A Project-based Approach to Introductory R and Python for Data Science

David Stokes | Mahmoud Harding

Table of contents

Introduction

> **Data Perspectives**
>
> What does it mean to be a data scientist?
> The answer to this question might depend on who you ask! Although data science applications may involve various disciplines (e.g., mathematics, statistics, computer science), skills (e.g., programming, visualizing), and roles (e.g., data analysis, software & tool development), certain processes transcend the many categorizations within this field. Furthermore, domain knowledge and data investigations of interest may come from humanitarian fields, social sciences, engineering, educational settings, and beyond. Thus, a data scientist may have multiple roles and focuses.

The Data Science Workflow

Rather than choosing a specific definition, you can think about data science from a process perspective. The following framework developed by Lee et al. (1), is an example of how your data science process may unfold for a given project.

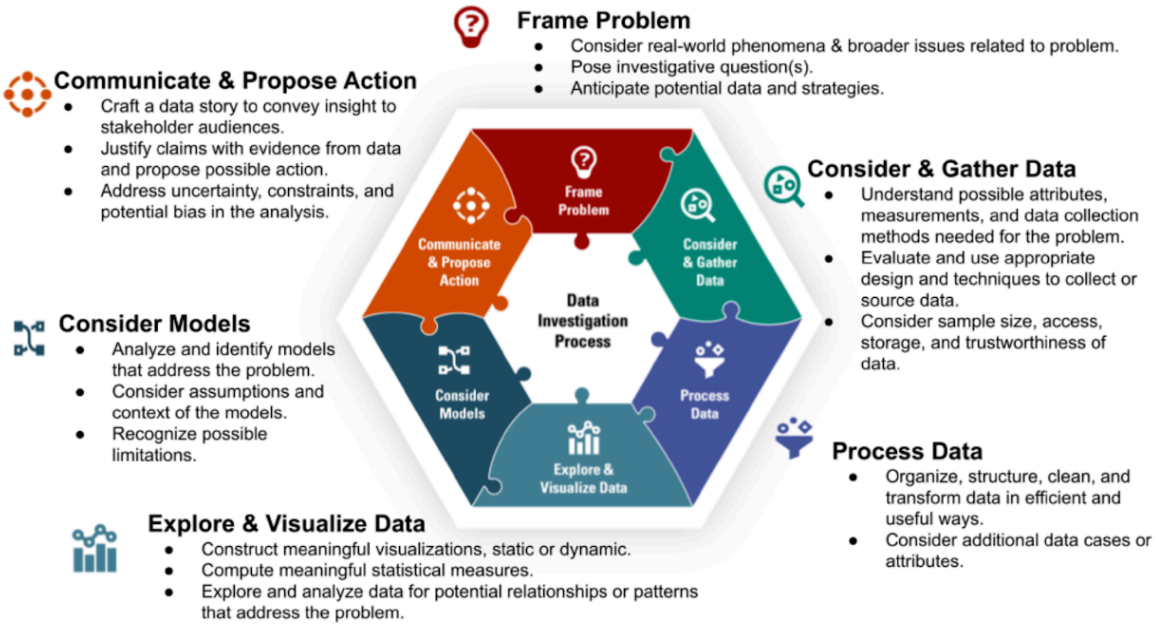


Figure 1: Data Science Workflow Example

Notice that the diagram represents the various data science processes in a non-linear way. Although the project you will pursue is structured through a sequence of milestones, it is common and often necessary to revisit prior stages of a data science workflow. For example, it may not be unusual to begin with data collection before framing a problem, and this may be driven by access to a particular data source. As explorations unfold, you might find the need to consider other angles of inquiry. On the other hand, questions of interest may drive the data collection process.

As we explain later, your course project will begin with a pursuit based on your interest. You will find a dataset that is of interest to you and this will be the starting point for subsequently framing a data investigation (i.e., you will consider and gather data as the initial workflow step).

Why R and Python?

So, why is this course focused on data science programming using R and Python?

For one reason why R and Python are important for data science, let's consider the latest Kaggle report State of Data Science and Machine Learning - Kaggle 2022 that presents results

from a survey of data science industry professionals from around the world. In this presentation, both Python and R are in the top three programming languages for data scientists. Furthermore, RStudio was used approximately by one in four data scientists and Jupyter Notebooks were used by over 9 out of 10 data scientists who responded to the survey.

Since the time of this study, RStudio has integrated Python into its platform. In particular quarto files can contain both R and Python code, as well as other languages, and can render reports and presentations in many different common files types, such as pdfs, word documents, and powerpoint presentations among others.

So, R and Python are not only powerful and useful languages to accomplish a given data science tasks, but they are also widely used by professional practitioners. This is for good reason - both R and Python are often found to be the intuitive tool of choice. Furthermore, individuals from all fields of study and professions desiring to derive reproducible insights from data should feel confident that the tools that are within their grasp are the industry standards.

Course outcomes

Our goal in creating this book is to provide you with a resource to guide your R and Python programming skill acquisition through a data-centric project-based approach. A fundamental idea behind this approach involves facilitating your agency in continued data science learning and up-skilling. So, you will not only learn to program in R and Python, but you will also be able to extend your learning in a self-directed capacity with a broader data science workflow in mind.

The purpose of this book is to provide material for you to increase your knowledge and comfort with programming in R and Python through relevant and applied data explorations and a real world data investigation experience. The R and Python for data science skills that you will acquire will be utilized to create a data science project.

Applied data science may involve goal-oriented steps that help to move projects towards a certain or desired outcome. In the next chapter we present project development steps from this applied focus. We also include big picture considerations that can transcend a given data science project. Of course, since we are learning programming for data science through an application, we need data! And before that, we need to setup our environment in such a way to help us organize our processes and files in service of our investigation of interest.

How will we use this book?

This book will be used as a course preview, review, and supplement to the material that will be covered in the class. Suggested readings will be provided, typically, in corresponding sequence

with this resource. This book will be used as a guide to the course project and overall course objectives in addition to related big picture data science considerations.

Setting up your environment

Installing RStudio

People interested in using R on their local devices can download the latest version of the language through the Comprehensive R Archive Network (CRAN) at The R Project for Statistical Computing Website.

The RStudio IDE

RStudio is an integrated development environment (IDE). As an IDE, RStudio provides a way to create & edit code, render presentations, connect with version control platforms and more.

Let's look at the default layout and the RStudio desktop features that will help us build our data science workflows as we learn the R programming language.

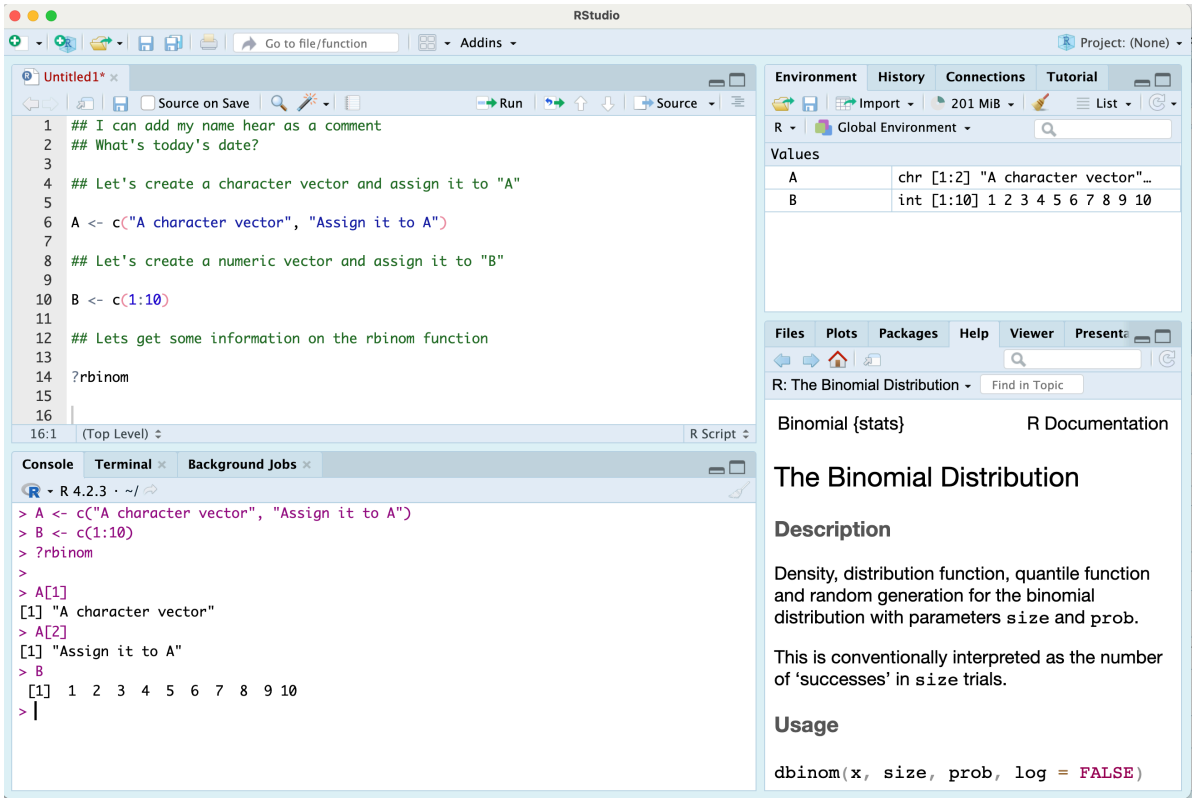


Figure 2: RStudio desktop

In the image above, there are four main windows. The upper left is the Source window where you can open various files to write, manage, and save your code. Notice that there is an "Untitled1" tab in this window, which is an unnamed R Script - a basic file type that allows you to write and manage your code. In an R script we might read in data and write code to gather information about the data, such as the size of the dataset and the column names. R uses certain colors to delineate different aspects of your code (e.g., distinguishing "comments" from executable code).

Beneath the Source window is the Console. The console displays the code that is run and the results of the code that is run. You can also type code directly into the console and even save different objects to memory from this window. However, the console does not save or retain your code once you have ended your R session, as where R Scripts and other file types managed through the Source window are designed for this purpose.

The window on the top right contains many tabs, including the Environment tab. The Environment tab provides information on the objects, or accessible elements, in your session's memory. Other tabs in the window may include the History tab, and perhaps a tab that interfaces with version control platforms (e.g., GitHub).

The final window, in the bottom right, also contains many tabs. The `Files` tab provides information about your current working directory and the files that exist within that working directory. The Plots tab can display features such as graphs generated from code (e.g., a scatterplot). In the image above, the information displayed in the bottom right window is from a call to the Help feature which has been done through the line of code that reads `?rbinom`. The contents appear under the corresponding Help tab.

Your RStudio view is customizable through the Pane Layout options. For the purposes of this course, the default display will work well. Our focus will mainly be on the Source/Editor and the Console. We may reference the Files tab, the Help feature, and display plots within the Plot window; and we may investigate information within the Environment. However, these useful features support our data science workflow which will mainly be captured within our editor.

In this section, we introduced a lot of vocabulary without detailed explanation. As you move through the course we expect that all of the information referenced here will be clear to you and encourage you to look back at this section, in particular, and the accompanying image as a reflection of your progress. You will be able to describe all that appears in the visual!

Setting up your course workspace (R projects)

Organization is a fundamental part of a project workflow. For our project, we can leverage the computer's folder system to efficiently and effectively store and access our course materials. Given that we are working in RStudio, we can do this by setting up an R project.

An R project is a directory that can contain all of your files related to an analysis, for example. This may be datasets, images, scripts, output and more. For our course, the R project we create will be used to keep track of your course materials and assignments. The R project space we will create will be your organizational space that points to what you are doing in this course.

When you set up your R project you will create a folder on your computer and new documents that you add to that folder will be a part of the files in your R project environment. Conversely, files that you create in your R project session will be added to the corresponding computer folder by default.

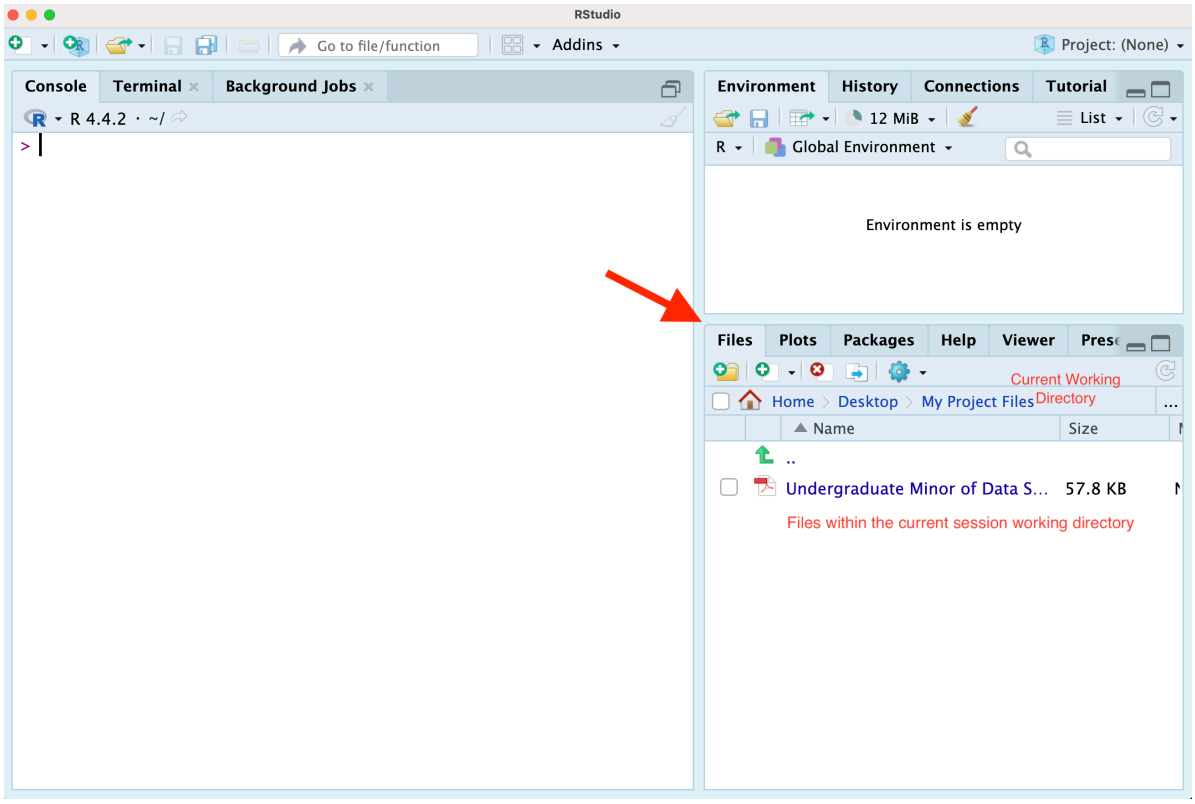


Figure 3: Files Window

The Files Window

Note that the *Files* tab/window is an interface to your computer files, so modifying files through this window correspondingly modifies the files on your computer, outside of R.

1

2 Project Part 1: Finding a dataset

Data Science investigations often involve deriving insights from existing or observational data, which differs from the process of designing an experiment, or collecting data for a specific use. Such an investigation may involve data that is publicly available and may also bound the scope of the project. In addition, relevance, interest, quality, background knowledge, and other aspects may be factors that influence the questions you want to consider and the data you would like to obtain.

We encourage you to choose data that is relevant and of interest to you. Given that there are many ways to go about this process and each person may have their own interests, we offer some general guidelines so that your data investigation is productive and allows you to apply the programming skills and framework that you will learn.

2.1 Dataset requirements

When choosing your dataset of interest you should think about dataset characteristics that may allow you to pose dynamic and insightful questions, and will allow you to explore potential trends, patterns, and relationships. These considerations include the number of observations, the number of variables and the types of the variables within the data.

Please note the following characteristics to guide your choice of a dataset for your project.

Data accesibility is a nice feature. If everyone is able to access the data you choose they will also be able to explore and investigate the data in a potentially similar or different way.

- Choose a publicly available dataset without restrictions or special permissions.

Your data should contain a sufficient amount of information, but also be manageable within the context of the project and course goals.

- Choose a dataset between 300 and 10,000 observations (rows).

In order to consider various and common data types and to be able to compare, group, visualize and perform other data moves and transformations:

- Choose a dataset with 10 to 30 variables (columns), that includes both quantitative and categorical data types. Your chosen data may also include character or text-based data types.

Also consider datasets with a time component, which may give you the option to investigate or observe temporal change.

2.1.1 Other dataset considerations

Other considerations for choosing your dataset may include the format of the available data. For example, if you have a time feature in your data, this can be formatted as a column for each different measure across time (wide format), or as multiple rows per unit, representing different times (long format). Both formats can be useful depending on your project step and the corresponding goal, but such datasets may require more procedural steps than others (e.g., data wrangling).

You should also attempt to identify potential hurdles such as missing data. In this case, you will want to understand why some data is missing and be able to identify references that explain this occurrence. Furthermore, you should ensure that your intended data use complies with ethical guidelines. As a guide on broad considerations, designed for agencies but also applicable to individuals, (e.g., transparency, purpose specification, and others), you can reference the Fair Information Practice Principles (2).

If you would like your class project to involve data from your research, make sure to adhere to the guidelines above and consult your instructor for additional requirements such as permission from your advisor and compliance with the related research guidelines.

Once you find a dataset, you will need to be able to work with it! So, let's explore some initial steps to get started using the R programming language through the RStudio platform.

2.2 Data Types & Structures

Voltron is a classic cartoon from the 1980s, with later versions made in the 2000s. In this cartoon various independent lions, each with unique attributes, merge together to form the Voltron Robot.

Data types and data structures are fundamental building blocks in any programming language. Data types are like categories of information that come with certain properties, such as the type of operations that can be performed on them. Data structures often involve collections of data types and allow for efficient organization, manipulation, and analysis of the various datatypes, or information contained in a given dataset.

So, data types are like the Voltron lions and can be combined into a data structure to create something new and functional, like the Voltron robot. As you learn more about data types and structures, consider an analogy that you can make to paint a picture of the relationship between these programming building blocks.

R provides a range of data types, including numerical, categorical, logical and others. The main data structures in R are vectors, matrices, lists, and dataframes (or tibbles).

2.2.1 Data Types

Numeric is the default quantitative data type in R. Let's look at some examples in the code below.

```
# (in R, variable names are flexible but are case sensitive)
num1 <- 5 # Here I'm storing the value 5 in an object called num1
num.2 <- pi
num_3 <- .07
Num4 <- 10e-3
```

Each of the numbers above (5, pi, .07, and 10e-3) are recognized as numeric data types. The integer, 5, and decimal value, .07, may be obvious numeric candidates, but what about `pi` and `10e-3`?

R has certain "reserved" objects that are built into the language. `pi` is an example and is a reserved name for value 3.141593... Likewise, 10e-3 is how R represents scientific notation. This expression is equivalent to $10 * 10^{-3}$. In your future R use, you might encounter certain model output that reports large or small values in this scientific notation format.

There are a few additional very important components of the code above. Notice that each of the numeric data types have been associated with various names through the $\leftarrow$ symbol. The arrow pointing to the left is an `assignment statement`, which serves to store each of the numeric data types into the corresponding name. Our numbers are now objects that we can reference by name!

```
num1 # this is the name I gave to the value 5
```

```
[1] 5
```

```
Num4 # this should be 10e-3 or 0.01
```

```
[1] 0.01
```

From the executed code, you can see that calling the variables by name (or referencing them) results in their display. In general, calling an object by its name (running code that references stored information) displays its contents. Now that we know how to store and retrieve information (such as numeric data types), let's move on to the final component of the code we've observed so far.

Notice in both code snippets there are lines with the # symbol followed by descriptions. These are called `code comments` and are an essential part of the reproducible coding process. Within a code snippet (or code chunk) anything following the # symbol on a particular line does not execute. R treats all things that follow the # as comments, or notes, that aren't part of the code that should be run.

In the first code chunk we made an explanatory comment to emphasize that R object names can be specified in different ways, but are case sensitive. Other restrictions on R object names can be found here: R's Naming Rules. In the second code chunk, we made comments following the executable code to specify what we should observe in the code output.

> **Looking Forward**
>
> We will emphasize the importance of code comments in the data science programming workflow throughout this book.

Numeric data types are not just for storing numeric data. We can also perform operations on numeric data, such as addition, subtraction, exponentiation, and others.

```
1 + 3
```

```
[1] 4
```

```
num1*Num4 # 5*0.01
```

```
[1] 0.05
```

```
num1^2
```

```
[1] 25
```

Character data includes non numeric information, such as strings or text data, and is specified in quotes.

Let's look at some examples.

```
charA <- "Whaz up,"
charB <- "whazz upp,"
charC <- "whazzz uppp!"


## In R, you can pass objects to a function.
```

```
# Here is an example using the paste() function,
# with charA, charB, and charC

paste(charA, charB, charC)
```

```
[1] "Whaz up, whazz upp, whazzz uppp!"
```

Note that numbers, that would be numeric by default, can be specified as character by using quotes.

```
A <- "7"
B <- "200"

A
```

```
[1] "7"
```

Logical data types in R represent Boolean values that can be either true or false. Logicals are very useful for checking conditions and are used to construct if-then statements and subsetting data, for example.

```
1 + 1 == 2
```

```
[1] TRUE
```

```
8 > 9
```

```
[1] FALSE
```

Now that we've learned about R data types, and before we learn about R data structures, let's pivot to a different topic that will allow us to bring tools into our R environment to facilitate both or data science learning and programming applications.

2.3 Packages & Libraries

R packages contain supplements to all of the wonderful and useful default tools that exist in base R. An R package is typically built for a particular purpose and might contain customized functions, data sources, & related documentation. An example is the ggplot2 R package that is built for data visualization. This package includes data visualization functions built on "geometries" and useful documentation for each function is included in the package for easy reference.

R packages can be installed right through the console by using the `install.packages()` function, or more conveniently through the Install feature found within the Packages pane.

Once a package is installed it is stored in a location that we call a library. The package contents can be loaded into your current R session from this storage location, sort of like checking out a book from, well, a library. You can "check out" a package and use its contents in your current R session by referencing the package name inside the `library()` function.

```
#  Example syntax of calling the library function
#  to load the (already installed) ggplot2 package
#  into our current R session

library(ggplot2)
```

> **Looking Ahead**
>
> In chapter 7 will see how to use the ggplot2 package to visualize our data.

2.4 Importing data

Recall, the file structure you set up for your R project. The folder associated with your project holds your course files. When your R project is open in your current RStudio session the folder contents can be seen and accessed through the Files pane. This means that your current "working directory" is your R project folder.

You can also import other files into your R session that are not necessarily in your R project by referencing them through their file path - their location on your computer. So, a dataset of a particular file type can be imported using this method whether or not it's in your current working directory.

For a .csv file the base R import function is `read.csv()`. You would add the dataset filepath of interest, in quotes, inside of this function to import the data.

Let's take a look at an example using the college dataset from the ISL with R, 2nd Edition Resources. Since we learned about packages and libraries, for this example we can load the `readr` package from our library and use a function from this resource.

```
## Since the dataset is in my current working
#  directory, I only need to specify the name
#  as the file path
library(readr)
college <- read_csv("College.csv")
```

If you're thinking, "but wait! We're using RStudio. There must be another way," then you'd be right.

As we mentioned before, within the Environment tab there is an "Import Dataset" feature that will allow us to search our computers file system for the dataset we wish to import, and this will generate the syntax for us with the function and file path built in.

2.5 Data Structures

2.5.1 Vectors

A vector in R is a fundamental data structure that represents a sequence of elements, all of which must be of the same type. For example, every element in a character vector is a character data type, every element in a numeric vector is a numeric data type, and so on. Vectors form the foundation for more complex data structures like matrices and dataframes (or tibbles).

Vectors are one-dimensional data structures where each element corresponds to an index value according to its position. Note that in R, the index values start at 1. This means that the element in the first position of a vector corresponds to index 1 and the element in the second position corresponds to index 2, and so on. This indexing allows for referencing vector elements or groups of elements by their location, which can be leveraged to facilitate data processing and manipulation.

