#4-3: Data Analysis

CS SCHOLARS – PROGRAMMING

Learning Goals

Read and write data from **files**

Use **built-in libraries** to interpret **protocols** in files

Reformat data to find, add, remove, or reinterpret pre-existing data

Perform **basic analyses** on data, including calculating **statistics** and **probabilities**, to answer simple questions

Data Analysis

Data Analysis Gains Insights on Data

Data Analysis is the process of using computational or statistical methods to **gain insight** about data.

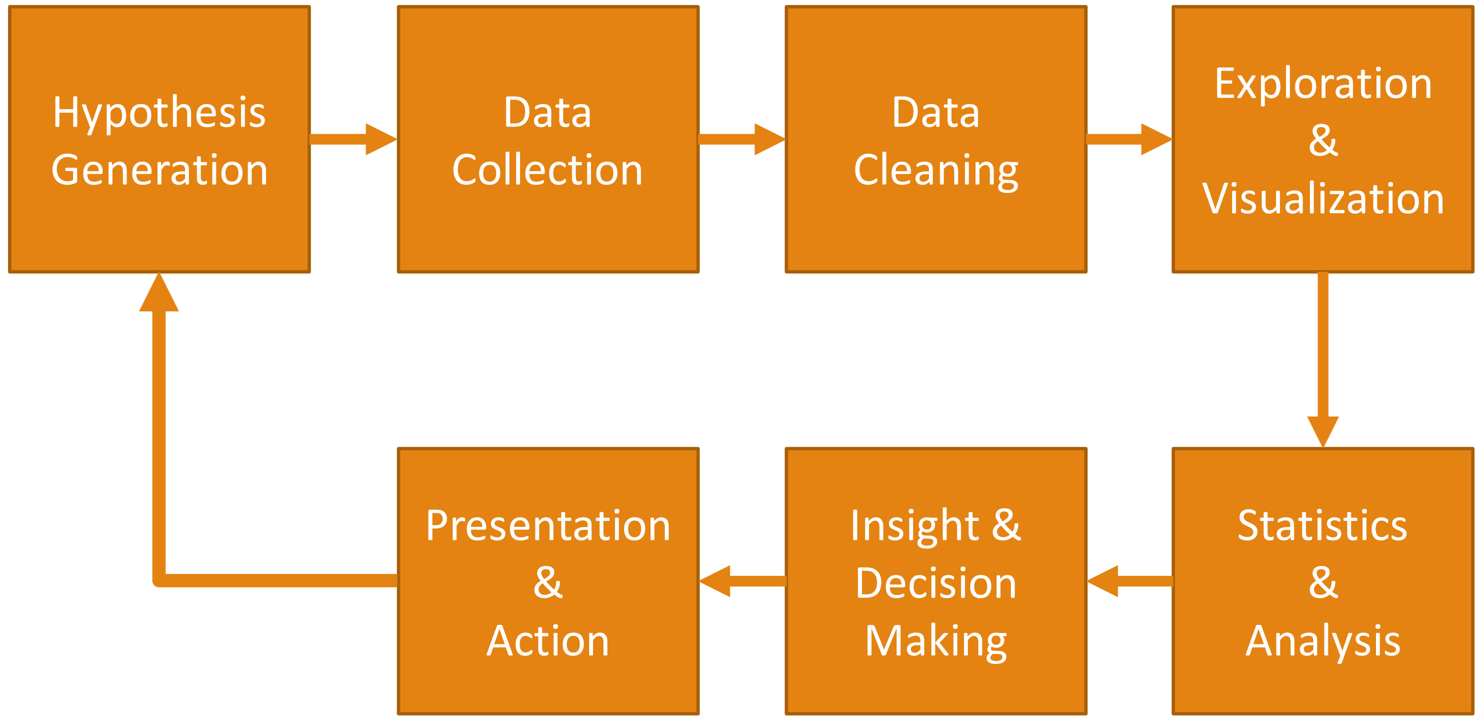
Data Analysis is used widely by many organizations to answer questions in many different domains. It plays a role in everything from advertising and fraud detection to airplane routing and political campaigns.

Data Analysis is also used widely in **logistics**, to determine how many people and how much stock is needed and where they should go.

Data Analysis Process

The full process of data analysis involves multiple steps to acquire data, prepare it, analyze it, and make decisions based on the results.

