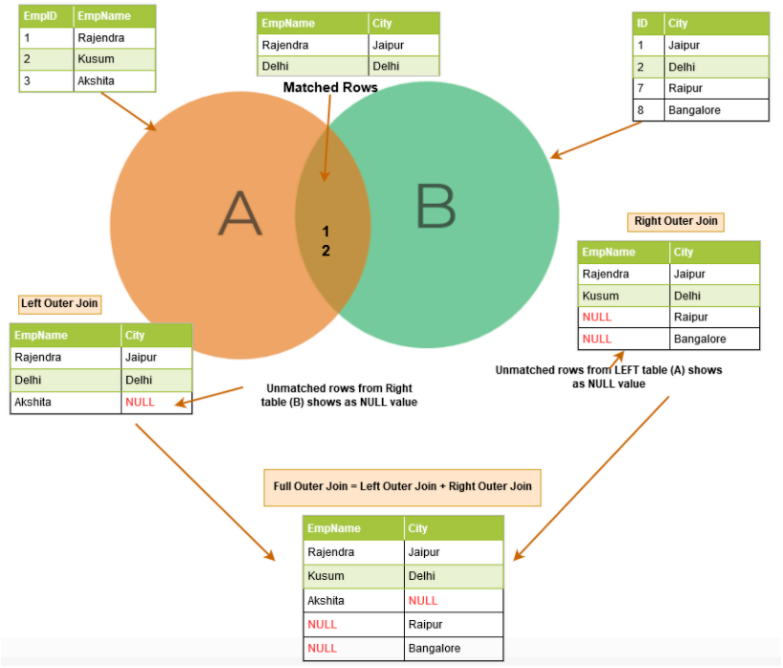


Figure 8.3: An illustrative example of table join.

```
employee = data.frame(EmpID = c(1,2,3), EmpName = c("Rajendra", "Kusum", "Akshita"))
city = data.frame(ID = c(1,2,7,8), City =c("Jaipur", "Delhi", "Raipur", "Bangalore"))
```

- Inner Join

```
innerjoin = merge(x=employee, city, by.x = 'EmpID', by.y ='ID', all = FALSE)
innerjoin
```

```
##   EmpID  EmpName   City
## 1     1 Rajendra Jaipur
## 2     2    Kusum  Delhi
```

- Outer Join

```
innerjoin = merge(x = employee, y = city, by.x = 'EmpID', by.y ='ID', all = TRUE)
innerjoin
```

```
##   EmpID  EmpName      City
## 1     1 Rajendra    Jaipur
## 2     2    Kusum     Delhi
## 3     3  Akshita      <NA>
## 4     7     <NA>    Raipur
## 5     8     <NA> Bangalore
```

- Left Join

```
leftjoin = merge(x = employee, y = city, by.x = 'EmpID', by.y ='ID', all.x = TRUE)
leftjoin
```

```
##   EmpID  EmpName   City
## 1     1 Rajendra Jaipur
## 2     2    Kusum  Delhi
## 3     3  Akshita   <NA>
```

- Right Join

```
rightjoin = merge(x = employee, y = city, by.x = 'EmpID', by.y ='ID', all.y = TRUE)
rightjoin
```

```
##   EmpID  EmpName      City
## 1     1 Rajendra    Jaipur
## 2     2    Kusum     Delhi
## 3     7     <NA>    Raipur
## 4     8     <NA> Bangalore
```

8.2.3 Merging Data with Mutating Joins in dplyr

The package **dplyr** has the following four join functions corresponding to the options in the base R function 'merge()'.

The mutating joins add columns from y to x, matching rows based on the keys:

- `inner_join()`: includes all rows in x and y.
- `left_join()`: includes all rows in x.
- `right_join()`: includes all rows in y.
- `full_join()` (also called **outer join**): includes all rows in x or y (also called outer join)..

If a row in x matches multiple rows in y, all the rows in y will be returned once for each matching row in x.

To use mutating joins, we first rename key variables so that primary keys have the same name.

```
employee.new = employee
employee.new$ID = employee$EmpID  # adding the new renamed ID
employee.new = employee.new[, -1]     # drop the old ID variable
employee.new
```

```
##    EmpName ID
## 1 Rajendra  1
## 2    Kusum  2
## 3  Akshita  3
```

- Inner Join

```
inner_join(employee.new, city, by = "ID")
```

```
##    EmpName ID   City
## 1 Rajendra  1 Jaipur
## 2    Kusum  2  Delhi
```

- Left Join

```
left_join(employee.new, city, by = "ID")
```

```
##    EmpName ID   City
## 1 Rajendra  1 Jaipur
## 2    Kusum  2  Delhi
## 3  Akshita  3   <NA>
```

- right Join

```
right_join(employee.new, city, by = "ID")
```

```
##    EmpName ID      City
## 1 Rajendra  1    Jaipur
## 2    Kusum  2     Delhi
## 3     <NA>  7    Raipur
## 4     <NA>  8 Bangalore
```

- Full (Outer) Join

```
full_join(employee.new, city, by = "ID")
```

```
##    EmpName ID      City
## 1 Rajendra  1    Jaipur
## 2    Kusum  2     Delhi
## 3  Akshita  3      <NA>
## 4     <NA>  7    Raipur
## 5     <NA>  8 Bangalore
```

8.2.4 Use of Pipe Operator %>% with Mutating Joins

The pipe operator, written as `%>%` takes the output of one function and passes it into another function as an argument. This allows us to link a sequence of analysis steps using functions in `dplyr` and `tidyr` in data wrangling.

- Inner Join

```
pipe.innerjoin <- employee %>% inner_join(city, by = c("EmpID" = "ID"))
pipe.innerjoin
```

```
##   EmpID  EmpName   City
## 1     1 Rajendra Jaipur
```

```
## 2     2    Kusum  Delhi
```

- Full (Outer) Join

```
pipe.outerjoin <- employee %>% full_join(city, by = c("EmpID" = "ID"))
pipe.outerjoin
```

```
##   EmpID  EmpName      City
## 1     1 Rajendra    Jaipur
## 2     2    Kusum     Delhi
## 3     3  Akshita      <NA>
## 4     7     <NA>    Raipur
## 5     8     <NA> Bangalore
```

- Left Join

```
pipe.leftjoin <- employee %>% left_join(city, by = c("EmpID" = "ID"))
pipe.leftjoin
```

```
##   EmpID  EmpName   City
## 1     1 Rajendra Jaipur
## 2     2    Kusum  Delhi
## 3     3  Akshita   <NA>
```

- Right Join

```
pipe.rightjoin <- employee %>% right_join(city, by = c("EmpID" = "ID"))
pipe.rightjoin
```

```
##   EmpID  EmpName      City
## 1     1 Rajendra    Jaipur
## 2     2    Kusum     Delhi
## 3     7     <NA>    Raipur
## 4     8     <NA> Bangalore
```

8.3 Basic Data Management: Subsetting Data

Another important data management task is to subset data sets to extract the desired information for analyses and visualization.

Two operations are used to subset a data set: select/drop columns and select rows that meet certain conditions.

The working data set in the section is the well-known `iris` data set that has 4 numerical variables (attributes of iris flowers) and a categorical variable (species of iris flowers).

8.3.1 Accessors in R [, [[and $

When subsetting a data set, it is unavoidable to access the value(s) of certain variable(s). Three R accessors are commonly used in R coding.

- [subsetting a data set

This R accessor is probably the most commonly used. When we want a subset of an object using [. Remember that when we take a subset of the object you get **the same type** of thing. Thus, the subset of a vector will be a vector, the subset of a list will be a list and the subset of a data.frame will be a data.frame.

- [[extracting one item

The double square brackets are used to extract one element from potentially many. For vectors yield vectors with a single value; data frames give a column vector; for a list, one element:

For example,

```
letters[[3]]            # extracts the third element in the vector of all lower case l
```

```
## [1] "c"
```

```
iris[["Petal.Length"]]  # extract the variable named 'Petal.Length' in the data frame.
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4 1.7 1
##  [28] 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1
##  [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0 4.9 4
##  [82] 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6
## [109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4
## [136] 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

The double square bracket looks as if we are asking for something deep within a container. We are not taking a slice but reaching to get at the one thing at the core.

- Interact with $

The accessor that provides the least unique utility is also probably used the most often used. $ is a special case of [[in which we access a single item by actual name. The following are equivalent:

```
iris$Petal.Length
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4 1.7 1
##  [28] 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1
##  [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0 4.9 4
##  [82] 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6
## [109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4
## [136] 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

```
iris[["Petal.Length"]]
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4 1.7 1.5 1.7 1.5
##  [28] 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5
##  [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0 4.9 4.7 4.3 4.4
##  [82] 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9
## [109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4.9 5.6 5.8
## [136] 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

8.3.2 Subsetting Data in Base R

Selecting/Dropping Columns

Subsetting a data set by selecting or dropping a subset of variables (columns) from a data set is straightforward.

For example, we can define a subset of the `iris` data set by selecting all numerical variables.

```
iris.names = names(iris)
iris0 = iris[, iris.names[1:4]]
iris0[c(1:3, 60:63, 110:113),]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1            5.1         3.5          1.4         0.2
## 2            4.9         3.0          1.4         0.2
## 3            4.7         3.2          1.3         0.2
## 60           5.2         2.7          3.9         1.4
## 61           5.0         2.0          3.5         1.0
## 62           5.9         3.0          4.2         1.5
## 63           6.0         2.2          4.0         1.0
## 110          7.2         3.6          6.1         2.5
## 111          6.5         3.2          5.1         2.0
## 112          6.4         2.7          5.3         1.9
## 113          6.8         3.0          5.5         2.1
```

We can also create the same data set by dropping variables in the original data set. For example

```
iris02 = iris[, -5]
iris02[c(1:3, 60:63, 110:113),]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1            5.1         3.5          1.4         0.2
## 2            4.9         3.0          1.4         0.2
## 3            4.7         3.2          1.3         0.2
## 60           5.2         2.7          3.9         1.4
## 61           5.0         2.0          3.5         1.0
## 62           5.9         3.0          4.2         1.5
```

```
## 63          6.0         2.2          4.0         1.0
## 110         7.2         3.6          6.1         2.5
## 111         6.5         3.2          5.1         2.0
## 112         6.4         2.7          5.3         1.9
## 113         6.8         3.0          5.5         2.1
```

Selection/Dropping Rows

This is also relatively straightforward. The basic idea is to identify row IDs to select or drop the corresponding rows. The R function `which()` can this trick!

The following example illustrates the way of using `which()` to subsetting data.

1. Selecting One Species of Iris Flowers

```
setosa.id = which(iris$Species == "setosa")
setosa.flower = iris[setosa.id,]
setosa.flower[1:10,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

2. Selecting Two Species of Iris Flowers

The following three code chunks create the same data set.

Method 1:

```
not.setosa.id01 = which(iris$Species != "setosa")
not.setosa.flower01 = iris[not.setosa.id01,]
not.setosa.flower01[1:10,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 51          7.0         3.2          4.7         1.4 versicolor
## 52          6.4         3.2          4.5         1.5 versicolor
## 53          6.9         3.1          4.9         1.5 versicolor
## 54          5.5         2.3          4.0         1.3 versicolor
## 55          6.5         2.8          4.6         1.5 versicolor
## 56          5.7         2.8          4.5         1.3 versicolor
## 57          6.3         3.3          4.7         1.6 versicolor
## 58          4.9         2.4          3.3         1.0 versicolor
## 59          6.6         2.9          4.6         1.3 versicolor
```

```
## 60          5.2         2.7          3.9         1.4 versicolor
```

Method 2:

```
not.setosa.id02 = which(iris$Species == "virginica" | iris$Species =="versicolor")
not.setosa.flower02 = iris[not.setosa.id02,]
not.setosa.flower02[1:10,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 51          7.0         3.2          4.7         1.4 versicolor
## 52          6.4         3.2          4.5         1.5 versicolor
## 53          6.9         3.1          4.9         1.5 versicolor
## 54          5.5         2.3          4.0         1.3 versicolor
## 55          6.5         2.8          4.6         1.5 versicolor
## 56          5.7         2.8          4.5         1.3 versicolor
## 57          6.3         3.3          4.7         1.6 versicolor
## 58          4.9         2.4          3.3         1.0 versicolor
## 59          6.6         2.9          4.6         1.3 versicolor
## 60          5.2         2.7          3.9         1.4 versicolor
```

Method 3:

```
not.setosa.id03 = which(iris$Species %in% c("versicolor", "virginica"))
not.setosa.flower03 = iris[not.setosa.id01,]
not.setosa.flower03[1:10,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 51          7.0         3.2          4.7         1.4 versicolor
## 52          6.4         3.2          4.5         1.5 versicolor
## 53          6.9         3.1          4.9         1.5 versicolor
## 54          5.5         2.3          4.0         1.3 versicolor
## 55          6.5         2.8          4.6         1.5 versicolor
## 56          5.7         2.8          4.5         1.3 versicolor
## 57          6.3         3.3          4.7         1.6 versicolor
## 58          4.9         2.4          3.3         1.0 versicolor
## 59          6.6         2.9          4.6         1.3 versicolor
## 60          5.2         2.7          3.9         1.4 versicolor
```

8.3.3 Subsetting Data with `dplyr`

dplyr provides helper tools for the most common data manipulation tasks. It is built to work directly with data frames and has the ability to work directly with data stored in an external database. We can conduct queries on the database directly and pull back into R only what we need for analysis.

Since selecting/dropping variables is straightforward (particularly when using %>%). Next, we provide a few examples showing how to use `filter()` to select/drop rows with certain conditions.

- Filtering by one criterion

```
filter(iris, Species == "setosa")
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
## 13          4.8         3.0          1.4         0.1  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 16          5.7         4.4          1.5         0.4  setosa
## 17          5.4         3.9          1.3         0.4  setosa
## 18          5.1         3.5          1.4         0.3  setosa
## 19          5.7         3.8          1.7         0.3  setosa
## 20          5.1         3.8          1.5         0.3  setosa
## 21          5.4         3.4          1.7         0.2  setosa
## 22          5.1         3.7          1.5         0.4  setosa
## 23          4.6         3.6          1.0         0.2  setosa
## 24          5.1         3.3          1.7         0.5  setosa
## 25          4.8         3.4          1.9         0.2  setosa
## 26          5.0         3.0          1.6         0.2  setosa
## 27          5.0         3.4          1.6         0.4  setosa
## 28          5.2         3.5          1.5         0.2  setosa
## 29          5.2         3.4          1.4         0.2  setosa
## 30          4.7         3.2          1.6         0.2  setosa
## 31          4.8         3.1          1.6         0.2  setosa
## 32          5.4         3.4          1.5         0.4  setosa
## 33          5.2         4.1          1.5         0.1  setosa
## 34          5.5         4.2          1.4         0.2  setosa
## 35          4.9         3.1          1.5         0.2  setosa
## 36          5.0         3.2          1.2         0.2  setosa
## 37          5.5         3.5          1.3         0.2  setosa
## 38          4.9         3.6          1.4         0.1  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 40          5.1         3.4          1.5         0.2  setosa
## 41          5.0         3.5          1.3         0.3  setosa
```

```
## 42          4.5         2.3          1.3         0.3  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 44          5.0         3.5          1.6         0.6  setosa
## 45          5.1         3.8          1.9         0.4  setosa
## 46          4.8         3.0          1.4         0.3  setosa
## 47          5.1         3.8          1.6         0.2  setosa
## 48          4.6         3.2          1.4         0.2  setosa
## 49          5.3         3.7          1.5         0.2  setosa
## 50          5.0         3.3          1.4         0.2  setosa
```

```
filter(iris, Sepal.Length > 6)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1           7.0         3.2          4.7         1.4 versicolor
## 2           6.4         3.2          4.5         1.5 versicolor
## 3           6.9         3.1          4.9         1.5 versicolor
## 4           6.5         2.8          4.6         1.5 versicolor
## 5           6.3         3.3          4.7         1.6 versicolor
## 6           6.6         2.9          4.6         1.3 versicolor
## 7           6.1         2.9          4.7         1.4 versicolor
## 8           6.7         3.1          4.4         1.4 versicolor
## 9           6.2         2.2          4.5         1.5 versicolor
## 10          6.1         2.8          4.0         1.3 versicolor
## 11          6.3         2.5          4.9         1.5 versicolor
## 12          6.1         2.8          4.7         1.2 versicolor
## 13          6.4         2.9          4.3         1.3 versicolor
## 14          6.6         3.0          4.4         1.4 versicolor
## 15          6.8         2.8          4.8         1.4 versicolor
## 16          6.7         3.0          5.0         1.7 versicolor
## 17          6.7         3.1          4.7         1.5 versicolor
## 18          6.3         2.3          4.4         1.3 versicolor
## 19          6.1         3.0          4.6         1.4 versicolor
## 20          6.2         2.9          4.3         1.3 versicolor
## 21          6.3         3.3          6.0         2.5  virginica
## 22          7.1         3.0          5.9         2.1  virginica
## 23          6.3         2.9          5.6         1.8  virginica
## 24          6.5         3.0          5.8         2.2  virginica
## 25          7.6         3.0          6.6         2.1  virginica
## 26          7.3         2.9          6.3         1.8  virginica
## 27          6.7         2.5          5.8         1.8  virginica
## 28          7.2         3.6          6.1         2.5  virginica
## 29          6.5         3.2          5.1         2.0  virginica
## 30          6.4         2.7          5.3         1.9  virginica
## 31          6.8         3.0          5.5         2.1  virginica
## 32          6.4         3.2          5.3         2.3  virginica
## 33          6.5         3.0          5.5         1.8  virginica
```

```
## 34          7.7         3.8          6.7         2.2  virginica
## 35          7.7         2.6          6.9         2.3  virginica
## 36          6.9         3.2          5.7         2.3  virginica
## 37          7.7         2.8          6.7         2.0  virginica
## 38          6.3         2.7          4.9         1.8  virginica
## 39          6.7         3.3          5.7         2.1  virginica
## 40          7.2         3.2          6.0         1.8  virginica
## 41          6.2         2.8          4.8         1.8  virginica
## 42          6.1         3.0          4.9         1.8  virginica
## 43          6.4         2.8          5.6         2.1  virginica
## 44          7.2         3.0          5.8         1.6  virginica
## 45          7.4         2.8          6.1         1.9  virginica
## 46          7.9         3.8          6.4         2.0  virginica
## 47          6.4         2.8          5.6         2.2  virginica
## 48          6.3         2.8          5.1         1.5  virginica
## 49          6.1         2.6          5.6         1.4  virginica
## 50          7.7         3.0          6.1         2.3  virginica
## 51          6.3         3.4          5.6         2.4  virginica
## 52          6.4         3.1          5.5         1.8  virginica
## 53          6.9         3.1          5.4         2.1  virginica
## 54          6.7         3.1          5.6         2.4  virginica
## 55          6.9         3.1          5.1         2.3  virginica
## 56          6.8         3.2          5.9         2.3  virginica
## 57          6.7         3.3          5.7         2.5  virginica
## 58          6.7         3.0          5.2         2.3  virginica
## 59          6.3         2.5          5.0         1.9  virginica
## 60          6.5         3.0          5.2         2.0  virginica
## 61          6.2         3.4          5.4         2.3  virginica
```

- When multiple expressions are used, they are combined using `&` (logical AND) or `|` (logical OR)

```
filter(iris, Species == "setosa" & Sepal.Length > 5 )
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           5.4         3.9          1.7         0.4  setosa
## 3           5.4         3.7          1.5         0.2  setosa
## 4           5.8         4.0          1.2         0.2  setosa
## 5           5.7         4.4          1.5         0.4  setosa
## 6           5.4         3.9          1.3         0.4  setosa
## 7           5.1         3.5          1.4         0.3  setosa
## 8           5.7         3.8          1.7         0.3  setosa
## 9           5.1         3.8          1.5         0.3  setosa
## 10          5.4         3.4          1.7         0.2  setosa
## 11          5.1         3.7          1.5         0.4  setosa
## 12          5.1         3.3          1.7         0.5  setosa
```

```
## 13          5.2         3.5          1.5         0.2  setosa
## 14          5.2         3.4          1.4         0.2  setosa
## 15          5.4         3.4          1.5         0.4  setosa
## 16          5.2         4.1          1.5         0.1  setosa
## 17          5.5         4.2          1.4         0.2  setosa
## 18          5.5         3.5          1.3         0.2  setosa
## 19          5.1         3.4          1.5         0.2  setosa
## 20          5.1         3.8          1.9         0.4  setosa
## 21          5.1         3.8          1.6         0.2  setosa
## 22          5.3         3.7          1.5         0.2  setosa
```

```
filter(iris, Species == "setosa" | Sepal.Length > 7 )
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1           5.1         3.5          1.4         0.2    setosa
## 2           4.9         3.0          1.4         0.2    setosa
## 3           4.7         3.2          1.3         0.2    setosa
## 4           4.6         3.1          1.5         0.2    setosa
## 5           5.0         3.6          1.4         0.2    setosa
## 6           5.4         3.9          1.7         0.4    setosa
## 7           4.6         3.4          1.4         0.3    setosa
## 8           5.0         3.4          1.5         0.2    setosa
## 9           4.4         2.9          1.4         0.2    setosa
## 10          4.9         3.1          1.5         0.1    setosa
## 11          5.4         3.7          1.5         0.2    setosa
## 12          4.8         3.4          1.6         0.2    setosa
## 13          4.8         3.0          1.4         0.1    setosa
## 14          4.3         3.0          1.1         0.1    setosa
## 15          5.8         4.0          1.2         0.2    setosa
## 16          5.7         4.4          1.5         0.4    setosa
## 17          5.4         3.9          1.3         0.4    setosa
## 18          5.1         3.5          1.4         0.3    setosa
## 19          5.7         3.8          1.7         0.3    setosa
## 20          5.1         3.8          1.5         0.3    setosa
## 21          5.4         3.4          1.7         0.2    setosa
## 22          5.1         3.7          1.5         0.4    setosa
## 23          4.6         3.6          1.0         0.2    setosa
## 24          5.1         3.3          1.7         0.5    setosa
## 25          4.8         3.4          1.9         0.2    setosa
## 26          5.0         3.0          1.6         0.2    setosa
## 27          5.0         3.4          1.6         0.4    setosa
## 28          5.2         3.5          1.5         0.2    setosa
## 29          5.2         3.4          1.4         0.2    setosa
## 30          4.7         3.2          1.6         0.2    setosa
## 31          4.8         3.1          1.6         0.2    setosa
## 32          5.4         3.4          1.5         0.4    setosa
```

```
## 33          5.2         4.1          1.5         0.1    setosa
## 34          5.5         4.2          1.4         0.2    setosa
## 35          4.9         3.1          1.5         0.2    setosa
## 36          5.0         3.2          1.2         0.2    setosa
## 37          5.5         3.5          1.3         0.2    setosa
## 38          4.9         3.6          1.4         0.1    setosa
## 39          4.4         3.0          1.3         0.2    setosa
## 40          5.1         3.4          1.5         0.2    setosa
## 41          5.0         3.5          1.3         0.3    setosa
## 42          4.5         2.3          1.3         0.3    setosa
## 43          4.4         3.2          1.3         0.2    setosa
## 44          5.0         3.5          1.6         0.6    setosa
## 45          5.1         3.8          1.9         0.4    setosa
## 46          4.8         3.0          1.4         0.3    setosa
## 47          5.1         3.8          1.6         0.2    setosa
## 48          4.6         3.2          1.4         0.2    setosa
## 49          5.3         3.7          1.5         0.2    setosa
## 50          5.0         3.3          1.4         0.2    setosa
## 51          7.1         3.0          5.9         2.1 virginica
## 52          7.6         3.0          6.6         2.1 virginica
## 53          7.3         2.9          6.3         1.8 virginica
## 54          7.2         3.6          6.1         2.5 virginica
## 55          7.7         3.8          6.7         2.2 virginica
## 56          7.7         2.6          6.9         2.3 virginica
## 57          7.7         2.8          6.7         2.0 virginica
## 58          7.2         3.2          6.0         1.8 virginica
## 59          7.2         3.0          5.8         1.6 virginica
## 60          7.4         2.8          6.1         1.9 virginica
## 61          7.9         3.8          6.4         2.0 virginica
## 62          7.7         3.0          6.1         2.3 virginica
```

- To refer to column names that are stored as strings, use the `.data` pronoun:

```
vars <- c("Sepal.Length", "Petal.Length")
cond <- c(6, 5)
subset.iris <- iris %>%
  filter(
    .data[[vars[[1]]]] > cond[[1]],
    .data[[vars[[2]]]] < cond[[2]]
  )
head(subset.iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          7.0         3.2          4.7         1.4 versicolor
## 2          6.4         3.2          4.5         1.5 versicolor
## 3          6.9         3.1          4.9         1.5 versicolor
## 4          6.5         2.8          4.6         1.5 versicolor
```

```
## 5          6.3         3.3          4.7         1.6 versicolor
## 6          6.6         2.9          4.6         1.3 versicolor
```

8.3.4 Variable Definition and Variable Type Conversion

- Define New Variables

Defining new variables based on the existing variables is straightforward in R using the basic arithmetic and mathematical operations. When using `%>%`, `dplyr()` is used to define new variables.

- Variable Type Conversion

Type conversions in R work as you would expect. For example, adding a character string to a numeric vector converts all the elements in the vector to the character.

1. Use `is.foo` to test for data type foo. Returns TRUE or FALSE

`is.numeric(), is.character(), is.vector(), is.matrix(), is.data.frame()`

2. Use `as.foo` to explicitly convert it.

`as.numeric(), as.character(), as.vector(), as.matrix(), as.data.frame)`

8.4 Importing/Exporting Data

8.5 Importing Dara

There are different functions in various R libraries to read data to R.

- Base R and Libraries Come with Base R

R loading functions in {utils}: `read.table()`, `read.csv()`, `read.csv2()`, `read.delim()`, and `read.delim2()`

- Functions in {tidyverse}

As a part of {tidyverse}, the library {readr} has several functions to read the data in common formats.

`read_table(), read_delim(), read_csv(), read_csv2(), read_tsv()`

- Read data set generated by other programs such as SAS, SPSS, etc.

Several libraries are useful to load special formats of data to R. Three important libraries are

`{xlsx, Hmisc, foreign}`.

8.5.1 Exporting Data

We have learned how to use `dplyr` to extract information from or summarize your raw data, we may want to export these new data sets to share them with other people or for archival.

Similar to the `read_csv()` function used for reading CSV files into R, there is a `write_csv()` function that generates CSV files from data frames.

Let's assume our data set under the name, `final_data`, is ready, we can save it as a CSV file in our `data` folder using the following code.

```
write_csv(final_data, file = "data/final_data.csv")
```

8.6 Overview of `Tidyverse` (Optional)

There are several R libraries that have powerful tools for data wrangling and information extraction. Tidyverse is a collection of essential R packages for data science. There 8 packages under the `tidyverse umbrella` that help us in performing and interacting with the data.

8.6.1 Packages for Data Wrangling and Transformation

- **dplyr** provides helper tools for the most common data manipulation tasks. It is built to work directly with data frames and has the ability to work directly with data stored in an external database. We can conduct queries on the database directly, and pull back into R only what we need for analysis.
- **tidyr** addresses the common problem of wanting to reshape the data with a sophisticated layout for plotting and usage by different R functions.
- **stringr** deals with string variables. It plays a big role in processing raw data into a cleaner and easily understandable format.
- **forcats** is dedicated to dealing with categorical variables or factors. Anyone who has worked with categorical data knows what a nightmare they can be.

8.6.2 Packages for Data Import and Management

- **tibble** is a new modern data frame with nicer behavior around printing, subsetting, and factor handling. It keeps many important features of the original data frame and removes many of the outdated features.
- **readr** package is recently developed to deal with reading in large flat files quickly. The package provides replacements for functions like `read.table()` and `read.csv()`. The analogous functions in {readr} are `read_table()` and `read_csv()`.

8.6.3 Functional Programming with Library {purrr}

- **purrr** is a new package that fills in the missing pieces in R's functional programming tools. This is not a coding class. We will not use 'purrr' in this class.

8.6.4 Data Visualization and Exploration

ggplot2 is a powerful and flexible R package for producing elegant graphics. The concept behind `ggplot2` divides plot into three different fundamental parts: `Plot = data + Aesthetics + Geometry`.

The principal components of every plot can be defined as follow:

- **Aesthetics** is used to indicate x and y variables. It can also be used to control the color, the size or the shape of points, the height of bars, etc.
- **Geometry** defines the type of graphics (histogram, box plot, line plot, density plot, dot plot, etc.)

This will be one of the primary tools for this class.

8.7 Data Management with `dplyr` (Optional)

What we can do in the standard SQL can also be done with `dyplr`. For the convenience of illustration, we use a simple well-known built-in `iris data set`.

8.7.1 Common `dplyr` Functions

The following is the list of functions in `dplyr`.

- **select()**: sub-setting columns.

To select columns of a data frame, use `select()`. The first argument to this function is the data frame (`iris`), and the subsequent arguments are the columns to keep. For example

```
iris.petal = select(iris, Petal.Length, Petal.Width, Species)
str(iris.petal)
```

```
## 'data.frame':    150 obs. of  3 variables:
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

To select all columns except certain ones, put a "-" in front of the variable to exclude it. For example, we **exclude** Petal information and only **keep** Sepal information, we can use the following code

```
iris.sepal = select(iris, -Petal.Length, -Petal.Width)
str(iris.sepal)
```

```
## 'data.frame':    150 obs. of  3 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 .
```

This will select all the variables in surveys except for `Petal.Length`, and `Petal.Width`.

- **filter()**: sub-setting rows on conditions.

For example, if we only select one species `Versicolor`, we can use the following code.

```
versicolor = filter(iris, Species =="versicolor")
summary(versicolor)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length   Petal.Width          Species
##  Min.   :4.900   Min.   :2.000   Min.   :3.00   Min.   :1.000   setosa    : 0
##  1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.00   1st Qu.:1.200   versicolor:50
##  Median :5.900   Median :2.800   Median :4.35   Median :1.300   virginica : 0
##  Mean   :5.936   Mean   :2.770   Mean   :4.26   Mean   :1.326
##  3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.60   3rd Qu.:1.500
##  Max.   :7.000   Max.   :3.400   Max.   :5.10   Max.   :1.800
```

If we subset a data set by selecting a certain number of columns and row with multiple conditions, pipe operator `%>%` will make subsetting easy. For example, if we only want to study `sepal width` and `length` of `setosa` where petal length is less than 1.5. The following code using `%>%`

```
resulting.subset <- iris %>%
             filter(Petal.Length < 1.5, Species =="setosa") %>%
             select(Sepal.Length, Sepal.Width, Species)
summary(resulting.subset)
```

```
##   Sepal.Length    Sepal.Width          Species
##  Min.   :4.300   Min.   :2.300   setosa    :24
##  1st Qu.:4.600   1st Qu.:3.000   versicolor: 0
##  Median :4.900   Median :3.350   virginica : 0
##  Mean   :4.896   Mean   :3.333
##  3rd Qu.:5.100   3rd Qu.:3.525
##  Max.   :5.800   Max.   :4.200
```

Note that, multiple conditional statements are separated by `,` or `&`. Using `%>%`, we don't need to include the data set as the first argument.

- **mutate()**: creating new columns by using information from other columns.

Frequently we want to create new columns based on the values in existing columns. For example, we want to define two ratios of the sepal and petal widths and sepal and petal lengths. For this, we'll use mutate().

```
expanded.data <- iris %>%
             mutate(length.ratio = Sepal.Length/Petal.Length,
                    width.ratio = Sepal.Width/Petal.Width)
summary(expanded.data)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species    length.ratio
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50   Min.   :1.050
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50   1st Qu.:1.230
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50   Median :1.411
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199                   Mean   :2.018
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800                   3rd Qu.:3.176
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500                   Max.   :4.833
```

- **group_by()** and **summarize()**: creating summary statistics on grouped data.

'group_by()' is often used together with 'summarize()', which collapses each group into a single-row summary of that group. 'group_by()' takes as arguments the column names that contain the categorical variables for which you want to calculate the summary statistics.

The following code yields a set of summarized statistics including the mean of sepal width and length as well as the correlation coefficients in each of the three species.

```
summary.stats <- iris %>%
             group_by(Species) %>%
             summarize(sepal.width.avg = mean(Sepal.Width),
                       sepal.length.avg = mean(Sepal.Length),
                       corr.sepal = cor(Sepal.Length, Sepal.Width))
summary.stats
```

```
## # A tibble: 3 x 4
##   Species    sepal.width.avg sepal.length.avg corr.sepal
##   <fct>                <dbl>            <dbl>      <dbl>
## 1 setosa                3.43             5.01      0.743
## 2 versicolor            2.77             5.94      0.526
## 3 virginica             2.97             6.59      0.457
```

All R functions such as `min()`, `max()`, , that yield summarized statistics can be used with `summarize()`. We can also filter out some observations before we compute the summary statistics.

```
summary.stats.filtering <- iris %>%
             filter(Petal.Length < 5) %>%
```

```
            group_by(Species) %>%
            summarize(sepal.width.avg = mean(Sepal.Width),
                      sepal.length.avg = mean(Sepal.Length),
                      corr.sepal = cor(Sepal.Length, Sepal.Width))
summary.stats.filtering
```

```
## # A tibble: 3 x 4
##   Species    sepal.width.avg sepal.length.avg corr.sepal
##   <fct>                <dbl>            <dbl>      <dbl>
## 1 setosa                3.43             5.01      0.743
## 2 versicolor            2.77             5.92      0.519
## 3 virginica             2.8              5.85      0.643
```

- **arrange()**: sorting results.

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by decreasing the order of mean weight.

```
summary.stats.sort <- iris %>%
            filter(Petal.Length < 5) %>%
            group_by(Species) %>%
            summarize(sepal.width.avg = mean(Sepal.Width),
                      sepal.length.avg = mean(Sepal.Length),
                      corr.sepal = cor(Sepal.Length, Sepal.Width)) %>%
             arrange(desc(corr.sepal))
summary.stats.sort
```

```
## # A tibble: 3 x 4
##   Species    sepal.width.avg sepal.length.avg corr.sepal
##   <fct>                <dbl>            <dbl>      <dbl>
## 1 setosa                3.43             5.01      0.743
## 2 virginica             2.8              5.85      0.643
## 3 versicolor            2.77             5.92      0.519
```

The resulting data set can also be sorted by multiple variables.

- **count()**: counting discrete values.

When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, `dplyr` provides `count()`. For example, if we wanted to count the number of rows of data for each species after we filter out all records with Petal Length < 5, we would do:

```
summary.count <- iris %>%
            filter(Petal.Length < 5) %>%
            group_by(Species) %>%
            summarize(count = n(), sort = TRUE)
summary.count
```

```
## # A tibble: 3 x 3
##   Species    count sort
##   <fct>      <int> <lgl>
## 1 setosa        50 TRUE
## 2 versicolor    48 TRUE
## 3 virginica      6 TRUE
```

If we wanted to count a combination of factors, say `factor A` and `factor B`, we would specify the first and the second factor as the arguments of `count(factor A, factor B)`.

8.7.2 Reshaping Functions in `tidyr`

The `tidyr` package complements `dplyr` perfectly. It boosts the power of dplyr for data manipulation and pre-processing. To illustrate how to use these functions, we consider defining a subset from `iris` that contains only two variables: Sepal Length and Species.

```
life.expectancy <-read.csv("https://raw.githubusercontent.com/pengdsci/sta553.html/main/life_expe
life.expectancy[1:5, 1:10]
```

```
##                country X1799 X1800 X1801 X1802 X1803 X1804 X1805 X1806 X1807
## 1          Afghanistan  28.2  28.2  28.2  28.2  28.2  28.2  28.1  28.1  28.1
## 2               Angola  27.0  27.0  27.0  27.0  27.0  27.0  27.0  27.0  27.0
## 3              Albania  35.4  35.4  35.4  35.4  35.4  35.4  35.4  35.4  35.4
## 4              Andorra    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 5 United Arab Emirates  30.7  30.7  30.7  30.7  30.7  30.7  30.7  30.7  30.7
```

`sub.iris` is called a **long table**. We can reshape this **long table** to a **wide table** using `spread()` function.

- **gather()**: The function "gathers" multiple columns from the data set and converts them into key-value pairs. `gather()` takes four principal arguments:
  - data set
  - the key column variable we wish to create from column names.
  - the values column variable we wish to create and fill with values associated with the key.

`The names of the columns we use to fill the key variable (or to drop).`

Here we exclude `country` from being `gather()`ed.

```
life.expectancy.long <- life.expectancy %>%
  gather(key = "Year",        # the column names of the wide table
         value = "lifeExp",   # the numerical values of the table
         - country,           # drop country variable: its value will not be gathered (stacked)!
         na.rm = TRUE)        # removing records with missing values
```

```
##
head(life.expectancy.long)
```

```
##                country  Year lifeExp
## 1          Afghanistan X1799    28.2
## 2               Angola X1799    27.0
## 3              Albania X1799    35.4
## 5 United Arab Emirates X1799    30.7
## 6            Argentina X1799    33.2
## 7              Armenia X1799    34.0
```

We can use `substr()` to remove `X` from the variable `Year` as shown in the following code.

```
correct.life.exp.data <- life.expectancy.long %>%
                      mutate(year = substr(Year,2,5)) %>%
                      select(-Year)
head(correct.life.exp.data)
```

```
##                country lifeExp year
## 1          Afghanistan    28.2 1799
## 2               Angola    27.0 1799
## 3              Albania    35.4 1799
## 5 United Arab Emirates    30.7 1799
## 6            Argentina    33.2 1799
## 7              Armenia    34.0 1799
```

For illustrative purposes, we look at a small subset of the iris data set.

```
mini.iris <- iris[c(1, 51, 101), ]
mini.iris
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 51           7.0         3.2          4.7         1.4 versicolor
## 101          6.3         3.3          6.0         2.5  virginica
```

We list (`select`) the columns to be stacked explicitly as arguments of `gather()` in the following code.

```
mini.iris.w2l <- mini.iris %>%
          gather(key = "flower.att", value = "measurement",
           Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
mini.iris.w2l
```

```
##       Species   flower.att measurement
## 1      setosa Sepal.Length         5.1
## 2  versicolor Sepal.Length         7.0
## 3   virginica Sepal.Length         6.3
```

```
## 4      setosa  Sepal.Width         3.5
## 5  versicolor  Sepal.Width         3.2
## 6   virginica  Sepal.Width         3.3
## 7      setosa Petal.Length         1.4
## 8  versicolor Petal.Length         4.7
## 9   virginica Petal.Length         6.0
## 10     setosa  Petal.Width         0.2
## 11 versicolor  Petal.Width         1.4
## 12  virginica  Petal.Width         2.5
```

We can also use `"-"` operator to exclude the column(s) to be `gather()ed` to make the code cleaner.

```
mini.iris.w2l0 <- mini.iris %>%
          gather(key = "flower.att", value = "measurement", -Species)
mini.iris.w2l0
```

```
##       Species   flower.att measurement
## 1      setosa Sepal.Length         5.1
## 2  versicolor Sepal.Length         7.0
## 3   virginica Sepal.Length         6.3
## 4      setosa  Sepal.Width         3.5
## 5  versicolor  Sepal.Width         3.2
## 6   virginica  Sepal.Width         3.3
## 7      setosa Petal.Length         1.4
## 8  versicolor Petal.Length         4.7
## 9   virginica Petal.Length         6.0
## 10     setosa  Petal.Width         0.2
## 11 versicolor  Petal.Width         1.4
## 12  virginica  Petal.Width         2.5
```

- **spread()**: takes two columns and "spreads" them into multiple columns. It takes three principal arguments:
  - the data
  - the key column `(categorical)` variable whose values will become new column names.
  - the value column `(numerical or categorical)` variable whose values will fill the new column variables.

Further arguments include filling which, if set, fills in missing values with the value provided.

