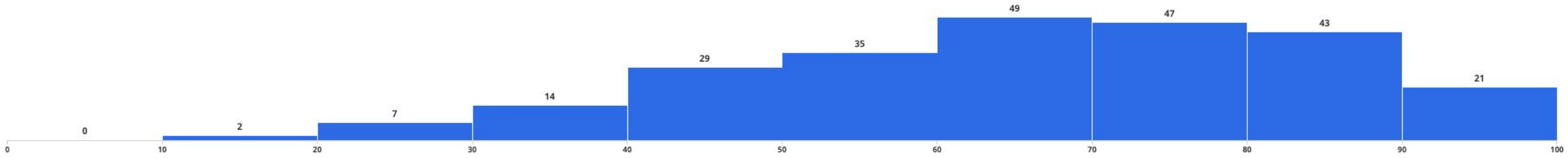
Data Analysis – Modeling and Parsing

15-110 – Monday 11/11

Announcements

- **Hw5** was due today
  - How did it go?
- **Exam2** median: 68.1
  - This is lower than we usually expect. You should review your exam feedback to see what went wrong and use that information to study for the final exam.
  - Complete Exam 2 Reflection by Friday @ 11:59pm – Worth 5 points on Check 6-1: https://forms.gle/A7v9dX1DTcextxTv5



Hw6 – Introduction

Hw6 and its Checks work differently from the prior assignments. Instead of solving a bunch of small problems, you'll build a **guided project** that uses one of the three applications to solve an interesting problem in another field using programming.

The programming problems in Check6-1 will support the work in Check6-2, and the code from both will support the work in Hw6.

Because of this, **revision deadline is early for Check 6-1 and there is no revision deadline for Check 6-2 or HW 6**. Make sure to start each assignment promptly!

(There are also short written assignments for Check6-1 and Check6-2 that cover the lecture content – don't forget to do those too).

