Python Mega-Course: 10 Apps Notes:

Notes taken for “The Python Mega Course: Build 10 Real World Applications” on Udemy, taught by Ardit Sulce.

Notes taken by Travis Rillos.

List of Apps:

- **App 1**: Web Mapping with Python: Interactive Mapping of Population and Volcanoes
- **App 2**: Controlling the Webcam and Detecting Objects
- **App 3 (part 1)**: Data Analysis and Visualization with Pandas and Matplotlib
- **App 3 (part 2)**: Data Analysis and Visualization - In-Browser Interactive Plots
- **App 4**: Web Development with Flask - Build a Personal Website
- **App 5**: GUI Apps and SQL: Build a Book Inventory Desktop GUI Database App
- **App 6**: Mobile App Development: Build a Feel-Good App
- **App 7**: Web-Scraping - Scraping Properties for Sale from the Web
- **App 8**: Flask and PostGreSQL - Build a Data Collector Web App
- **App 9**: Django & Bootstrap Blog and Translator App
- **App 10**: Build a Geography Web App with Flask and Pandas
- **Bonus App:** Building an English Thesaurus
- **Bonus App:** Building a Website Blocker
- **Bonus App:** Data Visualization with Bokeh

Table of Contents

Section 1 – Welcome:

Course Introduction:

- Just an overview.
- This course will include how to program with Python from scratch, so I may end up skipping a lot of notes for the first 10 sections or so.
- There are **39 Sections**.

- There's a Discord channel: https://discord.gg/QWArvbdZVZ

Section 2 – Getting Started with Python:

Section Introduction:

- Sounds like we use VSCode for this class. Sweet.

Section 3 – The Basics: Data Types:

Python Interactive Shell:

- For Windows, run **py -3** in the terminal to start the interactive shell.
- Useful for testing some throwaway code; interactive shell doesn't save code.
- Creating .py files is better for creating reusable code.

Terminal:

- Tip about splitting the terminal in two. This way we can run both the **powershell terminal** and the **Python Interactive Shell** side-by-side.
- This allows us to run test code in the interactive code and run .py code in the terminal.

Data Type Attributes:

- Showed a useful command, **dir()**, which can be used very effectively in the Interactive Shell to find out what operations can be performed on a given subject (methods or properties).
    - Running **dir(list)** shows everything that can be performed on a list.
    - Running **dir(int)** shows everything that can be performed on an integer.
- He used the example of running **dir(str)** to see what can be performed on a string, chose **"upper"** from the list, then ran **help(str.upper)** to find out what it does.
    - This showed that "upper" is a method, which "Returns a copy of the string converted to uppercase".
- Note: Functions follow the naming convention **function()** while methods follow the naming convention **.method()**.

How to Find Out What Code You Need:

- To find a complete list of built-in functions, run **dir(__builtins__)**. These are functions that aren't attached to a specific data type.
- We didn't find an "average" or "mean" function, but there was a "**sum**" function. Between that and **len**, we can calculate an average for a list of floats.

What Makes a Programmer a Programmer:

- Three things you need to know to make any program:
    - Syntax
    - Data Structures
    - Algorithm

How to Use Datatypes in the Real World:

- In our example of creating a Dictionary of student names and grades, would we manually create this dictionary in the real world? Unlikely. The data would be stored in something like an Excel file.
- There are ways to automatically input data from an Excel file into Python.
- We will be doing this later in the course.

Section 4 – The Basics: Operations with Data Types:

More Operations with Lists:

- Went over a few methods such as **.append()**, **.index()**, and **.clear()**. Pretty basic stuff.
- Used **dir(list)** and **help(list.append)** etc to see what all can be done to lists.

Accessing List Items:

- In our "basics.py" with the list "monday_temperatures" in it, we used **monday_temperatures.__getitem__(1)** to get the item at Index 1, which was 8.8.
- He then showed that instead of that, we can just use **monday_temperatures[1]** and we get the same result.
- The version with the double underscores ("__getitem__(1)") is probably the private method within the function, that the "**[1]**" syntax calls to.

Accessing List Slices:

- To access a portion of a list **monday_temperatures = [9.1, 8.8, 7.5, 6.6, 9.9]**, we can use the syntax:
    - **monday_temperatures[1:4]**
    - To find the items at index 1, index 2, and index 3.
- We can also use **monday_temperatures[:2]** to get every item before index 2, or the first two items.
- A similar shortcut, **monday_temperatures[3:]** gives us the values from index 3 to the end of the list.

Accessing Items and Slices with Negative Numbers:

- Get last item of list with **monday_temperatures[-1]**. Super basic, but super useful.
    - In this case, running **monday_temperatures[-5]** gives us the first item again.
- Running **monday_temperatures[-2:]** with a colon gives us everything from the second-to-last item to the end of the list, or the last two items of the list.

Accessing Characters and Slices in Strings:

- Strings have the exact same indexing system as lists (duh).
- We can also index a string that's part of a list:
    - **monday_temperatures = ['hello', 1, 2, 3]**
    - **monday_temperatures[0]**
        - → 'hello'
    - **monday_temperatures[0][2]**
        - → 'l'

Accessing Items in Dictionaries:

- Started with the dictionary **student_grades = {"Mary": 9.1, "Sim": 8.8, "John": 7.5}** and input that in the Python interactive shell.
- Running **student_grades[1]** gives us **KeyError: 1** because the dictionary doesn't have a key called 1.
- However, running **student_grades["Sim"]** gives us 8.8.
- Instead of integers, dictionaries have **keys** as their indexes.
- He gave an example of why this can be very useful by writing a short English-to-Portuguese translation dictionary, then running **eng_port["sun"]** to output "**sol**".

Tip: Converting Between Datatypes:

Sometimes you might need to convert between different data types in Python for one reason or another. That is very easy to do:

From tuple to list:

```
1. >>> cool_tuple = (1, 2, 3)
2. >>> cool_list = list(cool_tuple)
3. >>> cool_list
4. [1, 2, 3]
```

From list to tuple:

```
1. >>> cool_list = [1, 2, 3]
2. >>> cool_tuple = tuple(cool_list)
3. >>> cool_tuple
4. (1, 2, 3)
```

From string to list:

```
1. >>> cool_string = "Hello"
2. >>> cool_list = list(cool_string)
3. >>> cool_list
4. ['H', 'e', 'l', 'l', 'o']
```

From list to string:

```
1. >>> cool_list = ['H', 'e', 'l', 'l', 'o']
2. >>> cool_string = str.join("", cool_list)
3. >>> cool_string
4. 'Hello'
```

As can be seen above, converting a list into a string is more complex. Here `str()` is not sufficient. We need `str.join()`. Try running the code above again, but this time using `str.join("---", cool_list)` in the second line. You will understand how `str.join()` works.

Section 5: The Basics: Functions and Conditionals:

Creating Your Own Functions:

- Started with an example from earlier in the course where we calculated our own average because there was no built-in function to do so:

```
student_grades = [9.1, 8.8, 7.5]

mysum = sum(student_grades)
length = len(student_grades)
mean = mysum / length
print(mean)
```

- Rather than do this, we can wrap these calculations in our own mean function that can then be used on other lists as well.
- I added some exception handling (to only accept a list and only return a float) to the code he presented:

```
def mean(mylist: list) -> float:
    the_mean = sum(mylist) / len(mylist)
    return the_mean

student_grades = [8.8, 9.1, 7.5]
print(mean(student_grades))
```

- He also ran **print(type(mean), type(sum))** in the same code and showed that ***mean*** *was class 'function' and* ***sum*** *was class 'builtin_function_or_method'.*

Intro to Conditionals:

- What if in the previous code, we passed a dictionary instead of a list?
    - In my case, my code has some error handling.
- We'd get an error because '+' can't be used on an 'int' and a 'str'. Our function isn't designed to process dictionaries, just lists. However, we can fix this with conditionals.

If Conditional Example:

- Note: I'll have to take my exception handling out for forcing the input to be a list. Don't know how to accept two different input data types yet.

```
def mean(myinput) -> float:

    if type(myinput) == list:
        the_mean = sum(myinput) / len(myinput)
    elif type(myinput) == dict:
        the_mean = sum(myinput.values()) / len(myinput)
    else:
        print("Invalid input type. Must be list or dictionary")

    return the_mean

monday_temperatures = [8.8, 9.1, 9.9]
student_grades = {"Mary": 9.1, "Sim": 8.8, "John": 7.5}
print(mean(student_grades))
print(mean(monday_temperatures))
```

- Added some basic exception handling in the **else** statement that he didn't have. He just routed any list inputs straight into "else".

Conditional Explained Line by Line:

- In this video he just went through and explained what was going on line-by-line. Basic stuff.

More on Conditionals:

- Stuff on Booleans, True/False, and how this works in conditionals.
- He mentioned the use of **if isinstance(myinput, dict)** as a useful bit of syntax. I should use that more often in my own code.
- He mentions that there are some very advanced reasons why the isinstance syntax is better to use, but that we won't get into that until later in the course.

Elif Conditionals:

- And yet I already used one in my earlier code. The structure of his course still makes sense for absolute beginners, but these first few sections are a bit of a slog.

Section 6: The Basics: Processing User Input:

User Input:

- We're going to be taking user input in the form of a temperature, to run through a function.

```
def weather_condition(temperature: float) -> str:
    if temperature > 7:
        return "Warm"
    elif temperature <= 7:
        return "Cold"
    else:
        return "Invalid input. Please enter a number."

user_input = float(input("Enter temperature: ")) ← ← ←
print(weather_condition(user_input))
```

- Added some exception handling again.
- We had to make sure the input was converted to a float (or an int), or else the program will take the input in as a string by default.

String Formatting:

- Now here's some wildcard syntax I don't see too often yet:

```
user_input = input("Enter your name: ")

message = "Hello %s!" % user_input ← ← ←
print(message)
```

- The <**%s**> and <**% user_input**> in particular is an interesting way to go about inputting that name. An **f-string** would probably also work if I can remember the proper syntax for one.
- Oh wait, he did one:

```
user_input = input("Enter your name: ")
message = "Hello %s!" % user_input
message = f"Hello {user_input}" ← ← ←
print(message)
```

- He noted that the f-string method works for Python 3.6 and above. The other method works for Python 2 and Python 3.
- You may want to program for an older version of Python, depending on the webserver you're running it on.

String Formatting with Multiple Variables:

- To use multiple variables, you more-or-less just add them on.

```
name = input("Enter your name: ")

surname = input("Enter your surname: ")
message = "Hello %s %s!" % (name, surname)
message = f"Hello {name} {surname}!"
print(message)
```

-

More String Formatting:

There is also another way to format strings using the *"{}".format(variable)* form. Here is an example:

```
1. name = "John"
2. surname = "Smith"
3. 
4. message = "Your name is {}. Your surname is {}".format(name, surname)
5. print(message)
```

Output: *Your name is John. Your surname is Smith*

Cheatsheet: Processing User Input:

In this section, you learned that:

- A Python program can get **user input** via the `input` function:
- The **input function** halts the execution of the program and gets text input from the user**:**

```
1. name = input("Enter your name: ")
```

- The input function converts any **input to a string**, but you can convert it back to int or float:

```
1. experience_months = input("Enter your experience in months: ")
2. experience_years = int(experience_months) / 12
```

- You can also **format strings** with:

```
1. name = "Sim"
2. experience_years = 1.5
3. print("Hi {}, you have {} years of experience".format(name, experience_years))
```

Output: `Hi Sim, you have 1.5 years of experience.`

Section 7: The Basics: Loops:

For Loops: How and Why:

- For loop iteration. Basic.

Dictionary Loop and String Formatting:

Here is an example that combines a dictionary loop with string formatting. The loop iterates over the dictionary and it generates and prints out a string in each iteration:

```
1. phone_numbers = {"John": "+37682929928", "Marry": "+423998200919"}
2.
3. for pair in phone_numbers.items():
4.     print(f"{pair[0]} has as phone number {pair[1]}")
```

And here is a better way to achieve the same results by iterating over keys and values:

```
1. phone_numbers = {"John": "+37682929928", "Marry": "+423998200919"}
2.
3. for key, value in phone_numbers.items():
4.     print(f"{key} has as phone number {value}")
```

In both cases, the output is:

```
John has as phone number +37682929928
Marry has as phone number +423998200919
```

While Loops: How and Why:

- He showed an infinite loop for starters. Interesting choice.

While Loop Example with User Input:

- Just a basic example to check if a username is correct.

```
username = ''

while username != 'pypy':
    username = input("Enter username: ")
```

-

While Loops with Break and Continue:

- Same functionality as previous, but different method:

```
while True:
    username = input("Enter username: ")
    if username == 'pypy':
        break
    else:
        continue
```

- He says he prefers this method over the previous one because it gives you more control over the workflow. He also finds it more readable.

Cheatsheet: Loops:

- We also have **while-loops**. The code under a while-loop will run as long as the while-loop condition is true:

```
1. while datetime.datetime.now() < datetime.datetime(2090, 8, 20, 19, 30,
   20):
2.     print("It's not yet 19:30:20 of 2090.8.20")
```

The loop above will print out the string inside `print()` over and over again until the 20th of August, 2090.

Section 8: Putting the Pieces Together: Building a Program:

Section Introduction:

- The purpose of this section is to fill in gaps in Python knowledge, to make everything work together.

Problem Statement:

- He showed off just the output of a program called **textpro.py**.
- The program takes some basic input sentences and then reformats them with proper capitalization and punctuation.
- Input prompts end when the input is "**\end**".

Approaching the Problem:

- We're going to look closely at the output ("It's good weather today. How is the weather there? There are some clouds here.").
- It's good to have a very clear idea of what the output should be.
- We look at the output and figure out how it can be broken down into smaller tasks.
- We're going to accomplish this with multiple functions.

Building the Maker Function:

- We tested several methods in our Python interactive shell as we went along, to test that their functionality would work for us.
    - **"how are you".capitalize()** gave us "How are you"
        - We wouldn't use .title() here because that would capitalize (almost) every word.
    - **"how are you".startswith(("who", "what", "where", "when", "why", "how"))** checks the phrase against a tuple containing all our interrogative words. This is how we can decide whether a sentence should end with a **"?"** or not.

- Here's what we had by the end of the lecture:

```
def sentence_maker(phrase):
    interrogatives = ("who", "what", "where", "when", "why", "how")
    capitalized = phrase.capitalize()
    if phrase.startswith(interrogatives):
        return "{}?".format(capitalized)
    else:
        return "{}.".format(capitalized)

print(sentence_maker("how are you"))
```

- We tested with the phrase "how are you" to check functionality, and it came back properly formatted:
    - → How are you?

Constructing the Loop:

- We want to add the **user input** now, and we use a **while loop** to divide the flow of the program:

```
def sentence_maker(phrase):
    interrogatives = ("who", "what", "where", "when", "why", "how")
    capitalized = phrase.capitalize()
    if phrase.startswith(interrogatives):
        return "{}?".format(capitalized)
    else:
        return "{}.".format(capitalized)

results = []
while True:
    user_input = input("Say something: ")
    if user_input == "\end":
        break
    else:
        results.append(sentence_maker(user_input))

print(results)
```

- Our outputs at this stage are still in the form of lists. Lists of phrases that have been properly formatted, but still lists. We want strings.
    - → [‘Weather is good.’, ‘How are you?’]

Making the Output User-Friendly:

- Now we want to concatenate all these strings using the **.join()** method.
- The example he ran in the Python interactive shell was:
  - >>> **"-".join(["how are you", "good good", "clear clear])**
  - → 'how are you-good good-clear clear'
- The **.join()** method joins items together in a string, with whatever is in between the quotation marks separating the items:

```
def sentence_maker(phrase):
    interrogatives = ("who", "what", "where", "when", "why", "how")
    capitalized = phrase.capitalize()
    if phrase.startswith(interrogatives):
        return "{}?".format(capitalized)
    else:
        return "{}.".format(capitalized)

results = []
while True:
    user_input = input("Say something: ")
    if user_input == "\end":
        break
    else:
        results.append(sentence_maker(user_input))

print(" ".join(results)) ← ← ←
```

- Here we used **" ".join(results)** to turn the list of formatted phrases into a string, with a space in between them all.

Section 9: List Comprehensions:

Section Introduction:

- Primary difference between List Comprehensions and for-loops is that List Comprehensions are written in a single line while for-loops are written in multiple lines.
- They're a special case of for-loops that are used when you want to construct a list.

Simple List Comprehension:

- The first example here involves presenting a list of temperatures in Celsius, but without the decimal points. This is often down to save disk space.

- Here's how a list of temperatures would be re-calculated to add decimal points using a for-loop:

```
temps = [221, 234, 340, 230]

new_temps = []
for temp in temps:
    new_temps.append(temp / 10)

print(new_temps)
```

- However, there's a neater way to accomplish this using just a single line of Python code:

```
temps = [221, 234, 340, 230]

new_temps = [temp / 10 for temp in temps]

print(new_temps)
```

- Much neater. There's an in-line for-loop in the new_temps list.

List Comprehension with If Conditional:

- Similar to previous, but in this case we include some invalid data (-9999). We want to ignore this one.

```
temps = [221, 234, 340, -9999, 230]

new_temps = [temp / 10 for temp in temps if temp != -9999]

print(new_temps)
```

More Examples:

- Define a function that takes a list of both strings and integers and only returns the integers.
  - Ex.: **foo([99, 'no data', 95, 94, 'no data'])** returns **[99, 95, 94]**:

```
def foo(data):
    new_data = [item for item in data if isinstance(item, int)]
    return new_data
```

- Define a function that takes a list of numbers and returns the list containing only the numbers greater than 0.
  - Ex.: **foo([-5, 3, -1, 101])** returns **[3, 101]**:

```
def foo(data):
    new_data = [item for item in data if item > 0]
    return new_data
```

List Comprehension with If-Else Conditional:

- If you want to add an **else** statement in list comprehension (such as "if number != -9999 else 0") the order is a little different from what we're used to in if-else conditionals.

```
temps = [221, 234, 340, -9999, 230]

new_temps = [temp / 10 if temp != -9999 else 0 for temp in temps] ← ← ←

print(new_temps)
```

- Need to get used to this order more often.

More Examples:

- Define a function that takes a list of both numbers and strings, and returns numbers or 0 for strings:

```
def foo(data):
    new_data = [item if isinstance(item, int) else 0 for item in data]
    return new_data
```

- Define a function that takes a list containing decimal numbers as strings, then sums those numbers and returns a float:

```
def foo(data):
    new_data = [float(item) for item in data]
    return(sum(new_data))
```

Cheatsheet: List Comprehensions:

In this section, you learned that:

- A list comprehension is an expression that creates a list by iterating over another container.
- A **basic** list comprehension:
  ```
  1. [i*2 for i in [1, 5, 10]]
  ```
  Output: `[2, 10, 20]`
- List comprehension with **if** condition:
  ```
  1. [i*2 for i in [1, -2, 10] if i>0]
  ```
  Output: `[2, 20]`
- List comprehension with an **if and else** condition:
  ```
  1. [i*2 if i>0 else 0 for i in [1, -2, 10]]
  ```
  Output: `[2, 0, 20]`

Section 10: More About Functions:

Functions with Multiple Arguments:

- Separate the parameters with a comma while defining the function (basic stuff).
- Calling the function will now take two arguments.

Default and Non-default Parameters and Keyword and Non-keyword Arguments:

- Example of a function with "default parameters" set:
  - **def area(a, b = 6)**
  - You can also manually assign a new value for **b** even if there's a default setting
- Example of function being called with "keyword arguments":
  - **print(area(a = 4, b = 5)**
  - Also called "non-positional arguments".
  - A "positional argument" would be where there's no keyword and the position of the argument defines its meaning, i.e. **print(area(4, 5))**.
    - **print(area(b= 5, a = 4)** also works.