```
mini.iris.l2w <- mini.iris.w2l %>%
  spread(key = "flower.att", value = "measurement")
head(mini.iris.l2w)
```

```
##      Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1     setosa          1.4         0.2          5.1         3.5
```

```
## 2 versicolor          4.7         1.4         7.0         3.2
## 3  virginica          6.0         2.5         6.3         3.3
```

Chapter 9

Introduction to Ggplot

The data properties are typically numerical or categorical values, while the visual properties include the x and y positions of points, colors of lines, heights of bars, and so on. The process of creating a data visualization is to map the data properties to visual properties.

In R's base graphics functions, each mapping of data properties to visual properties is its special case. Changing the mappings in the base R graphics may require restructuring the data utilizing completely different plotting commands, or both.

On the other hand, `ggplot2` is a system for declaratively creating graphics, based on The Grammar of Graphics. We provide the data, and tell ggplot2 how to map variables to aesthetics and what graphical primitives to use, `ggplot()` takes care of the details.

The graphic functions in base R are powerful, but in general, it is believed that `ggplot()` is better.

For those who program in Python, It is good to know that `plotnine` is an implementation of a grammar of graphics in **Python**, it is based on `ggplot2()`.

For those who program in SAS, the SAS ODS graphics are roughly analogous to R's `ggplot()` although it is not a direct implementation of The Grammar of Graphics.

9.1 Basics of `ggplot()`

Plotting with `ggplot2` is based on "adding" plot layers and design elements on top of one another, with each command added to the previous ones with a plus symbol (`+`). The result is a multi-layer plot object that can be saved, modified, printed, exported, etc.

`ggplot()` objects can be highly complex, but the basic order of layers will usually look like this:

1. Begin with the baseline `ggplot()` command - this "opens" the ggplot and allows subsequent functions to be added with `+`. Typically the data set is also specified in this command

2. Add `"geom"` layers - these functions visualize the data as geometries (shapes), e.g. as a bar graph, line plot, scatter plot, histogram (or a combination!). These functions all start with `geom_` as a prefix.

3. Add design elements to the plot such as axis labels, titles, fonts, sizes, color schemes, legends, or axes rotation

We can check the tidyverse reference site for more details at https://ggplot2.tidyverse.org/reference/index.htm

A simple example of skeleton code is as follows. We will explain each component in the code below.

```
# plot data from my data columns as red points
ggplot(data = my_data)  +               # use the dataset "my_data"
  geom_point(                           # add a layer of points (dots)
    mapping = aes(x = col1, y = col2),  # "map" data column to axes
    color = "red")  +                   # other specification for the geom
  labs()+                               # here you add titles, axes labels, etc.
  theme()                               # here you adjust color, font, size etc
                                        # title, etc.)
```

In the following sections, we will detail each of the components in the above code.

9.2 Structure of `ggplot()`

The opening command of any `ggplot2` plot is `ggplot()`. This command simply creates a blank canvas upon which to add layers. It "opens" the way for further layers to be added with a `+` symbol.

Typically, the command ggplot() includes the `data = argument` for the plot. This sets the default data set to be used for subsequent layers of the plot.

This command will end with a `+` after its closing parentheses. This leaves the command "open". The `ggplot` will only execute/appear when the full command includes a final layer **without** a `+` at the end.

```
# This will create a plot that is a blank canvas
ggplot(data = linelist)
```

9.3 Geoms

The above code creates a blank canvas. We need to create geometries (shapes) from our data (e.g. bar plots, histograms, scatter plots, box plots).

This is done by adding layers of "geoms" to the initial `ggplot()` command. Many `ggplot2` functions create "geoms". Each of these functions begins with "geom_", so we will refer to them generically as `geom_XXXX()`.

There are over 40 geoms in ggplot2 and many others created by fans. View them at the `ggplot2` gallery. Some common `geoms` are listed below:

- Histograms - `geom_histogram()`
- Bar charts - `geom_bar()` or `geom_col()`
- Box plots - `geom_boxplot()`
- Points (e.g. scatter plots) - `geom_point()`
- Line graphs - `geom_line()` or `geom_path()`
- Trend lines - `geom_smooth()`

We can display one or multiple `geoms` in one plot. Each is added to previous `ggplot2` commands with a `+`, and they are plotted sequentially such that later `geoms` are plotted on top of previous ones.

9.4 Mapping Data to Plot

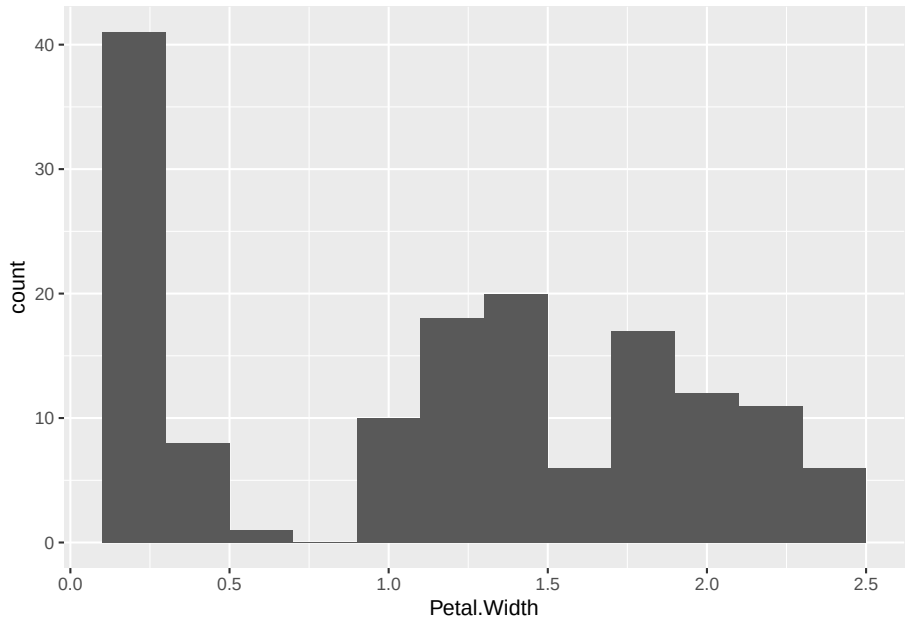
`geom` functions require mapping (assigning) columns in the data to components of the plot like the axes, shape colors, shape sizes, etc. The mappings must be wrapped in the `aes()` function, so we would write something like `mapping = aes(x = col1, y = col2)`.

For example, in the following example using `iris data`, Sepal Length is mapped to the x-axis, and Sepal Width is mapped to the y-axis. After a +, the plotting commands continue. A shape is created with the "geom" function geom_point().

```
ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point()
```

When creating a histogram, only one variable is used. See the following example.

```
ggplot(data = iris, mapping = aes(x = Petal.Width)) +
  geom_histogram(binwidth = 0.2)
```

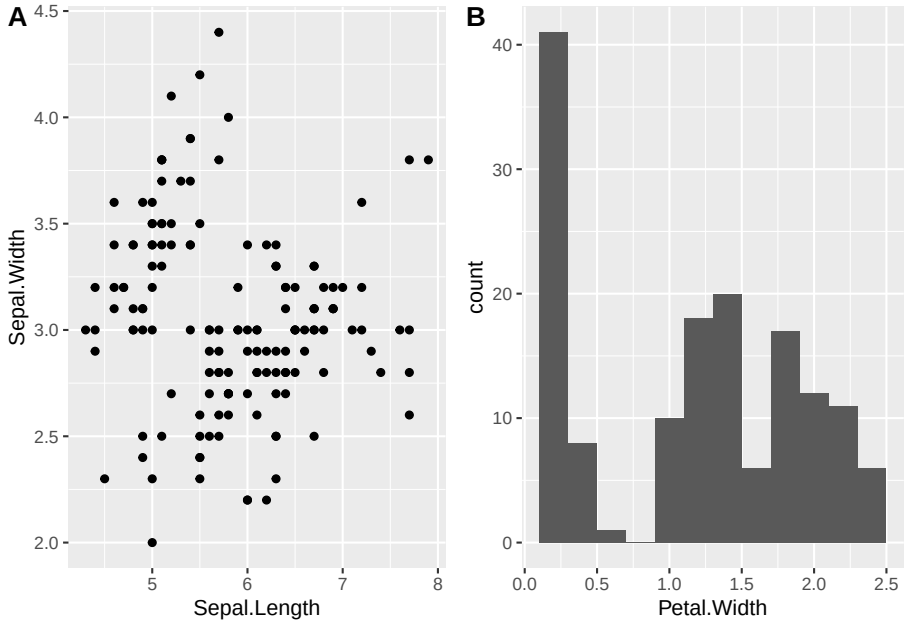


9.4.1 Arranging Multiple Grobs on the Same Page

In the above subsection, we create two graphs on two different pages. Sometimes, we want to place two more graphs on the same page for comparison purposes. In base R, we have graphic functions such as `par()` and `layout()` to set up a layout for the graphic page.

In this note, we introduce library `cowplot` to arrange multiple graphical objects (a.k.a `grobs`) on a page.

```
## name the two plots first and then call the two grobs in the layout function
## scatter plot
scatter = ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point()
## histogram
hist = ggplot(data = iris, mapping = aes(x = Petal.Width)) +
  geom_histogram(binwidth = 0.2)
## use plot_grid() in {cowplot} to layout the two plots
plot_grid(scatter, hist, labels=c("A", "B"), ncol = 2, nrow = 1)
```



9.4.2 Plot Aesthetics

In `ggplot` terminology a plot "aesthetic" has a specific meaning. It refers to colors, sizes, transparencies, placement, etc. of the plotted data. `Not all geoms will have the same aesthetic options`, but many can be used by most `geoms`.

Here are some examples:

- `shape` = Display a point with `geom_point()` as a `dot`, `star`, `triangle`, or `square`, etc.
- `fill` = The interior color (e.g. of a bar or boxplot)
- `color` = The exterior line of a bar, boxplot, etc., or the point color if using `geom_point()`
- `size` = Size (e.g. line thickness, point size)
- `alpha` = Transparency (1 = opaque, 0 = invisible)
- `binwidth` = Width of histogram bins
- `width` = Width of "bar plot" columns
- `linetype` = Line type (e.g. `solid`, `dashed`, `dotted`)

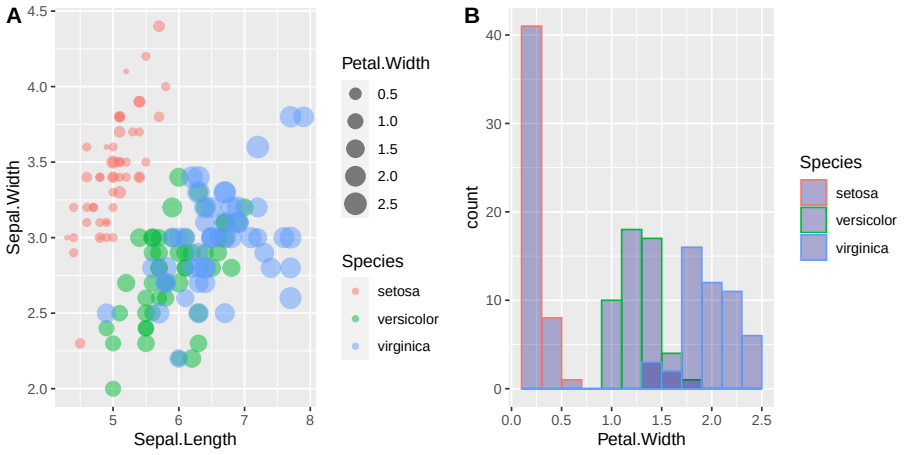
The aesthetics of plot objects can be assigned values in two ways:

1. Assigned a static value (e.g. color = "blue") to apply across all plotted observations

2. Assigned to a column of the data (e.g. color = hospital) such that the display of each observation depends on its value in that column

We have already added binwidth to the above histogram. Next, we add color to the histogram

```
# Change histogram plot line colors by groups
scatter01 <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                              color = Species,
                              size = Petal.Width)) +
  geom_point(alpha = 0.5)
# Overlaid histograms
hist01 <- ggplot(iris, aes(x = Petal.Width, color=Species)) +
          geom_histogram(fill="navy",
                         alpha = 0.3,
                         position = "identity",
                         binwidth = 0.2)
## use plot_grid() in {cowplot} to layout the two plots
plot_grid(scatter01, hist01, labels=c("A", "B"), ncol = 2, nrow = 1)
```



9.4.3 Labels in `ggplot()`

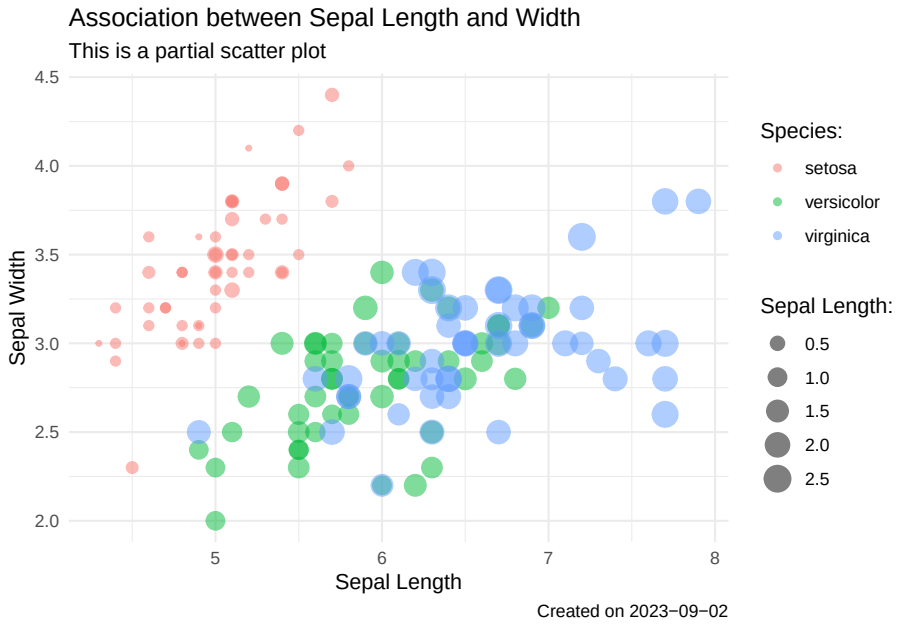
Surely you will want to add or adjust the plot's labels. These are most easily done within the `labs()` function which is added to the plot with `+` just as the `geoms` were.

Within `labs()` you can provide character strings to these arguments:

- `x =` and `y =`: The x-axis and y-axis title (labels)
- `title =`: The main plot title
- `subtitle =`: The subtitle of the plot, in smaller text below the title
- `caption =`: The caption of the plot, in bottom-right by default

Here is a plot we made earlier, but with nicer labels:

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                                color = Species,
                                size = Petal.Width)) +
              geom_point(alpha = 0.5) +
              labs(
                  x = "Sepal Length",
                  y = "Sepal Width",
                  # label for legends
                  size = "Sepal Length:",
                  color = "Species:",
                  title = "Association between Sepal Length and Width",
                  subtitle = "This is a partial scatter plot",
                  caption = paste("Created on", Sys.Date())) +
              theme_minimal()    # minimal theme
```



9.4.4 Themes in `ggplot()`

The theme system in `ggplot()` does not affect how the data is rendered by `geoms`, or how it is transformed by scales. `Themes` don't change the perceptual properties of the plot, but they do help you make the plot aesthetically pleasing or match an existing style guide. `Themes` give us control over things like fonts, ticks, panel stripes, and backgrounds.

In other words, when creating the plot we determine how the data is displayed, and then after it has been created we can edit every detail of the rendering, using the theming system.

The theming system is composed of four main components:

- Theme elements specify the non-data elements that we can control. For example,
  - `plot.title` controls the appearance of the plot title;
  - `axis.ticks.x` controls the ticks on the x-axis;
  - `legend.key.height` controls the height of the keys in the legend.
- Each element is associated with an element function, which describes the visual properties of the element. For example, `element_text()` sets the font size, color and face of text elements like `plot.title`.
- The `theme()` function which allows you to override the default theme elements by calling element functions, like `theme(plot.title = element_text(colour = "red"))`.

- Complete themes, like `theme_grey()` set all of the theme elements to values designed to work together harmoniously.

Here are some especially common theme() arguments. You will recognize some patterns, such as appending .x or .y to apply the change only to one axis.

`theme()` argument	What it adjusts
`plot.title = element_text()`	The title
`plot.subtitle = element_text()`	The subtitle
`plot.caption = element_text()`	The caption (family, face, color, size, angle, vjust, hjust…)
`axis.title = element_text()`	Axis titles (both x and y) (size, face, angle, color…)
`axis.title.x = element_text()`	Axis title x-axis only (use `.y` for y-axis only)
`axis.text = element_text()`	Axis text (both x and y)
`axis.text.x = element_text()`	Axis text x-axis only (use `.y` for y-axis only)
`axis.ticks = element_blank()`	Remove axis ticks
`axis.line = element_line()`	Axis lines (colour, size, linetype: solid dashed dotted etc)
`strip.text = element_text()`	Facet strip text (colour, face, size, angle…)
`strip.background = element_rect()`	facet strip (fill, colour, size…)

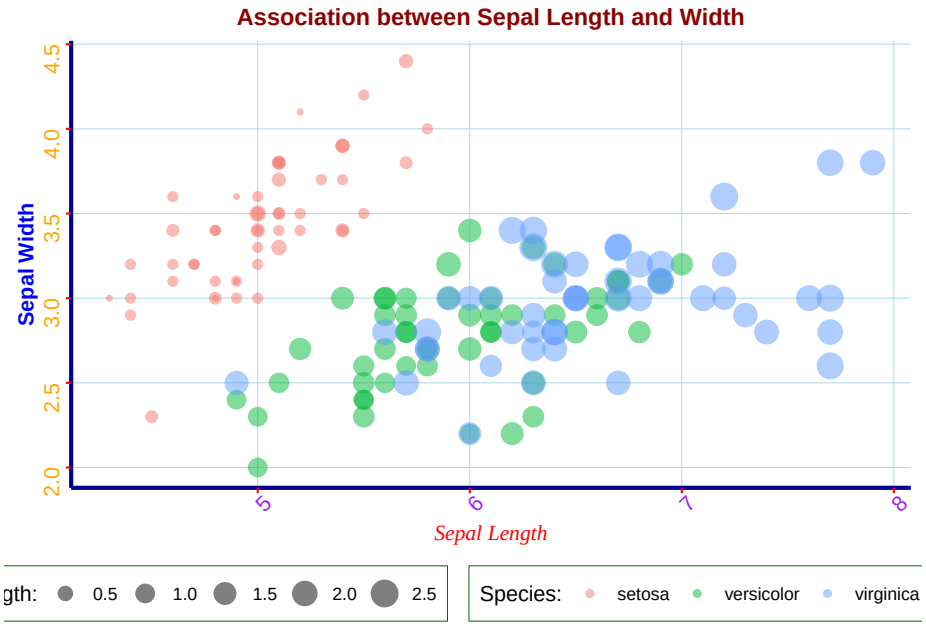
To get the complete list of themes, run the following code

```
#theme_get()
```

To make sure the plot can stand alone, we need to provide the plot with axes, legend labels, title, and tweaking the color scale for appropriate colors.

```
# adding themes
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                                color = Species,
                                size = Petal.Width)) +
              geom_point(alpha = 0.5) +
              labs(
                  x = "Sepal Length",
                  y = "Sepal Width",
                  # label for legends
                  size = "Sepal Length:",
                  color = "Species:",
                  title = "Association between Sepal Length and Width" ) +
              theme_minimal()  +   # minimal theme
              theme( # list of themes applied to the plot
                    # plot title features
                    # font family: c("sans", "serif", "mono")
                    # font face: c("plain", "bold", "italic", "bold.italic")
                    plot.title = element_text(face = "bold",
```

```
                                        size = 12,
                                        family = "sans",
                                        color = "darkred",
                                        hjust = 0.5), # left(0),right(1)
                    # Labels of axes
                    axis.title.x = element_text(color = "red",
                                                face = "italic",
                                                family = "serif",
                                                hjust = 0.5),
                    axis.title.y = element_text(color = "blue",
                                                face = "bold",
                                                vjust = 0.5),
                    axis.ticks = element_line(color = "red",
                                              size = 0.5),
                    axis.line = element_line(color = "darkblue",
                                             size = 1,
                                             linetype = "solid"),
                    # Axis tick marks
                    axis.text.x = element_text(face="plain",
                                               color="purple",
                                               size=11,
                                               angle=45),
                    axis.text.y = element_text(face="plain",
                                               color="orange",
                                               size=11,
                                               angle=90),
                    # Features of legend
                    legend.background = element_rect(fill = "white",
                                                     size = 0.1,
                                                     color = "darkgreen"),
                    legend.justification = c(0.9, 0.8),
                    legend.position = "bottom",
                    ## Panel grid
                    panel.grid.major = element_line(color = "lightblue",
                                                    size = 0.1),
                    panel.grid.minor = element_blank()
)
```



9.4.5 Complete Components of Theme

Themes are a powerful way to customize the `non-data components` of the plots: i.e. titles, labels, fonts, background, gridlines, and legends. To give our plots a consistent customized look, we can define a theme function and call the theme function in any `ggplots`.

The `tidyverse` official website provides a comprehensive document on theme components in 'ggplot". https://ggplot2.tidyverse.org/reference/theme.html. Numerous examples have illustrated how to use various theme components.

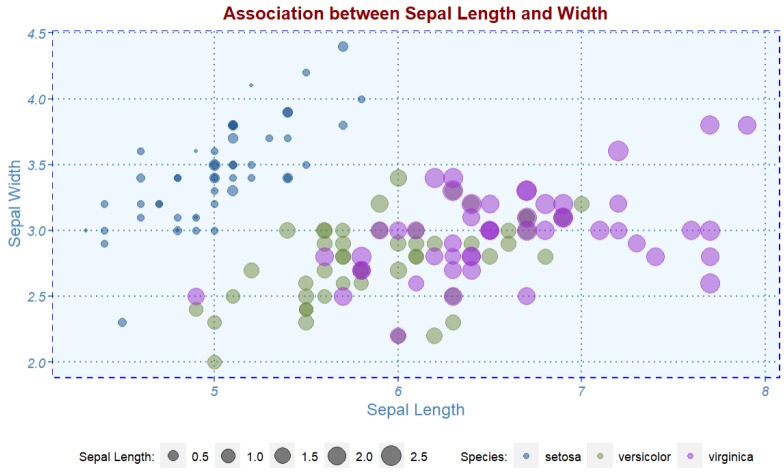
We can define a theme function that can be reused to customize the plots. For example, we define the following theme and use it in different plots.

```
myplot.theme <- function() {
  theme(
    plot.title = element_text(face = "bold",
                              size = 12,
                              family = "sans",
                              color = "darkred",
                              hjust = 0.5), # left(0),right(1)
    # add border 1)
    panel.border = element_rect(colour = "blue",
                                fill = NA,
                                linetype = 2),
    # color background 2)
```

```
    panel.background = element_rect(fill = "aliceblue"),
    # modify grid 3)
    panel.grid.major.x = element_line(colour = "steelblue",
                                      linetype = 3,
                                      size = 0.5),
    panel.grid.minor.x = element_blank(),
    panel.grid.major.y =  element_line(colour = "steelblue",
                                       linetype = 3,
                                       size = 0.5),
    panel.grid.minor.y = element_blank(),
    # modify text, axis and colour 4) and 5)
    axis.text = element_text(colour = "steelblue",
                             face = "italic",
                             family = "Times New Roman"),
    axis.title = element_text(colour = "steelblue",
                              family = "Times New Roman"),
    axis.ticks = element_line(colour = "steelblue"),
    # legend at the bottom 6)
    legend.position = "bottom",
    legend.key.size = unit(0.6, 'cm'), #change legend key size
    legend.key.height = unit(0.6, 'cm'), #change legend key height
    legend.key.width = unit(0.6, 'cm'), #change legend key width
    legend.title = element_text(size=8), #change legend title font size
    legend.text = element_text(size=8)) #change legend text font size
}
```

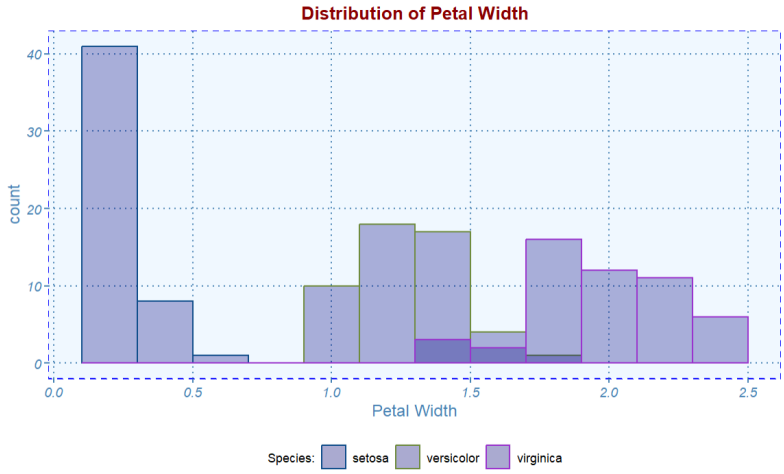
Now we use the above theme in the following scatter plots. Instead of using the colors based on the value of species, we manually select colors to encode the values of species. The following URL links to a PDF document with colors in R. http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

```
# Change histogram plot line colors by groups
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                                color = factor(Species),
                                size = Petal.Width)) +
            geom_point(alpha = 0.5) +
            scale_color_manual(values=c("dodgerblue4", "darkolivegreen4","darkorchid3
            labs(
                x = "Sepal Length",
                y = "Sepal Width",
                ## labels of color and size
                size = "Sepal Length:",
                color = "Species:",
                title = "Association between Sepal Length and Width") +
            myplot.theme()
```



Next, we plot a histogram using the same theme.

```
ggplot(iris, aes(x = Petal.Width, color=Species)) +
          geom_histogram(fill="navy",
                         alpha = 0.3,
                         position = "identity",
                         binwidth = 0.2) +
  scale_color_manual(values=c("dodgerblue4", "darkolivegreen4",
                              "darkorchid3")) +
               labs(
                 x = "Petal Width",
                 color = "Species:",
                 title = "Distribution of Petal Width") +
           myplot.theme()
```



9.5 Adding Annotations to Graphics

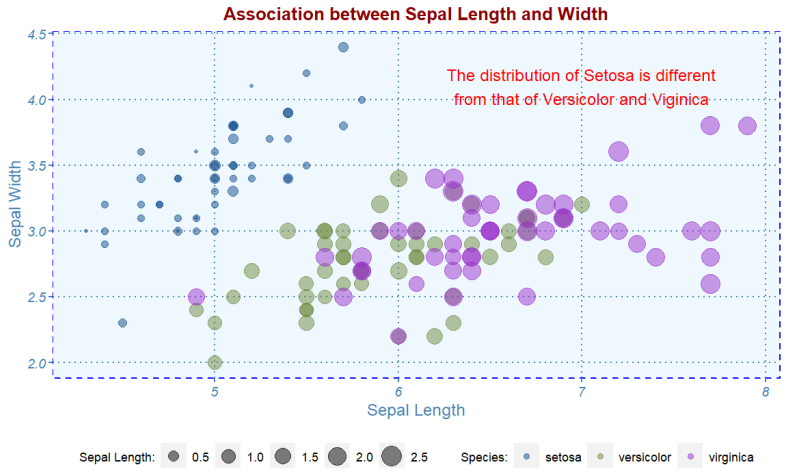
To make the graphic more informative, sometimes we may want to add annotations to the graphic. If we create a statistical and probabilistic graphic, occasionally we need to add mathematical equations with Greek letters to the graphics.

9.5.1 Adding Text Annotation to Graphics

9.5.1.1 Adding Plain Text to Graphics

To add plain text to graphics in ggplot, we use the function `annotate()` with given coordinates. For example, the scatter plot shows two separate groups.

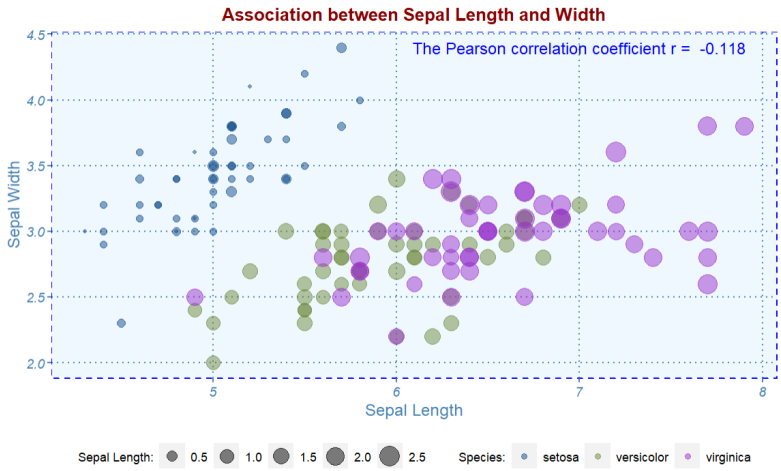
```
# Change histogram plot line colors by groups
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                              color = factor(Species),
                              size = Petal.Width)) +
             geom_point(alpha = 0.5) +
             scale_color_manual(values=c("dodgerblue4", "darkolivegreen4", "darkorchid3
             labs(
                 x = "Sepal Length",
                 y = "Sepal Width",
                 ## labels of color and size
                 size = "Sepal Length:",
                 color = "Species:",
                 title = "Association between Sepal Length and Width") +
             myplot.theme() +
             annotate(geom="text",
                      x=7,
                      y=4.1,
                      label=paste("The distribution of Setosa is different",
                      "from that of Versicolor and Viginica", sep = "\n"),
                      color="red",
                      hjust = 0.5)
```



Several other alternatives we can use to add text to graphics created using 'ggplot".

9.5.1.2 Passing Parameters in Annotation

```
# Change histogram plot line colors by groups
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                              color = factor(Species),
                              size = Petal.Width)) +
             geom_point(alpha = 0.5) +
             scale_color_manual(values=c("dodgerblue4", "darkolivegreen4", "darkorchid3")) +
             labs(
                 x = "Sepal Length",
                 y = "Sepal Width",
                 ## labels of color and size
                 size = "Sepal Length:",
                 color = "Species:",
                 title = "Association between Sepal Length and Width") +
             myplot.theme() +
              annotate(geom="text" ,
                       x=7,
                       y=4.4,
                       label=paste("The Pearson correlation coefficient r = ",
                                   round(cor(iris$Sepal.Length, iris$Sepal.Width),3)),
                          color = "blue")
```



The correlation coefficient between sepal width and sepal length is calculated directly from the data and passed to the annotation in the graphic. Note that we used a very handy and important graphic function `paste()` when adding the annotation.

9.5.2 Adding Mathematical Equations to Graphics

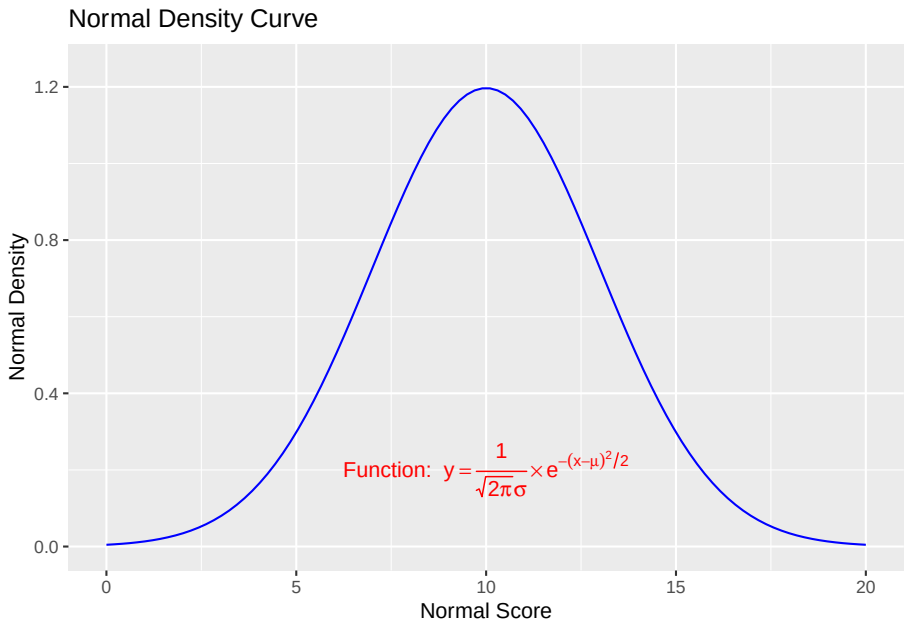
Mathematical expressions made with the text `geoms` using `parse = TRUE` in `ggplot2` have a format similar to those made with `plotmath()` and `expression()` in base R, except that they are stored as strings, rather than as expression objects.

To mix regular text with expressions, use single quotes within double quotes (or vice versa) to mark the plain-text parts. Each block of text enclosed by the inner quotes is treated as a variable in a mathematical expression.

Bear in mind that, in R's syntax for mathematical expressions, we **can't** simply put a variable right next to another without something else in between. To display two variables next to each other, put a `*` operator between them. When `*` is displayed in a graphic, it is treated as an invisible multiplication sign (for a visible multiplication sign, use `%*%`):

```
x.axis <- seq(0, 20, length.out = 100)
y.axis <- (1/sqrt(2*pi)*3)*exp(-(x.axis-10)^2/(2*9))
normal.data = data.frame(x=x.axis , y=y.axis)
##
ggplot(normal.data, aes(x = x.axis, y = y.axis)) +
     geom_line(color = "blue") +
     coord_cartesian(ylim = c(0, 1.25), xlim=c(0,20)) +
     labs(
                  x = "Normal Score",
```

```
                y = "Normal Density",
                title = "Normal Density Curve") +
      annotate("text", x = 10, y = 0.2,
               parse = TRUE, size = 4,
              label = "'Function:  ' * y==frac(1, sqrt(2*pi)* sigma) %*% e^{-(x- mu)^2/2}",
              color = "red")
```

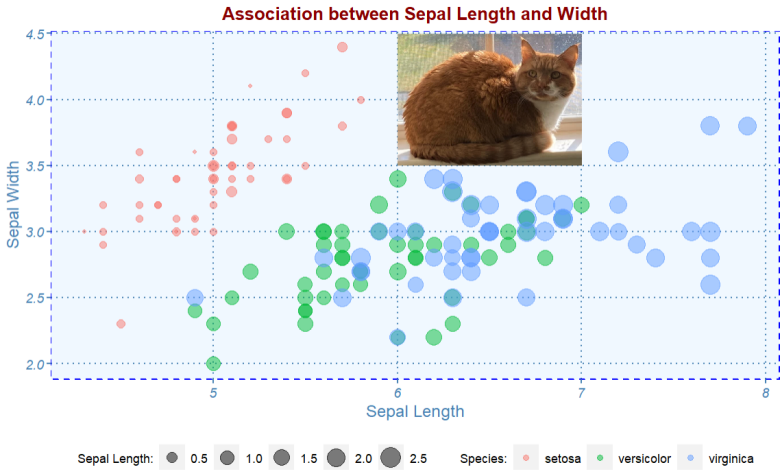


9.5.3 Adding Images to Existing ggPlots

To embed a PNG image to an existing graph created by `ggplot`, we need to use `readPNG()` in library **png** to load the image to R and `getURLcontent()` in the **RCurl** to insert the image to the graph.

```
library("png")
my_cat <-  readPNG('img05/cat.png')
raster.cat <- as.raster(my_cat)
# Change histogram plot line colors by groups
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                              color = factor(Species),
                              size = Petal.Width)) +
             geom_point(alpha = 0.5) +
             labs(
                 x = "Sepal Length",
                 y = "Sepal Width",
                 ## labels of color and size
```

```
      size = "Sepal Length:",
      color = "Species:",
      title = "Association between Sepal Length and Width") +
myplot.theme() +
 annotation_raster(raster.cat, 6, 7, 3.5, 4.5)
```



9.5.4 Removing Chart Junks

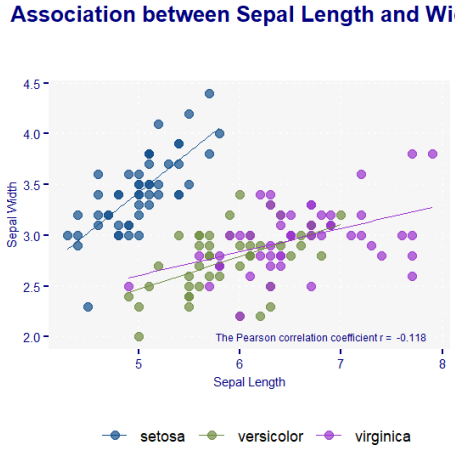
We remove some unnecessary marks and channels from the chart via theme: change the background color, grid, and plot title.

```
myplot.theme_new <- function() {
  theme(
    #ggplot margins
     plot.margin = margin(t = 50,  # Top margin
                          r = 30,  # Right margin
                          b = 30,  # Bottom margin
                          l = 30), # Left margin
    ## ggplot titles
    plot.title = element_text(face = "bold",
                              size = 12,
                              family = "sans",
                              color = "navy",
                              hjust = 0.5,
                              margin=margin(0,0,30,0)), # left(0),right(1)
    # add border 1)
    panel.border = element_rect(colour = NA,
                                fill = NA,
                                linetype = 2),
    # color background 2)
```

```
    panel.background = element_rect(fill = "#f6f6f6"),
    # modify grid 3)
    panel.grid.major.x = element_line(colour = 'white',
                                      linetype = 3,
                                      size = 0.5),
    panel.grid.minor.x = element_blank(),
    panel.grid.major.y =  element_line(colour = 'white',
                                       linetype = 3,
                                       size = 0.5),
    panel.grid.minor.y = element_blank(),
    # modify text, axis, and color 4) and 5)
    axis.text = element_text(colour = "navy",
                             #face = "italic",
                             size = 7,
                             #family = "Times New Roman"
                             ),
    axis.title = element_text(colour = "navy",
                              size = 7,
                              #family = "Times New Roman"
                              ),
    axis.ticks = element_line(colour = "navy"),
    # legend at the bottom 6)
    legend.position = "bottom",
    legend.key.size = unit(0.6, 'cm'), #change legend key size
    legend.key.height = unit(0.6, 'cm'), #change legend key height
    legend.key.width = unit(0.6, 'cm'), #change legend key width
    #legend.title = element_text(size=8), #change legend title font size
    legend.title=element_blank(),  # remove all legend titles
    legend.key = element_rect(fill = "white"),
    #####
    legend.text = element_text(size=8)) #change legend text font size
}
```

```
# Change histogram plot line colors by groups
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                              color = factor(Species)), linetype = Species) +
             geom_point(size = 2, alpha = 0.7) +
             stat_smooth(method = lm, se=FALSE, size = 0.3) +
             scale_color_manual(values=c("dodgerblue4", "darkolivegreen4", "darkorchid3")) +
             labs(
                 x = "Sepal Length",
                 y = "Sepal Width",
                 ## labels of color and size
                 #size = "Sepal Length",
                 #color = NA,
```

```
    title = "Association between Sepal Length and Width") +
myplot.theme_new() +
 annotate(geom="text" ,
          x=6.8,
          y=2,
          label=paste("The Pearson correlation coefficient r = ",
                      round(cor(iris$Sepal.Length, iris$Sepal.Width),3)),
             size = 2,
             color = "navy") +
  coord_fixed(1)    ## This changes the aspect ratio of the graph
```



9.6 Aminated Graph with `gganimate()`

`gganimate()` extends the grammar of graphics as implemented by `ggplot2` to include the description of animation. It does this by providing a range of new grammar classes that can be added to the plot object in order to customize how it should change with time.

- `transition_*()` defines how the data should be spread out and how it relates to itself across time.
- `view_*()` defines how the positional scales should change along with the animation.
- `shadow_*()` defines how data from other points in time should be presented at the given point in time.
- `enter_*()/exit_*()` defines how new data should appear and how old data should disappear during the course of the animation.
- `ease_aes()` defines how different aesthetics should be eased during tran-

sitions.