Let's explore a few vectors by first creating one ourselves.

```
# the c() function "combines" elements (separated by commas) into a vector.
even_vec <- c(2,4,6,8,10)

odd_vec <- even_vec - 1

even_vec # display the elements of even_vec
```

```
[1]  2  4  6  8 10
```

```
odd_vec # display the elements of odd_vec
```

```
[1] 1 3 5 7 9
```

Note that in the above code, we "combined" our data values into a vector by using the c() function which is designed for this purpose. Wrapping our vector values in the c() function tells R to treat the group of values as a vector data structure.

> **i** Looking Ahead: Vectorization
>
> In the code above we created a vector which contained the first 5 even integers. By subtracting 1 from this vector, we were able to create a new object containing the first 5 odd integers. In other words, through the code above the subtraction operation was applied to each element of even_vec to create odd_vec. This process of applying an operation to each element of a vector without having to iterate through each one in a loop is known as `vectorization`.

So, now that we've created some vectors out of thin air, let's return to the dataset we loaded into our environment to find a vector existing among its peers (other vectors!). But first, let's take a look at some of the observations and variables in the college dataset.

Institution	Private	Apps	Accept	Enroll
Abilene Christian University	Yes	1660	1232	721
Adelphi University	Yes	2186	1924	512
Adrian College	Yes	1428	1097	336
Agnes Scott College	Yes	417	349	137
Alaska Pacific University	Yes	193	146	55

The display shows the first 5 rows and columns of the college dataset. We can see that the variable types appear to change across the rows and remain similar within each column. The columns are our vectors!

2.5.2 Dataframes

A dataframe in R is a two-dimensional data structure. Dataframes can store different types of data in each column, such as numeric, character, or logical data types all in one group. We can think of a dataframe as a collection of vectors. However, for typical datasets that may be represented as a dataframe structure not only are the elements of the columns related (e.g., by representing the same data type and variable) but also the rows, which typically represent

an observation of many values (or measurements, or data types) on a single unit or individual. This explains what we observed in the college dataset output.

Each element within a dataframe is accessed using a pair of indices, one for the row and one for the column. This indexing system allows for organized data storage and facilitates access and manipulation of entire rows, columns, or specific elements within them.

Let's print out the first variable of the first observation.

```
# displays element (1,1) of the dataset.
college[1,1]
```

```
# A tibble: 1 x 1
  Institution
  <chr>
1 Abilene Christian University
```

Since we read in the college data with the `read_csv()` package instead of the `read.csv()` package in base R, the default data structure is a tibble. Tibbles are essentially the same as dataframes.

Finally for our dataframes and tibbles, lets observe code that will demonstrate how indices can be used to display the default output from referencing the first 5 rows and columns of our imported dataset.

```
# gets rows 1 to 5, and gets columns 1 to 5
college[1:5,1:5]
```

```
# A tibble: 5 x 5
  Institution                 Private  Apps Accept Enroll
  <chr>                       <chr>   <dbl>  <dbl>  <dbl>
1 Abilene Christian University Yes      1660   1232    721
2 Adelphi University          Yes      2186   1924    512
3 Adrian College              Yes      1428   1097    336
4 Agnes Scott College         Yes       417    349    137
5 Alaska Pacific University   Yes       193    146     55
```

Notice that the data structure is listed as a tibble, and along with the data we also see each column data type between the column names and the data values.

2.6 A Few More Data Structures

2.6.1 Matrices (aka Matrixes)

A matrix in R is also a two-dimensional data structure where elements are arranged in rows and columns. Matrices are not as flexible as dataframes. One restriction on matrices is that all elements in a matrix must be of the same type. Due to this restriction, matrices are like two dimensional vectors.

As with a dataframe, each element in the matrix is accessed using a pair of indices, one for the row and one for the column. This structured indexing allows for organized storage of data and supports operations that involve manipulating entire rows, columns, or specific elements within the matrix.

2.6.2 Lists

Lists are another type of important data structure in R. Lists are vary flexible and can have as elements entire data structures of various types. Lists can be useful in building functions and other complex tasks that require a flexible data structure. As useful as lists can be, for this course the data structure of choice is the dataframe (or tibble)!

3 Project Part 2: Creating a data dictionary

Data Perspectives

Data is information that reflects a particular process or observation. This information may contain imprecisions due to errors from measurement limitations, collection mishaps, and other reasons. Variability is an inherent characteristic of data and is seen in natural fluctuations across the different units on which data is collected, or through differences from sample to sample. Both error and variability in data are important to characterize throughout a data investigation and may impact decisions such as the need to collect more data and the strength attached to a particular communication, claim, or trend. In recognition of the presence or variability and error in data, even disregarding the data generation process, we should recognize *data as information, not truth.* When we add in the particular data generation process, we must recognize the potential for some biases and decisions that include certain aspects and omit others. We should recognize data has *degrees of inclusion and exclusion.*

3.1 Basics in R

3.1.1 Comments

Suppose you need to work on code for a project as a member of a team. Or, perhaps you are working on code for multiple projects and need to switch between these projects at different points of the process.

- What do you imagine are some of the challenges you might you face when working over an extended period of time?

Or, what if some time has passed since you last worked on a data science project involving coding and you need to get back to it. Upon resuming your work, you might ask: "Where did I leave off with my code, and what was the purpose of this line at this point?". In sharing your code, maybe one of your collaborators is wondering why you implemented a particular data step, function, or method at a given line.

- Would you want to start at the beginning of a file and retrace the logic up to the stopping point in order to understand what should be next?

This is possible and can be a beneficial review of your process, but as your project and code develop over time, this approach would become more and more time consuming. Even with a line by line review it's possible that you wouldn't recall all of the reasons for your coding choices and the corresponding purposes.

A better method to facilitate effective work on coding projects over time or with collaborators, is to use clear and informative `code comments`. Comments are an essential part of the coding process, especially for reproducibility. In addition to tracking clear and informative information about your coding process, comments can be incorporated directly into the lines of code to benefit your understanding of your data science coding process, as well as that of others who will see or work with your code.

As you saw in the previous chapter, in R, code comments begin with the `#` character.

```
# The text following the hashtag on this line is part of a comment
# Here is another comment
```

In the midst of code, comments can be added on a different line from the code you'd like to run, or on the same line of the code you'd like to run as long as the hashtag follows the executable code. Recall, everything following the hashtag is treated like a comment.

```
# I'm creating a vector of important numbers called Important_Numbers.

Important_Numbers <- c("177","243","388") # characters
# These are product identifiers
```

A scenario (comments)

Imagine your collaborator gave you code to modify in order to run some basic calculations and you were tasked with summarizing the characteristics of the quantitative values - including statistical measures such as means. Since you only read this book up to chapter 1, you decided to just input the variables into the mean() function without checking any metadata, such as variable types. You proceed to run the code shown below on the `Important_Numbers` variable (that is part of a larger dataset).

```
mean(Important_Numbers)
```

```
Warning in mean.default(Important_Numbers): argument is not numeric or logical:
returning NA
```

```
[1] NA
```

To your surprise, the code throws an error!

Luckily, your collaborator made a note about this variable through a comment. Upon looking through the code, and without having to recognize the quoted numbers (that were embedded in a much larger code sequence), you are able to easily see that the variable `Important_Numbers` was composed of character values, intentionally, to reflect their designation as product identifiers. Since you know you (and R) cannot compute the mean of a character vector you were able to identify this as the source of the error! Furthermore, from the helpful comment, you understand the reason why these numbers are being treated as character values. Upon reflection, you realize that your thoughtful collaborator helped you facilitate the debugging process and also provided you with additional information about the variable through their informative code commenting practices. You remove `Important_Numbers` from the calculations, check the metadata of the other variables, and proceed with the calculations on the appropriate information, error free.

We can also imagine a scenario where some lines of code may be in an unfamiliar syntax (R has so many ways to perform the same task), and your best way of gaining insight into the purpose of the code may likely be through having access to informative code comments. Also, keep in mind that R errors are not always as specific as the output we see above and may appear apart from a particular line of code.

Now that we know how to make comments within code and can see how they can be useful, we will continue to use them as an important and essential part of our data science coding, collaboration, and reproducibility workflow.

3.1.2 Variables

In a typical dataset, `variables` are often measurements or characteristics of `observations`. Observations are some unit on which measurements are taken. For example, "height" (measured in feet), and "primary language" might be recorded for people who live in Wake County, North Carolina. In this case, we would call "height" and "primary language" variables, and the different people who live in Wake County whose heights and primary languages were recorded would be the observational units that make up the observations in the dataset. Variables are usually represented by columns within a dataset, and each observation (represented by the rows) has an associated set of variable measurements, or information.

By contrast, in R a variable is an `object` that has some value or values associated with it. In the R programming context, the use of the term variable is more general. In the programming context case, for example, a variable may contain an entire dataset (and the associated rows and columns). The different uses of this term are important to note since what is meant by the term "variable" may depend on the context.

> **Variables (and terminology) vary!**
>
> We may use the term variable to refer to an object that can be referenced and has general information associated with it (the programming context), or as a column in a dataset containing information about observations (the data science context).

3.1.3 Assignment Statements

Consider the general programming use of the term variable: How can we create a variable and how can we reference it? As you've already seen, the answer to this question is to store your created object into a reference via an `assignment statement`.

In R, you can use two different symbols to assign information to a variable. These are `<-` and `=`. It is also possible to make assignment statements from left to right using `->`, but R programmers tend to use `<-`, specifically. This standard left pointing arrow is what we will use as an assignment statement throughout this book.

For a glimpse into the history of the R assignment statement see the resource here: Why do we use arrow as an assignment operator?

3.2 Data Dictionaries

We mentioned the benefit of code comments in collaborative processes. Even before the data-focused coding process begins, a more important workflow step involves creating a `data dictionary`. Data dictionaries are to datasets as code comments are to code.

Data sources may come with varying levels of background information. We can refer to this information about our data as `metadata`. This background information on a dataset is essential to the steps of an associated investigation of the data. For example, as you progress towards later stages of the course project where you will communicate insights from data, consider how you might go about this communication without knowing the data source (or generating mechanism), or the information on variable ranges, categories, or other useful descriptors. If you didn't know this information, how could you hope to communicate it to others?

For the course project and the dataset you choose, think of yourself as the data curator. Part of your responsibility in assuming this role is to create a data dictionary. The objective of creating the data dictionary is for you to be able to hand it and the dataset it describes to someone else and that person would have all the information they need to understand where the data is from, what the data contains, and what the data is about, including context, possible uses, limitations and more.

Connection to the Data Science Workflow

Recall the reference to the data science workflow in the introduction. Considering the categories of this particular framework, the development of the data dictionary could be a part of many of the stages. However, more importantly than situating the data dictionary in a particular workflow stage is recognizing how important the information within the data dictionary is to all stages. Although not all data dictionaries are created equal, the ones that we will create are designed to inform and impact the entire data science workflow!

3.2.1 Data Dictionary Requirements

Let's consider some essential components of the data dictionary that you will create for your course project.

- **Background Information**
  - Describe what your dataset is about (e.g., what are the observations and what's being measured and/or characterized)
  - Identify the source of your data.
- **General Metadata**
  - List the dimensions of the dataset.
  - Are there missing values? If so, explain why there are missing values and how the missing values are coded in the dataset.
- **Variable Characteristics**
  - List and describe the variables in your dataset.
  - For categorical variables, list the levels and for quantitative variables give the units of measure and the data ranges.
  - Describe whether your data contains unique identifiers and other variable types such as character or text data and what these variables represent.
- **Other considerations**:
  - Describe how the data was collected or generated.
  - Describe by whom the data was collected and the purpose and intended use case (e.g., what questions were asked and what answers were sought) for the data that was collected.
  - What are the limitations of the data, including information that may be missing from the data dictionary.
  - Consider what additional information may be useful for others to know about the data and include this when appropriate.

Given that you are not designing the data collection process, and the data you access may have prior documentation shortcomings, all of the criteria above may not be knowable or accessible. This would be an inherent limitation but still can be described in your data dictionary. In such cases you should recognize the incomplete information and think about how you could avoid such omissions in your future data curation roles.

3.3 Describing Your Data

R has many functions built in to the base version of the language. You have already seen examples, such as `read.csv()` and `paste()`, and likely have a good idea about what R functions do. Still, it can be useful to recognize that R functions tend to work and look like those you may have seen in something like a math class. Like a mathematical function, an R function takes in parameters inside parentheses (e.g., a file path) and returns specific results (e.g., a dataframe). In regard to describing our data, we can leverage R functions to make the process both more efficient and insightful.

> **Looking Ahead - R Functions**
>
> Beyond the base R functions, supplemental functions often exist in R packages that can be installed. However, there is sometimes a need for `user-defined functions` which we can customize and build, ourselves, if and when needed. We will see some examples of user-defined functions in subsequent chapters.

Next, let's consider an example of how we can describe our data and create & verify our data dictionary.

3.3.1 Example: Data Steps for Data Dictionaries

The Marine6 dataset contains 815 observations of six marine species across 10 North American Regions from the years 1970 to 2020. Measures include location information (longitude, latitude, and depth) for certain species observed within a particular region, within a particular year. This dataset represents a subset of the data seen at the EPA - Climate Change Indicators website.

The paragraph above is a minimal example of the background information that should be a part of your data dictionary. To expand upon the background description and address other components of the data dictionary criteria, we will read in the Marine6 dataset and use some functions to access general metadata and information on the variables within.

```
# Loading in the data, using the readr package
library(readr)
Marine6 <- read_csv("Marine6.csv")
```

As noted above, we want to include the dimensions of the data in our data dictionary. Even if the information is given with the data, or is perhaps known from a process like viewing the data in a spreadsheet, we can use the `dim()` function to quickly figure this out and verify that the data that was read in matches the expected size. We noted above in the description that there should be 815 observations (rows) in the data. Let's confirm.

```
# check the dimensions of the dataset
dim(Marine6)
```

```
[1] 815  10
```

Confirmed! The dim() function allows us to quickly and easily see that we have 815 rows and 10 columns in our data. The 815 rows were expected, but only three measures (latitude, longitude, and depth) were mentioned in the description. So, what's up with 10 columns, then?

Since we're using R - that's right you guessed it - there are many ways to inquire about this. The code below demonstrates one way that is particularly helpful.

```
# What are the variables in the data named?
colnames(Marine6)
```

```
 [1] "Year"                      "Region"
 [3] "Species"                   "Latitude"
 [5] "Latitude Standard Error"   "Longitude"
 [7] "Longitude Standard Error"  "Depth"
 [9] "Depth Standard Error"      "Common Name"
```

Using the `colnames()` function, from the output it's now clear that six of the 10 measures were described in the background information (the three location measures, Year, Region, and Species). We can also see that along with the location measures there are associated standard errors for each, and one additional variable called "Common Name". But let's dive deeper to find out even more about the variables in the Marine6 dataset.

```
# Let's get default summaries for the variable
summary(Marine6)
```

```
     Year         Region            Species             Latitude
 Min.   :1970   Length:815         Length:815         Min.   :26.51
 1st Qu.:1991   Class :character   Class :character   1st Qu.:32.40
 Median :2000   Mode  :character   Mode  :character   Median :37.92
 Mean   :1999                                         Mean   :37.54
 3rd Qu.:2010                                         3rd Qu.:42.23
 Max.   :2020                                         Max.   :57.50
                                                      NA's   :5
 Latitude Standard Error   Longitude       Longitude Standard Error
 Min.   :0.0000          Min.   :-159.98   Min.   :0.0000
 1st Qu.:0.1770          1st Qu.: -79.90   1st Qu.:0.2362
 Median :0.2267          Median : -75.05   Median :0.3298
 Mean   :0.2942          Mean   : -76.79   Mean   :0.3859
 3rd Qu.:0.3342          3rd Qu.: -68.63   3rd Qu.:0.4475
 Max.   :4.4575          Max.   : -60.24   Max.   :1.7813
 NA's   :34              NA's   :5         NA's   :34
     Depth        Depth Standard Error Common Name
 Min.   :  7.48   Min.   : 0.0000      Length:815
 1st Qu.:  8.43   1st Qu.: 0.1287      Class :character
 Median : 52.98   Median : 4.4436      Mode  :character
 Mean   : 67.22   Mean   : 5.7448
 3rd Qu.:119.63   3rd Qu.: 9.2985
 Max.   :255.06   Max.   :63.2701
 NA's   :53       NA's   :73
```

There is a lot to take in about the output here, so let's examine what information the `summary()` function is giving us. It appears that for the variables that we might assume to be numeric, the summary function returns a "5 number summary" which includes the quartiles, a mean value and something else that says "NA's". This is the standard output of the summary function for variables that are of type numeric. The "NA's" in the output represents a count of missing values for the respective variable.

Notice that for the other non-numeric variables we just get a length, and then the output "character" for the class and mode. This is the default summary function output for variables of type character, and it's not very useful in learning about the contents of the variable. However, we can fix that!

The variables "Region", "Species", and "Common Name" contain classifications or categories of information. We can treat these types of varibles as `factors`. As an example of a factor variable, let's consider a variable called "Color". Let's say that in the dataset that included this color variable, each observation could be classified as "red", "blue", or "green", meaning that each observation that has a color value recorded would have a corresponding label within the ["red","blue","green"] set of colors. If we designated "Color" as a factor, then the ["red","blue","green"] set would be what we call the `levels` of this factor variable.

Let's see what happens in the summary if we consider our character values as factors. In the code below, we will use the `$` operator to extract the "Region" variable from the Marine6 data and use a nested function statement to input this variable into `summary()` as a factor.

```
# getting the summary of the variable
# "Region" in the Marine6 dataset
# treated as a factor

summary(as.factor(Marine6$Region))
```

```
            Gulf of Alaska             Gulf of Mexico
                        12                         74
Gulf of St. Lawrence South           Maritimes Summer
                        48                         99
         Northeast US Fall        Northeast US Spring
                       164                        132
         Southeast US Fall        Southeast US Spring
                        87                         92
       Southeast US Summer          West Coast Annual
                        93                         14
```

Aha! This output is much more useful. In particular, we can now see the levels of the "Region" variable and see that there are 10 different regions represented in the data, as described in the background information. The numbers below the region levels are the counts. We can see that there are 12 measurements for the "Gulf of Alaska" region among the other regions in the data.

Before we move on, let's revisit the line of code that led to this output. We used two functions and one operator simultaneously. Recall, our mathematical functions analogy. Just as with math functions, you can use function nesting or composition with R functions. The `$` operator is the `extract` operator. So, in this code we extracted the Region variable from the data and nested the factor conversion function, `as.factor()`, within the `summary()` function to create the output of interest. What we observed in the output is what the summary function returns by default when the input is of type factor.

Since the factor version of our character variables returns useful summary information, let's go ahead and update the data so that this is displayed for each of the variables that we would like to consider this way.

```
# convert characters to factors (one.. at.. a.. time)
Marine6$Region <- as.factor(Marine6$Region)
Marine6$Species <- as.factor(Marine6$Species)
Marine6$`Common Name` <- as.factor(Marine6$`Common Name`)
```

Let's see what the new summary gives us.

```
# summaries of our quantitative and categorical variables
summary(Marine6)
```

```
      Year                        Region                      Species
 Min.   :1970   Northeast US Fall  :164   Centropristis striata:180
 1st Qu.:1991   Northeast US Spring:132   Homarus americanus   :187
 Median :2000   Maritimes Summer   : 99   Larimus fasciatus    :158
 Mean   :1999   Southeast US Summer: 93   Orthasterias koehleri: 26
 3rd Qu.:2010   Southeast US Spring: 92   Sphyraena guachancho :126
 Max.   :2020   Southeast US Fall  : 87   Urophycis chuss      :138
                (Other)            :148
    Latitude     Latitude Standard Error   Longitude
 Min.   :26.51   Min.   :0.0000          Min.   :-159.98
 1st Qu.:32.40   1st Qu.:0.1770          1st Qu.: -79.90
 Median :37.92   Median :0.2267          Median : -75.05
 Mean   :37.54   Mean   :0.2942          Mean   : -76.79
 3rd Qu.:42.23   3rd Qu.:0.3342          3rd Qu.: -68.63
 Max.   :57.50   Max.   :4.4575          Max.   : -60.24
 NA's   :5       NA's   :34              NA's   :5
 Longitude Standard Error     Depth        Depth Standard Error
 Min.   :0.0000           Min.   :  7.48   Min.   : 0.0000
 1st Qu.:0.2362           1st Qu.:  8.43   1st Qu.: 0.1287
 Median :0.3298           Median : 52.98   Median : 4.4436
 Mean   :0.3859           Mean   : 67.22   Mean   : 5.7448
 3rd Qu.:0.4475           3rd Qu.:119.63   3rd Qu.: 9.2985
 Max.   :1.7813           Max.   :255.06   Max.   :63.2701
 NA's   :34               NA's   :53       NA's   :73
              Common Name
 American lobster    :187
 Banded drum         :158
 Black sea bass      :180
 Guachanche barracuda:126
 Rainbow star        : 26
 Red hake            :138
```

Now we can successfully talk about the information that is within the categorical variables in the data. You already knew "there are so many ways to do a thing in R," so this is only one (perhaps inefficient) way to change the Marine6 character variables to type factor. As another method, we could have initially modified the way the data was read in to automatically convert

character variables to factors, or we could have even considered a fancier way (e.g., those user-defined functions we'll talk about soon). However, when there is a manageable amount of things to modify, sometimes the quickest way is a line by line method as shown.

Just to be clear, in the code above, prior to the latest summary, we overwrote the character variables with their factor versions and re-saved them into themselves. Here is the translation for the line `Marine6$Region <- as.factor(Marine6$Region)`: Convert the variable "Region" in the Marine6 dataset into a factor and store it into the variable Region in the Marine6 dataset. There are many reasons why certain ways of doing may be better than others, but one takeaway for the time being is that R code runs in sequence and this has implications for code that overwrites certain objects.

> **Looking Ahead**
>
> The tidyverse contains the `dplyr` package with data processing functions that can be leveraged to write efficient code and minimize overwriting, storage, and other coding inefficiencies that may result from solely using base R.

Another useful base R data descriptive-focused function is the `str()` function which includes the data dimensions, the variable types, example values and more. Also, recall the dataset output referenced in project step one. In the same way as viewing the first few rows of data through index references, the `head()` function, with a dataset as input, can be used to help you "see what the data looks like". Adding a few example rows of data to your data dictionary can also be helpful and the head function can provide this information.

You can now imagine ways to go about extracting and verifying the information for your data dictionary, which will often be a combination of researching and data descriptive-focused programming. As in the case where some information is not knowable from prior documentation, there are some components of the data dictionary criteria that may look different depending on the dataset you find. For example, it is not reasonable or necessarily helpful to list out every single level of a factor if there are too many (tens of levels or more). For instance, in the Marine6 dataset, instead of listing out every level of the "Region" variable, we might just describe this as "a variable with 10 levels representing North American bodies of water during different seasons." On the other hand, we might enumerate all six levels of the "Common Name" variable.

Note that in creating your data dictionary, the format is important. When summarizing your variables include descriptions and explanation and not just output from the summary function. Your data dictionary should tell a story about the data and is not just a replication of data summaries. As you progress in your data investigation, you can always return to your data dictionary for iterative updates, corrections, or improvements.

4 Project Part 3.1: Diving into Data Exploration - R

Now that we are familiar with our dataset "metadata" let's consider a more thorough exploratory process informed by the background knowledge encoded into the data dictionary. To do this, we will access some data science programming tools and concepts that will provide us with the means to dive and delve into our data.

4.1 Data Moves

To begin exploring your data in more detail, you will likely need to modify your dataset in some way. This may involve removing variables or observations, creating groupings or categories, performing operations on certain values, and other transformations of the data. For example, if you want to understand or visualize a particular trend in your data, you may need to index your observations in a particular order. This might be a necessary step if your data has a temporal component and you want to visualize a trend or phenomenon with respect to time.

Erickson et al. (3), refer to such data transformations as "data moves" and understanding what these are can facilitate your data science explorations and investigations regardless of your programming language or platform of choice. We will consider data moves as a fundamental part of our programming and as a guide to facilitating the many aspects of our data exploration and investigations process, from data cleaning to data visualization and beyond. Our version of data moves will be connected to the dataframe object structure that is part of both R and Python. These concepts are the SCUBA gear that will allow us to wade through the depths of our data with purpose and confidence.