Functions with an Arbitrary Number of *Non-keyword* Arguments:

- Some built-in functions take a specific number of arguments:
  - **len()** takes exactly 1 argument.
  - **isinstance()** takes exactly 2 arguments.
- Other built-in functions can take an arbitrary number of arguments:
  - **print()** can take any number of arguments.
- In this lecture, we're going to create a function that can take any number of arguments when called.
- Do define a function like this, we use the syntax:
  - **def mean(*args):**
  - "args" is a pretty standard name for this, that almost all Python programmers use.
  - If we simply **return args**, we get a tuple back that's full of the arguments we passed in.
  - Note that keyword arguments would not work in this situation.

```
def mean(*args): ← ← ←
    return sum(args) / len(args)

print(mean(1, 3, 4))
```

More Examples:

- Define a function that takes an indefinite number of strings and returns an alphabetically sorted list containing all the strings converted to uppercase:

```
def foo(*args):
    words = [word.upper() for word in args]
    return sorted(words)
```

- Or:

```
def foo(*args):
    words = []
    for word in args:
        words.append(word)
    return sorted(words)
```

-

Functions with an Arbitrary Number of *Keyword* Arguments:

- In the previous case we defined our function with **def mean(*args)**.
- The case with keyword arguments is similar:
    - **def mean(**kwargs):** with "kwargs" being a standard convention.
    - However, this takes keyword arguments only. Unnamed arguments will cause an error.
    - Returning these arguments gives us a **dictionary** with the **keyword names** being the '**keys**' and the **arguments** being the '**values**'.
    - Running **print(func(**kwargs(a=1, b=2, c=3))** yields **{'a': 1, 'b': 2, 'c': 3}**.
    -
- Functions with an arbitrary number of keyword arguments are *more rarely* used than functions with an arbitrary number of non-keyword arguments.

Section 11: File Processing:

Section Introduction:

- Storing data *outside* Python in external files.
- Text files, .csv files, databases.

Processing Files with Python:

- He had created a text file called **fruits.txt** containing:
    - pear
    - apple
    - orange
    - mandarin
    - watermelon
    - pomegranate
- In the next lecture, we'll use Python to *read* this file.

Reading Text from a File:

- My Python file, **file-process.py** is in the same directory as my copy of **fruits.txt**.
- The code to open this file is:

```
myfile = open("fruits.txt")
print(myfile.read())
```

- The argument in the **open()** method is the filepath for the .txt file. In this case, just giving the name of the .txt file should be enough because both files are in the same directory.
- Note: I couldn't get it to work at first, even though both files were in the same directory for Section 11. I ended up running "**pwd**" in bash and it turns out my ***working directory*** was one level up, so I ran "**cd**" to get into the directory both were saved in.

File Cursor:

- The cursor starts at the first character of the file we're reading in, and goes through to the end of the file.
- At the end of reading a file, the cursor is at the end of the file. Running **print(myfile.read())** on two or more lines of code won't do anything.
- What you could do instead is to save **myfile.read()** into a variable, and then you can print out that variable multiple times instead.

```
myfile = open("fruits.txt")
content = myfile.read()

print(content)
print(content)
print(content)
```

Closing a File:

- When you create a file object, a file object is created in RAM. It's going to remain there until your program ends.
- Therefore, it would be a good idea to close the file at the end of the program.

```
myfile = open("fruits.txt")
content = myfile.read()
myfile.close()

print(content)
```

- However, there's also a better way to do this, which we'll cover in the next lecture.

Opening Files Using "with":

- Using the **with** method does all the opening, reading, and closing as a block:

```
with open("fruits.txt") as myfile:
    content = myfile.read()

print(content)
```

Different Filepaths:

- For this, we'll be moving **fruits.txt** to another directory.
- We need to add the filepath into our **open()** function:

```
with open("files/fruits.txt") as myfile:
    content = myfile.read()

print(content)
```

Writing Text to a File:

- We started by running the **help(open)** function to see its attributes.
- The first two are most important: **file** and **mode='r'** (meaning the default mode is "read").
- We're going to create a new file, **vegetables.txt** using the "w" write option.

```
with open("files/vegetables.txt", "w") as myfile:
    myfile.write("Tomato\nCucumber\nOnion\n")
    myfile.write("Garlic")
```

- Note: If the filename already exists, Python will overwrite the existing file.
- The special character **\n** is useful to make sure items are written on new lines.

More Examples:

- Define a function that takes a single string **character** and a **filepath** as parameters and returns the **number of occurrences** of that character in the file:

```
def foo(character, filepath):
    count = 0
    with open(filepath) as myfile:
        content = myfile.read()
    for char in content:
        if char == character:
            count += 1
        else:
            pass
    return count
```

Appending Text to an Existing File:

- We want to add two more lines to our existing **vegetables.txt** file. It currently has:
  - Tomato
  - Cucumber
  - Onion
  - Garlic
- If you look at the **help(open)** documentation and scroll down, you'll see an option to set the mode argument to **"x"** ("create a new file and open it for writing"). Unlike the "w" option, this will not overwrite a file if it already exists.
- There's also a mode argument **"a"** ("open for writing, appending to the end of the file if it exists"). We're going to use this to add **Okra** to the list:

```
with open("files/vegetables.txt", "a") as myfile:
    myfile.write("Okra")
```

- Running this adds Okra to the end of the existing file, but not on a new line. The last line will read as "GarlicOkra". There wasn't a break-line ("**\n**") in the existing file.
- To fix this, we change the code to:

```
with open("files/vegetables.txt", "a") as myfile:
    myfile.write("\nOkra")
```

- He then showed us an example of trying to append and *read* right after. However, since we set the mode to **"a"**, we can't read, and we get an error.
- To get around this we look in the **help(open)** documentation and see an add-on option **"+"** ("open a disk file for updating (reading and writing)").

```
with open("files/vegetables.txt", "a+") as myfile: ← ← ←
    myfile.write("\nOkra")
    content = myfile.read()

print(content)
```

- However, just running this doesn't print anything out. We need to add something else as well: the **.seek(0)** method to put the cursor at the zero position again:

```
with open("files/vegetables.txt", "a+") as myfile:
    myfile.write("\nOkra")
    myfile.seek(0) ← ← ←
    content = myfile.read()

print(content)
```

- The cursor goes back to the beginning, and then reads down to the end of the file.

Cheatsheet: File Processing:

In this section, you learned that:

- You can **read** an existing file with Python:

```
with open("file.txt") as file:
    content = file.read()
```

- You can **create** a new file with Python and **write** some text on it:

```
with open("file.txt", "w") as file:
    content = file.write("Sample text")
```

- You can **append** text to an existing file without overwriting it:

```
with open("file.txt", "a") as file:
    content = file.write("More sample text")
```

- You can both **append and read** a file with:

```
with open("file.txt", "a+") as file:
    content = file.write("Even more sample text")
    file.seek(0)
    content = file.read()
```

Section 12: Modules:

Section Introduction:

- This section is about importing functions/modules/libraries from elsewhere.

Resources for This Section:

- **"Time" Documentation**
  - https://docs.python.org/3/library/time.html
- **OS Documentation**
  - https://docs.python.org/3/library/os.html
- **Pandas Documentation**
  - https://pandas.pydata.org/docs/
- **temps_today.csv** for download, saved to Section 12 folder.

Built-in Modules:

Note: Resource for this lecture is a link to "Time" Documentation on Python's website.

- We can search built-in **methods** using **dir(str)** for example.
- We can search built-in **functions** using **dir(__builtins__)**.
- Running the following code will print the contents of "vegetables.txt" forever:

```
while True:
    with open("files/vegetables.txt") as file:
        print(file.read())
```

- "Tomato" will be printed to the console forever at a speed that depends on your processor.
- However, what if we don't want this to happen? What if we want to read the content every 10 seconds instead?
- Checking **dir(__builtins__)** shows that we don't have any built-in functions that can do that.
- However, we can check built-in modules with the following syntax in the Python interactive shell:
    - **>>> import sys**
    - **>>> sys.builtin_module_names**
    - This gives us a list of built-in module names, which includes one called "**time**". We then run:
    - **>>> import time**
    - Running **dir(time)** shows that it has a **.sleep()** method.
    - Running **help(time.sleep)** shows us that it can be used by passing the number of seconds into the parenthesis.
    - Running **time.sleep(3)** pauses the script/command line for a count of 3 seconds.

- It's good practice to import modules at the very beginning of Python scripts:

```
import time ← ← ←

while True:
    with open("files/vegetables.txt") as file:
        print(file.read())
        time.sleep(10) ← ← ←
```

- Importing **time** and then adding **time.sleep(10)** causes our program to print out the files contents every 10 seconds.
- We tested this by changing "Tomato" to "Onion" and then "Garlic" between these 10-second intervals. The updated file contents were printed out each time.

- Not everything comes as a **built-in** module, however. In the next few lectures, we'll discuss how to import modules/libraries from other sources.

Standard Python Modules:

Note: Resources for this lecture are links to "Time" and "OS" Documentation on Python's website.

- He showed that if you delete the file that's being read before the next 10-second interview is up, you get an error (duh) and your script will stop running.
- What if we want to keep running the script even if the file is no longer there?
- To do that, we're going to make use of the **OS module**.
    - You'll notice that **os** isn't among the built-in Python modules listed when we run **sys.builtin_module_names**.
    - To find out where **os** lives, we can run **sys.prefix** in the Python interactive shell, which will give us a filepath. It may be different depending on which operating system one is running.
    - Navigating to that directory by typing **start <filepath location>** (for Windows) will open a File Explorer window for that location. For Max or Linux, use **open <filepath location>** instead of "start".
        - Note: My file structure looked a lot different than his Mac version, so it may take me some extra time to find out where my stuff is compared to his.
    - In that folder, go into "**Lib**". There's a list of .py files here, and **os** is among them.
    - If we open **os** in our IDE, we see that it's Python code. Note: Don't make any changes to Python standard files.

- We can also use **dir(os)** to see what methods it has available.
- From this list, we're going to use **path**.
- Running **os.path.exists("files/vegetables.txt")** will check if our file exists and returns True or False. We can make use of that fact in our Python program.
- What we want to do is, before opening our "vegetables.txt" file in "read" mode, we want to check if it exists. If we don't do that and the file gets deleted, we're going to get an error.
- We'll create an **if-else** conditional using **os.path.exists("files/vegetables.txt")** to handle situations where the file doesn't exist or gets deleted.

```
import time
import os

while True:
    if os.path.exists("files/vegetables.txt"):
        with open("files/vegetables.txt") as file:
            print(file.read())
    else:
        print("File does not exist.")
    time.sleep(10)
```

- Note: We want the **time.sleep(10)** method outside of the if-else conditional because we want it to run that way no matter what.
- We then let the script run while we variously deleted and recreated the file.

Third-Party Modules:

Note: Resources for this lecture are links to "Time", "OS", and Pandas Documentation. Pandas is a third-party library. We'll also use the "temps_today.csv" we downloaded.

- We previously played with our simple "vegetables.txt" file, but what if we want to do work on real-world data? For this lecture, we'll be working on the *"***temps_today.csv**" file.
    - In our CSV file, we have two weather stations and the temperatures that each one recorded.
    - We're going to read our CSV file, but instead of printing out its contents, we're going to print out an average value of all the values every 10 seconds (in real life we'd do this every 24 hours).
- So far we've only loaded data as a string, but in this case we'll be working with **floats**. We could do some operations to split and convert the data from strings to floats, but that would be like reinventing the wheel. So instead of that, we're going to **import pandas**. Pandas doesn't come by default with Python, so we import it this way:
    - In **bash**, we run **pip3 install pandas** to install it.
    - "**pip**" is a Python library that's used to install other Python libraries. You may have to run **pip3.8**, **pip3.9**, **pip3.10** depending on the version you're using.
- Rather than being a *module*, pandas is a collection of modules, which we call a "*package*".

Third-Party Module Example:

*Note: You have to be running in an **Anaconda environment** to get pandas to work. I had to go into View → Command Palette → search for "Python: Select Interpreter" and choose one from the list.*

- After importing **pandas**, we read in the CSV data into a variable, and then we can play around with printing out the **mean**:

```
import time
import os
import pandas ← ← ←

while True:
    if os.path.exists("files/temps_today.csv"):
        data = pandas.read_csv("files/temps_today.csv") ← ← ←
        print(data.mean()) ← ← ←
    else:
        print("File does not exist.")
    time.sleep(10)
```

- We can also print out the average for just one of the weather stations with a minor change:

```
import time
import os
import pandas

while True:
    if os.path.exists("files/temps_today.csv"):
        data = pandas.read_csv("files/temps_today.csv")
        print(data.mean()["st1"]) ← ← ←
    else:
        print("File does not exist.")
    time.sleep(10)
```

- What pandas is doing with the CSV is, it's creating its own object/datatype called a **DataFrame**.
  - Running **>>> type(data)** on our "data" variable returns:
  - **<class 'pandas.core.frame.DataFrame'>**

Cheatsheet: Imported Modules:

In this section, you learned that:

- **Builtin objects** are all objects that are written inside the Python interpreter in C language.
- **Builtin modules** contain builtins objects.
- Some builtin objects are not immediately available in the global namespace. They are parts of a builtin module. To use those objects the module needs to be **imported** first. E.g.:

```
1. import time
2. time.sleep(5)
```

- **A list of all builtin modules** can be printed out with:

```
1. import sys
2. sys.builtin_module_names
```

- **Standard libraries** is a jargon that includes both builtin modules written in C and also modules written in Python.
- **Standard libraries** written in Python reside in the Python installation directory as *.py* files. You can find their directory path with `sys.prefix`.
- **Packages** are a collection of *.py* modules.
- **Third-party libraries** are packages or modules written by third-party persons (not the Python core development team).
- Third-party libraries can be **installed** from the terminal/command line: Windows:

  `pip install pandas` or use `python -m pip install pandas` if that doesn't work.
- Mac and Linux:

  `pip3 install pandas` or use `python3 -m pip install pandas` if that doesn't work.

Section 13: Using Python with CSV, JSON, and Excel Files:

The "pandas" Data Analysis Library:

- A library providing data structures and data analysis tools/code within Python.
- It also has visualization tools such as **bokeh**, which we'll cover later in the course.
- **Pandas** is better than, say, just an Excel spreadsheet for analyzing a large amount of data.

Installing pandas and IPython:

- Install **pandas** using **pip3.10 install pandas**
    - I already took care of this in the last section, and had to change my interpreter to **Anaconda** to get it to work.
- Install **IPython** interactive shell using **pip3.10 install ipython**.
    - Ipython is an enhanced interactive shell that provides better printing for large text. This ability makes Ipython suitable for data analysis because the program prints data in a well-structured format.
- To use IPython, simply type "ipython" into terminal.

Getting Started with pandas:

Note: Resource for this lecture is a link to the Pandas Documentation.

- He started by running **ipython** in a Windows CMD. Interesting.
- He then mentioned **Jupyter Notebook**, which is even better at data analysis and working with data.
    - It's like a combination of a Python shell and a Python editor.
    - Browser-based.
- However, for this lecture we're keeping things simple with just **pandas**.

- Into **ipython**, we ran **import pandas** to start off.
- We then created a DataFrame variable, **df1 = pandas.DataFrame([[2, 4, 6], [10, 20, 30]])**, a list of two lists. Entering **df1** afterwards returns the formatted DataFrame.
    - Up top are the Column Names (0, 1, 2) and to the side are the indexes (0, 1).
    - "The beauty of pandas is that you can have your own column names if you like":
    - **df1 = pandas.DataFrame([[2, 4, 6], [10, 20, 30]], columns=["Price", "Age", "Value"])**
    - You can also pass custom names for the indexes:
    - **df1 = pandas.DataFrame([[2, 4, 6], [10, 20, 30]], columns=["Price", "Age", "Value"], index = ["First", "Second"])**
    - However, you won't normally need to do this for indexes; there could be hundreds or thousands or millions of them.
- There are other ways to pass in a DataFrame as well, which may be less common, such as a list of dictionaries:
    - **df2 = pandas.DataFrame([{"Name": "John"}, {"Name": "Jack"}])**: this outputs a table of the names John and Jack with "Name" as the column name.
    - If you want to add "Surname", you'd add another key-value pair:
    - **df2 = pandas.DataFrame([{"Name": "John", "Surname": "Johns"}, {"Name": "Jack"}])**
    - This returns the table with the "Surname" column added. Top entry is John Johns; second entry has "NaN" for "Surname".
- There are two basic ways to build DataFrames on the fly. However, you'll usually be pulling data from files such as CSVs, Excel files, JSON files, etc.
- You can learn more about the DataFrames you create by using:
    - **type(df1)**: returns **pandas.core.frame.DataFrame**.
    - **dir(df1)**: returns the **methods** that can be used on the DataFrame.
    - Running **df1.mean()** gives you the mean of each column.
    - Running **df1.mean().mean()** gives the mean of the entire table.
    - Typing **type(df1.mean())** returns "pandas.core.series.Series".
    - Typing **type(df1.Price)** also returns "pandas.core.series.Series".
    - A DataFrame is made of Series.

Installing Jupyter:

- To install:
  - Run **pip3.10 install jupyter**.
- To run Jupyter:
  - Type **jupyter notebook** into the terminal.
  - If that doesn't work, try **py -3.10 -m jupyter notebook**.
  - When it works, you'll see Jupyter Notebook open up in your default browser.
  - If you don't want to install Jupyter Notebook or can't install it, you can use Jupyter in the cloud. The link to this is here: https://colab.research.google.com/#create=true.

Getting Started with Jupyter:

Note: Resource for this lecture is a link to Jupyter Notebook Documentation.

- He started by downloading Jupyter in a Windows CMD prompt. To start Jupyter, he said it's good practice to first create a folder (he named his "test3"), and then **shift + right-click** inside the folder and choose "**Open Command Window Here**".
  - Note: The option I'm given is to open PowerShell instead. Let's see if that works.
  - Type **jupyter notebook** here. This opens Jupyter Notebook to your default browser.
    - Doing things this way ensures that all your Notebook files will be saved in the directory you opened the CMD/PowerShell from.
    - You can also manually **cd** into the folder you want.

- On J**upyter Notebook** in your browser, you can click on the "New" dropdown menu and select "Python 3" so that the kernel will be Python 3. If you've associated other languages with Jupyter they will also appear here.
- By default, the name of the Notebook is "**Untitled**", but you can change this to whatever you want. We renamed this to "**Testing**", and if you go to the file you opened Jupyter Notebook from you'll find a file called "**Testing.ipynb**" in there. This is an IPython Notebook extension.
  - Each input "**cell**" in Jupyter Notebook can be thought of as a line in a normal Python interactive shell, but you can type multiple lines without executing by hitting enter after each line. To execute, press **CTRL + Enter**.
  - To create a new cell, hit **ALT + Enter**.
  - To execute the current cell AND go to the next cell in one move, use **Shift + Enter**.
  - You can delete cells by hitting **ESC** and then hitting **dd** for each cell you want to delete.
- Basically we have two modes: a **command mode** (press **ESC**) when you see grey outline, and **edit mode** (press **Enter**) where it's outlined in green. In edit mode, you can go into a cell and add more lines to it.
  - You can go to **Help** and then select the **list of Keyboard Shortcuts** to see more of these shortcuts.

- If you want to re-open an existing Notebook, you go to the directory it's saved in, right-click to open CMD/PowerShell, and type **jupyter notebook** to re-open it in your browser. The file will be just as you left it.

- Jupyter Notebook is best used for doing data explorations. So if you're working with data analysis or data visualization.
- You can load data tables into it using **import pandas**. We type that into the first cell, then Shift + Enter to open a second cell and type **df = pandas.read_csv()** to read in a CSV. He used this to pull up a CSV from his computer that looked to be a well-formatted table of temperatures.
- You can also use Jupyter Notebook for **web-scraping**.
    - In the first cell, he typed:
        - **from bs4 import BeautifulSoup**
        - **import requests**
        - **print(1)**
    - And in the second cell he typed:
        - **r=requests.get(https://en.wikipedia.org/wiki/Eagle)**
        - **print(r.content)**
    - And in a third cell he had typed:
        - **soup=BeautifulSoup(r.content)**
        - **print(soup.prettify)**
    - in order to show that you can do work (scroll up and down, read, etc) on different cells without messing up any of them.