The logic behind the gganimate is to create a sequence of images and then make a gif image. We need to write HTML to include this gif in the RMarkdown document.

```
library(gapminder)

p <- ggplot(gapminder, aes(x = gdpPercap,
                           y=lifeExp,
                           size = pop,
                           colour = country)) +
        geom_point(aes(size = pop, ids = country ),
                   show.legend = FALSE,
                   alpha = 0.7) +
        scale_color_viridis_d() +      # color pallets
        scale_size(range = c(2, 12)) +
        scale_x_log10() +
        labs(x = "GDP per capita",
             y = "Life expectancy") +
        ## gganimate command
       transition_time(year)
##
anim_save("LifeExp.gif", p)
#  animate(p, renderer = gifski_renderer())  # this command will pop-up a new graphic window show
```

https://github.com/pengdsci/sta553/raw/main/ggplot/LifeExp.gif

Since the gif image is made of individual static images, it is different from the interactive plot presented in the previous sections that have the capability of showing mode information of the data via hover message.

The next gif graph consists of 5 panels, each representing a continent. They are also fig images. Therefore, no hover message is available for these gif figures.

We use the {gifki} package to render the images in the form of gif and then include the gif image into the RMarkdown document directly.

```
w <- ggplot(gapminder, aes(gdpPercap, lifeExp,
                   size = pop, colour = country)) +
           geom_point(alpha = 0.7, show.legend = FALSE) +
           scale_colour_manual(values = country_colors) +
           #scale_color_manual(values=c("dodgerblue4", "darkolivegreen4","darkorchid3")) +
           #scale_color_brewer(palette="Set1") +
           scale_size(range = c(2, 12)) +
           scale_x_log10() +
           # break down the previous single plot by continent
           # facet_wrap(~continent) +     # create multiple panels according to the continents
           # Here comes the gganimate specific bits
```

```
           labs(title = 'Year: {frame_time}',
                    x = 'GDP per capita',
                    y = 'life expectancy') +
           transition_time(year) +
           ease_aes('linear')
###
animate(w, renderer = gifski_renderer(),
          rewind = TRUE)
```

https://raw.githubusercontent.com/pengdsci/sta553/main/ggplot/LifeExpRewind.gif

The above code does not save the generated gif image to the document folder (directory). If need to save it from the viewer window to the designated folder and then embed it to a web page create by tools other than the RMarkdown.

Next, we create a group gif using facet_wrap() function. The code is the same as the above example except for one additional function call.

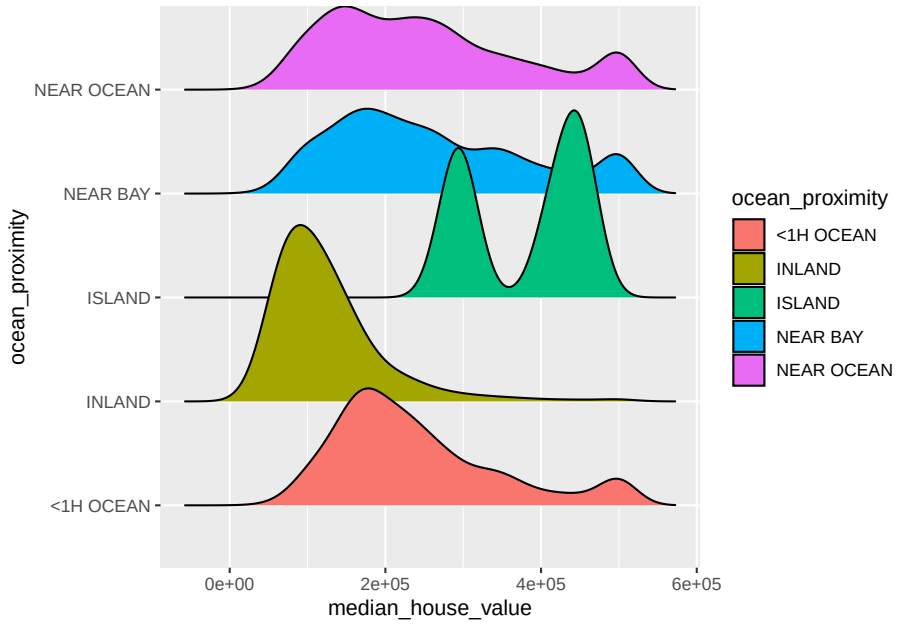
```
w <- ggplot(gapminder, aes(gdpPercap, lifeExp,
                  size = pop, colour = country)) +
           geom_point(alpha = 0.7, show.legend = FALSE) +
           scale_colour_manual(values = country_colors) +
           #scale_color_manual(values=c("dodgerblue4", "darkolivegreen4","darkorchid3"
           #scale_color_brewer(palette="Set1") +
           scale_size(range = c(2, 12)) +
           scale_x_log10() +
           # break down the previous single plot by continent
           facet_wrap(~continent) +     # create multiple panels according to the cont
           # Here comes the gganimate specific bits
           labs(title = 'Year: {frame_time}',
                    x = 'GDP per capita',
                    y = 'life expectancy') +
           transition_time(year) +
           ease_aes('linear')
###
animate(w, renderer = gifski_renderer(),
          rewind = TRUE)
```

The above code generates the same gif image and sends it to the preview window.

9.7 Ridgetline Plot with ggridges Library

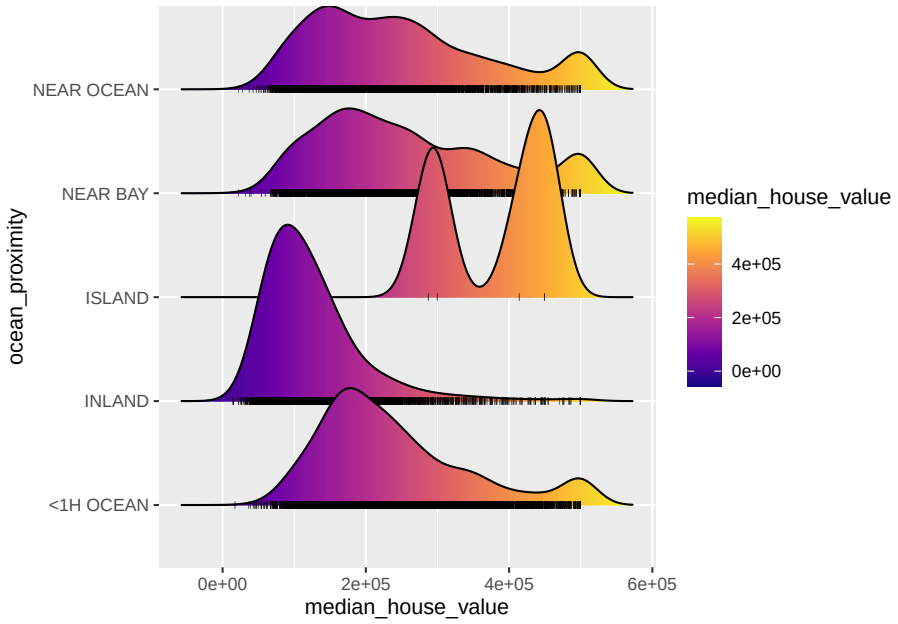
The ridgeline plot is a useful 3D like compare multiple densities. It creates 3D impression and has gained increasing popularity. Here we use the California Housing Data that is available on the Project Data Set https://projectdat.s3.amazonaws.com/datasets.html.

```
CalHousing = read.csv("https://raw.githubusercontent.com/pengdsci/sta553.html/main/data/ca-housir
ggplot(CalHousing, aes(x = median_house_value, y = ocean_proximity, fill = ocean_proximity)) +
  geom_density_ridges()
```



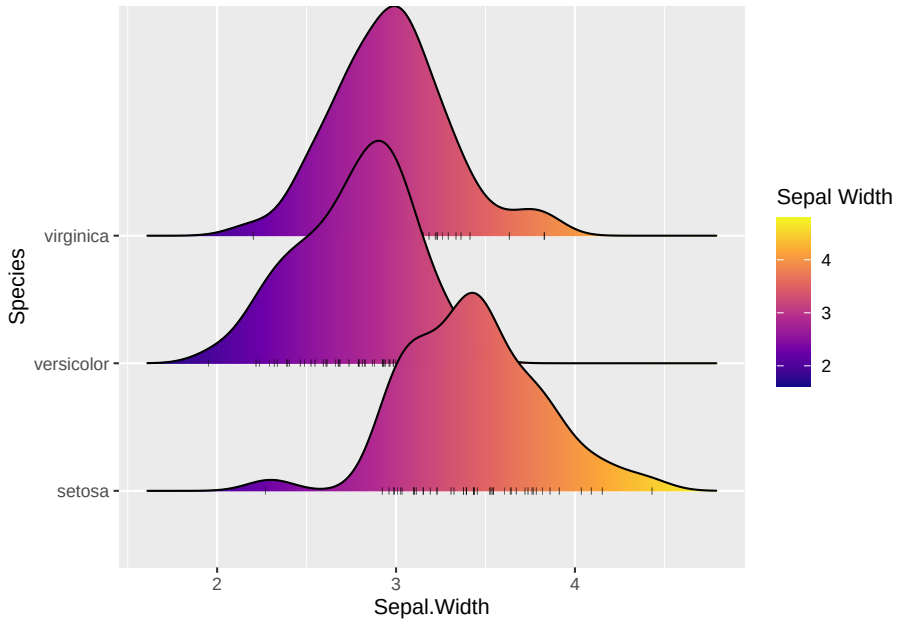
You can pass `stat(x)` or `factor(stat(x))` to the fill argument of `aes` and use `geom_density_ridges_gradient` and a continuous fill color scale to fill each ridgeline with a gradient.

```
ggplot(CalHousing, aes(x = median_house_value, y = ocean_proximity, fill = stat(x))) +
      geom_density_ridges_gradient(jittered_points = TRUE,
                                   position = position_points_jitter(width = 0.05, height
    point_shape = '|', point_size = 1, point_alpha = 1, alpha = 0.3,) +
  scale_fill_viridis_c(name = "median_house_value", option = "C")
```



Next we explore the distribution of continuous variable in the iris data set. As an example, we make the following ridgeline plot to see the distribution of sepal widths across the species.

```
ggplot(iris, aes(x = Sepal.Width, y = Species, fill = stat(x))) +
      geom_density_ridges_gradient(jittered_points = TRUE,
                                   position = position_points_jitter(width = 0.
    point_shape = '|', point_size = 1, point_alpha = 1, alpha = 0.3,) +
  scale_fill_viridis_c(name = "Sepal Width", option = "C")
```



The above distributions have the similar shapes (variations) but with different means. This also indicates the ANOVA model between sepal width and species is appropriate.

9.8 Other Extensions to ggplot

We have used ggplot extensions **{gganimate}** to create animated graphs and **{ggridges}** to create ridgeline graphs to compare multiple densities. There are several other important ggplot extensions that enhance the basic ggplots.

- ggdendro - controls the appearance and display of your cluster analyses
- ggthemes - contains themes and scales that enhance the standard ggplots.
- ggpubr - makes it easy to produce publication-ready plots using ggplot.
- Plotly - bring interactivity to ggplots. We will spend a week on plotly().
- patchwork - arrange multiple R plots on the same graphics page
- ggmap - is a powerful package for visualizing spatial data and models. It layers data on top of static maps from popular online sources. We will use these packages to make maps later.
- ggrepel - to give ggplot2 users greater control over how text labels appear in their charts.
- ggcorrplot - control the appearance of the matrix, from altering the color,

shape, or size of the boxes (as in the circle-matrix above), to adding coefficient labels, reordering the matrix according to hierarchical clustering, and so on.

- GGally - brings together many useful additional visualization functionality, all in one package.
- ggiraph -is htmlwidget that can be extended to an existing ggplot2 bar chart, scatterplot, boxplot, map, etc., and do things like displaying a tooltip of your choice.

9.9 Save `ggplot` Images

A `ggplot` can be saved to different file formats, including PDF, SVG vector files, PNG, TIFF, JPEG, etc.

We can either print directly a `ggplot` into `PNG/PDF` files or use the convenient function `ggsave()` for saving a `ggplot`.

The default of `ggsave()` is to export the last plot that you displayed, using the size of the current graphics device. It also guesses the type of graphics device from the extension.

9.9.1 General Steps

The standard procedure to save any graphics from R is as follows:

- Open a graphic device using one of the following functions:
  - `pdf("r-graphics.pdf")`,
  - `svg("r-graphics.svg")`,
  - `png("r-graphics.png")`,
  - `tiff("r-graphics.tiff")`,
  - `jpeg("r-graphics.jpg")`, etc.
- Additional arguments indicating the width and the height (in inches) of the graphics region can be also specified in the mentioned function.
- Create and print a plot. Close the graphic device using the function `dev.off()`.

9.9.2 Save `ggplot` into a PDF File

The following code illustrates how to save a ggplot in a folder in PDF format.

```
# scatter plots
iris.scatter <- ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
          geom_point()
## box-plot
iris.boxplot <- ggplot(iris, aes(Species, Sepal.Length)) +
```

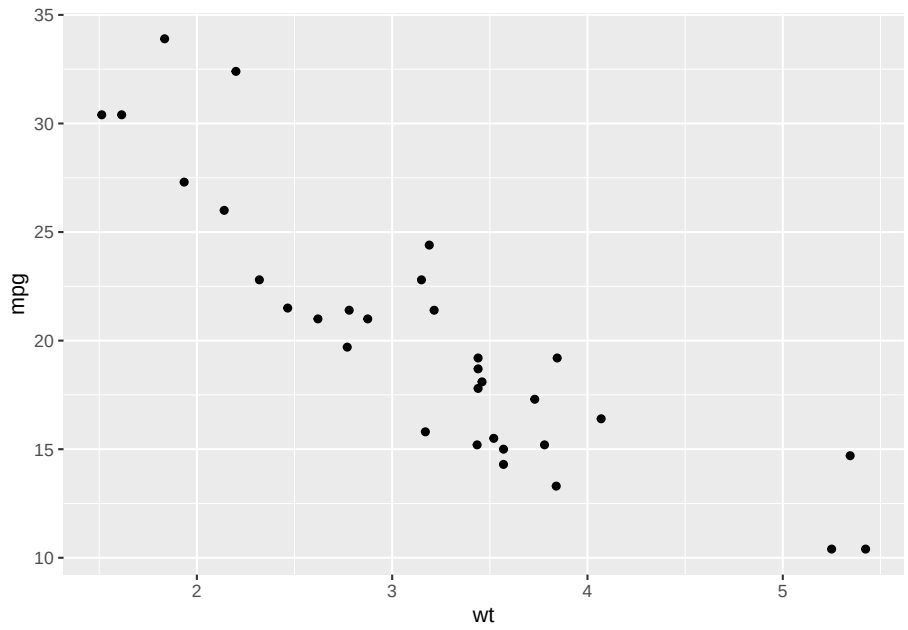
```
  geom_boxplot()
# Print plots to a pdf file: one page per PDF file
pdf("savePDFggplot.pdf")   # save the PDF file in ggplot folder.
print(iris.scatter)      # Plot 1 --> in the first page of PDF
print(iris.boxplot)      # Plot 2 ---> in the second page of the PDF
dev.off()
```

```
## pdf
##   2
```

9.10 Save ggplot with ggsave()

It's also possible to make a ggplot and save it from the screen using the function ggsave().

```
# 1. Create a plot: displayed on the screen (by default)
ggplot(mtcars, aes(wt, mpg)) + geom_point()
```




```
# 2.1. Save the plot to a pdf
ggsave("mtcarmyplot.pdf")
# 2.2 OR save it to png file
ggsave("mtcarmyplot.png")
```

We can also save multiple plots in the sample format to a single file. We can use `plot_grid()` in **{cowplot}** to make two figures on the same graphic page

and then use `ggsave()` to save it to a single file.

```
#
p1 <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
p2 <- ggplot(mtcars, aes(wt)) + geom_histogram()
combinedPlot <- plot_grid(p1, p2, labels=c("A", "B"),
                          ncol = 2, nrow = 1)
##
ggsave("CombinedPlot.png", plot = combinedPlot)
```

Chapter 10

Interactive Statistics Graphics

This note is all about interactive plots. However, interactive plots cannot be rendered in the PDF and EPUB. Screenshots will be included in the PDF and EPUB versions of this eBook. The HTML version of this eBook will keep the interactivity of all graphics! The code that generated the corresponding plots is still included in all versions of this eBook.

10.1 Plotly

Plotly has a rich and complex set of features. The most common features are:

- Tooltip "hover" info
- Zoom in and out of graphs
- Users can export graphs as an image
- Integrating multiple graphs
- Template hover info
- Animations and moving graphics

One can feed a `ggplot` to `plotly` to render ggplot via `plotly`. Compared to the base R plotting function `plot()`, `plot_ly()` is more technical and poorly documented. However, the following factors may make `plotly` the best option:

- Graphs presented in a digital/online format
- Users interact with the graph
- more customizable than ggplot
- rendering graphics in a higher resolution

In this note, we introduce the basic statistical graphics using the `plotly` package.

`plotly` graphics automatically contain interactive elements that allow users to modify, explore, and experience the visualized data in new ways.

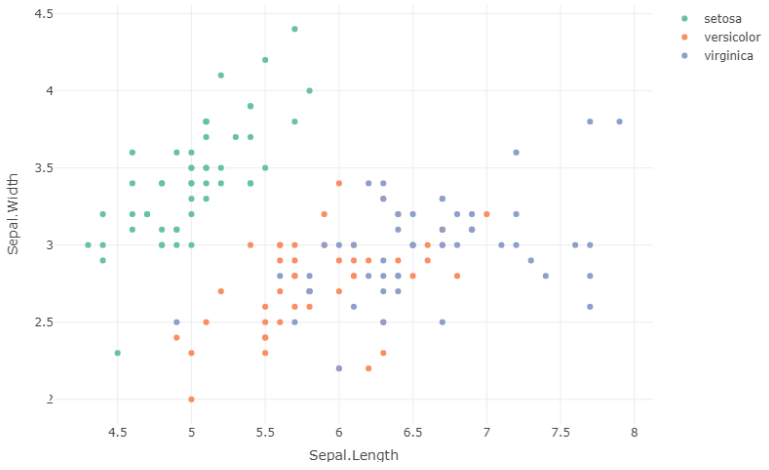
The coding effort is similar to that of SAS ODS graphics. To use `plot_ly()`, we need to install (if not done) and load the `plotly` package. We use the well-known iris data set in the following plots. A nice `plotly` cheat sheet can be found at https://github.com/pengdsci/sta553/blob/main/ref/r_plotly_cheat_sheet.pdf

10.2 ScatterPlot

10.2.1 The Default Plot

First, we make a simple interactive scatter plot using sepal length and width. We can view the information about the variables and color coding information in the hover text. The labels of axes and legend titles and labels are default.

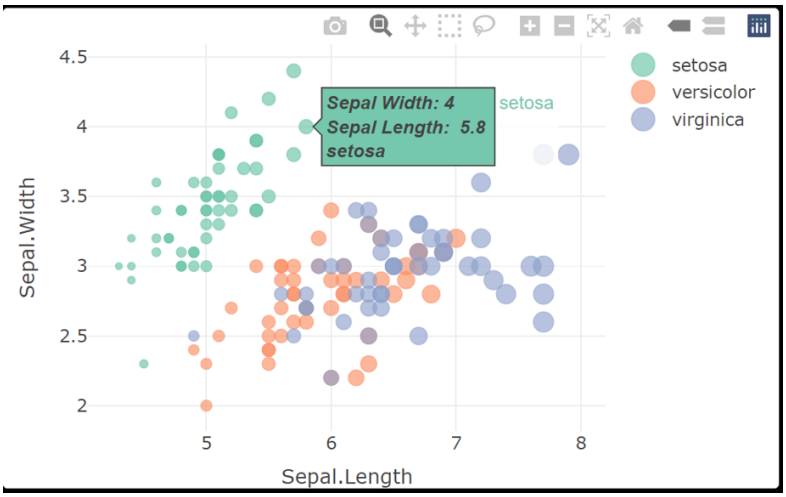
```
plot_ly(
    data = iris,
    x = ~Sepal.Length,  # Horizontal axis
    y = ~Sepal.Width,   # Vertical axis
    color = ~factor(Species),  # must be a numeric factor
     type = "scatter",
     mode = "markers")
```



10.2.2 Addiing Additional Information Through `hovertemplate`

We can also add additional information to the plot to enhance the interactivity of the plot. For example, we can (1) modify the point size using the value of a numerical variable; (2) add text to the hover text using the `text` option to show the class label; (3) formulate the hover text using `hovertemplate` option.

```
plot_ly(
    data = iris,
    x = ~Sepal.Length,  # Horizontal axis
    y = ~Sepal.Width,   # Vertical axis
    color = ~factor(Species),  # must be a numeric factor
     text = ~Species,     # show the species in the hover text
     ## using the following hovertemplate() to add the information of the
     ## two numerical variable to the hover text.
     hovertemplate = paste('<i><b>Sepal Width<b></i>: %{y}',
                           '<br><b>Sepal Length</b>:  %{x}',
                           '<br><b>%{text}</b>'),
     alpha  = 0.9,
     size = ~Sepal.Length,
     type = "scatter",
     mode = "markers")
```



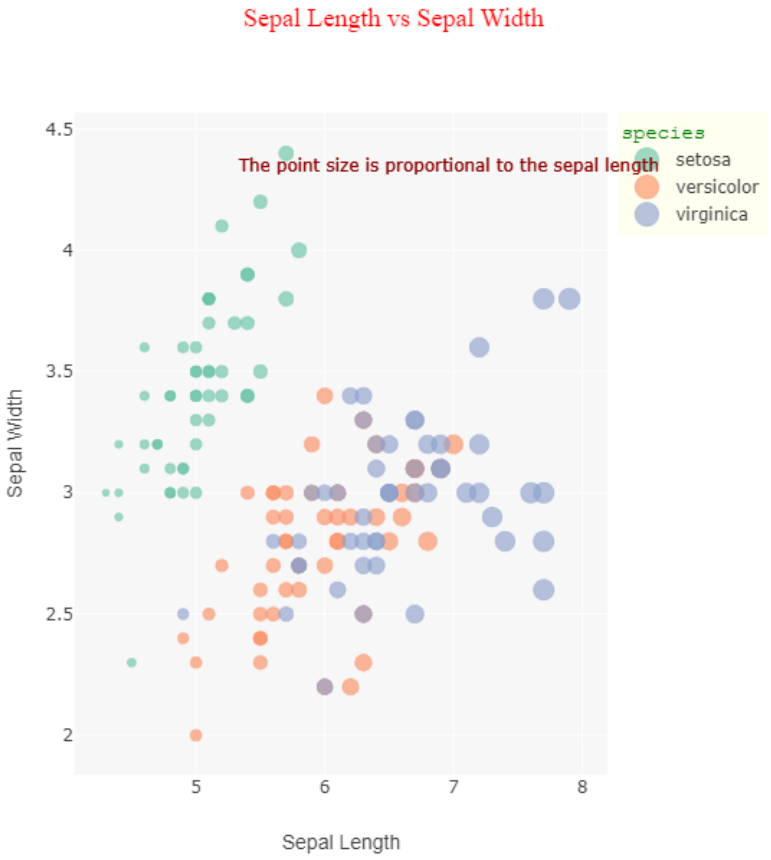
10.2.3 Enhancing the Plot with Layout() Function

Titles and axis labels are important in any visualization, to include a meaningful title, informative labels, and annotations to the plotly plot, we can use the layout() function. The following code only gives you some design ideas you can use to enhance your plotly charts. The detailed list of configurations can be found on plotly's reference page at https://plotly.com/r/reference/layout/

```
plot_ly(
    data = iris,
    x = ~Sepal.Length,  # Horizontal axis
    y = ~Sepal.Width,   # Vertical axis
    color = ~factor(Species),  # must be a numeric factor
     text = ~Species,     # show the species in the hover text
```

```
  ## using the following hovertemplate() to add the information of the
  ## two numerical variable to the hover text.
  hovertemplate = paste('<i><b>Sepal Width<b></i>: %{y}',
                        '<br><b>Sepal Length</b>:  %{x}',
                        '<br><b>%{text}</b>'),
  alpha  = 0.9,
  size = ~Sepal.Length,
  type = "scatter",
  mode = "markers"
) %>%
 layout(
   ## graphic size
   with = 700,
   height = 700,
   ### Title
   title =list(text = "Sepal Length vs Sepal Width",
                       font = list(family = "Times New Roman",  # HTML font family
                                   size = 18,
                                  color = "red")),
   ### legend
   legend = list(title = list(text = 'species',
                              font = list(family = "Courier New",
                                          size = 14,
                                         color = "green")),
                 bgcolor = "ivory",
                 bordercolor = "navy",
                 groupclick = "togglegroup",  # one of  "toggleitem" AND "togglegrou
                 orientation = "v"  # Sets the orientation of the legend.

                 ),
   ## margin of the plot
   margin = list(
           b = 120,
           l = 50,
           t = 120,
           r = 50
   ),
   ## Background
   plot_bgcolor ='#f7f7f7',
   ## Axes labels
          xaxis = list(
                 title=list(text = 'Sepal Length',
                            font = list(family = 'Arial')),
                 zerolinecolor = 'red',
                 zerolinewidth = 2,
                 gridcolor = 'white'),
```

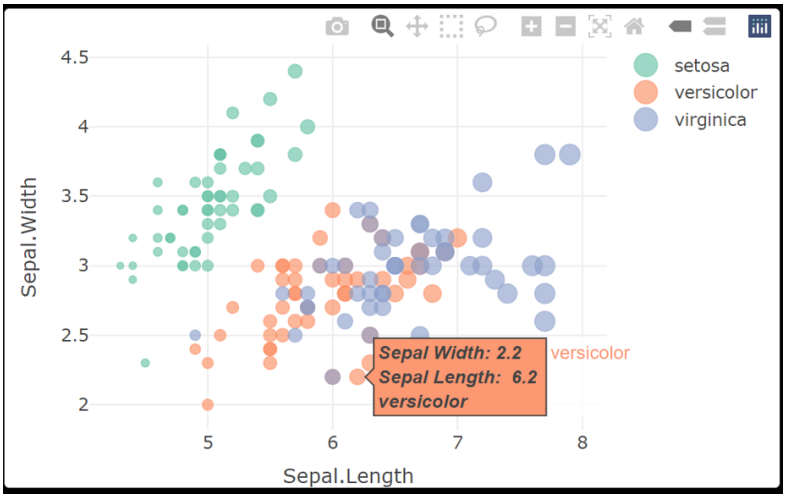
```
            yaxis = list(
                    title=list(text = 'Sepal Width',
                               font = list(family = 'Arial')),
                    zerolinecolor = 'purple',
                    zerolinewidth = 2,
                    gridcolor = 'white'),
      ## annotations
      annotations = list(
                     x = 0.7,   # between 0 and 1. 0 = left, 1 = right
                     y = 0.9,   # between 0 and 1, 0 = bottom, 1 = top
                  font = list(size = 12,
                              color = "darkred"),
                  text = "The point size is proportional to the sepal length",
                  xref = "paper",  # "container" spans the entire `width` of the plot.
                                   # "paper" refers to the width of the plotting area only.
                  yref = "paper",  #  same as xref
                xanchor = "center", #  horizontal alignment with respect to its x position
                yanchor = "bottom", #  similar to xanchor
              showarrow = FALSE
            )
)
```




```
myPlotlyLayout <- function(){
  layout(
      ## graphic size
      with = 700,
      height = 700,
      ### Title
      title =list(text = "Sepal Length vs Sepal Width",
                          font = list(family = "Times New Roman",  # HTML font family
                                      size = 18,
                                     color = "red")),
      ### legend
      legend = list(title = list(text = 'species',
                                 font = list(family = "Courier New",
```

```
                                          size = 14,
                                         color = "green")),
                bgcolor = "ivory",
                bordercolor = "navy",
                groupclick = "togglegroup",  # one of  "toggleitem" AND "togglegroup".
                orientation = "v"  # Sets the orientation of the legend.

                ),
     ## margin of the plot
     margin = list(
             b = 120,
             l = 50,
             t = 120,
             r = 50
     ),
     ## Background
     plot_bgcolor ='#f7f7f7',
     ## Axes labels
            xaxis = list(
                   title=list(text = 'Sepal Length',
                              font = list(family = 'Arial')),
                   zerolinecolor = 'red',
                   zerolinewidth = 2,
                   gridcolor = 'white'),
           yaxis = list(
                   title=list(text = 'Sepal Width',
                              font = list(family = 'Arial')),
                   zerolinecolor = 'purple',
                   zerolinewidth = 2,
                   gridcolor = 'white'),
      ## annotations
      annotations = list(
                    x = 0.7,   # between 0 and 1. 0 = left, 1 = right
                    y = 0.9,   # between 0 and 1, 0 = bottom, 1 = top
                 font = list(size = 12,
                             color = "darkred"),
                 text = "The point size is proportional to the sepal length",
                 xref = "paper",  # "container" spans the entire `width` of the plot.
                                  # "paper" refers to the width of the plotting area only.
                 yref = "paper",  #  same as xref
               xanchor = "center", #  horizontal alignment with respect to its x position
               yanchor = "bottom", #  similar to xanchor
             showarrow = FALSE
           )
 )
     }
```

```
plot_ly(
    data = iris,
    x = ~Sepal.Length,  # Horizontal axis
    y = ~Sepal.Width,   # Vertical axis
    color = ~factor(Species),  # must be a numeric factor
     text = ~Species,     # show the species in the hover text
     ## using the following hovertemplate() to add the information of the
     ## two numerical variable to the hover text.
     hovertemplate = paste('<i><b>Sepal Width<b></i>: %{y}',
                           '<br><b>Sepal Length</b>:  %{x}',
                           '<br><b>%{text}</b>'),
     alpha  = 0.9,
     size = ~Sepal.Length,
     type = "scatter",
     mode = "markers"
   )
```



10.2.4 Rendering A GGPLOT with ggplotly

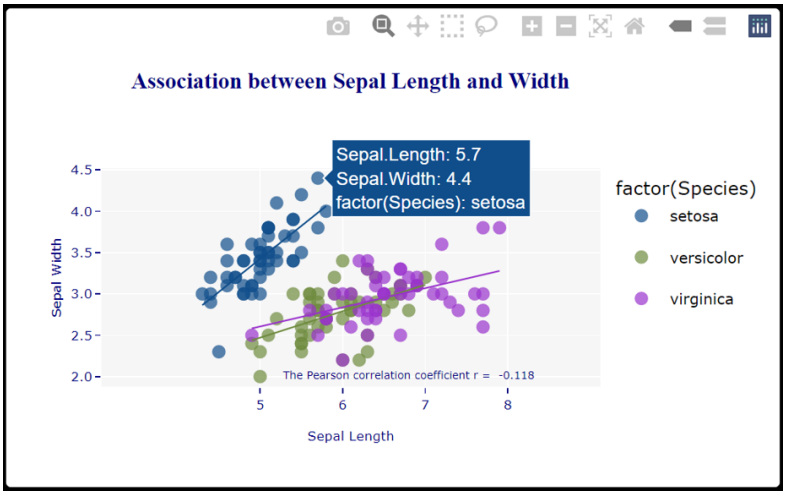
We can also render a ggplot in using ggplotly to bring interactivity to the plot.

```
myplot.theme_new <- function() {
  theme(
    #ggplot margins
     plot.margin = margin(t = 50,  # Top margin
                          r = 30,  # Right margin
                          b = 30,  # Bottom margin
                          l = 30), # Left margin
    ## ggplot titles
    plot.title = element_text(face = "bold",
```

```
                              size = 12,
                              family = "sans",
                              color = "navy",
                              hjust = 0.5,
                              margin=margin(0,0,30,0)), # left(0),right(1)
    # add border 1)
    panel.border = element_rect(colour = NA,
                                fill = NA,
                                linetype = 2),
    # color background 2)
    panel.background = element_rect(fill = "#f6f6f6"),
    # modify grid 3)
    panel.grid.major.x = element_line(colour = 'white',
                                      linetype = 3,
                                      size = 0.5),
    panel.grid.minor.x = element_blank(),
    panel.grid.major.y =  element_line(colour = 'white',
                                       linetype = 3,
                                       size = 0.5),
    panel.grid.minor.y = element_blank(),
    # modify text, axis and colour 4) and 5)
    axis.text = element_text(colour = "navy",
                             #face = "italic",
                             size = 7,
                             #family = "Times New Roman"
                             ),
    axis.title = element_text(colour = "navy",
                              size = 7,
                              #family = "Times New Roman"
                              ),
    axis.ticks = element_line(colour = "navy"),
    # legend at the bottom 6)
    legend.position = "bottom",
    legend.key.size = unit(0.6, 'cm'), #change legend key size
    legend.key.height = unit(0.6, 'cm'), #change legend key height
    legend.key.width = unit(0.6, 'cm'), #change legend key width
    #legend.title = element_text(size=8), #change legend title font size
    legend.title=element_blank(),  # remove all legend titles
    legend.key = element_rect(fill = "white"),
    #####
    legend.text = element_text(size=8)) #change legend text font size
}
```

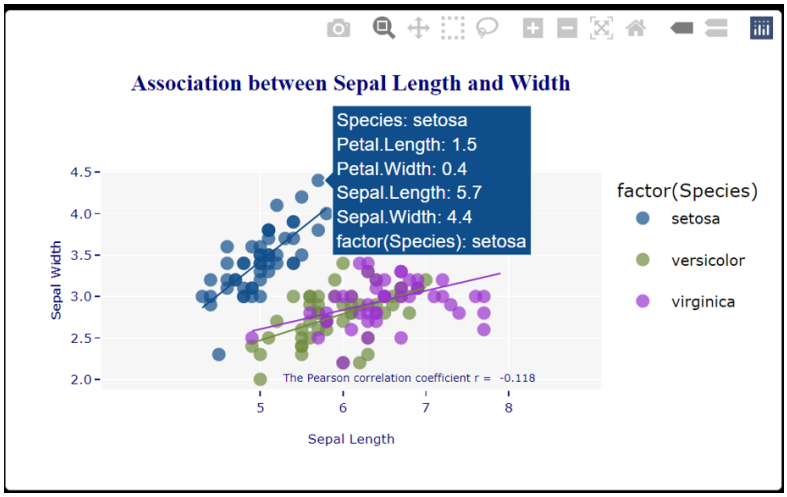
```
# Change histogram plot line colors by groups
p <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
```

```
                      color = factor(Species)), linetype = Species) +
            geom_point(size = 2, alpha = 0.7) +
            stat_smooth(method = lm, se=FALSE, size = 0.3) +
            scale_color_manual(values=c("dodgerblue4", "darkolivegreen4", "darkorchid3
            labs(
                x = "Sepal Length",
                y = "Sepal Width",
                title = "Association between Sepal Length and Width") +
            myplot.theme_new() +
             annotate(geom="text" ,
                      x=6.8,
                      y=2,
                      label=paste("The Pearson correlation coefficient r = ",
                                  round(cor(iris$Sepal.Length, iris$Sepal.Width),3)),
                         size = 2,
                         color = "navy") +
              coord_fixed(1)    ## This changes the aspect ratio of the graph
ggplotly(p)
```




```
# Change histogram plot line colors by groups
p <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                      color = factor(Species)), linetype = Species) +
            # to add more information about the variables in the data set
            # use labels to denote the variable names inside the function aes()
            aes(label=Species, label2=Petal.Length, label3=Petal.Width) +
            geom_point(size = 2, alpha = 0.7) +
            stat_smooth(method = lm, se=FALSE, size = 0.3) +
            scale_color_manual(values=c("dodgerblue4", "darkolivegreen4", "darkorchid3
            labs(
                x = "Sepal Length",
```

```
                y = "Sepal Width",
                title = "Association between Sepal Length and Width") +
            myplot.theme_new() +
             annotate(geom="text" ,
                      x=6.8,
                      y=2,
                      label=paste("The Pearson correlation coefficient r = ",
                                  round(cor(iris$Sepal.Length, iris$Sepal.Width),3)),
                         size = 2,
                         color = "navy") +
              coord_fixed(1)    ## This changes the aspect ratio of the graph
ggplotly(p)
```



10.3 Barplot

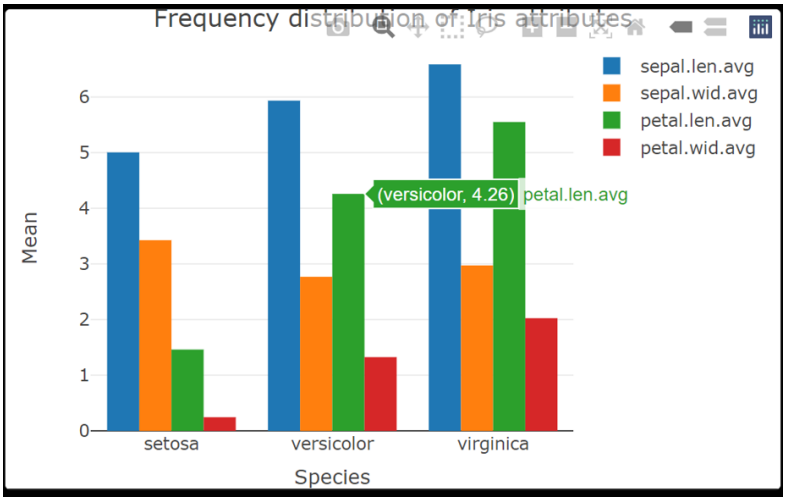
We will create a summarized data set to make bar plots. We define a data set to store the mean of sepal length and sepal width by species using the `dyplr` and `tidyr` approaches.

```
barplotdata <- iris %>%
  group_by(Species) %>%
  summarize(sepal.l.avg = mean(Sepal.Length),
            sepal.w.avg = mean(Sepal.Width),
            petal.l.avg = mean(Petal.Length),
            petal.w.avg = mean(Petal.Width))
kable(head(barplotdata))
```

Species	sepal.l.avg	sepal.w.avg	petal.l.avg	petal.w.avg
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026

Next, we draw a group bar chart.