4.2 Data Moves in Base R

Typical dataset modifications involve data moves such as filtering, subsetting, grouping, and merging. In chapter 3, we looked at the Marine6 dataset. This 815x10 dimension dataset was created through a series of data moves from a larger dataset consisting of 48,237 rows and nine columns. Creating this dataset of interest involved data exploration and data moves such as filtering, subsetting, grouping, and creating a new variables.

4.2.1 Filtering

The `filtering` data move is what we will use to reduce or examine a dataset based on certain `row criteria`. Since we have information about the range of years in the dataset, from our data dictionary, we might wonder "How many observations were recorded from the beginning, in 1970?".

```
# read in the Marine6 dataset using base R
# convert characters to factors upon reading in
Marine6 <- read.csv("Marine6.csv", stringsAsFactors = TRUE )

# find the minimum value for the variable "Year"
# aka, the earliest recordings

min(Marine6$Year)
```

```
[1] 1970
```

In the code above, we read in the Marine6 data and used the `min()` function on the "Year" variable to get the earliest date. The output confirms that the earliest year recoded in the dataset was 1970. We've identified our row criteria on which we will filter the data to investigate our question of interest.

```
# filter the data set to observations of interest
# get the number of observations
nrow(Marine6[Marine6$Year == 1970,])
```

```
[1] 2
```

In the code above we used the logical statement `Marine6$Year == 1970` in the row index position to filter the Marine6 dataset (e.g., get all rows in the dataset where the logical statement is true). This output served as the input for the `nrow()` function which counts the number of rows and returned the answer to our inquiry.

Like the briefly mentioned process of creating the Marine6 dataset, in many instances, we might filter our data to create a new object, or dataset, of interest. We might store all observations according to a specific set of years.

```
# Store observation from 1970 to 2000 in M6_2000
M6_2000 <- Marine6[Marine6$Year <= 2000,]
summary(M6_2000$Year) # max should be 2000
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1970    1983    1991    1989    1996    2000
```

4.2.2 Subsetting

We can use column criteria to reduce our data to only certain variables of interest. We can distinguish the `subsetting` data move from filtering based on the use of column vs. row criteria, although both operations result in what can be considered subsets of the data.

```
# summary only of the subset of variables of interest
summary(Marine6[,c(1,2,8,10)])
```

```
      Year                        Region        Depth       
 Min.   :1970   Northeast US Fall  :164   Min.   :  7.48  
 1st Qu.:1991   Northeast US Spring:132   1st Qu.:  8.43  
 Median :2000   Maritimes Summer   : 99   Median : 52.98  
 Mean   :1999   Southeast US Summer: 93   Mean   : 67.22  
 3rd Qu.:2010   Southeast US Spring: 92   3rd Qu.:119.63  
 Max.   :2020   Southeast US Fall  : 87   Max.   :255.06  
                (Other)            :148   NA's   :53      
               Common.Name
 American lobster    :187
 Banded drum         :158
 Black sea bass      :180
 Guachanche barracuda:126
 Rainbow star        : 26
 Red hake            :138
```

In the code above, we used a vector of indices to get a subset of the Marine6 variables. In larger datasets, particularly for gleaning insights from the `summary()` function, subsetting data in this way can render the output more informative and digestible.

4.2.3 Grouping (Loop Example)

The `grouping` data move may serve various purposes, but is often used to create categories for the purposes of comparisons. Often, the grouping data move is a specific case of creating a new variable or attribute, particularly when the grouping information is needed for subsequent analysis. For example, a new variable containing grouping information may be a parameter that is used to create a data visualization in a certain way.

In the Marine6 dataset, we have species of fish and species of not-fish. We might be interested in comparing the changes in depths over time between these more general classifications. To set this comparison up, we can use grouping.

```
# Using a for loop to create a new group
for(i in 1:nrow(Marine6))
{
  if(Marine6$Common.Name[i] %in% c("American lobster","Rainbow star"))
  { Marine6$Group[i] = "Other" }
  else if(is.na(Marine6$Common.Name[i]))
  { Marine6$Group[i] = NA }
  else
  { Marine6$Group[i] = "Fish" }
}
Marine6$Group <- as.factor(Marine6$Group)
summary(Marine6$Group)
```

```
 Fish Other
  602   213
```

Let's examine the code above, which contains a `for loop`. This programming sequence iterates through each observation in the Marine6 data and returns the values "Other", "NA", or "Fish" based on the specified condition-criteria. In the first condition we introduced the **`%in%`** operator which is a logical statement that returns `TRUE` if the value of "Common.Name" matches any of the characters in the object `c("American lobster","Rainbow star")`, or `FALSE` otherwise. The next condition uses `is.na()`, also a logical statement, which returns `TRUE` if the value of "Common.Name" is missing, and `FALSE` otherwise. Although there are no missing values in "Common.Name", this may not always be the case for a given variable and we should account for this missing value condition in order to run our loop without error. Finally, the last remaining case includes all other common names not specified in the first condition. We do not need to explicitly write this out as it is included in the `else` statement. Note how our data dictionary can provide this factor level information to us. We may even revise our data dictionary to include information about the levels that correspond to fish and those that do not.

4.2.4 Additional Data Moves

In some cases, there may be information spread across multiple datasets that we want to combine into a single dataframe. If each of these dataset has a common "identifier" which links the information, we can use the **merging** data move to accomplish this task. For example, we might want to add instructor information to student data, based on a common course ID. We can merge data with the base R `merge()` function where the inputs would be each dataset and the common identifier on which the merge would be based.

The **creating hierarchy** data move may not be different from a combination of grouping and creating a new variable. However, certain models are based on data hierarchy, and depending on the dataset there may be a need to create this structure for a related analysis. For example, we may have a dataset that contains information on states, counties, and schools. Across the states there may be counties (and schools) that have the same names. In order to distinguish one county from another, information about the state would be necessary. Thus, we would `nest` our counties within the states (and the schools within the counties), and this nesting would create a hierarchical data structure. As with comparing our fish and non-fish groups, we might be interested in visualizing the variation in county test scores across a sample of states, and the hierarchical information would be essential to such a task.

In the examples above, we used based R and `bracket notation` (i.e., referencing indices through conditions in the row aand column spaces of our dataframe) to perform our data moves. Needless to say, there are more ways in R to perform our data moves. In fact, many data moves directly correspond to functions that exist through the `dplyr` package in the "tidyverse". Even better, these functions can really simplify our coding process.

4.3 Tidyverse Data Moves - Tidy Moves

In the previous section, we demonstrated data moves one at a time. We could have easily added both row-filtering and column-subsetting information at the same time to create our new dataset of interest. These do not need to be independent steps, but combining these steps into one may lead to rather long code statements that could potentially be visually difficult to parse (and revise). Let's see an example of this.

```
# summary of a subset of variables
# for the observation prior to 2001 and in one region of interest
summary(Marine6[Marine6$Year <= 2000 &
                  Marine6$Region == "Northeast US Fall", c(1,2,8,10)])
```

```
      Year                              Region       Depth
 Min.   :1974   Northeast US Fall          :93   Min.   : 21.50
 1st Qu.:1980   Gulf of Alaska             : 0   1st Qu.: 34.83
 Median :1987   Gulf of Mexico             : 0   Median : 89.55
 Mean   :1987   Gulf of St. Lawrence South: 0   Mean   : 75.41
 3rd Qu.:1995   Maritimes Summer           : 0   3rd Qu.:108.07
 Max.   :2000   Northeast US Spring        : 0   Max.   :134.87
                (Other)                    : 0   NA's   :1
               Common.Name
 American lobster    :27
 Banded drum         :12
```

```
Black sea bass      :27
Guachanche barracuda: 0
Rainbow star        : 0
Red hake            :27
```

In the code above, we added two conditions into the row space to filter by observations before 2001 and only in the "Northeast US Fall" region. We also added the column space index subsetting criteria, all in the same statement. This accomplishes the filtering and subsetting task of interest in short order, but we can certainly improve on the readability of this code. Also, it's not ideal to have to revisit column information to find out which column names correspond to which indices. So, is there a better - or at least different - way?

4.3.1 A different way (and the pipe operator)

The tidyverse that we previewed in chapter 3 contains a very handy package called dplyr. One of the cool things about the `dplyr` package is that many of the functions that exist within it can be put into direct correspondence with our data moves! Before we jump into the these functions, we need an essential tool called the `pipe` operator, which we can get from the `magrittr` package.

```
# I can add two seperate statements on one line using ";"
library(magrittr); library(dplyr)

#example using the pipe operator
Marine6 %>% summary()
```

```
      Year                       Region                     Species
 Min.   :1970   Northeast US Fall  :164   Centropristis striata:180
 1st Qu.:1991   Northeast US Spring:132   Homarus americanus   :187
 Median :2000   Maritimes Summer   : 99   Larimus fasciatus    :158
 Mean   :1999   Southeast US Summer: 93   Orthasterias koehleri: 26
 3rd Qu.:2010   Southeast US Spring: 92   Sphyraena guachancho :126
 Max.   :2020   Southeast US Fall  : 87   Urophycis chuss      :138
                (Other)            :148
    Latitude     Latitude.Standard.Error   Longitude
 Min.   :26.51   Min.   :0.0000          Min.   :-159.98
 1st Qu.:32.40   1st Qu.:0.1770          1st Qu.: -79.90
 Median :37.92   Median :0.2267          Median : -75.05
 Mean   :37.54   Mean   :0.2942          Mean   : -76.79
 3rd Qu.:42.23   3rd Qu.:0.3342          3rd Qu.: -68.63
```

```
 Max.   :57.50   Max.   :4.4575          Max.   : -60.24
 NA's   :5       NA's   :34              NA's   :5
 Longitude.Standard.Error     Depth        Depth.Standard.Error
 Min.   :0.0000           Min.   :  7.48   Min.   : 0.0000
 1st Qu.:0.2362           1st Qu.:  8.43   1st Qu.: 0.1287
 Median :0.3298           Median : 52.98   Median : 4.4436
 Mean   :0.3859           Mean   : 67.22   Mean   : 5.7448
 3rd Qu.:0.4475           3rd Qu.:119.63   3rd Qu.: 9.2985
 Max.   :1.7813           Max.   :255.06   Max.   :63.2701
 NA's   :34               NA's   :53       NA's   :73
              Common.Name    Group
 American lobster    :187   Fish :602
 Banded drum         :158   Other:213
 Black sea bass      :180
 Guachanche barracuda:126
 Rainbow star        : 26
 Red hake            :138
```

In the code above, the `%>%` symbol is the operator of interest. The `pipe` operator applies the operations that follows (to the right) to the inputs that precede it (to the left). Just to understand a little more about this useful tool, let's briefly step away from our dataset context to see another example of how the pipe operator works.

```
# What's the stadard deviation of [2,6,10]?
c(2,6,10) %>% var() %>% sqrt()
```

```
[1] 4
```

In the example above, we applied the variance function `var()` to [2,6,10], and took the square root of the variance output by applying the `sqrt()` function. This is the same as `sqrt(var(c(2,6,10)))`, but (not so) arguably easier to read, and definitely less prone to parentheses errors. Now that we've seen the `pipe` operator, let's use it to smooth out those data moves!

4.3.2 Filtering 2.0

Previously, using base R, we filtered the Marin6 dataset in order to count the number of observations that corresponded only to the year 1970. Let's try this using the appropriate `dplyr` method, aptly known as the `filter()` function.

```
Marine6 %>%
  filter(Year == 1970) %>% #I can just name the variable
  nrow() #counting the previously filtered data
```

```
[1] 2
```

Wow! Above we were able to count the number of observations corresponding to the year 1970 using `filter()` together with the `pipe` operator. This new format flexibility also allows for in line commenting relative to each step.

4.3.3 Subsetting 2.0

Next, in base R, we used bracket notation and variable indices to subset our data (to produce a summary of interest). Let's take a look at how we can go about accomplishing this subsetting data move (and summary) using the `select()` function.

```
# a summary of the selected variables
Marine6 %>%
  select(Year, Region, Depth, Common.Name) %>%
  summary()
```

```
      Year                       Region        Depth       
 Min.   :1970   Northeast US Fall  :164   Min.   :  7.48  
 1st Qu.:1991   Northeast US Spring:132   1st Qu.:  8.43  
 Median :2000   Maritimes Summer   : 99   Median : 52.98  
 Mean   :1999   Southeast US Summer: 93   Mean   : 67.22  
 3rd Qu.:2010   Southeast US Spring: 92   3rd Qu.:119.63  
 Max.   :2020   Southeast US Fall  : 87   Max.   :255.06  
                (Other)            :148   NA's   :53      
              Common.Name
 American lobster    :187
 Banded drum         :158
 Black sea bass      :180
 Guachanche barracuda:126
 Rainbow star        : 26
 Red hake            :138
```

Notice that we can specify the variable names as they are in the dataset. This is preferable to using indices which can lack sufficient information about what the subset should actually contain.

4.3.4 Grouping 2.0

The grouping data move may serve various purposes and in some cases it is not necessary to create new variables to group data for comparisons. For example, in the `dplyr` package, we can use `group_by()` to create a "grouped" data frame where subsequent functions are applied to each group.

```
Marine6 %>%
  group_by(Common.Name) %>%
  count()
```

```
# A tibble: 6 x 2
# Groups:   Common.Name [6]
  Common.Name              n
  <fct>                <int>
1 American lobster       187
2 Banded drum            158
3 Black sea bass         180
4 Guachanche barracuda   126
5 Rainbow star            26
6 Red hake               138
```

In the code above, we were able to generate a comparison of counts for the variable on which the grouping was based. We input our grouped data frame (created by `group_by()`) into the `count()` function to generate a summary similar to what we got through applying the `summary()` function to `as.factor(Marine6$Common.Name)`.

4.3.5 A few more dplyr examples

Recall the rather involved base R loop that we created for the purpose of adding a grouping variable to our Marine6 dataset. Let's consider how we could do this in the `dplyr` setting.

```
# function to categorize the different common names
fishCat <- function(x)
{ #x is the dataset variable of interest
  ifelse(is.na(x), NA,
         ifelse(x %in% c("American lobster","Rainbow star"),
                "Other","Fish"))
}

# creating a new grouping variable
```

```
Marine6 %>%
  mutate(Group = fishCat(Common.Name)) %>%
  group_by(Group) %>%
  count()
```

```
# A tibble: 2 x 2
# Groups:   Group [2]
  Group     n
  <chr> <int>
1 Fish    602
2 Other   213
```

In the code above we created a `user defined function` (specific to our Marine6 data context) that takes in a dataset variable and returns either NA, "Fish," or "Other." The function uses nested `ifelse()` statements to account for the various criteria, rather than the more general if-else statement that we used for the loop in our base R grouping example. Although this function does not require anything beyond base R, we at least are able to see, in case you hadn't heard, that there is more than one way to do a thing in R. In addition, `ifelse()` is a `vectorized` function and applies the specified conditions to each element simultaneously for a more efficient process!

Below the user-defined `fishCat()` function we introduced the `mutate()` function. This function allows us to create a new variable from one that exists in the referenced dataset. Just as with our base R example, we named this variable "Group." We used `fishCat()` to create the values of "Group" based on the components of "Common.Name". Finally, we applied the `group_by()` function and counted the frequencies within each of our newly created "Group" categories.

Now, lets look at one last function that will combine our filtering and grouping into one sequence of pipes and operations.

```
Marine6 %>%
  filter(Year <= 2000, Region == "Northeast US Fall") %>% #filtering
  select(Year, Region, Depth, Common.Name) %>% #subsetting
  summary()
```

```
      Year                                Region       Depth
 Min.   :1974   Northeast US Fall             :93   Min.   : 21.50
 1st Qu.:1980   Gulf of Alaska                : 0   1st Qu.: 34.83
 Median :1987   Gulf of Mexico                : 0   Median : 89.55
 Mean   :1987   Gulf of St. Lawrence South: 0   Mean   : 75.41
 3rd Qu.:1995   Maritimes Summer              : 0   3rd Qu.:108.07
```

```
Max.   :2000   Northeast US Spring       : 0   Max.   :134.87
               (Other)                   : 0   NA's   :1
              Common.Name
American lobster    :27
Banded drum         :12
Black sea bass      :27
Guachanche barracuda: 0
Rainbow star        : 0
Red hake            :27
```

> **Consider**
>
> In our 2.0 tidyverse comparisons vs. the base R versions, the readability feature of applying data moves with `dplyr` may have come into focus. As you continue to apply data moves, consider the advantages and disadvantages of the two different methods presented here. Which methods might work best for you and do you imagine using both base R and `dplyr` for your data moves in R?

In this section we learned about important data moves that represent common programming applications for data dives and explorations within the data science workflow. We realized these data moves using base R and the `dplyr` package (with help from `magrittr`). Furthermore, we introduced loops and user-defined functions as examples of how various concepts come together and play a role in the data exploration process. As you move forward with using data moves for your data dive, think about the questions that come to mind via this exploration. Your data moves can even guide you towards a new data hypothesis!

5 Python - the Basics

Now that we're oriented with R and even have some data moves in our pocket, let's turn towards the most popular programming language for data science (right above R), python. With learning both Python and R, you will be able to leverage the best of both languages to your data science advantage!

> **Note**
>
> ##Tools for contemporary issues
> As we noted in the introduction, R and Python are two of the most commonly used and preferred data science programming languages. Both languages have accessible tools and capabilities that meet the needs of data science, its evolution, and its role in **exploring contemporary issues** and **specifying exciting discoveries**. These include packages for exploratory data analysis, machine learning, visualization, and more. And, of course, the methods we emphasize allow for transparency and reproducibility, as components of a larger data science workflow and ethical framework.
> What contemporary issues excite you about data science, and what exciting discoveries do you hope to make?

Before we begin our python journey, let's consider some platforms that will allow us to utilize the python language interactively. Jupyter Notebooks

A Jupyter Notebook is a web-based interactive computing platform that supports programming in Python, R, and other languages. Jupyter notebooks allow users to create and share documents that combine code, equations, visualizations, and formatted text. Similarly to RMarkdown and Quarto in RStudio, Jupyter notebooks provide a unified work space where both code and its output can be displayed within the same document.

The image below is an example of a Jupyter Notebook.

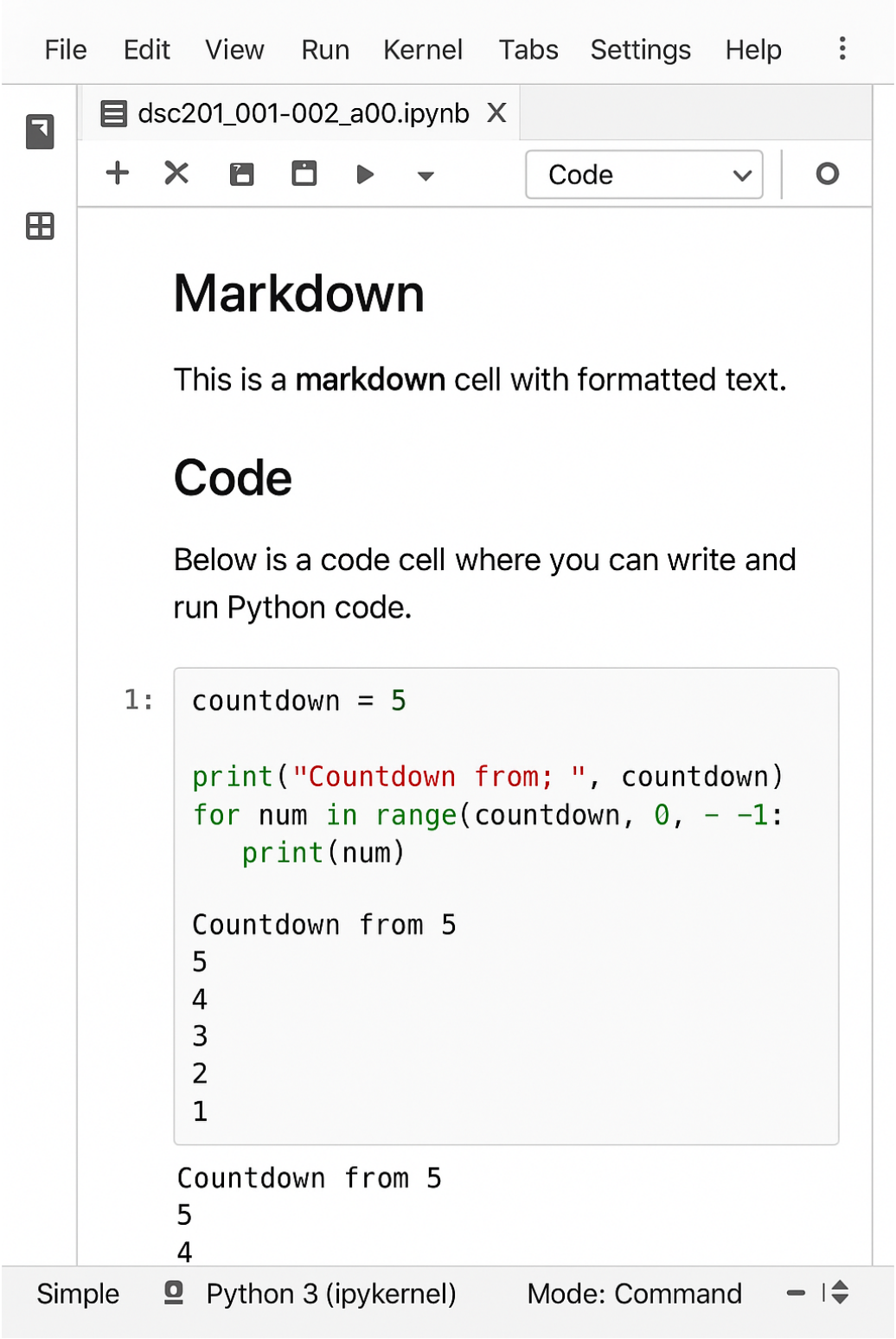


Figure 5.1: Jupyter Notebook Example

As shown in the image above, Jupyter notebooks can have a combination of markdown cells, code cells, and associated output. Jupyter notebooks have a menu and a toolbar, which provide options for managing the document, switching between code and markdown cells, and running code cells.

5.1 Options for your Jupyter notebook workspace

A JupyterLab is an interactive development environment for working with Jupyter notebooks. Having access to a JupyterLab eliminates the need to download, install, and configure Jupyter notebooks on your device. Options for working with Jupyter notebooks outside of a JupyterLab include Anaconda, Visual Studio Code, and other platforms, such as cloud-based Google Colab. Consider your access needs and choose the platform that works best for you!

5.2 Let's talk about Python

Python is a general-purpose programming language, created by Guido van Rossum, first released in 1991. It was designed to have a straightforward, readable syntax for ease of understanding and coding. Currently, python is used across various fields, including web development, data science, and artificial intelligence.

5.3 Comments in Python

> **Note**
>
> ##Recall
> Comments should be written as complete sentences and used to provide context or explain decisions that cannot be effectively communicated through other means like descriptive variable names or the structure of your code. Comments should clarify why specific coding choices were made, detail the processes used to implement those choices, and explain the underlying concepts or assumptions that support them.

At times, as seen in the code chunk below, it can be helpful to place comments directly above the code they reference. This can add clarity and context to the code that follows, and may be a more readable option when inline comments do not work as well.

```
# An arithmetic expression that returns a value.

5 + 3
```

```
8
```

Another use of comments, especially in the collaborative coding process, can be to provide guidance to help those you work with understand your project workflow. For example, comments can indicate when code updates are needed or can serve as a point of reference that specifies the next tasks that need to be completed.

```
# TODO: Check for missing values in the data set.
```

5.4 Python Data Types

Python provides a variety of built-in data types and data structures. We will emphasize the datatypes that are most common, essential for beginners, and that arise in exploratory data analysis settings.

These core data types include:

Data Type	Description
Integer (int)	Represents whole numbers, e.g., 10 or -5.
Float (float)	Represents decimal numbers, e.g., 3.14 or -10.5.
String (str)	Represents sequences of characters or text, e.g., 'Hello' or 'Bien, y tu'
Boolean (bool)	Represents logical values, True or False.