Loading CSV Files:

Note: Resources for this lecture include a link to Jupyter Notebook documentation and "supermarkets.zip".

- He opened up "**supermarkets.xlsx**" to show the data inside it (ID, address, city, etc). The same data is in 5 different files in different formats: .csv .json, .xlsx, a commas.txt, and a semicolons.txt.
- He noted that a .json file looks a lot like a Python dictionary.

- Inside the folder with all these files, we start **jupyter notebook**. All 5 of the files are listed in the tree.
- We then created a new Python 3.
    - A trick he likes to do right in the first cell is to type/run:
        - **import os**
        - **os.listdir()**
        - This gives you a list of filenames that you have in the current directory. That way you have all the names right in front of you and you don't have to switch to your directory to get those names.
- He then ran **import pandas** in the second cell.
- In the third cell he started loading all the files one by one:
    - **df1=pandas.read_csv("supermarkets.csv")**
    - **df1**
    - This output a nicely formatted table.

Exercise: Loading JSON Files:

In the previous lecture, you learned that you can load a CSV file with this code:

```
1. import pandas
2. df1 = pandas.read_csv("supermarkets.csv")
```

Try loading the `supermarkets.json` file for this exercise using `read_json` instead of `read_csv`.

The supermarkets.json file can be found inside the supermarkets.zip file attached in the previous lecture.

- Running the above and outputting it in a new cell outputted an identically formatted table as in the .csv example.

Note on Loading Excel Files:

In the next lecture, you will learn how to load Excel files in Python with *pandas*. For this, you need *pandas* (which you have already installed) and also two other dependencies that *pandas* needs for opening Excel files. You can install them with *pip*:

`pip3.9 install openpyxl` (needed to load Excel *.xlsx* files)
`pip3.9 install xlrd` (needed to load Excel old *.xls* files)

Loading Excel Files:

- Similar to the previous examples, we want to read in an .xlsx file with:
  - **df3=pandas.read_excel("supermarkets.xlsx", sheet_name=0)**
  - **df3**
- Note that you need to pass in a second argument for the sheet number. Excel files can contain multiple sheets, so to get the data from the first sheet, sheet_name=0 is needed.
- A similar table to the last few examples is output.

Loading Data from Plain Text Files:

- Data structures separated by commas (or semicolons).
- For this we run:
  - **df4=pandas.read_csv("supermarkets-commas.txt")**
  - **df4**
- This resulted in a similar table to the previous examples.
- Note that "CSV" stands for "Comma-Separated Value" or "Character-Separated Value".
- As a result of this, we can pass in a second "separator" argument:
  - **df5=pandas.read_csv("supermarkets-semi-colons.txt", separator=";")**
  - **df5**
  - However, this resulted in an error that said something like "read_csv does not have a 'separator' argument". To find out what we CAN use, he ran **pandas.read_csv?** In a separate cell. Up near the top, it listed "**sep=' a'"** as the argument name, so:
  - **df5=pandas.read_csv("supermarkets-semi-colons.txt", sep=";")**
  - **df5**
  - This resulted in a table similar to the rest. And that's all the files.

Set Table Header Row:

- He opened **supermarkets.json** and compared it to what **pandas** output to show that the header row matched between the two.
- He then pointed out that sometimes you end up with data where there's no header line. He had a "data.txt" file that was the same as the other files, but without the header row in it.
    - If you load that data.txt and print it out, whichever row is at the top is set as the header, meaning in this case that some of the data was presented in **bold** and treated like a header row.
- To get around this, you would run:
    - **df8=pandas.read_csv("data.txt", header=None)**
    - **df8**
    - This outputs a header of index numbers starting at 0.
- From here, we can then assign column names to all these numbers (shown in next lecture).

Set Column Names:

- Picking up from last lecture:
- He ran:
    - **df8.columns = ["ID", "Address", "City", "State", "Country", "Name", "Employees"]**
    - **df8**
    - Note: The order is important here.

Set Index Column:

- Sometimes you might want just a slice of the table, or a specific value within it. To do this you need to coordinate between the column and the row.
- This can be done with the automatically assigned **index numbers** on the lefthand side of the table, or even with the numbers in the **ID** column.
- From our previous example table, we run:
    - **df8.set_index("ID")**
    - Note that when you apply this method, it produces a new table with the ID as the index. However, if you output the original table again, it's back to having the old index numbers.
    - You can get around this by running: **df9=df8.set_index("ID")**
- Another way you can do this is:
    - **df8.set_index("ID", inplace=True)** to skip a step. This modifies df8 permanently.
    - However, you need to be careful with this. As an example, he ran **df8.set_index("Address", inplace=True)** to show that the "ID" column is now gone.
- However, there's a way to avoid this:
    - **df8.set_index("Name", inplace=True, drop=False)**; this keeps the column from deleting.
    - "Name" is now an index, but the "Name" column is also still present to the right.

Filtering Data from a pandas DataFrame:

- Deleting, adding, or modifying rows and columns in your DataFrame.
    - Note: At this point we're working with a DataFrame that's had its index set to be "Address" without dropping any other columns.
- How DataFrames are indexed and how you can slice/extract from them:
    - **Label-based indexing**:
        - You use the labels of rows and columns to access your data.
        - In label-based indexing you want to use the **.loc[]** method.
        - **df7.loc["735 Dolores St":"332 Hill St", "Country":"ID"]** gives you a range from one address to the other and a range from "Country" to "ID"**.**
        - You can also access single cells of your table by inputting single labels instead of ranges.
        - You can also convert the data you extract into a list:
            - **list(df7.loc[:,"Country"])** for example.
    - **Position-based indexing.**
        - You use indexes instead of label names.
        - You use the **.iloc[]** method.
        - **df7.iloc[1:3,1:4]**
        - However, similar to Python lists, this is upper-bound exclusive. The higher index gets left out.
        - You can also get all rows with **[: , 1:4]** or a single row **[3 , 1:4]**.

Deleting Columns and Rows:

- Similar to terminology I know from MySQL.
    - **df7.drop("City", 1)** to delete the "City" column. Note that this is not an in-place operation and will not update your DataFrame.
        - Pass **0** to drop a row.
        - Pass **1** to drop a column.

- You can also make changes in-place with:
    - **df7=df7.drop("332 Hill St", 0)**

- If you want to drop columns or rows based on indexing:
    - **df7.drop(df7.index[0:3], 0)** deletes the rows from index 0 to index 3.
    - **df7.drop(df7.columns[0:3], 1)** deletes the rows from index 0 to index 3.

Updating and Adding New Columns and Rows:

- Syntax for adding a **column**:
  - **df7["Continent"]=["North America"]**
  - Running that on its own gives you an error saying "length of values is not equal to length of index".
  - **df7["Continent"]=["North America", "North America", "North America", "North America", "North America"]**
  - or:
  - **df7["Continent"]=df7.shape[0]*["North America"]**
    - Note: Running **df7.shape** outputs "(5, 7)", meaning 5 rows and 7 columns. ".shape[0]" multiplies "North America" by 5 in this case, to fill it in for all 5 rows.
  - This is an in-place operation.

- Syntax for modifying a **column**:
  - **df7["Continent"]=df7["Country"]**+ ", " + "North America"
  - **df7**
  - This updates the "Continent" column to change its contents from "North America" to "USA, North America".

- Syntax for adding a new **row**:
  - In his words, "this can be a bit tricky". There's no easy method to pass a row to a DataFrame.
  - **df7_t=df7.T** where **.T** is the "*transpose*" method. This swaps your rows and columns.
  - We can now do :
  - **df7_t["My Address"]=["My City", "My Country", 10, 7, "My Shop", "My State", "My Continent"]**
  - There's now a new column with those row entries tacked onto the right end.
  - Now: **df7=df7_t.T** transposes our transposed table and updates **df7** to include the new row at the bottom.

- Syntax to modify an existing **row**:
  - You'd modify it at stage where it's in its transposed state:
  - **df7_t["3666 21st St"]=["My City", "My Country", 10, 7, "My Shop", "My State", "My Continent"]**
  - Then executing all the lines after that will update everything.

Note:

We are going to use `Nominatim()` in the next video. `Nominatim()` currently has a bug. To fix this problem, whenever you see these lines in the next video:

```
1. from geopy.geocoders import Nominatim
2. nom = Nominatim()
```

change them to these

```
1. from geopy.geocoders import ArcGIS
2. nom = ArcGIS()
```

The rest of the code remains the same.

Data Analysis Example: Converting Addresses to Coordinates:

- We're going to grab the addresses from our DataFrame and convert it into **latitude** and **longitude** coordinates.
- This process is called **geo-coding**, and its reverse is **reverse geo-coding**.
- We're going to add two columns to our DataFrame: one for latitude and one for longitude.
- **Pandas** can't do this directly, so we're going to use a library called **geopy**.
    - Run **pip3.10 install geopy**.
    - After it's installed:
    - Running **import geopy** followed by **dir(geopy)** shows that one of its module is "**geocoders**". Now, "geocoders" needs an internet connection to work, so keep that in mind. It takes the address and it sends it to an online service that has all these in a database, and it'll calculate the corresponding latitude and longitude values.
    - ~~Run **from geopy.geocoders import Nominatim**~~
    - Update:
    - **from geopy.geocoders import ArcGIS**
    - **nom = ArcGIS()**
    - **nom.geocode("3995 23rd St, San Francisco, CA 94114")**
    - This outputs a Location datatype with the full address followed by the latitude and longitude. It's rare, but sometimes you get a None object for non-existing addresses.
- You can store the result in a variable to work on it:
    - **n=nom.geocode("3995 23rd St, San Francisco, CA 94114")**
    - Running **n.latitude** outputs the latitude, **n.longitude** does longitude.

- Now how about converting an entire column of a DataFrame into a latitude and longitude?
- We'll be starting with our original .csv at this point:

```
import pandas
from geopy.geocoders import ArcGIS
nom=ArcGIS()
df=pandas.read_csv("supermarkets.csv")
df
```

- We need to construct a column or modify an existing one:
    - **df["Address"]=df["Address"] + ", " + df["City"] + ", " + df["State"] + ", " + df["Country"]**
    - **df**
- And now we see the full address printed in the "Address" column for each row. We now need to send that string to the geocode method for all those. **Pandas** allows us to do this without iterating/looping.
    - **df["Coordinates"] = df["Address"].apply(nom.geocode)**
    - **df**
- After a few seconds, this is calculated and output. My screen was too small to see the coordinates, but running **df.Coordinates[0]** shows just that portion for whichever index.

- Now, we may want to add a new column each for the latitude and the longitude:

    - **df[“Latitude”]=df[“Coordinates”].apply(lambda x: x.latitude if x != None else None)**
    - **df[“Latitude”]=df[“Coordinates”].apply(lambda y: y.latitude if y != None else None)**
    - **df**
- And we’re done!!

Section 14: Numerical and Scientific Computing with Python and Numpy:

What is Numpy?

Note: Resources for this lecture include Numpy Documentation and "smallgray.png".

- He started by zooming in on a grayscale image made up of 15 pixels ("smallgray.png").
- Each pixel has a numerical value that is converted to visual colors.
- Python stores pixels/colors/images as arrays of numbers (he went into Jupyter Notebook for this part):
    - This image could be represented as a list of three other lists (one for each row), and in each list you could have 5 different numbers (for the 5 columns).
    - This isn't the most efficient way to do this, as lists occupy lots of memory, and therefore they slow down operations on them.
    - This can be solved by **numpy** which is a Python library that provides a multidimensional array object.
- The first thing you want to do is import numpy (which should've been installed with **pandas**, because pandas is based on numpy):
    - **import numpy**
    - **n=numpy.arange(27)**
    - **n**
    - This outputs an array of "**array([0, 1, …, 26, 27])**". This is just a one-dimensional list. Checking its **type()** returns **numpy.ndarray** meaning an N-dimensional array.
    - Running **print(n)** just gives us a list of **[0, 1, …, 26, 27]**.
- Now let's see what a 2-dimensional array looks like:
    - **n.reshape(3, 9)**
    - This gives us an array of 3 lists:
        - **array([[0, 1, …, 8], [9, 10, …, 17], [18, 19, …, 26]])**
    - An example of a 2-dimensional array would be the pixels in that image (or any image).
- We could also do a 3-dimensional array:
    - **n.reshape(3, 3, 3)**
    - This gives us an array of 3 lists of 3 lists of 3:
        - See right →

```
In [15]: n.reshape(3,3,3)

Out[15]: array([[[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8]],

                [[ 9, 10, 11],
                 [12, 13, 14],
                 [15, 16, 17]],

                [[18, 19, 20],
                 [21, 22, 23],
                 [24, 25, 26]]])
```

- He noted the similarities between Numpy arrays and Python lists.
- Numpy makes it easier to iterate through these arrays. You can also make numpy arrays out of Python lists.
- To show this:

- To show this, he copied a list he manually made at the beginning of the lecture:
    - **[[123, 12, 123, 12, 33], [ ], [ ]]**
- Then he created a new object:
    - **m=numpy.asarray([[123, 12, 123, 12, 33], [ ], [ ]])**
    - **print(m)**
    - Printing this out showed they print out the same, but are different datatypes when you run **type()** on them.

Installing OpenCV:

Note: Resource for this lecture is "OpenCV Documentation".

In the next lecture, and in Section 17, we will use the OpenCV image processing library. Let us first make sure you have installed the OpenCV library. OpenCV is also referred to as `cv2` in Python.

How to Install OpenCV

To install OpenCV for Python 3.9 on **Mac** or **Linux**, execute the following in the terminal:

- `python3.9 -m pip install opencv-python`

To install OpenCV for Python 3.9 on **Windows**, execute the following in the terminal:

- `py -3.9 -m pip install opencv-python`

Note: The above commands work for Python 3.9. You may need to replace the `3.9` part from the commands with the number of the Python version you are using in your system. For example, you may need to type `python3.10` instead of `python3.9.`

Once the installation completes, open a Python session and try:

- `import cv2`

If you get no errors, you installed OpenCV successfully. If you get an error, see the FAQs below:

FAQs

1. My OpenCV installation didn't work on Windows

Solution:

1. Uninstall OpenCV with:

- `py -3.9 -m pip uninstall opencv-python`

2. Download a wheel (.whl) file from this link and install the file with pip. Make sure you download the correct file for your Windows and your Python versions. For example, for Python 3.6 on Windows 64-bit, you would do this:

- `py -3.9 -m pip install opencv_python-3.2.0-cp39-cp39m-win_amd64.whl`

3. Try to import cv2 in Python again. If there's still an error, type the following again in the command line:

- `py -3.9 -m pip install opencv-python`

4. Try importing cv2 again. It should work at this point.

2. My OpenCV installation didn't work on Mac

Solution:

If `python3.9 -m pip install opencv-python` here are alternative steps to install OpenCV:
1. Install *brew*.
To install *brew,* open your terminal, and execute the following:

- `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`

2. OpenCV depends on GTK+, so install that dependency first with brew (always from the terminal):

- `brew install gtk+`

3. Install OpenCV with brew:

- `brew install opencv`

4. Open Python and try to import OpenCV with:

- `import cv2`

If you get no errors, you installed OpenCV successfully.

3. My OpenCV installation didn't work on Linux

1. Open your terminal and execute the following commands, one by one:

```
1. sudo apt-get install libqt4-dev
2. cmake -D WITH_QT=ON ..
3. make
```

```
4. sudo make install
```

2. 2. If the above commands don't work, execute this:

```
1. sudo apt-get install libopencv-*
```

3. Then, install OpenCV with pip: `python3.9 -m pip install opencv-python`

4. Import cv2 in Python. If you get no errors, you installed OpenCV successfully.

Convert Images to Numpy Arrays:

- He started by testing that he could import cv2 correctly (let it run in its own cell):
    - **import cv2**
    - **im_g=cv2.imread("smallgray.png", 0)**:
        - **0** if you want to read the image in grayscale.
        - **1** if you want to read the image in BGR (blue-green-red).
    - **im_g**
    - This returns a 2-dimensional array, 3 lists of 5 values. Each value corresponds to one of the grayscale pixels.
    - The grayscale numbers range from 0 to 255, with 0 being pitch black and 1 being pure white. Our three white pixels are represented in the array as 255.

- However, if we change our second argument to **1**:
    - **im_g=cv2.imread("smallgray.png", 1)**:
    - **im_g**
    - We get a 3-dimensional array instead. Each of the three parts of the array is an array of 5 lists of 3 numbers. This is because the color values are bands layered on top of each other.
    - The three layers are the **blue**, the **green**, and the **red**.
    - Keep in mind that when printed out, the columns are presented horizontal and the rows are presented vertical.

- So this is how we get **numpy** arrays out of images. But what if we want to get images out of a numpy array?
    - **cv2.imwrite("newsmallgray.png", im_g)**
    - This returns **True** and creates a new image named "newsmallgray.png" in our folder.

Indexing, Slicing, and Iterating Numpy Arrays:

- This is similar to slicing a list: **a=[1,2,3]**, **a[0:1]** gives 1, **a[0:2]** gives [1,2]. With numpy arrays it's more or less the same thing, except you may have 2 or 3 dimensions.
- We'll start with **indexing** our 2-dimensional array:
  - **im_g=cv2.imread("smallgray.png", 0)**:
  - **im_g[0:2]** returns an array of the first two rows.
  - If we want to then slice the 3rd and 4th columns:
  - **Im_g[0:2, 2, 4]** returns us just those columns from those rows.
  - So slicing goes **rows first**, **then columns next**.
  - You can also use **im_g.shape** to see the shape of your array: (3, 5) or 3 rows, 5 columns.

- Next up, **iterating** over an array:

```
for i in im_g:
        print(i)
```

- This will print out the **rows**: the **i-axis is rows**.
  - You'll get **3** rows of **5** values.

- If you want to iterate through **columns**, you'd want to use **im_g.T** to transpose the array.

```
for i in im_g.T:
        print(i)
```

  - This will give you **5** rows (transposed columns) of **3** values each.

- If you want to iterate **value-by-value**:

```
for i in im_g.flat:
        print(i)
```

  - This prints out each value individually, in order.

Stacking and Splitting Numpy Arrays:

- Still working with **im_g** array from previous lecture.
- First off, we're going to *stack* two **numpy arrays**:
    - For this we're going to start by creating a new storage variable:
    - **ims=numpy.hstack((im_g, im_g, im_g))** for horizontal stack, with a tuple of numpy arrays because it can only take one argument.
        - This stacks the arrays horizontally, side-by-side, looking like a matrix that's longer in the x-direction.
    - **ims=numpy.vstack((im_g, im_g, im_g))** for vertical stack.
        - This stacks the arrays on top of each other, in the y-direction.

```
In [51]: im_g
Out[51]: array([[187, 158, 104, 121, 143],
                [198, 125, 255, 255, 147],
                [209, 134, 255,  97, 182]], dtype=uint8)

In [69]: ims=numpy.hstack((im_g,im_g,im_g))

In [71]: print(ims)
[[187 158 104 121 143 187 158 104 121 143 187 158 104 121 143]
 [198 125 255 255 147 198 125 255 255 147 198 125 255 255 147]
 [209 134 255  97 182 209 134 255  97 182 209 134 255  97 182]]

In [73]: ims=numpy.vstack((im_g,im_g,im_g))

In [74]: print ims
[[187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]
 [187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]
 [187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]]
```

- Note that if you try and concatenate arrays that have different dimensions, you'll get an error.