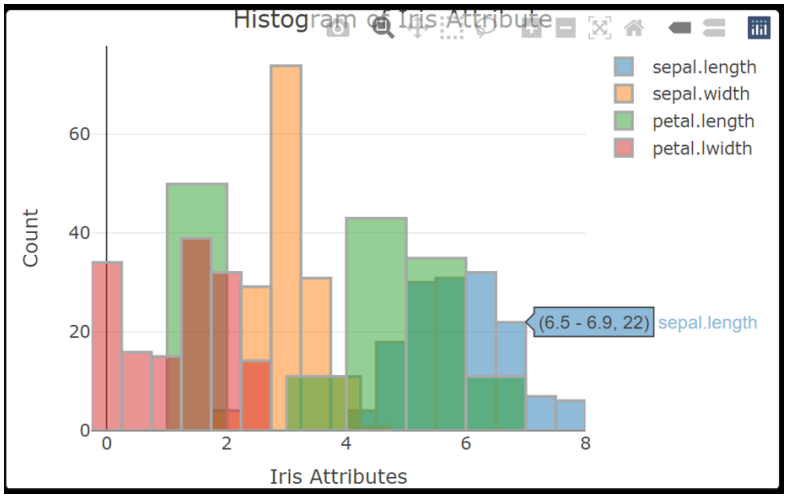
```
plot_ly(
  data = barplotdata,
   x = ~Species,
   y = ~sepal.l.avg,
   type = "bar",
   name = "sepal.len.avg" ) %>%
    add_trace(y=~sepal.w.avg, name = "sepal.wid.avg") %>%
    add_trace(y=~petal.l.avg, name = "petal.len.avg") %>%
    add_trace(y=~petal.w.avg, name = "petal.wid.avg") %>%
    layout( yaxis = list(title ="Mean"),
            title = "Frequency distribution of Iris attributes")
```



10.4 Histogram

```
plot_ly(
  data = iris,
   x = ~ Sepal.Length,
   type = "histogram",
   nbinsx = 10,
   name = "sepal.length",
   alpha = .5,
   marker = list(line = list(color = "darkgray",  width = 2)) ) %>%
   ## adding additional histograms and stack them
```

```
    add_histogram(x = ~Sepal.Width,
                  name = "sepal.width", nbinsx = 10, alpha = 0.5,
                  marker = list(line = list(color = "darkgray",  width = 2))) %>%
    add_histogram(x = ~Petal.Length,
                  name = "petal.length",nbinsx = 10, alpha = 0.5,
                  marker = list(line = list(color = "darkgray",  width = 2))) %>%
    add_histogram(x = ~Petal.Width,
                  name = "petal.lwidth",nbinsx = 10, alpha = 0.5,
                  marker = list(line = list(color = "darkgray",  width = 2))) %>%
  layout(barmode = "overlay",
         title = "Histogram of Iris Attribute",
         xaxis = list(title = "Iris Attributes",
                      zeroline = TRUE),
         yaxis = list(title = "Count",
                      zeroline =TRUE))
```

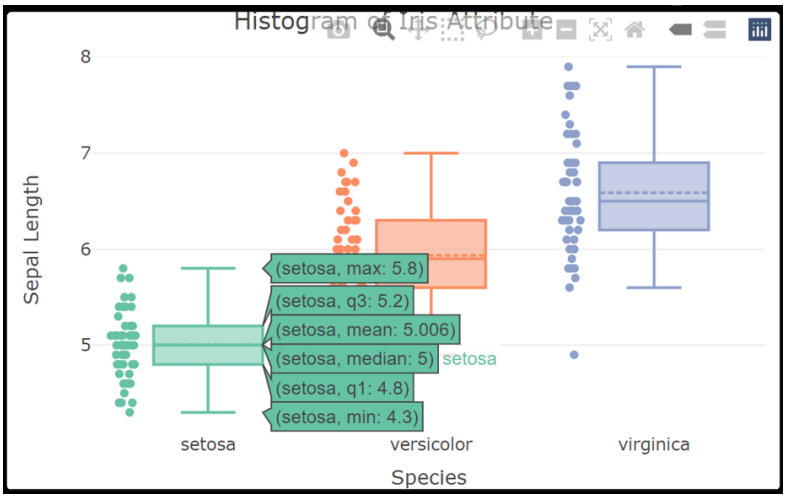


10.5 Boxplot

Drawing a boxplot is straightforward in `plotly`.

```
plot_ly(
  data = iris,
  y = ~ Sepal.Length,
  x = ~Species,
  type = "box",
  color = ~Species,
  boxpoints = "all",
  boxmean = TRUE,
  showlegend = FALSE ) %>%
   layout(title = "Histogram of Iris Attribute",
```

```
        xaxis = list(title = "Species",
                     zeroline = TRUE),
        yaxis = list(title = "Sepal Length",
                     zeroline =TRUE))
```

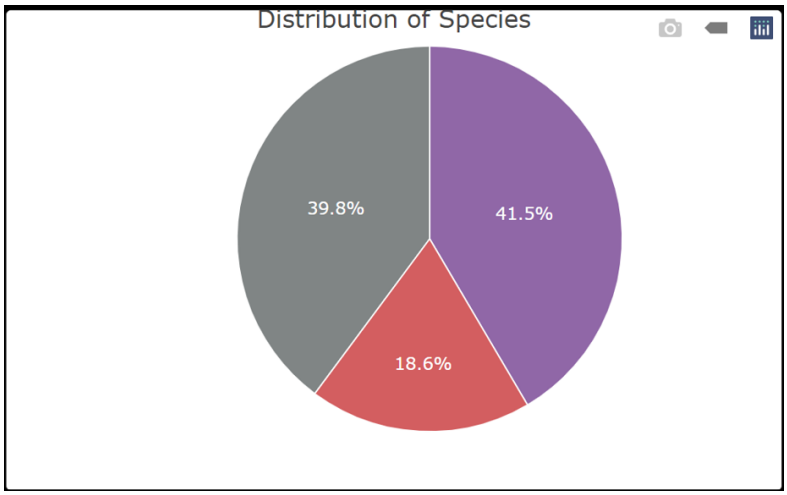


10.6 Pie Chart

We first define a subset from the iris data by filtering out observations with a sepal length of less than 5. The pie chart will be created to see the distribution of species in the subset of the iris data. Keep in mind that the pie chart is constructed based on a frequency table.

```
# define a working data set
subiris <- iris[iris$Sepal.Length > 5,5]
## create a frequency table in the form of data frame.
piedata = data.frame(cate =as.vector(unique(subiris)),
                     freq = as.vector(table(subiris)))
# define a color vector
colors <- c('rgb(211,94,96)', 'rgb(128,133,133)', 'rgb(144,103,167)')
# make a pie chart
plot_ly(piedata, labels = ~cate, values = ~freq, type = 'pie',
        textposition = 'inside',
        textinfo = 'label + percent',
        insidetextfont = list(color = '#FFFFFF'),
        hoverinfo = 'text',
        marker = list(colors = colors,
                      line = list(color = '#FFFFFF', width = 1)),
                      #The 'pull' attribute can also be used to create space between tl
        showlegend = FALSE) %>%
         layout(title = 'Distribution of Species',
```

```
              xaxis = list(showgrid = FALSE, zeroline = FALSE,
                           showticklabels = FALSE),
              yaxis = list(showgrid = FALSE, zeroline = FALSE,
                           showticklabels = FALSE))
```

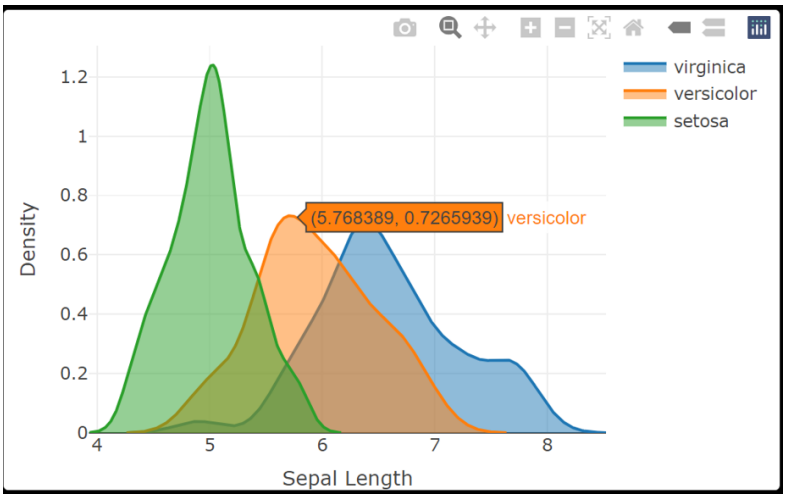


10.7 Density Curve

Assume that we want to compare the distribution of the sepal length of the tree iris flowers. One way to do this comparison is to plot the three estimated density curves.

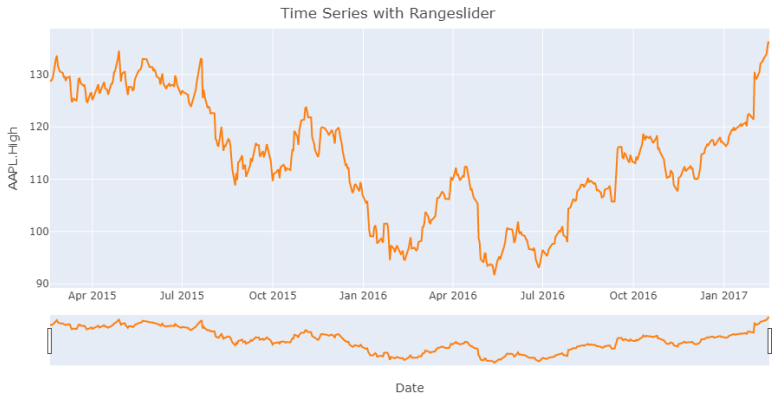
```
# define three densities
sepal.len.setosa <- iris[which(iris$Species == "setosa"),]
setosa <- density(sepal.len.setosa$Sepal.Length)
sepal.len.versicolor <- iris[which(iris$Species == "versicolor"),]
versicolor <- density(sepal.len.versicolor$Sepal.Length)
sepal.len.virginica <- iris[which(iris$Species == "virginica"),]
virginica <- density(sepal.len.virginica$Sepal.Length)
# plot density curves
fig <- plot_ly(x = ~virginica$x, y = ~virginica$y,
               type = 'scatter', mode = 'lines',
               name = 'virginica',
               fill = 'tozeroy')  %>%
           # adding more density curves
       add_trace(x = ~versicolor$x, y = ~versicolor$y,
                 name = 'versicolor', fill = 'tozeroy')  %>%
       add_trace(x = ~setosa$x, y = ~setosa$y,
                 name = 'setosa', fill = 'tozeroy')  %>%
       layout(xaxis = list(title = 'Sepal Length'),
              yaxis = list(title = 'Density'))
```

```
fig
```



10.8 Serial Plot

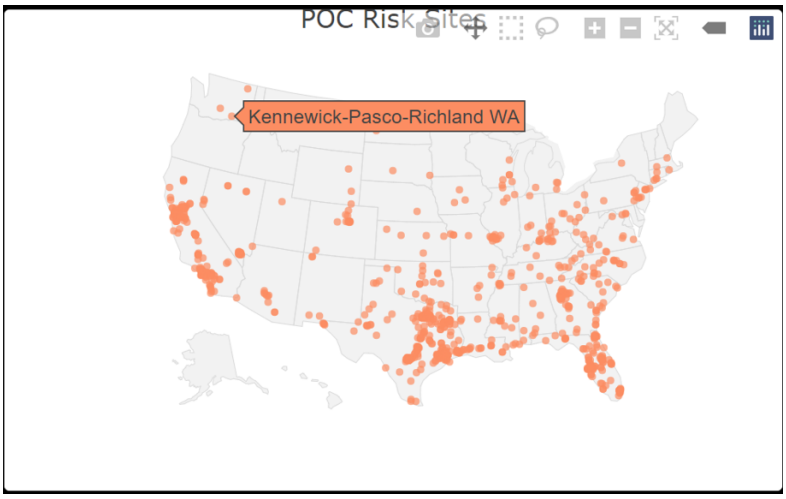
```
stock <- read.csv('https://raw.githubusercontent.com/pengdsci/sta553.html/main/data/fi
##
fig <- plot_ly(stock, type = 'scatter', mode = 'lines')    %>%
       add_trace(x = ~Date, y = ~AAPL.High)    %>%
       layout(showlegend = F,
              title='Time Series with Rangeslider',
              xaxis = list(rangeslider = list(visible = T)))  %>%
       layout(xaxis = list(zerolinecolor = '#ffff',
                       zerolinewidth = 2,
                       gridcolor = 'ffff'),
              yaxis = list(zerolinecolor = '#ffff',
                       zerolinewidth = 2,
                       gridcolor = 'ffff'),
              plot_bgcolor='#e5ecf6', width = 900)
fig
```



10.9 Plotly Maps

Several map libraries are available in R. In this example, we use the `plot_geo()` function from `plotly` to plot on a map.

```
## preparing data
poc <- read_csv("https://raw.githubusercontent.com/pengdsci/sta553.html/main/data/POC.csv")[,c(7,
poc.site <- poc[poc$POC == 1,]
# geo styling
geostyle <- list(scope = 'usa',
                 projection = list(type = 'albers usa'),
                 showland = TRUE,
                 landcolor = toRGB("gray95"),
                 subunitcolor = toRGB("gray85"),
                 countrycolor = toRGB("gray85"),
                 countrywidth = 0.5,
                 subunitwidth = 0.5
               )
## plotting map
fig <- plot_geo(poc.site, lat = ~ycoord, lon = ~xcoord) %>%
       add_markers(text = ~ SITE_DESCRIPTION,
                   color = "red",
                   symbol = I("circle"),
                   size = I(8),
                   hoverinfo = "text" )   %>%
        layout( title = 'POC Risk Sites', geo = geostyle)
fig
```


POC Risk Sites
Kennewick-Pasco-Richland WA

Chapter 11

Interactive Maps

There is a very rich set of tools for interactive geospatial visualization. This note introduces various R tools and Tableau to create interactive maps for visualizing spatial patterns.

11.1 Map Types

There are two common ways of representing spatial data on a map:

- Defining regions on a map and distinguishing them based on their value on some measure using colors and shading. This type of map is usually called `choropleth map`.
- Marking individual points on a map based on their longitude and latitude (e.g., archaeological dig sites; baseball stadiums; voting locations, etc.). This type of map is also called a `scatter map`.

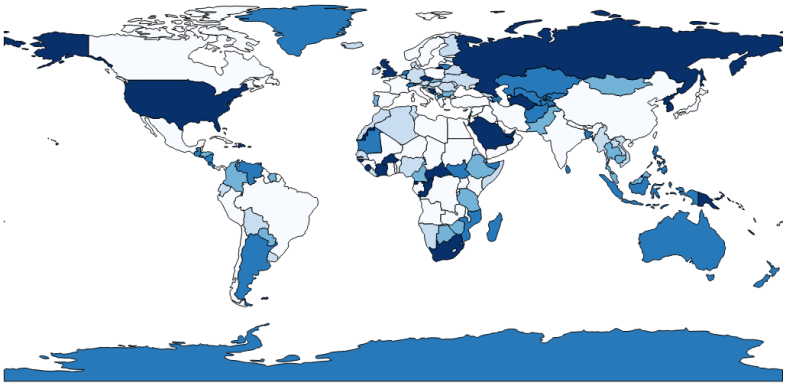
Plotting `scatter maps` uses geocode and is relatively easier to create. However, a choropleth map is constructed using data with a special structure with shape information. It is relatively harder to construct a choropleth map.

A basemap provides context for additional layers that are overlaid on top of the basemap. Basemaps usually provide location references for features that do not change often like boundaries, rivers, lakes, roads, and highways. Even on basemaps, these different categories of information are in layers. Usually, a basemap contains this basic data, and then extra layers with particular infor-

mation from a particular data set, are overlaid on the base map layers for visual analysis.

In this note, the basemaps come primarily from the open-data-source-based OpenStreepMap.

Choropleth Map*



Scatter Map*

11.2 Leaflet Maps

We will use the R `leaflet` library to create both reference maps and choropleth maps and plot data on maps to display spatial patterns.

Reference Maps*

11.2.1 1. Introduction

`Leaflet` is one of the most popular open-source JavaScript libraries for interactive maps. It's used widely in practice. It has many nice features. R package `leaflet` allows us to make interactive maps using map tiles, markers, polygons, lines, popups, etc.

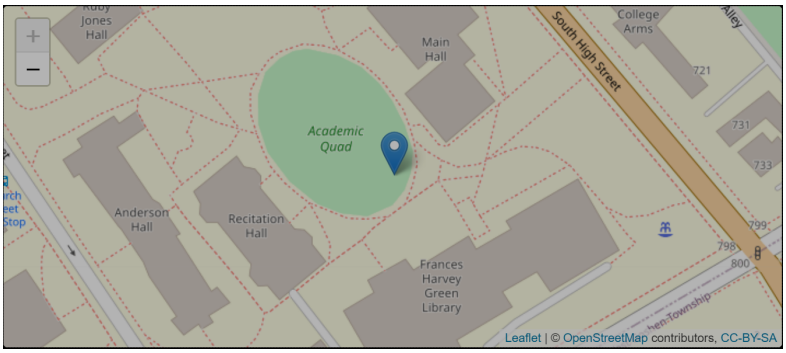
The function `leaflet()` returns a Leaflet map widget, which stores a list of objects that can be modified or updated later. Most functions in this package have an argument map as their first argument, which makes it easy to use the pipe operator `%>%`.

Creating a leaflet map with R library `leaflet` consists of the following steps.

- Create a map widget by calling leaflet().
- Add layers (i.e., features) to the map by using layer functions (e.g. addTiles, addMarkers, addPolygons) to modify the map widget.
- Repeat the previous as desired.
- Print the map widget to display it.

Let's look at the following simple example.

```
# r fig.align='center', fig.height=4, fig.width=6}
# library(leaflet)        # it has been loaded in the setup chunk.
# define a leaflet map
m <- leaflet() %>%
     setView(lng=-75.5978, lat=39.9522, zoom = 20) %>%
     addTiles() %>%        # Add default OpenStreetMap map tiles
     addMarkers(lng=-75.5978, lat=39.9522)
m    # Print the map
```



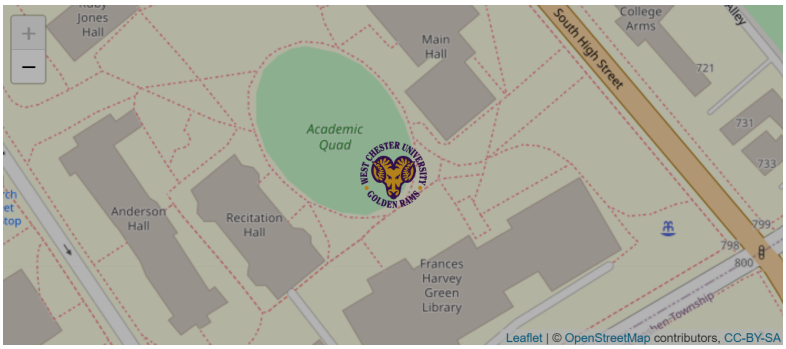
11.2.2 2. Customizing Marker Icons

We can manipulate the attributes of the map widget using a series of methods.

- `setView()` sets the center of the map view and the zoom level;
- `fitBounds()` fits the view into the rectangle [lng1, lat1] – [lng2, lat2];
- `clearBounds()` clear the bound, so that the view will be automatically determined by the range of latitude/longitude data in the map layers if provided;

We can also define our own markers and add them to the map object. For example, we use WCU's logo as a custom marker and add it to the previous map.

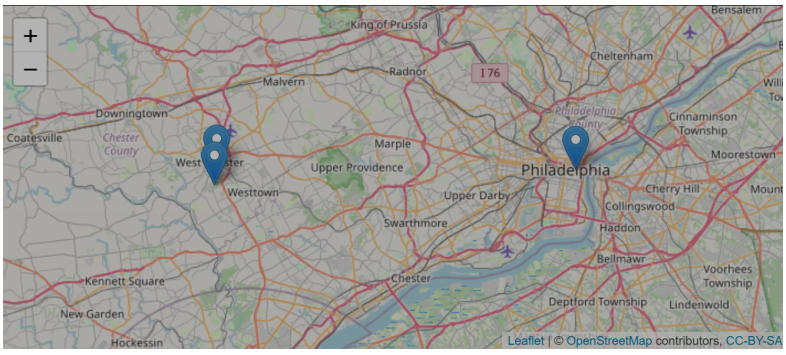
```
#r fig.align='center', fig.height=4, fig.width=6}
# define a marker using WCU's logo.
wcuicon <- makeIcon(
  iconUrl = "https://github.com/pengdsci/sta553/blob/main/image/goldenRamLogo.png?raw=t
  iconWidth = 60, iconHeight = 60
  )
# define a leaflet map
m <- leaflet() %>%
     setView(lng=-75.5978, lat=39.9522, zoom = 20) %>%
     addTiles() %>%        # Add default OpenStreetMap map tiles
     addMarkers(lng=-75.5978, lat=39.9522,  icon = wcuicon)
m    # Print the map
```



11.2.3 3. Popups and Labels

Popups are small boxes containing arbitrary HTML, that point to a specific point on the map. We can use the `addPopups()` function to add a standalone popup to the map. When you click the marker in the following map, you will see a popup with the name of the WCU campus.

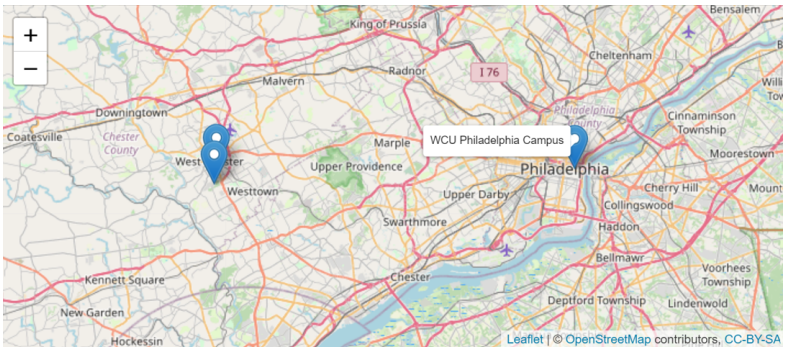
```
#r fig.align='center', fig.height=4, fig.width=6}
df <- read.csv(textConnection(
"Name, Lat, Long
WCU Philadelphia Campus,39.9518,-75.1525
WCU South Campus,39.9373,-75.6011
WCU Main Campus, 39.9524,-75.5982"
))

leaflet(df) %>%
  addTiles() %>%
  setView(lng=-75.3768, lat=39.9448, zoom = 10) %>%
  addMarkers(~Long, ~Lat, popup = ~htmlEscape(Name))
```



We can also change the popups in the above map to labels. The modified code is shown below

```
#r fig.align='center', fig.height=4, fig.width=6}
```

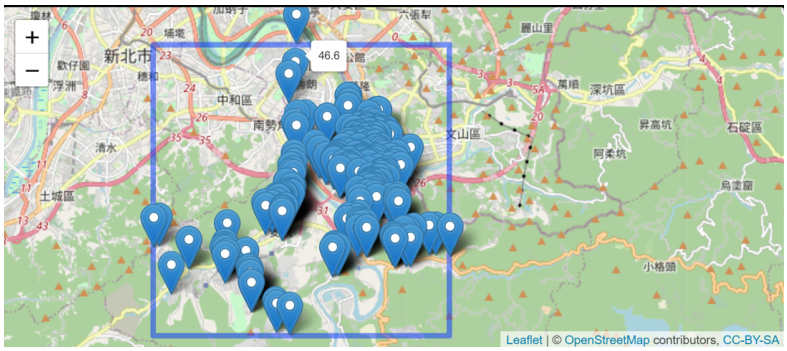
```
df <- read.csv(textConnection(
"Name, Lat, Long
WCU  Philadelphia Campus,39.9518,-75.1525
WCU  South Campus,39.9373,-75.6011
WCU  Main Campus, 39.9524,-75.5982"
))

leaflet(df) %>%
  addTiles() %>%
  setView(lng=-75.3768, lat=39.9448, zoom = 10) %>%
  addMarkers(~Long, ~Lat, label = ~htmlEscape(Name))
```



11.2.4 4. Examples Using Real-World Data

In the following example, we use a few leaflet functions to add some features such as drawing highlight boxes, labels, etc. to the map.

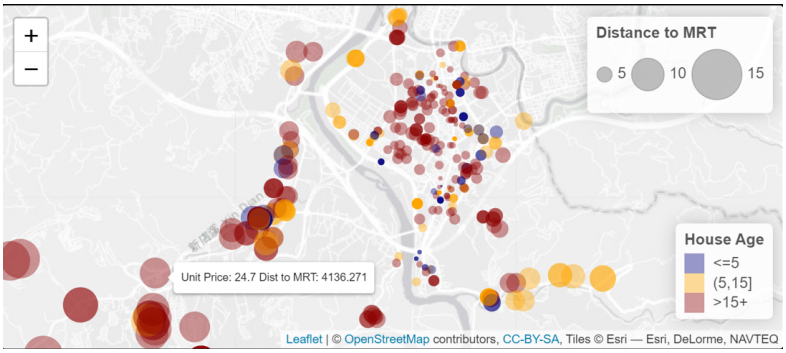
```
#r fig.align='center', fig.height=4, fig.width=6}
# Define the bounding box using the range of longitude/latitude coordinates
# from the given data set
housing.price <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/map/
# making static leaflet map
leaflet(housing.price) %>%
  addTiles() %>%
  setView(lng=mean(housing.price$Longitude), lat=mean(housing.price$Latitude), zoom =
   addRectangles(
    lng1 = min(housing.price$Longitude), lat1 = min(housing.price$Latitude),
    lng2 = max(housing.price$Longitude), lat2 = max(housing.price$Latitude),
    #fillOpacity = 0.2,
    fillColor = "transparent"
    ) %>%
  fitBounds(
    lng1 = min(housing.price$Longitude), lat1 = min(housing.price$Latitude),
    lng2 = max(housing.price$Longitude), lat2 = max(housing.price$Latitude) ) %>%
  addMarkers(~Longitude, ~Latitude, label = ~PriceUnitArea)
```



In the next map based on the data, we add more information to that map to display higher dimensional information.

```
housing.price <- na.omit(read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/map/Rea
## color coding a continuous variable:
colAge <- cut(housing.price$HouseAge, breaks=c(0, 5, 15, max(housing.price$HouseAge)+1), right =
colAgeNum <- as.numeric(colAge)
colors <- rep("navy", length(colAge))
colors[which(colAgeNum==2)] <- "orange"
colors[which(colAgeNum==3)] <- "darkred"
## Define label with hover messages

label.msg <- paste(paste("Unit Price:", housing.price$PriceUnitArea),
                   paste("Dist to MRT:",housing.price$Distance2MRT))

#labels = cat(label.msg)
# making leaflet map
leaflet(housing.price) %>%
  addTiles() %>%
  setView(lng=mean(housing.price$Longitude), lat=mean(housing.price$Latitude), zoom = 13) %>%
  #OpenStreetMap, Stamen, Esri and OpenWeatherMap.
  addProviderTiles("Esri.WorldGrayCanvas") %>%
  addCircleMarkers(
            ~Longitude,
            ~Latitude,
            color = colors,
            radius = ~ sqrt(housing.price$Distance2MRT/10)*0.7,
            stroke = FALSE,
            fillOpacity = 0.4,
            label = ~label.msg)  %>%
  addLegend(position = "bottomright",
            colors =c("navy",  "orange", "darkred"),
            labels= c("<=5", "(5,15]"," >15+"),
            title= "House Age",
            opacity = 0.4) %>%
```

```
  addLegendSize(position = 'topright',
                  values = sqrt(housing.price$Distance2MRT/10)*0.7,
                   color = 'gray',
                fillColor = 'gray',
                 opacity = .5,
                   title = 'Distance to MRT',
                   shape = 'circle',
             orientation = 'horizontal',
                  breaks = 5)
```



11.3 Choropleth Maps

> A choropleth map brings together two datasets: spatial data representing a partition of geographic space into distinct districts, and statistical data representing a variable aggregated within each district. There are two common conceptual models of how these interact in a choropleth map: in one view, which may be called "district dominant," the districts (often existing governmental units) are the focus, in which a variety of attributes are collected, including the variable being mapped. In the other view, which may be called "variable dominant," the focus is on the variable as a geographic phenomenon. (Wikipedia)

Since constructing a choropleth map requires more coding effort to create a data set with a special structure. The following R libraries need to be used.

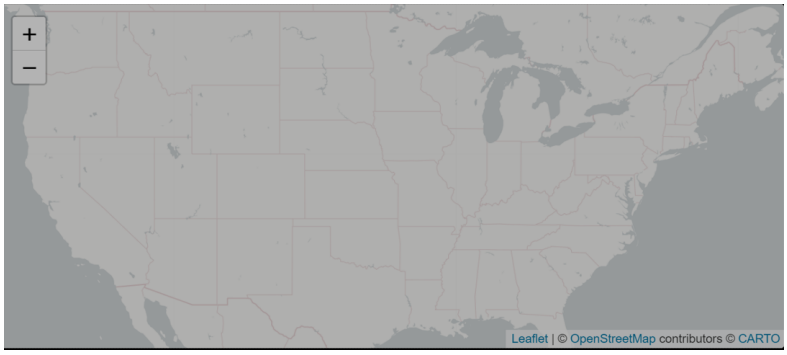
```
library(leaflet)
library(magrittr)
library(rgdal)
library(geojsonio)
library(htmltools)
library(htmlwidgets)
library(stringi)
library(RColorBrewer)
```

These libraries have been loaded in the set-up chunk. Next, we will start with a very simple data set to create choropleth maps.

11.3.1 Loading Data

The sample data contains the unit electricity price for all US states in 2018. We load this data and create a base map that is centered on the geographic center of the US.

```
state.electricity = read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/state_e
m <- leaflet() %>%
  addProviderTiles(providers$CartoDB.PositronNoLabels)  %>%
  # center of the US based on the geocode.
  setView(lng = -96.25, lat = 39.50, zoom = 4)
m
```

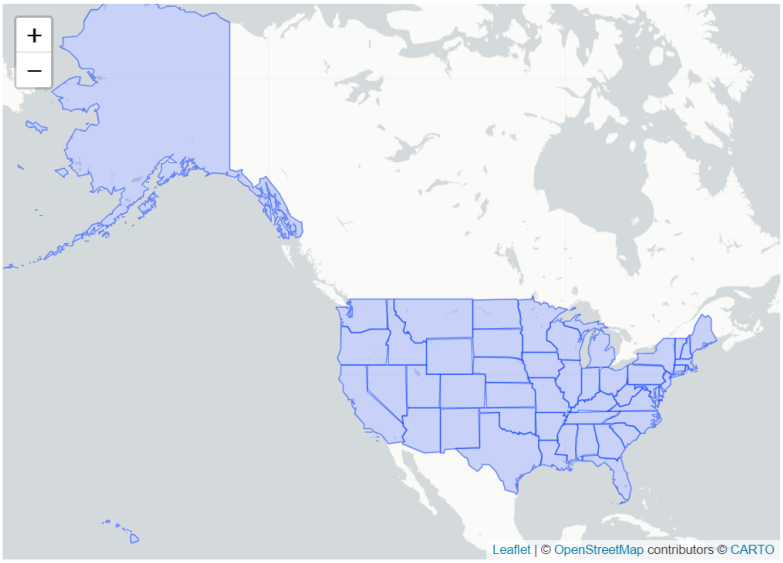


11.3.2 Load Map Shape Data

The shape data of the US map is available at https://github.com/pengdsci/sta553/raw/main/data/us_states.geojson. We load the shape data and use it to draw the clear border of states and fill states on the basemap with the same color.

```
library(sf)
#states <- st_read("C:/peng/eBooks/STA553/us_states.geojson")
states <- st_read("https://github.com/pengdsci/sta553/raw/main/data/us_states.geojson")
# Since  R package {rgdal} just retired, readOGR() cannot be used anymore to read .shp file!
# online geodata converter is also very useful: https://mygeodata.cloud/converter/
# states<- readOGR("C:/peng/eBooks/STA553/img07/cb_2019_us_state_5m.shp")
# states <- readOGR("https://github.com/pengdsci/sta553/raw/main/data/cb_2019_us_state_5m.shp")
m <- leaflet() %>%
  addProviderTiles(providers$CartoDB.PositronNoLabels)  %>%
  setView(lng = -96.25, lat = 39.50, zoom = 4) %>%
  addPolygons(data = states,
              weight = 1)  # weight represents the width of the state border.
```

```
m
```

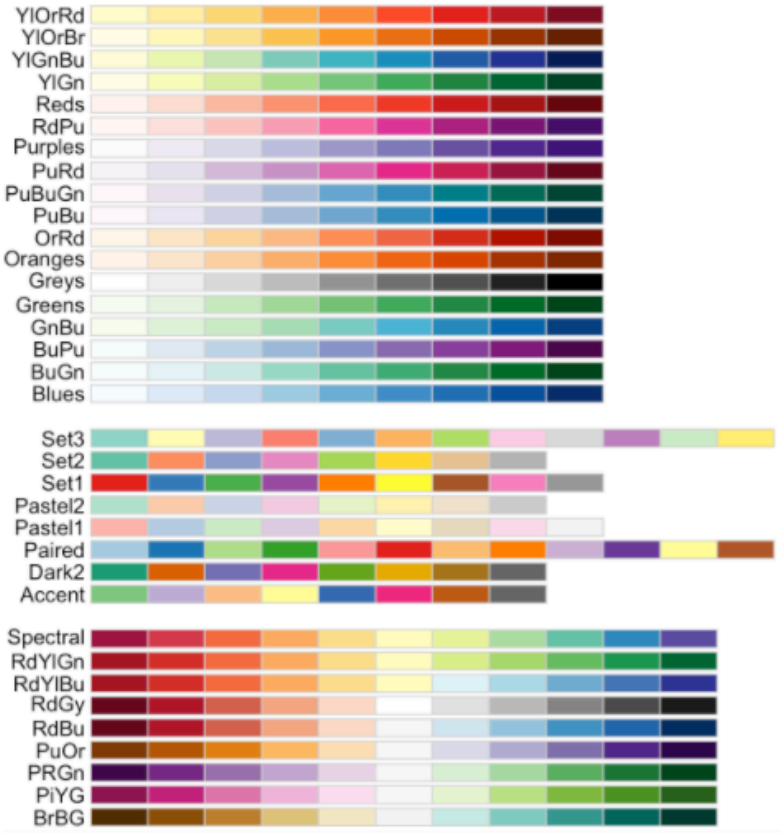


11.3.3 Merge the Map Data with the Price Data

Now we merge. By including `all.x = F` as an argument, we specify that elements of states that do not have a match in `dat` (e.g., Guam) should not be retained in the merged object. We can also go ahead and drop `Hawaii` and `Alaska` from the merged data frame because they won't be represented in the final choropleth.

We need to define some rules for coloration. To map colors to continuous values, we use `colorNumeric()`, specifying the color palette that values should be mapped to and the values. Here, we use the *"YlOrRd"* palette from `RColorBrewer`.

Other available color palettes can be found in the following chart.



Alternatively, we can map colors to bins of values instead of doing so continuously. In the electricity cost data, values range from ~7 cents to ~19 cents. We can break this range up into discrete colorable bins:

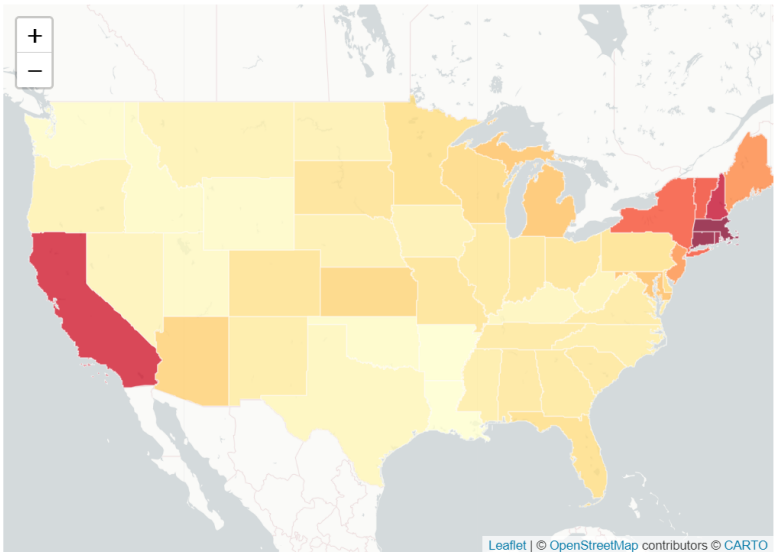
```
states <- merge(states, state.electricity , by = 'NAME', all.x = F)
states <- states[!(states$NAME == 'Hawaii' | states$NAME == 'Alaska'), ]
# Define continuous numeric color palette based on the price of unit electricity: YlOrRd
paletteNum <- colorNumeric('YlOrRd', domain = states$centskWh)
##
costBins <- c(7:19, Inf)     # define a sequence of integers to pick up the corresponding colors
paletteBinned <- colorBin('YlGnBu', domain = states$centskWh, bins = costBins)
```

11.3.4 Color States Based On Unit Price

`colorNumeric()` and `colorBin()` each generate a function to be used when creating a choropleth. States in this choropleth will be colored using the continuous function since the price is a continuous variable.

We now insert `paletteNum()` in the `addPolygons()` function below.

```
m <- leaflet() %>%
  addProviderTiles(providers$CartoDB.PositronNoLabels)  %>%
  setView(lng = -96.25, lat = 39.50, zoom = 4) %>%
  addPolygons(data = states,
              # state border stroke color
              color = 'white',
              # soften the weight of the state borders
              weight = 1,
              # values >1 simplify the polygons' lines for less detail
              # but faster loading
              smoothFactor = .3,
              # set opacity of polygons
              fillOpacity = .75,
              # specify that the each state should be colored per paletteNum()
              fillColor = ~paletteNum(states$centskWh))
m
```



11.3.5 5. Labeling Cost Information

We'll use `sprintf()` in combination with `lapply()` and `HTML()` (from library `{htmltools}`) to generate a formatted, HTML-tagged label for each state.

The `cbind()` statement stitches the labels onto the states objects. The labels can be included in the choropleth by adding a `label = argument` in `addPolygons()`, and `labelOptions =` provides additional customizability (label color, etc.).

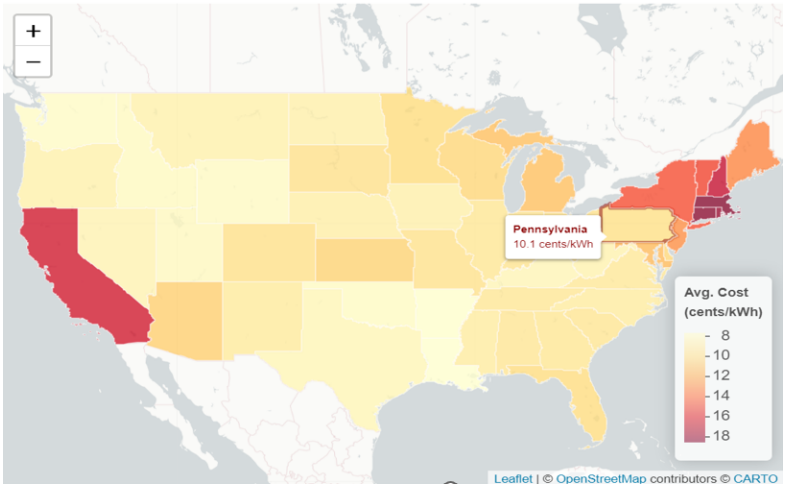
```
stateLabels <- sprintf('<b>%s</b><br/>%g cents/kWh',
                       states$NAME, states$centskWh) %>%
               lapply(function(x) HTML(x))
```

```
### combine the label with the states data
states <- cbind(states, matrix(stateLabels, ncol = 1, dimnames = list(c(), c('stateLabels'))))
```

11.3.6 6. Finalizing The Map

Finally, we'll take advantage of Leaflet's interactivity features by adding a `highlightOptions =` argument as part of `addPolygons()`. This allows us to define a response for sections of the map when a cursor passes over them. We'll keep it simple: While being moused over, states will be emphasized by a blue line.

```
m <- leaflet() %>%
  addProviderTiles(providers$CartoDB.PositronNoLabels) %>%
  setView(lng = -96.25, lat = 39.50, zoom = 3.5) %>%
  addPolygons(data = states,
              color = 'white',
              weight = 1,
              smoothFactor = .3,
              fillOpacity = .75,
              fillColor = ~paletteNum(states$centskWh),
              label = ~stateLabels,
              labelOptions = labelOptions(
                       style = list(color = 'darkred'), #list(color = 'gray30')
                      textsize = '10px'),
              highlightOptions = highlightOptions(
                        weight = 3,
                       color = 'darkred' )
              ) %>%
  addLegend(pal = paletteNum,
            values = states$centskWh,
            title = '<small>Avg. Cost<br>(cents/kWh)</small>',
            position = 'bottomright')
m
```



11.4 Plotly Map

`plotly` aims to be a general-purpose visualization library, and thus, doesn't aim to be the most fully-featured geospatial visualization toolkit.

`plotly` uses several different ways to create maps – each with its strengths and weaknesses. It utilizes plotly.js's built-in support to render the basemap layer. The types of basemap used in plotly are Mapbox (third party software that requires an access token) and D3.js powered basemap. In other words, plotly does not use OpenStreetMap that is used in `leaflet`, `Mapviewer`, `ggplot2`,`Shiny`, and Tableau.

We will not use `Mapbox` in this note and focus on the D3.js basemap that does not have many details. The plot function `plot_geo()` will be used to make quick maps.