We'll focus mainly on three steps: Data Cleaning, Exploration & Visualization, and Statistics & Analysis



Data is Complicated

Before diving into data analysis, we have to ask a general question. What does data **look** like?

Data varies greatly based on the context; every problem is unique.

Example: let's collect our own data! Fill out the following short survey:

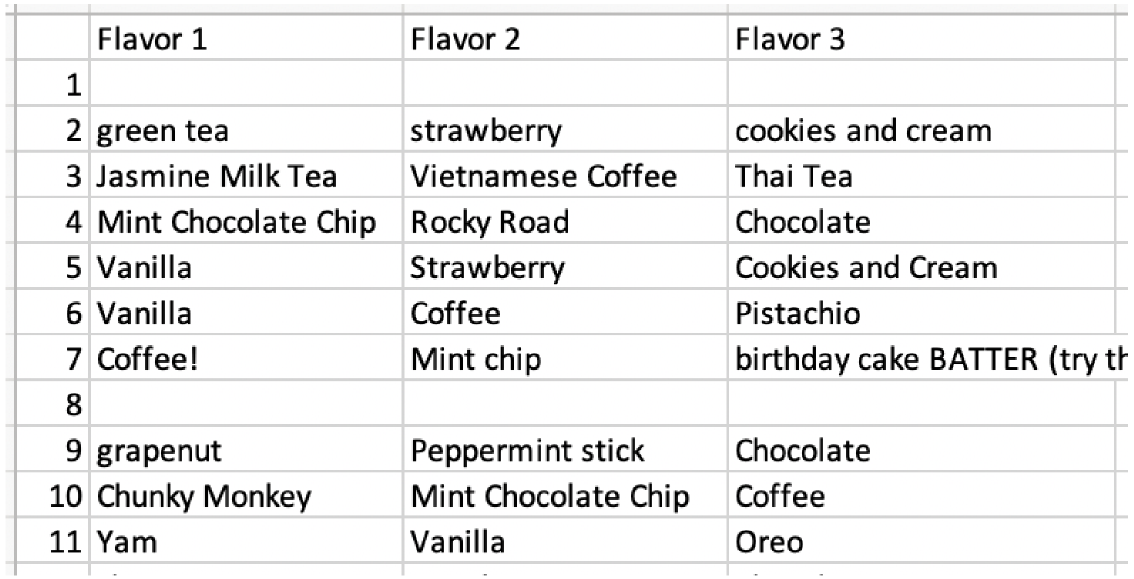
https://forms.gle/PAD8NMt6LZHXJnQ2A

Data is Messy

Let's look at the results of our ice cream data.

Most likely, there are some **irregularities** in the data. Some flavors are capitalized; others aren't. Some flavors might have typos. Some people who don't like ice cream might have put 'n/a', or 'none', or 'I'm lactose intolerant'. And some flavors might have multiple names – 'green tea' vs. 'matcha'.

Data Cleaning is the process of taking raw data and smoothing out all these differences. It can be partially automated (all flavors are automatically made lowercase) but usually requires some level of human intervention.



	Flavor 1	Flavor 2	Flavor 3
1			
2	green tea	strawberry	cookies and cream
3	Jasmine Milk Tea	Vietnamese Coffee	Thai Tea
4	Mint Chocolate Chip	Rocky Road	Chocolate
5	Vanilla	Strawberry	Cookies and Cream
6	Vanilla	Coffee	Pistachio
7	Coffee!	Mint chip	birthday cake BATTER (try th
8			
9	grapenut	Peppermint stick	Chocolate
10	Chunky Monkey	Mint Chocolate Chip	Coffee
11	Yam	Vanilla	Oreo

Reading Data from Files

Reading Data From Files

Once data has been cleaned, we need to access that data in a Python program. That means we need to **read data from a file**.

Recall that all the files on your computer are organized in **directories**, or **folders**. When you're working with files, always make sure you know which sequence of folders your file is located in. A sequence of folders from the top-level of the computer to a specific file is called a **filepath**.

Opening Files in Python

To interact with a file in Python we'll need to access its contents. We can do this by using the built-in function `open(filepath)`. This will create a **File object** which we can read from or write to.

```
f = open("sample.txt")
```

`open()` can either take a full filepath or a **relative path** (relative from the location of the python file). It's usually easiest to put the file you want to read/write in the same directory as the python file so you can simply refer to the filename directly. (In repl.it, you can do this by uploading a file to the same repository as your code file).