- Next up, we have splitting a **numpy array**:
    - We start by creating storage variable:
    - **lst=numpy.hsplit(ims, 3)** gives us an error saying "array split doesn't result in an equal division".
        - The reason for this is that the array has 5 columns.
    - **lst=numpy.hsplit(ims, 5)** gives us vertical arrays of 9 values, representing each column split off from the total array.
    -
    - **lst=numpy.vsplit(ims, 3)** gives us three vertically stacked arrays made up of three rows each of the previously stacked array.

h-split:

```
[[187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]
 [187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]
 [187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]]

In [77]: lst=numpy.hsplit(ims,5)

In [78]: lst

Out[78]: [array([[187],
                 [198],
                 [209],
                 [187],
                 [198],
                 [209],
                 [187],
                 [198],
                 [209]], dtype=uint8), array([[158],
                 [125],
                 [134],
                 [158],
                 [125],
                 [134],
                 [158],
                 [125],
                 [134]], dtype=uint8), array([[104],
                 [255],
                 [255],
                 [104],
                 [255],
                 [255],
```

v-split:

```
In [74]: print(ims)

[[187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]
 [187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]
 [187 158 104 121 143]
 [198 125 255 255 147]
 [209 134 255  97 182]]

In [79]: lst=numpy.vsplit(ims,3)

In [80]: lst

Out[80]: [array([[187, 158, 104, 121, 143],
                 [198, 125, 255, 255, 147],
                 [209, 134, 255,  97, 182]], dtype=uint8),
          array([[187, 158, 104, 121, 143],
                 [198, 125, 255, 255, 147],
                 [209, 134, 255,  97, 182]], dtype=uint8),
          array([[187, 158, 104, 121, 143],
                 [198, 125, 255, 255, 147],
                 [209, 134, 255,  97, 182]], dtype=uint8)]
```

- Note that **type(lst)** outputs that it's a Python list of numpy arrays.

Section 15: App 1: Web Mapping with Python: Interactive Mapping of Population and Volcanoes:

Demo of the Web Map:

Note: Resource for this lecture is "Webmap_datasources.zip".

- Showed off his web-based map made with **Folium**, which is a Python library.
- It has 3 layers:
    - A base map with names, etc.
    - A polygon layer that shows populations of countries.
    - Point layer for volcano locations.

Creating an HTML Map with Python:

Note: Resource for this lecture is a link to "Folium Documentation".

- Doing everything in this project strictly with Python would make Python very heavy. It doesn't have this sort of functionality.
- So we're going to use a third-party library, **Folium**, to help build this map.
    - Note: He had to redo some videos in this section because Folium had made some changes.
- Starting off, we're going to create an empty folder to store files for this project.
- From the empty folder, he right-clicked and opened an **Atom** session in that location. Not sure why he's using Atom now, but I'm going to stick with **VSCode** at first to see if I can just keep using that.
    - So far so good with VSCode (written from next lecture video).
- Run **pip3.10 install folium** to install it.
- In our interactive shell, we run **import folium**. If we don't get an error, then it installed successfully.
- We then create a map object with **map = folium.Map**. "Map" is the class that creates this object.
    - We can also check **>>> dir(folium)** to get a list of available objects that we can use.
    - We can also check **>>> help(folium.Map)** to see what we can pass to this map object.
    - Basically, it allows us to write Python code that will automatically be converted into JavaScript, HTML, and CSS code, since you need these three things to make an interactive webpage.
- So:
    - **import folium**
    - **map = folium.Map(location=[80, -100])**
    - **map.save("Map1.html")**
        - If we open this .html in a browser, it opens a web map at a random location in northern Canada, "Meighen Island". If you pan around and scroll in, you see more details of the map.
        - If you want a different starting location, you can change the coordinates. You can search a place on Google Maps, right-click and select "what's here", then copy/paste those coordinates. I think I'll change mine to **[47.608597, -122.333759]**, placing it somewhere in downtown Seattle.
    - We can also add a "Zoom" parameter:
    - **map = folium.Map(location=[47.608597, -122.333759], zoom_start=6)**
    - **map.save("Map1.html)** this zoomed the map out.

Adding a Marker to the Map:

- The default layer our map comes in with is from OpenStreetMas.
- But we can also add other **base layers** and even **point markers**.
    - ***Note:*** *This is where the note he left comes in, to use* ***tiles = "Stamen Terrain"*** *instead of tiles = "Mapbox Bright" from the Note between these two lectures*.
- Running **>>> help(folium.Map)** again and scrolling down to the "**Parameters**" section shows that we can pass in a parameter called **tiles**.
- We set this to **tiles = "Stamen Terrain"**:

```
import folium
map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")
map.save("Map1.html")
```

- Now when we open our **"Map1.html"**, we see that the background has changed, and we have a new base map.
- Now we're going to add some **point markers** on top of the base map.
    - We check **>>> dir(folium)** and we see that there's an object class called "**Marker**" and one called "**CircleMarker**". We do this by creating a "child" for our "map" object:
    - **map.add_child(folium.Marker())**
    - Now, this **.Marker()** method expects some arguments, so we run:
        - **>>> help(folium.Marker)** tells us it can take "**location**", "**popup**", and "**icon**" arguments.

```
import folium
map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")

map.add_child(folium.Marker(location=[47.9, -122.7], popup="Hi I am a Marker", →
icon=folium.Icon(color='green'))) ← ← ←

map.save("Map1.html")
```

- Now when we refresh **Map1.html**, these changes are present.

- However, he has a suggestion for adding "children" to our map object. He suggests creating a "feature group", **fg = folium.FeatureGroup(name="My Map")**; this allows us to add multiple children to our map, inside of a feature group bucket.
- We then add **map.add_child(fg)** at the end before saving, so all the children in the feature group come it at once. Keeps code more organized.

- **See on next page:**

```
import folium
map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")

fg = folium.FeatureGroup(name="My Map") ← ← ←
fg.add_child(folium.Marker(location=[47.9, -122.7], popup="Hi I am a Marker", →
icon=folium.Icon(color='green'))) ← ← ←

map.add_child(fg) ← ← ←

map.save("Map1.html")
```

Practicing “for-loops” by Adding Multiple Markers:

- We’re going to use a for-loop to add multiple markers to the map:

```
import folium
map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")

fg = folium.FeatureGroup(name="My Map")

for coordinates in [[48.00, -122.70], [47.00, -121.75]]:
    fg.add_child(folium.Marker(location=coordinates, popup="Hi I am a Marker",
    icon=folium.Icon(color='green')))

map.add_child(fg)

map.save("Map1.html")
```

Practicing File Processing by Adding Markers from Files:

Note: Resources for this lecture include links to "Folium Documentation" and "Pandas Documentation".

- We started this lecture by opening "Volcanoes.txt" (or "Volcanoes.csv" if we chose to convert it) and looking at the different fields/column names. Much of this information can be useful in making our map features, but we're especially interested in **LAT** and **LON** coordinates.
- In the Python interactive shell, he ran:
  - **import pandas**
  - **data = pandas.read_csv("Volcanoes.csv")**
  - **data**
  - This outputs all the .csv data formatted into a nice table, a **DataFrame**.
- Now he's thinking of creating two lists out of these DataFrame columns, one for latitude and one for longitude.
- We're then going to pass these into our for-loop to iterate in the "location=" variable:

```
import pandas
import folium

data = pandas.read_csv("Volcanoes.csv")
lat = list(data["LAT"])
lon = list(data["LON"])

map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")

fg = folium.FeatureGroup(name="My Map")

for lt, ln in zip(lat, lon):
    fg.add_child(folium.Marker(location=[lt, ln], popup="Hi I am a Marker",
    icon=folium.Icon(color='green')))

map.add_child(fg)

map.save("Map1.html")
```

Practicing String Manipulation by Adding Text on the Map Popup Window:

- In this lecture, we're going to add the Elevation values to the popup window for each marker.
- We extract the list and pass it into the for-loop just like the latitude and longitude values.
    - **elev = list(data["ELEV"])**
    - Then pass **lt, ln, el** in **zip(lat, lon, elev):**
        - Note: He paused the video to say that you may get a blank webpage sometimes if there are quotes (') in the strings. To avoid that, change the popup argument to:
        - **popup=folium.Popup(str(el), parse_html=True)**
        - However, this wasn't an issue in my case. Could be useful information later.

```
import pandas
import folium

data = pandas.read_csv("Volcanoes.csv")
lat = list(data["LAT"])
lon = list(data["LON"])
elev = list(data["ELEV"]) ← ← ←

map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")

fg = folium.FeatureGroup(name="My Map")

for lt, ln, el in zip(lat, lon, elev): ← ← ←
    fg.add_child(folium.Marker(location=[lt, ln], popup=str(el)+"m",
    icon=folium.Icon(color='green'))) ← ← ←

map.add_child(fg)

map.save("Map1.html")
```

Adding HTML on Popups:

Note that if you want to have stylized text (bold, different fonts, etc) in the popup window you can use HTML. Here's an example:

```
import folium
import pandas

data = pandas.read_csv("Volcanoes.txt")
lat = list(data["LAT"])
lon = list(data["LON"])
elev = list(data["ELEV"])

html = """<h4>Volcano information:</h4>
Height: %s m
"""

map = folium.Map(location=[38.58, -99.09], zoom_start=5, tiles="Mapbox Bright")
fg = folium.FeatureGroup(name = "My Map")

for lt, ln, el in zip(lat, lon, elev):
    iframe = folium.IFrame(html=html % str(el), width=200, height=100)
    fg.add_child(folium.Marker(location=[lt, ln], popup=folium.Popup(iframe), icon = folium.Icon(color = "green")))


map.add_child(fg)
map.save("Map_html_popup_simple.html")
```

You can even put links in the popup window. For example, the code below will produce a popup window with the name of the volcano as a link which does a Google search for that particular volcano when clicked:

```
import folium
import pandas

data = pandas.read_csv("Volcanoes.txt")
lat = list(data["LAT"])
lon = list(data["LON"])
elev = list(data["ELEV"])
name = list(data["NAME"])

html = """
Volcano name:<br>
<a href="https://www.google.com/search?q=%%22%s%%22" target="_blank">%s</a><br>
Height: %s m
"""

```

```
map = folium.Map(location=[38.58, -99.09], zoom_start=5, tiles="Mapbox Bright")
fg = folium.FeatureGroup(name = "My Map")

for lt, ln, el, name in zip(lat, lon, elev, name):
    iframe = folium.IFrame(html=html % (name, name, el), width=200, height=100)
    fg.add_child(folium.Marker(location=[lt, ln], popup=folium.Popup(iframe), icon = folium.Icon(color = "green")))

map.add_child(fg)
map.save("Map_html_popup_advanced.html")
```

Practicing Functions by Creating a Color Generation Function for Markers:

- So now we have a map with 62 markers for volcano locations in the US.
- However, we can convey more information if we change the colors of some of the map markers to denote elevation:
  - **Green:** 0 – 1000m
  - **Orange:** 1000 – 3000m
  - **Red:** 3000m +
- Currently we're just passing an argument to **fg.add_child()** that says **icon=folium.Icon(color='green')**. We want to replace this based on elevation.
  - Unfortunately, Folium doesn't have native functionality to do this.
  - We need to use Python code functionalities to do this.
  - We're going to use a **function**.

```
def color_producer(elevation):
    if elevation < 1000:
        return 'green'
    elif 1000 <= elevation < 3000:
        return 'orange'
    else:
        return 'red'
```

```
for lt, ln, el, name in zip(lat, lon, elev, name):
    iframe = folium.IFrame(html=html % (name, name, el), width=200, height=100)
        fg.add_child(folium.Marker(location=[lt, ln], popup=folium.Popup(iframe),
        icon=folium.Icon(color=color_producer(el))))
```

- Note: You can play around with the elevation delimiters based on the data. I decided that leaving things at "1000" produced too much orange and very little green, so I changed them to "1500" in my code.

Our Code So Far:

```
import pandas
import folium

data = pandas.read_csv("Volcanoes.csv")
lat = list(data["LAT"])
lon = list(data["LON"])
elev = list(data["ELEV"])
name = list(data["NAME"])

html = """
Volcano name:<br>
<a href="https://www.google.com/search?q=%%22%s%%22" target="_blank">%s</a><br>
Height: %s m
"""

def color_producer(elevation):
    if elevation < 1500:
        return 'green'
    elif 1500 <= elevation < 3000:
        return 'orange'
    else:
        return 'red'

map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")

fg = folium.FeatureGroup(name="My Map")

for lt, ln, el, name in zip(lat, lon, elev, name):
      iframe = folium.IFrame(html=html % (name, name, el), width=200, height=100)
      fg.add_child(folium.Marker(location=[lt, ln], popup=folium.Popup(iframe),
      icon=folium.Icon(color=color_producer(el))))

map.add_child(fg)
map.save("Map1.html")
```

Tip on Adding and Stylizing Markers:

You can use `dir(folium)` to look for possible methods of creating circle markers. Among the methods you will see `Marker`, which we previously used.

Once you locate the method, consider using the `help` function to look for possible arguments you can pass to the method for styling the circle markers.

Solution: Add and Stylize Markers:

- Changed the marker style to a **CircleMarker** with a radius of 6 pixels.

```
for lt, ln, el, name in zip(lat, lon, elev, name):
      iframe = folium.IFrame(html=html % (name, name, el), width=200, height=100)
      fg.add_child(folium.CircleMarker(location=[lt, ln], radius=6,
      popup=folium.Popup(iframe), fill_color=color_producer(el), color='grey',
      fill_opacity=0.7))
```

Exploring the Population JSON Data:

- So far we have **two layers** in our map:
    - We have the base layer with geographical information (the one we set to "Stamen Terrain"). This is our "**line layer**".
    - We also have our location markers for our volcanoes, our "**point layer**".
- We want to add a **third layer** to our map:
    - **Polygon layer**, to wrap areas in polygons.
    - We're going to set a polygon layer to show population by country.
- To do this, we use the **folium.GeoJson** object:

```
fg.add_child(folium.GeoJson())
```

- For the next part, he suggested we open the "**world.json**" file in a light-weight text editor such as Notepad to take a look at it.
    - Opening it in Atom (in his case) or VSCode (in my case) could cause problems if your computer is slow.
- There's a LOT of data in this JSON file.
- **GeoJson** is a special case of JSON. It always starts with curly-braces, and it's a string that's like a Python dictionary, with "keys" and "values".

Practicing JSON Data by Adding a Population Map Layer from the Data:

- The **folium.GeoJson()** method takes an argument "data", which we set to a classic Python function **open()**. So we add:

```
fg.add_child(folium.GeoJson(data=(open("world.json", 'r'))))
```

- Since "world.json" is in the same location as our .py file, we don't have to input the full file location.
    - Now, running the .py file with just the above line created an error, saying we need to add encoding, **encoding='utf-8-sig'** after the 'r'.
    - Also, he added a note that the recent version of Folium needs a string instead of a file as data input. Therefore, we may need to add a **read() method**:

```
fg.add_child(folium.GeoJson(data=(open("world.json", 'r', encoding='utf-8-
sig').read())))
```

- Now when we run the .py file and open/refresh our map, country borders are now outlined by blue polygon lines.
    - He noted that the GeoJson file we loaded could've had lines or points, not just polygon data.

Stylizing the Population Layer:

- We have population data in our “world.json” file.
- We’re going to change the color of our polygons based on population.
- We’re going to add a **style_function=** to our **folium.GeoJson(data=open(),)** section. This argument, “style_function” takes a **lambda function** as its argument.
    - Ex.: **l = lambda x: x**2**
    - **l(5)** returns **25**.

```
fg.add_child(folium.GeoJson(data=open("world.json", 'r', encoding='utf-8-sig').read(),
style_function=lambda x: {'fillColor':'yellow'})) ← ← ←
```

- Now when we run the .py and reload our map, the polygons are all filled with “yellow”.
- We can now play around with this by adding conditionals inside the dictionary in the lambda function:
    - **{‘fillColor’:’green’ if x[‘properties’][‘POP2005’] < 10000000 else ‘orange’ }** for example.

```
fg.add_child(folium.GeoJson(data=open("world.json", 'r', encoding='utf-8-sig').read(),
style_function=lambda x: {'fillColor':'green' if x['properties']['POP2005'] < 10000000
else 'yellow' if 10000000 <= x['properties']['POP2005'] < 20000000
else 'orange' if 20000000 <= x['properties']['POP2005'] < 30000000
else 'red'}))
```

Adding a Layer Control Panel:

- Now we want to add a feature that allows us to turn the custom layers on and off. Specifically the marker layer and the polygon layer.
- To do this, we use the **LayerControl** class of Folium:
  - **map.add_child(folium.LayerControl())**
  - Running the .py with just this makes a box appear in the upper-right of your map.
  - The box contains "**stamenterrain**", which you can't turn off, and "**My Map**" which can be toggled with a check-box.
  - Both the polygon layer and the point layer are toggled on and off at the same time, as both are held within "My Map" currently.
  - Therefore, you want to split **fg = folium.FeatureGroup(name="My Map")** into two separate parts. The polygon layer and the point layer are both added to **fg** separately already.

```
fgv = folium.FeatureGroup(name="Volcanoes")

for lt, ln, el, name in zip(lat, lon, elev, name):
    iframe = folium.IFrame(html=html % (name, name, el), width=200, height=100)
    fgv.add_child(folium.CircleMarker(location=[lt, ln], radius=6,
    popup=folium.Popup(iframe),
    fill_color=color_producer(el), color='grey', fill_opacity=0.7))

fgp = folium.FeatureGroup(name="Population")

fgp.add_child(folium.GeoJson(data=open("world.json", 'r', encoding='utf-8-sig').read(),
style_function=lambda x: {'fillColor':'green' if x['properties']['POP2005'] < 10000000
else 'yellow' if 10000000 <= x['properties']['POP2005'] < 20000000
else 'orange' if 20000000 <= x['properties']['POP2005'] < 30000000
else 'red'}))

map.add_child(fgv)
map.add_child(fgp)
```

- There are other ways to accomplish this besides splitting the feature group in two, such as adding the GeoJson to the map directly, but for the purposes of this program where we're adding multiple children to "Volcanoes" at a time, we'd have a separate layer for every volcano.
  - But for "Population", we could've added GeoJson directly.

App 1: Full Code:

```
import pandas
import folium


# This section extracts data from 'Volcanoes.csv' to iterate into map
data = pandas.read_csv("Volcanoes.csv")
lat = list(data["LAT"])
lon = list(data["LON"])
elev = list(data["ELEV"])
name = list(data["NAME"])


# This section formats popup information and adds Google link
html = """
Volcano name:<br>
<a href="https://www.google.com/search?q=%%22%s%%22" target="_blank">%s</a><br>
Height: %s m
"""

# Function to change the Map Marker color based on elevation
def color_producer(elevation):
    if elevation < 1500:
        return 'green'
    elif 1500 <= elevation < 3000:
        return 'orange'
    else:
        return 'red'


# This section creates our initial map object
map = folium.Map(location=[47.60, -122.33], zoom_start=6, tiles="Stamen Terrain")


# This line creates a feature group for volcanoes
fgv = folium.FeatureGroup(name="Volcanoes")


# This section adds Marker Point coordinates, other data to map
for lt, ln, el, name in zip(lat, lon, elev, name):
    iframe = folium.IFrame(html=html % (name, name, el), width=200, height=100)
    fgv.add_child(folium.CircleMarker(location=[lt, ln], radius=6,
    popup=folium.Popup(iframe),
    fill_color=color_producer(el), color='grey', fill_opacity=0.7))
```

```
# This line creates a feature group for population
fgp = folium.FeatureGroup(name="Population")


# This section adds polygon layer for population map
fgp.add_child(folium.GeoJson(data=open("world.json", 'r', encoding='utf-8-
sig').read(),
style_function=lambda x: {'fillColor':'green' if x['properties']['POP2005'] <
10000000
else 'yellow' if 10000000 <= x['properties']['POP2005'] < 20000000
else 'orange' if 20000000 <= x['properties']['POP2005'] < 30000000
else 'red'}))


# This line adds the feature groups for volcanoes and for population
map.add_child(fgv)
map.add_child(fgp)


# This section adds layer-control functionality to map
map.add_child(folium.LayerControl())


map.save("Map1.html")
```

Section 16: Fixing Programming Errors:

Syntax Errors:

- He claims this section/lecture is the most important one of the course.
- In Python, we have two basic types of errors:
    - **Syntax Errors**: "Parsing Errors"
    - **Exceptions**: "Runtime Errors"
- The **first line** of an error points you to the **name of the file** that has the error, then a comma. After the comma, it points you to the **line** where the error is.
- Under that, Python **prints out the line** in the terminal, **showing the error**. It even includes an arrow pointing roughly to where the error is in the line.
- Under that, you have the **error type** (such as "**SyntaxError**"). There's a description after a colon, sometimes a very useful and specific one.