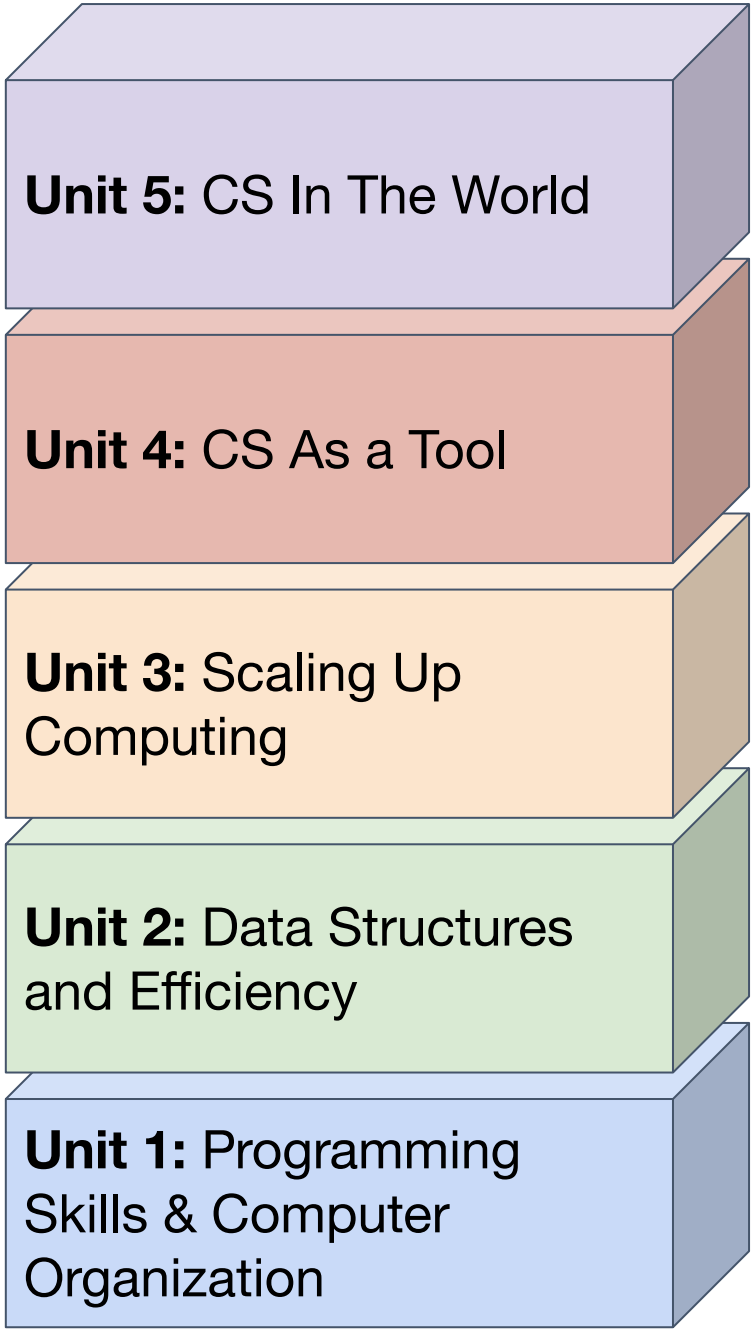
Hw6 – Getting Started

To get started on Hw6, review the **General Guidelines:** https://www.cs.cmu.edu/~110/hw/hw6_general.pdf

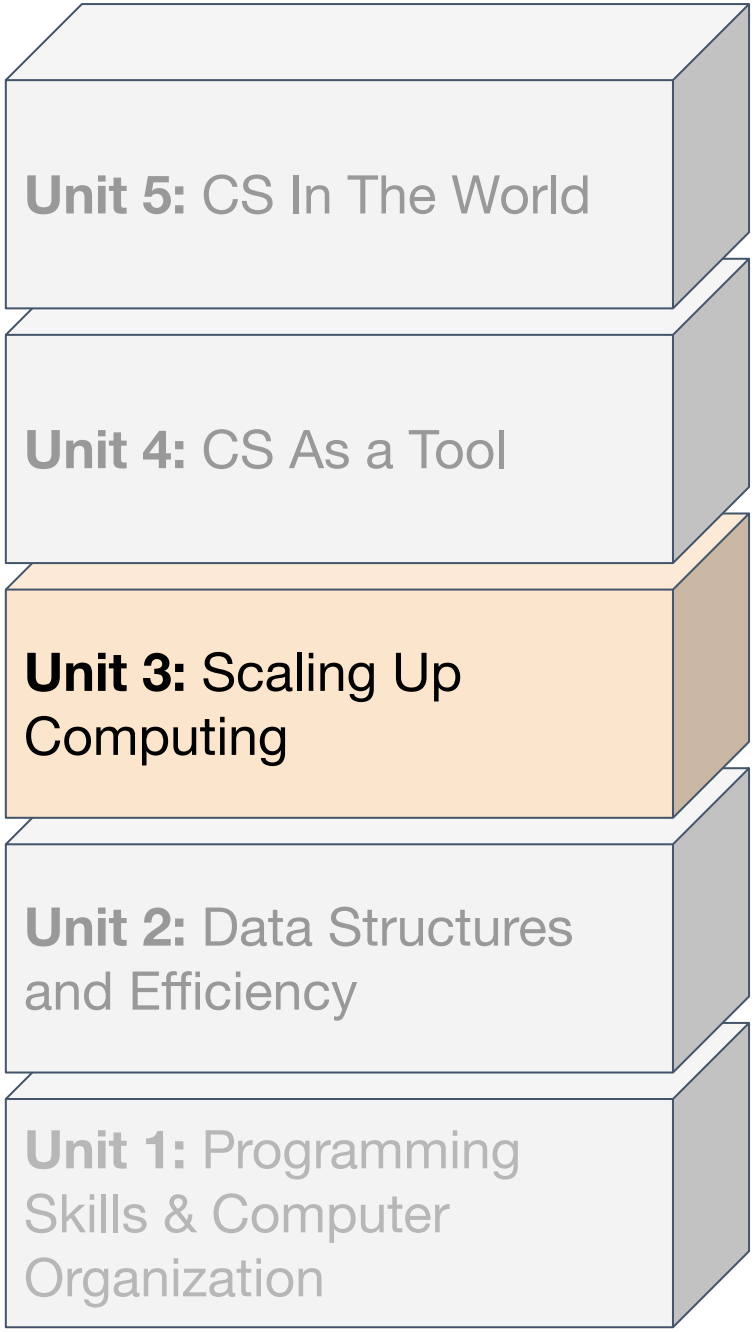
Once you've picked a project, download the instructions & starter files from the table at the bottom of the assignments page: https://www.cs.cmu.edu/~110/assignments.html

Then fill out this form to let us know which project you are doing (deadline **Wednesday 11/13 at 11:59 PM**): https://forms.gle/V6S81LUku8wHRJ6a9

We’ve finished Unit 3!