Reading and Writing from Files

When we open a file we need to specify whether we plan to **read from** or **write to** the file. This will change the **mode** we use to open the file.

```
f = open("sample.txt", "r") # read mode
lines = f.readlines() # reads the lines of a file as a list of strings
# or
text = f.read() # reads the whole file as a single string

f = open("sample2.txt", "w") # write mode
f.write(text) # writes a string to the file
```

Only one instance of a file should be kept open at a time, so you should always **close** a file once you're done with it.

```
f.close()
```

Be Careful When Programming With Files!

WARNING: when you write to files in Python backups are not preserved. If you overwrite a file, the previous contents are gone forever. **Be careful when writing to files**.

WARNING: if you have multiple Python files open in Pyzo and you try to open a file from a relative path, Pyzo might get confused. To be safe, when working with files, only have one file open in Pyzo at a time. And make sure to '**Run File as Script**' when working with files.

Activity: Read a File

You do: Download the file `chat.txt` from the schedule page and move it to the same folder as a python script. Try using `open` and `read` to open the file and read the contents, then `print` the contents.

If Python says a filename doesn't exist when you're sure that it does, go to office hours or message the TAs to get help; there's a few common problems that can occur.

Common file reading issues:

- make sure the file is actually in the same directory as your python script
- make sure the filename you've entered is actually the filename (including the filetype at the end!)
- make sure you're using **Run File as Script** (execute usually won't work)
- make sure only one file is open in Pyzo

Sidebar: os library for advanced files

The os library lets you directly interact with your computer's operating system. You can use this library to further modify files on your computer. The following functions are especially useful:

```
os.listdir(path) # returns a list of files in the folder

os.path.exists(path) # returns True if the given path exists

os.rename(a, b) # changes file a's name to b

os.remove(path) # deletes the file
```

Special Characters

As we start working with file text, we'll need to account for characters that are hard to represent in string values. These include the enter character (**newline**) and the tab character (**tab**). We can't type these directly into a string, so we'll use a shorthand instead:

```
"ABC\nDEF" # newline, or pressing enter/return
"ABC\tDEF" # tab
```

The \ character is a special character that indicates an **escape sequence**. It is modified by the letter that follows it. These two symbols are treated as a single character by the interpreter.

File Libraries

Data has Many Different Formats

Once you've read data from a file you need to determine what the **protocol** of that data is. That will inform how you interpret the data in Python.

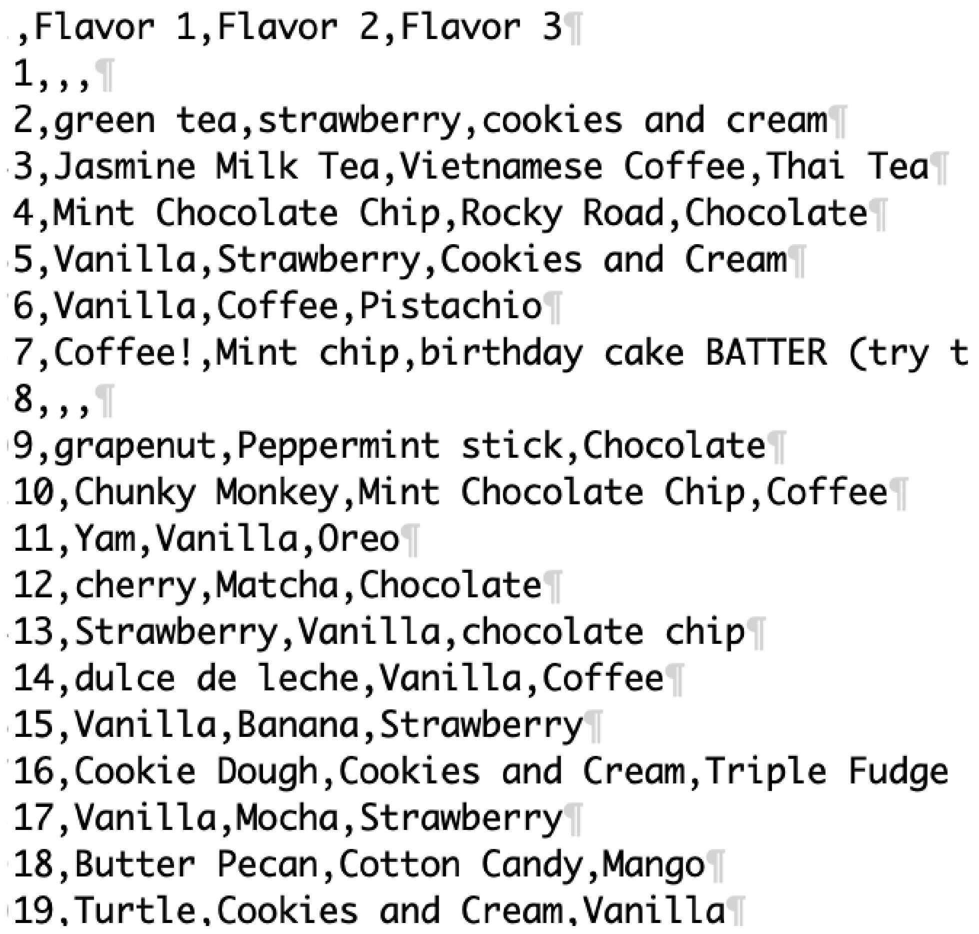
There are lots of different protocols! We'll just look at one for now – **CSV**. We'll also talk about how to deal with common data formats (like timestamps) and unusually formatted data.