- SyntaxErrors are also known as "**parsing errors**" because they're caught while your Python code is being parsed.

Runtime Errors:

- All errors that aren't Syntax Errors are **Exceptions**, such as "**TypeError**", "**NameError**", "**ZeroDivisionError**", etc.
- Note that, while you want to look at the line where the error is flagged, you also want to look at the line above it. For example, ("SyntaxError") if you forget to close a parenthesis, the error may actually be happening in the previous line because it expected a closed parenthesis.
- Python first checks for SyntaxErrors, and then looks for Exceptions.
- A **TypeError** exception can give you more useful information, such as:
    - "**TypeError: unsupported operand type(s) for +: 'int' and 'str'**".

- These are errors that occur during runtime, hence they are "**runtime errors**".

- There are many other of these error types besides TypeError.
- You may also get a **NameError**, such as if you run **print(c)** without assigning a value to the variable **c**.
    - "**NameError: name 'c' is not defined**".
- There is also the **ZeroDivisionError** for when you try and divide by 0.

How to Fix Difficult Errors:

- If you're unsure what an error message means (such as "**ZeroDivisionError: division by zero**"), you can copy the message and then Google it and looking up solutions on **Stack Overflow**.
    - If you can't find an answer, you can ask your own question.
    - Note: The *structure* of your question is very important for getting a good answer.

How to Ask a Good Programming Question:

- Include **error type**.
- Include the **expected output**.
- Include **details** about error in question.
    - Error type.
    - Entire error **traceback**.
- Include copy of the **code** you're working with.
    - Highlight the code and the error.
    - It's better to include the code as text rather than a screenshot so that another programmer can just copy/paste.

Making the Code Handle Errors by Itself:

- We're using the "divide by zero" problem again:

```
def divide(a, b):

    return a / b

print(divide(1, 0))
```

- If a user/programmer passes **print(divide(1,0))**, we'll get a **ZeroDivisionError**.
- If you have other functions or other lines of code that you want to execute as well and the user passed **0**, all the other lines wouldn't be executed, and the program would crash.
- Instead, we use **try / except**:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "You cannot divide by zero."

print(divide(1, 0))
```

- We can "except" other runtime error types as well, such as **NameError** or **TypeError**.
- Explicitly naming the specific error helps you find and fix bugs.

Section 17: Image and Video Processing with Python:

Section Introduction:

- We'll be working with **Computer Vision** in this section.
- This will work on both images and videos, as videos are more-or-less just *stacks of images*.
- We'll be using **OpenCV** ("Open-source Computer Vision"), **cv2**.

Installing the Library:

Note: Resource for this page is a link to "Cv2 Documentation".

- I previously installed OpenCV in **Section 14** to work with numpy, pandas, Jupyter Notebook, and that small grayscale image. The instructions for this page look to be identical.

Loading, Displaying, Resizing, and Creating Images:

- We want to **import cv2** into our program and then set a variable:
  - **img=cv2.imread("galaxy.jpg", 0)**
  - **0** for grayscale.
  - **1** for color.
  - **-1** for playing with transparency.
- Once **img** has been created:
  - Running **print(type(img))** returns **<class 'numpy.ndarray'>**
  - Running **print(img)** prints a list of lists of values for pixels.
    - In the case of grayscale, this is a 2-dimensional array.
  - Running **print(img.shape)** prints "**(1485, 990)**", which is the pixel width and length.
  - Running **print(img.ndim)** prints how many dimensions the array has.

```
import cv2

img=cv2.imread("galaxy.jpg", 0)

print(type(img))
print(img)
print(img.shape)
print(img.ndim)
```

- Switching our **0** argument to **1** for a color image and then running all those same print statements:
  - Returns the same class, **numpy.ndarray**.
  - A **3-dimensional array** or series of matrices.
  - "**(1485, 990, 3)**"
  - **3** (-dimensions)
- We're going to stick with the grayscale image for now.
- Running:
  - **cv2.imshow("Galaxy", img)** displays the image (named "Galaxy") on the screen.
  - **cv2.waitKey(0)** with **0** will cause the window to close as soon as the user presses any key.
    - **cv2.waitKey(2000)** would cause the image to be up on the screen for 2000 milliseconds (2 seconds).
  - **cv2.destroyAllWindows()**

```
cv2.imshow("Galaxy", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- Note that the size in which the image comes up on screen is at its **pixel size**.
-
-

- You can resize the image by adding a line before the **imshow()** method:
    - **resized_image=cv2.resize(img, (1000, 500))**
    - This shows the image in a resized (somewhat stretched) state.

```
import cv2

img=cv2.imread("galaxy.jpg", 0)

resized_image=cv2.resize(img, (1000,500))
cv2.imshow("Galaxy", resized_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- What's happening with this **.resize()** method is that Python is actually *resizing the numpy array* and creating a *new array* with dimensions (1000, 500). It will **interpolate** those values to go from one to the other.
- If you want to keep the **aspect ratio** of the image:

```
import cv2

img=cv2.imread("galaxy.jpg", 0)

resized_image=cv2.resize(img, (int(img.shape[1]/2),
int(img.shape[0]/2)))
cv2.imshow("Galaxy", resized_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- Now if we want to save/write the resized image:
    - We use **cv2.imwrite("Galaxy_resized.jpg", resized_image)**

```
import cv2

img=cv2.imread("galaxy.jpg", 0)

resized_image=cv2.resize(img, (int(img.shape[1]/2),
int(img.shape[0]/2)))
cv2.imshow("Galaxy", resized_image)
cv2.imwrite("Galaxy_resized.jpg", resized_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Exercise: Batch Image Resizing:

- Turns out this took something called a **glob**, which we hadn't gone over yet. Skipped to the Solution pages.

Solution: Batch Image Resizing:

```
import cv2
import glob
images=glob.glob("*.jpg")
for image in images:
    img=cv2.imread(image,0)
    re=cv2.resize(img,(100,100))
    cv2.imshow("Hey",re)
    cv2.waitKey(500)
    cv2.destroyAllWindows()
    cv2.imwrite("resized_"+image,re)
```

I first created a list containing the image file paths and then iterated through the aforementioned list.

The loop: reads each image, resizes, displays the image, waits for the user input key, closes the window once the key is pressed, and writes the resized image. The name of the resized image will be "resized" plus the existing file name of the original image.

```
import cv2
import glob

# Saves a glob of all images as 'images'
images = glob.glob("./resizer_images/*.jpg")

for image in images:
    img=cv2.imread(image, 0)
    resized_image=cv2.resize(img, (100, 100))
    cv2.imshow("Resized Image", resized_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    cv2.imwrite("./resizer_images/resized_"+image, resized_image)
```

- Even after following him, I can't get things to write in the relative filepath…

Solution Further Explained:

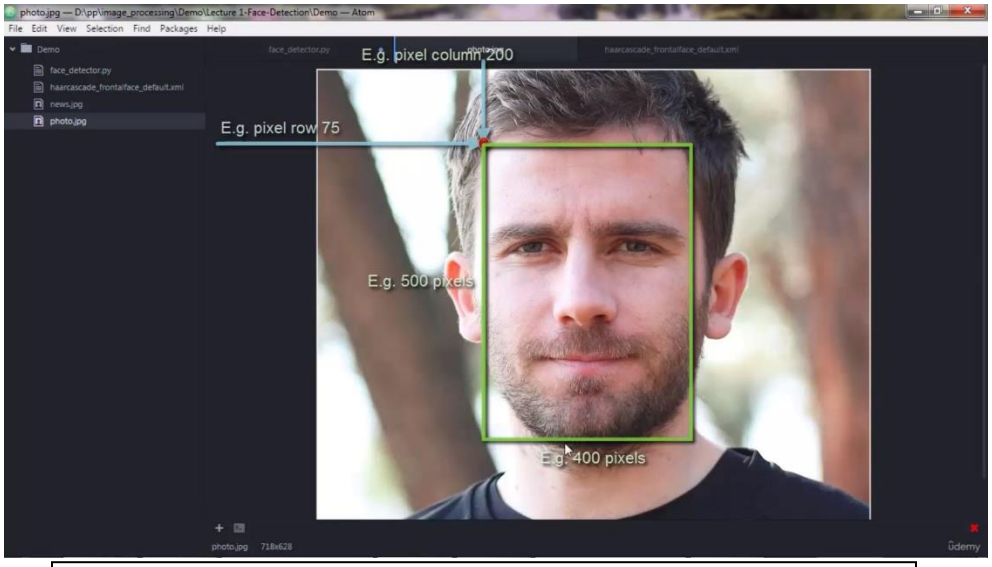
Note: Resources for this lecture include links to "Cv2 Documentation" and "Glob Documentation".

- Didn't solve my issues with getting a relative path to work (writing issues).
- Q&A didn't give me exactly what I wanted either.

Detecting Faces in Images:

Note: Resources for this lecture include "Files.zip" and link to "Cv2 Documentation".

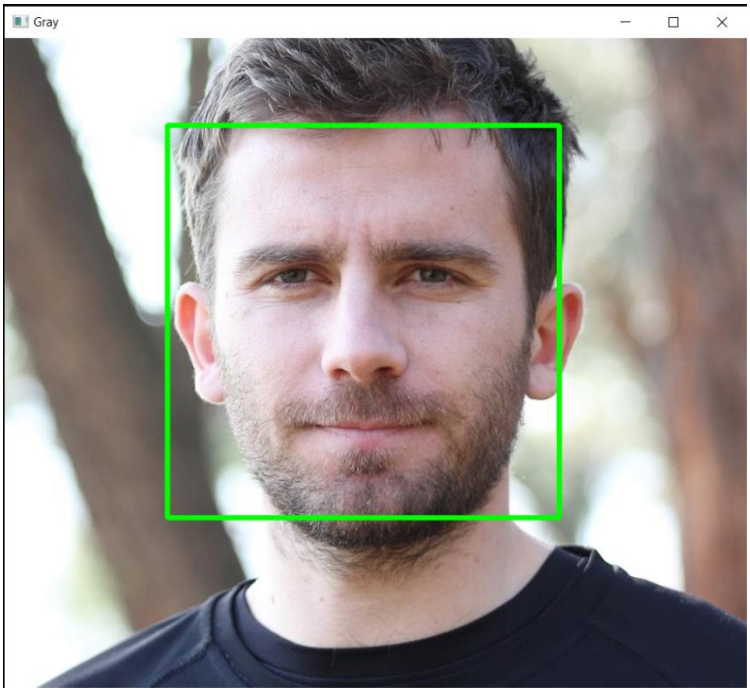
- We'll be using the **frontal face haarcascade** that was included in the .zip file.
    - More **haarcascades** for other types of images can be found on Github.
- First we **import cv2**,
- Then we set:
    - **face_cascade=cv2.CascadeClassifier("haarcascade_frontalface_default.xml")**
    - **img=cv2.imread("photo.jpg")**
    - Note: We're not passing a second argument here, meaning we'll be reading in a color picture of a face. We want to do facial recognition in grayscale, but we want to return the color version at the end:
    - **gray_img=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)**
- We're going to use a cv2 function that will give the pixel coordinates of the face it finds:




```
faces=face_cascade.detectMultiScale(gray_img,
scaleFactor=1.05,
minNeighbors=5)
```

- After running this, we decided to **print(type(faces))** and **print(faces)** to see what the result was.
    - **faces** is **<class 'numpy.ndarray'>**
    - Prints out as **[[155 83 382 382]]** for the coordinates of the above picture.

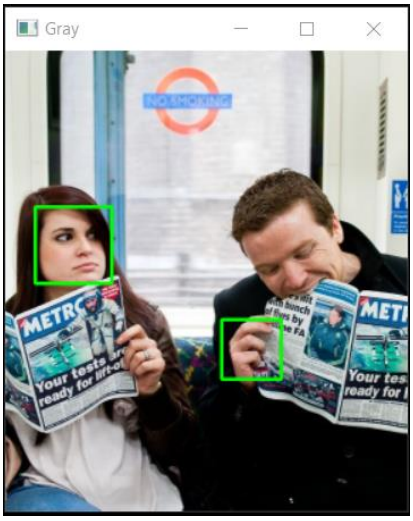
- Now we're going to go ahead and draw that rectangle on the image.



Code:

```
import cv2

face_cascade=cv2.CascadeClassifier("haarcascade_frontalface_default.xml")

img=cv2.imread("news.jpg")
gray_img=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Creates 'faces' numpy array
faces=face_cascade.detectMultiScale(gray_img,
scaleFactor=1.05,
minNeighbors=5)

for x, y, w, h in faces:
    img=cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 3)

print(type(faces))
print(faces)

resized=cv2.resize(img, (int(img.shape[1]/2), int(img.shape[0]/2)))
cv2.imshow("Gray", resized)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- Next up, we want to try our program on a more challenging picture, **“news.jpg”**.
- Just running that picture through my existing program gives this:
    - The man’s hand is flagged as a face.
    - Faces on the newspapers aren’t flagged.
- To fix this, we’re going to play around with some of the values that we input into our **faces=face_cascade.detectMultiScale()** method.
- Looking at the printout, we have coordinates for the woman’s face and for the hand: **[45 221 107 107]** and **[304 379 85 85]**.
- By changing **scaleFactor=1.05** to **scaleFactor=1.1**, we can keep the program from flagging the man’s hand.
- We might be able to detect the man’s face, but the instructor doesn’t think we’ll be able to do it with just these tools, because they have limitations.



Capturing Video with Python:

- He claims we'll either hate this lecture or we'll love it…
- We'll be reading frames/images one-by-one.
- We're going to us **globs** in this lecture.
- We can use this method to read a video either from a **webcam** or from a **video file**.
- We **import cv2**.
- We create a variable **video=cv2.VideoCapture()** which can take as an argument an **integer** (**0**, **1**, **2**, **3** for example, depending on how many cameras you have) or the **filepath** string of a video.
    - You may have more than one camera, such as a built-in camera for your computer, or an external camera. They'll have an index, such as **0**.
    - Each camera you have will have its own index.
    - Since this laptop only has **1** camera, we'll be using **index 0** as our argument:

```
video=cv2.VideoCapture(0)
```

- After you set your **video** variable, you want to **release** it:
    - **video.release()**
- Running the program at this early point doesn't appear to do anything, but (and I think this depends on your computer/camera) the camera should turn on for a second. You may see its light turn on (I didn't).
    - The **video=cv2.VideoCapture(0)** method opens the camera.
    - The **video.release()** method closes the camera a moment later.
- We can give the camera more time to be on before being released by adding **import cv2, time** to the beginning, and then adding **time.sleep(3)** before we release the camera.
    - Still no camera light for me though. I even tried 10 seconds.

```
video=cv2.VideoCapture(0)

time.sleep(3)
video.release()
```

- Now, we don't actually *see* a camera image on our screen yet because we haven't added a line telling the program to show one yet. To do this we add:
    - **check, frame = video.read()**, where **check** is a Boolean and **frame** is a numpy array.
    - We ran **print(check** and got True. This tells us that the video is running.
    - We ran **print(frame)** and got a numpy array of 3 x 3 matrices (3-dimensional array, color). This image is the first image that the video captures.

```
import cv2, time

video=cv2.VideoCapture(0)

check, frame = video.read()

print(check)
print(frame)

time.sleep(3)

video.release()
```

- We're going to **recursively** run through all the images that the camera captures.
- Next we add what we need to show the image(s):
    - **cv2=imshow("Capturing", frame)**; note: this shows only the first image that is captured.
- We also want to add a **cv2.waitKey(0)** method so pressing any key closes the window, and we want to end with a **cv2.destroyAllWindows()**.
- We also might want to create a converted grayscale version:
    - **gray=cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)**

- Now, how about we show an actual **video** instead of just a still image? To do this, we need to use a **while-loop**:

```
check, frame = video.read()

print(check)
print(frame)

gray=cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
time.sleep(3)
cv2.imshow("Capturing", gray)

cv2.waitKey(0)
```

- **^^^ The above lines need to be put inside the while loop ^^^**
- Now, using **while True** will cause a script to go on forever unless you force a stop.
- However, in this case, the **cv2.waitKey(0)** gives us a way out. However, using that will still only produce a single (first) image.
- Changing to **cv2.waitKey(2000)** updates the image window ever 2000 milliseconds (or 2 seconds), but then we're in an infinite while-loop until we force a stop.
- The way around this is to force the while loop to check if a specific key has been hit at the end of every loop, and then **break** if it has:

```
    if key==ord('q'):
        break
```

- We can also change the **waitKey** to **1000** or another value, so it refreshes more often.
    - **Note:** You have to click on the image window before pressing 'Q' to get the quit function to work.
    - Setting **waitKey(1)** gives really good resolution.

- If you want to know how many frames are being generated, there's another trick we can use.
- We can set a **frame_count = 1** and then add 1 for every loop, then print it out at the end. We got about 51 frames within 3 seconds.

Finished Code:

```
import cv2, time

# Captures video from webcam
video=cv2.VideoCapture(0)

frame_count = 1
#
while True:
    frame_count += 1

    # A Boolean and a numpy array
    check, frame = video.read()

    print(check)
    print(frame)

    # Creates grayscale version, opens image in window
    gray=cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    #time.sleep(3)
    cv2.imshow("Capturing", gray)

    key=cv2.waitKey(1)

    # Pressing the 'q' key will exit the loop
    if key==ord('q'):
        break

print(frame_count)
video.release()
cv2.destroyAllWindows()
```

Section 18: App 2: Controlling the Webcam and Detecting Objects:

Demo of the Webcam Motion Detector App:

- The demonstration showed a video window that appeared to use things we learned in last section's last two lectures: "Detecting Faces in Images" and "Capturing Video with Python".
- The program is going to record video from the webcam and **detect motion**.
- It's also going to record the time that motion was detected and **apply a timestamp**, both to when the object entered the video frame and when the object left the video frame.
- There appeared to be three different frames used to calculate/represent this data:
    - **Color Frame**
    - **Threshold Frame**
    - **Delta Frame**
- Once you press 'Q' for 'quit' at the end, it loads an interactive "**Motion Graph**" to give you a visual idea of when motion was detected. This graph appears to have been constructed with **Bokeh**.
- He notes that this sort of program can be loaded into a **Raspberry Pi**.

Detecting Moving Objects from the Webcam:

- We start with a ~~**motion-detector.py**~~ program that's identical to our finished code from the "Capturing Video with Python" program in the last Section.
    - **Note:** Program was renamed to **motion_detector.py** in later sections because of an issue arising from the "-" symbol.
- We want to add motion detection to this, but how to we plan the program out?
- To show his concept for it, he had a folder with **four pictures** in it:
    - **Background.png (Initial Frame)** – This is the baseline image the camera is "used" to seeing. We're going to use the first frame the camera takes as a *static background*. Will be converted to grayscale for the comparison.
        - Personally I would consider this an *assumption*. What if you had a situation where the first frame had someone in it? Better to have the program compare the mean value of the frames over time.
    - **Color_Frame.png (Color Frame)** – This image showed him in the frame. The program should notice the difference from the baseline. Will be converted to grayscale for the comparison.
    - **Difference.png (Delta Frame)** – This is the difference between the grayscales of the first two images, calculated by numpy. The high-difference areas (the whiter areas) of the image are where the most motion is going on.
    - **Threshold.png (Threshold Frame)** – Tell the program "if you see a difference in the Delta Frame of more than 100 intensity, convert that to completely white pixels, convert all others to completely black.