Choropleth Maps

In the following, we will introduce the steps for creating Choropleth maps. Since Choropleth maps need to fill and color small regions such as district, county, states, etc., it requires the data set to have a special structure that contains shape information. Two plot constructor functions `plot_ly()` and `plot_geo()` will be introduced to create choropleth maps.

- `plot_ly()` requires specifying `type = choropleth` to make a map (basemap from plotly.js). Information in the data set is integrated into the maps by various arguments of `plot_ly()` and relevant graphic functions that are compatible with `plot_ly()`.
- `plot_geo()` requires `addTrace()` to make choropleth maps and integrate data information to the maps with relevant arguments in `addTrace()` and graphic functions compatible with `plot_geo()`.

Choropleth Maps with `plot_ly()`

In general, making choropleth maps with `plot_ly()` requires two main types of input:

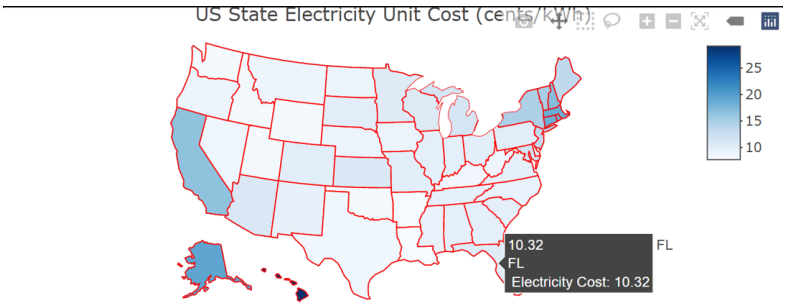
- Geometry information provided by
  - one of the built-in geometries within plot_ly such as US states and world countries. See the following example 1: Visualizing 2018 electricity cost per state.
  - a supplied GeoJSON file where each feature has either an id field or some identifying value in properties. See the following example 2: visualizing the unemployment rate of US counties
- A list of values indexed by feature identifier. They control the features in the map including boundary, filled colors, legend, hover text, etc.

Example 1: US electricity cost by States in 2018. The arguments `locations =` and `locationmode =` tell `plot_ly` what map information should be used to create the base map. Other arguments and functions are used to control different features of the resulting map. One cautionary note is that `plot_ly()` only uses state abbreviations as the state name.

In the code, the state abbreviation `state.abb` is a built-in data set. Several other built-in data sets about each state are also available. Check the website http://stats4stem.weebly.com/r-statex77-data.html for more information on these data sets.

```
# Map data preparation
electricitycost <-read.csv("https://github.com/pengdsci/sta553/raw/main/data/state_electricity_da
electricitycost$State <- state.abb  # add state abbrevs to specify locations in plot_ly()
# Create hover text
electricitycost$hover <- with(electricitycost, paste(State, '<br>', "Electricity Cost:", centskWh
# Make state borders white
borders <- list(color = toRGB("red"))
# Set up some mapping options
map_options <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  showlakes = TRUE,
  lakecolor = toRGB('white')
)
plot_ly( z = ~electricitycost$centskWh,
        text = ~electricitycost$hover,
        locations = ~electricitycost$State,
        type = 'choropleth',
        locationmode = 'USA-states',
        color = electricitycost$centskWh,
        colors = 'Blues',
        marker = list(line = borders)) %>%
```

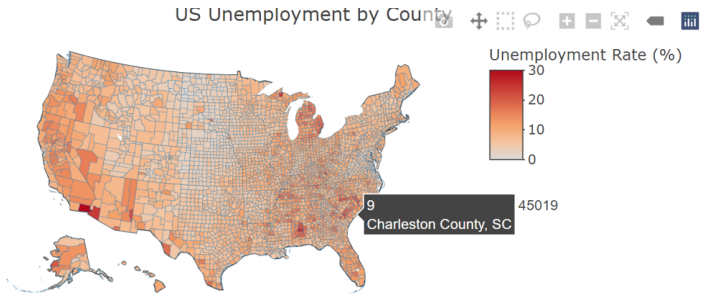
```
  layout(title = 'US State Electricity Unit Cost (cents/kWh)',
         geo = map_options)
```



Example 2: The unemployment rates of US counties. This example requires a JSON file to provide necessary geometric information (shape polygon) about each county in the US. The argument `locations =` accepts FIPS (Federal Information Process System) for US county maps. The geometric information of the US county shape is supplied in a JSON file and used through the argument `geojson =`.

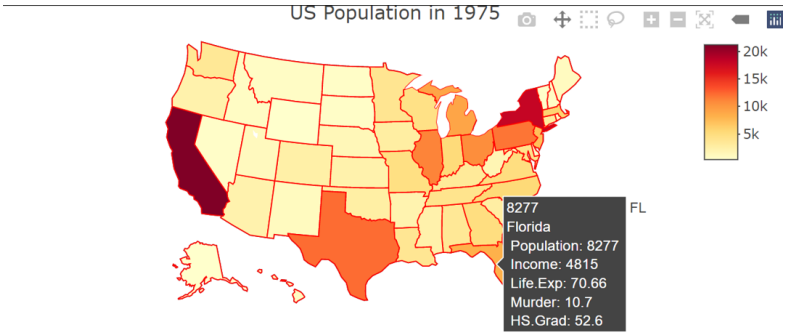
```
# It takes a few minutes to draw county-level boundaries.
#
url <- 'https://github.com/pengdsci/sta553/raw/main/data/geojson-counties-fips.json'  #
counties <- rjson::fromJSON(file=url)
load("img07/unemp.rda")
df=unemp
g <- list(
      scope = 'usa',
      projection = list(type = 'albers usa'),
      showlakes = TRUE,
      lakecolor = toRGB('white')
    )
###
fig <- plot_ly()  %>%
  add_trace( type = "choropleth",
          geojson = counties,
        locations = df$fips,
                z = df$rate,
       colorscale = "GnBu",
             zmin = 0,
             zmax = 30,
             text = df$name, # hover mesg
           marker = list(line=list(width=0.2))
          )   %>%
  colorbar(title = "Unemployment Rate (%)")  %>%
```

```
  layout( title = "US Unemployment by County",
          geo = g)
fig
```



Example 3 US states facts. Similar to example 1, but with more variables. The data set is built-in in the base R package.

```
# Create data frame
state_pop <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/USStatesFacts.
# Create hover text
state_pop$hover <- with(state_pop,
                        paste(STName, '<br>', "Population:", Population,
                              '<br>', "Income:", Income,
                              '<br>', "Life.Exp:", Life.Exp,
                              '<br>', "Murder:", Murder,
                              '<br>', "HS.Grad:", HS.Grad))
# Make state borders white
borders <- list(color = toRGB("red"))
# Set up some mapping options
map_options <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  showlakes = TRUE,
  lakecolor = toRGB('white')
)
plot_ly(z = ~state_pop$Population,
        text = ~state_pop$hover,
        locations = ~state_pop$State,
        type = 'choropleth',
        locationmode = 'USA-states',
        color = state_pop$Population,
        colors = 'YlOrRd',
        marker = list(line = borders)) %>%
  layout(title = 'US Population in 1975', geo = map_options)
```

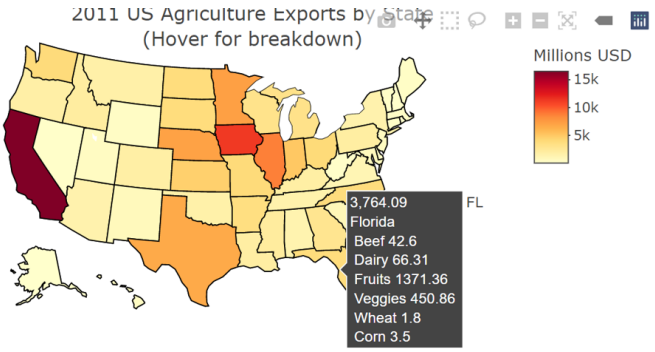


11.4.1 2. Choropleth Maps with `plot_geo()`

Making a choropleth map with `plot_geo` requires less effort to prepare the shape data. The geo-information was called through `locations =` and `locationmode =`.

```
# library(plotly)
# read in cv data
df <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/2011_us_ag_
## Define hover text
df$hover <- with(df, paste(state, "<br>",
                           "Beef", beef, "<br>",
                           "Dairy", dairy, "<br>",
                           "Fruits", total.fruits, "<br>",
                           "Veggies", total.veggies, "<br>",
                           "Wheat", wheat, "<br>",
                           "Corn", corn))
# give state boundaries a white border
l <- list(color = toRGB("white"), width = 2)
# specify some map projection/options
g <- list(     scope = 'usa',
          projection = list(type = 'albers usa'),
           showlakes = TRUE,
           lakecolor = toRGB('white')
      )
## plot map
m <- plot_geo(df, locationmode = 'USA-states') %>%
     add_trace(        z = ~total.exports,
                    text = ~hover,
               locations = ~code,
                   color = ~total.exports,
                  colors = 'YlOrRd'
              )  %>%
     colorbar(title = "Millions USD")  %>%
```

```
     layout( title = '2011 US Agriculture Exports by State<br>(Hover for breakdown)',
             geo = g
           )
m
```



Scatter Maps

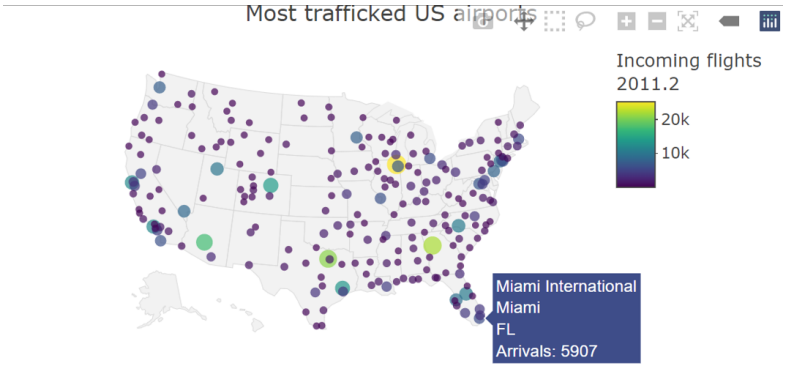
11.4.2 3.Scatter Map with `plot_geo()` and `add_markers()`

A Scatter map is relatively easier to make since we only plot the base map using the longitude and latitude. No map shape information is needed for scatter maps.

Example 4 US Airport Traffic.

```
#library(plotly)
df <- read.csv('https://raw.githubusercontent.com/pengdsci/sta553/main/data/2011_february_us_airp
# geo styling
g <- list(      scope = 'usa',
           projection = list(type = 'albers usa'),
             showland = TRUE,
            landcolor = toRGB("gray95"),
         subunitcolor = toRGB("gray85"),
         countrycolor = toRGB("gray85"),
         countrywidth = 0.5,
         subunitwidth = 0.5
       )
###
fig <- plot_geo(df, lat = ~lat, lon = ~long) %>%
  add_markers( text = ~paste(airport, city, state,
                             paste("Arrivals:", cnt),
                             sep = "<br>"),
               color = ~cnt,
               symbol = "circle",
               size = ~cnt,
               hoverinfo = "text")   %>%
```

```
  colorbar(title = "Incoming flights<br>2011.2")  %>%
  layout( title = 'Most trafficked US airports',
          geo = g )

fig
```

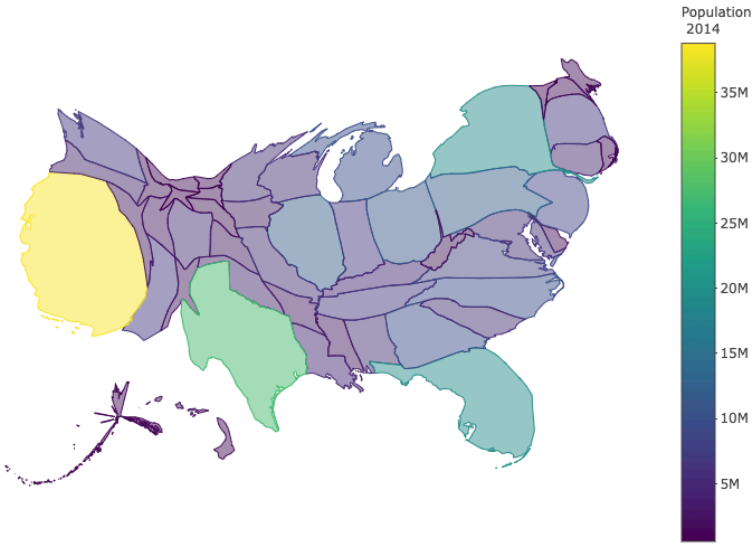


Custom Maps*

11.4.3 4. Custom Map With Special Libraries

Sometimes, we may want to use custom maps to represent spatial information. For example, if we want to visualize the area of US states, the previous US maps are fine. If we want to represent the population size (Example 3), we may want to use a map such that the displayed area is proportional to the population size but not the geographical area. These types of custom maps need special tools to construct. Show you an exam without providing code to make the map.

Example 4 US population by states.



11.5 Mapview Maps

> *Every* `mapview` *map loads several background maps and has a lot of advanced rendering capabilities that can be used to view large data. Because of this reason, each interactive* `mapview` *map is a huge file in size. Since this eBook is hosted on GitHub which has a limitation on the size of the individual file (25MB), we will NOT include interactive* `mapview` *maps in this section. Instead, we will include some screenshots to show what* `mapview` *maps look like. You can run the code on a local machine to view the interactivity of these maps.*

`mapview` provides functions to very quickly and conveniently create interactive visualizations of spatial data. Its main goal is to fill the gap of quick (not presentation grade) interactive plotting to examine and visually investigate both aspects of spatial data, the geometries, and their attributes.

It can also be considered a data-driven application programming interface (API) for the leaflet package as it will automatically render correct map types, depending on the type of the data (points, lines, polygons, raster).

In addition, it makes use of some advanced rendering functionality that will enable the viewing of much larger data than is possible with leaflet.

`mapview()` - view (multiple) spatial objects on a set of background maps

`viewExtent()` - view extent / bounding box of spatial objects

`viewRGB()` - view RGB true- or false-color images of raster objects

`mapshot()` - easily save maps (including leaflet maps) as HTML and/or png (or other image formats).

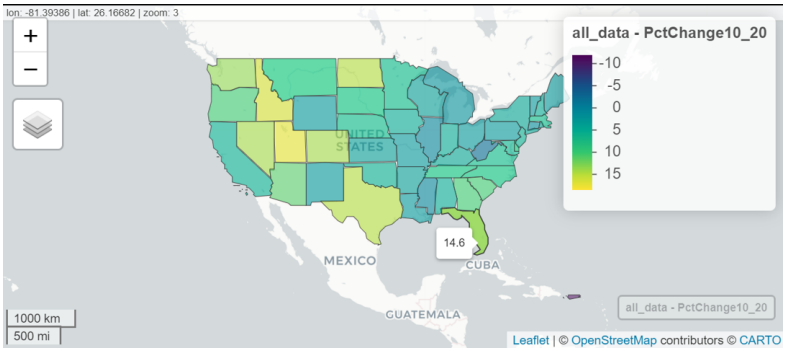
`mapview()` function can work for a quick view of the data, providing choropleths, background maps, and attribute popups. Performance varies on the object and customization can be tricky.

In this note, we will use three data sets to illustrate how to make maps with `mapview()`: state population, state population (50 state), and state capitals.

11.5.1 1. Choropleth Map with `mapview`

```
library(tigris)
library(mapview)
library(dplyr)
library(sf)
#invisible({capture.output({
## Download shapefile for all states into R in tigris
## states() is a function in library {tigris}
us_geo <- states(cb = F, resolution = '20m')
#})
#})
```

```
## caution: need to tell R that GEOIS should be a character variable since
## the same GEOID is character variable in the shape file us_geo with
## some leading zeros!
pop_data <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/state
## merger two data use the primary key: GEOID.
all_data <- inner_join(us_geo, pop_data, by = c("GEOID" = "GEOID"))
##
mapview(all_data, zcol = "PctChange10_20")
```



We can see some of the unique features of `mapview`:

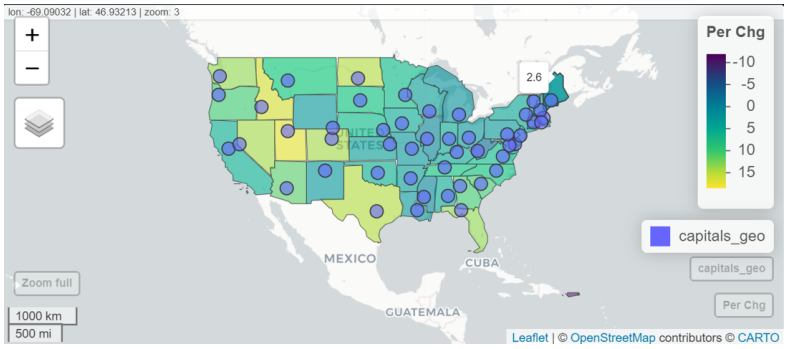
1. The default `mapview()` is created with one small line of code.
2. `mapview()` Choose defaults based on the type of geospatial file. code `mapview(all_data, zcol = "PctChange10_20")`
3. The default popup includes every field in my data that is useful to explore the data.

There are unwanted features in the above default choropleth map that need to be improved. For example, too much information in the hover text, lengthy legend title, etc. We will revisit this map and enhance the choropleth map.

11.5.2 2. Scatter Map with `mapview`

We now use `state capital` with longitude and latitude and plot the capital of each state. The idea is similar to that of `ggplot`: we make the scatter plots on the existing choropleth map using the geocode in the new `sf` object defined based on the new data set.

```
library(tigris)
library(mapview)
library(sf)
## load the location data
capitals <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/us-state-capita
##
capitals_geo <- st_as_sf(capitals,
                         coords = c("longitude", "latitude"),
                         crs = 4326)
## we add the above layer to the previously created map
mapview(all_data, zcol = "PctChange10_20", layer.name = "Per Chg") + capitals_geo
```



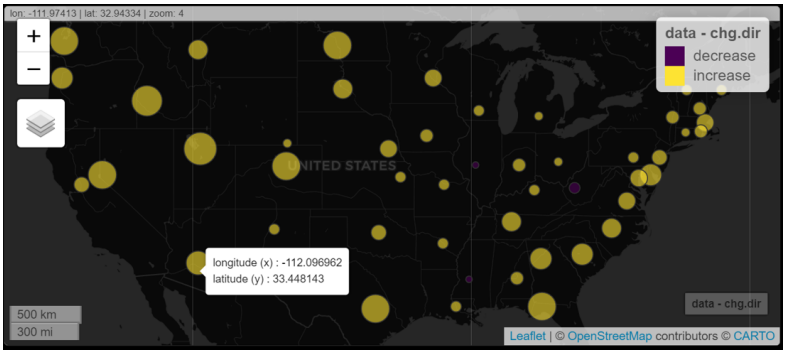
11.5.3 3. Scatter Map Without Using `sf` Object

We can also use `mapview()` to make a scatter plot directly on the basemap without using a choropleth map as the background. This simply uses the data

with longitude and latitude to make the scatter map without converting the data to an 'sf' object.

The set to be used is a regular R data frame with longitude and latitude. Two new variables were added to the merged data set to make more information geographic representation of the change to the population from 2010 to 2020.

```
library(mapview)
pop_data <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/state
capitals <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/us-st
## inner join the above two data frames
state.pop.geo <- inner_join(pop_data, capitals, by = "State")
state.pop.geo$chg.size <- abs(state.pop.geo$PctChange10_20)
chg.dir <- rep("increase", dim(state.pop.geo)[1])
chg.dir[which(state.pop.geo$PctChange10_20 < 0)] = "decrease"
state.pop.geo$chg.dir = chg.dir
#
mapview(state.pop.geo, xcol = "longitude", ycol = "latitude",
        crs = 4269,
        grid = FALSE,
        na.col ="red",
        zcol = "chg.dir",
        cex = "chg.size",
        popup = TRUE,
        legend = TRUE)
```



11.5.4 4. Customizing Default Maps

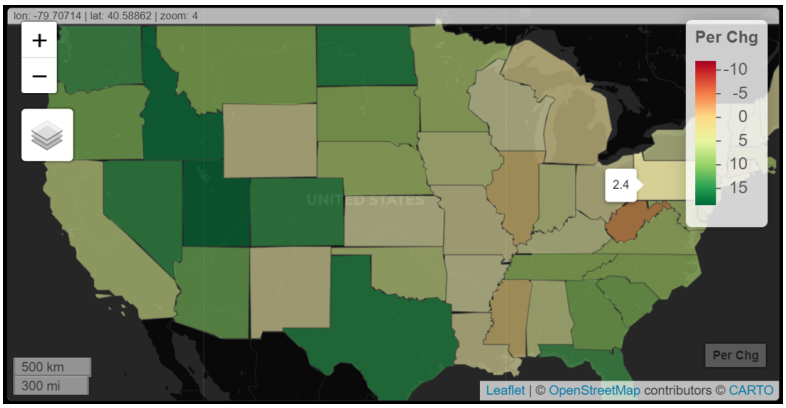
The following custom map enhances the default map from several perspectives.

- We can customize map options such as color for polygon boundary lines and `col.regions` to fill the polygon Colors and `alpha.regions` for transparency.
- We can rename a layer with the extension `layer.name` if we want a more user-friendly layer name which will appear in the legend, the bottom right

button, and when you open the layer button toward the top left.

- The default popup table contains the complete record. In practice, we only want to display a partial list of variables.
- We set the center for viewing the map and the level of zoom.

```
library(tigris)
library(mapview)
library(leafpop)
library(sf)
CustomMap <- mapview(all_data,
        ## popup option allows to select variables to be displayed in the popup.
        ## by default, it shows all variables in the data set.
        popup = popupTable(all_data,
                           zcol = c("State",
                                    "Pop2010",
                                    "Pop2020",
                                    "PctChange10_20")),
        zcol = "PctChange10_20",
        layer.name = "Per Chg",
        col.regions = brewer.pal(11,"RdYlGn"),
        alpha.regions = 0.6)
##
CustomMap@map %>% setView(-98.35, 39.50, zoom = 4)
```

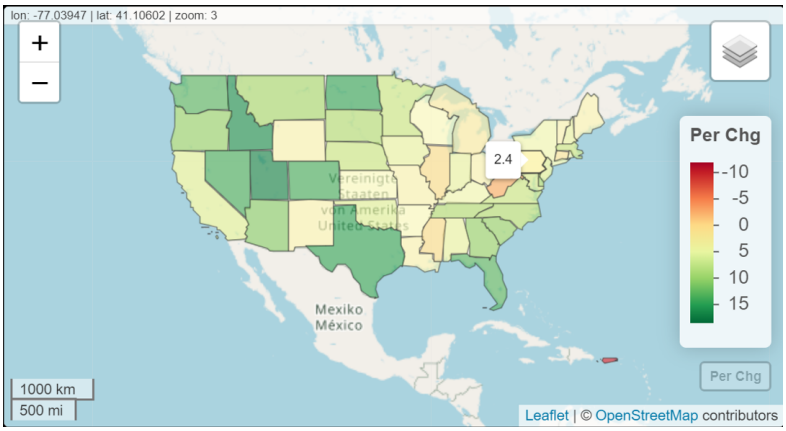


11.5.5 5. Mapview Options

It is not straightforward to add features to the map directly via `mapview()`. However, `MapviewOption()` can be used to add some new features or modify some existing features on the map. The following example shows some modifications.

```
library(mapview)
```

```
library(leafpop)
library(sf)
mapviewOptions(basemaps = c("Esri.WorldShadedRelief",
                            "OpenStreetMap.DE",
                            "CartoDB.Positron",
                            "CartoDB.DarkMatter",
                            "OpenTopoMap"),
               raster.palette = grey.colors,
               na.color = "magenta",
               legend.pos = "bottomright",
               layers.control.pos = "topright")
##
mapview(all_data,
        zcol = "PctChange10_20",
        layer.name = "Per Chg",
        col.regions = brewer.pal(11, "RdYlGn"),
        alpha.regions = 0.6)
```



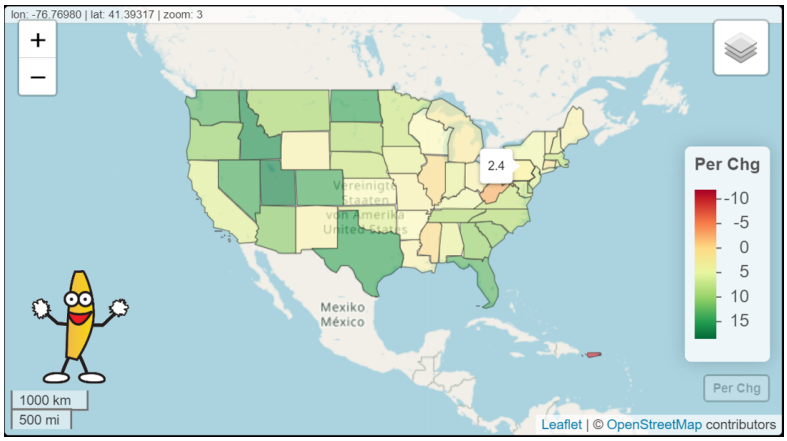
11.5.6 6. More Enhancements of Mapview (Leaflet)

We can enhance mapview by adding more features to the existing mapview. In the following, we will add a few new features to the one created earlier. Add an image (WCU logo and gif) to the map.

Example 1: Adding a gif image to the map.

```
library(mapview)
library(leafpop)
library(sf)
library(leafem)
CustomMap <- mapview(all_data,
        popup = popupTable(all_data,
                           zcol = c("State",
```

```
                                    "Pop2010",
                                    "Pop2020",
                                    "PctChange10_20")),
        zcol = "PctChange10_20",
        layer.name = "Per Chg",
        col.regions = brewer.pal(11,"RdYlGn"),
        alpha.regions = 0.6)
## adding .gif image to the map.
leafem::addLogo(CustomMap, "https://github.com/pengdsci/sta553/raw/main/image/banana.gif",
                position = "bottomleft",
                offset.x = 5,
                offset.y = 40,
                width = 100,
                height = 100)
```

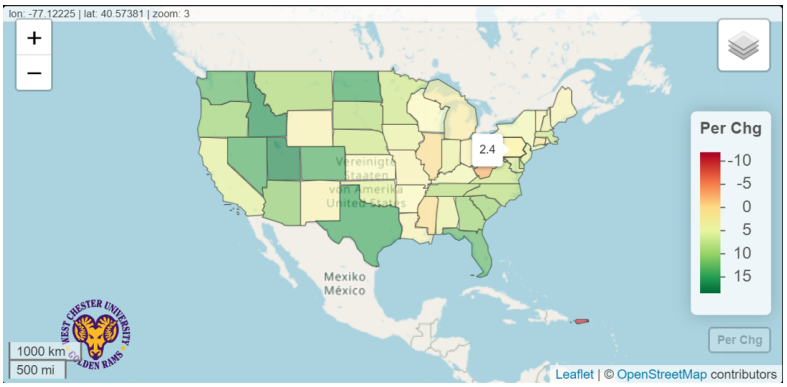


Example 2: Adding an image to the map.

We add the WCU logo to the map.

```
library(mapview)
library(leafpop)
library(sf)
library(leafem)
##
img <- "https://github.com/pengdsci/sta553/raw/main/image/goldenRamLogo.png"
##
CustomMap <- mapview(all_data,
        popup = popupTable(all_data,
                           zcol = c("State",
                                    "Pop2010",
                                    "Pop2020",
                                    "PctChange10_20")),
        zcol = "PctChange10_20",
```

```
        layer.name = "Per Chg",
        col.regions = brewer.pal(11,"RdYlGn"),
        alpha.regions = 0.6)
      ###
CustomMap %>% leafem::addLogo(img, url = "https://github.com/pengdsci/sta553/raw/main/
###
```



11.6 Thematic Maps

The `tmap` package is a relatively new way to plot thematic maps in R. Thematic maps are geographical maps in which spatial data distributions are visualized. This package offers a flexible and layer-based approach to creating thematic maps, such as choropleths and bubble maps. The syntax for creating plots is similar to that of ggplot2.

`tmap_mode()` will be used to determine the interactivity of the map: `tmap_mode("plot")` produces static maps and `tmap_mode("view")` produces interactive maps.

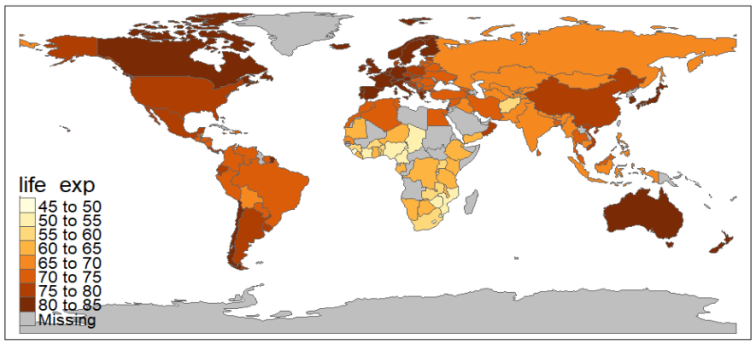
11.6.1 1. Choropleth Maps

We will use a built-in world shapefile, `World`, that contains information about population, gdp, life expectancy, income, happiness index, etc.

Both choropleths and scatter maps will be illustrated using the built-in data.

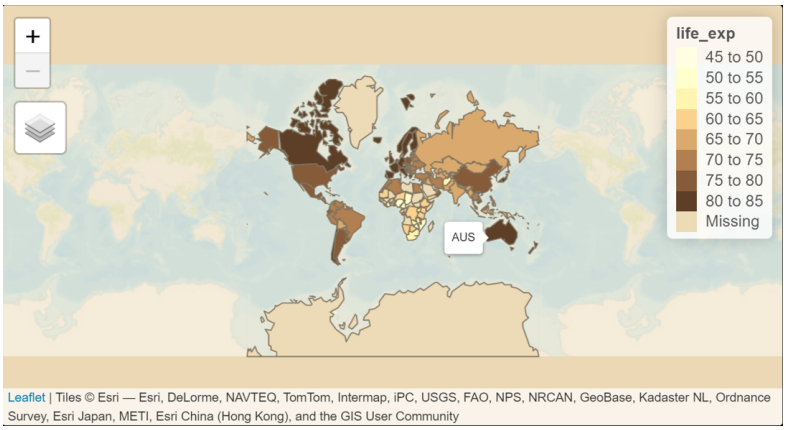
Example 1: Choropleths: The default world map with the distribution of mean life expectancy among all countries. The default `tmap_mode` is set to be `plot`. The default `tmap` map is static.

```
library(tmap)
data(World)
tm_shape(World) +
    tm_polygons("life_exp")
```



Example 2: Interactive Map: use the above static map as the base map and add interactive features to the maps. The mode can be set with the function `tmap_mode()`, and toggling between the modes can be done with the 'switch' `ttm()` (which stands for toggle thematic map.

```
library(tmap)
#
tmap_mode("view")  # "view" gives interactive map; "plot" gives static map.
##
## tmap_style set to "classic"
tmap_style("classic")
## other available styles are: "white", "gray", "natural",
## "cobalt", "col_blind", "albatross", "beaver", "bw", "watercolor"
tmap_options(bg.color = "skyblue",
             legend.text.color = "white")
##
tm_shape(World) +
      tm_polygons("life_exp",
                  legend.title = "Life Expectancy") +
      tm_layout(bg.color = "gray",
                inner.margins = c(0, .02, .02, .02))
```

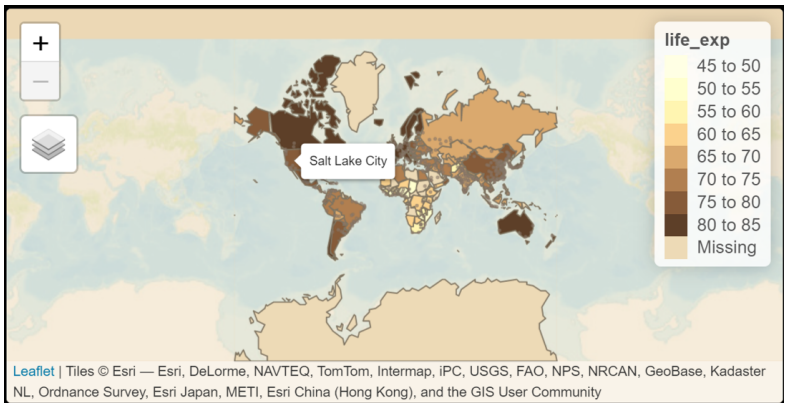


11.6.2 Mixed Maps (Multilayer)

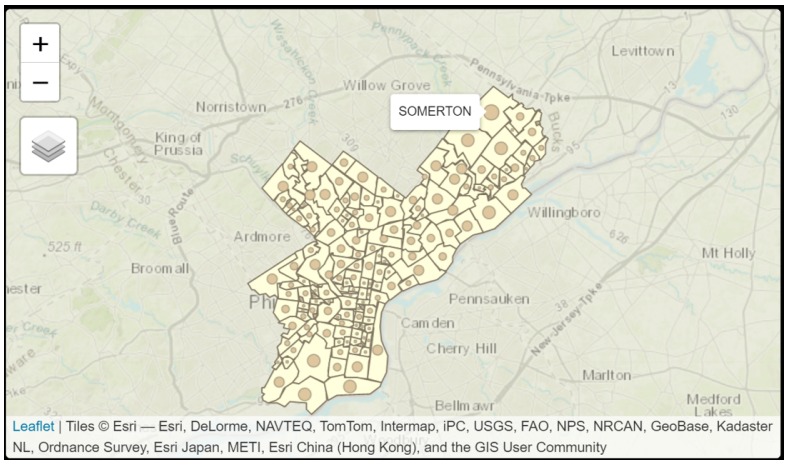
Through a multilayer map, we can make a choropleth and place another on top of it. In the following example, we add one additional layer using the `metro` shapefile and plot the center of the metro area to get a scatter map.

Example 3: Mixed Map: two-layer mixes maps

```
library(tmap)
#*
data(metro)
##
tmap_mode("view")  # "view" gives interactive map;
#tmap_style("classic") ## tmap_style set to "classic"
## other available styles are: "white", "gray", "natural",
## "cobalt", "col_blind", "albatross", "beaver", "bw", "watercolor"
tmap_options(bg.color = "skyblue",
             legend.text.color = "white")
##
tm_shape(World) +
      tm_polygons("life_exp",
                  legend.title = "Life Expectancy") +
      tm_layout(bg.color = "gray",
                inner.margins = c(0, .02, .02, .02)) +
tm_shape(metro) +
      tm_symbols(col = "purple",
                 size = "pop2020",
                 scale = .5,
                 alpha = 0.5,
                 popup.vars=c("pop1950", "pop1960", "pop1980","pop1990",
```



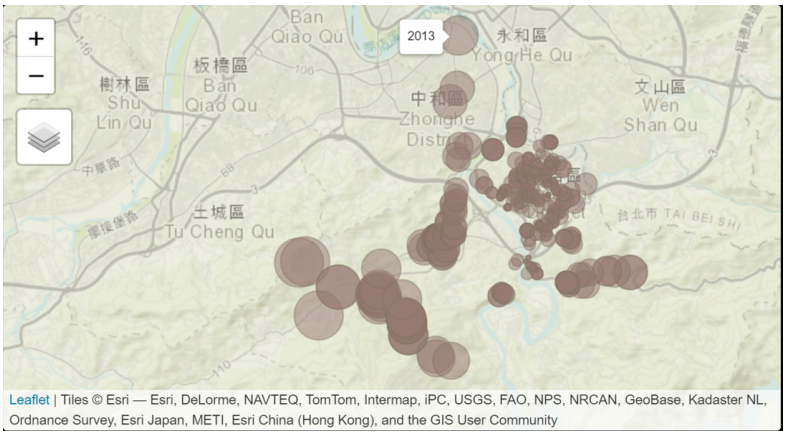

```
library(spData)
library(sf)
library(mapview)
gj = "https://github.com/azavea/geo-data/raw/master/Neighborhoods_Philadelphia/Neighborhoods_Phil
##
gjsf = st_read(gj)
library(tmap)
#    tm_shape(World) +
    tm_shape(gjsf)  +
      tm_polygons(legend.show = FALSE)  +
      tm_bubbles("shape_area",
                 #col = "shape_area",
                 #breaks=seq(1276674, 129254597, length = 6),
                 palette="-RdYlBu",
                 contrast=1)
```



11.6.3 Scatter Maps With geoCoordinates

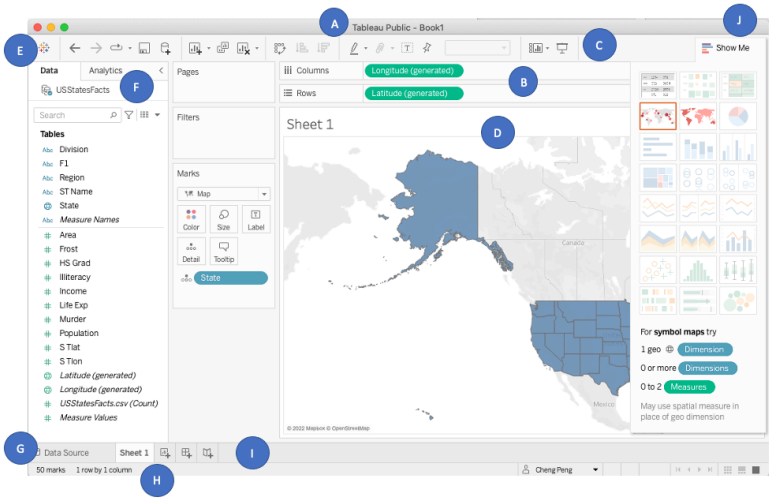
We use the real estate data set to make a scatter map using the library `tmap`. We first need to define an `sf` object using `st_as_sf()` that shapefile with an individual point based on the longitude and latitude in the data set.

```
library(tmap)
library(sf)
realestate0 <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/map/Rea
realest <- realestate0[, -1]
## Create a shapefile with POINT type.
realest <-st_as_sf(realest, coords=c("Longitude","Latitude"), crs = 4326)
###
tm_shape(realest) +
  tm_dots(col = "purple",
          size = "Distance2MRT",
          alpha = 0.5,
          popup.vars=c("HouseAge", "PriceUnitArea", "NumConvenStores"),
          shapes = c(1, 0))
```



11.7 Tableau Maps

We create a choropleth map and a scatter map respectively in this note. Before creating maps, we first look to understand the structure of Tableau's sheet.



We can see that each Tableau book has several components:

- **A - Workbook name** - A workbook contains sheets. A sheet can be a worksheet, a dashboard, or a story.
- **B - Cards and shelves** - Drag fields to the cards and shelves in the work space to add data to your view.
- **C - Toolbar** - Use the toolbar to access commands and analysis and navigation tools.
- **D - View** - This is the canvas in the work space where you create a visualization (also referred to as a "viz").
- **E - Start page icon** - Click this icon to go to the Start page, where you can connect to data. For more information, see Start Page.
- **F - Side Bar** - In a worksheet, the sidebar area contains two tabs: the Data pane and the Analytics pane.
- **G - Data Source** - Click this tab to go to the Data Source page and view your data.
- **H - Status bar** - Displays information about the current view.
- **I - Sheet tabs** - Tabs represent each sheet in your workbook. This can include worksheets, dashboards, and stories. You can rename and add more of these sheets, dashboards, and stories if needed.
- **Show Me** - Click this toggle to select 24 built-in charts and the information needed to create these charts.

11.7.1 Choropleth Map

The data set we use for a choropleth map can be downloaded from https://raw.githubusercontent.com/pengdsci/sta553/main/data/USStatesFacts.csv.

You need to download and save this data file in a folder and then connect it to Tableau Public (or Tableau Online).