CSV Files are Like Spreadsheets

Comma-Separated Values (CSV) files store data in two dimensions. They're effectively spreadsheets.

The data we collected on ice cream was downloaded as a CSV. If we open it in a plain text editor, you can see that values are separated by **commas**.

These files don't always have to use commas as separators, but they do need a **delimiter** to separate values (maybe spaces or tabs).



```
,Flavor 1,Flavor 2,Flavor 3¶
1,,,¶
2,green tea,strawberry,cookies and cream¶
3,Jasmine Milk Tea,Vietnamese Coffee,Thai Tea¶
4,Mint Chocolate Chip,Rocky Road,Chocolate¶
5,Vanilla,Strawberry,Cookies and Cream¶
6,Vanilla,Coffee,Pistachio¶
7,Coffee!,Mint chip,birthday cake BATTER (try t
8,,,¶
9,grapenut,Peppermint stick,Chocolate¶
10,Chunky Monkey,Mint Chocolate Chip,Coffee¶
11,Yam,Vanilla,Oreo¶
12,cherry,Matcha,Chocolate¶
13,Strawberry,Vanilla,chocolate chip¶
14,dulce de leche,Vanilla,Coffee¶
15,Vanilla,Banana,Strawberry¶
16,Cookie Dough,Cookies and Cream,Triple Fudge
17,Vanilla,Mocha,Strawberry¶
18,Butter Pecan,Cotton Candy,Mango¶
19,Turtle,Cookies and Cream,Vanilla¶
```

Reading CSV Data into Python

We could open a CSV file as plaintext and parse the file as we read it. Or we could use the **csv library** to make reading the file easier.

This library creates a **Reader** object out of a File object. This can be automatically converted to a 2D list, with rows separated by newlines and columns separated by the delimiter.

We can pass a keyword argument `delimiter` into `csv.reader` to change the delimiter.

```
import csv

f = open("icecream.csv", "r")
reader = csv.reader(f)

data = list(reader)

print(data)

f.close()
```

Writing CSV Data to a File

What if we've processed data in a 2D list and want to save it as a CSV file?

Create a CSV **Writer** object based on a file. You can use it to write one row at a time using `writer.writerow(row)`.

Again, the delimiter can be set to values other than a comma by updating the optional parameter `delimiter`.

```
import csv

data = [[ "chocolate", "mint chocolate",
          "peppermint" ],
        [ "vanilla", "matcha", "coffee" ],
        [ "strawberry", "mango", "cherry" ]]

# need newline="" to avoid double newlines
f = open("results.csv", "w", newline="")
writer = csv.writer(f)

for row in data:
    writer.writerow(row)

f.close()
```

Activity: Parse the ice cream data

You do: download the `icecream.csv` file and try to open and read it using the `csv` library. Load the data into a 2D list variable and try printing it out.

`datetime` Library

If you're working with data that includes timestamps, the `datetime` module is useful for parsing information out of the timestamp.

There are functions that let you get the day, month, hour, minute, or whatever else you might want out of a timestamp. You can also convert timestamps between different formats (like `"mm/dd/yy"` to `"dd-mm-yyyy"`).