- We're going to find the contours around the completely white areas of the Threshold Frame, then write a **for-loop** that will iterate through all the contours of the current frame. Inside that loop, we'll check if the areas inside those contours is more than 500 pixels, then consider it a moving object.
- Next we'll draw a rectangle around the contours that were greater than 500 pixels and then show the rectangle over the original (live) color image.
- We'll also record the times that the moving object entered and exited the frame.

- Now onto our code. We removed a few lines from last time that we won't need for this. Then, we want to store the first frame in a variable to compare later frames to, so up at the top we add:

```
import cv2, time

first_frame=None
```

  - **None** is a special Python value that we can use to create a variable without assigning anything to it.
- Next we need to write a conditional inside our while-loop, with a **continue** in it.

```
    if first_frame is None:
        first_frame=gray
        continue
```

- This assigns the very first frame to the variable **first_frame**. This only happens once.
- Using **continue** sends the program back to the beginning of the while-loop.
- With this in place, we can now apply the **Delta Frame**. Now, we want to go up to our **gray** variable image and apply a Gaussian blur with **gray=cv2.GaussianBlur(gray,(21, 21), 0)** to reduce noise and make things easier to calculate usefully. The tuple (21, 21) is the width and height of the Gaussian kernel, and the 0 is the standard deviation.
- Now we can calculate our **delta_frame**:

```
    delta_frame=cv2.absdiff(first_frame, gray)
```

- We also added an **imshow** to show off the delta_frame.

- Once we have this, we need to classify the delta_frame values so we can assign a threshold. Let's say if the difference of compared values is more than 30, then we classify that as a white pixel (which corresponds to a value of 255). We say "There's definitely motion going on there".
- If the difference of compared values is less than 30, then we classify it as a black pixel (which corresponds to a value of 0). "There isn't much motion going on here".
- We do the above using the **.threshold()** method of the **cv2 library:**

```
thresh_frame=cv2.threshold(delta_frame, 30, 255, cv2.THRESH_BINARY)
```

- This takes the **delta_frame**, the threshold of **30**, the new assigned value of **255** (white), and the "**threshold method**" of **THRESH_BINARY**. There are quite a few threshold methods to choose from, but we're going for binary.
- At the end of this, we also added another **imshow** for "Threshold Frame".
- However, we got an error, because the .threshold() method returns a tuple of two values. This won't input into imshow, so we need to tack a **[1]** on the end:

```
thresh_frame=cv2.threshold(delta_frame, 30, 255, cv2.THRESH_BINARY)[1]
```

- Now we want to "smooth" our threshold frame (get rid of the black holes within our white objects) using the **.dilate()** method.

```
thresh_frame=cv2.threshold(delta_frame, 30, 255, cv2.THRESH_BINARY)[1]
thresh_frame=cv2.dilate(thresh_frame, None, iterations=2)
```

- We pass in **thresh_frame**, **None** for the array (if you already have a kernel array and want things to be very sophisticated, you can use this here), and **iterations=2** for how many times we want to go through the image to remove those holes.
-
- Now we want to find the "contours" of our white threshold areas. We have two methods to choose from: "find contours" and "draw contours".
    - With the "**.findContours()**" method, we find the contours in the image and store them in a tuple.
    - With the "**.drawContours()**" method, it draws the contours in an image.

```
(cnts,_) = cv2.findContours(thresh_frame.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

- This method takes your **image** (it's a good idea to use a copy() here), an argument **cv2.RETR_EXTERNAL** for retrieving the external contours, and **cv2.CHAIN_APPROX_SIMPLE** as an approximation method that OpenCV will apply for finding the contours.

- So far, we've **iterating** through the current frame, **blurring** it, converting it to **grayscale**, finding the **delta frame**, applying the **threshold**, and then **finding all the contours** within the image.
- Next, what we want to do is we want to filter our contours. We want only contours with areas bigger than, say, 1000 pixels.
- For that, we need to iterate over our contours and **continue** over them if they're less than 1000 pixels:

```
    for contour in cnts:
        if cv2.contourArea(contour) < 1000:
            continue
        (x, y, w, h) = cv2.boundingRect(contour)
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 3)
```

Full Code, Starting on Next Page:

Full Code:

```
import cv2, time

first_frame=None
video=cv2.VideoCapture(0)

while True:
    check, frame = video.read()

    gray=cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    gray=cv2.GaussianBlur(gray, (21, 21,), 0)

    if first_frame is None:
        first_frame=gray
        continue

    delta_frame=cv2.absdiff(first_frame, gray)

    thresh_frame=cv2.threshold(delta_frame, 30, 255, cv2.THRESH_BINARY)[1]
    thresh_frame=cv2.dilate(thresh_frame, None, iterations=2)

    (cnts,_) = cv2.findContours(thresh_frame.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    for contour in cnts:
        if cv2.contourArea(contour) < 1000:
            continue
        (x, y, w, h) = cv2.boundingRect(contour)
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 3)

    cv2.imshow("Capturing", gray)
    cv2.imshow("Delta Frame", delta_frame)
    cv2.imshow("Threshold Frame", thresh_frame)
    cv2.imshow("Color Frame", frame)

    key=cv2.waitKey(1)
    print(gray)
    print(delta_frame)

    if key==ord('q'):
        break
video.release()
cv2.destroyAllWindows
```

Storing Object Detection Timestamps in a CSV File:

- Now that we have the basics of our motion-detector.py working, we may want to store data in a .csv file.
- First we want to decide where in our program the state changes from "no motion" to "motion" when an object moves into the frame.
    - Note: When I run my code, the lighting in my house gives a lot of background noise, so my output might say that there's constantly motion in the frame.
        - Changing the **if cv2.contourArea(contour) < 10000:** and turning off my dining room light helped a bit.
- We also add a variable, **status = 0**, at the beginning of our **while True:** loop and we set it to **status = 1** after the **if cv2.contourArea(contour)** line. So we have →

```
while True:
    check, frame = video.read()
    status = 0 ← ← ←
```

- → up at the top of the while-loop, and we have →

```
    for contour in cnts:
        if cv2.contourArea(contour) < 10000: ← ← ←
            continue
        status = 1 ← ← ←
        (x, y, w, h) = cv2.boundingRect(contour)
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 3)
```

- → down where the contour iteration happens.
    - Unfortunately, background noise does set mine to **print(status)** → **1** still…
    - Angling my laptop screen/camera up towards the ceiling fixed the problem. Everything's running the way it should now, with the proper **status** outputs and everything.
- We can now apply a date-time method with this. To do this, we need to figure out exactly when our **status** changes from **0** to **1**. To track this, we add a new empty list **status_list = []** up near the very top of the program:

```
import cv2, time

first_frame=None
status_list=[] ← ← ←
times=[] ← ← ← (added later when appending datetimes to status changes)
```

-
- Now we want to append the status to that list:
-
-
-

- Now we want to append the status to that list:

```
    for contour in cnts:
        if cv2.contourArea(contour) < 10000:
            continue
        status = 1
        (x, y, w, h) = cv2.boundingRect(contour)
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 3)

    status_list.append(status)
```

- We also **print(status_list)** at the end, outside the while-loop. This prints out a list of every time the status became **1** (or back to **0**). We're going to use this to track our timestamps, which we do with a conditional:

```
    status_list.append(status)
    if status_list[-1] == 1 and status_list[-2] == 0:
        times.append(datetime.now())
    if status_list[-1] == 0 and status_list[-2] == 1:
        times.append(datetime.now())
```

- This conditional checks the last two items of the **status_list** to see when it changes from **0** to **1** or from **1** to **0**. However, if we run it as is, we'll get a range error during the first loops, because our **status_list** doesn't yet have enough items to index over. We fix this with:

```
import cv2, time
from datetime import datetime

first_frame=None
status_list=[None, None] ← ← ←
times=[]
```

- We now have **datetimes** for every time an object enters or leaves the frame.
- Now, sometimes you may quit the script while an object is still in the frame, so your **times** list won't have an exit time for that final incident. To fix this, we go to the end and add a conditional inside the **if key==ord('q)** conditional:

```
    if key==ord('q'):
        if status_list==1:
            times.append(datetime.now())
        break
```

- That should add the missing timestamp to **times**.
-
- The next thing we want to do is to take our **times** list and put it in a **pandas** DataFrame, and then into a .csv file.
-

- The next thing we want to do is to take our **times** list and put it in a **pandas** DataFrame, and then into a .csv file.
- We'll need a **Start** column for when an object enters the frame and an **End** column for when an object exits the frame (or when the script ends).
- First we need to **import pandas** and set an empty pandas DataFrame:

```
import cv2, time, pandas
from datetime import datetime

first_frame=None
status_list=[None, None]
times=[]
df=pandas.DataFrame(columns=["Start", "End"])
```

- The next thing we want to do is go to the end—outside the while-loop—and iterate through all of our **times** values and append them to our DataFrame:

```
print(status_list)
print(times)

for i in range(0, len(times), 2):
    df=df.append({"Start":times[i], "End": times[i+1]}, ignore_index=True)

df.to_csv("Times.csv")
```

- This will fill in our "**Start**" column with **times[i]** datetime values and our "**End**" column with **times[i+1]** datetime values, stepping through 2 at a time. The **ignore_index=True** argument I'm unsure about. We then use **df.to_csv("Times.csv")** to export this DataFrame to a .csv.
    - Note: I got a "FutureWarning" from **pandas** saying that the **frame.append** method is deprecated and to use **pandas.concat** instead. I couldn't find an easy way to convert to this that used my existing code, kept getting errors I don't want to deal with right now.
- Note: Opening up the .csv in VSCode gives a lot more information off the bat than opening it in Excel. To see more information in Excel, you need to go into "Format" and change to a different format than the default.

Full Code:

```
import cv2, time, pandas
from datetime import datetime

# Creates empty variables for later conditionals
first_frame=None
status_list=[None, None]
times=[]
df=pandas.DataFrame(columns=["Start", "End"])

video=cv2.VideoCapture(0)

while True:
    check, frame = video.read()
    status = 0

    # Creates grayscale version and applies Gaussian blur
    gray=cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    gray=cv2.GaussianBlur(gray, (21, 21,), 0)

    # If first time running, uses first image as the Background Frame
    if first_frame is None:
        first_frame=gray
        continue

    # Creates Delta Frame to calculate differences for motion capture
    delta_frame=cv2.absdiff(first_frame, gray)


    # Creates Threshold Frame to classify differences for motion capture
    thresh_frame=cv2.threshold(delta_frame, 30, 255, cv2.THRESH_BINARY)[1]
    thresh_frame=cv2.dilate(thresh_frame, None, iterations=2)


    # Find threshold contours
    (cnts,_) = cv2.findContours(thresh_frame.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    for contour in cnts:
        if cv2.contourArea(contour) < 10000:
            continue
        status = 1
        (x, y, w, h) = cv2.boundingRect(contour)
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 3)
    status_list.append(status)
```

```
    # Records date-times for status changes
    if status_list[-1] == 1 and status_list[-2] == 0:
        times.append(datetime.now())
    if status_list[-1] == 0 and status_list[-2] == 1:
        times.append(datetime.now())


    # Shows all frames
    cv2.imshow("Capturing", gray)
    cv2.imshow("Delta Frame", delta_frame)
    cv2.imshow("Threshold Frame", thresh_frame)
    cv2.imshow("Color Frame", frame)

    key=cv2.waitKey(1)
    #print(gray) # numpy array
    #print(delta_frame) # numpy array

    if key==ord('q'):
        if status_list==1:
            times.append(datetime.now())
        break

    #print(status) # To check status changes during motion capture
print(status_list) # To check status change list
print(times) # To check datetime timestamps

for i in range(0, len(times), 2):
    df=df.append({"Start":times[i], "End": times[i+1]}, ignore_index=True)

df.to_csv("Times.csv")

video.release()
cv2.destroyAllWindows
```

-
- **Note:** Program was renamed to **motion_detector.py** in later sections because of an issue arising from the "-" symbol.
-

Section 19: Interactive Data Visualization with Python and Bokeh:

Introduction to Bokeh:

- Good for visualizing data in a browser.
- A more modern alternative to Matplotlib and Seaborn.
- Sounds like we'll mostly be working with **Jupyter Notebook** for this section.
- This section looks like it'll be a lot of exercises and static pages rather than lecture videos.
- Looks like we'll also be working with the **webcam app** again. Probably graphing motion capture based on time.

Installing Bokeh:

If you haven't installed Bokeh yet, you can easily install it with pip from the terminal:

```
pip install bokeh
```

Or you use pip3:

```
pip3 install bokeh
```

Your First Bokeh Plot:

Note: Resource for this lecture is a link to Bokeh Documentation.

- He installed Jupyter Notebook to show that process again, then opened a Jupyter session, started a new Python session inside that, and renamed the session "Basic Graph".
    - Note: He said that we could use another IDE like VSCode instead of Jupyter Notebook if we prefer, but I'm going to follow along with him in Jupyter Notebook.
- **Basic Graph code:**

```
# Making a basic Bokeh line graph
```

```
# Importing Bokeh
from bokeh.plotting import figure
from bokeh.io import output_file, show

# Prepare some data
x=[1,2,3,4,5] #Note: these lists need to be the same length
y=[6,7,8,9,10]

# Prepare the output file
output_file("Line.html")

# Create a figure object
f=figure()

# Create line plot
f.line(x,y)

# Write the plot in the figure object
show(f)
```

- This creates an HTML of the line graph, which opens in the browser (whether using Jupyter Notebook or VSCode/another IDE).
- He then went over some of the toolbar stuff (zoom, pan, etc) and showed some of the features of this *interactive graph*.

- Next up we're going to do a practice exercise for plotting triangles and circles.
- As a hint, he mentioned that we can run **dir(f)** to find out what properties our *figure* object has.

Exercise: Plotting Triangles and Circles:

- Note: We just plotted the same three points, but the *glyphs* representing those points were either a **triangle** or a **circle**. Other than that, the code was almost identical between the two. I was expecting something a little more geometrically interesting to happen, like a triangle between all three points and then a (large) circle that passes through all three points.

Using Bokeh with Pandas:

Note: Resources for this lecture are "data.csv" and documentation for Pandas and Bokeh.

- Note: Turns out you can copy/paste entire cells in Jupyter Notebook if you're in Command Mode. Neat.
- To show how to import data from a CSV file instead of Python lists, he created a CSV file from scratch. Looks like it's the same data points as when we were working with lists, just in CSV form.
- He created this in the same working directory that our Jupyter Notebook files are in (I downloaded the resources version and pasted it in mine).
- We only had to change around a few things to make this work:

```
# Making a Bokeh line graph from CSV

# Importing Bokeh and pandas
from bokeh.plotting import figure
from bokeh.io import output_file, show
import pandas

# Prepare some data
df=pandas.read_csv("data.csv")
x=df["x"]
y=df["y"]

# Prepare the output file
output_file("Line_from_csv.html")

# Create a figure object
f=figure()

# Create line plot
f.line(x,y)

# Write the plot in the figure object
show(f)
```

- The next few exercises were pretty similar.

Exercise: Plotting Education Data:

- This exercise included a link (https://pythonizing.github.io/data/bachelors.csv) to a 'bachelors.csv' file to use for this. We more-or-less plug in this new CSV into our existing Python code and then change the **x=df["x"]** (and **y**) to the new column labels "**Year**" and "**Engineering**".
- There were two ways to do this:
    - The first way was what I did. I downloaded the CSV and placed it in the same working directory, then changed things around:

```
# Prepare some data
df=pandas.read_csv("bachelors.csv") ← ← ←
x=df["Year"] ← ← ←
y=df["Engineering"] ← ← ←

# Prepare the output file
output_file("education.html")
```

    - The other option—which he used in the **Solution** page—is to paste the entire URL into **df=pandas.read_csv("URL")**, which I could see being a much simpler way to go about this (as long as you have a stable internet connection when running the code):

```
# Prepare some data
df=pandas.read_csv("https://pythonizing.github.io/data/bachelors.csv") ← ← ←
x=df["Year"]
y=df["Engineering"]

# Prepare the output file
output_file("education.html")
```

- That's a pretty cool trick.

Note on Loading Excel Files:

In the next lecture, you will learn how to load Excel files in Python with *pandas*. For this, you need *pandas* (which you have already installed) and also two other dependencies that *pandas* needs for opening Excel files. You can install them with *pip*:

`pip3.9 install openpyxl` (needed to load Excel *.xlsx* files)

`pip3.9 install xlrd` (needed to load Excel old *.xls* files)

Changing Plot Properties:

You can add a title to the plot, set the figure width and height, change title font, etc. Below is a summary of properties which can be added to change the style of the plot:

```
1. import pandas
2. from bokeh.plotting import figure, output_file, show
3.
4. p=figure(plot_width=500,plot_height=400, tools='pan', logo=None)
5.
6. p.title.text="Cool Data"
7. p.title.text_color="Gray"
8. p.title.text_font="times"
9. p.title.text_font_style="bold"
10. p.xaxis.minor_tick_line_color=None
11. p.yaxis.minor_tick_line_color=None
12. p.xaxis.axis_label="Date"
13. p.yaxis.axis_label="Intensity"
14.
15. p.line([1,2,3],[4,5,6])
16. output_file("graph.html")
17. show(p)
```

Exercise: Plotting Weather Data:

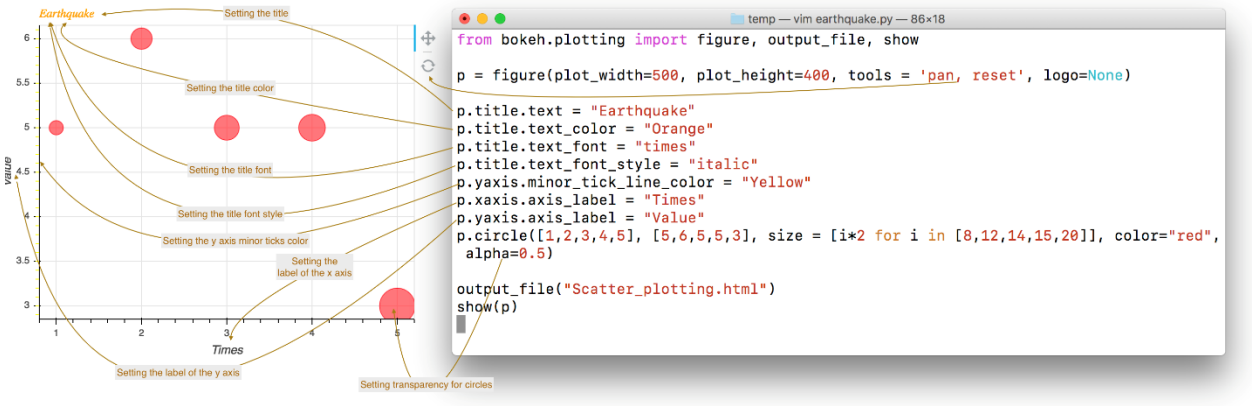
- Added the options from "Changing Plot Properties" and changed some variables to better align with our **verlegenhuken.xlsx** file that we're working with.
- Changed **x** and **y** to **df["Temperature"]** and **df["Pressure"]** as my axes from the data, divided both by **10** per the exercise hint with "**/=**" (he doesn't use many of that style of reassignment, I've noticed, like "+=" or "*=" for example).
- **Note:** Kept getting an error at the "**df=pandas.read_csv**()" part until I looked at the solution and realized I needed to change it to "**df=pandas.read_excel**()".
- Even after fixing that, kept getting errors with the version of the .xlsx file in the file path. Searched the Q&A section and heard that something about the data might be corrupted. One suggestion was to open the file in **LibreOffice Calc** and then resave it as an .xlsx from there. Downloaded LibreOffice, tried that, and it finally worked. Had to change the filepath in "**df=pandas.read_excel()**" to just "**df=pandas.read_excel("verlegenhuken_resave.xlsx",sheet_name=0)**", so it just reads the file in the same directory now, but at least it worked.

Code:

```
# Plotting weather data
from bokeh.plotting import figure
from bokeh.io import output_file, show
import pandas

df=pandas.read_excel("verlegenhuken_resave.xlsx",sheet_name=0)
df["Temperature"] /= 10
df["Pressure"] /= 10