The following are steps for making a choropleth map:

1. Load the .csv file to Tableau (Public);
2. Click `sheet1` in the bottom left taskbar;
3. Drag variable `State` (on the left navigation panel under the table) to the main drop field (Tableau considers `State` as a geo-variable); at the same time, the two generated `Longitude(generated)` and `Latitude(generated)` appear in the column and row fields automatically.
4. Click the `Show Me` (on the right side of a tiny color bar chart) in the top right of the screen;
5. You will see a list of graphs. Click the middle `world map` in the second row, you will see an initial choropleth map.
6. Click `Show Me` again to close the popup. We can click the legend on the top-right color to change the color of the map (if you like).
7. To add more information to the hover text, you drag the variables on the list to the small icon labeled with `Detail`.
8. Click `Sheet 1` to change it to a meaningful title.
9. Finally we label the states by their abbreviations. To do this, drag `State` to `Label` in the `Marks` table (next to `Detail`).
10. You can edit the hover text by clicking `Tooltip`.

The resulting map can be viewed on the Tableau Public Server at https://public.tableau.com/app/profile/cpeng/viz/US-States-Facts/Sheet1

11.7.2 Scatter Map

We use housing price data with the longitude and latitude associated with each property. The data set is at https://raw.githubusercontent.com/pengdsci/sta553/main/map/Realestate.csv

As we did in the previous example, we downloaded the data set and saved it in a folder.

The following are steps to create a scatter map.

1. Open the Tableau and connect the data source to Tableau.

2. After the data has been loaded to Tableau, click `Sheet1`, you will see the list of variables on the left panel.
3. Click `Latitude` -> `Geographic Role` -> `Latitude`; do the same thing to `Longitude`.
4. Drag `Latitude` to the `Columns` field and `Longitude` to the `Rows` field. You will see a single point in `Sheet 1`. The two variables were automatically renamed as `AVG(Latitude)` and `AVG(Longitude)`.
5. Click `AVG(Latitude)` and select `dimension`, you will see a line plot in `Sheet 1`. Do the same thing to `AVG(Longitude)`. Now you see a scatter plot.
6. Click `Show Me` (top-right corner of `Book 1`) and select the left-hand side map icon (the first one in the second row), you will see an initial scatter map.
7. We want to use the size of the point to reflect the unit price. We drag `PriceUnitArea` to `Size` card in the `Marks` shelf.
8. Click `Show Me` to close the chart menu. Click `SUM(Price Unit Area)` (top-right corner) to change the point size.
9. I drag `Transaction Year` to the `Color` card to reflect the transaction year. We should choose a divergent color scale.
10. Drag variables to the `Detail` card to be shown in the hover text.
11. Since many unit prices are close to each other, there are overlapped points. So we want to change the level of opacity. To do this, click `Color` card, choose the appropriate level of opacity, and edit the color to make a better map.
12. Add a meaningful title.
13. Right click the map and select `Map Layers` ake the changes on the map background and layers.
14. Other edits and modifications to improve the map.

The resulting map can be viewed on the Tableau Public Server at https://public.tableau.com/app/profile/cpeng/viz/RealEstateData_16469067466610/Sheet1

11.8 R Color Palettes

Since color coding is particularly important in map representation. We can use the following code to view various defined color scales (continuous and discrete) in the R library `RColorBrewer`. * **Sequential palettes** are suited to ordered data that progress from low to high (gradient). The palettes names are: `Blues`,

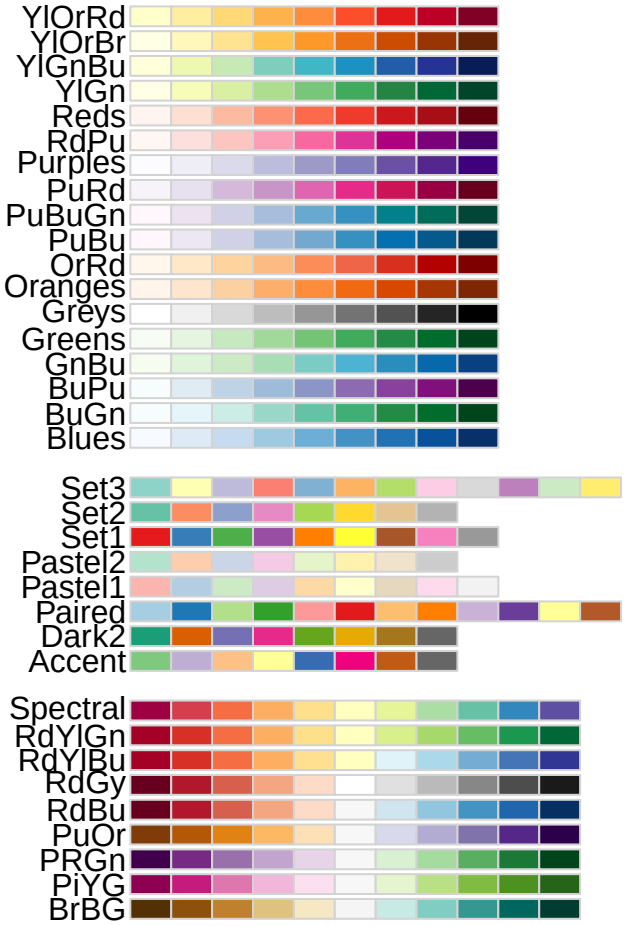
`BuGn`, `BuPu`, `GnBu`, `Greens`, `Greys`, `Oranges`, `OrRd`, `PuBu`, `PuBuGn`, `PuRd`, `Purples`, `RdPu`, `Reds`, `YlGn`, `YlGnBu`, `YlOrBr`, `YlOrRd`.

- **Qualitative palettes** are best suited to represent nominal or categorical data. They do not imply magnitude differences between groups. The palette names are: `Accent`, `Dark2`, `Paired`, `Pastel1`, `Pastel2`, `Set1`, `Set2`, `Set3`.

- **Diverging palettes** put equal emphasis on mid-range critical values and extremes at both ends of the data range. The diverging palettes are: `BrBG`, `PiYG`, `PRGn`, `PuOr`, `RdBu`, `RdGy`, `RdYlBu`, `RdYlGn`, `Spectral`.

```
library("RColorBrewer")
display.brewer.all()
```

11.8.1 All Palettes


YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues
Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent
Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG

11.8.2 2. Color-bind Friendly Palettes

```
library("RColorBrewer")
display.brewer.all(colorblindFriendly = TRUE)
```

11.8.3 3. Color Palette Codes

```
library("RColorBrewer")
kable(brewer.pal.info)
```

	maxcolors	category	colorblind
BrBG	11	div	TRUE
PiYG	11	div	TRUE
PRGn	11	div	TRUE
PuOr	11	div	TRUE
RdBu	11	div	TRUE
RdGy	11	div	FALSE
RdYlBu	11	div	TRUE
RdYlGn	11	div	FALSE
Spectral	11	div	FALSE
Accent	8	qual	FALSE
Dark2	8	qual	TRUE
Paired	12	qual	TRUE
Pastel1	9	qual	FALSE
Pastel2	8	qual	FALSE
Set1	9	qual	FALSE
Set2	8	qual	TRUE
Set3	12	qual	FALSE
Blues	9	seq	TRUE
BuGn	9	seq	TRUE
BuPu	9	seq	TRUE
GnBu	9	seq	TRUE
Greens	9	seq	TRUE
Greys	9	seq	TRUE
Oranges	9	seq	TRUE
OrRd	9	seq	TRUE
PuBu	9	seq	TRUE
PuBuGn	9	seq	TRUE
PuRd	9	seq	TRUE
Purples	9	seq	TRUE
RdPu	9	seq	TRUE
Reds	9	seq	TRUE
YlGn	9	seq	TRUE
YlGnBu	9	seq	TRUE
YlOrBr	9	seq	TRUE
YlOrRd	9	seq	TRUE

11.8.4 4. Functions for Selecting Specific Color Palettes

Two functions can be used to display a specific color palette or return the code of the palette.

- `display.brewer.pal(n, name)` displays a single `RColorBrewer` palette by specifying its name.
- `brewer.pal(n, name)` returns the hexadecimal color code of the palette.

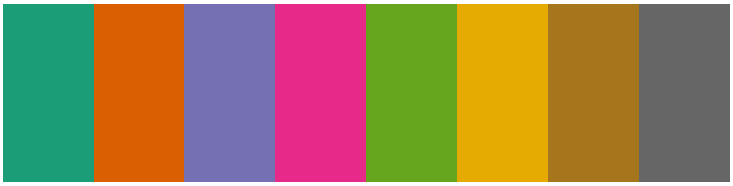
The two arguments:

`n` = Number of different colors in the palette, minimum 3, maximum depending on palette.

`name`= A palette name from the lists above. For example name = RdBu.

Example 1: Display the first 8 colors of palette `Dark2`.

```
# View a single RColorBrewer palette by specifying its name
display.brewer.pal(n = 8, name = 'Dark2')
```



Dark2 (qualitative)

Example 2: Return the hexadecimal of the first 8 colors of palette `Dark2`.

```
# Hexadecimal color specification
kable(t(brewer.pal(n = 8, name = "Dark2")))
```

#1B9E77	#D95F02	#7570B3	#E7298A	#66A61E	#E6AB02	#A6761D	#666666

A. Functions Calling Specific `rcolorbrewer` Palette in `ggplot()`

The following color scale functions are available in ggplot2 for using the `rcolorbrewer` palettes:

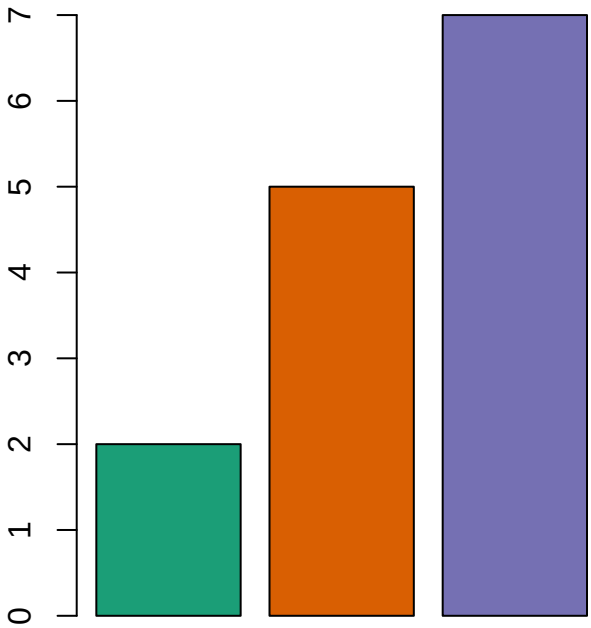
`scale_fill_brewer()` for box plot, bar plot, violin plot, dot plot, etc.

`scale_color_brewer()` for lines and points

B. Functions Calling Specific `rcolorbrewer` Palette in Base Plots

The function `brewer.pal()` is used to generate a vector of colors.

```r
# Barplot using RColorBrewer
barplot(c(2,5,7), col = brewer.pal(n = 3, name = "Dark2"))
```



Chapter 12

A `mapview` Map Demo

Since the each individual file size cannot be bigger that 25 MB, we only show a simple 'mapview' map so we can we can interact with this type of maps.

`mapview` provides functions to very quickly and conveniently create interactive visualizations of spatial data. Its main goal is to fill the gap of quick (not presentation grade) interactive plotting to examine and visually investigate both aspects of spatial data, the geometries, and their attributes.

It can also be considered a data-driven application programming interface (API) for the leaflet package as it will automatically render correct map types, depending on the type of the data (points, lines, polygons, raster).

In addition, it makes use of some advanced rendering functionality that will enable the viewing of much larger data than is possible with leaflet.

`mapview()` - view (multiple) spatial objects on a set of background maps

`viewExtent()` - view extent / bounding box of spatial objects

`viewRGB()` - view RGB true- or false-color images of raster objects

`mapshot()` - easily save maps (including leaflet maps) as HTML and/or png (or other image formats).

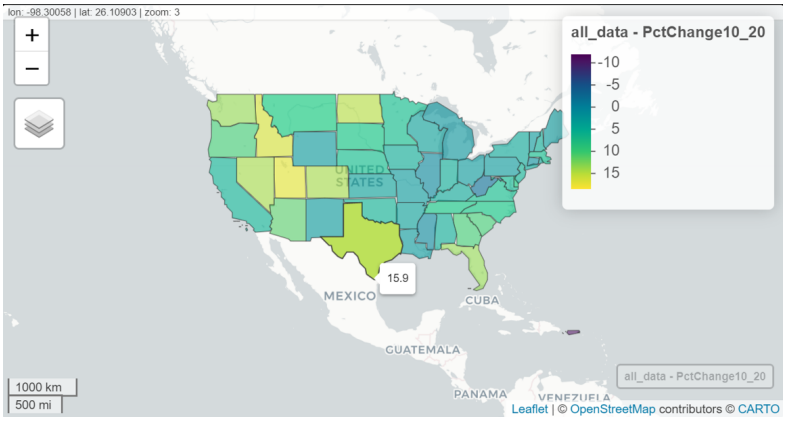
`mapview()` function can work for a quick view of the data, providing choropleths, background maps, and attribute popups. Performance varies on the object and customization can be tricky.

In this note, we will use three data sets to illustrate how to make maps with `mapview()`: state population, state population (50 state), and state capitals.

Choropleth Map with `mapview` - Demo

```
library(tigris)
library(mapview)
library(dplyr)
library(sf)
#invisible({capture.output({
## Download shapefile for all states into R in tigris
## states() is a function in library {tigris}
us_geo <- states(cb = F, resolution = '20m')
#})
#})
```

```
## caution: need to tell R that GEOIS should be a character variable since
## the same GEOID is character variable in the shape file us_geo with
## some leading zeros!
pop_data <- read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/data/state
## merger two data use the primary key: GEOID.
all_data <- inner_join(us_geo, pop_data, by = c("GEOID" = "GEOID"))
##
mapview(all_data, zcol = "PctChange10_20")
```



Chapter 13

Creating Maps Using Shapefiles

In this note, we use examples to illustrate how to create shapefiles and export built-in shape files from R.

13.1 Existing Base Map File and New Information

We use the built-in world map file, `World`, from the package `sf`(simple features), a standardized way to encode spatial vector data. The `World` data set has 17 variables including country names and the corresponding abbreviations, population densities, geometry (information of geo-polygon with longitude and latitude), etc. The data set has 177 countries. We will use the country names to define a primary key to merge the new data (with updated information) with the existing data.

The new information to be included in the new data set is based on several data sets available at https://projectdat.s3.amazonaws.com/datasets.html#week12

The information we will add to the `World` data set is:

1. Income from years 2000, 2005, 2010, 2015 and name them as inc00, inc05, inc10, inc15.
2. Life expectancy from years 2000, 2005, 2010, 2015 and name them as life00, life05, life10, life15.
3. Primary defined from the country names and their abbreviations.

```
#data(World)
World0 <- st_read(system.file("shapes/world.gpkg", package="spData"))
```

```
## Reading layer `world' from data source `C:\Users\75CPENG\AppData\Local\R\win-library
## Simple feature collection with 177 features and 10 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -89.9 xmax: 180 ymax: 83.64513
## Geodetic CRS:  WGS 84
```

```
inc <- read.csv("https://pengdsci.github.io/datasets/income_per_person.csv")
income <- data.frame(country = gsub(" ", "", inc$geo), inc00 = inc$X2000, inc05 = inc$
###
lifexp <- read.csv("https://pengdsci.github.io/datasets/life_expectancy_years.csv")
life.exp <- data.frame(country =  gsub(" ", "", lifexp$geo), lif00 = lifexp$X2000, lif
###
pop <- read.csv("https://pengdsci.github.io/datasets/population_total.csv")
popsize <- data.frame(country = gsub(" ", "", pop$geo), pop00 =pop$X2000, pop05 = pop$
###
region <- read.csv("https://pengdsci.github.io/datasets/countries_total.csv")
regions <- data.frame(country =  gsub(" ", "", region$name), iso_a3 = region$alpha.3)
###
IncLifeExp <- merge(income, life.exp, by = 'country')
IncLifeRegion <- merge(IncLifeExp, regions, by = 'country')
IncLifRegPop <- merge(IncLifeRegion, popsize, by = 'country')
IncLifRegPop$iso_a2 <- substr(IncLifRegPop$iso_a3, 1,2)
###
myWorld <- merge(World0, IncLifRegPop, by = 'iso_a2')
```

13.2 Creating Shapefiles from Dataframes

The next data set contains the geocode of the world's capital cities. This information will be used to create pop-ups to include specific information in the data. The geocode of the capital city can be found at https://www.kaggle.com/nikitagrec/world-capitals-gps. I also placed a copy of the data set at https://raw.githubusercontent.com/pengdsci/sta553/main/map/WorldCapitalGeocode.csv

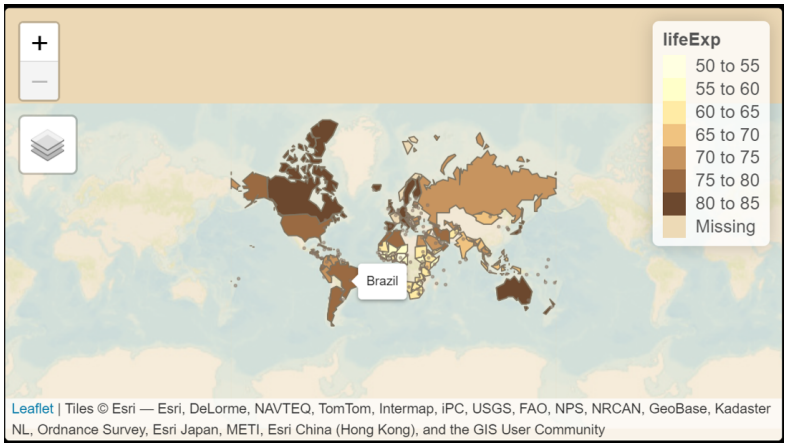
```
geocode <-read.csv("https://raw.githubusercontent.com/pengdsci/sta553/main/map/WorldCa
#geometry = paste('POINT (',CapitalLongitude ,',',CapitalLatitude,')')
#geocode$geometry = geometry
geocode$country <- gsub(" ", "", geocode$CountryName)
capital <- st_as_sf(geocode, coords = c("CapitalLongitude", "CapitalLatitude"), crs = 
###
IncLifeRegionCap <- merge(capital, IncLifeRegion, by = 'country')
IncLifeRegCapPop <- merge(IncLifeRegionCap, popsize, by = 'country')
```

13.3 Thematics Map with Created and Built-in Shapefiles

In this section, we use thematic maps to illustrate how to use modified shapefile (with additional information) and shapefiles created from data frames.

13.3.1 Example 1: Gapminder Data

```
library(tmap)
##
tmap_mode("view")  # "view" gives interactive map;
#tmap_style("classic") ## tmap_style set to "classic"
## other available styles are: "white", "gray", "natural",
## "cobalt", "col_blind", "albatross", "beaver", "bw", "watercolor"
tmap_options(bg.color = "skyblue",
             legend.text.color = "white")
##
tm_shape(myWorld) +
      tm_polygons("lifeExp",
                  legend.title = "Life Expectancy") +
      tm_layout(bg.color = "gray",
                inner.margins = c(0, .02, .02, .02)) +
tm_shape(IncLifeRegCapPop) +
      tm_symbols(col = "purple",
                 size = "pop15",
                 scale = .5,
                 alpha = 0.5,
                 popup.vars=c("CapitalName", "pop15", "inc00", "inc05", "inc10","inc15", "lif00",
```



13.3.2 Example 2: Philadelphia Neighborhood Shapefiles in Json

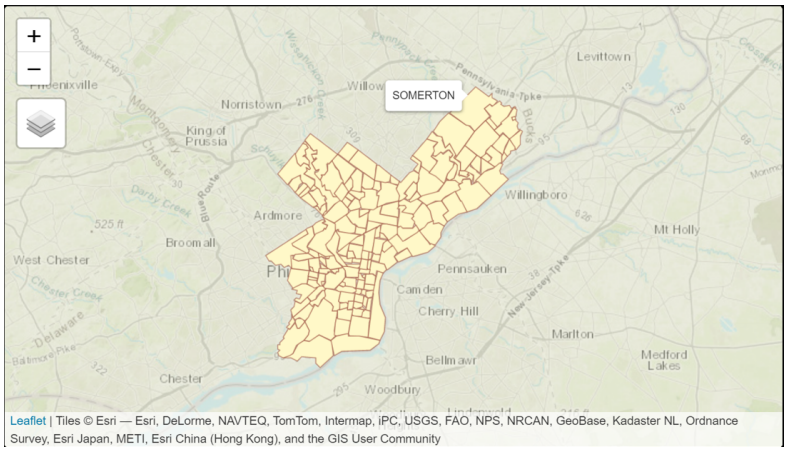
We can use the shapefile of any place to draw the base map of the place. For example, we can find the shapefile of the Philadelphia neighborhood at https://www.opendataphilly.org/dataset/covid-vaccinations/resource/473c9589-111b-43c9-a4a2-2dbe91f6dd7b?inner_span=True and draw the map of the Philadelphia neighborhood using the following map.

```
library(sf)
gshp = "https://github.com/azavea/geo-data/raw/master/Neighborhoods_Philadelphia/Neighb
philly <- st_read(gshp)
```

```
## Reading layer `Neighborhoods_Philadelphia' from data source
##   `https://github.com/azavea/geo-data/raw/master/Neighborhoods_Philadelphia/Neighbo
##   using driver `GeoJSON'
## Simple feature collection with 158 features and 8 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -75.28027 ymin: 39.867 xmax: -74.95576 ymax: 40.13799
## Geodetic CRS:  WGS 84
```

Since `rgeos/rgdal/maptools` are retired (because the maintainer is retired). Many packages that depend on some of the functions in these will have some issues at the moment. Json shape file is suggested for the moment. `.shp` seems to have issues to be loaded to R.

```
library(tmap)
tm_shape(philly) +
     tmap_options(check.and.fix = TRUE) +
      tm_polygons(border.col = "red",
                  border.alpha = 0.5) +
      tm_layout(bg.color = "skyblue",
                aes.color = c(fill = "skyblue", borders = "grey40",
  dots = "black", lines = "red", text = "black", na = "grey70"),
                inner.margins = c(0, .02, .02, .02))
```




```
if (!require("Stat2Data")) {
   install.packages("Stat2Data")
   library(Stat2Data)
}
# knitr::opts_knit$set(root.dir = "C:/Users/75CPENG/OneDrive - West Chester University of PA/Docu
# knitr::opts_knit$set(root.dir = "C:\\STA490\\w05")

knitr::opts_chunk$set(echo = TRUE,
                      warning = FALSE,
                      result = TRUE,
                      message = FALSE)
```

Chapter 14

Introduction to Tableau

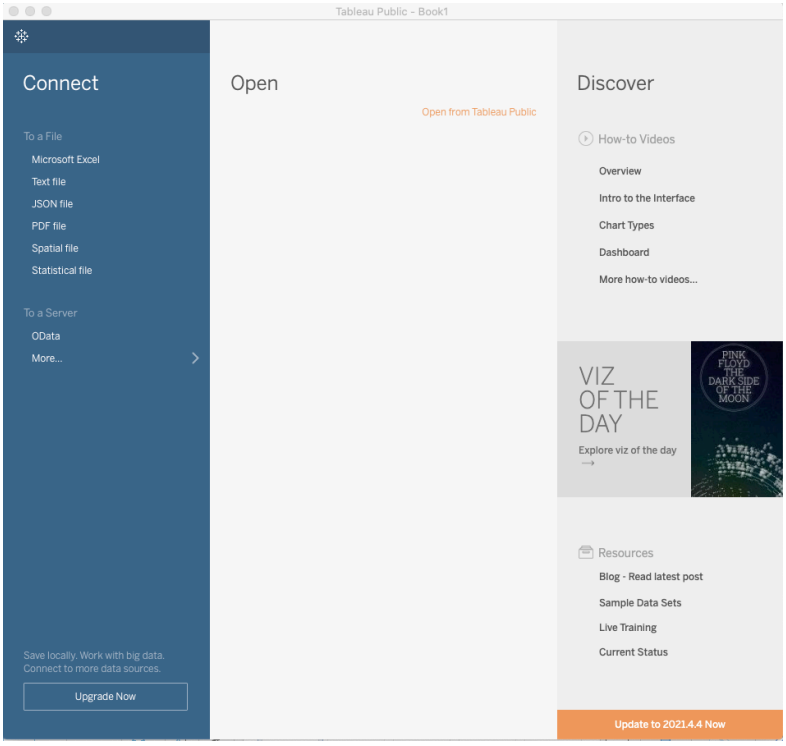
The Hawks data set was collected by students from Cornell College. It is built in in the R library {Stat2Data}. I also made a copy and posted it at https://raw.githubusercontent.com/pengdsci/sta553/main/Tableau/hawks.csv. You can download it and save it to a folder on your machine.

```
data("Hawks")
write.csv(Hawks, file="/Users/chengpeng/WCU/Teaching/2022Spring/STA553/tableau/hawks.csv")
```

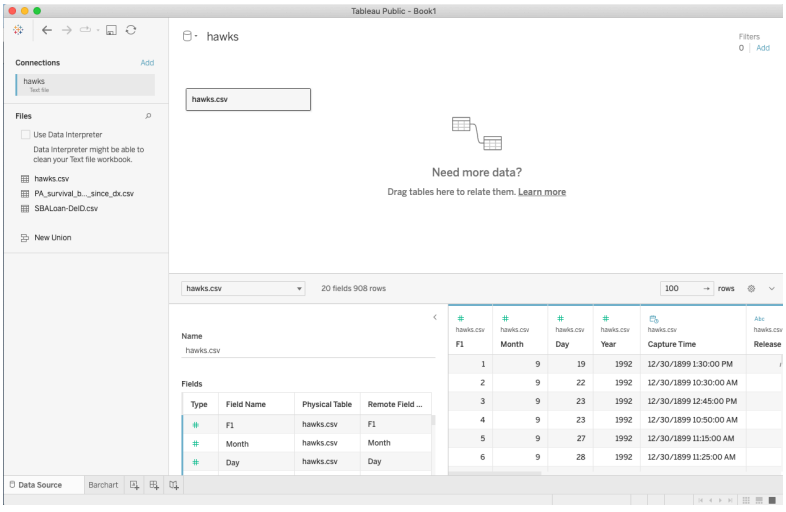
Tableau uses manual manual-driven approach to load data in certain formats. The following figure shows the types of external data that are connected to Tableau. There are some built-in data sets available in Tableau for practice purposes.

14.1 Data Loading

We open the program and see the following UI.



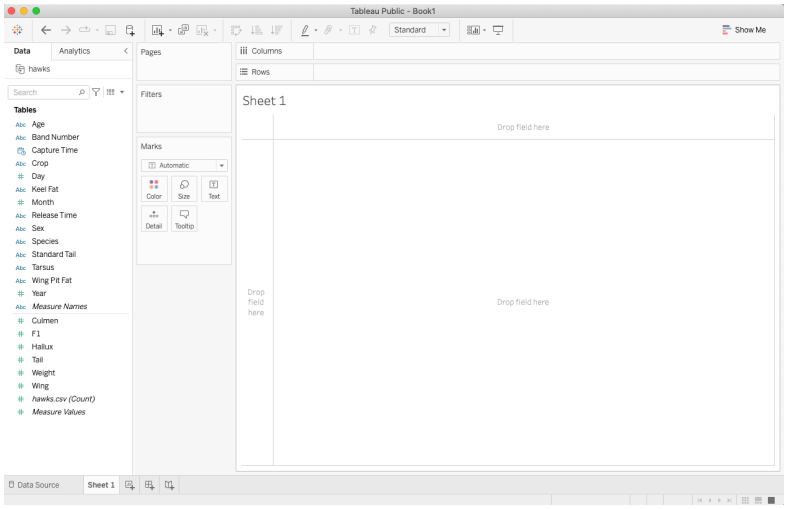
The Hawks data is in CSV format, we choose `text file` to connect to the data set. After the data is connected, we will see the following Data Source page with brief information on the data set.



We can explore the variables in the data set on the data source page. We can connect to multiple data sets and merge them on this page.

14.2 Opening Work Sheet

Click the Sheet Tab in the bottom left of the data source data, we will the list of the variables and the panels of visualization tools for making charts.



Statistical charts are created on different sheets. We can add more sheets as needed and change the default sheet name to a meaningful name.

Next, we create commonly used statistical charts in separate sheets.

14.3 Basic Statistical Charts with Existing Variables

14.3.1 Bar Chart

Bar charts are one of the most common data visualizations. We can use them to quickly compare data across categories, highlight differences, show trends and outliers, and reveal historical highs and lows at a glance. Bar charts are especially effective when you have data that can be split into multiple categories.

We consider the distribution of hawk species. Change `Sheet 1` to `barChart`.

Step 1: Drag `Species` to the Column field

Step 2: Drag `Species` to the Row field and change it to counts (frequencies).

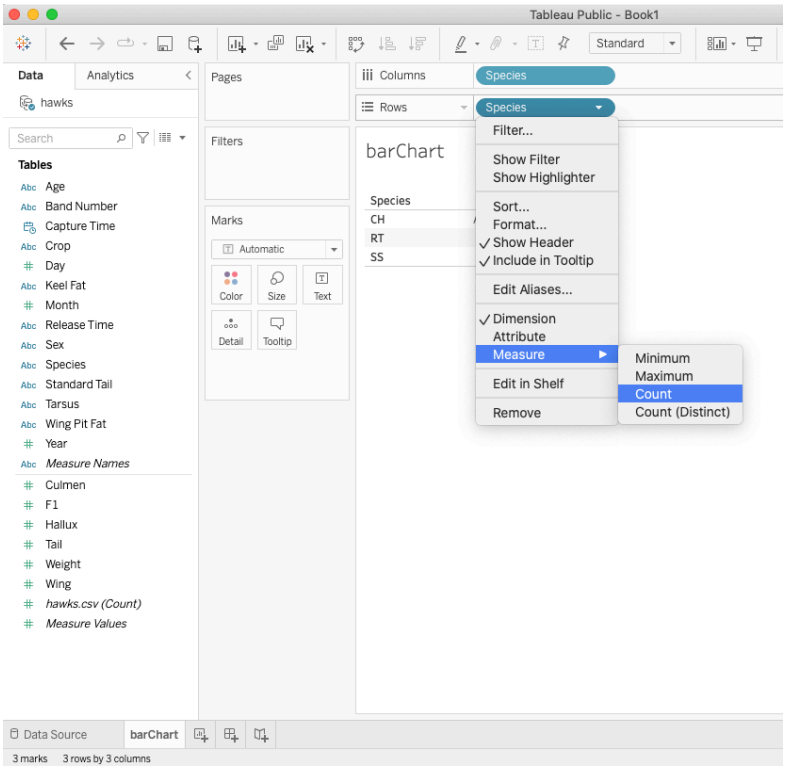
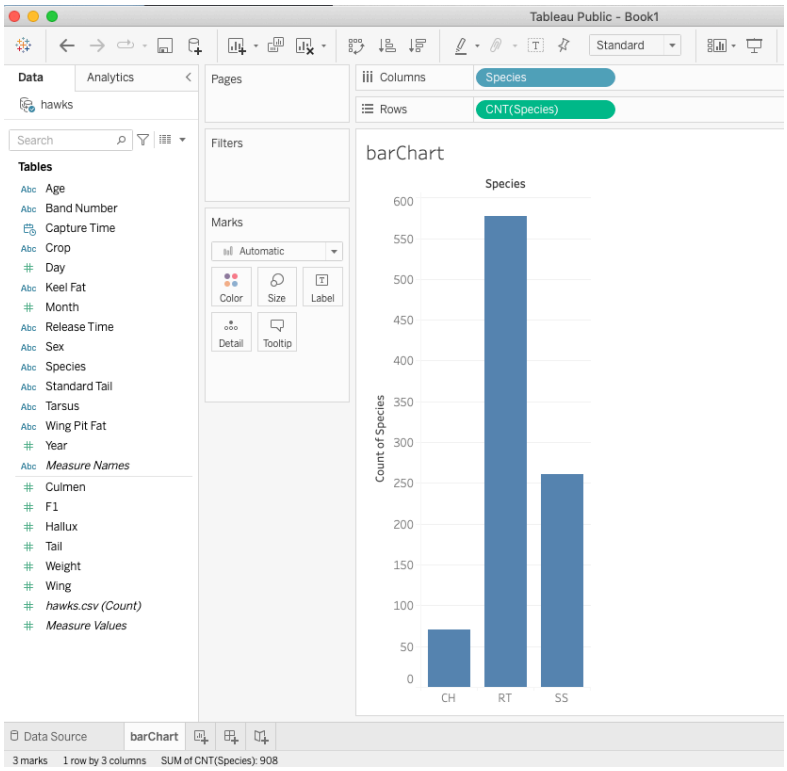


Tableau Public - Book1
Standard
Data
Analytics
hawks
Search
Tables
Abc Age
Abc Band Number
Capture Time
Abc Crop
Day
Abc Keel Fat
Month
Abc Release Time
Abc Sex
Abc Species
Abc Standard Tail
Abc Tarsus
Abc Wing Pit Fat
Year
Abc Measure Names
Culmen
F1
Hallux
Tail
Weight
Wing
hawks.csv (Count)
Measure Values
Pages
Filters
Marks
Automatic
Color
Size
Text
Detail
Tooltip
Columns
Species
Rows
Species
Filter...
Show Filter
Show Highlighter
Sort...
Format...
Show Header
Include in Tooltip
Edit Aliases...
Dimension
Attribute
Measure
Edit in Shelf
Remove
Minimum
Maximum
Count
Count (Distinct)
barChart
Species
CH
RT
SS
Data Source
barChart
3 marks
3 rows by 3 columns



14.3.2 Pie Chart

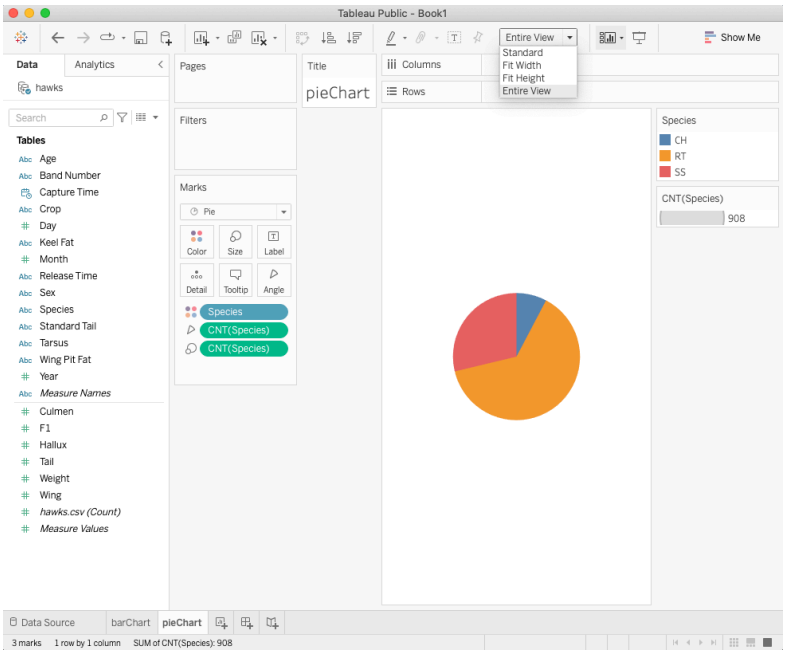
Pie charts are powerful for adding detail to other visualizations. Alone, a pie chart doesn't give the viewer a way to quickly and accurately compare information. Since the viewer has to create context on their own, key points from your data are missed. Instead of making a pie chart the focus of your dashboard, try using it to drill down on other visualizations.

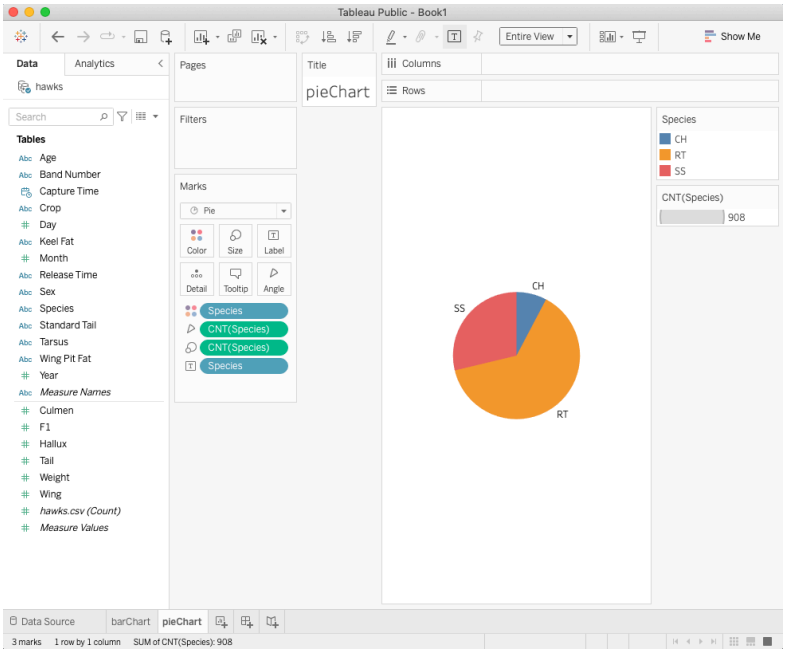
Step 1: Repeat the two steps in bar chart.

Step 2: Click `Show Me` on the top-right of the UI and select `piechart icon`.

Step 3: Select the `Entire View` from the top panel drop-down menu.

Step 4: drag `Species` to `Label` in **Marks** panel.





14.3.3 Histogram

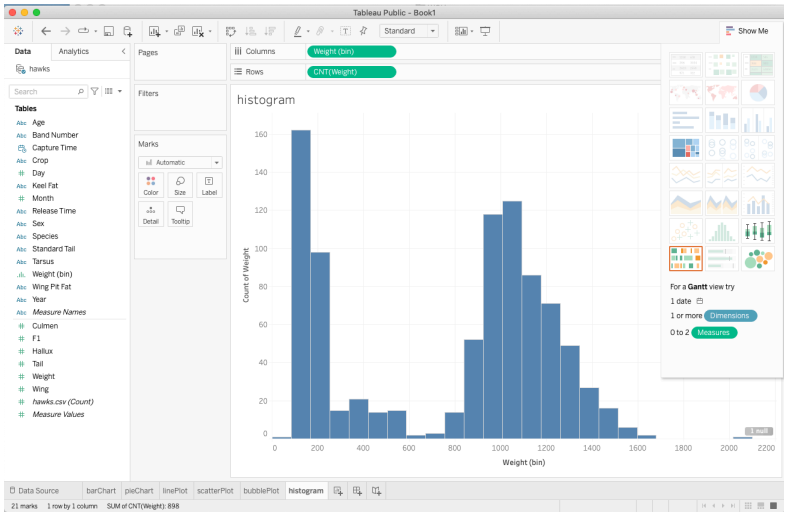
A histogram is a chart that displays the shape of a distribution. A histogram looks like a bar chart but groups values for a continuous measure into ranges or

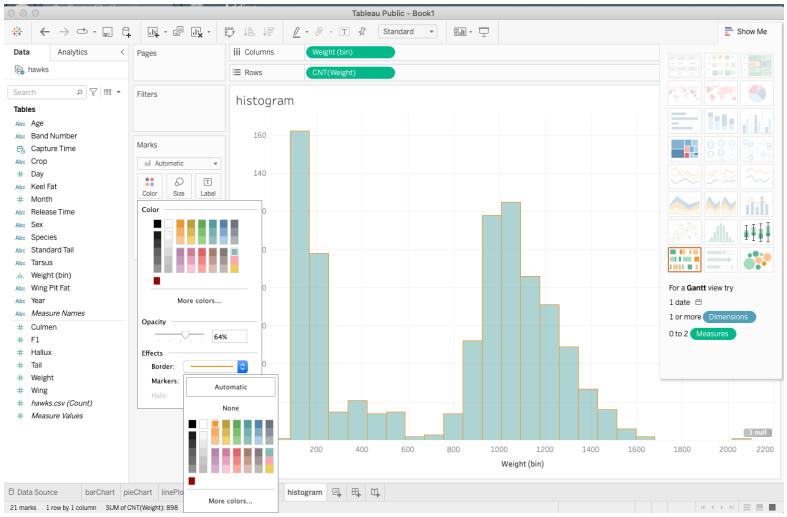
bins.