You can also get the timestamp at the moment the line of code runs. This is useful when you're generating log files.

```
import datetime

day = datetime.date.fromisoformat("2020-01-12")
# datetime.date(2020, 1, 12)
day.strftime("%d-%m-%y") # 12-01-2020

datetime.date.today() # day, month, year
datetime.date.today().month # just month
datetime.datetime.now()
# also hour, minute, seconds
```

Reading Plaintext Data

A lot of the data we work with might not fit nicely into a known format like CSV. If we can read this data in a simple text editor, we call this **plaintext data**.

To work with plaintext, you need to identify what kinds of **patterns** exist in the data and use that information to structure it. The patterns you identify may depend on which question you're trying to answer.

Reformatting

Questions to Ask

When parsing data in a plaintext file, start by identifying the pattern; then ask yourself a few questions about that pattern.

- Does the pattern occur across lines, or some other delimiter?
- Where is the information in a single line/section?
- What comes before or after the information you want?

Tools to Use

Once you've identified where the information is located, use **string slicing** and **string methods** to separate out the information you need.

Slicing (`s[start:end:step]`) can be used to remove parts of the data that are unnecessary.

The **split** method (`s.split(".")`) can be used to break up data that is separated by a known delimiter.

The **index** method (`s.index(":")`) can be used to find the location of the beginning or end of a section. That can be combined with slicing or splitting to isolate the needed data.

The **strip** method (`s.strip()`) can be used to remove whitespace (spaces, tabs, and newlines) from the front and back of a string. This is useful for isolating the core text of a string.

Example: Parsing a Chat Log

`chat.txt` is a dataset based on a chat log from a class. (All student names have been modified to preserve student privacy).

How could we get the names of everyone who participated in the chat? What's the **pattern**?

```
14:54:28        From  Malika : Could I use recursion
for AuthorMap?

14:56:03        From  Ed : yep

15:00:22        From  Arman : what is str.digits?

15:01:21        From  Margaret Reid-Miller   to
Kelly Rivers(Privately) : We only hear the music when
you speak

15:08:31        From  Ed : how would you know if it
were O(n**.5)?
```

Example: Parsing a Chat Log

Each message occurs on an individual line; split the text based on newlines (`"\n"`).

`"From"` occurs before each name and `" : "` occurs afterwards. `Index` those indices and slice based on them.

Use `strip` to clear extra whitespace.

```
f = open("chat.txt", "r")
text = f.read()
f.close()

people = [ ]
for line in text.split("\n"):
    start = line.index("From") + \
            len("From")
    line = line[start:]
    end = line.index(" : ")
    line = line[:end]
    line = line.strip()
    people.append(line)
print(people)
```

Example: Parsing a Chat Log

A few lines don't match the pattern; account for those too.

If statements are useful when something breaks a pattern.

```
...
    line = line[:end]
    if "(Privately)" in line:
        end = line.index("to")
        line = line[:end]
    line = line.strip()
...
```

Activity: getMessages(text, name)

You do: write a function getMessages(text, name) that takes the text of the chat in a string and returns a list of all the messages posted by the person with the given name. You may want to use the code we just wrote, which we've put in the function getName to the right.

Try testing your code by running getMessages(text, "Malika") with text as defined earlier. You should get:

```
['Could I use recursion for AuthorMap?',
 'Why do we count 2 for a merge split?',
 'Oh, ok, thank you!']
```

Hint: what information do you need to check on each line to see if it should be included?

```
def getName(line):
    start = line.index("From") + len("From")
    line = line[start:]
    end = line.index(":")
    line = line[:end]
    if "(Privately)" in line:
        end = line.index(" to ")
        line = line[:end]
    line = line.strip()
    return line
```

Update Values with Index Assignment

Once we've parsed our data into an appropriate format, we may need to change the structure to achieve the analysis we want. Let's assume that we're working with a 2D list produced from the ice cream data.