Python is also an object-oriented programming language meaning almost everything that you interact with is an object. Basic data types, like a number (e.g., 10) or a string (e.g., "Kon'nichiwa sekai") are objects that come with built-in capabilities. For example, numbers support arithmetic operations, while strings allow for actions like concatenation used for combining strings together.

Many objects in Python, like numbers and strings, are instances of built-in classes. A class is essentially a template that defines the structure and behavior of objects. It specifies what attributes, or properties, the objects will have and what methods, or actions, they can perform. The int data type is an example of such a class. It has attributes, such as an associated value (e.g., 10), and methods or actions that can be invoked, like addition (e.g., 10 + 5 returns 15).

5.4.1 Determining a variable type

To determine the particular type (or class) of a variable in Python, we can use the aptly named type() function. We can easily imagine uploading a dataset, perhaps with limited data dictionary information, and having the need to understand or verify variable types.

Let's look at a few examples below to explore default associations for different forms of data we might encounter.

```
# Exploring the class of a counting number, like 10

type(10)
```

```
<class 'int'>
```

Now we see that python returns the class int for the number 10. Let's explore the types of a few other examples of common data points.

```
# decimal

type(3.14159)
```

```
<class 'float'>
```

```
# a phrase

type("Kon'nichiwa sekai")
```

```
<class 'str'>
```

```
# a logical value

type(True)
```

```
<class 'bool'>
```

The output lists the corresponding classes in the order in which the code was executed, from top to bottom. Note that the bool (logical) value True is case sensitive, where the first letter is capitalized.

5.4.2 Objects & Operators

As noted above, certain classes have certain operators with which they can interact. However, the same operator can perform different actions in different contexts. In this case, the context would be the variable types to which the operator is applied. To understand the multi-functionality of certain operators let's observe the examples in the following two blocks of code.

```
# Let's apply `+` to two numbers (`int` + `float`)

10 + 3.14159
```

```
13.14159
```

As we might expect, applying the + operator to two numbers results in their addition. But, what if the values are non-numeric - will this produce an error?

```
# Let's apply `+` to words

"in a day," + " or twoooo"
```

```
'in a day, or twoooo'
```

Aha! The + operators also works on the class str (e.g., phrases) and results in concatenation.

> **Know your object**
>
> An important takeaway is that understanding the characteristics of objects and the associated methods is essential to the programming process.

Notice, in the code above, we added a space to the concatenation by making this space a part of the second string.

5.4.3 Additional arithmetic operators in Python

Operator	Description	Example	Result
'+'	Addition	5 + 3	8
'-'	Subtraction	5 - 3	2
'*'	Multiplication	5 * 3	15

‘/‘	Division	5 / 3	1.6667
‘%‘	Modulus (remainder)	5 % 3	2
‘**‘	Exponentiation (power)	5 ** 3	125

5.5 Variables - revisited

With respect to the programming context, in Python a variable is a named reference to an object. Variables can be thought of as labels associated with stored data. These labels can be referenced in your program for viewing or for the purposes of accomplishing other data processing tasks. Just as in R, variables in Python can contain anything from a single value to a complete dataset, or even more.

Variable names in Python can only include lower and upper case letters, digits from 0-9, and underscores (_). Although digits are allowed within a variable name, the names cannot begin with a digit. Variable names also cannot contain a space or be a reserved word - a word that has a predefined meaning and specific purpose within Python - like True.

Recall that we strive for clarity and reproducibility in our data science programming processes. So, despite the flexibility we may have with naming a particular object, or variable, it is a good practice to use descriptive variable names when appropriate. Following this practice can improve the readability and comprehension of your program, which are important in general and particularly in collaboration.

```
# what did these numbers mean again?

x = "201"
y = "001"
z = 1
```

Above, we used perhaps convenient, but absolutely uninformative, variable names for three different numbers (although of different types). However, it wouldn't take much longer to think of informative variable names, as in the code below.

```
# course information

course_code = "201"
course_section = "001"
CreditHours = 1
```

In addition, a little time upfront can save a lot of time in the long run. For example, if you revisit this code later on, you likely won't have to wonder what the variable course_code

contains, as where you might have to investigate or memorize the contents of a variable labeled x.

As you know, informative code comments are great and also essential to our processes. Keep in mind, though, that informative comments alone are not as useful as informative comments and descriptive variable names!

> **Variable name conventions**
>
> As you develop your programming style, and think about future data science collaborations, you may even want to consider a process for naming conventions. Common naming conventions like camelCase, PascalCase, and snake_case are useful choices that can be made to be descriptive and are acceptable across various programming languages, including R and Python (4).

In this chapter, we will follow the naming conventions outlined in PEP 8 (Python Enhancement Proposal 8), which is the official style guide for Python code. For this, snake_case will be used for variables and user-defined functions, and all caps with underscores (ALL_CAPS) will be used for constants.

5.6 Assignment statements - revisited

In R, it is typical to use the <- operator for assignment statements. For assignment statements in Python the = operator is used in all cases for assigning values to variables.

```
# assignment statement in Python are made with "="

# assign 5 to the variable radius
radius = 5

# let's create a variable "diameter" using "radius"
diameter = 2 * radius

# call diameter to display it's contents
diameter
```

```
10
```

Above we used the assignment operator = to create the variable radius and then used an expression to create a new variable called diameter. We can call our variables simply by referencing them as in the output above that is the result of running the line of code with `diameter`.

5.7 Data structures in Python

Python has built-in data structures to store collections of data. These data structures include:

Data.Structure	Description
List (‘list‘)	An ordered collection that can store different data types, where elements can be added, r
Tuple (‘tuple‘)	Similar to a list, but once created, its elements cannot be changed. Example: (1, ”two”,
Dictionary (‘dict‘)	An unordered collection of key-value pairs where each key must be unique and values are

5.7.1 Lists

A `list` in Python is an ordered collection of items that may be of different types of data, such as numbers, strings, or even other lists. Lists are `mutable`, meaning you can add, modify, or remove their elements. Each element in a list is stored in a specific position called an `index`.

> **Note**
>
> In Python the list indices begin at 0! This means that the first element in a list is represented by the 0 index position. The second element in a list is represented by the 1 index positions, and so on. This is a marked difference from indexing in R.

Below, the list `[1, "two", 3.14, False]` contains four elements: an integer (`1`), a string (`"two"`), a float (`3.14`), and a Boolean (`False`). In this case, the first element (`1`) is represented by the `zeroth position` (or the 0 index number), the second element (`"two"`) is represented by the `first position` (or 1 index number), and so on. In general, the indices increase by 1 for each subsequent element of a list.

In the code below, as assignment statement is used to assign the list `[1, "two", 3.14, False]` to the variable `example_list`.

```
# Create a list of the elements 1, "two", 3.14, False

example_list = [1, "two", 3.14, False]

# Display the contents of example_list

example_list
```

```
[1, 'two', 3.14, False]
```

We can access a specific element of `example_list` by denoting its index position inside of brackets (i.e., using `bracket notation`). Below we get the first element of example_list by referencing the zeroth index position.

```
# Access the first element of example_list (in the zeroth index position)

example_list[0]
```

```
1
```

To view each element in the list individually, we can specify the relevant indices within the list, via bracket notation, as inputs to the `print()` function.

```
print(example_list[0])
```

```
1
```

```
print(example_list[1])
```

```
two
```

```
print(example_list[2])
```

```
3.14
```

```
print(example_list[3])
```

```
False
```

Can we view list elements in sequence without having to type out each one? Of course! One way to do this is to use the `slice` syntax denoted by a colon `:`. For example, example_list[1:3], extracts the elements starting from index 1 up to, but not including, index 3.

```
# Access elements starting from index 1 up to, but not including, index 3.

example_list[1:3]
```

```
['two', 3.14]
```

A slice will return a list even if it is a single element.

```
# Access elements starting from index 2 up to, but not including, index 3.

example_list[2:3]
```

```
[3.14]
```

Take a moment to consider how you would get the first through third elements of `example_list` (or a general list).

5.7.1.1 Common list methods

In data science, Python lists are often used to store, organize, and manipulate data using built-in list methods. A `method` is a function that is associated with an object type and is used to perform actions or operations on that object. Methods in Python are called using a specific syntax known as `dot notation`. In dot notation, the format is object.method(arguments). The method comes after the dot and tells Python what action to perform on the object that precedes it.

While there is a comprehensive list of methods available for lists, we will focus on those most commonly used. In particular, this section will cover methods used for adding, removing, and rearranging elements within a list.

5.7.1.1.1 .append()

The `.append()` method adds a single element to the end of a list. The element can be of any data type such as a number, string, or even another list. This operation is done `in place`, meaning it modifies the original list directly and does not require an additional assignment statement.

```
# Create a list of the elements 1, "two", 3.14, False

example_list = [1, "two", 3.14, False]

# Add the element 5 to example_list

example_list.append(5)

# Display the contents of example_list

example_list
```

```
[1, 'two', 3.14, False, 5]
```

5.7.1.2 .remove()

The `.remove()` method removes only the first occurrence of a specified element from a list, even if the element appears multiple times. Like .append(), the .remove() method performs the operation in place.

```
# Create a list of the elements 1, "two", 3.14, False

example_list = [1, "two", 3.14, False]

# Remove the element 'two' in example_list

example_list.remove("two")

# Display the contents of example_list

example_list
```

```
[1, 3.14, False]
```

The code below illustrates how only the first object of a repeating list element is removed using `.remove()`.

```
# A list with "two times" repeated

repeat2x_list = [1, "two times", "two times", 3]

# Display the contents of repeat2x_list

repeat2x_list
```

```
[1, 'two times', 'two times', 3]
```

```
# Remove the element 'two times' in repeat2x_list

repeat2x_list.remove("two times")

# Display the contents of repeat2x_list after .remove()

repeat2x_list
```

```
[1, 'two times', 3]
```

5.7.1.3 .sort()

The `.sort()` method arranges the elements of a list. Ascending order is the default arrangement. However, unlike the two prior methods, the `.sort()` method only works on homogeneous lists. This means that all elements in the list must be of the same data type (e.g., all integers or all strings).

To sort a list in descending order, you can use the `reverse=True` argument within the .sort() method. This will reverse the default sorting behavior, placing the largest elements first.

```
# A list containing the elements "elephant", "zebra", "tiger", "panda"

animals = ["elephant", "zebra", "tiger", "panda"]

# Sort in ascending order

animals.sort()

# Display the contents

animals
```

```
['elephant', 'panda', 'tiger', 'zebra']
```

```
# A list containing the elements "mango", "apple", "cherry", "banana"

fruits = ["mango", "apple", "cherry", "banana"]

# Sort in ascending order

fruits.sort(reverse = True)

# Display the contents

fruits
```

```
['mango', 'cherry', 'banana', 'apple']
```

5.7.2 Tuple

In Python, a `tuple` is an ordered collection of items that can be of different data types, such as numbers, strings, or even other tuples. Like lists, each element in a tuple is assigned an

index, starting at 0 for the first element, 1 for the second, and so on, incrementing by 1 for each subsequent element.

Below, we use an assignment statement to assign the tuple (1, “two”, 3.14, False) to the variable `example_tuple`.

```
# Create a tuple of the elements 1, "two", 3.14, False

example_tuple = (1, "two", 3.14, False)

# Display the content of example_tuple

example_tuple
```

```
(1, 'two', 3.14, False)
```

So is there a difference between lists and tuples? Well, yes…

The main difference between a list and a tuple is that lists are mutable (you can change their elements), while tuples are immutable (their elements cannot be changed once created).

5.7.3 Dictionary

A dictionary in Python is an unordered collection of `key-value pairs` that can store items of different data types, such as numbers, strings, lists, tuples, or even other dictionaries. Unlike lists or tuples, a dictionary does not use numeric index values. Instead, each element (or value) of a dictionary is accessed using its corresponding, unique key. In addition to being unique, dictionary keys must be immutable data types meaning they cannot be changed after being created. For example, keys can be numbers, strings, or tuples!

> **i** Tuples as dictionary keys
>
> Even though tuples themselves are immutable, they can still contain mutable objects like lists. However, to use a tuple as a key in a dictionary, everything inside the tuple must also be immutable. For example, a tuple that contains a list cannot be used as a dictionary key, because lists can change: But, dictionary keys must stay the same.

For example, the dictionary `{1 : "Turtle Power", "id" : "Leonardo", 2 : "pizza", "total" : 4, "leader" : True}` contains five key-value pairs: a number key `1` with a value of `"Turtle Power"`, a string key `"id"` with a value of `"Leonardo"`, a number key `2` with a string value of `"pizza"`, a string key `"total"` with a number value of `4`, and a string key

"leader" with a Boolean value of True. Each key is a unique identifier for its corresponding value.

Below, we use an assignment statement to assign the dictionary {1 : "Turtle Power", "id" : "Leonardo", 2 : "pizza", "total" : 4, "leader" : True} to the variable tmnt_dict.

```
# A dictionary with key-value pairs
# 1-"Turtle Power", "id"-"Leonardo", 2-"pizza", "total"-4, "leader"-True

tmnt_dictionary = {1 : "Turtle Power", "id" : "Leonardo", 2 : "pizza", "total" : 4, "leader"

# Display the contents

tmnt_dictionary
```

```
{1: 'Turtle Power', 'id': 'Leonardo', 2: 'pizza', 'total': 4, 'leader': True}
```

Since dictionaries are not ordered the values cannot be accessed using indices. Instead, to access the values in a dictionary we need to use key names with bracket notation. For instance, the expression tmnt_dictionary[“id”] can be used to retrieve the value associated with the key ”id”.

Below, we use the print() function, our dictionary reference and the keys in bracket notation to view each of the values.

```
print(tmnt_dictionary[1])
```

```
Turtle Power
```

```
print(tmnt_dictionary["id"])
```

```
Leonardo
```

```
print(tmnt_dictionary[2])
```

```
pizza
```

```
print(tmnt_dictionary["total"])
```

```
4
```

```
print(tmnt_dictionary["leader"])
```

```
True
```

Since a dictionary is a mutable data structure, we can add new key-value pairs using notation that is similar to how we access values. For example, the assignment statement tmnt_dictionary[(“Type”, “id2”)] = [“Nunchucks”, “Michelangelo”] adds the key-value pair (“Type”, “id2”)-[“Nunchucks”, “Michelangelo”] to tmnt_dictionary, where the key (“Type”, “id2”) is a tuple and the value [“Nunchucks”, “Michelangelo”] is a list.

```
# Add the key-value pair ("Type", "id2")-["Nunchucks", "Michelangelo"] to tmnt_dictionary

tmnt_dictionary[("Type", "id2")] = ["Nunchucks", "Michelangelo"]

# Display tmnt_dictionary

tmnt_dictionary
```

```
{1: 'Turtle Power', 'id': 'Leonardo', 2: 'pizza', 'total': 4, 'leader': True, ('Type', 'id2'
```

5.7.4 Common dictionary methods

Dictionaries are often used in data science to store and organize data. To facilitate the use of dictionaries, Python has a set of methods that can be used to perform actions on these data structures. Next, we’ll look at a few methods used that can be used to add and access key-value pairs within a list.

5.7.4.1 .items()

The .items() method is used to retrieve all key-value pairs in a dictionary. It returns a read-only view of the dictionary data called `dict_items`. In Python, `dict_items` is a special type of object that shows all the key-value pairs in a dictionary as a collection of tuples. Each tuple contains one key and its corresponding value.

```
# A dictionary with key-value pairs

tmnt_dictionary = {1: 'Turtle Power', 'id': 'Leonardo', 2: 'pizza', 'total': 4, 'leader': Tru

# Display the key-value pairs as tuples using .items()
tmnt_dictionary.items()
```

```
dict_items([(1, 'Turtle Power'), ('id', 'Leonardo'), (2, 'pizza'), ('total', 4), ('leader', T
```

5.7.4.2 .keys()

The .keys() method can be used to retrieve all the keys in a dictionary as a dict_items object.

```
# display the keys of tmnt_dictionary

tmnt_dictionary.keys()
```

```
dict_keys([1, 'id', 2, 'total', 'leader', ('Type', 'id2')])
```

5.7.4.3 .values()

You likely can guess what the .values() method returns. If you were thinking that this method will return all of the values in a dictionary as a dict_items object, you'd be right!

```
# display the values of tmnt_dictionary

tmnt_dictionary.values()
```

```
dict_values(['Turtle Power', 'Leonardo', 'pizza', 4, True, ['Nunchucks', 'Michelangelo']])
```

Within a data science workflow, lists, tuples, and dictionaries can serve different purposes and meet different needs. In this way, each of these Python data structures are useful and important. For comparisons, lists are often better for storing and modifying collections of data as where tuples may be preferred for fixed data like names or IDs. Dictionaries would likely be preferable for organizing data that needs to be accessed using reference labels or keys.

5.8 Python libraries

A library in Python is similar to a package in R. A Python library is a collection of pre-written code that provides functions, classes, and modules to perform tasks like data manipulation, analysis, and visualization. Python offers several libraries that are commonly used in data science.

5.8.1 NumPy

`NumPy` (Numerical Python) is a library that facilitates numerical data processing. This library includes specialized data structures, such as multidimensional arrays, that allow for vectorized operations. Recall that vectorization allows for element-wise computations without the need for loops. `NumPy` also has built-in functions for fast and efficient mathematical and statistical calculations.

Before we can use `NumPy` features, we have to first import the library into a script or Jupyter Notebook. We can do this for a general Python library by using the `import` command followed by the library name. Some libraries, like `NumPy`, are commonly imported with an `alias` to simplify the code syntax when accessing features of the library. For example, `import numpy as np` allows you to use `np` as a shorthand when calling `NumPy` functions.

```
## Import the numpy library with the alias np
import numpy as np
```

The standard convention for calling `NumPy` functions is to prefix them with `np`.

```
# Store the mean of the list into ex_calc

ex_calc = np.mean([1, 2, 3, 4, 5])

# display the value of ex_calc using print()

print(ex_calc)
```

```
3.0
```

`NumPy` functions, like `np.mean()`, are primarily designed to work with `NumPy arrays`. However, many of them also accept Python lists as input because `NumPy` automatically converts lists into arrays internally. This was the case in the code above since `[1,2,3,4,5]` is a list. Although this conversion worked in this case, using arrays directly is better to ensure proper compatibility and performance.

5.8.2 pandas

R includes many data analysis features in its base version. This is not the case for base Python. So, in Python, we need to leverage libraries like `pandas` to get the specialized functions and tools we need for our data science tasks. So what is `pandas`?

`pandas` is a Python library used for data analysis and data processing. It provides data structures like `Series` and `DataFrames`, which can be used to store, organize, manipulate, and analyze data. `pandas` provides tools to facilitate data cleaning, data moves, like filtering & merging, data visualization and more.

As we did with `NumPy`, we can import `pandas` with the `import` statement and create an alias for it.

```
## Import the pandas library with the alias pd

import pandas as pd
```

After importing the pandas library, we gain access to data structures and tools essential for performing various data moves.

5.8.2.1 Series and DataFrames

The main data structures that we will use are the `Series` and the `DataFrame`.

A `Series` is a 1-dimensional labeled array that is similar to a vector in R. Each element has the same data type and is associated with an index, which serves as the label for accessing the data.

A `DataFrame` in pandas is a 2-dimensional labeled data structure that organizes data in a tabular format (i.e., rows and columns). A DataFrame can contain multiple data types because each column is a Series with its own data type. Columns are labeled (typically by name), and rows are identified by an index. A DataFrame is like a spreadsheet, where each row represents an observational unit and each column represents a characteristic or feature of that observed unit.

5.8.2.2 Pandas Functions

Pandas provides a comprehensive set of attributes and functions that can be used to analyze and visualize data. `Attributes` are properties of pandas objects that provide information about the object. For example, an attribute can be used to get the dimensions of a DataFrame.

Functions (also called methods) perform operations on pandas objects. We might use a function to compute the mean of the values in a numerical Series, for example. By using pandas attributes and functions, we can perform data moves to facilitate analysis and prepare data for visualization!

6 Project Part 3.2: Diving into Data Exploration - Python

6.1 Data Moves in Python

The Ramen Rater has been around since 2002. The website's founder, Hans Lienesch, reviews ramen noodles and records data on the brand, variety, style, country, and his own personal rating with scores given in increments of 0.25. As of now, the list contains reviews dating back to 2002. To expand upon the background description and address other components of the data dictionary criteria, we will read in the The Big List dataset and use some functions to access general metadata and information on the variables within.

```
## Load in the data into a pandas DataFrame, using the pd.read_csv function
import pandas as pd

ramen = pd.read_csv("theramenrater_big_list.csv")
```

When data is read in through the pandas read_csv method the default structure is a DataFrame. Recall, that DataFrames store data in tabular format (e.g., like rows and columns in a spreadsheet).

We could use pandas functions (or methods) to extract useful information about the data, such as column names and data types. This information is useful in general and is often an initial, default, step in a data science workflow upon reading in data. We can begin with using the .info() method which will return a summary of information about the structure of the DataFrame.

```
## List summary information about the ramen dataframe

ramen.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5015 entries, 0 to 5014
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
```

```
---  ------          --------------  -----
 0   review_number   5015 non-null   int64
 1   brand           5015 non-null   object
 2   variety         5015 non-null   object
 3   style           5015 non-null   object
 4   country         5015 non-null   object
 5   stars           5015 non-null   float64
dtypes: float64(1), int64(1), object(4)
memory usage: 235.2+ KB
```

The output above shows that there are 5015 entries (rows) and 6 columns. In the table, each row represents a column in the `ramen` dataset and `#` shows the index for the columns (from 0 and to 5), the column name, the Non-Null Count (i.e., the number of non-missing values), and the data type. The review_number column is int64 (numerical), the stars column is float64 (numerical), and the remaining columns are objects. When a column's data type is listed as object, the values could be strings, numbers, or a mix of both. All of this information is useful and includes things we would want to include in something like a data dictionary.

We could also retrieve individual pieces of this information using DataFrame attributes like `.shape` and `.columns`. `Attributes` are properties of pandas objects, such as metadata (e.g., column names, indices), that can be accessed using dot notation just like methods. Unlike methods, though, attributes are called without using parentheses `()`.

```
## Check the dimensions of the dataset

ramen.shape
```

```
(5015, 6)
```

```
## List the variables in the dataset

ramen.columns
```

```
Index(['review_number', 'brand', 'variety', 'style', 'country', 'stars'], dtype='object')
```

6.1.1 Accessing Column Values

To retrieve all values from a column in Pandas we can use bracket notation where the column name is placed inside of square brackets and enclosed by single or double quotes. For example, to access the values in the style column we can use ramen[‘style’]. The returned object is a Series.

```
ramen['style']
```

```
0       Pack
1       Bowl
2       Bowl
3       Bowl
4        Cup
        ...
5010    Pack
5011    Pack
5012    Pack
5013    Pack
5014     Cup
Name: style, Length: 5015, dtype: object
```

A benefit of returning a Series is that it supports vectorized operations. Pandas also provides built-in methods for Series that perform a wide range of tasks for both categorical and numerical data. In addition, Series have attributes that store information such as the data type and size.

6.1.2 Summarizing

The summarizing data move can be used to condense the contents of a variable into frequency counts or proportions for categorical variables, and other descriptive statistics of interest for numerical variables. As an example, we can use the `.value_counts()` method to tally the different styles of noodles.

```
## Create a frequency table of the labels in the 'styles' column

ramen['style'].value_counts()
```

```
style
Pack          2637
Bowl          1022
Cup           1002
Tray           228
Box            120
Restaurant       3
Bar              1
Bottle           1
```

```
Can              1
Name: count, dtype: int64
```

It appears that most of the reviewed noodles are sold in packs, bowls, or cups. Slightly more than half were in packs, while bowls and cups make up another 40%.