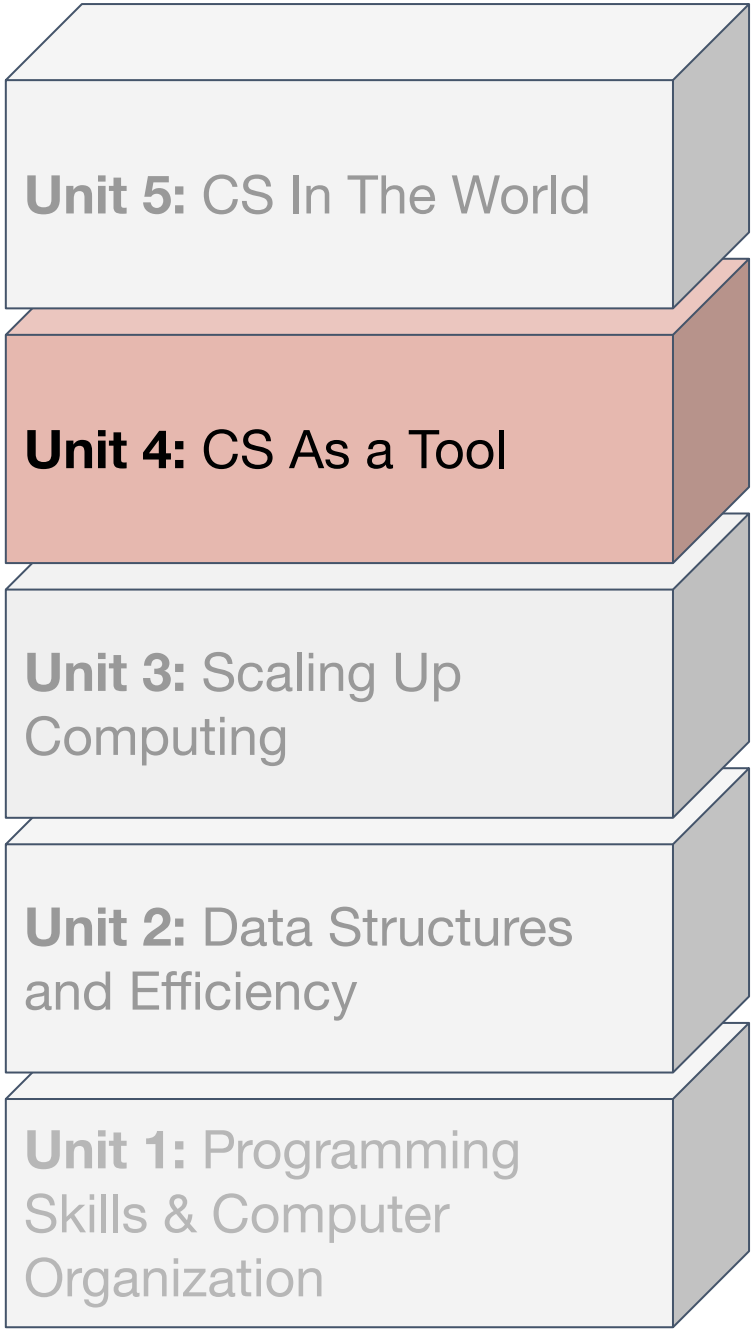


The topics we cover in this course build on each other!



Final includes Unit 3:

- Parallel Programming
- Distributed Computing and the Internet
- Fault Tolerance and Security
- Managing Large Code Projects



Unit 4 Topics:

CS As a Tool:

- Organizing data to help people answer questions
- Applications of CS:
  - Data analysis
  - Simulation
  - Machine learning

Learning Goals

- Identify whether features in a dataset are **categorical**, **ordinal**, or **numerical**
- Interpret data according to different **protocols:** CSV and JSON
- Use string operations and methods to extract data from **plaintext**
- **Reformat** data to find, add, remove, or reinterpret pre-existing data

Data Analysis

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

Data Analysis is used widely by organizations to answer questions in many different domains. Some examples:

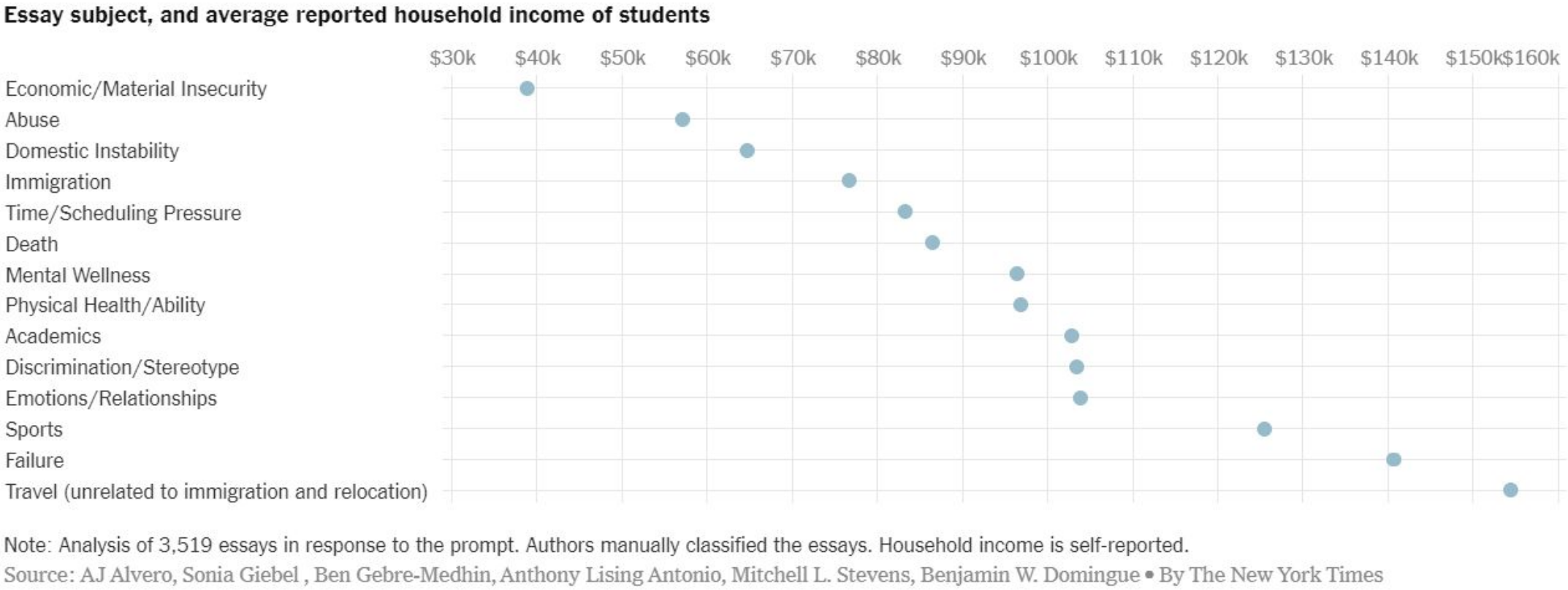
- Government
- Healthcare
- Marketing and Advertising
- Machine Learning

Lots of interesting graphs and data from NY Times* articles.

https://www.nytimes.com/column/whats-going-on-in-this-graph

You can use your CMU email to get a free NY Times subscription.

Example question: How does family income relate to which personal challenges students write about in their college essays?