To **update** a value, access the appropriate column in each row and change it. For example, you might want to convert a string to a different type via type-casting.

```
# Assume data is a 2D list parsed from the file
for row in range(len(data)):
    for col in range(len(data[row])):
        # Make all flavors lowercase
        data[row][col] = data[row][col].lower()
print(data)
```

Remove Values with pop()

To **remove** a value, pop an element of each row based on the column that needs to be removed.

```
# Assume data is a 2D list parsed from the file
for row in range(len(data)):
    data[row].pop(0) # remove the timestamp
    for col in range(len(data[row])):
        # Make all flavors lowercase
        data[row][col] = data[row][col].lower()
print(data)
```

Add Values with append()/insert()

To **add** a value, append or insert a new value into each row, potentially based on the pre-existing values.

```
# Assume data is a 2D list parsed from the file
for row in range(len(data)):
    data[row].pop(0) # remove the timestamp
    chocCount = 0 # count number of chocolate
    for col in range(len(data[row])):
        # Make all flavors lowercase
        data[row][col] = data[row][col].lower()
        if "chocolate" in data[row][col]:
            chocCount += 1
    # track chocolate count
    data[row].append(chocCount)
print(data)
```

Headers are Special Cases

Make sure to update the **header** according to a separate rule!

```
# Assume data is a 2D list parsed from the file
header = data[0]
header.pop(0) # remove the timestamp
header.append("# chocolate")
for row in range(1, len(data)):
    data[row].pop(0) # remove the timestamp
    chocCount = 0 # count number of chocolate
    for col in range(len(data[row])):
        # Make all flavors lowercase
        data[row][col] = data[row][col].lower()
        if "chocolate" in data[row][col]:
            chocCount += 1
    # track chocolate count
    data[row].append(chocCount)
print(data)
```

Analysis

Basic Data Analyses – Statistics Library

There are many basic analyses we can run on features in data to get a sense of what the data means. You've learned about some of them already in math or statistics classes, such as **mean, median,** and **mode**.

You can implement these in Python yourself, but you don't have to! There's already a **statistics** library that does this for you.

```
import statistics
data = [41, 65, 64, 50, 45, 13, 29, 14, 7, 14]

statistics.mean(data) # 34.2
statistics.median(data) # 35.0
statistics.mode(data) # 14
```

Example: Statistics of Ice Cream

We can start by measuring the **statistics** of the ice cream dataset.

The data is text, so we must turn it into numbers before performing analyses like `mean`.

Try counting the number of favorite flavors that fall into a set of flavor names for each person, then put those counts into a list to analyze.

```
def getFlavorCounts(data, flavorSubset):
    counts = []
    for row in range(1, len(data)): # skip header
        count = 0
        # skip chocolate count
        flavors = data[row][:len(data[row])-1]
        for flavor in flavors:
            if flavor in flavorSubset:
                count += 1
        counts.append(count)
    return counts

import statistics
statistics.mean(getFlavorCounts(data,
    [ "chocolate", "vanilla", "strawberry"]))
```

Activity: Most Popular Ice Cream

You do: write a bit of code to calculate the most popular ice cream across all preferences in the `icecream.csv` dataset.

Hint: with this problem, we don't need to convert the data to numbers! The easiest way to do this is to make a list of all ice cream preferences, then use the `mode` function to find the most common string.

Calculating Probabilities

You'll also often want to calculate probabilities based on your data.

In general, the probability that a certain data type occurs in a dataset is the count of how often it occurred, divided by the total number of data points.

```
lst.count(item) / len(lst)
```

Conditional probability (the probability of something occurring given another factor) is slightly more complicated.

Create a modified version of the list that contains only those elements with that factor; then you can use the same equation.

```
newLst = []
for x in lst:
    if meetsProperty(x):
        newLst.append(x)
newLst.count(item) / len(newLst)
```

Example: Probability of Ice Cream

Now let's look at the **probabilities** of the dataset.

To get the probability that someone likes a certain flavor, change the code slightly to count whether or not that flavor shows up in each person's preferences.

Then divide by the total number of data points at the end.

```
def getFlavorProb(data, flavor):
    flavorCount = 0
    for i in range(1, len(data)): # skip header
        # skip chocolate count
        flavors = data[row][:len(data[row])-1]
        if flavor in flavors:
            flavorCount += 1
    return flavorCount / (len(data) - 1) # skip header

print(getFlavorProb(data, "chocolate"))
```

More Analysis Methods

There's plenty of other data analysis methods we could cover – bucketing, detecting outliers, dealing with missing data – but what kind of method you need will depend entirely on the context of the problem you're solving.

Use algorithmic thinking to translate the analysis you want to perform into a set of steps, then translate those steps into code!

Learning Goals

Read and write data from **files**

Use **built-in libraries** to interpret **protocols** in files

Reformat data to find, add, remove, or reinterpret pre-existing data

Perform **basic analyses** on data, including calculating **statistics** and **probabilities**, to answer simple questions