We can use the `.describe()` method to get a comprehensive summary of a numerical variable. The `describe()` method returns a summary that includes the number of non-missing values, as well as statistical measures that include the mean, standard deviation, minimum, 25th percentile (Q1), 50th percentile (median), 75th percentile (Q3), and maximum.

```
## Generate summary statistics for the 'stars' column

ramen['stars'].describe()
```

```
count    5015.000000
mean        3.735278
std         1.090282
min         0.000000
25%         3.250000
50%         3.750000
75%         4.500000
max         5.000000
Name: stars, dtype: float64
```

The output above allows us to get an idea of the distribution of the data. It appears the ratings may be slightly skewed to the right when observing the differences between Q2 and the median vs. the median and Q3. Looking ahead, a related visualization could provide an informative visual depiction of the data distribution.

6.1.3 Filtering

Recall that the filtering data move is what we use to reduce or examine a dataset based on certain row criteria. In our ramen dataset we have the country of origin for each noodle and it may be of interest to analyze the information related to a specific country. For example, suppose we wanted to filter for all the noodles that are manufactured in Japan. How can we do this?

6.1.3.1 Boolean Masks

A Boolean mask is typically a Series containing `True` and `False` values. It is created by applying comparison operators (`==`, `>`, `<`, etc.) or logical conditions (`&`, `|`, etc.). To create a Boolean mask for the rows where the country is Japan, we can use the comparison statement `ramen['country'] == "Japan"`. This returns a Series that can be used to filter out, or hide, the rows we do not want (e.g., countries other than Japan).

```
## Create a Boolean mask where 'country' == "Japan"

ramen['country'] == "Japan"
```

```
0       False
1       False
2       False
3       False
4       False
        ...
5010    False
5011    False
5012    False
5013    False
5014    False
Name: country, Length: 5015, dtype: bool
```

So, now that we have boolean masks at our disposal, we can use one to filter the data of interest. To do this, we can assign the filter criteria to a variable (perhaps named mask) and use it to filter the dataframe.

```
## Create the Boolean mask where 'country' == "Japan"

mask = ramen['country'] == "Japan"

## Filter the dataframe
## The .head() method limits the output to the first 5 rows

ramen[mask].head()
```

```
      review_number     brand                  variety style country  stars
1105           3464  Takamori           Agodashi Udon  Pack   Japan    5.0
1106           4613    Ippudo  Akamaru Modern Ramen   Box   Japan    5.0
```

```
1107           3382    Itsuki              Akikara Ramen  Pack   Japan    5.0
1108           4308   Marutai  Asahikawa Soy Sauce Ramen  Pack   Japan    5.0
1109           2974    Itomen               Bansyu Ramen  Pack   Japan    5.0
```

Note: The use of the `.head()` method returns only the first five rows (from the filtered dataframe). What method could we use to see that the filtered data contains 1039 entries?

Next, just in case the opportunity to travel to Japan arises and so as to lean on data for the best ramen experience, let's find out about the ramen that's rated the best!

```
## Find the average 'stars' rating
## The .mean() method calculates the mean of a numerical Series
mean_stars = ramen['stars'].mean()

## Create the Boolean mask where 'country' == "Japan" and 'stars' greater
## than or equal to the average stars rating
mask = (ramen['country'] == "Japan") & (ramen['stars'] >= mean_stars)

## Filter the dataframe and display the number of rows that match the condition
ramen[mask].shape[0]
```

```
660
```

It looks like we have 660 options to choose from, so maybe we should narrow the search. But first, let's see what's going on in the code above.

- We took the mean of the ratings and stored it in `mean_stars`.
- Then, added more criteria to our `mask` variable to filter rows where the country is Japan and (`&`) the ratings are greater than or equal to (`>=`) the overall average.
- Lastly, we applied the mask criteria to our ramen data and called the `shape` attribute (which gives the dimensions).
  - NOTE: Since Python indices begin at zero, the `[0]` index of shape yields the number of rows.

> **So, what about ramen?**
>
> Although ramen was originally invented in China, it was Japan that refined and popularized it, and as a result, modern ramen is distinctly Japanese. So much so that it is featured in the Shin-Yokohama Ramen Museum, which documents its rich history and tasty transformations over time!
> Note, that contextual information can facilitate your data exploration workflow and efficacy.

6.1.3.2 Query

In programming, a query is a request to access or retrieve information from a database or dataset based on conditions or criteria. Queries may also include instructions to filter, sort, or aggregate the data.

In Pandas, the `.query()` method is specifically used to filter rows in a DataFrame based on a string input that consists of a boolean expression. For example, we can assign a string that contain a boolean expression (i.e., filtering criteria) to a variable and pass it as the parameter to .query().

Let's use this method to show a different way to filter the ramen DataFrame for rows where the country is equal to Japan.

```
## Define a query string to filter the DataFrame for rows where the 'country' is "Japan"

q = "country == 'Japan'"

## Use the .query() method to filter the DataFrame and access the first 5 entries (with the

ramen.query(q).head()
```

	review_number	brand	variety	style	country	stars
1105	3464	Takamori	Agodashi Udon	Pack	Japan	5.0
1106	4613	Ippudo	Akamaru Modern Ramen	Box	Japan	5.0
1107	3382	Itsuki	Akikara Ramen	Pack	Japan	5.0
1108	4308	Marutai	Asahikawa Soy Sauce Ramen	Pack	Japan	5.0
1109	2974	Itomen	Bansyu Ramen	Pack	Japan	5.0

Notice that the results are identical to those obtained using a Boolean mask. While the advantages of using `.query()` may not be clear for a single condition, its benefits become more apparent with more complex filtering. For example, suppose we want to find extreme cases where the noodle ratings are 0.0 or 5.0, for Myojo brand, Tray-style noodles, manufactured in Japan.

```
## Define a query string to filter the DataFrame for
## rows where the 'country' is "Japan",
## the style of the noodles is "Tray",
## the brand is "Myojo", and
## the stars rating is either 0.0 or 5.0

q = 'country == "Japan" and style == "Tray" and brand == "Myojo" and (stars == 0.0 or stars =
```

```
## Use the .query() method to filter the DataFrame with criteria q
ramen.query(q)
```

```
      review_number  brand  ... country stars
1112           3940  Myojo  ...   Japan   5.0
1151           2951  Myojo  ...   Japan   5.0
1170           4686  Myojo  ...   Japan   5.0
1171           1103  Myojo  ...   Japan   5.0
1172           2801  Myojo  ...   Japan   5.0
1173           5014  Myojo  ...   Japan   5.0
1174           3470  Myojo  ...   Japan   5.0
1175           3231  Myojo  ...   Japan   5.0
1270           2906  Myojo  ...   Japan   5.0
2142           3505  Myojo  ...   Japan   0.0

[10 rows x 6 columns]
```

The output shows partial information for the first ten rows of the filtered dataset of interest. In terms of the code, in this instance, the query string improves readability and reduces the required syntax in comparison to specifying the same criteria with a Boolean masks. When using a Boolean mask, you need to repeatedly reference the DataFrame and use bracket notation. In contrast, `.query()` lets you reference column names directly as variables in a string expression, which is often more efficient.

6.1.4 Subsetting

Recall that the subsetting data move involves reducing the data based on column criteria. We can subset a Pandas DataFrame by specifying columns to keep using bracket notation, or by dropping columns that are not needed using the `.drop()` method. For bracket notation, we can input a list object consisting of column names inside brackets as shown below. Subsetting the data frame in this way results in a new DataFrame with the columns of interest.

```
## List of columns to select

columns = ['brand', 'style', 'stars']

## Subset the ramen dataframe
## ramen[['brand', 'style', 'stars']].head() results in the same

ramen[columns].head()
```

```
    brand style  stars
0   Maggi  Pack   5.00
1  Suimin  Bowl   5.00
2  Suimin  Bowl   5.00
3  Suimin  Bowl   5.00
4  Suimin   Cup   4.25
```

Now, let's subset by dropping unneeded columns, using `.drop()`.

```
## Drop the review_number column from the ramen dataframe

ramen = ramen.drop(columns = ['review_number'])

## Verify the column was removed

ramen.columns
```

```
Index(['brand', 'variety', 'style', 'country', 'stars'], dtype='object')
```

6.1.5 Grouping

Recall that the grouping data move results in certain categorizations that can allow for realted comparisons via statistics, visualizations, models and more. In the Pandas library, the `.groupby()` methods groups rows based on values in one or more variables. For example, we may be interested in groupings based on levels within a specified categorical variable (e.g., ramen varieties). Once the grouping has been implemented we can apply aggregation methods, such as summing the totals, for variety.

```
## Create a groupby object

style_grps = ramen.groupby('style')
```

Above, we created a variable `style_grps`. This is a grouped object that contains DataFrames organized into subgroups based on the levels within the `style` column. Each subgroup contains a separate DataFrame consisting of row information corresponding to one of the unique levels (e.g., "Cup", "Can", etc.) in `style`.

In the code below, the command `style_grps.groups.keys()` displays all the subgroup names (i.e., the grouping criteria or the levels of `style`) in the `style_grps` object. Note, that the `.groups` attribute of a grouped object is a `dictionary` on which we can run the `.keys()` method.

```
## Displays all the subgroup names in grps

style_grps.groups.keys()
```

```
dict_keys(['Bar', 'Bottle', 'Bowl', 'Box', 'Can', 'Cup', 'Pack', 'Restaurant', 'Tray'])
```

Various data moves are related and can be complimentary. Although the grouping data move does not remove row information (but rather partitions the rows) and filtering does remove certain rows, we could choose to use either of these data moves to accomplish certain desired tasks. For example, if we were interested in comparing the average style rating across each style, we could use our filtering data move for each of the 9 different styles and compute the means of the stars ratings for each filtered dataset. Or, we could leverage grouping to compute the means in a more efficient way using our `style_grps` object. One difference in using grouping vs. filtering besides reduction in code is that computations applied to the grouped object are vectorized (and also more efficient in that way). The mean stars per group calculation is demonstrated below.

```
## Calculate the mean stars rating for each style in the style_grps grouped object
## The .sort_values method arranges the output in ascending order

style_grps['stars'].mean().sort_values()
```

```
style
Cup           3.493693
Can           3.500000
Restaurant    3.583333
Tray          3.595965
Bowl          3.664540
Pack          3.851422
Bottle        4.000000
Box           4.060417
Bar           5.000000
Name: stars, dtype: float64
```

Applying the `mean()` function to our grouped object results in a Series. We can then apply the .sort_values() method to this Series so that the group means appear in ascending (or descending) order.

From the output we can see that the three highest mean ratings are for Bar, Box and Bottle. To complement the mean rating analysis, we should also examine the number of occurrences of each style, and we can do this by applying the appropriate method to our same `style_grps` object.

```
## Return the counts of non-missing values in the stars column
## for each style in the style_grps object
## The .sort_values() method arranges the output in ascending order

style_grps['stars'].size().sort_values()
```

```
style
Bar              1
Bottle           1
Can              1
Restaurant       3
Box            120
Tray           228
Cup           1002
Bowl          1022
Pack          2637
Name: stars, dtype: int64
```

From the output we can see that two of our top three mean ratings are only represented by 1 review. So, we can regard the other averages (besides restaurant) represented by many more ratings as more reliable with respect to the style categorization. The frequencies may reflect the preferences of the rater, or may be reflective of a concept like "availability". At least, we have an idea about a line of inquiry that may be worth investigating and we might want to put more or less faith in some mean ratings than others. ### Calculating (a new attribute)

The calculating a new attribute data move allows us to create new variables from existing ones. We might calculate a new attribute to reveal underlying patterns in the data, to facilitate comparisons, or to preparing data for analysis, among other useful transformations. We can calculate various types of new attributes including numerical or categorical variables. For example, two numerical variables might be combined in a formula to calculate a new quantity, or used to sort observations into groups like small, medium, large. These calculated values become part of the dataset and can be used to support further exploration.

6.1.5.1 A calculating data move example

The American Federation of Labor and Congress of Industrial Organizations (AFL-CIO) website provides data on CEO compensation packages in the United States. The information on the compensation packages includes base salary, bonuses, stock awards, and other earnings, all captured in a dataset compiled from AFL-CIO data.

Below are the attributes and descriptions for this dataset:

```r
library(knitr)
library(kableExtra)

# Create the data frame
compensation_table <- data.frame(
  Name = c( "ticker", "salary", "bonus",
            "stock_awards", "option_awards","non_equity_comp",
            "pension_change","other_comp"
  ),
  Description = c( "Stock ticker symbol for the company.",
                   "Base annual salary.", "Additional cash bonus.",
                   "Value of stock granted.", "Value of stock options granted.",
                   "Performance-based cash compensation not tied to equity.",
                   "Increase in pension value and deferred compensation earnings.",
                   "Miscellaneous compensation (e.g., perks, benefits)."
  )
)

kable(compensation_table,
      caption = "CEO Compensation Packages - Variable Descriptions (AFL-CIO)",
      booktabs = TRUE,
      longtable = TRUE,
      align = "l") %>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

Table 6.1: CEO Compensation Packages - Variable Descriptions (AFL-CIO)

Name	Description
ticker	Stock ticker symbol for the company.
salary	Base annual salary.
bonus	Additional cash bonus.
stock_awards	Value of stock granted.
option_awards	Value of stock options granted.
non_equity_comp	Performance-based cash compensation not tied to equity.
pension_change	Increase in pension value and deferred compensation earnings.
other_comp	Miscellaneous compensation (e.g., perks, benefits).

Although the dataset does not contain a column representing total compensation, we can calculate this attribute from the existing data. We could also create additional attributes that represent various measures like the percentage of total compensation coming from stock or bonus pay.

```
# Load the CEO compensation data
ceo_pay = pd.read_csv("ceo_compensation_summary.csv")

# Calculate total compensation and store the result in the dataframe in a new column called t
ceo_pay['total'] = (
    ceo_pay['salary'] +
    ceo_pay['bonus'] +
    ceo_pay['stock_awards'] +
    ceo_pay['option_awards'] +
    ceo_pay['non_equity_comp'] +
    ceo_pay['pension_change']
)
```

Now that we've computed this new column, additional analysis and exploration can be performed to better understand CEOs payment trends. In addition, we could use the summarizing data move to calculate statistics such as the mean, median, and standard deviation of the new `total` attribute.

6.1.6 A little extra - Merging & Joining

Merging and joining involve combining information from various datasets based on some identifying criteria. These terms are often used interchangeably.

Consider the `ceo_pay` dataset, which includes details about the CEO compensation structure but lacks information about the companies and the particular CEOs. If we had the additional company and CEO information in another dataset and a common identifier between these disparate sources, we could merge the data into a single source containing all of the information.

As it turns out, the `company` dataset, read in below, has information on company and ceo names. In addition, both the `company` and `ceo_pay` datasets contain a common identifier, `ticker`. So, we can use this identifier to merge the two data sources into one.

```
# Load the company information data

company = pd.read_csv("company_info.csv")

# Join the CEO pay dataset with the company dataset

ceos_and_pay = pd.merge(ceo_pay, company, on = 'ticker', how = 'inner')

# Show the new combined data column names
```

```
ceos_and_pay.columns
```

```
Index(['ticker', 'salary', 'bonus', 'stock_awards', 'option_awards',
       'non_equity_comp', 'pension_change', 'other_comp', 'total', 'city',
       'state', 'display_name', 'fiscal_year', 'ceo_name'],
      dtype='object')
```

```
# preview the first 5 rows

ceos_and_pay.head()
```

```
  ticker   salary  ...  fiscal_year                     ceo_name
0    TPG   509615  ...       2023.0           Mr. Jon  Winkelried
1     CG   838462  ...       2023.0  Mr. Harvey Mitchell Schwartz
2   AVGO  1200000  ...       2024.0               Mr. Hock E. Tan
3   PANW   750000  ...       2024.0      Mr. Nikesh  Arora C.F.A.
4   COTY  3549000  ...       2024.0               Ms. Sue Y. Nabi

[5 rows x 14 columns]
```

Above we used the `pd.merge()` function (in the Pandas library) to combine the two dataframes based on the `ticker` variable/identifier in both datasets. We specified the identifier `ticker` as the `on` parameter value, and we specified `inner` as the `how` parameter value. The `inner` specification defines an inner-join which means only the rows where both datasets have a matching `ticker` value are merged and kept in the resulting dataset. In general `pd.merge()` is used to combine DataFrames based on one or more common keys (i.e., columns).

Now that we have additional information in our `ceos_and_pay` dataset we have the option to explore entirely new data investigations, such as CEO salaries by industry, or salaries by other company characteristics. And of course, to do this, we can leverage data moves!

7 Project Part 4.1: Visualizations & Trends in Python

Data visualization is the practice of representing data graphically to reveal patterns, trends, and insights that might be difficult to interpret from tables or raw numbers alone. The fundamentals of data visualization include selecting appropriate graphical representations, ensuring clarity and accuracy, and emphasizing key insights while avoiding misleading representations (5).

While data visualization involves many concepts, techniques, and considerations, we will focus on fundamental visualization types commonly used in exploratory data analysis (e.g., bar charts, histograms, box plots, line charts, and scatter plots). We can create these visualizations in Python and R using relatively simple coding techniques.

There are many considerations that need to be taken into account when determining how to effectively create visualizations. When making decisions that impact the characteristics of a chosen visualizations we can aim to:

- simplify information,
- concisely communicate findings,
- inform or support decision-making,
- depict relationships within data, among other considerations.

For this course we'll focus on choosing data visualizations that help to identify patterns within data.

7.1 A few guidelines

To understand features of the data we can start by selecting a visualization that is appropriate for a particular data type. For example, if we're attempting to gain insight into a single variable consisting of categorical data, we might select a bar chart. This choice of visualization could allow us to compare the frequencies of the various factor levels. For a numerical variable, a histogram could provide similar information and depict relative frequencies or, more generally, the distribution of data values. To gain insights into bivariate relationships, we might consider mosaic plots for two categorical variables, or scatterplots for two quantitative variables. We could even extend the information depicted in a 2-dimensional visualization

to multiple variables by including color, facets, and other overlays. Other visualizations that may be informative in depicting trends are line charts, which can show patterns over time (e.g., a time series), or perhaps a spaghetti plot to visualize grouped trends over time. For determining a graphical representation of data, consider: "What visualization can I choose to clearly emphasize trends and patterns while avoiding clutter and unnecessary complexity."

Here are a few guidelines:

- minimize clutter
- use appropriate scales and dimensions
- clearly label axes
- consider order when appropriate (e.g., ordinal information)

Your visualization should be self explanatory to a viewer. This means that there should not be a need to explain all of the details of the visual beyond the information conveyed by the visualization itself.

7.2 Visualizations in python

Although R has excellent visualization capabilities, such as those available through the ggplot2 library, visualizations in Python can be embedded into a larger data science workflow when Python is the preferred language and switching languages or platforms is not desirable. However, utilizing the best of both languages is always an option, especially in platforms that can accommodate multiple languages, such as Quarto, Jupyter Notebook, VSCode, Google Colab and others. In this chapter, we focus on visualizations in python.

7.2.1 Common libraries for python visualizations

Below, we will preview two common python visualization libraries and learn how to leverage these methods to learn about our data and investigate trends within.

matplotlib

`matplotlib` is a Python library that provides a framework for generating and customizing visualizations. It supports various plot types, including line plots, scatter plots, bar charts, histograms, and more. `matplotlib` integrates well with other libraries, such as pandas and supports various output formats, including PNG, PDF, and SVG.

matplotlib.pyplot

The primary module from `matplotlib` that we'll use is `pyplot`. To access `pyplot` functionality, we need to import both the `matplotlib` library and the `pyplot` module.

```
import matplotlib.pyplot as plt
```

In the code above, we used dot notation where `matplotlib` is the library, `pyplot` is the module, and `plt` is the alias.

pandas

`pandas` provides built-in plotting functionality that allows you to generate visualizations directly from Series and DataFrames. It uses `matplotlib.pyplot` for plotting through the `.plot()` method. This offers a simple way to create basic plots, which can be further customized using `matplotlib` attributes such as `plt.title()`, `plt.xlabel()`, and `plt.ylabel()`.

```
import pandas as pd
```

7.3 Chart Types

The updated Star Wars dataset comes from the webpage Introduction to Data Analysis Using the Star Wars Dataset by Fabricio Batista Narcizo. This dataset includes information on 87 characters from the first seven episodes of the Star Wars movie saga (the Original Trilogy, the Prequel Trilogy, and The Force Awakens). The author documents the process used to update the Star Wars dataset with additional information from the Star Wars API (SWAPI).

The Star Wars dataset contains both categorical and numerical variables that describe each character. We will use this data to demonstrate data visualization in Python, specifically using the `matplotlib.pyplot` module and the `.plot()` function from `pandas`.

We will import Pandas and Matplotlib, load the Star Wars dataset, review the variables within, and create related visualizations.

```
import pandas as pd
import matplotlib.pyplot as plt

starwars = pd.read_csv("updated_starwars.csv")
starwars.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 87 entries, 0 to 86
Data columns (total 25 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   name         87 non-null     object
```

```
 1   height       86 non-null     float64
 2   mass         65 non-null     float64
 3   hair_color   82 non-null     object
 4   skin_color   86 non-null     object
 5   eye_color    87 non-null     object
 6   birth_year   50 non-null     float64
 7   birth_era    50 non-null     object
 8   birth_place  37 non-null     object
 9   death_year   62 non-null     float64
 10  death_era    62 non-null     object
 11  death_place  57 non-null     object
 12  sex          87 non-null     object
 13  gender       87 non-null     object
 14  pronoun      87 non-null     object
 15  homeworld    83 non-null     object
 16  species      87 non-null     object
 17  occupation   87 non-null     object
 18  cybernetics  7 non-null      object
 19  abilities    55 non-null     object
 20  equipment    62 non-null     object
 21  films        87 non-null     object
 22  vehicles     15 non-null     object
 23  starships    20 non-null     object
 24  photo        87 non-null     object
dtypes: float64(4), object(21)
memory usage: 17.1+ KB
```

7.3.1 Bar Plots

A univariate bar plot is a data visualization used to represent categorical data frequencies. The height or length of each bar (and the associated axis or bar labels) typically corresponds to the level frequencies. Bar plots can be used to compare categories and more generally to visualize a categorical distribution.

7.3.1.1 A (Star Wars) Example

Even though Star Wars took place in a galaxy far, far away, spanning multiple planets, space stations, and star systems, many of the main characters were human or droid.

```
# Count and display the number of characters for each species in the dataset.

starwars['species'].value_counts()
```

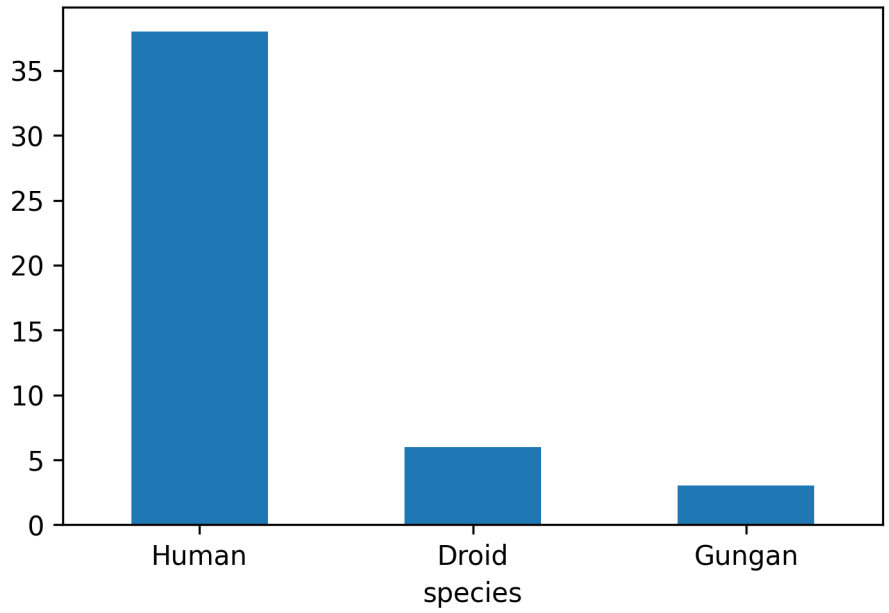
```
species
Human               38
Droid                6
Gungan               3
Mirialan             2
Kaminoan             2
Twi'lek              2
Wookiee              2
Trandoshan           1
Yoda's species       1
Hutt                 1
Rodian               1
Neimodian            1
Toydarian            1
Sullustan            1
Mon Calamari         1
Zabrak               1
Aleena               1
Vulptereen           1
Xexto                1
Toong                1
Cerean               1
Dug                  1
Ewok                 1
Iridonian Zabrak     1
Nautolan             1
Quermian             1
Tholothian           1
Kel Dor              1
Chagrian             1
Geonosian            1
Iktotchi             1
Clawdite             1
Besalisk             1
Skakoan              1
Muun                 1
Togruta              1
Kaleesh              1
```

```
Umbaran                1
Pau'an                 1
Name: count, dtype: int64
```

As seen from the output, `.value_counts()` returned the various species counts. The returned object is a `Series` with unique levels (species) as the index and the corresponding counts as values, sorted in descending order of frequency.

A bar plot with all 39 species represented, would likely break the "minimize clutter" rule, eventhough all of the related information may be important. In particular representing the levels with only one representative in the chart would not necessarily be the best representation. As an alternative, we could use data moves to create a new "other" category that contains the species with fewer than a cutoff count. Or, to represent that a majority of characters come from a few specific species, we can simplify the plot by displaying only the top three frequency categories. Below, we take the second approach.