Step 1: drag `weight` to the column field.

Step 2: Click `show me`, in the drop-down menu and select the `histogram` icon.

Step 3: Click the `color` icon in the Marks panel to adjust the color and border of the histogram.





14.3.4 Box-plot

We can use box plots, also known as box-and-whisker plots, to show the distribution of values along an axis. Boxes indicate the middle 50 percent of the data (that is, the middle two quartiles of the data's distribution).

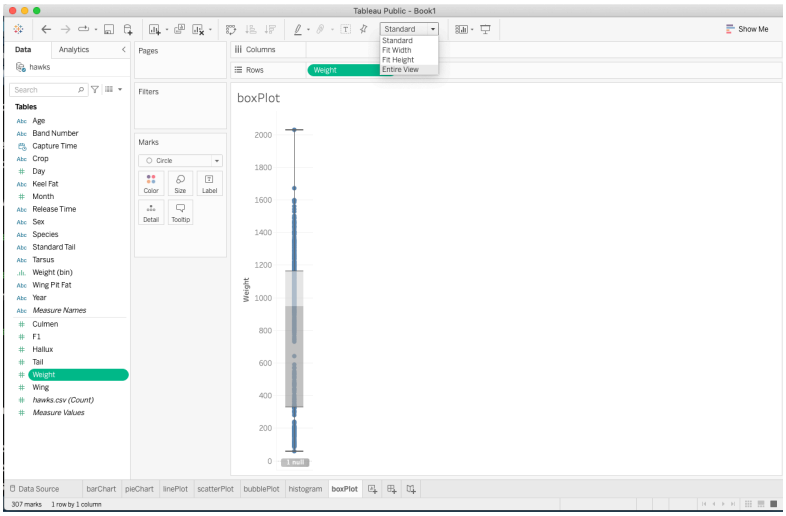
The following steps are used to create a simple box plot.

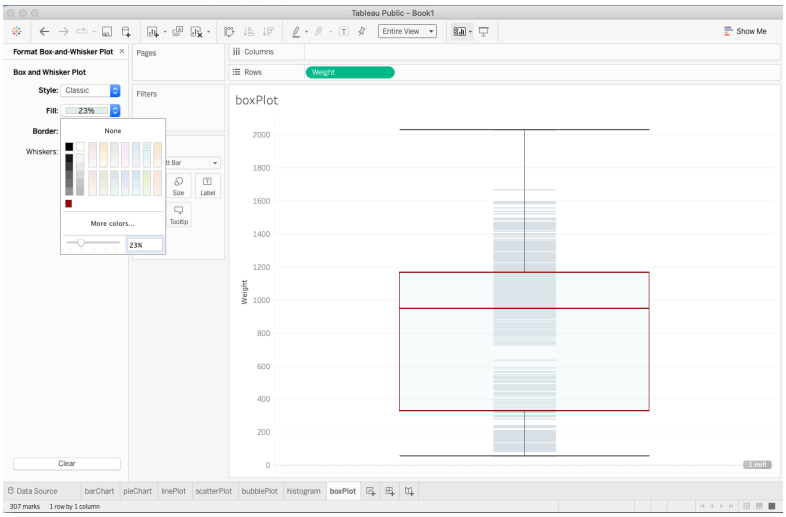
Step 1: add a numerical variable to the sheet (we use `weight` in this example).

Step 2: change `weight` to dimension. It will automatically create a default box plot with data points plotted on the numerical axis.

Step 3: change the appearance of the box-plot by selecting **Entire View** (see the screenshot)

Step 4: choose `Gantt Bar` to show the density of the values and edit the boxplot to get a better chart.





14.3.5 Line Chart

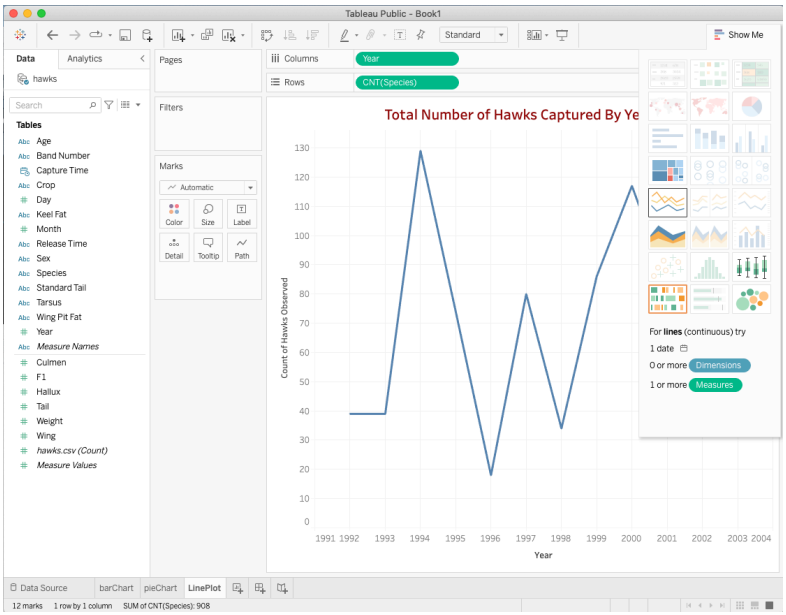
The line chart, or line graph, connects several distinct data points, presenting them as one continuous evolution. Use line charts to view trends in data, usually over time (like stock price changes over five years or website page views for the month). The result is a simple, straightforward way to visualize changes in one value relative to another.

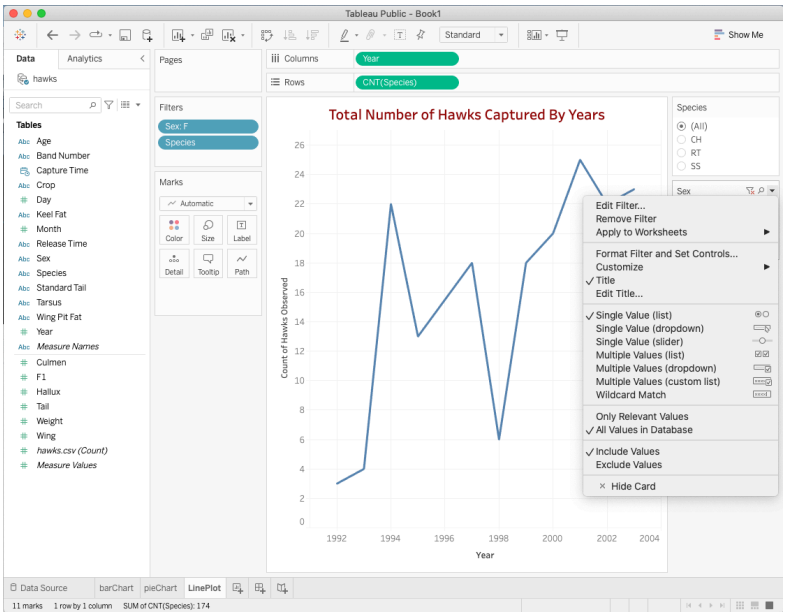
Step 1: Drap `year` to the column field and `Species` to the row field and convert them into frequencies.

Step 2: Click `Show Me` on the top-right of the UI and select the `line plot` icon.

Step 3: Right-click `Species` and `Sex` and send them to the **Filter** panel.

Step 4: Choose an appropriate display form of the filter (see the right panel of the following screenshot)





14.3.6 Scatter Plot

Scatter plots are an effective way to investigate the relationship between different variables, showing if one variable is a good predictor of another, or if they tend to change independently. A scatter plot presents lots of distinct data points on a single chart. The chart can then be enhanced with analytics like cluster analysis or trend lines.

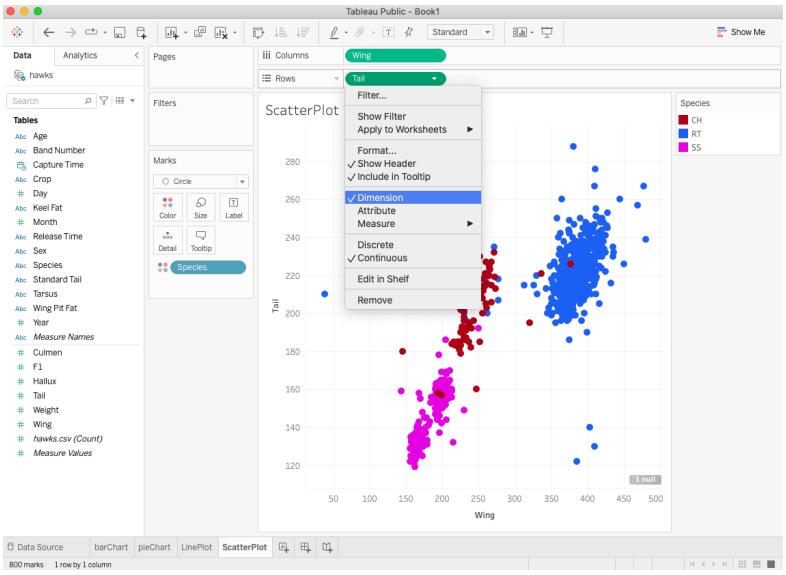
Let's explore the association between the lengths of wings and tails of hawks across the species. The following steps create a simple scatter plot in Tableau.

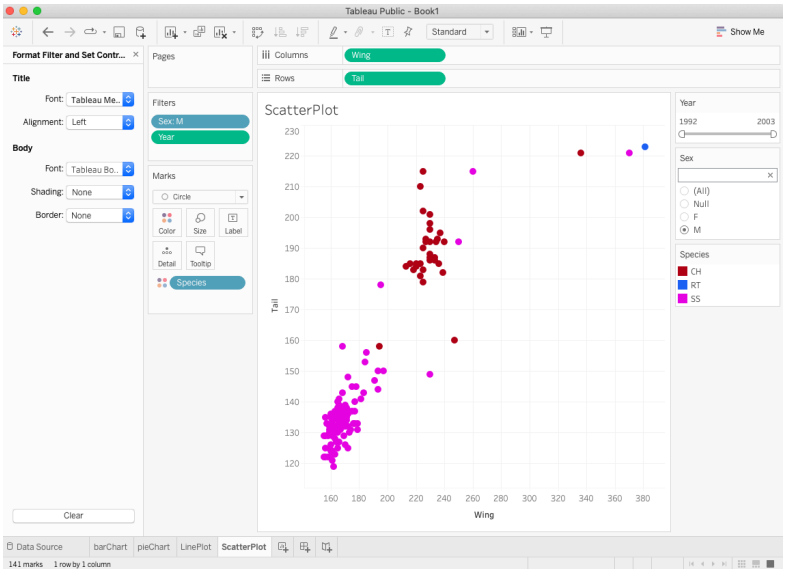
Step 1: Drag the two numerical variables to column and row fields.

Step 2: Change the two aggregated variables (by default) to `dimension` (see the left-hand side screenshot).

Step 3: Color code the species (drag `species` to the **color** mark).

Step 4: Choose the categorical variables to define filters to explore the association of a subset of the data (partial association) using a drop-down menu, radio button, slider, etc.





14.3.7 Bubble Chart

Although bubbles aren't technically their own type of visualization, using them as a technique adds detail to scatter plots or maps to show the relationship between three or more measures. Varying the size and color of circles create visually compelling charts that present large volumes of data at once.

A bubble chart is modified from a regular scatter plot. We next use the above scatter plot as a base plot and make the point size proportional to the value of

variable `wing`.

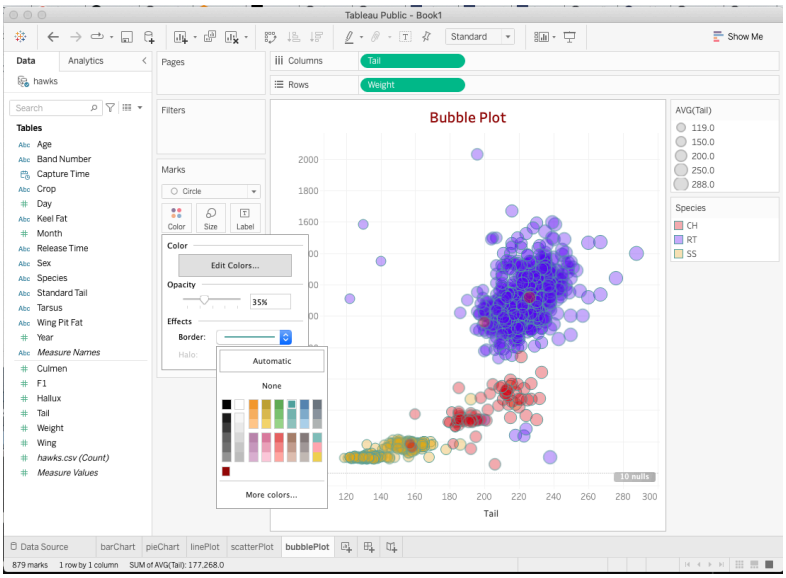
Step 1: create a basic scatter plot (following steps 1-4 in the previous section of the scatter plot).

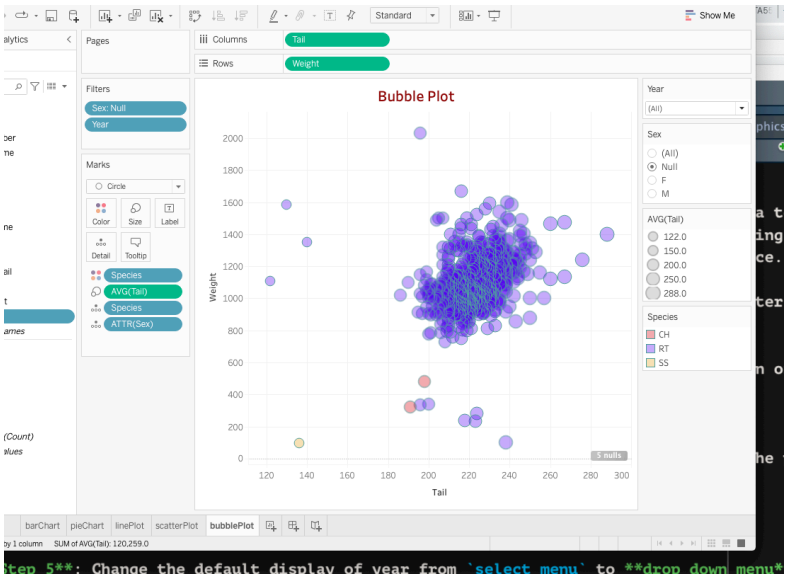
Step 2: drag variable `wing` to **size** icon (Marks panel).

Step 3: right **color** icon in Marks panel to adjust transparency and modify the point border to make partially overlapped points distinguishable.

Step 4: convert `year` to a string variable and add `sex` and `year` to the filter.

Step 5: Change the default display of year from `select menu` to **drop-down menu** and `sex` to **radio button**





14.3.8 Treemap

Treemaps relate different segments of your data to the whole. As the name of the chart suggests, each rectangle in a treemap is subdivided into smaller rectangles, or sub-branches, based on its proportion to the whole. They make efficient use of space to show the percent total for each category.

14.3.9 Maps

Maps are a no-brainer for visualizing any kind of location information, whether it's postal codes, state abbreviations, country names, or your own custom geocoding. If you have geographic information associated with your data, maps are a simple and compelling way to show how location correlates with trends in your data. Let's look at a small data set with geo-information. The data set can be found at https://raw.githubusercontent.com/pengdsci/datasets/main/Realestate.csv. We first download this data and save it to a local folder so we can connect the data to Tableau.

The following steps will create a map to view the spatial distribution of properties in the Bay Area.

Step 1. Drag `longitude` and `latitude` to row and column fields respectively.

Step 2. Click `Show me` and select the World Map in the list of the template plots.

Step 3. Go to the top menu bar, and click `Map` to select a background map.

Step 4. Click the `Color` shelf in the Marks field, and change the default color

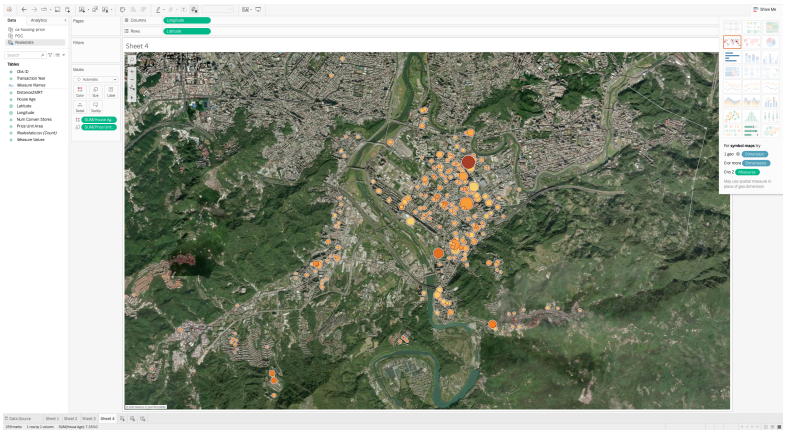
to an appropriate color.

Step 5. Choose an appropriate color.

Step 6 Select an appropriate variable to determine the point size.

Step 7 Drag the variable you want to display in the hover text.

The following screenshot shows the above steps.



The actual map is available on the Tableau Public Server at https://public.tableau.com/app/profile/cpeng/viz/Book1_16487389941160/Sheet4?publish=yes

14.3.10 Density Maps

Density maps reveal patterns or relative concentrations that might otherwise be hidden due to an overlapping mark on a map—helping you identify locations with greater or fewer numbers of data points. Density maps are most effective when working with a data set containing many data points in a small geographic area.

Let's use the POC (US gas station data) as an example of how to deal with many data points. The data can be found at: https://github.com/pengdsci/datasets/raw/main/POC.csv. We first download this data file save it into a local folder and then connect Tableau to this data.

The following suggested steps will create a density map for the US gas stations.

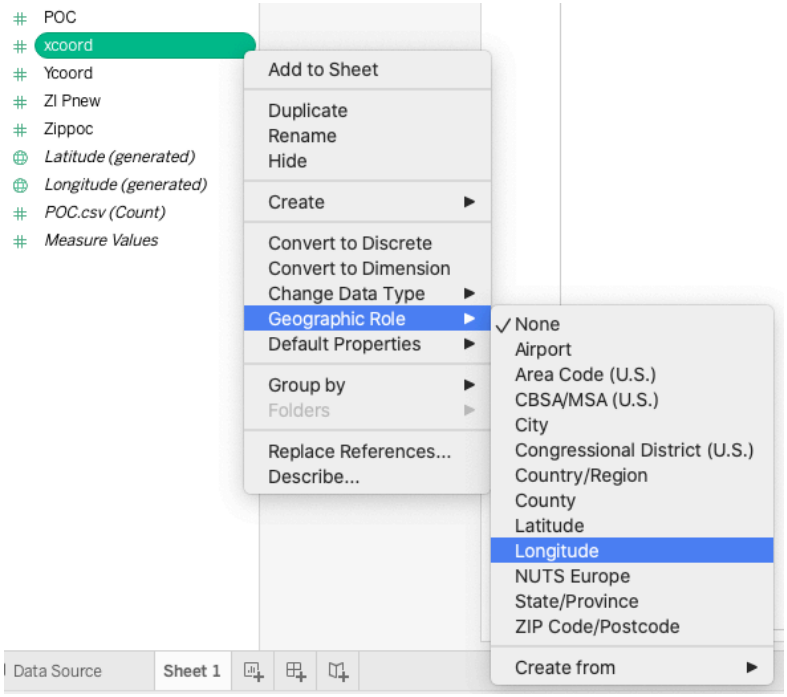
Step 1. Convert `xcoord` and `ycoord` to longitude and latitude (see the left screenshot below).

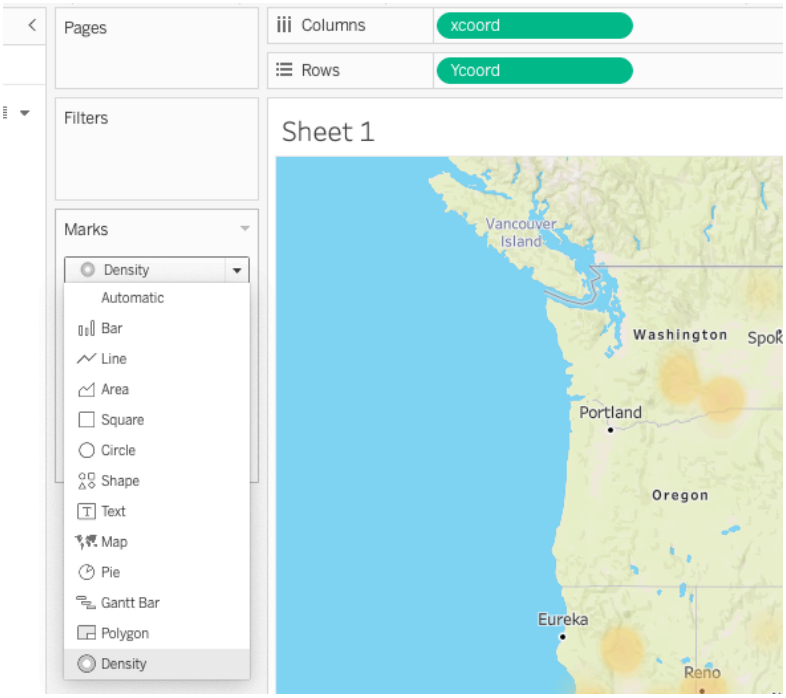
Step 2. Drag `xcoord` and `ycoord` to row and column fields respectively.

Step 3. In the drop-down menu of the Marks field, select `density`.

Step 4. Go to the top menu bar, and click `Map` to select a background map.

Step 5. Click the `Color` shelf in the Marks field, change the default color to an appropriate color.





14.4 Basic Charts with Derived Variables

Tableau has a lot of built-in functions that can be used to define derived variables. This section uses several examples to illustrate how to use some of the commonly used functions for creating statistical graphics. The complete list of these built-in functions can be found at https://help.tableau.com/current/pro/desktop/en-us/functions_all_alphabetical.htm

14.5 Tableau Dashboards

The data set is to be used in this case study. The visualization with be created using Tableau.

We first load the working data to R perform a simple exploratory data analysis and then decide what specific visualizations will be created.

The description of the data can be found at: https://github.com/pengdsci/sta553/raw/main/dash/mushroom-description.pdf

The data set can be found at: https://github.com/pengdsci/sta553/raw/main/dash/mushroom-data.csv

```
mushroom = read.csv("https://github.com/pengdsci/sta553/raw/main/dash/mushroom-data.csv")
names(mushroom)
```

```
##  [1] "class"                "cap.diameter"         "cap.shape"            "cap.surface"
##  [5] "cap.color"            "does.bruise.or.bleed" "gill.attachment"      "gill.spacing"
##  [9] "gill.color"           "stem.height"          "stem.width"           "stem.root"
## [13] "stem.surface"         "stem.color"           "veil.type"            "veil.color"
## [17] "has.ring"             "ring.type"            "spore.print.color"    "habitat"
## [21] "season"
```

Three numerical variables are summarized in the following.

```
summary(mushroom[,c(2,10,11)])
```

```
##   cap.diameter     stem.height       stem.width
##  Min.   : 0.380   Min.   : 0.000   Min.   :  0.00
##  1st Qu.: 3.480   1st Qu.: 4.640   1st Qu.:  5.21
##  Median : 5.860   Median : 5.950   Median : 10.19
##  Mean   : 6.734   Mean   : 6.582   Mean   : 12.15
##  3rd Qu.: 8.540   3rd Qu.: 7.740   3rd Qu.: 16.57
##  Max.   :62.340   Max.   :33.920   Max.   :103.91
```

```
char.var = mushroom[,-c(2,10,11)]
names(char.var)
```

```
##  [1] "class"                "cap.shape"            "cap.surface"          "cap.color"
##  [5] "does.bruise.or.bleed" "gill.attachment"      "gill.spacing"         "gill.color"
##  [9] "stem.root"            "stem.surface"         "stem.color"           "veil.type"
## [13] "veil.color"           "has.ring"             "ring.type"            "spore.print.color"
## [17] "habitat"              "season"
```

```
list(class = table(char.var$class),
     cap.shape = table(char.var$cap.shape),
     cap.surface = table(char.var$cap.surface),
     cap.color = table(char.var$cap.color),
     does.bruise.or.bleed = table(char.var$does.bruise.or.bleed),
     gill.attachment = table(char.var$gill.attachment),
     gill.spacing = table(char.var$gill.spacing),
     gill.color = table(char.var$gill.color),
     stem.root = table(char.var$stem.root),
     stem.surface = table(char.var$stem.surface),
     stem.color = table(char.var$stem.color),
     veil.type = table(char.var$veil.type),
     veil.color = table(char.var$veil.color),
     has.ring = table(char.var$has.ring),
     ring.type = table(char.var$ring.type),
     spore.print.color = table(char.var$spore.print.color),
     habitat = table(char.var$habitat),
```

```
    season = table(char.var$season)
)
```

```
## $class
##
##     e     p
## 27181 33888
##
## $cap.shape
##
##     b     c     f     o     p     s     x
##  5694  1815 13404  3460  2598  7164 26934
##
## $cap.surface
##
##           d     e     g     h     i     k     l     s     t     w     y
## 14120  4432  2584  4724  4974  2225  2303  1412  7608  8196  2150  6341
##
## $cap.color
##
##     b     e     g     k     l     n     o     p     r     u     w     y
##  1230  4035  4420  1279   828 24218  3656  1703  1782  1709  7666  8543
##
## $does.bruise.or.bleed
##
##     f     t
## 50479 10590
##
## $gill.attachment
##
##           a     d     e     f     p     s     x
##  9884 12698 10247  5648  3530  6001  5648  7413
##
## $gill.spacing
##
##           c     d     f
## 25063 24710  7766  3530
##
## $gill.color
##
##     b     e     f     g     k     n     o     p     r     u     w     y
##   954  1066  3530  4118  2375  9645  2909  5983  1399  1023 18521  9546
##
## $stem.root
##
```

```
##           b     c     f     r     s
## 51538  3177   706  1059  1412  3177
##
## $stem.surface
##
##           f     g     h     i     k     s     t     y
## 38124  1059  1765   535  4396  1581  6025  2644  4940
##
## $stem.color
##
##     b     e     f     g     k     l     n     o     p     r     u     w     y
##   173  2050  1059  2626   837   226 18063  2187  1025   542  1490 22926  7865
##
## $veil.type
##
##           u
## 57892  3177
##
## $veil.color
##
##           e     k     n     u     w     y
## 53656   181   353   525   353  5474   527
##
## $has.ring
##
##     f     t
## 45890 15179
##
## $ring.type
##
##           e     f     g     l     m     p     r     z
##  2471  2435 48361  1240  1427   353  1265  1399  2118
##
## $spore.print.color
##
##           g     k     n     p     r     u     w
## 54715   353  2118  1059  1259   171   182  1212
##
## $habitat
##
##     d     g     h     l     m     p     u     w
## 44209  7943  2001  3168  2920   360   115   353
##
## $season
##
##     a     s     u     w
```

```
## 30177  2727 22898  5267
```

The above frequency table indicates that several categorical variables have a significantly high percentage of missing values. Since we only perform visual analytics to illustrate how to use Tableau to create dashboards, we will not perform any data management for modeling purposes.

14.5.1 Design Dashboards with Tableau

We briefly introduced the basic statistics charts using Tableau. In this note, we choose both categorical and quantitative variables in the working data set to construct individual charts with Tableau and then **demonstrate how to use these charts to construct a dashboard** with Tableau. We will not write detailed steps here since there are too many different ways to do the same thing.

14.5.1.1 Individual Charts

We will construct five descriptive charts: a two-way contingency table, a donuts chart (a variation pie chart), a scatter plot, box plots, and a histogram.

14.5.1.2 Reactive Dashboard

We will use four individual charts to construct a dashboard that includes a reactive filter to update all charts in the dashboard.

14.5.2 Tableau Story Point

Tableau can create a form presentation of the existing individual chart so we can tell the story based on the Tableau charts.

14.6 Some Youtube Tutorials on Tableau

- https://www.youtube.com/watch?v=GrT0wlQ2LZQ fancy pie charts
- <https://public.tableau.com/views/US-States-Facts/Sheet1?:language=en-US&:display_count=n&:origin=viz_share_link
- https://public.tableau.com/views/US-States-Facts/Sheet1?:language=en-US&publish=yes&:display_count=n&:origin=viz_share_link
- https://www.youtube.com/watch?v=iFGt6j7GZX0 density curve
- https://www.youtube.com/watch?v=JvE2q0hWIIo
- https://www.youtube.com/watch?v=IIH19j_YG24 Excellent video!
- https://www.youtube.com/watch?v=BcKMPRpHqZ0 Dashboard
- https://www.youtube.com/watch?v=ZfpUzp8mBSw Donut chart

- https://www.youtube.com/watch?v=gWZtNdMko1k&list=PLWPirh4EWFpGXTBu8ldLZGJCUeTMBpJFK 91-video tutorials

Chapter 15

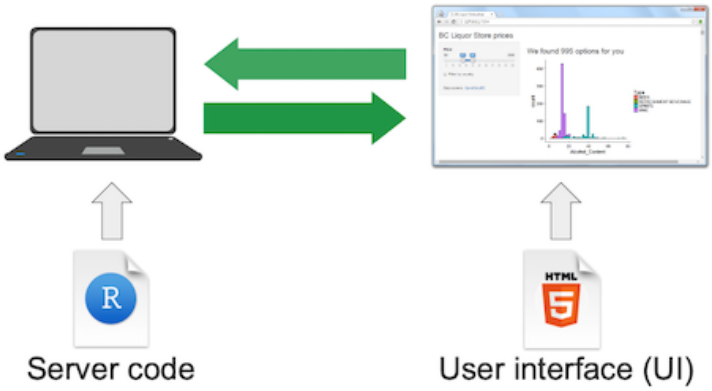
Getting Started with R Shiny

RShiny apps must run on a shiny server. This eBook (all three formats will be stored in the GitHub repository. We will not be able to show the interactivity of apps. Instead, screenshots will be included in this eBook to show the GUI designed based on the code. One can run the provided source code and run it on the local machine and see the interactive effect of the apps.

Shiny is a powerful and flexible R package that makes it easy to build interactive web applications and dynamic dashboards straight from R. These applications can be hosted on a standalone webpage or embedded in R Markdown documents.

Shiny is 10 years old now!

```
include_graphics("shiny/uiserver.png")
```



15.1 The IDE of RShiny Apps

Studio IDE has an environment for developing R shiny apps. There are two different ways to write shiny apps code in R:

Single File Method: In this single file, we write a `ui` function to design the UI and a `server` function to process user input information and output computed information.

Two File Method: In this method consists of two separate files: `ui.R` for designing UI and `server.R` for processing and computing the user input information. Both script files must be saved in the same folder you created for this app. If we choose to use this method, R will automatically create the two template files in the designated folder.

15.2 Embedding Shiny Apps in RMarkdown

We can also write shiny apps in RMarkdown. This is convenient and useful for drafting analysis reports that contain shiny apps and other interactive plots. This note is prepared to illustrate how to develop shiny apps with numerous examples. When writing shiny apps in RMarkdown, we have to specify `runtime: shiny` in the YAML header in order to correctly render the apps.

15.3 Components of An Shiny Applications

Programmatically, a Shiny application is simply a directory containing an R script called `app.R` which is made up of a user interface (`ui`) object and a `server` function. This folder could also contain some additional data, scripts, or other resources required to support the application.

```
include_graphics("shiny/shinyskeleton.png")
```

```
 # Load package shiny
library(shiny)
# Define UI for application
ui <- fluidPage(


)

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


}

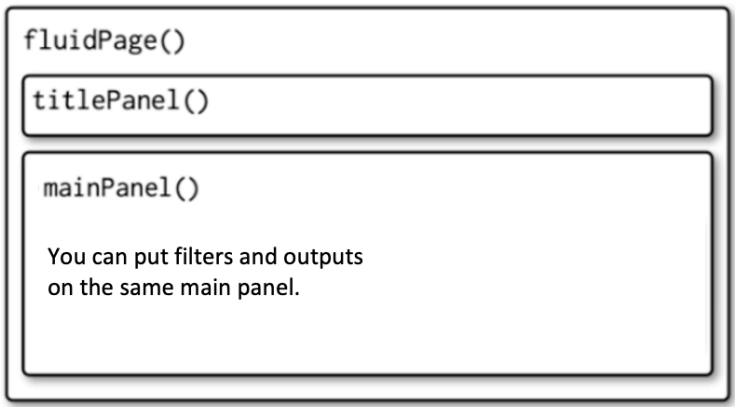
# Build and run the application
shinyApp(ui = ui, server = server)
```

- Function fluidPage() creates a dynamic HTML user interface you see when you look at an RShiny app. Convention is to save this as an object named ui.
- Function server() is user-defined and contains R commands your computer or external server need to run the app.
- Function shinyApp() builds the app based on the user interface and server pair of code.

15.4 The Anatomy of a Shiny Application

The following example shiny app uses a single panel layout. It simply stacks all informational panels on the same `fluidPage` (graphical page). We use the layout as an example to illustrate the structure of the R shiny app and use it to develop the first R shiny app for this class.

```
include_graphics("shiny/shinySinglePanel.png")
```



- `ui` designs the layout of the app uses `mainPanel()`. There are many different layout designs to be discussed later.
- `sliderInput` designs the slider input widget.
- `server()` contains all R code that generates the output.

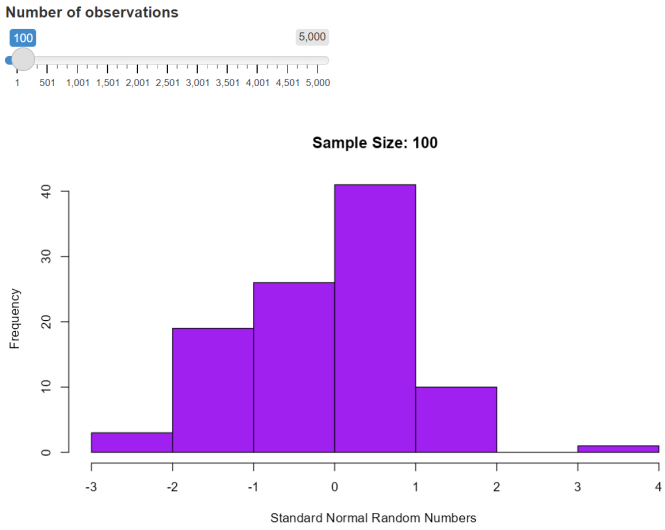
```
library(shiny)
### global R code
###
### if you have R functions that are used repeatedly or the
### The function itself is sophisticated, you can put it here
### to make your code tidy
##
```

```
##
### user interface - layout web interface for input and outputs
###
ui <- fluidPage(
    mainPanel(                                              # main panel, in general, we
                                                            # a sidePanel for inputs
        sliderInput(inputId = "obs",                        # sliderInput - one of the in
                                                            # 'obs' = input parameter via
                    label = "Number of observations",       # The title of the slider
                      min = 1,                              # the minimum input value
                      max = 5000,                           # the maximum input value
                    value = 100),                           # the default input is set to

        plotOutput(outputId = "distPlot")         # a place holder for the output to be
                                                  # created inside the server function
                                                  # "distPlot" is the reference of the

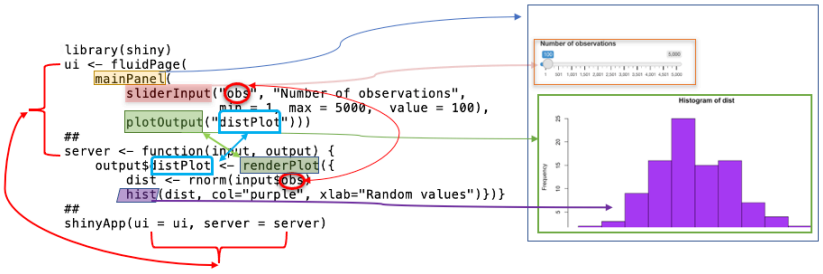
    )
)
#######
####### information to be processed and computed from the server side.
####### All code we wrote for visualization can be placed inside the server function as
#######
server <- function(input, output) {    # sever function passes two parameters:
                                       # 'input' = value from UI's sliderInput "obs",
                                       # 'output' = 'disPlot' in the output place holde
    output$distPlot <- renderPlot({    # 'renderPlot' prepares output plot - pay attent
                                       # This is how we pass the input information to t
                                       # and render the computed information to display
        dist <- rnorm(input$obs)       # simulate standard normal random numbers
                                       #  'obs' = from sliderInput in the UI
        hist(dist,                     # make a histogram use the regular plot function
            col="purple",              # fill the vertical bar with a color
            xlab="Standard Normal Random Numbers ",       # add a horizontal label
            main = paste("Sample Size:", input$obs)
            )
    })
}
### link the ui and server functions
shinyApp(ui = ui, server = server)
```

```
include_graphics("shiny/w11-shiny00.png")
```



15.5 How Shiny Apps Work?

```
include_graphics("shiny/shinyHowAppWorks.png")
```



15.6 Some Built-in Demonstrations of Shiny Apps

Package Shiny comes with some example apps. Enter any of the following in the R Console to see the Shiny app in action along with the code. The built-in shiny apps are named using their corresponding input widget names.

We modified the function and renamed it as `runExample2` to call apps without showing the source R code. If you only want to see the `siderbarPanel` and the `mainePanel`, you simply use the function `runExample()`. The source code of the modified function is not included in this class note. If you are interested in looking at the code, use the hyperlink in the following code chunk to view it.

```
source("https://raw.githubusercontent.com/pengdsci/sta553/main/shiny/shinyCodeExtractor
#runExample2("01_hello")        # a histogram
#runExample("02_text")         # tables and data frames
#runExample("03_reactivity")   # a reactive expression
#runExample("04_mpg")          # global variables
#runExample("05_sliders")      # slider bars
#runExample("06_tabsets")      # tabbed panels
#runExample("07_widgets")      # help text and submit buttons
#runExample("08_html")         # Shiny app built from HTML
#runExample("09_upload")       # file upload wizard
#runExample("10_download")     # file download wizard
#runExample("11_timer")        # an automated timer
```

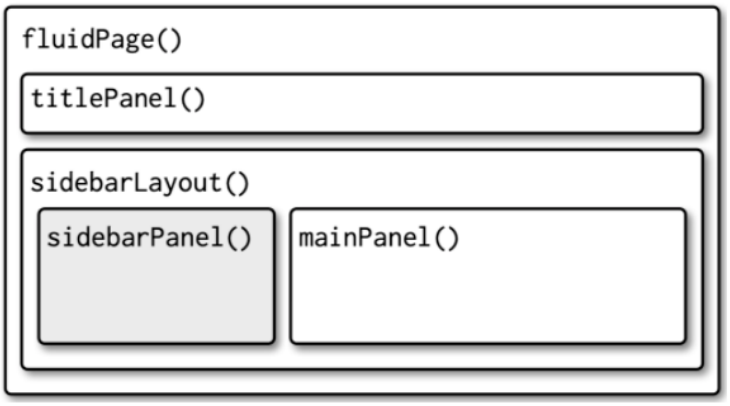
15.7 UI Layout Designs - A Glance

The UI function is responsible for designing the user interface design that includes the layout of the application and placeholders for the desired outputs.

15.7.1 Siderbar Layout

The basic shiny app side bar layout has the following structure.

```
knitr::include_graphics("shiny/shinySiderbarLayout.png")
```



The UI function should look like

```
ui = fluidPage(
        titlePanel(),
        sidebarLayout(
           sidebarPanel(),
           mainPanel()
        )
     )
```

For example, we can modify the above code to create an app with a sidebar navigation panel and an output main panel.