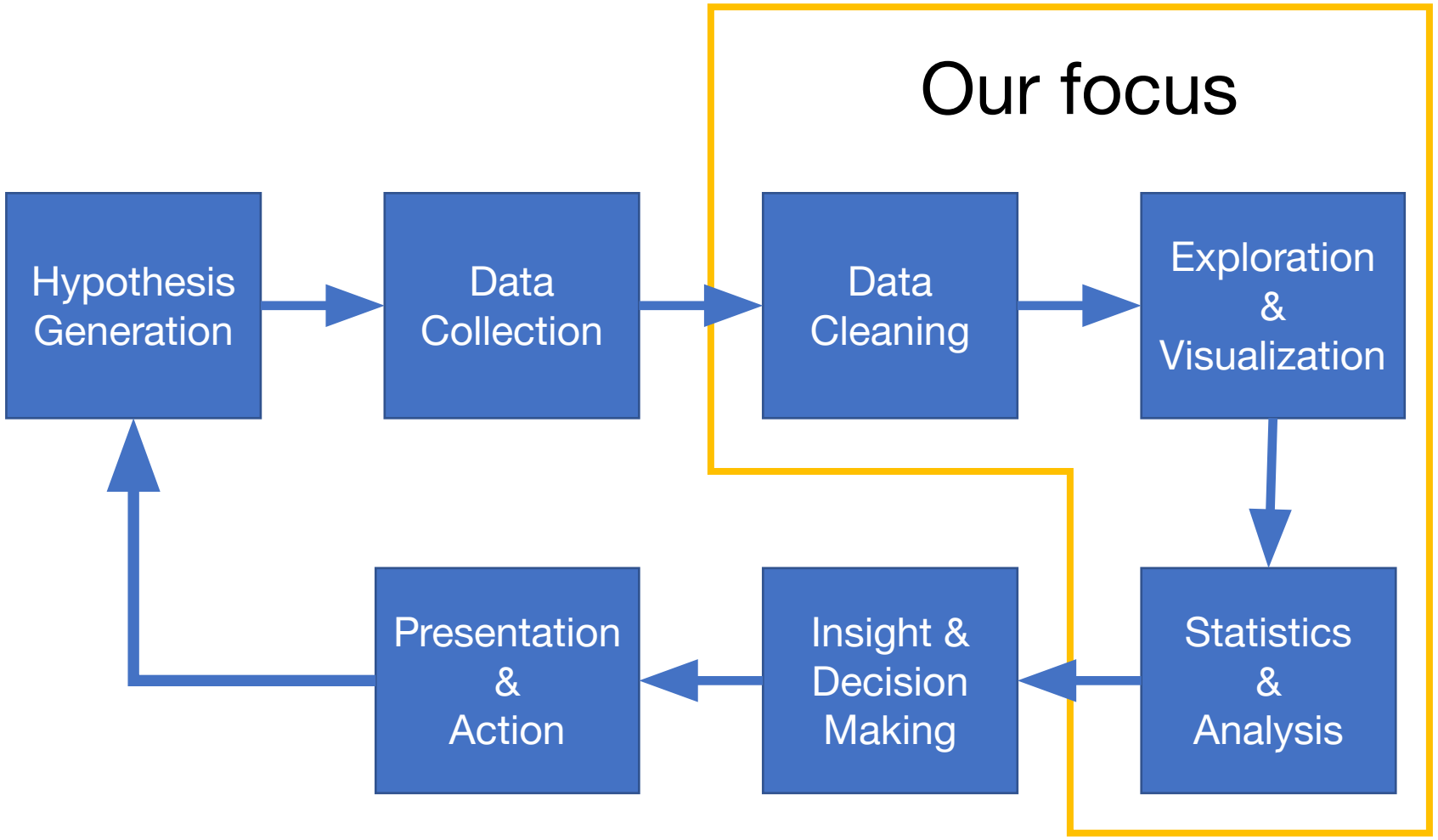


Source:
https://www.nytimes.com/2021/12/09/learning/whats-going-on-in-this-graph-jan-5-2022.html

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:

1. **Data Cleaning**
2. Exploration & Visualization
3. Statistics & Analysis



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://tinyurl.com/110-ice-cream-f24

Data Cleaning is the process of taking messy data and smoothing out all irregularities.

Usually 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 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

When we work with simple data, that data often falls into one of **three types** which will determine what analysis we run.

Categorical: Data fall into one of several categories. Those categories are separate and cannot be compared.

Example: style of house (ranch, split-level, two-story, duplex, Victorian, etc.)

Ordinal: Data fall into separate categories, but those categories **can** be compared – they have a specific **order**.

Example: what is the condition of the house? (poor, fair, good, excellent, new)

Numerical: Data are **numbers**. We can perform mathematical operations on it and compare it to other data.

Example: how large is the house in square feet?

Activity: Data types

You do: what type of data are our ice cream flavors – categorical, ordinal, or numerical?

What if we added a column asking how many times the person ate ice cream in the past week? Would that be categorical, ordinal, or numerical?

REVIEW: Reading Data from Files

When building more complex programs, we will often want to read data stored in a file.

Recall that all the files on your computer are organized in **directories**, or **folders**. The file structure in your computer is a **tree** – directories are the inner nodes (recursively nested) and files are the leaves.

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**.

For example, **Users > rware > Documents > sample.txt** refers to the file **sample.txt** in the **Documents** folder, which is in the **rware** folder, which is in the **Users** folder, which is at the top level of the computer.

We can open files in Python using the built-in function `open(filepath)`.

This will create a **File object** which we can read from or write to.

```
f = open("/Users/rware/Documents/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.