```
starwars['species'].value_counts()[0:3].plot(kind = 'bar', rot = 0)
```



In the code above, `.value_counts()` tabulates the occurrences of each species in descending order of frequency, `[0:3]` filters for the top three entries, and `.plot(kind = 'bar')` generates the bar plot. The `rot = 0` parameter is an option that displays the axis labels horizontally.

Observing the graph and considering our filtering criteria, we now have a visual representation that shows that the Star Wars character species distribution is human-centric, despite the countless planets across a vast galaxy in which the saga unfolds.

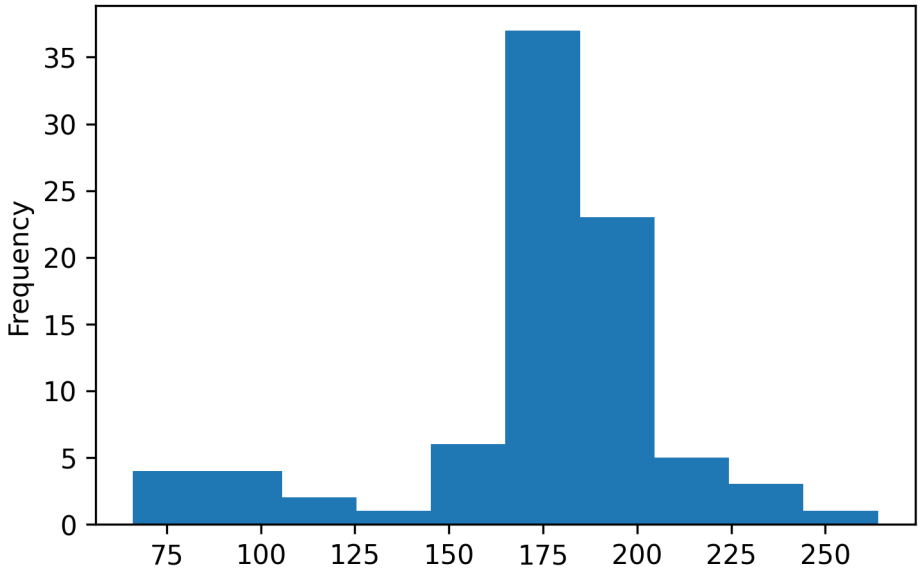
7.3.2 Histograms

A histogram is similar to a barplot, but depicts frequencies of binned numerical values vs raw categorical counts. In a histogram, the number of bins is a parameter that impacts the shape of the visualization, as where in a bar plot the number of bars usually corresponds to the number of categorical levels (barring transformation like the filtering we did above). Another common difference between univariate barplots and histograms is that histograms can be displayed using different vertical-axis scales, such as counts, relative frequencies, or densities. When shown on a density scale, the areas of the bars sum to 1 like a probability density function. In contrast, barplots typically display frequencies or proportions for categorical variables, with no regard to bin width or area.

Histograms can reveal characteristics of the data distribution, including patterns such as modality, shape, skewness, unusual observations and other features that may yield insight into the underlying properties of the variable under consideration.

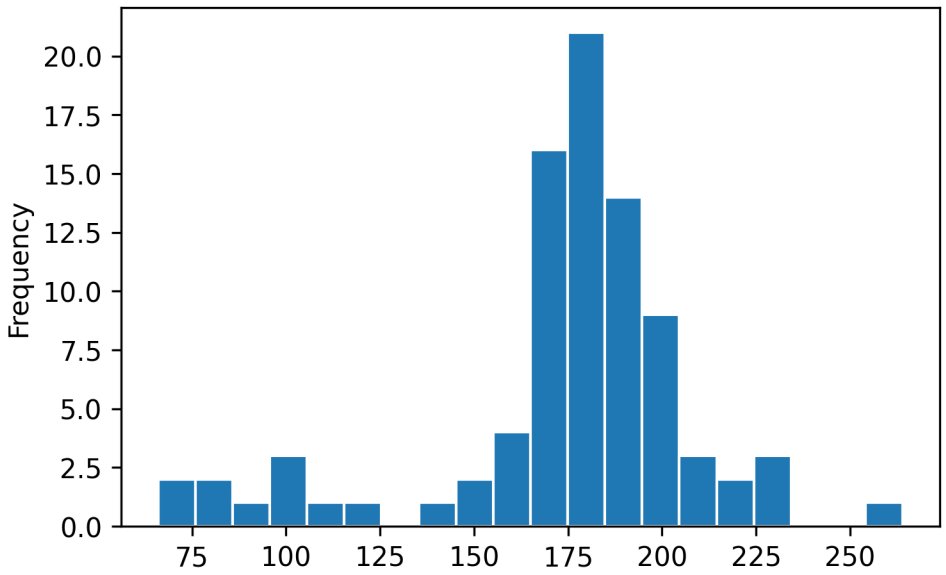
So now that we know about histograms, let's create one to get a better sense of the distributions of the characters' heights.

```
starwars['height'].plot(kind = 'hist')
```



The histogram captures most heights between approximately 160 and 200 centimeters. Additionally, the distribution appears to be unimodal and has a slight left skew where more extreme heights are for shorter characters.

```
# histogram w modified number of bins
# adding edgecolor helsp distinguish bins

starwars['height'].plot(kind = 'hist', bins = 20, edgecolor = 'white')
```



In the code above, we've adjusted the default `bins` parameter, which impacts the bin widths. Correspondingly the number of observations falling within a bin have decreased. In general, the choice of bins can impact the visualization and so interpretations can be subjective in this way. Thus, a histogram can be particularly useful in an exploratory phase of a data science workflow but may not be the best tool on which to base formal or inferential decisions. Nonetheless, in the modified histogram we see a more granular view of the distribution of the Star Wars characters' heights.

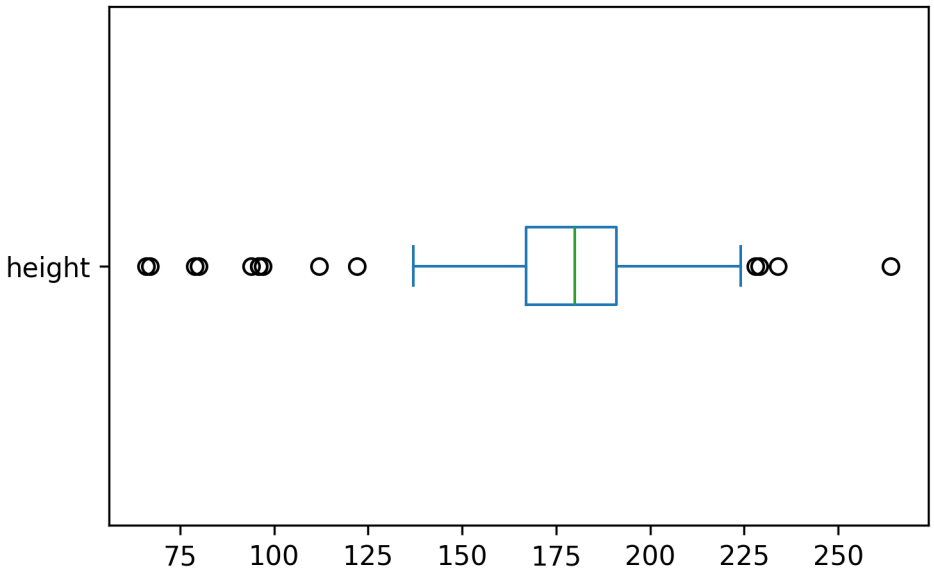
7.3.3 Boxplots

A boxplot is a visual representation of the "five number summary." Boxplots depict the distribution of numerical data using the five values, typically represented as a rectangular box and extended lines know as whiskers. The typical five numbers (corresponding to the five number summary) are the minimum, first quartile (Q1), median (Q2), third quartile (Q3), and maximum. The box spans from Q1 to Q3 and represents the middle 50 percent of the data, with a line inside the box marking the median (Q2 value). The whiskers extend from each side of the box and may include the smallest and largest values. Some boxplots have heuristic cutoffs for the whiskers, where the minimum and maximum values are not captured within this range. Matplotlib uses the 1.5 times the interquartile range (IQR) convention for

the whisker bounds, where the IQR is the distance between the first quartile (Q1) and the third quartile (Q3), when the minimum and maximum values extend beyond this range.

Like histograms, boxplots can depict skewness and provide visual information about potential unusual observations or values. Boxplots are often used to assess and compare variation and measures of center across groups.

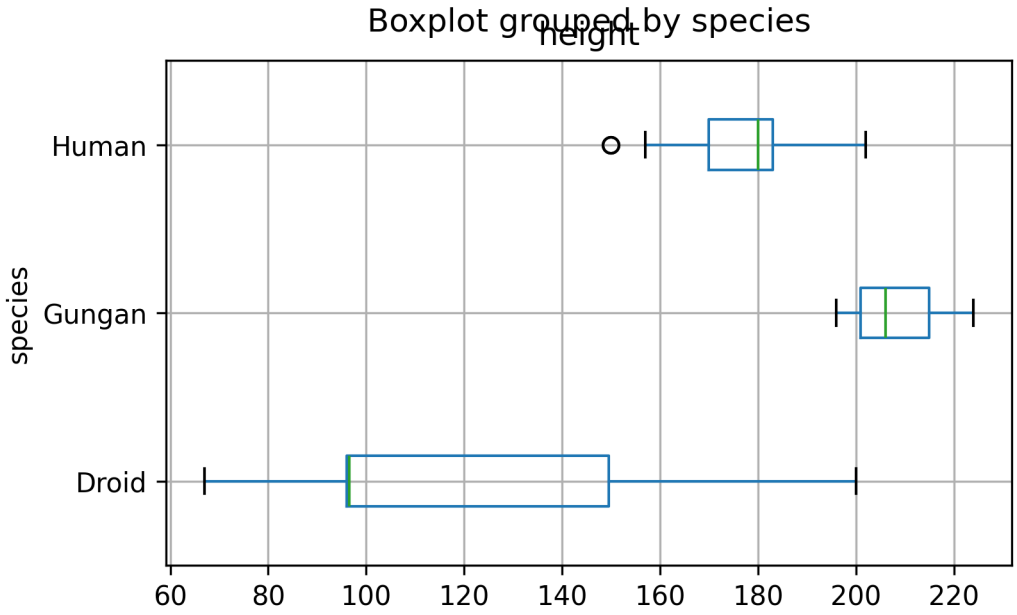
```
# A boxplot of heights
# NOTE: `vert=False` changes default display from vertical to horizontal.

starwars['height'].plot(kind = 'box', vert = False)
```



Like the histogram of the same variable, the boxplot above shows the distribution of the characters' heights, but also provides information about the distributional percentiles, or more specifically the quartiles. We can see that the median (Q2) height is approximately at 180 centimeters. We can also see that 50% of the characters have heights approximately between 165 (Q1) and 190 (Q3) centimeters. We can imagine overlaying this boxplot on the histogram and seeing the connections and similar information displayed by each visual.

Next, let's consider how heights might vary across human and other nonhuman character species. For this, we can filter the data using our most frequent species classifications, Human, Droid, and Gungan. Then we can generate side-by-side boxplots for each group to gain insight into and compare the distributions of heights across these species.

```
# Define a list with the species to include
species_groups = ['Human', 'Droid', 'Gungan']

# Create a Boolean mask to filter for those species
mask = starwars['species'].isin(species_groups)

# Create a horizontal box plot of height by species
starwars[mask].boxplot(column = 'height', by = 'species', vert = False)
```



In the code above, the `.isin()` method (from `pandas`) checks whether each value in column is found within the specified list. For each column entry, if in the list the method returns True and otherwise, False. This result is a Boolean Series that can be used to filter the rows of a DataFrame.

Returning to the visual, we can see that the distribution of heights across the three different species are very different. For example, there is almost no overlap between the heights represented by the boxplots. These differences were hidden in the boxplot and histogram for the entire set of characters. This highlights the need to also consider relationships between variables and how these associations might influence certain observation and interpretations.

However, as we noted in observing the counts of each species, we should keep in mind the frequencies that are represented by the graphs. In particular, there are only three Gungans in the dataset. So, although the boxplot representing Gungan heights has the typical features, the underlying data from which it was generated consists of only a minimum, median, and

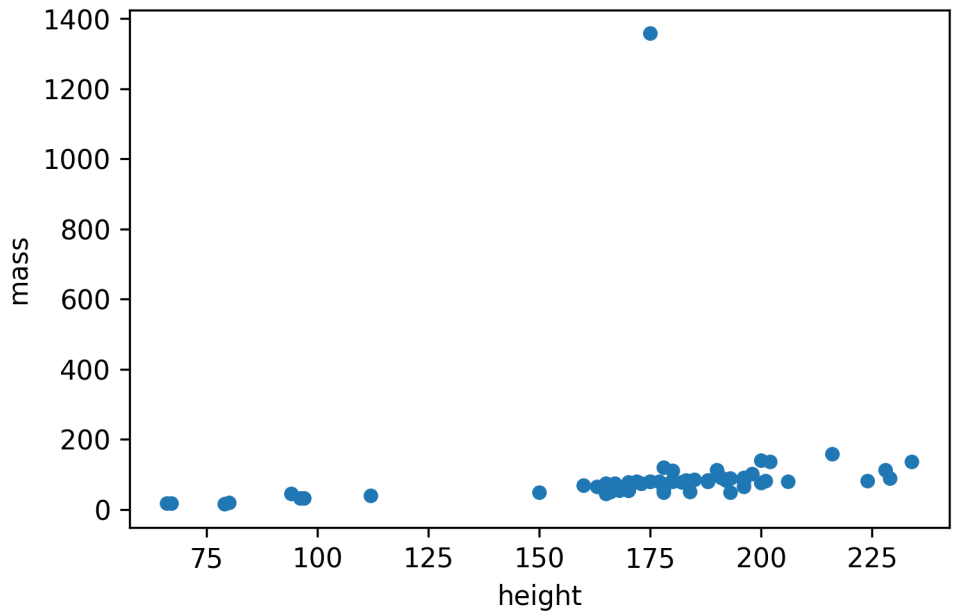
maximum. Similarly, the Droids are represented by a mere 6 entries, which does not make for an interesting (or useful) five number summary. So, once again, we see that context is paramount in the interpretations of these visuals and the degree of confidence that you may want to assign to them. In fact, you may have seen other droids in The Mandalorian!

7.3.4 Scatterplots

Bivariate scatterplots depict the ordered pairs of two numerical variables of interest as points on a coordinate plane. These visualizations can be used to investigate potential relationships between two variables and can be useful in visualizing patterns such as trends, clusters, and associations. Scatterplots can also show unusual observations and are a standard tool in many model diagnostic procedures, among other uses.

We can use a scatterplot to visualize the relationship between the Star Wars characters' heights and masses.

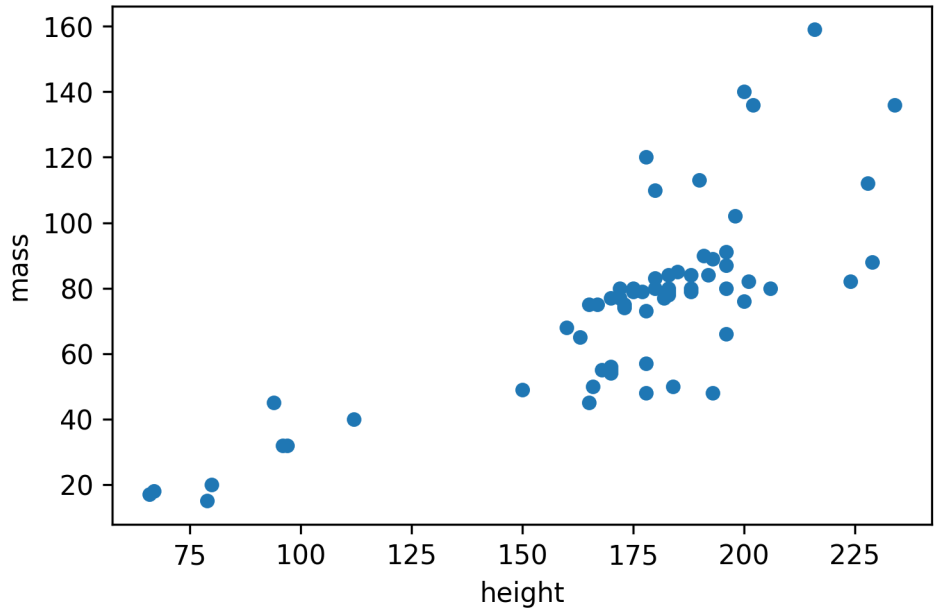
```
# A scatterplot of height vs. mass

starwars.plot(kind = 'scatter', x = 'height', y = 'mass')
```



From the graph, there appears to be a linear relationship between mass and height. However, what may be an increasing trend is obscured by the horizontal access scale. This is due to the observation with an extreme mass value close to 1400 kilograms. Due to the scale distortion

resulting from this single observation, we might want to examine how the picture changes if we were to exclude it.

```
# Find the highest mass in the dataset

max_mass = starwars['mass'].max()

# Create a Boolean mask that removes the character with the highest mass

mask = starwars['mass'] != max_mass

# Create a scatter plot without the outlier

starwars[mask].plot(kind = 'scatter', x = 'height', y = 'mass')
```



Upon excluding the observation with the extreme mass value and re-plotting the points, the suspected increasing trend (i.e., as height increases, mass increases) is more clear. Now that the variation in mass is more discernible via the new visualization, we might even want to consider more dynamic models (e.g., curvilinear), that might better explain the relationship between characters' heights and masses.

7.3.5 Line Charts

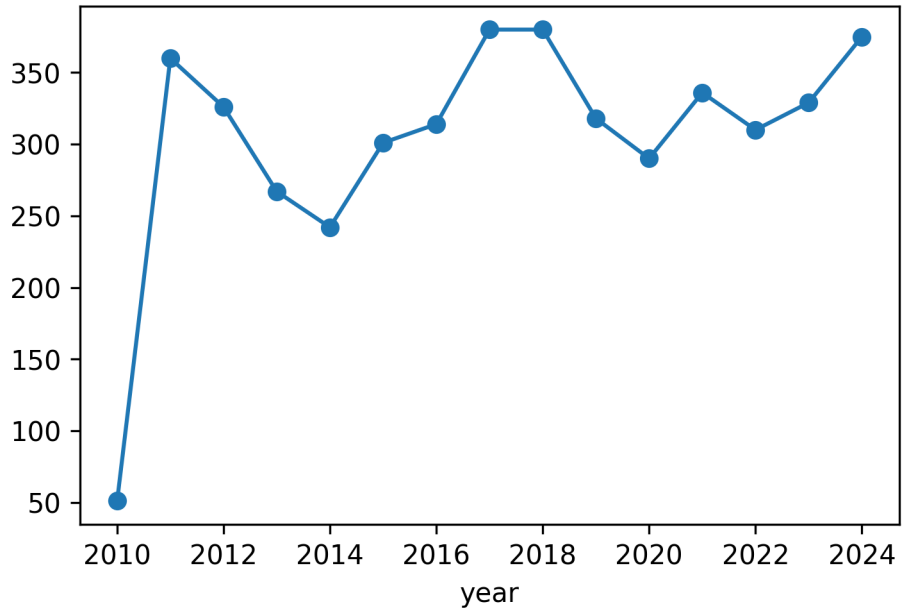
A two-dimensional line chart is a visual representation of data points (ordered pairs) connected by straight lines. These displays typically show numerical changes over an ordered interval. The horizontal axis is typically a numerical variable, a sequence, or an ordered set of meaningful categories.

Line charts are often used to observe series and changes with respect to time. Line charts can also help visualize non-temporal directions of change with respect to categories like "beginner", "intermediate", and "advanced"; "low", "medium", "high" or other groupings with various levels. The points in the visualization can represent single observations or summary statistics, such as category/group means, corresponding to the associated horizontal axis labels.

7.3.5.1 Line Chart Example

The Ramen Rater dataset consists of review beginning in 2002, but only has records of dates from 2010 onward. To examine how the frequency of instant noodle reviews has varied over the years we can use the `ratings` dataset (read in below). This dataset was created by extracting review numbers and corresponding dates from the Ramen Rater website. Unlike the Big List, which does not include date information, the ratings dataset enables grouping by year and allows for counting the number of reviews completed annually. This approach offers insight into how the Ramen Rater's review activity has changed over time.