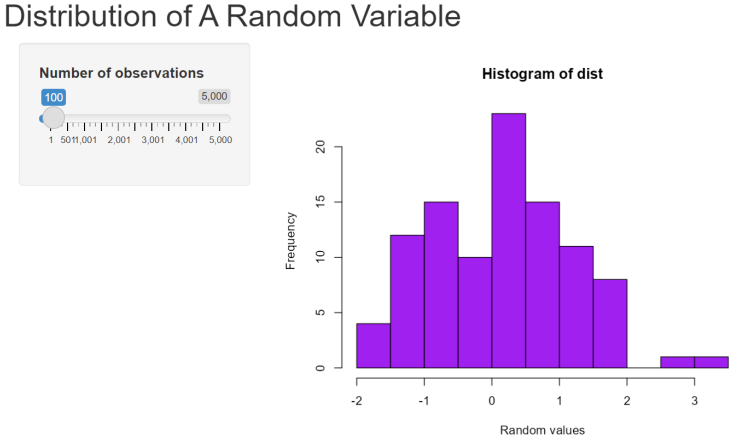
```
library(shiny)
### global R code
###
### if you have R functions that are used repeatedly or the
### the function itself is sophisticated, you can put it here
### to make your code tidy
##
##
### user interface - layout web interface for input and outputs
###
ui <- fluidPage(
    ##  overall title of  the app
    titlePanel("Distribution of A Random Variable"),      # title of the shiny app
    ## side bar navigation panel
    sidebarPanel(                                         # a sidePanel for inputs
        sliderInput(inputId = "obs",                      # sliderInput - one of the input widget
                                                          # 'obs' = input parameter via the slide
                    label = "Number of observations",     # The title of the slider
                      min = 1,                            # the minimum input value
                      max = 5000,                         # the maximum input value
                    value = 100),                         # the default input is set to be 100

    ),

    mainPanel(                                        # main panel, in general, we could add
        plotOutput(outputId = "distPlot")             # a place holder for the output to be
                                                      # created inside the server function.
                                                      # "distPlot" is the reference of the plot outpu
    )
)
#######
####### information to be processed and computed from server side.
####### all code we wrote for visualization can place inside the server function as needed.
#######
server <- function(input, output) {    # sever function passes two parameters:
                                       # 'input' = value from UI's sliderInput "obs",
                                       # 'output' = 'disPlot' in the output place holder.
    output$distPlot <- renderPlot({    # 'renderPlot' prepares output plot - pay attention to the
                                       # this how you pass the input information to the server
                                       # and render the computed information to display in the ou
        dist <- rnorm(input$obs)       # simulate standard normal random numbers
                                       #  'obs' = from sliderInput in the UI
        hist(dist,                     # make a histogram use the regular plot function hist()
```

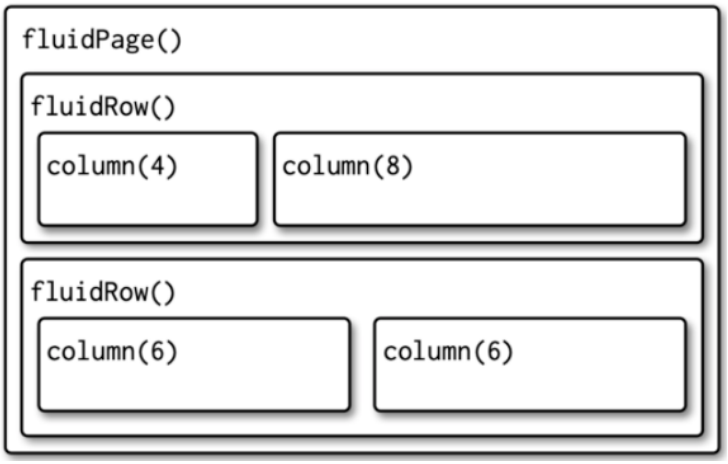
```
            col="purple",              # fill the vertical bar with a color
            xlab="Random values")      # add horizontal label
    })
}
### link the ui and server functions
shinyApp(ui = ui, server = server)
```

```
knitr::include_graphics("shiny/w11-shiny01.png")
```



In some situations, we more complex UI may be necessary, UI function can define a more sophisticated layout. For example, we can consider multi-row UI design such as the following layout.

```
include_graphics("shiny/shinyMultiRowLayout.png")
```



`Each row is made up of 12 columns` and the first argument to column() gives how many of those columns to occupy. We will illustrate how to develop

shiny apps using this layout later.

15.8 Some Basic Input Widgets

The 11 built-in apps in the demonstration have provided different forms of input methods. Here are some that are used frequently in practice. We have used `sliderInput` in the example. Next, we list a few more commonly used ones.

15.8.1 Free Text

This input method is practically useful when building apps that take information from the user directly for analysis and visualization. The following UI design is a general text input.

```
include_graphics("shiny/shinyTextInput.png")
```

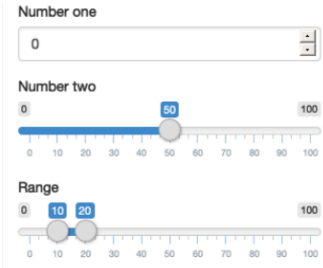
```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "What's your password?"),
  textAreaInput("story", "Tell me about yourself", rows = 3)
)
```

15.8.2 Numeric Inputs

In `sliderInput`, we can select one or a range of numbers as input. We can also design a dialog box to type a specific number as an input.

```
include_graphics("shiny/shinyNumericInput.png")
```

```
ui <- fluidPage(
  numericInput("num", "Number one", value = 0, min = 0, max = 100),
  sliderInput("num2", "Number two", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```



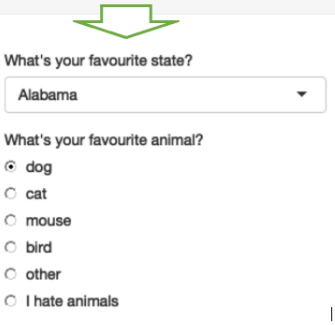
15.8.3 Limited Choices

Whenever a visualization involves a partition based on a discrete variable, selecting a value of a partition variable is useful. There are two different approaches to allow the user to choose from a pre-specified set of options: `selectInput()` and `radioButtons()`.

```
include_graphics("shiny/shinyLimitedChoices.png")
```

```
animals <- c("dog", "cat", "mouse", "bird", "other", "I hate animals")

ui <- fluidPage(
  selectInput("state", "What's your favorite state?", state.name),
  radioButtons("animal", "What's your favorite animal?", animals)
)
```



15.8.4 File Uploading and Downloading

When working data sets are large or in a special format, uploading and downloading files to Shiny Server becomes necessary. We can design a UI that allows users to upload for analysis and visualization and download the output data. We will cover this topic later.

15.9 Case Study I - Density of Normal Distribution

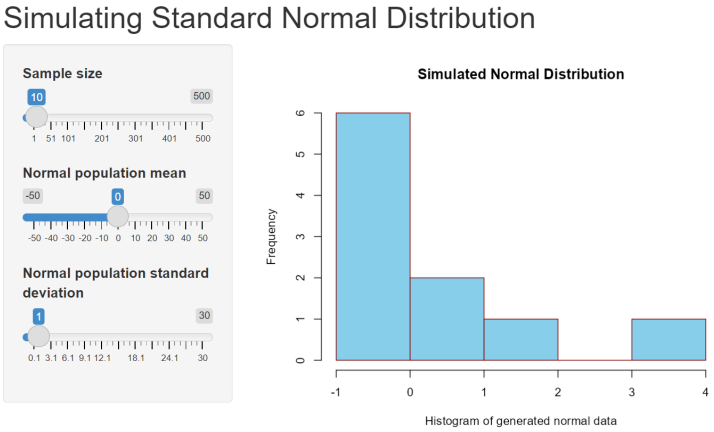
We revised the previous example and redesigned the layout by including a title, a sidebar, and a main panels. We place three input widgets requesting sample size, the population mean, and standard deviation. The output histogram will be placed in the main panel. The following is the detailed code

```
library(shiny)

# Define UI for app that draws a histogram ----
ui <- fluidPage(
    # App title ----
    titlePanel("Simulating Standard Normal Distribution"),
    # Sidebar layout with input and output definitions ----
    sidebarLayout(    # siderbarLayout:
         # Sidebar panel for inputs ==> the 1st parameter
         sidebarPanel(
               # Input: Slider for the number of bins ==> 2nd parameter
               sliderInput(inputId = "n",
                           label = "Sample size",
                           min = 1,
                           max = 500,
                           value = 10),
               # slider input: normal population mean
               sliderInput(inputId = "mu",
                           label = "Normal population mean",
                           min = -50,
                           max = 50,
                           value = 0),
               # normal population standard deviation
               sliderInput(inputId = "sigma",
                           label = "Normal population standard deviation",
                           min = 0.1,
                           max = 30,
                           value = 1)),
         # Main panel for displaying outputs ----
         mainPanel(
              # Output: Histogram ----
              plotOutput(outputId = "distPlot")
         )
   ))
# Define server logic required to draw a histogram ----
server <- function(input, output) {
  # Histogram of the simulated normal data ----
  # with requested number of bins
```

```
  output$distPlot <- renderPlot({
    # random number generation
    norm.score <- rnorm(input$n, input$mu, input$sigma)
    ##
    hist(norm.score, col = "skyblue", border = "darkred",
         xlab = "Histogram of generated normal data",
         main = "Simulated Normal Distribution")
      })
  }
# Create Shiny app ----
shinyApp(ui = ui, server = server)
```

```
include_graphics("shiny/w11-shiny02.png")
```



15.10 Case Study 2 - Central Limit Theorem

In this case study, we simulate the sampling distribution of the sample mean (i.e., the central limit theorem)

```
library(shiny)
# Define UI
ui <- fluidPage(
  # App title ----
  titlePanel(h3("Visualizing Distribution of Sample Means",
            align = "center", style = "color:navy", br(), br())
            ),

  # Sidebar layout with input and output definitions ----
  sidebarLayout(
    # Sidebar panel for inputs ----
    sidebarPanel(
```

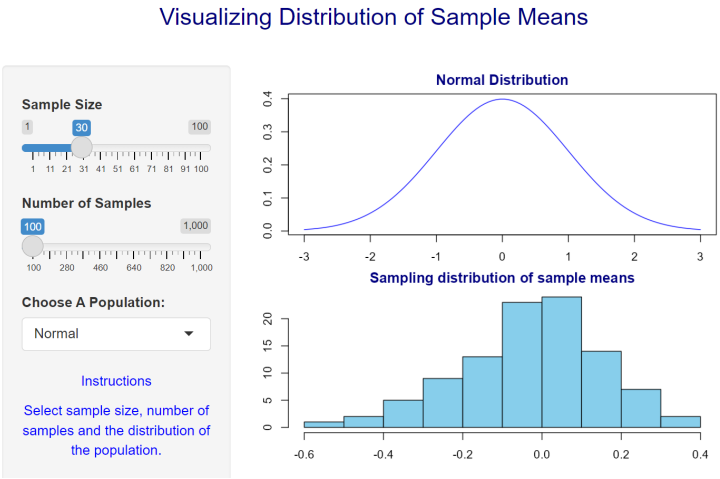
```
            # Input: Sample size
            p(sliderInput(inputId = "num", "Sample Size",
                          min = 1, max = 100, value = 30),
                          align = "center", style = "color:blue"),
            # Input: number of samples
            sliderInput(inputId = "num2", "Number of Samples",
                          min = 100, max = 1000, value = 100),
            # Input: Selector for populations---
            selectInput(inputId = "dist",
                          label = "Choose A Population:",
                          choices = list("Normal" = 1,
                                         "Uniform" = 2,
                                         "Exponential" = 3,
                                         "Binomial" = 4,
                                         "Poisson" = 5)),
            p("Instructions", align = "center", style = "color:blue"),
            p("Select sample size, number of samples
              and the distribution of the population.",
              align = "center", style = "color:blue")),

    # Main panel for displaying outputs ----
    mainPanel(  # Output: Histogram ----
                plotOutput(outputId = "distPlot"))
          )
  )

# Define server logic to summarize and view selected dataset ----
server <- function(input, output) {
    ##
    output$distPlot = renderPlot({
           n = input$num      # sample size
        num2 = input$num2     # number of samples
        xbar = NULL           # zero vector with num2 zeroes
        for(i in 1:num2)
           {
             if (input$dist == 1) xbar[i] = mean(rnorm(n))
             if (input$dist == 2) xbar[i] = mean(runif(n))
             if (input$dist == 3) xbar[i] = mean(rexp(n))
             if (input$dist == 4) xbar[i] = mean(rbinom(n,20,0.35))
             if (input$dist == 5) xbar[i] = mean(rpois(n,1.5))
        }
        par(mfrow = c(2,1), mar = c(2, 2, 2, 2))
           if (input$dist == 1) {
               x=seq(-3, 3, length=100)
               plot(x, dnorm(x), type="l", col = "blue",
                    xlab="", ylab="",
```

```
                    main="Normal Distribution",
                    col.main = 'navy')
            }
            if (input$dist == 2) {
              x=seq(0, 1, length=100)
              plot(x, dunif(x), type="l", col = "blue",
                    xlab="", ylab="",
                    main="Uniform Distribution",
                    col.main = 'navy')
            }
            if (input$dist == 3) {
              x=seq(0, 6, length=100)
              plot(x, dexp(x), type="l", col = "blue",
                    xlab="", ylab="",
                    main="Exponential Distribution",
                    col.main = 'navy')
            }
            if (input$dist == 4) {
              x = 0:20
              plot(x, dbinom(x, 20, 0.25), type="h", col = "blue",
                    xlab="", ylab="",
                    main="Binomial Distribution",
                    col.main = 'navy')
            }
            if (input$dist == 5) {
              x = 0:8
              plot(x, dpois(x, 1.5), type="h", col = "blue",
                    xlab="", ylab="",
                    main="Poisson Distribution",
                    col.main = 'navy')}
          ###
          hist(xbar, xlab="Random Numbers",
              main = "Sampling distribution of sample means",
              col="skyblue",
              col.main = 'navy')
        })
  }
# Create Shiny app ----
shinyApp(ui, server)
```

```
include_graphics("shiny/w11-shiny03.png")
```



15.11 More Effective Code

```
library(shiny)
# Define UI
ui <- fluidPage(
  # App title ----
  titlePanel(h3("Visualizing Distribution of Sample Means",
             align = "center", style = "color:navy", br(), br())
             ),

  # Sidebar layout with input and output definitions ----
  sidebarLayout(
    # Sidebar panel for inputs ----
    sidebarPanel(
            # Input: Sample size
            p(sliderInput(inputId = "num", "Sample Size",
                        min = 1, max = 100, value = 30),
                        align = "center", style = "color:blue"),
            # Input: number of samples
            sliderInput(inputId = "num2", "Number of Samples",
                        min = 100, max = 1000, value = 100),
            # Input: Selector for populations---
            selectInput(inputId = "dist",
                        label = "Choose A Population:",
                        choices = list("Normal" = 1,
                                       "Uniform" = 2,
                                       "Exponential" = 3,
                                       "Binomial" = 4,
                                       "Poisson" = 5)),
```

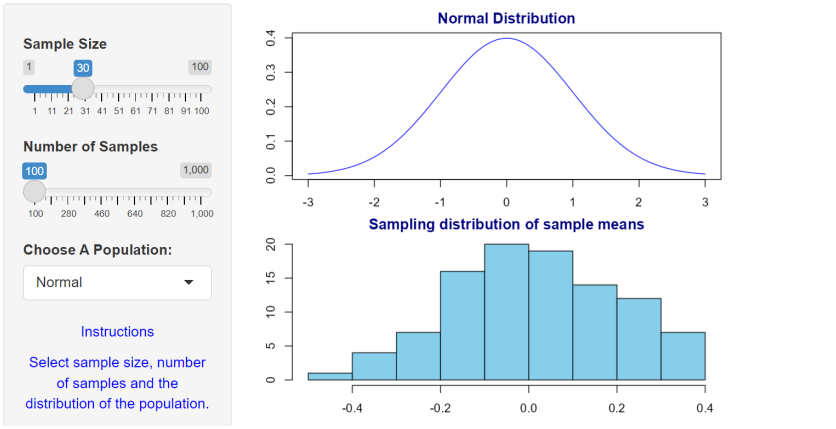
```
            p("Instructions", align = "center", style = "color:blue"),
            p("Select sample size, number of samples
              and the distribution of the population.",
              align = "center", style = "color:blue")),

    # Main panel for displaying outputs ----
    mainPanel(  # Output: Histogram ----
                plotOutput(outputId = "distPlot"))
          )
  )

# Define server logic to summarize and view selected dataset ----
server <- function(input, output) {
    ##
    output$distPlot = renderPlot({
           n = input$num      # sample size
        num2 = input$num2     # number of samples
       par(mfrow = c(2,1), mar = c(2, 2, 2, 2))
       if (input$dist == 1) {
            xbar <- apply(matrix(rnorm(n*num2), ncol=num2),2,mean)
            x=seq(-3, 3, length=100)
            plot(x, dnorm(x), type="l", col = "blue",
            xlab="", ylab="",
            main="Normal Distribution",
            col.main = 'navy')
          }else if (input$dist == 2) {
            xbar <- apply(matrix(runif(n*num2), ncol=num2),2,mean)
            x=seq(0, 1, length=100)
            plot(x, dunif(x), type="l", col = "blue",
                    xlab="", ylab="",
                    main="Uniform Distribution",
                    col.main = 'navy')
          }else if (input$dist == 3) {
            xbar <- apply(matrix(rexp(n*num2), ncol=num2),2,mean)
               x=seq(0, 6, length=100)
               plot(x, dexp(x), type="l", col = "blue",
                    xlab="", ylab="",
                    main="Exponential Distribution",
                    col.main = 'navy')
          }else if (input$dist == 4) {
            xbar <- apply(matrix(rbinom(n*num2,20,0.35), ncol=num2),2,mean)
            x = 0:20
            plot(x, dbinom(x, 20, 0.25), type="h", col = "blue",
                    xlab="", ylab="",
                    main="Binomial Distribution",
                    col.main = 'navy')
```

```
          }else if (input$dist == 5) {
            xbar <- apply(matrix(rpois(n*num2,1.5), ncol=num2),2,mean)
            x = 0:8
            plot(x, dpois(x, 1.5), type="h", col = "blue",
                    xlab="", ylab="",
                    main="Poisson Distribution",
                    col.main = 'navy')
          }
          ###
          hist(xbar, xlab="Random Numbers",
              main = "Sampling distribution of sample means",
              col="skyblue",
              col.main = 'navy')
        })
  }
# Create Shiny app ----
shinyApp(ui, server)
```

```
include_graphics("shiny/w11-shiny04.png")
```



Chapter 16

Shiny UI and Server Design

Shiny is a powerful and popular web framework for R programmers to elevate the way people consume analytics for both technical and non-technical decision-makers. It is used in many organizations from start-ups to top-trafficked websites.

You can see from the last lecture that we can create a highly interactive shiny app to engage users in the applications in simulations. This lecture focuses on creating shiny apps for data analysis. The benefits of Shiny in business analytics obvious

16.1 Naming Conventions Revisited

Most programming languages seem to have their own take on how users should name something. This is not the case in R. Actually, R does not have an official naming convention. R community uses naming conventions in many other languages. Below is a list of some of the most common conventions.

- **myVariableName** - lower camel case. This naming convention is used in Java and JavaScript.
- **MyVariableName** - upper camel case / PascalCase. This naming convention is used for class names in many languages including Java, Python, and JavaScript.
- **my__variable__name** - underscore_separated (also called snake case). This naming convention is used for function and variable names in many languages including C++, Perl, and Ruby.
- **my.variable.name** - Period.separated. This naming convention is unique to R and used in many core functions such as `as.numeric` or `read.table`.

- **myvariancename** - all lower case. This naming convention is common in MATLAB. Note that a single lowercase name, such as mean, conforms to all conventions but UpperCamelCase.

In the shiny library, the lowerCamelCase conventions are used. In the back-end coding, you can use any of the above naming conventions or a hybrid of different naming conventions.

16.2 Working Data Set

This note shows the steps of creating an R shiny application for data analysis.

R Shiny applications are effective tools for exploratory visual analytics. As in other analyses, it is essential to understand the data, know the critical information, and use appropriate tools and effective approaches to convey the information to your audience.

Creating shiny applications is a combination of arts, technology, and science.

In this note, I use the well-known `iris data set` to illustrate how to create shiny applications in data analysis.

We create two code chunks to explain the two modules (ui and server) separately.

16.3 UI Module: Front-end Design

Graphical Analysis Plan* ()

1. Scatter plot
2. Linear regression -inferential table
3. Prediction

User input information (UI)

1. Species
2. Selection of X and Y variable
3. Input value of X for prediction

I use three different input control widgets in this case study. A list of commonly used input control widgets as well as the code to create the widget can be found at https://shiny.rstudio.com/gallery/widget-gallery.html.

We next design the UI in the following code chunk.

```
library(shiny)
##  Working data set
##  The iris data set will be used in this app.
iris0 = read.table("https://raw.githubusercontent.com/pengdsci/sta553/main/shiny/iris0
```

```
x.names = names(iris0)[-c(1,6)]
y.names =names(iris0)[-1]
##
# Define UI for random distribution app ----
ui <- fluidPage(
  # App title ----
    titlePanel("Analysis of Iris Data"),
  #############
  # Sidebar layout with input and output definitions ----
    sidebarLayout(
    # Sidebar panel for inputs ----
      sidebarPanel(
         # Input: Select the random distribution type ----
         radioButtons("species", "Species",
                     c("setosa",
                       "versicolor",
                       "virginica",
                       "all"),
                     inline = FALSE,
                     selected = "all"),

      # br() element to introduce extra vertical spacing ----
      hr(),

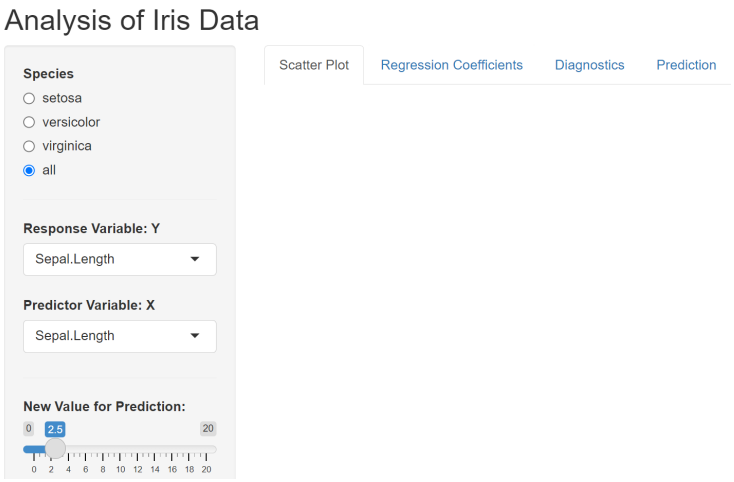
      # Input: Slider for the number of observations to generate ----
      selectInput("Y",
                  "Response Variable: Y",
                  y.names),

      selectInput("X",
                  "Predictor Variable: X",
                  x.names),
      ####
       hr(),
      ###
        sliderInput("newX", "New Value for Prediction:", 2.5, min = 0, max = 20, step = 0.1)
       ),
    # Main panel for displaying outputs ----
    mainPanel(
      # Output: Tabset w/ plot, summary, and table ----
      tabsetPanel(type = "tabs",
                  tabPanel("Scatter Plot", plotOutput("plot")),
                  tabPanel("Regression Coefficients", tableOutput("table")),
                  tabPanel("Diagnostics",plotOutput("diagnosis")),
                  tabPanel("Prediction", plotOutput("predPlt"))
```

```
      )

    )
  )
)
##
## Server function to be written in the next section for analysis.
server <- function(input, output) {}
## Connection between the UI and the Server
shinyApp(ui = ui, server = server)
```

```
include_graphics("shiny/w11-shinyAnalysis01.png")
```



16.4 Server Module: Back-end Coding

In the next code chunk, I copy the UI function written in the previous section and draft the server function.

I use species as a filter variable. I also allow users to choose response and predictor variables to build a linear regression model and use the model to predict by providing new values of the predictor variable.

We chose `tabset` layout to design this shiny app.

```
library(shiny)
##  Working data set
##  The iris data set will be used in this app.
iris0 = read.table("https://raw.githubusercontent.com/pengdsci/sta553/main/shiny/iris0
x.names = names(iris0)[-c(1,6)]
y.names =names(iris0)[-1]
##
```

```
# Define UI for random distribution app ----
ui <- fluidPage(

  # App title ----
        titlePanel(
           h4("Analysis of Iris Data",
                  align = "left", style = "color:navy", br(), br())),
  #############
  # Sidebar layout with input and output definitions ----
    sidebarLayout(
    # Sidebar panel for inputs ----
      sidebarPanel(

            tags$head(
               tags$style("body {background-color: white }")),

         # Input: Select the random distribution type ----
         radioButtons("species", "Species",
                   c("setosa",
                     "versicolor",
                     "virginica",
                     "all"),
                   inline = FALSE,
                   selected = "all"),

      # br() element to introduce extra vertical spacing ----
      br(),
      # Input: Slider for the number of observations to generate ----
       selectInput("Y",
                  "Response Variable: Y",
                  y.names),

       selectInput("X",
                  "Predictor Variable: X",
                  x.names,
                  selected = x.names[2]),
        ####
      hr(),
      ###
      sliderInput("newX", "New Value for Prediction:", 2.5, min = 0, max = 20, step = 0.1),
      ###
      ### The following code adds additional decorative and contact information ########
       hr(),
       HTML('<p><center><img src="https://github.com/pengdsci/sta553/blob/main/image/goldenRamLog
                           width="80" height="80"></center></p>'),
       HTML('<p style="font-family:Courier; color:Red; font-size: 20px;"><center>
```

```
                <font size =2> <a href="mailto:cpeng@wcupa.edu">
                <font color="purple">Report bugs to C. Peng </font></a> </font></cente
          ),      # close sidebarPanel
       ########################################################################

    # Main panel for displaying outputs ----
    mainPanel(
      # Output: Tabset w/ plot, summary, and table ----
      tabsetPanel(type = "tabs",
                  tabPanel("Scatter Plot", plotOutput("plot")),
                  tabPanel("Regression Coefficients", tableOutput("table")),
                  tabPanel("Diagnostics",plotOutput("diagnosis")),
                  tabPanel("Prediction", plotOutput("predPlt"))
              )
          )      # close mainPanel
      )          # close sideLayout
  )              # close fluidPage

###################################
###################################
##       Server function
###################################
###################################
server <- function(input, output) {
###################################
##     Some R functions
###################################
#### Subsetting data based on Species
  workDat = function(){
      if (input$species == "setosa") {
          workingData = iris0[which(iris0$Species == "setosa"),]
      } else if (input$species == "versicolor") {
          workingData = iris0[which(iris0$Species == "versicolor"),]
      } else if (input$species == "virginica") {
          workingData = iris0[which(iris0$Species == "virginica"),]
      } else {
          workingData = iris0
      }
   workingData
  }
  ######################################
  #######   Scatter Plots
  ######################################
  output$plot <- renderPlot({
    dataset = workDat()[,-1]   # define the working data set
    #####
```

```
   plot(dataset[,input$X], dataset[,input$Y],
        xlab = input$X,
        ylab = input$Y,
        main = paste("Relationship between", input$Y, "and", input$X)
       )
   ## adding a regression line to the plot
    abline(lm(dataset[,input$Y] ~ dataset[,input$X]),
           col = "blue",
           lwd = 2)
})

 ######################################
#######   Scatter Plots
######################################
output$table <- renderTable({
  br()
  br()
  dataset = workDat()[,-1]
  # define the working data set
  m0 = lm(dataset[,input$Y] ~ dataset[,input$X])
  #summary(m0)
  regcoef = data.frame(coef(summary(m0)))

  ##
  regcoef$Pvalue = regcoef[,names(regcoef)[4]]
  ###
  regcoef$Variable = c("Intercept", input$X)
  regcoef[,c(6, 1:3, 5)]

})
######################################
#######   Diagnostics
######################################
output$diagnosis <- renderPlot({
  dataset = workDat()[,-1]   # define the working data set
  #####
  m1=lm(dataset[,input$Y] ~ dataset[,input$X])
  par(mfrow=c(2,2))
  plot(m1)
})

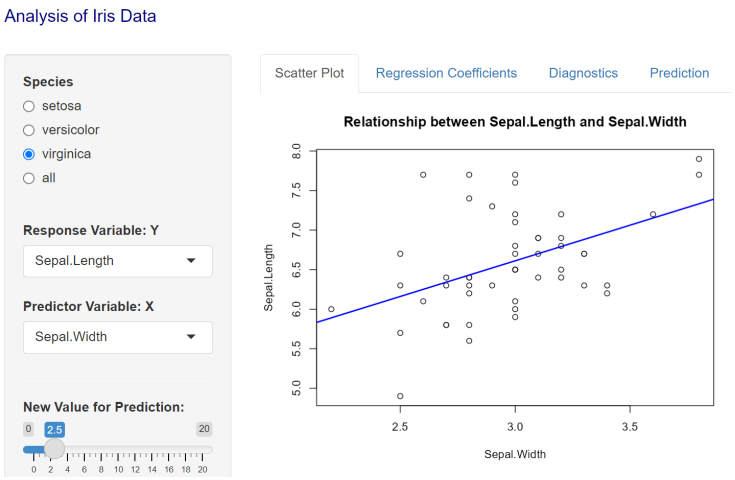
######################################
#######       Scatter Plots
######################################
output$predPlt <- renderPlot({
  dataset = workDat()[,-1]   # define the working data set
```

```
    ###
    m3 = lm(dataset[,input$Y] ~ dataset[,input$X])

    pred.y = coef(m3)[1] + coef(m3)[2]*input$newX
         #####
    plot(dataset[,input$X], dataset[,input$Y],
    #    xlab = input$X,
         ylab = input$Y,
         main = paste("Relationship between", input$Y, "and", input$X)
       )
        ## adding a regression line to the plot
     abline(m3,
            col = "red",
            lwd = 1,
            lty=2)
    points(input$newX, pred.y, pch = 19, col = "red", cex = 2)
  })
}

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

```
include_graphics("shiny/w11-shinyAnalysis02.png")
```



16.5 File Uploading

In the above iris analysis app, we preload the data set before writing `ui` and `server` functions. Sometimes, we may want to write an app to perform basic analysis for a given data set, in this case, we need to let the app upload the data

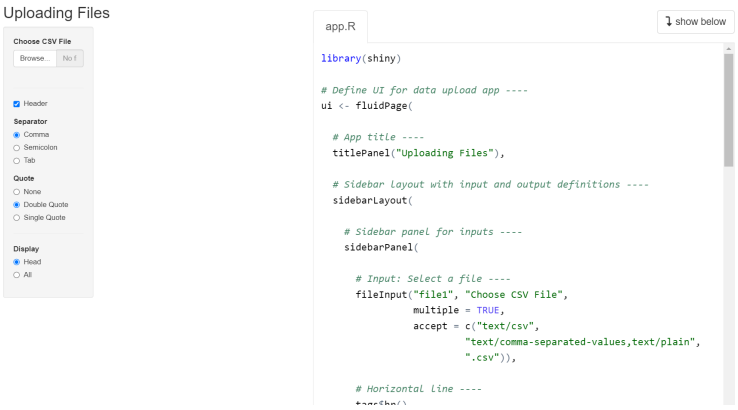
set for analysis. That means that we may want to use a file up-loading widget to load the file to R shiny apps.

The following code lists 11 shiny apps for different purposes. We can run the shiny app demo to get a sample code and then modify it for our own applications. For example, the following demo #9 has file uploading capability.

```
library(shiny)
source("https://raw.githubusercontent.com/pengdsci/sta553/main/shiny/shinyCodeExtractor.txt")
#runExample2("01_hello")        # a histogram
#runExample("02_text")         # tables and data frames
#runExample("03_reactivity")   # a reactive expression
#runExample("04_mpg")          # global variables
#runExample("05_sliders")      # slider bars
#runExample("06_tabsets")      # tabbed panels
#runExample("07_widgets")      # help text and submit buttons
#runExample("08_html")         # Shiny app built from HTML
runExample("09_upload")       # file upload wizard
#runExample("10_download")     # file download wizard
#runExample("11_timer")        # an automated timer
```

To run a specific demo, you only need to remove the hashtag and run the code chunk. For example, running demo 9, we get something like the following: app + code.

```
include_graphics("shiny/w11-shinyAnalysis03.png")
```



Chapter 17

Shiny Dashboard and Storyboard

We have two package options for building Shiny dashboards: flexdashboard and shinydashboard.

17.1 flexdashboard

Easy interactive dashboards for R that `use R Markdown to publish` a group of **related data visualizations** as a dashboard, support a wide variety of components including

- `htmlwidgets` - A framework for embedding JavaScript visualizations into R;
- `base`, `lattice`, and `grid` graphics;
- `tabular data`;
- `gauges` and `value boxes`;
- `text annotations`.

A flexdashboard is flexible and easy to specify row and column-based layouts with re-sizing to fill the browser. 'It is just a document that looks like a dashboard. However, it

- contains some specific widgets designed to work in a dashboard layout.
- offers storyboard layouts for presenting sequences of visualizations and related commentary
- can use Shiny to drive visualizations dynamically by specifying `runtime: shiny` in the YAML head.

- can only run interactive code client-side (in embedded JavaScript).

17.2 shinydashboard

An R package for creating dashboard-style layouts with Shiny that uses

- Bootstrap (a free open source front-end-framework develops faster, easier, responsive web pages) for layout;
- uses AdminLTE, which is a theme built on top of Bootstrap.

Unlike `flexdashboard`, **shinydashboard** has three structured components:

```
## app.R ##
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)

server <- function(input, output) { }

shinyApp(ui, server)
```

`shinydashboard` needs a server behind it to execute R code on user input. It can implement a dashboard layout that contains some specific widgets designed to work in a dashboard layout. It can also run interactive code either by processing server-side (in R) or client-side (in embedded JavaScript).

17.3 Three Web Application Terms

- **Responsive**: adapting layouts to a variety of screen and window sizes.
- **Interactive**: elements that react to your actions on the site.
- **Reactive**: automatically checking and updating data from the server without refreshing the website.

17.4 Using Flexdashboard

This note outlines the flexdashboard using Shiny.

17.4.1 Components

We can use flexdashboard to publish groups of `related data visualizations` as a dashboard. A flexdashboard can either be static (a standard web page) or dynamic (a Shiny interactive document). A wide variety of components can be included in flexdashboard layouts, including:

- Interactive data visualizations based on htmlwidgets.
- R graphical output including base, lattice, and grid graphics.
- Tabular data (with optional sorting, filtering, and paging).
- Value boxes for highlighting important summary data.
- Gauges for displaying values on a meter within a specified range.
- Text annotations of various kinds.

17.4.2 Basic Flexdashboar Layout

The layout design is relatively simpler than that of shinydashboard.

17.4.2.1 Single Column

Flexdashboards are divided into columns and rows, with output components delineated using `level 3 markdown headers (###)`. By default, dashboards are laid out within a single column, with charts stacked vertically within a column and sized to fill available browser height.

```
include_graphics("shiny/flexSingleColumn.png")
```

17.4.2.2 Multiple Column

To layout charts using multiple columns, we introduce a `level 2 markdown header (--------------)` for each column.

```
include_graphics("shiny/flexMultipleColumn.png")
```

17.4.2.3 Row Orientation

We can also choose to orient dashboards row-wise rather than column-wise by specifying the `orientation: rows` option.

```
include_graphics("shiny/flexRowOrientation.png")
```

```
---
title: "Row Orientation"
output:
  flexdashboard::flex_dashbo
    orientation: rows
---

Row
-----------------------------

### Chart 1

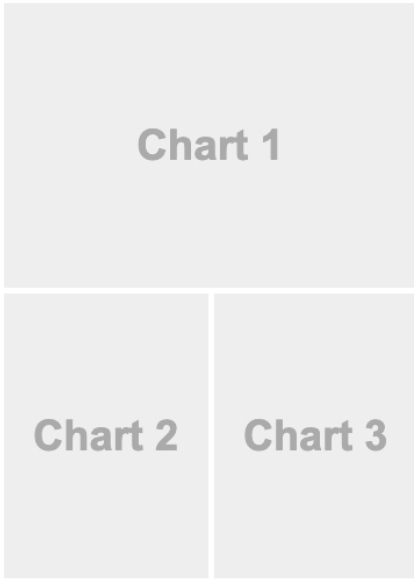
```{r}

```

Row

Chart 2

```{r}

```

Chart 3

```{r}

```
```



17.4.2.4 Grid Layout

This layout uses the default `vertical_scroll: fill` behavior. It might be preferable to allow the page to scroll (vertical_layout: scroll).

`orientation: rows` is used to ensure that the chart baselines line up horizontally.

```
include_graphics("shiny/flexGrid.png")
```

```
---
title: "Row Orientation"
output:
  flexdashboard::flex_dashboard:
    orientation: rows
---

Row
-----------------------------------

### Chart 1

```{r}
```

Chart 2

```{r}
```

Row

Chart 3

```{r}
```

Chart 4

```{r}
```
```

17.4.2.5 Tabset Column

This layout displays the right column as a set of two tabs. Tabs are especially useful when you have a large number of components to display and prefer not to require the user to scroll to access everything.

```
include_graphics("shiny/flexTabsetColumn.png")
```

```
---
title: "Tabset Column"
output: flexdashboard::flex_dashboa
---

Column
-----------------------------------

### Chart 1

```{r}
```

Column {.tabset}

Chart 2

```{r}
```

Chart 3

```{r}
```
```

17.4.2.6 Input Sidebar

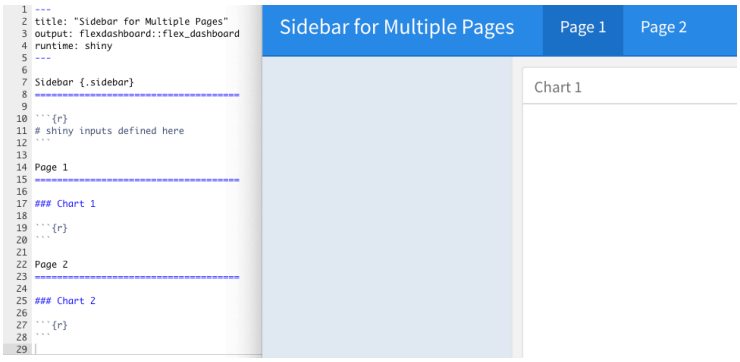
This layout demonstrates how to add a sidebar to a flexdashboard page (Shiny-based dashboards will often present user input controls in a sidebar). To include a sidebar you add the `.sidebar` class to a `level 2 header` (`------------------`):

```
include_graphics("shiny/flexInputSidebar.png")
```

17.4.2.7 Global Input Sidebar

If we have a layout that uses Multiple Pages you may want the sidebar to be global (i.e. present for all pages). To include a global sidebar you add the .sidebar class to a `level 1 header` (`======================`):

```
include_graphics("shiny/flexGlobalInputSidebar.png")
```



17.5 Storyboards

Storyboards are an alternative to the row and column-based layout schemes described above that are well suited to presenting a sequence of data visualizations and related commentary.

17.5.1 Storyboard Basics

To create a storyboard layout you do the following:

- Add the storyboard: true option to the dashboard.
- Include a set of `level 3 (###)` dashboard components. Each component will be allocated its own frame in the storyboard, with the section title used as the navigation caption.

The basics of creating storyboards are explained below. For a more complete example, see the HTML Widgets Showcase storyboard.

17.5.2 Storyboard Pages

We can layout one or more pages of a dashboard as storyboards and then use traditional row and column layouts on the other pages. To do this, you exclude the storyboard: true option and instead add the `{.storyboard}` class to pages you want to layout as storyboards.

17.5.3 Commentary

We may wish to add commentary alongside the frames in your storyboard. To do this you add a horizontal rule separator (***) after the main body of frame content. Content after the separator will appear in a commentary sidebar to the right.