# Set plot size settings
p=figure(plot_width=500, plot_height=400, tools='pan')
# Set plot settings
p.title.text="Temperature and Air Pressure"
p.title.text_color="Gray"
p.title.text_font="times"
p.title.text_font_style="bold"
p.xaxis.minor_tick_line_color=None
p.yaxis.minor_tick_line_color=None
p.xaxis.axis_label="Temperature (°C)"
p.yaxis.axis_label="Pressure (hPa)"

# Create plot and prepare output file
p.circle(df["Temperature"],df["Pressure"], size=0.5)
output_file("weather-data.html")
show(p)
```

Changing Visual Attributes:

Once you have built a basic plot, you can customize its visual attributes, including changing the `title` color and font, adding labels for `xaxis` and `yaxis`, changing the color of the axis ticks, etc. All these properties are illustrated in the diagram below:



And here is the code if you want to play around with it:

```
1. from bokeh.plotting import figure, output_file, show
2. p = figure(plot_width=500, plot_height=400, tools = 'pan, reset')
3. p.title.text = "Earthquakes"
4. p.title.text_color = "Orange"
5. p.title.text_font = "times"
6. p.title.text_font_style = "italic"
7. p.yaxis.minor_tick_line_color = "Yellow"
8. p.xaxis.axis_label = "Times"
9. p.yaxis.axis_label = "Value"
10. p.circle([1,2,3,4,5], [5,6,5,5,3], size = [i*2 for i in [8,12,14,15,20]],
    color="red", alpha=0.5)
11. output_file("Scatter_plotting.html")
12. show(p)
```

For a complete list of visual attributes, see the Styling Visual Attributes documentation page of Bokeh.

Creating a Time-Series Plot:

Note: Resources for this lecture are "adbe.csv", "Google+Link.txt", and the documentation links.

- We're going to use financial data found at a Google link (included in that .txt file). We're also going to be passing the full link to the **adbe.csv** file instead of just the file name (hopefully it works this time, but at least I know a fix now if it doesn't).
- He opened the **CSV** file to show 7 columns: "**Date**", "**Open**", "**High**", "**Low**", "**Close**", "**Volume**", "**Adj Close**". We're going to be using "**Date**" as our x-axis and one of the others as our y-axis.
    - Note: That was his file, labeled "Table.csv", but the downloadable version **adbe.csv** has one less column, losing the "Adj Close" column. However, the CSV is *from* the Google link.
- Note: Couldn't get link method to work, had to use the downloadable **adbe.csv** (at least I didn't have to re-save it).
- Note: Couldn't get "**p=figure(width=500, height=250, x_axis_type="datetime", responsive=True)**" to work; it said something like "could not find 'responsive' keyword". Took out the "**responsive**" keyword argument and it worked after that.
    - I followed some threads in the Q&A, and I guess the people at Bokeh changed the keyword around. Should now be "**sizing_mode="scale_both"**".

Code:

```
from bokeh.plotting import figure, output_file, show
import pandas

# Read data, parse on "Date"
df=pandas.read_csv("adbe.csv",parse_dates=["Date"])

# Create figure object with x-axis set as a 'datetime', scaling set
p=figure(width=500, height=250, x_axis_type="datetime",sizing_mode="scale_both")

p.line(df["Date"],df["Close"], color="Orange",alpha=0.5) ← ← ←

output_file("Timeseries.html")
show(p)
```

- Note: We can graph against different columns by changing the "**Close**" above to any of the others.

More Visualization Examples with Bokeh:

- He started by showing how to get multiple glyphs in one plot. It was pretty much just copy/pasting a "**p.line()**" method to create a "**p.circle()**" method and then changing some attributes around:

```
p.line([1,2,3,4,5],[5,6,5,5,3],
size=[i*2 for i in [8,12,14,15,20]],color="red",alpha=0.5)

p.circle([i*2 for i in
[1,2,3,4,5]],[5,6,5,5,3],size=8,color="olive",alpha=0.5)
```

- This created circle points that were at points 2x further along the x-axis compared to the line plot.

- For plotting different kinds of graphs, he pointed us to a section of the Bokeh documentation specifically about plotting: https://docs.bokeh.org/en/latest/docs/user_guide/plotting.html. There's a lot of useful stuff on there (the **hexbin()** in particular looks really interesting and pretty), and I got caught up reading through it for a while.
    - There's a lot of example code in the documentation that I could copy/paste and play around with in the future. That might be a good way to actually get me interested in reading documentation. Hard enough to motivate myself with the ADHD, but here's a potential way around that.
- He decided to focus on the *quadrate* or **quad()** plot, where the top, bottom, left, and right values to plot rectangles with their points. Sounds like we're going to use these plots to visualize the times from our motion detector video program.

Plotting Time Intervals from the Data Generated by the Webcam App:

- He started by showing the end result we want: a quadrate graph showing times that objects entered and exited the frame. We can also hover over the quads to show a popup with more information about them.
- Currently, ~~**motion-detector.py**~~ (see note below) outputs the datetimes into a **CSV file**. This will make it very easy to work with in Bokeh, based on all the practice examples we went through.
- Say we have 100 items in our **status_list**; this means 100 frames in this video.
- To start off though, he suggested making a minor change to our program to avoid some memory problems. He went to the section that checks the last two items of the status_list and pointed out that we don't need to keep *all* of them. We only need to keep the times *where the state changes*. So, we add →

```
    status_list = status_list[-2:]
```

- → before the conditionals checking for changes.
- Now we think about where we want to put our code for Bokeh. Now, bokeh takes a DataFrame as an input, and as it turns out we're already creating a DataFrame towards the end of the program.
- At this point, he decided to create a new program, "**plotting.py**" that will actually drive the code to plot the data.
    - **Note:** I found out you can't have a "-" in the name if you want to import from one .py program to another. You have to use "_", so I renamed that program to **motion_detector.py**:

```
from motion_detector import df
```

- From there, we make our plotting program look an awful lot like the previous **bokeh** plotting programs we've done:

```
from motion_detector import df
from bokeh.plotting import figure, show, output_file

p=figure(x_axis_type='datetime', height=100, width=500,
sizing_mode="scale_both", title="Motion Graph")

q=p.quad(left=df["Start"], right=df["End"], bottom=0, top=1, color="green")

output_file("Graph.html")
show(p)
```

- Note: I still get that FutureWarning error from the **cv2** program.
-
- Once he showed us his plot, he pointed out that we don't really need numbers or graduations on the y-axis for our purposes. He also pointed out that the x-axis was **measured in seconds**, which in this case means that's how many seconds have elapsed since the last minute-count. We don't really see the big picture from just 10 seconds of video.

- To remove the ticker on the y-axis and the y-axis part of the grid, we modify the **figure object** by adding:

```
p.yaxis.minor_tick_line_color=None
p.yaxis[0].ticker.desired_num_ticks=1
```

- So our full code so far is:

```
from motion_detector import df
from bokeh.plotting import figure, show, output_file

p=figure(x_axis_type='datetime', height=100, width=500, sizing_mode="scale_both",
title="Motion Graph")
p.yaxis.minor_tick_line_color=None
p.yaxis[0].ticker.desired_num_ticks=1

q=p.quad(left=df["Start"], right=df["End"], bottom=0, top=1, color="green")

output_file("Graph.html")
show(p)
```

- We also want to add the popup labels with more information, but we'll add those **hover** capabilities in the next lecture.

Implementing a Hover Feature:

- From **bokeh.models** we're going to import **HoverTool**. This is a built-in tool that will allow us to implement our hover feature on our graph:

```
from motion_detector import df
from bokeh.plotting import figure, show, output_file
from bokeh.models import HoverTool ← ← ←
```

- After we've created our **figure** object, we're going to add a **hover** object. This **hover** object takes an argument **tooltips** which takes a *list* of *tuples* which will contain the names we want (in this case, "Start: " and "End: "):

```
p=figure(x_axis_type='datetime', height=100, width=500,
sizing_mode="scale_both", title="Motion Graph")
p.yaxis.minor_tick_line_color=None
p.yaxis[0].ticker.desired_num_ticks=1

hover=HoverTool(tooltips=[("Start","@Start"), ("End","@End")]) ← ← ←
p.add_tools(hover)
```

- Now when the graph shows up, there are some hover tooltips over the green graph bars that say "Start: ???" and "End: ???" Now we just need to get the actual values in there. We do this by importing **ColumnDataSource** after HoverTool. This is used to provide data to a bokeh plot. For some DataFrames, objects, and functions, you need to convert them into a **ColumnDataSource** object:

```
from motion_detector import df
from bokeh.plotting import figure, show, output_file
from bokeh.models import HoverTool, ColumnDataSource ← ← ←

cds=ColumnDataSource(df) ← ← ←
```

- We then need to modify our **.quad** object down below:

```
q=p.quad(left="Start", right="End", bottom=0, top=1, color="green",
source=cds) ← ← ←
```

- Notice we don't need the **df[]** callouts in our .quad arguments anymore, just "Start" and "End".
- Now the hover tooltips return some data, but it's not properly formatted as *datetimes*, so we need to format that.

```
df["Start_string"]=df["Start"].dt.strftime("%Y-%m-%d %H:%M:%S")
df["End_string"]=df["End"].dt.strftime("%Y-%m-%d %H:%M:%S")
```

-
- We also need to change some inputs in our **hover** object to point to these formatted datetimes:

```
hover=HoverTool(tooltips=[("Start: ","@Start_string"),
("End: ","@End_string")])
```

-
- Now our finished code looks like this:

```
from motion_detector import df
from bokeh.plotting import figure, show, output_file
from bokeh.models import HoverTool, ColumnDataSource

# Changes df times into formatted datetimes
df["Start_string"]=df["Start"].dt.strftime("%Y-%m-%d %H:%M:%S")
df["End_string"]=df["End"].dt.strftime("%Y-%m-%d %H:%M:%S")

# Passes DataFrame data to ColumnDataSource object
cds=ColumnDataSource(df)

# Creates a Figure "p" as our "Motion Graph"
p=figure(x_axis_type='datetime', height=100, width=500, sizing_mode="scale_both",
title="Motion Graph")
p.yaxis.minor_tick_line_color=None
p.yaxis[0].ticker.desired_num_ticks=1

# Creates a tooltip hover function
hover=HoverTool(tooltips=[("Start: ","@Start_string"), ("End: ","@End_string")])
p.add_tools(hover)

# Formats our graph
q=p.quad(left="Start", right="End", bottom=0, top=1, color="green", source=cds)

output_file("Graph.html")
show(p)
```

Section 20: App 3 (Part 1): Data Analysis and Visualization with Pandas and Matplotlib:

Preview of the End Results:

Note: Resource for this section is "reviews.csv".

- He says this is one of the most important projects of the course.
- Python has overtaken languages such as **r** that were geared towards data analysis and visualization. The vast amounts of libraries in Python make this easy and fluid.
- He showed off a series of interactive charts for "**Analysis of Course Reviews**":
    - **Average Rating by Week**: A plotted curve.
    - **Number of Ratings by Course**: A pie chart.
    - **Average Rating by Month by Course**: Reminds me of the population statistics chart at the end of an AoE2 playthrough.

Installing the Required Libraries:

To build this app we need to install a few Python libraries. Please run the following commands in your terminal to install the correct library versions even if you have the libraries installed already:

Installing **justpy** (library for building web apps and data visualization):
`pip3.9 install justpy==0.1.5`

Installing **pandas** (library for data analysis):
`pip3.9 install pandas==1.2.2`

Installing **pytz** (library for datetime calculations between timezones)
`pip3.9 install pytz==2021.1`

Installing **matplotlib** (library for quick data visualization)
`pip3.9 install matplotlib==3.3.4`

Installing **jupyter** (library that enables a reach interactive Python shell)
`pip3.9 install jupyter`

Note: *The commands above assume you are using Python 3.9. If you are using another version of Python please change* `pip3.9` *to reflect the other version of Python you are using (e.g.,* `pip3.8`*).*

Exploring the Dataset with Python and pandas:

*Note: It seems like a lot of this section will be done in **Jupyter Notebook**.*

- We created a folder called **reviews_analysis** and placed our **reviews.csv** file inside it, then opened a **Jupyter Notebook** session inside. Then we hit "New" and chose "Python 3". We renamed our Jupyter Notebook to "**reviews**".
- We then imported pandas and set **data = pandas.read_csv("reviews.csv")**. When we run **data** in a new line, it returns a very long (truncated) .csv list of classes and reviews (45000 rows and 4 columns). If we run **data.head()**, it will print out the first (5) rows of the DataFrame only. It's good to keep the **head** of the DataFrame displayed here to give us some visual reference for what we're working with.
- Running **data.shape** in the next cell gives us the shape, or the number of rows and the number of columns (45000, 4).
- Running **data.columns** gives us the names of the columns (even though we already see that in our data.head() cell).
- Usually when we're working with data, we have some specific columns that we're interested in. In this case, we might be interested in seeing an overview of the **Rating** column. To see a histogram of the distribution of Ratings, we run **data.hist("Rating")**.
    - **Note:** Jupyter Notebook allows us to simply run this on its own, and a histogram will pop up. It does this by installing a dependency, matplotlib-inline.
    - To get it to run in another IDE (i.e. VSCode), we need to **import matplotlib.pyplot as plt**, create a new DataFrame **df = pandas.DataFrame(data)**, and we need to run **plt.hist(df['Rating'])** down below.

```
import pandas
import matplotlib.pyplot as plt

data = pandas.read_csv("reviews.csv")

df = pandas.DataFrame(data)

plt.hist(df['Rating'])
plt.show()
```

-
- In the next few lectures, we're going to zoom in on our data and look at it in more detail.

Selecting Data:

- *Note: This section and some of its terminology really reminds me of MySQL*.
- He started by opening a fresh session in Jupyter Notebook, then navigated to and opened his **reviews.ipynb** file. He wanted to point out that if you've just reopened a Jupyter Notebook file and then try and simply run **data** to access that object, you'll get a NameError.
    - This is because Jupyter Notebook treats this entire script and all its individual cells as though they haven't been run yet, including assigning something to "data".
    - To fix this, you need to execute all the cells, either by going to each cell and pressing *SHIFT+ENTER*, or go up to the **Fast Forward** button up top to run all cells.
-
- Now, he wants to add a **markdown** cell *above* the top cell. To do this, we go up to the cell and press ESC, then press "**A**" on the keyboard. Then, still not entered in this new cell, we press "**M**" to change it to a markdown cell. This cell no longer expects Python code, it expects Markdown text.
- We typed "**## 1. Overview of the dataframe**" and then CTRL+Enter to change this to a *title*. Adding more **#**s causes it to appear in a smaller font.
-
- Down below our histogram from last time, we added another Markdown cell stating "**## 2. Selecting data from the dataframe**", then:
- Created a new Markdown cell below that saying "**### Select a column from the dataframe**".
- Below that we ran **data['Rating']** to output the data from just that column. This is the first step to extracting useful data from our column, such as the *mean*.
    - **data['Rating'].mean()**
    - Note: **type(data['Rating'].mean())** outputs **pandas.core.series.Series**.
-
- We created a new Markdown cell, "**### Select multiple columns**". The method for selecting multiple columns takes a list of lists:
    - **data[['Course Name', 'Rating']]**
    - Note: **type(data[['Course Name', 'Rating']])** outputs **pandas.core.frame.DataFrame**.
-
- We created a new Markdown cell, "**### Selecting a row**".
    - **data.iloc[***index of row***]**
    - **data.iloc[3]** outputs all the info from a row.
    - Note: **type(data.iloc[3])** outputs **pandas.core.series.Series**.
-
- We created a new Markdown cell, "**### Selecting multiple rows"**.
    - **data.iloc[1:3]**, (note, this takes a slice [1:3])
    - **type(data.iloc[1:3])** outputs **pandas.core.frame.DataFrame**.
-
- We created a new Markdown cell, "**### Selecting a section**". This is a cross-section of particular columns and particular rows that will give us a slice of the DataFrame.
    - **data[['Course Name', 'Rating']].iloc[1:3]**; we can use iloc here because we're working on a DataFrame.

- We created a new Markdown cell, "**### Selecting a cell**". Let's say we want to select the specific cell that is the cross-section of the row with index 2 and the column "Timestamp".
    - **data['Timestamp'].iloc[2]**
    - Note: **type(data['Timestamp'].iloc[2])** outputs **str** in this case, but it's going to output whatever type is in a given cell, such as **float**.
- There's also a faster way to cross-section a cell:
    - **data.at[2, 'Rating']**
    - He recommends this method.

Filtering the Dataset:

- We created a Markdown cell, "**## 3. Filtering data based on conditions**", then another below called "**### One condition**".
- We want to filter the data to show where the Rating is greater than 4:
    - **data[data['Rating'] > 4]**
    - This gives us all cases where the rating is 4.5 or 5.0 (it would've included 4.0 if we'd used >= instead).
    - Using **len(data[data['Rating'] > 4])** gives us **29758**.
    - Can also use **data[data['Rating'] > 4].count()**, which gives counts for **Course Name** (29758), **Timestamp** (29758), **Rating** (29758), and **Comment** (4927).
- You can also just return a column of this DataFrame with:
    - **data[data['Rating'] > 4]['Rating']**
    - This returns the column 'Rating', but with only values of 4.5 or 5.0.
    - To clarify how this works, he set:
    - **d2 = data[data['Rating'] > 4]** to set the DataFrame to a variable.
    - Then he ran **d2['Rating']** to get the sorted column 'Rating' again.
- We can also apply methods such as **.mean()** to our sorted data:
    - **d2['Rating'].mean()** gives us the mean of all ratings of 4.5 and/or 5.0.
-
- We then created a Markdown cell, "**### Multiple conditions**".
- Let's say we want to filter for where the 'Rating' is greater than 4 and the 'Course Name' is equal to "The Python Mega Course…":
    - **data[( ) & ( )]**
    - **data[(data['Rating'] > 4) & (data['Course Name'] == 'The Complete Python…)]**
    - We can also get the mean out of this filter:
    - **data[(data['Rating'] > 4) & (data['Course Name'] == 'The Complete Python…)].mean()**
-
- In the next lecture, we're going to look at filtering a database on *times*.

Time-Based Filtering:

Note: In addition to the usual Pandas and Datetime Documentation, resources for this lecture include Pytz Documentation.

- We started with a new main section, "**## 4. Time-based filtering**".
    - **data[(data['Timestamp'] >** 1st Jul., 2020 **) & (data['Timestamp'] <** 31st Dec., 2020 **)]**
    - To do this, we'll need a datetime object, as Python isn't smart enough to parse dates from strings (for example).
    - Up at the top of our program, we need to add **from datetime import datetime**.