```
# Load the Ramen Rater review data

ratings = pd.read_csv("ramen_reviews.csv")

# Group the data by year

grps = ratings.groupby('year')

# Create a line chart, excluding the last year (2025)

grps.size()[:-1].plot(kind='line', marker='o')
```



The line chart above shows changes in the number of reviews over time. Specifically, we can observe a sharp increase from 2010 to 2011 from approximately 50 reviews to over 360 reviews. For the remainder of years (2012 - 2025), the number of reviews fluctuate between 250 and 390. In general, we can see a relatively consistent recording of reviews over the years.

7.3.6 Customizing Visualizations

The `matplotlib` library provides a wide range of additional visualization customization options, including feature options such as adding titles and axis labels. Additional options, including some used above, include customized orientations, sizes, shapes, and colors. To explore the extensive list of customization options consult the Matplotlib documentation and consider leveraging the power of generative AI tools, too!

Next, we return to R to explore more visualizations with an eye towards exploration and communication.

8 Project Part 4.2: Visualizations & Trends in R

Now that we've explored data visualizations in Python, we can learn how to create dynamic data visuals in R, as well, including a few additional multivariate representations.

8.1 Base R plot features

Base R has a plot function, which for standard plots is very easy to use. To illustrate the utility of the base R plot function in connection with the communicative power of data visualizations, let's look at an example after a brief presentation of some background information.

8.1.1 Traffic Accidents - Example

The traffic accidents data set contains data compiled from the National Highway Traffic Safety Administration for incidents occurring in 2020. Variables range from vehicle information to characteristics of the drivers & passengers. The data, by itself, is a collection of various values and other bits of information. We could summarize this content with statistics, such as means and frequencies. However, we could consider another way to get a global summary of the data through creating a simple visualization. In particular, when data is related to location, and this information is available in the dataset, there is an opportunity to gain insights into data characteristics by way of a spatial representations.

Below, we will leverage the location information in the data in conjunction with the base R `plot()` function to demonstrate the power of data visualization and the ease of creating these in R. Furthermore, we'll consider some avenues of additional inquiry inspired by the visualizations of the data.

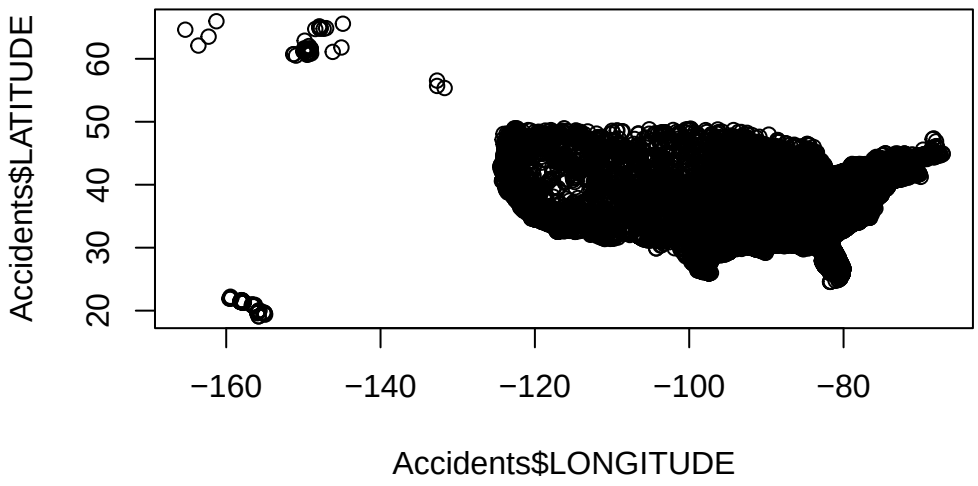
One variable in the Traffic Accidents data, `STATENAME`, contains information on the state in which a particular accident occurred. Because the data may have multiple rows representing different people associated with a particular accident we need to use the `ST_CASE` variable - a unique identifier for each accident - to filter the data to one observation per accident. This will allow us to get the frequencies of accidents while avoiding over-counting accidents with multiple entries in the dataset. The table below represents accurate counts of accidents per

State	Accidents	State	Accidents
Alabama	851	Missouri	911
Alaska	53	Montana	190
Arizona	965	Nebraska	217
Arkansas	585	Nevada	293
California	3555	New Hampshire	98
Colorado	573	New Jersey	547
Connecticut	279	New Mexico	364
Delaware	103	New York	963
District of Columbia	34	North Carolina	1412
Florida	3097	North Dakota	95
Georgia	1520	Ohio	1151
Hawaii	81	Oklahoma	596
Idaho	188	Oregon	461
Illinois	1086	Pennsylvania	1058
Indiana	815	Rhode Island	66
Iowa	304	South Carolina	962
Kansas	382	South Dakota	132
Kentucky	709	Tennessee	1119
Louisiana	760	Texas	3517
Maine	151	Utah	256
Maryland	540	Vermont	57
Massachusetts	327	Virginia	794
Michigan	1010	Washington	524
Minnesota	369	West Virginia	249
Mississippi	687	Wisconsin	561
		Wyoming	113

state, based on the data, obtained through the filtering described here in addition to other data moves.

Notice that both Alaska and Hawaii are included in the states for which accidents were recorded. Now, let's take a look at how we can create a visual of this table using the base R `plot()` function.

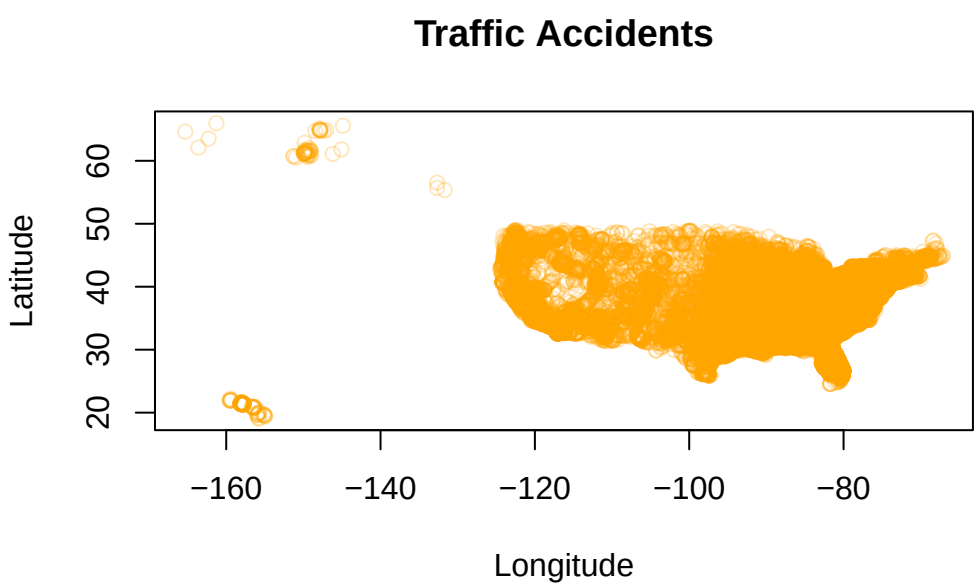
```
# the base R plot function takes simple
# (x,y) coordinates to generate a plot

plot(Accidents$LONGITUDE, Accidents$LATITUDE)
```



By simply adding two numeric variables into the plot function we were able to create a relatively informative visual of the data. For example, we could look at this graph and use our prior knowledge of maps and geography to infer that the data represent events that occurred in the United States. Knowing the context of the graph, we could also say that car accidents occur in almost all parts of the U.S. In general, with location data we may be able to relate to the context through examining our hometown or current location (e.g., North Carolina) if, or when, represented.

On the other hand, there is so much room for improvement. For this, we can even implement many graphical enhancements with the basic plot function through adding additional parameters and by also considering our visualization guidelines from the previous chapter.

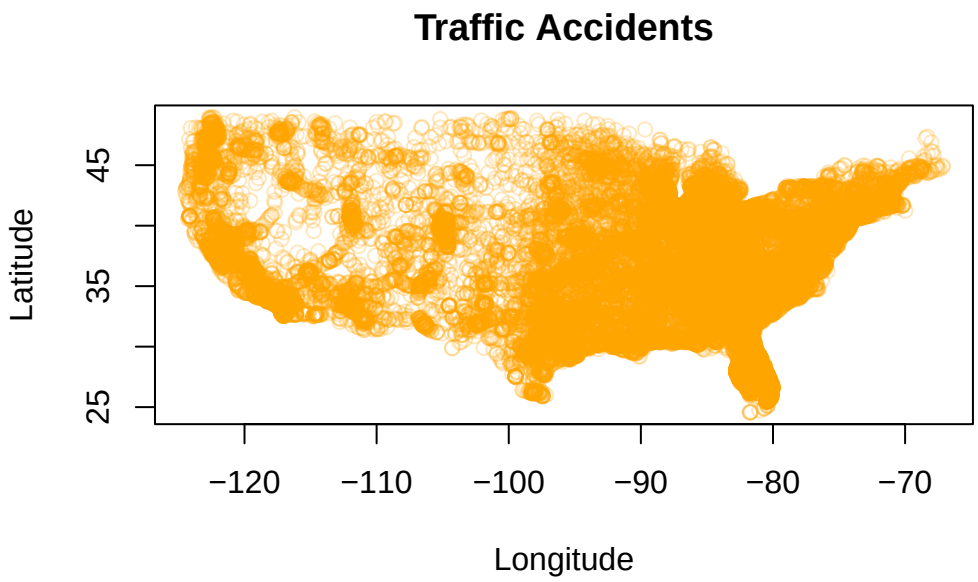
```
# plot with labels, color, and transparency
plot(Accidents$LONGITUDE, Accidents$LATITUDE,
     main = "Traffic Accidents", # main title
     xlab = "Longitude", # x-axis label
     ylab = "Latitude", # y-axis label
     col = alpha("orange",0.25), # specifies a color and transparency value
     pch = 1) # pch = 1 makes open circles
```



In the visualization above, we have gotten rid of the unnecessary clutter on our axis labels, added an informative title, and also added transparency to our (now orange) data points. The transparency also revels more information about the density of the accidents with respect to latitude and longitude.

In our guidelines, we also talked about how graph scales are important. Although the graph scales can be customized, the default scale ranges in R for numeric data depend on the ranges of the data points themselves. These ranges can have major influences on the usefulness and accuracy of a graphical representation. This influence is sometimes discovered or encountered in situations where there are one or more outliers in the data. For example, if most data is within a range of 1 to 10, and one data point is at 200, the corresponding visual will not display any variation in the majority of what could be your highly variable data. In this case, the differences between data points in the 1 and 10 region would not be distinguishable on a scale that ranges up to 200.

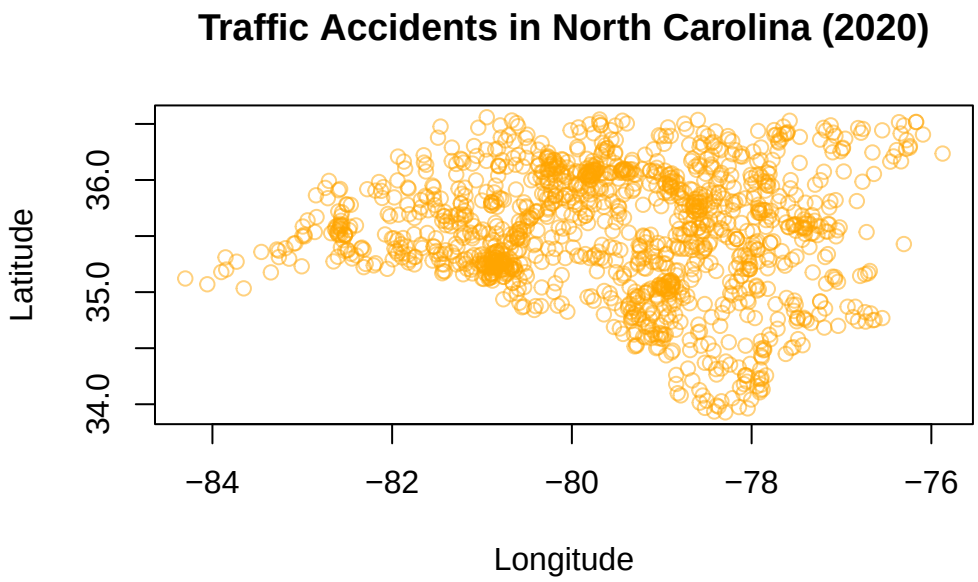
As with the hypothetical example above, although the traffic accidents data does not have what we would necessarily classify as outliers, including the accidents in Hawaii and Alaska impacts the graph in a similar, but less extreme, way. For an analysis including all states, we may have a need to display all of the related data. But let's see how things change if we restrict the data to the continental U.S.



With the graph rescaled to only include data from the continental U.S., we can now more readily visually discern accidents density variation across the (continental) U.S. As noted in the code comments, we specified the graphing coordinates criteria using the model formula operator ~ to generate the graph… because we can (and because there are so many ways to do a thing in R!).

Finally, let's create one last graph to Zoom in a bit on the home state of this book, North Carolina.

```
# NC subset
Accidents_NC <-
  Accidents %>%
  filter(STATENAME == "North Carolina")

# NC graph, accidents locations
plot(LATITUDE ~ LONGITUDE, data = Accidents_NC,
     main = "Traffic Accidents in North Carolina (2020)",
     xlab = "Longitude",
     ylab = "Latitude",
     col = alpha("orange",0.5),
     pch = 1)
```



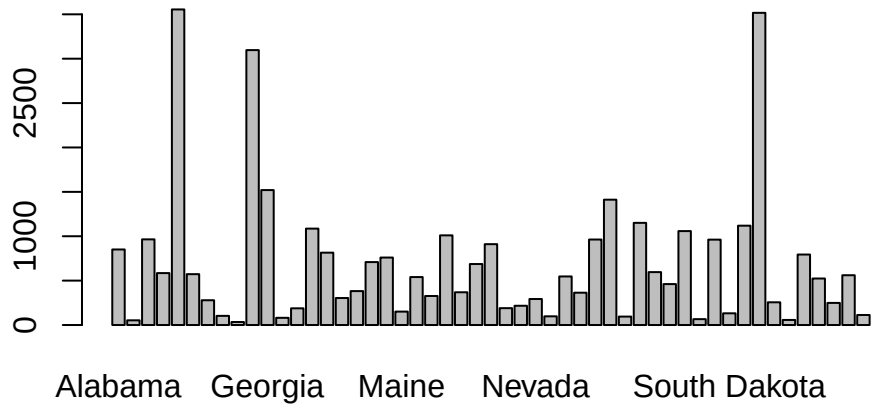
Pretty cool, right! In general, in base R:

- The default graph generated for two numeric variables from a dataset is a scatterplot.

As seen above, scatterplots can be particularly informative when they're related to location information and prior knowledge, but definitely have more general utility for various appropriate data types.

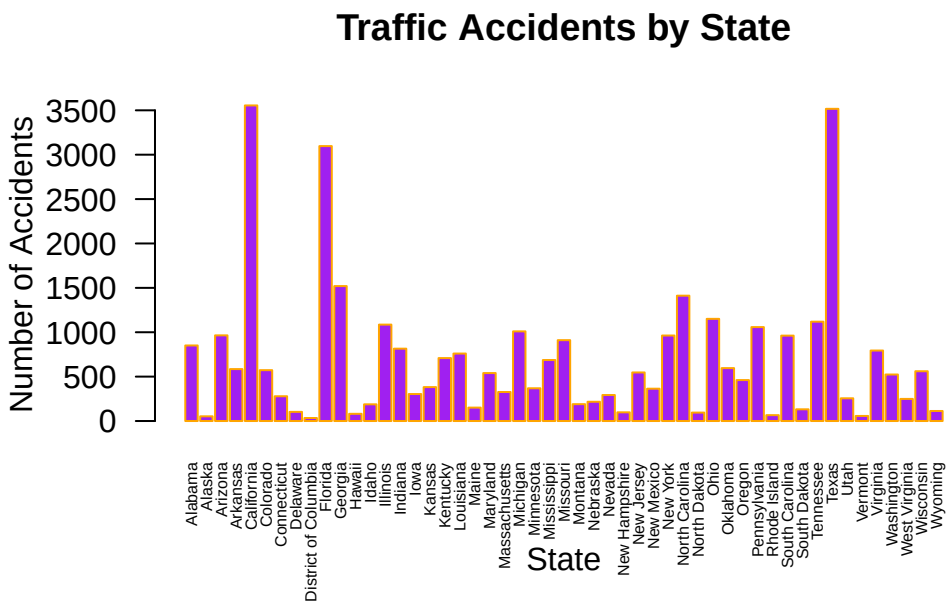
Now, what about categorical variables in base R?

```
# one variable, categorical
plot(Accidents$STATENAME) # barchart
```



Using the base R `plot()` function, the default graph for one categorical variable is a barchart. And now, we enhance.

```
plot(Accidents$STATENAME,
        col = "purple", # fill bar colors w purple
        border = "orange",  # Color bar borders orange
        las = 2,    # Rotate x-axis labels vertically
        cex.names = 0.5,  # Shrink x-axis labels
        main = "Traffic Accidents by State",
        xlab = "State")
# Adjust y-axis label to be horizontal
mtext("Number of Accidents", side = 2, line = 3, las = 0)
```



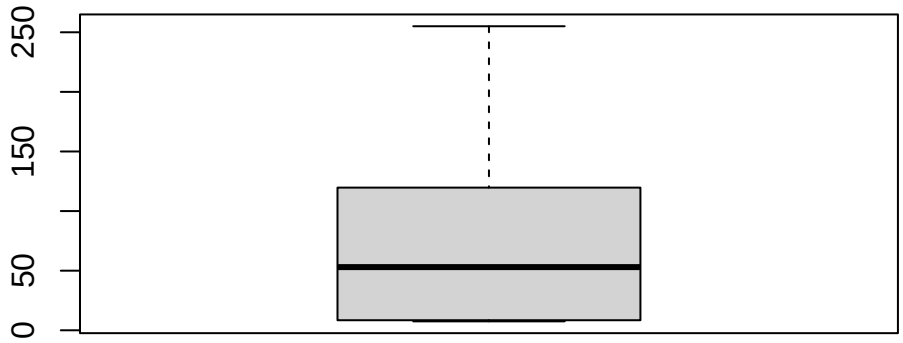
Based on the graphical representation, we might infer that the number of accidents in states are correlated with the respective state populations, or number of drivers within a state. Although that may be an obvious starting point, we might want to see if the proportion of drivers in each state is significantly different from the proportions of drivers, perhaps with a plan to investigate why potential differences may exist.

For two categorical variables as inputs into the base R plot function, the graph that is generated is what is known as a `mosaic plot`. Although this can be generated in the same way as previous examples, through the base R `plot()` function, we will use more useful packages to create such a visual where we can more easily modify the labels and colors to create an informative display.

8.2 Other common plots

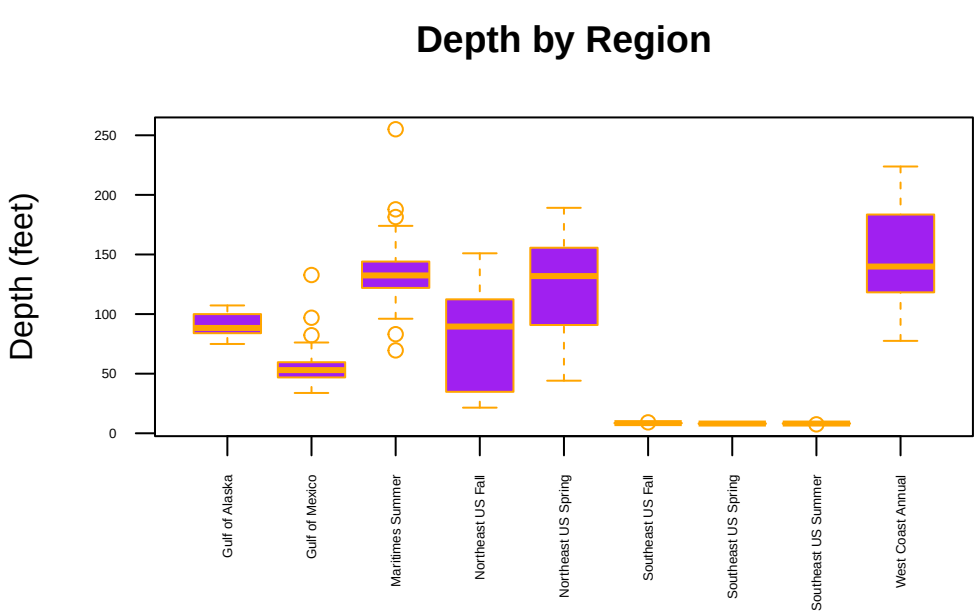
In R, boxplots and histograms are just as easy to implement. Let's look at the syntax for these visuals through a few examples.

```
boxplot(Marine6$Depth) # distribution of depth
```



We could also create multiple boxplots for comparison by using model syntax with one numeric and one categorical variable (e.g., to compare distributions of depth by region).

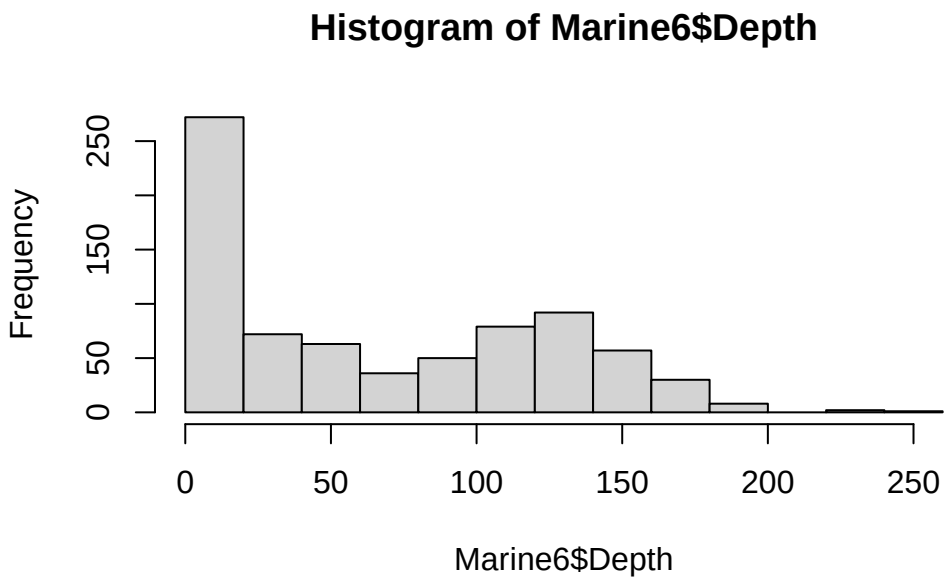
```
boxplot(Depth~Region, data=Marine6,
        col = "purple", # fill bar colors w purple
        border = "orange",  # Color bar borders orange
        main = "Depth by Region",
        cex.axis = 0.45, #shrink axis labels
        xlab = " ",
        ylab = " ",
        las = 2 # Rotate x-axis labels vertically
       )
mtext("Depth (feet)", side = 2, line = 3, las = 0)
```



From this plot, we might want to investigate the factors that contribute to the visually evident differences in depth across the region.

Finally, to generate a histogram we can use the function `hist()` in the code below.

```
hist(Marine6$Depth)
```



As with the other plot functions in base R, we could modify the histogram to include more bins, various colors, labels, and other features. However, when we're interested in customizing visualizations in R, the package to turn to is `ggplot2`!

8.3 ggplot2!

One of the best data visualization packages in R, and in general, is ggplot2 from the tidyverse. The ggplot2 syntax allows for building a variety of standard and customized graphs through layers. Similarly to features available in base R, the methods for visualizations built through `ggplot2` include aesthetics such as groups, colors, opacity, size and more - with the bonus of minimal code and access to excellent documentation. As with other libraries within the tidyverse, ggplot2 has a very useful resource that can be found here: ggplot2 Cheat Sheet.

With the ggplot2 package, we will expand our visualizations to show examples of useful multivariate graphs, including mosaic plots and spaghetti plots.

8.3.1 Examples with ggplot2

The Rising Global Temperature dataset consists of three variables, `gistemp`, `year`, and `ci95`.

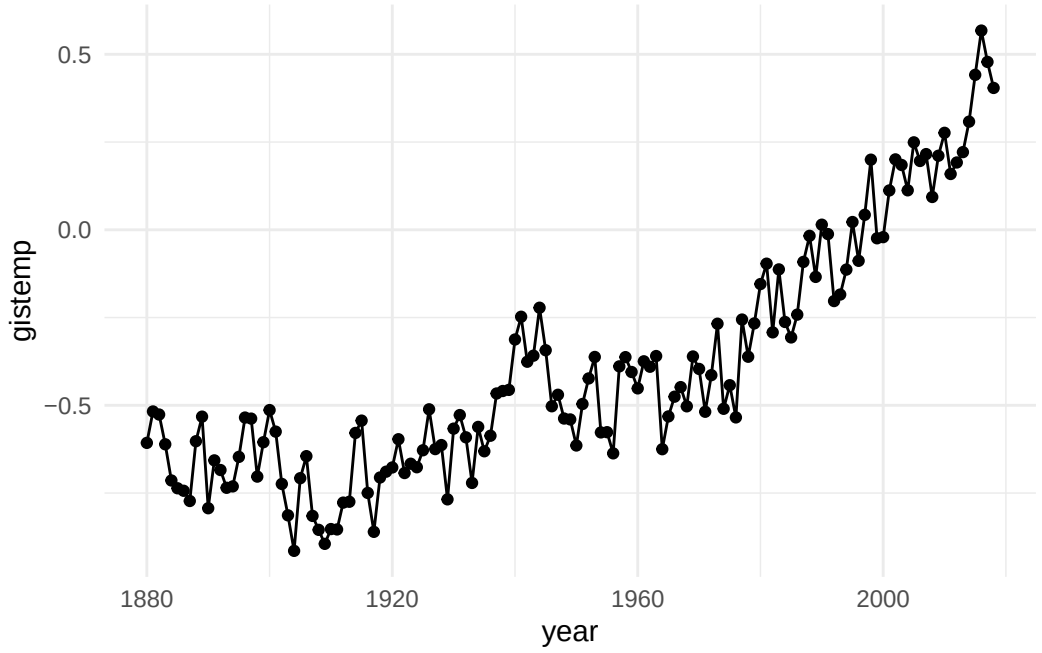
- `gistemp` represents yearly-average temperature differences from the average of 1986 - 2005, for each year including and between 1880 to 2018.

For example, a `gistemp` measurement of 0.2 for a given year would represent a yearly average temperature of 0.2 degrees Celsius warmer than the average temperatures from 1986 - 2005.

- `year` represents the year in which the temperatures were taken, and
- `ci95` represents the 95% confidence interval margins of error.

Below, we'll use the `ggplot2` package to visualize the temperature changes over time.

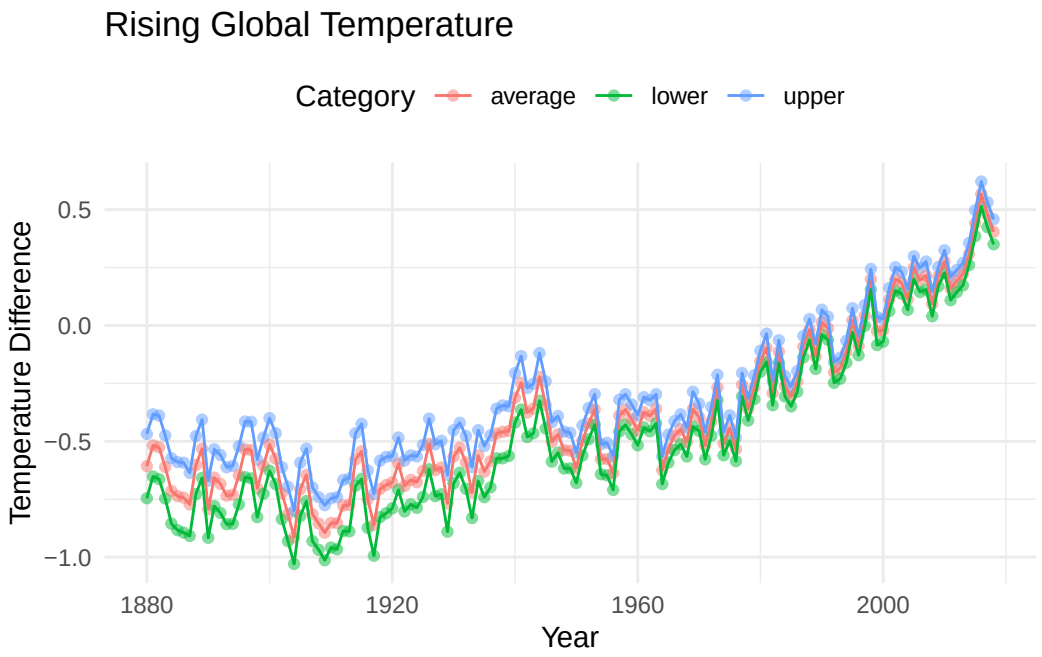
```
# RGT is the Rising Global Temperature dataset
# the aes() function indicates the variables to be plotted w.r.t. the axes
# geom_point() adds the points layer to the visual
# geom_line() adds the line layer to the visual
# theme_minimal is an option that minimizes background items, such as grid lines
# NOTE, each layer is added/connected to the next through the `+` operator

RGT %>%
  ggplot(aes(x = year, y = gistemp)) +
  geom_point() +
  geom_line() +
  theme_minimal()
```



From this plot, we can discern a generally increasing trend in the temperature differences over time. Patterns revealed here may not be so obvious in numerical summaries alone, especially in

terms of concisely communicating or presenting the trend to a general audience. Considering communication efficacy, we may see some opportunities to add some customization to our graph to enhance the quality and consider additional information in the dataset. For example, we could visualize the confidence interval information that accompanies each year measured, make our axis labels more informative and add color. We do this in the following visual.