```
f = open("sample.txt")
# if .py file is in Documents, will search for this file there
```

When opening a file we need to set the **mode**: whether we plan to **read from** or **write to** the file.

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

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

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

```
f.close()
```

Data Formats – CSV and JSON

When reading data from a file, you need to determine what is the **structure of the data**.

That will inform how you store the data in Python.

We'll discuss two formats here: **CSV and JSON**. Then we'll discuss how to deal with plaintext (text data not in a specific format). Many other formats exist, though!

Comma-Separated Values (CSV) files store data in two dimensions like 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¶
```

We can **open a CSV file** using the `csv` library.

Using the `csv` **library** to makes reading CSV files easier.

This library creates a **Reader** object out of a File object. That object can be cast to a 2D list, where each inner list corresponds to a line in the file, and the elements on the line (as separated by the delimiter) are the elements of the inner lists.

We can pass keyword arguments into the `csv.reader` call to set the delimiter.

```
import csv

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

data = list(reader)
print(data)

f.close()
```

JavaScript Object Notation (JSON) files store data that is nested, like trees.

JSON files are commonly used to store information that is organized in some structured way.

JSON files can store data types including Booleans, numbers, strings, lists, dictionaries, and any combination of the above.

```
{
  "vanilla" : 10,
  "chocolate" : {
                  "chocolate" : 15,
                  "chocolate chip" : 7,
                  "mint chocolate chip" : 5
                },
  "other" : [ "strawberry", "matcha", "coffee" ]
}
```

We can **open a JSON file** using the `json` library.

This time, we'll use `json.load(file)`. This function reads text from a file and produces a piece of data that matches the type of the outermost data in the text (usually a list or dictionary).

In our example from the last slide, the function would produce a dictionary mapping strings to integers, dictionaries, and lists.

```
import json

f = open("icecream.json", "r")
j = json.load(f)

print(j)

f.close()
```

Activity: Match Data Structure to Format

You do: which data format would you use to store the following types of data?

A) A hierarchical representation of employees in a company, organized based on who reports to whom.

B) A table of tax data where each person in the table has several columns of financial information.

Extracting Data from Plaintext

If we can read data into a simple text editor but can't fit it into a standard format, we call it **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 are trying to answer.

In the following code the variable `text` contains plaintext data:

```
filename = "sample.txt"
f = open(filename, "r") # read mode
text = f.read()
```

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?

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 previous 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

What data structure do we want to store the data in? Lists are a common example.

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

`"From"` occurs before each name and `" : "` occurs afterwards. `index` to find those locations 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()
...
```

Modifying Data

After parsing data into an appropriate format, we may need to **change the structure** to achieve the analysis we want.

This is very common in data analysis.

Assume that we are working with a 2D list produced from the ice cream data. How can we:

- change all the flavors to be lowercase?
- remove the timestamps from the dataset?
- add a new column that counts the number of chocolatey favorites?

To **update a value**, access the appropriate column in each row and **use indexing** to change it.

For example, you might want to convert a string to a different type via type-casting.

```
import csv

# Read in data from a CSV
f = open("icecream.csv", "r")
reader = csv.reader(f)
data = list(reader)
f.close()

# 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)
```

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

For example, you might want to remove user IDS when anonymizing data.

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

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

You can add data from a separate-but-connected dataset, or by performing small analyses on the existing data.

```
# Assume data is a 2D list parsed from the file
for row in range(len(data)):
    data[row].pop(0) # remove the ID
    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)
```

When using **headers**, make sure to treat them appropriately!

It is often easiest to skip the 0^{th} row in the loop and deal with it separately instead.

```
# Assume data is a 2D list parsed from the file
for row in range(len(data)):
    data[row].pop(0) # remove the ID
    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
    if row == 0:
        data[row].append("# chocolate")
    else:
        data[row].append(chocCount)
print(data)
```

Learning Goals

- Identify whether features in a dataset are **categorical**, **ordinal**, or **numerical**
- Interpret data according to different **protocols:** CSV and JSON
- Use string operations and methods to extract data from **plaintext**
- **Reformat** data to find, add, remove, or reinterpret pre-existing data

Extra Bits

The Pythonic way to open files using `with`.

Typically in Python, we use the with keyword to open files. The nice thing about this is we do not need to remember to call the close function.

```
f = open("icecream.csv", "r")
reader = csv.reader(f)
data = list(reader)
f.close()
```

```
with open("icecream.csv", "r") as f
    # file is open in the with body
    reader = csv.reader(f)
    data = list(reader)

# file is closed once we exit the with body
print(data)
```