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.

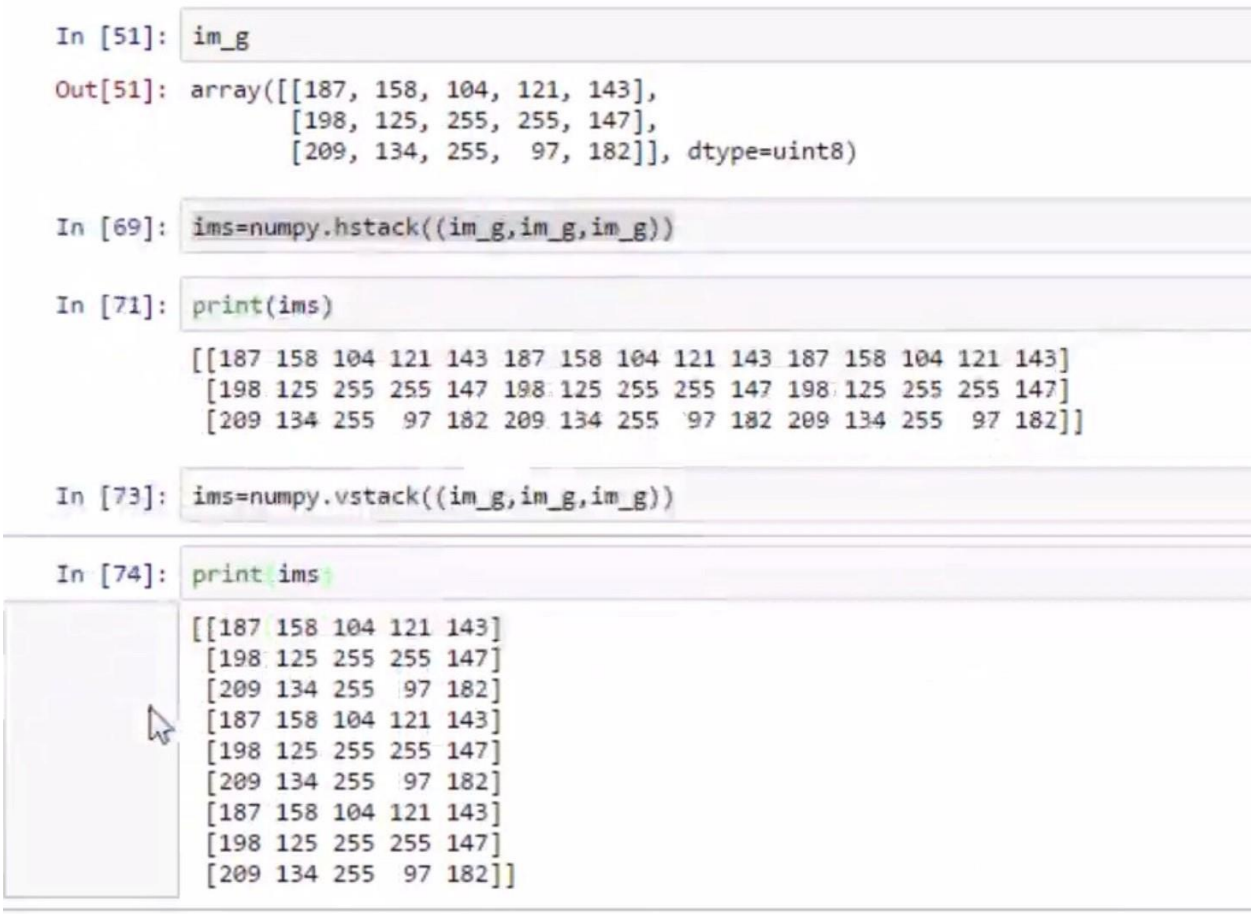
- If you want to iterate **value-by-value**:

```
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.



- 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:

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