```
# The alpha parameter adjust the opacity of the corresponding geom
# We can improve our various labels through the labs() function

longtemp %>%
  ggplot(aes(x = year, y = temperature_difference, color = category)) +
  geom_point(alpha = 0.5) +
  geom_line() +
  labs(x = "Year",
       y = "Temperature Difference",
       color = "Category",
       title = "Rising Global Temperature",) +
  theme_minimal() +
  theme(legend.position = "top")
```



Note that we referenced a dataset called `longtemp` in the last code block. This dataset was derived from `RGT` via a series of data moves that included creating new variables and creating hierarchy through nesting upper bounds, lower bounds, averages, and the corresponding category information within each year (although explicitly repeated in the dataset).

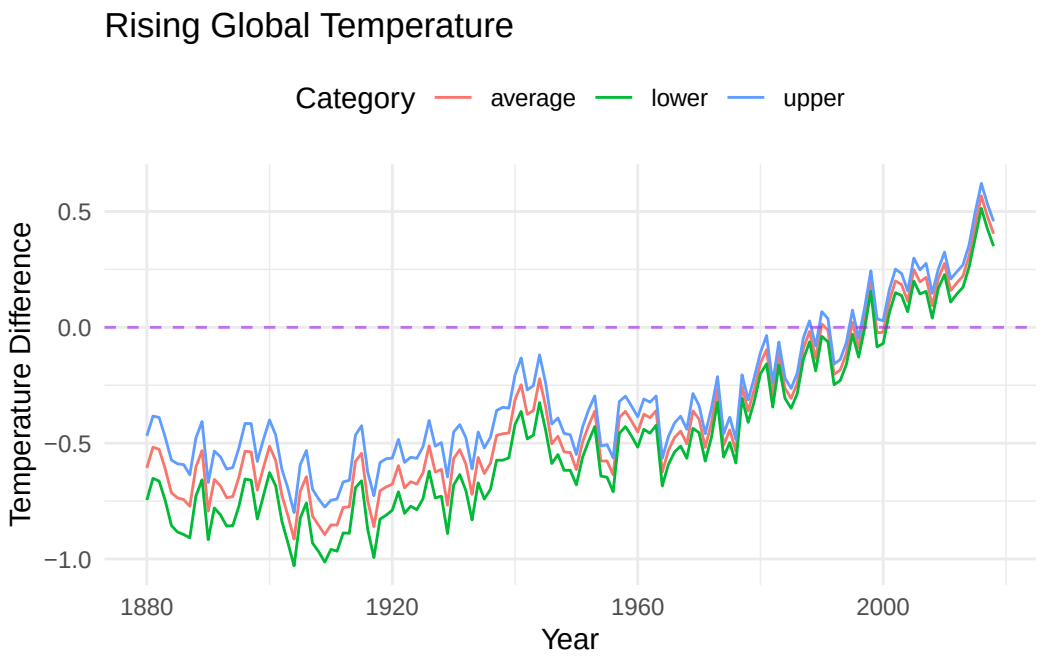
Here is an example of the new data structure needed for the graph.

```
# A tibble: 6 x 3
   year category temperature_difference
  <dbl> <chr>                      <dbl>
1  1880 average                   -0.607
2  1880 upper                     -0.468
3  1880 lower                     -0.746
4  1881 average                   -0.518
5  1881 upper                     -0.383
6  1881 lower                     -0.652
```

The graph corresponding to `longtemp` is also an example of a `spaghetti plot`, which is an extension of a line plot to include group (or category) information. These graphs are often used to depict longitudinal data but can be useful for conveying many group-based patterns.

Next, we make a few adjustments to present the trends in the Rising Global Temperatures data with a little less clutter (by removing the points), and with an additional line at our comparison average (where the temperature change is zero).

```
longtemp %>%
  ggplot(aes(x = year, y = temperature_difference,
             color = category)) +
  #geom_point(alpha = 0.5) +  (commented out, easy...)
  geom_line() +
  labs(x = "Year",
       y = "Temperature Difference",
       color = "Category",
       title = "Rising Global Temperature",) +
  theme_minimal() +
  theme(legend.position = "top") +
  geom_hline(yintercept = 0, linetype = "dashed", color="purple", alpha = .6)
```



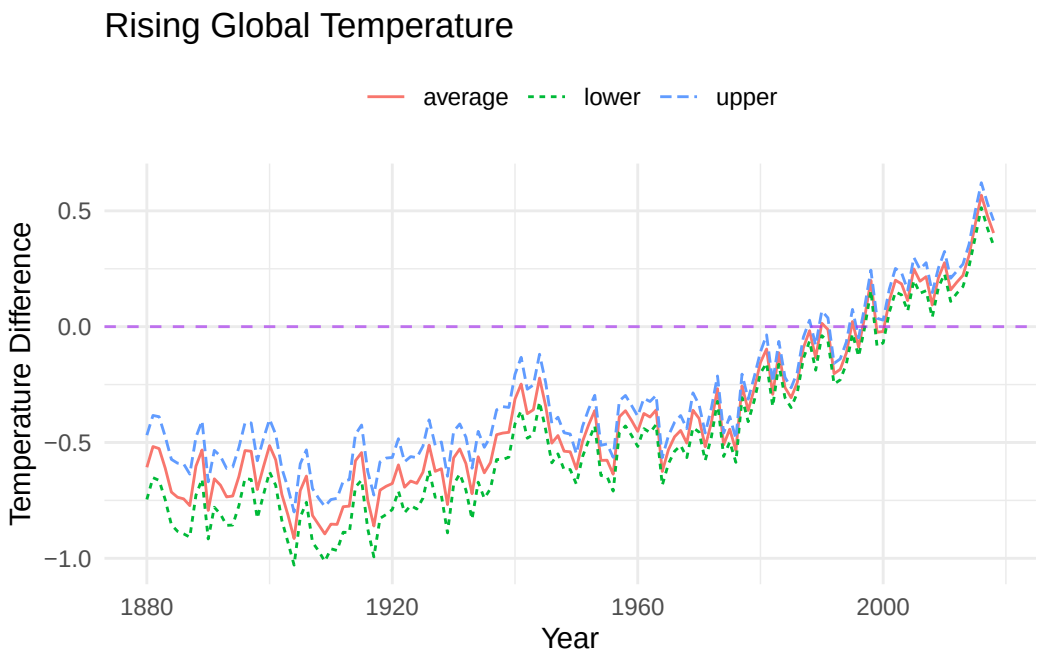
As you can see by comparing the code blocks, modifying visualizations through ggplot is quite convenient. In particular, we can overlay additional features by simply adding the relevant geom to the visualization.

Design Practices

When you create data visualizations, there are many choices that can be made. Considerations for user experience, for example, can help to guide your choices. These may include choosing accessible color palettes and adding descriptive text for a given visualization. Compare the code and corresponding images from the visuals above and below. How do you imagine the two different graphs might impact perception?

```
longtemp %>%
  ggplot(aes(x = year, y = temperature_difference,
             color = category, linetype = category)) +
  #geom_point(alpha = 0.5) +  (commented out, easy...)
  geom_line() +
  labs(x = "Year",
       y = "Temperature Difference",
       color = "Category",
       linetype = "Category",
       title = "Rising Global Temperature",) +
  theme_minimal() +
```

```
  theme(legend.position = "top",
        legend.title = element_blank()) +
  geom_hline(yintercept = 0, linetype = "dashed", color="purple", alpha = .6)
```



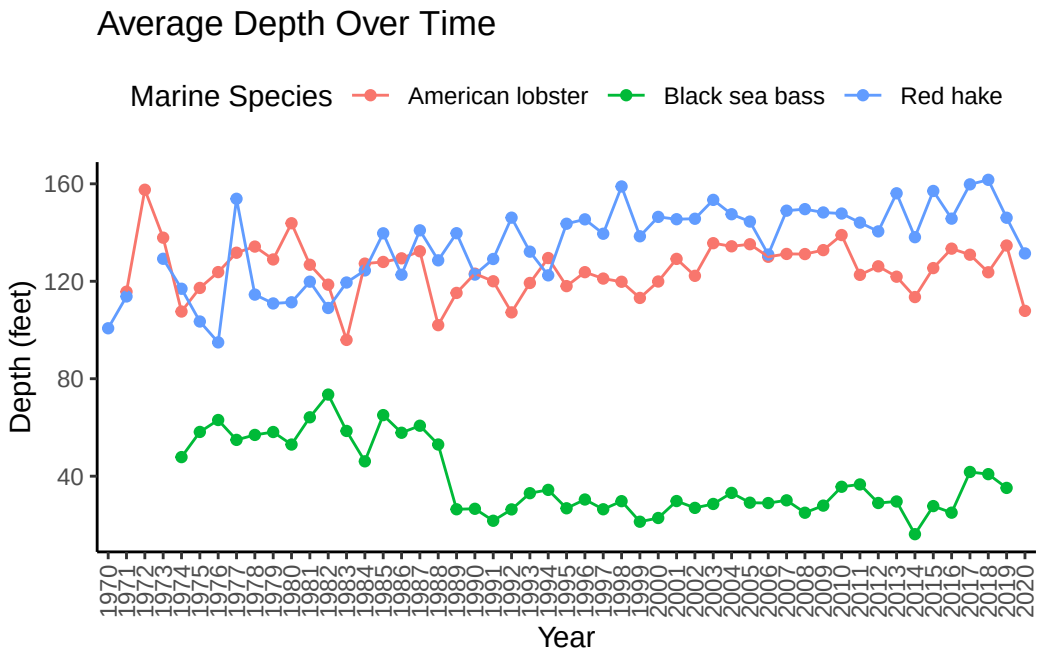
8.3.2 Faceting

As an alternative to plotting multiple groups on one graph, we may want to compare groups on the same scale on different graphs within the same visualization. We have a convenient way to do this through adding `facet` options. We can see an example of this using our familiar `Marine6` dataset.

For the following visualization, we used data moves (e.g, filtering, grouping, creating new variables) on the Marine6 dataset to calculate the average depth of the marine species for each year, and created a new dataset called `AvgDepth_Year`, referenced below.

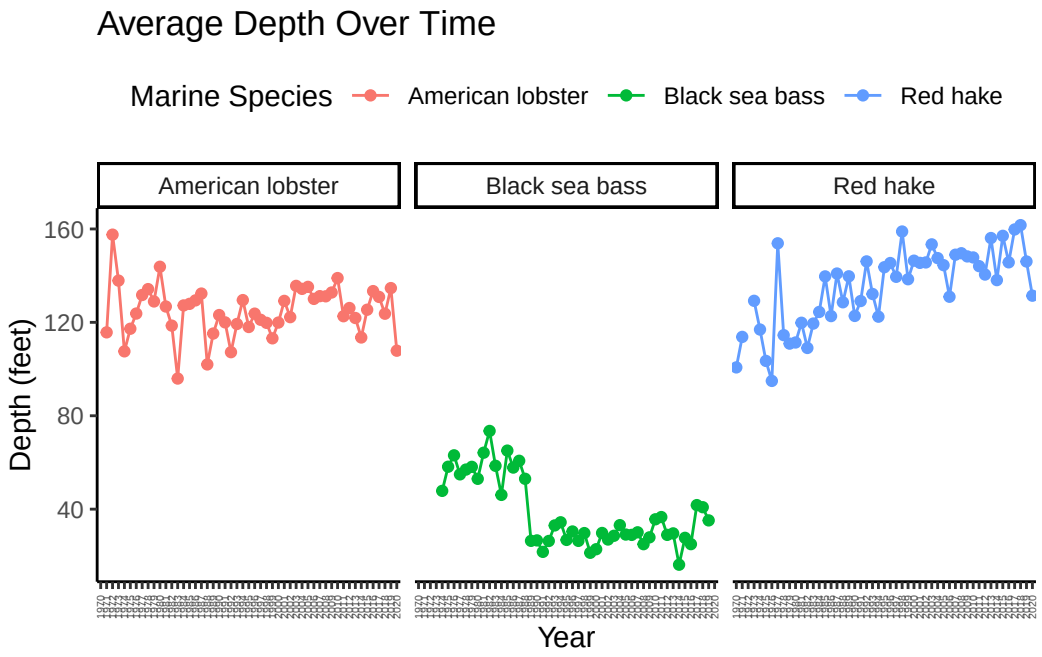
```
AvgDepth_Year %>%
  ggplot(aes(x = Year, y = `Average Depth`,
             group = `Marine Species`, color = `Marine Species`)) +
  geom_point() +
  geom_line() +
  theme_classic() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1),
```

```
        legend.position = "top") +
  labs(title = "Average Depth Over Time", y = "Depth (feet)")
```



Now, let's employ the facet option to see what this can do.

```
AvgDepth_Year %>%
  ggplot(aes(x = Year, y = `Average Depth`,
             group = `Marine Species`, color = `Marine Species`))  +
  geom_point() +
  geom_line() +
  facet_wrap(~ `Marine Species`) +
  theme_classic() +
  theme(axis.text.x = element_text(size = 4, angle = 90, vjust = 0.5, hjust=1),
        legend.position = "top") +
  labs(title = "Average Depth Over Time", y = "Depth (feet)")
```



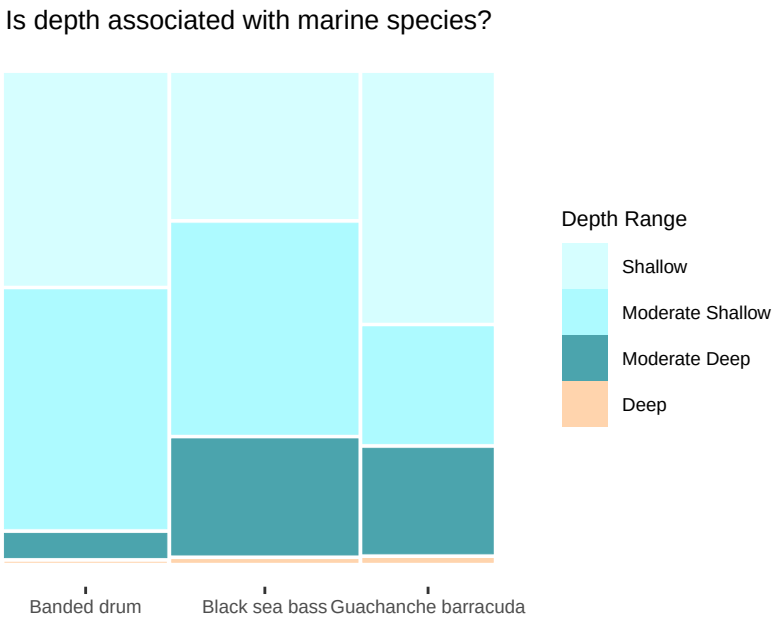
8.3.3 Categorical Example

Finally, let's check out an example of a useful plot for two categorical variables, a `mosaic plot`. Mosaic plots can yield insights into patterns of association, bivariate frequencies and more. Although this can be done in base R, the `ggmosaic` package is designed for this purpose and uses the same useful structure as `ggplot2`.

In the visual below, we have created a dataset `Mar6` from the `Marine6` data via filtering and creating new variables. `ggmosaic` enters the ggplot syntax as a geom object with parameters as seen in the code below. Following the addition of the mosaic geom, we customized the graph features to capture the features of interest in a concise and quality way. You may even notice a potential association from examining the visual. Could this visual help you tell a data story?

```
library(ggmosaic)

# let's visualize a contingency table...

# NOTE, ggmosaic requires a base R data.frame() object until it is updated to accommodate til

Mar6 %>%
  data.frame() %>%
```

```
ggplot() +
geom_mosaic(aes(x = product(Common.Name), fill = Depth_Range)) +
theme_mosaic() +
scale_fill_manual(values = c("#FFCA99","#1E8E99","#99F9FF","#CCFEFF"),
                  guide = guide_legend(reverse = TRUE)) +
labs(
  x = " ",
  y = " ",
  title = "Is depth associated with marine species?",
  fill = "Depth Range") +
theme(
  axis.text.y = element_blank(),
  axis.ticks.y = element_blank(),
  axis.text.x = element_text(size = 7),
  axis.title.y = element_text(size = 8),
  legend.title = element_text(size = 8),
  legend.text = element_text(size = 7),
  plot.title = element_text(size = 10, hjust = 0.5)
)
```



Data Moves for Data Viz

Data moves often go hand in hand with preparing data for visualizations. In this chapter, each visualization was preceded by data processing steps so that the information we wished to display was available in the right format for the generating functions. Understanding your data and related data manipulation processes (e.g., data moves) are essential to creating informative data depictions. Here, once again, we see (and visualize) the interconnectedness of the data science workflow!

9 Project Part 5: Communicating results

In the previous chapters we've gone from explaining our data and the associated metadata - via our data dictionaries - to diving into our data utilizing data moves. These pre-processing workflow steps served as a means towards grounding our investigations of interest, subsequently supported by the powerful tools of data visualization. Now, we are tasked with a culminating step in our workflow - communicating results.

To accomplish this final milestone, let's consider a few guidelines, criteria, and programming features that we can leverage to help with bringing the components of our workflow together as a presentation.

9.1 Presentation Guidelines

The bullet points below are the guidelines for your presentation. These considerations can be used beyond this project, more generally, as big picture concepts and explanatory needs for building and communicating a data investigation.

- Describe your data

The data description should include important contextual and background information. This is essentially a summary of your data dictionary.

- Explain what interests you about the dataset & why you chose it.

Although many data investigations aren't necessarily interest driven, making data connections through interest and relevance can be a powerful tool to boost, motivate and/or facilitate understanding. In some cases, identifying interests in the data or using interests to drive a data investigation can allow you to access your prior knowledge and experience. Or, perhaps you chose the data because you were interested in investigating something new.

- Explain your data investigation or research question of interest?

During this step, you should describe any hypotheses you generated or expectations that you may have had going into the investigations. Relatedly, you should describe your dataset characteristics in terms of your data investigation. This step might be different from your previous data descriptions in that you might desrcibe a subset of your data in terms of the hypotheses and questions you seek to investigate. Here, you might include more information

about the meaning and measures of certain variables. Additionally, you may have visualizations or tables that help you describe aspects of the data relevant to your investigation.

- Explain how you processed your data (e.g, tell us about your data moves).

For this step you can elaborate on the *why* behind your data filtering or subsetting choices (among other data processing choices). You might explain why you used averages, or why you created a new variable (e.g., created levels from a continuous variable, merged your supplemental data by a common ID, etc.).

- Explain what you did to investigate your question of interest?

For this step, describe what your analysis was and why you chose such a method. It's possible that your analysis step required additional data moves. For example, if you decided to visualize a trend in the data, perhaps you had to reformat information (as seen in previous chapters) in order to create your desired data representation.

- Explain what you found.

In communicating your findings, you should provide a contextual explanation. Or, if your data investigation and the related analysis methods were exploratory, you might describe the insight you found that require further investigation, perhaps including suggestions for additional needs and lines of inquiry. These might include recommendations for additional measures and sources that would allow for answering the related questions.

The need for extra information is also related to the limitations of your data. There may be limitations to the generalizability of your findings and the conclusions that can be garnered. For example, a visual trend might be a strong suggestion for a particular relationship, but the strength of the relationship and related statements you might want to make could be further supported by formal statistical modeling, analysis, and related diagnostics.

- Describe how this project relates to (or informs, or does not relate to or inform) your idea of data science & programming.

For this project, you should reflect on your process and connect it to data science and project workflows. You might look back at chapter one to see how your process aligned or differed from the frameworks presented there. Consider and explain where you might use AI in a particular project or workflow phase that you may not have already. Consider what you might do differently in a new individual project, or in a project team setting. Perhaps you have ideas for your personal data science workflow philosophy!

9.2 What else?

The format in which you choose to present your information is important. Many of the components above can be reported directly from default programming output. However, you should not just present the default output from something like the `summary()` function in R, for example. This information is undoubtedly useful, but you should reformat this type of output to include what's relevant and important with the same considerations you might have for a data visualization (e.g., minimizing clutter, adding contrast between rows of information, etc.).

Likewise, you should not include all of your code in your presentation. For example, there would not be a need to show your coding process for creating a particular data visualization. In fact, you might choose to hide all of your code within your presentation. You would have your (well commented) code accessible apart from your presentation as part of the reproducibility process and, in this case, as a course requirement.

9.3 A few resources

- If you choose to create your presentation within `quarto`, check out this resource: Quarto Presentations.
- To create formatted tables, consider the kableExtra package

9.4 Finally

Congratulations on learning to use R and Python within a data science workflow. You're now ready for your next step in your data science and AI learning journey!

References

1. Lee H, Mojica G, Thrasher E, Baumgartner P. Investigating data like a data scientist: Key practices and processes. *Statistics Education Research Journal* (2022) 21:3–3.

2. Council FP. Fair information practice principles. (n.d.)

3. Erickson T, Wilkerson M, Finzer W, Reichsman F. Data moves. *Technology Innovations in Statistics Education* (2019) 12:

4. Pruim R, Gîrjău M-C, Horton NJ. Fostering better coding practices for data scientists. *Harvard Data Science Review* (2023) 5:

5. Tufte ER. *The visual display of quantitative information*. Second. Cheshire, Connecticut: Graphics Press. (2001). https://www.edwardtufte.com/tufte/books_vdqi