```
## 4. Time-based filtering
data[(data['Timestamp'] >= datetime(2020, 7, 1)) & (data['Timestamp'] < datetime(2020, 12, 31))]
```

- However, we got a **TypeError** doing this. The reason for this is that **data['Timestamp']** is a column containing **strings**. These strings need to be converted to **datetimes** to allow for comparison.
- To do this, we need to go up to the top of our program and add another argument to our **.read_csv()** method:
    - **data=pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])**
- Now when we run it, we get another TypeError because the formatting of the datetimes is wrong between the ones we're comparing. Our parsed version is in **UTC**, but the version we input later is simply a **datetime**, or a "naïve datetime object". We need to declare an explicit time system for our input datetimes:
    - First we need to go up top again and run **from pytz import utc** to get our UTC object.
    - Then:

```
## 4. Time-based filtering
print(data[(data['Timestamp'] >= datetime(2020, 7, 1, tzinfo=utc)) &
(data['Timestamp'] < datetime(2020, 12, 31, tzinfo=utc))]) ← ← ←
```

Turning Data into Information:

- So far, we've only *extracted data* from our DataFrame, but the goal of *Data Analysis* is to turn ***data*** *into* ***information***.
- For this video, we're going to answer a series of questions, and he's already gone and created a bunch of Markdown cells to divide the sections:
- **## 5. From data to information**
    - **### Average rating**
    - **### Average rating for a particular course**
    - **### Average rating for a particular period**
    - **### Average rating for a particular period for a particular course**
    - **### Average of uncommented ratings**
    - **### Average of commented ratings**
    - **### Number of uncommented ratings**
    - **### Number of commented ratings**
    - **### Number of comments containing a certain word**
    - **### Average of commented ratings with "accent" in the comment**

- ### Average rating:
    - **data['Rating'].mean()**

- ### Average rating for a particular course:
    - **data['Course Name']=='The Python Mega Course: Build 10 Real World Applications']['Rating'].mean()**

- ### Average rating for a particular period:

```
data[(data['Timestamp'] > datetime(2020, 1, 1, tzinfo=utc)) &
     (data['Timestamp'] < datetime(2020, 12, 31,
tzinfo=utc))]['Rating'].mean()
```

- ### Average rating for a particular period for a particular course:

```
data[(data['Timestamp'] > datetime(2020, 1, 1, tzinfo=utc)) &
     (data['Timestamp'] < datetime(2020, 12, 31, tzinfo=utc)) &
    (data['Course Name']=='The Python Mega Course: Build 10 Real World
Applications')
    ]['Rating'].mean()
```

- ### Average of uncommented ratings:
    - **data[data['Comment'].isnull()]['Rating'].mean()**

- ### Average of commented ratings:
    - **data[data['Comment'].notnull()]['Rating'].mean()**

- ### Number of uncommented ratings:
    - **data[data['Comment'].isnull()]['Rating'].count()**

- ### Number of commented ratings:
    - **data[data['Comment'].notnull()]['Rating'].count()**

- ### Number of comments containing a certain word:
    - If we run ~~**data[data['Comment'].str.contains('accent')]**~~ we get an error, because Python can't search **NaN** fields in the 'Comment' column for a string. So:
    - **data[data['Comment'].str.contains('accent', na=False)]** gives us what we want

- ### Average of commented ratings with "accent" in comment:
    - **data[data['Comment'].str.contains('accent', na=False)]['Rating'].mean()**

- In the next lecture, we're going to learn about **Plotting**.

Aggregating and Plotting Average Ratings by Day:

- For this lecture, we go into our main Jupyter Notebook directory in the browser and **create a new Python file there**. The first thing we need to do in this new file is to *load the DataFrame*, so to do this we go into our previous file and copy/paste the first cell into our new one.
- Before we get to work creating our graph, we need to do some **data aggregation**. We want to aggregate average ratings for a given day. We can do this by using the **pandas .groupby() method**:
    - **day_average = data.groupby(['Timestamp'])**
    - However, this alone isn't able to properly group by day, because within the 'Timestamp' column there are entries from different times of the same day.
    - To fix this, we need do some data processing beforehand:
    - **data['Day'] = data['Timestamp'].dt.date**
    - This gives our DataFrame a new column called 'Day' that we can now group by.
    - **day_average = data.groupby(['Day'])**
    - Now, 'Day' is still not aggregated, so we need to give another command to tell pandas the method of aggregation, which in this case is the mean():
    - **day_average = data.groupby(['Day']).mean()**
    - Note: If we run **type(day_average)** the output is **pandas.core.frame.DataFrame**. However, it has only one column, 'Rating'; 'Day' is not a column, it's the index. We can access it using **day_average.index**.
    - We can also convert to a list with **list(day_average.index)**, which we can use to help plot the data points.

- Now for the **plotting**. We first need to **import matplotlib.pyplot as plt**.
- Then down below our day_average we run **plt.plot()**. This method takes an **x-value** and a **y-value** as its arguments.
    - For our x-value we want the days, so **day_average.index**
    - For our y-value we want the average rating, so **day_average['Rating']**
    - So: **plt.plot(day_average.index, day_average['Rating'])**
    - Note: VSCode version requires line **plt.show()** afterward to actually show a popup of the plot.
    - The **y-axis** shows the ratings, and it goes from 3.8 to 5.0 in this case; matplotlib picks that range automatically by looking at the data.
    - The days along the **x-axis** are kind of difficult to read, but we're going to fix that with formatting later. We can work with this by declaring a **.figure()** object and giving it a size:
        - **plt.figure(figsize=(25, 3))** ← (comes in nicely in Jupyter Notebook, but the VSCode popup is too big for my screen).
    - If you still feel this graph doesn't tell you enough about the trends, we can **downsample** the data, such as with **weekly data** or **monthly data**.

- We'll learn about **downsampling** in the next lecture.

Downsampling and Plotting Average Ratings by Week:

- We used some Markdown cells to separate out a few sections. The code from last lecture went into "**### Rating average/count by day**".
- Down below last lecture's code, we created a new Markdown cell, "**### Rating average by week**". We now want to aggregate ratings by week:
    - However, running **data['Week'] = data['Timestamp'].dt.week** tries to aggregate, say, the first week of 2018 with the first week of 2019, etc. Running **data['Week'].max()** gives us **53** weeks.
    - Let's try **data['Week'] = data['Timestamp'].dt.isocalendar().week**, but running **data['Week'].max()** still gives us **53**…
    - We need to use **data['Week'] = data['Timestamp'].dt.strftime('%Y-%U')**,
        - This parses *strings* from *time* (hence "strftime") and passes that a symbol for 'Year' (**'%Y'**) and one for 'Week' (**'%U'**).
        - He then went over some other symbols, such as the symbol for 'Month' (**'%m'**).
        - You can also separate these codes with different things, such as dash ('-'), colon (':'), and even space (' ').
        - You can look up a list of "Python datetime format codes" to find more.
- We then want to create a storage variable **week_average** and set it to group-by week:
    - **week_average = data.groupby(['Week']).mean()**
    - When we print this out, the 'Week' column is the index.

- Now it's time for the **plotting**.
    - **plt.plot(week_average.index, week_average['Rating'])**
    - You'll notice that the x-axis labels for this graph are smashed up against each other, making them difficult to read, and there are ways to fix that, but the instructor says it probably isn't worth doing in matplotlib.
    - He mentions some more advanced Python plotting libraries that we'll use in coming lectures.

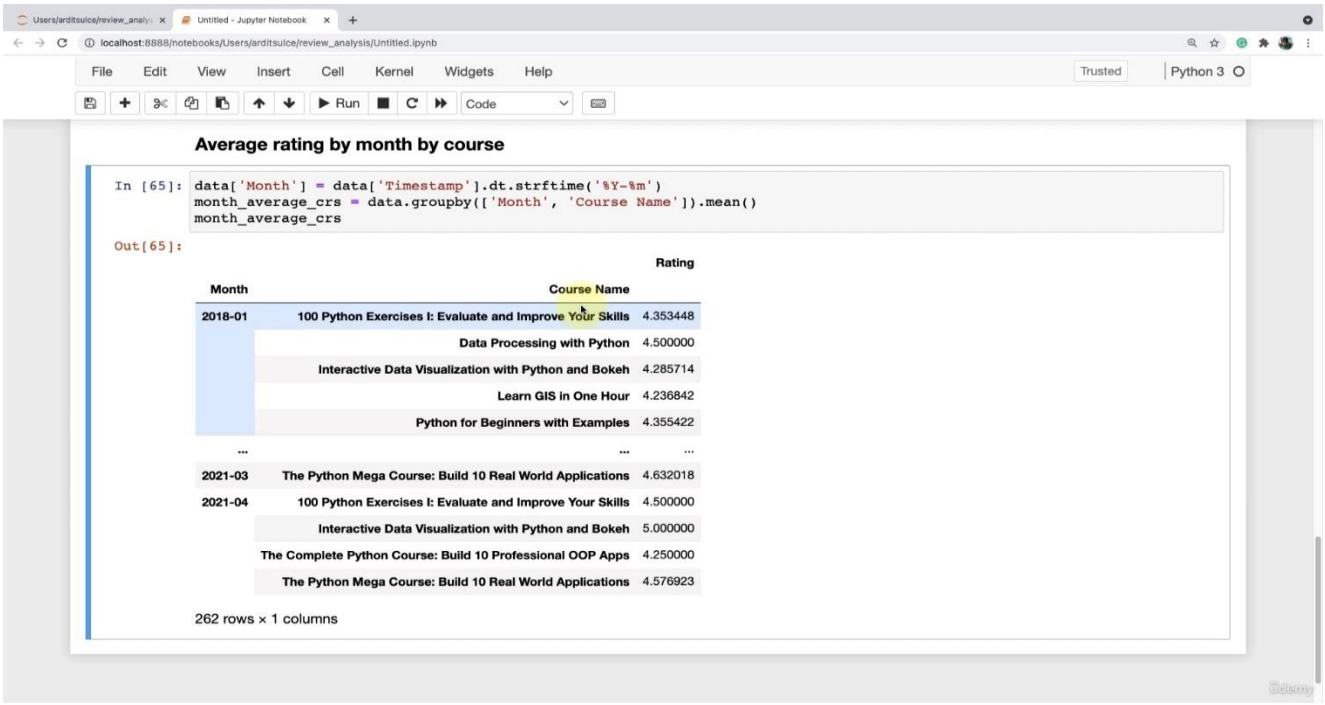
- Comparing the graph for daily averages compared to weekly averages, it becomes more apparent that **downsampling** can be better for showing trends.
- Next up, we're going to continue downsampling to show **average ratings per month**.

Downsampling and Plotting Average Ratings by Month:

- I noticed that this video was quite short (only 2 minutes long), so I decided to try my hand and programming this just with what I'd learned in previous lectures.
- I managed to get it right just by copy/pasting code from the 'Week' downsample and swapping out some arguments and variables:
    - Ran **data['Month'] = data['Timestamp'].dt.strftime('%Y-%m')** ← swapped out '%U" for 'Week' here with '%m' for 'Month'.
    - Ran **month_average = data.groupby(['Month']).mean()** to set a storage variable.
    - Ran **plt.plot(month_average.index, month_average['Rating'])** to plot it.
- I then watched through the lecture to double-check my work. Got it in one.
- You'll notice that in both the **weekly** and **monthly** graphs, the y-axis range has changed from the **daily** one.
- In the next lecture, we're going to learn how to put multiple lines in a graph to show more data.

Average Ratings by Course by Month:

- In this lecture we're going to generate a plot with different lines, and each line is going to represent the **average rating per course**.
- Starting out, we can reuse our **data['Month'] =** setup from the last lecture.
- However, we need a new variable **month_average_crs** to group by both 'Month' AND 'Course Name'. Running **month_average_crs = data.groupby(['Month', 'Course Name']).mean()** gives us this:

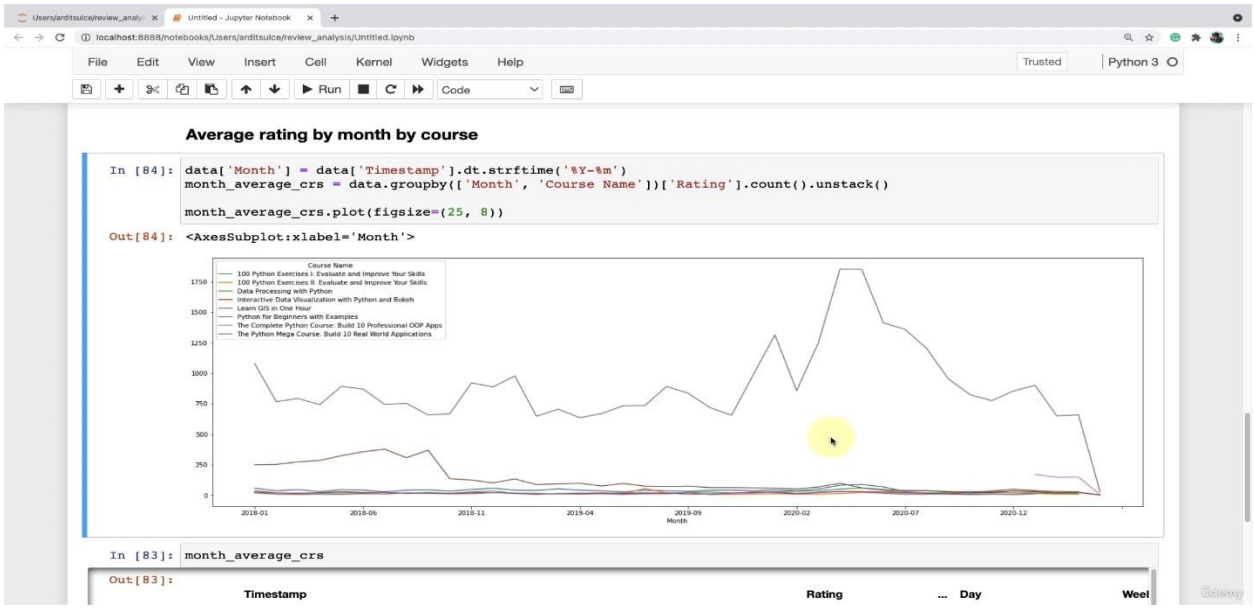


- The courses and their own averages are grouped by month now.
- It has **two indexes**: **'Month'** and '**Course Name**', and running **month_average_crs.index** shows this explicitly. Running **month_average_crs.columns** gives us only one column: '**Rating**'.

- To get this data into a better structure, we want to add an **.unstack()** method:
    - **month_average_crs = data.groupby(['Month', 'Course Name']).mean().unstack()**
- Now when we run **month_average_crs[-20:]**, we get a column for each course. You might notice that some entries are NaN; this is because a given course might not have been published yet.

- Now that we have a *useful data structure*, we can **plot it**. However, we can't use the method we used previously since we have so many more columns.
    - The **x-axis** is going to remain the '**Month**' of course.
    - The **y-axis** values will be different for every '**Course Name**'.
    - One way to do this would be to write a plot function **plt.plot()** multiple times, one for every Course.
    - Another way would be to use a loop to write separate plot functions.
    - However, a simpler way would be to just point to **month_average_crs.plot()**
    - To control figsize, we run **month_average_crs.plot(figsize=(25, 8))**

- He also showed us that if we run **.count()** instead of .mean(), the legend gets really ugly. This is because it's not just counting 'Rating', but also all of the other columns such as 'Timestamp' and 'Comment' as well. This is not very useful.
- So how do we extract just the '**Rating**' from this .count() method? Well actually it's very easy to do:
    - The .groupby() method returns a DataFrame.
    - We can extract a single column from the DataFrame:
    - **month_average_crs = data.groupby(['Month', 'Course Name'])['Rating'].count().unstack()**



What Day of the Week are People the Happiest?

- We're going to find out which day of the week has the average most positive ratings.
- We run:
    - **data['Weekday'] = data['Timestamp'].dt.strftime('%A')**
    - **weekday_average = data.groupby(['Weekday']).mean()**
    - **plt.plot(weekday_average.index, weekday_average['Rating'])**
- This gives us a chart, but the days are out of order (they're in alphabetical order it seems). To get them in the proper order, we need to input some code after *weekday_average* to see what's going on:
    - **weekday_average = weekday_average.sort_values('Weekday')**
    - So far, our code looks like this:

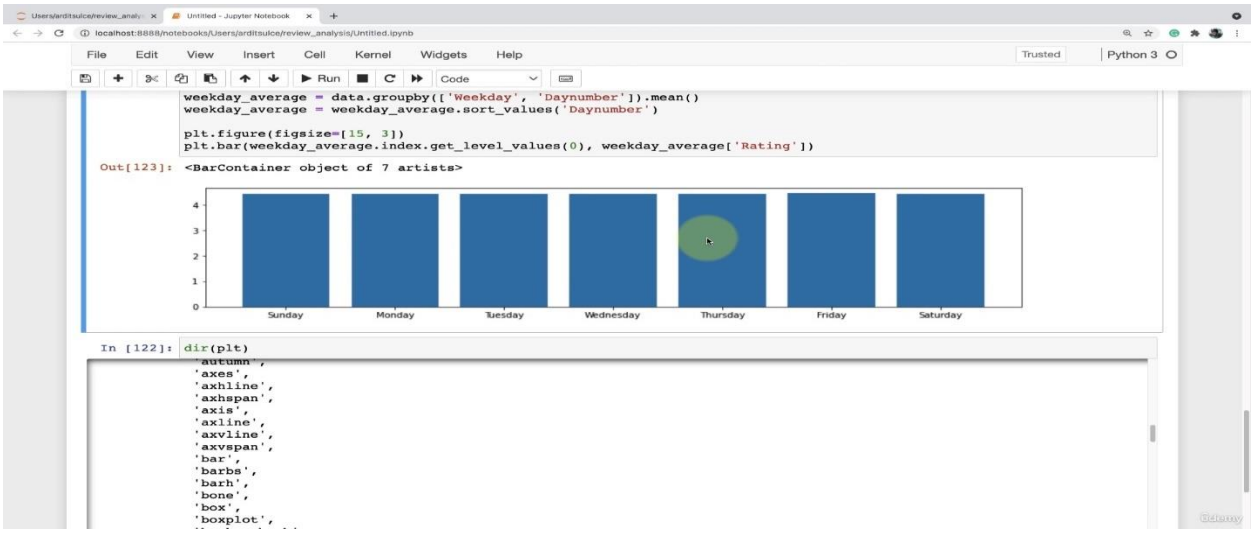
```
data['Weekday'] = data['Timestamp'].dt.strftime('%A')

weekday_average = data.groupby(['Weekday']).mean() # group
weekday_average = weekday_average.sort_values('Weekday') # order

plt.plot(weekday_average.index, weekday_average['Rating'])
plt.show()
```

    - **weekday_average**
        - outputs the alphabetical order. This is because when we run **weekday_average.index** on its own, it shows that it's indexing the days—as ***strings***—based on alphabetical order.
        - There are different ways to fix this.
    - We add: **data['Daynumber'] = data['Timestamp'].dt.strftime('%w')** after we add the 'Weekday' column up above.
    - We add another argument to our **.groupby()** method:
        - **weekday_average = data.groupby(['Weekday', 'Daynumber']).mean()**
    - We change our **.sort_values('Weekday')** to **.sort_values('Daynumber')**
    - And we tack on a **.get_level_values(0)** method to our **.index** method down in the plotting section:

```
data['Weekday'] = data['Timestamp'].dt.strftime('%A')
data['Daynumber'] = data['Timestamp'].dt.strftime('%w') ← ← ←

weekday_average = data.groupby(['Weekday', 'Daynumber']).mean() ← ← ←
weekday_average = weekday_average.sort_values('Daynumber') ← ← ←

plt.figure(figsize=[15, 3])          VVVVV
plt.plot(weekday_average.index.get_level_values(0), weekday_average['Rating'])
plt.show()
```

Other Types of Plots:

- So far, all the plots we've done in this section have been ***line plots***.
- So how do we go about using different types of graphs?
- All of these plots have been made with the **.plot()** method, but if we use **dir(plt)**, we can see other types of graphs, such as **.bar()**:
    - **plt.bar(weekday_average.index.get_level_values(0), weekday_average['Rating'])**



- Another choice is **.pie()**, but this option isn't well-suited to our data in this case and will cause an error. Running **help(plt.pie)** shows that **.pie()** expects a single argument **x**. So how can we make use of **.pie()** for our data?
- We created a new Markdown cell, **"### Number of ratings by course"**, then in the next cell we ran:
    - **share = data.groupby(['Course Name'])['Rating'].count()**
    - **print(share)**
    - We can use this total count of ratings per course now.
- In a new cell, we ran:
    - **plt.pie(share)**
    - Which gives us a pie chart. However, it doesn't have any labels, so:
    - **plt.pie(share, labels=share.index)**

```
### Number of ratings by course
share = data.groupby(['Course Name'])['Rating'].count()
# print(share) # check

plt.figure(figsize=[12, 5])
plt.pie(share, labels=share.index)
plt.show()
```

Section 21: App 3 (Part 2): Data Analysis and Visualization – in-Browser Interactive Plots:

Intro to the Interactive Visualization Section:

- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
-
-