

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. These notes and all app code and practice code organized in the following Github repository: <https://github.com/xroadtraveler/python-mega-course>

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:	13
Course Introduction:.....	13
Section 2 – Getting Started with Python:	13
Section Introduction:	13
Section 3 – The Basics: Data Types:	13
Python Interactive Shell:.....	13
Terminal:.....	13
Data Type Attributes:.....	14
How to Find Out What Code You Need:	14
What Makes a Programmer a Programmer:.....	14
How to Use Datatypes in the Real World:	14
Section 4 – The Basics: Operations with Data Types:	15
More Operations with Lists:.....	15
Accessing List Items:	15
Accessing List Slices:.....	15
Accessing Items and Slices with Negative Numbers:.....	15
Accessing Characters and Slices in Strings:.....	16
Accessing Items in Dictionaries:.....	16
Tip: Converting Between Datatypes:	17
Section 5: The Basics: Functions and Conditionals:	18
Creating Your Own Functions:	18
Intro to Conditionals:	18
If Conditional Example:	19
Conditional Explained Line by Line:	19
More on Conditionals:	19
Elif Conditionals:	19
Section 6: The Basics: Processing User Input:	20
User Input:	20
String Formatting:.....	20
String Formatting with Multiple Variables:.....	21
More String Formatting:	21
Cheatsheet: Processing User Input:.....	22

Section 7: The Basics: Loops:	23
For Loops: How and Why:.....	23
Dictionary Loop and String Formatting:.....	23
While Loops: How and Why:.....	23
While Loop Example with User Input:	24
While Loops with Break and Continue:.....	24
Cheatsheet: Loops:	24
Section 8: Putting the Pieces Together: Building a Program:	25
Section Introduction:	25
Problem Statement:.....	25
Approaching the Problem:.....	25
Building the Maker Function:.....	26
Constructing the Loop:.....	27
Making the Output User-Friendly:.....	28
Section 9: List Comprehensions:	29
Section Introduction:	29
Simple List Comprehension:	29
List Comprehension with If Conditional:.....	30
List Comprehension with If-Else Conditional:.....	31
Cheatsheet: List Comprehensions:	32
Section 10: More About Functions:	33
Functions with Multiple Arguments:	33
Default and Non-default Parameters and Keyword and Non-keyword Arguments:.....	33
Functions with an Arbitrary Number of <i>Non-keyword</i> Arguments:.....	34
Functions with an Arbitrary Number of <i>Keyword</i> Arguments:	35
Section 11: File Processing:	36
Section Introduction:	36
Processing Files with Python:.....	36
Reading Text from a File:	36
File Cursor:	37
Closing a File:	37
Opening Files Using “with”:	37
Different Filepaths:	38

Writing Text to a File:.....	38
Appending Text to an Existing File:.....	39
Cheatsheet: File Processing:	40
Section 12: Modules:.....	41
Section Introduction:	41
Built-in Modules:.....	42
Standard Python Modules:	43
Third-Party Modules:	44
Third-Party Module Example:.....	45
Cheatsheet: Imported Modules:.....	46
Section 13: Using Python with CSV, JSON, and Excel Files:.....	47
The “pandas” Data Analysis Library:.....	47
Installing pandas and IPython:.....	47
Getting Started with pandas:.....	48
Installing Jupyter:.....	49
Getting Started with Jupyter:.....	49
Loading CSV Files:	51
Exercise: Loading JSON Files:	51
Note on Loading Excel Files:	52
Loading Excel Files:	52
Loading Data from Plain Text Files:.....	52
Set Table Header Row:.....	53
Set Column Names:.....	53
Set Index Column:.....	53
Filtering Data from a pandas DataFrame:.....	54
Deleting Columns and Rows:	54
Updating and Adding New Columns and Rows:	55
Note:	56
Data Analysis Example: Converting Addresses to Coordinates:	57
Section 14: Numerical and Scientific Computing with Python and Numpy:.....	59
What is Numpy?.....	59
Installing OpenCV:.....	61
Convert Images to Numpy Arrays:.....	64

Indexing, Slicing, and Iterating Numpy Arrays:.....	65
Stacking and Splitting Numpy Arrays:.....	66
Section 15: App 1: Web Mapping with Python: Interactive Mapping of Population and Volcanoes:	68
Demo of the Web Map:	68
Creating an HTML Map with Python:.....	69
Adding a Marker to the Map:	70
Practicing “for-loops” by Adding Multiple Markers:.....	72
Practicing File Processing by Adding Markers from Files:.....	73
Practicing String Manipulation by Adding Text on the Map Popup Window:.....	74
Adding HTML on Popups:.....	75
Practicing Functions by Creating a Color Generation Function for Markers:	77
Tip on Adding and Stylizing Markers:.....	79
Solution: Add and Stylize Markers:	79
Exploring the Population JSON Data:.....	80
Practicing JSON Data by Adding a Population Map Layer from the Data:.....	80
Stylizing the Population Layer:.....	81
Adding a Layer Control Panel:.....	82
App 1: Full Code:	83
Section 16: Fixing Programming Errors:.....	85
Syntax Errors:	85
Runtime Errors:	85
How to Fix Difficult Errors:	86
How to Ask a Good Programming Question:	86
Making the Code Handle Errors by Itself:	86
Section 17: Image and Video Processing with Python:	87
Section Introduction:	87
Installing the Library:	87
Loading, Displaying, Resizing, and Creating Images:	88
Exercise: Batch Image Resizing:	90
Solution: Batch Image Resizing:	90
Solution Further Explained:	91
Detecting Faces in Images:.....	92
Capturing Video with Python:.....	95

Section 18: App 2: Controlling the Webcam and Detecting Objects:.....	98
Demo of the Webcam Motion Detector App:	98
Detecting Moving Objects from the Webcam:	98
Storing Object Detection Timestamps in a CSV File:	103
Section 19: Interactive Data Visualization with Python and Bokeh:.....	108
Introduction to Bokeh:.....	108
Installing Bokeh:.....	108
Your First Bokeh Plot:.....	109
Exercise: Plotting Triangles and Circles:.....	109
Using Bokeh with Pandas:.....	110
Exercise: Plotting Education Data:	111
Note on Loading Excel Files:	112
Changing Plot Properties:	112
Exercise: Plotting Weather Data:	113
Changing Visual Attributes:.....	114
Creating a Time-Series Plot:.....	115
More Visualization Examples with Bokeh:.....	116
Plotting Time Intervals from the Data Generated by the Webcam App:	117
Implementing a Hover Feature:.....	119
Section 20: App 3 (Part 1): Data Analysis and Visualization with Pandas and Matplotlib:.....	121
Preview of the End Results:	121
Installing the Required Libraries:	121
Exploring the Dataset with Python and pandas:.....	122
Selecting Data:	123
Filtering the Dataset:	124
Time-Based Filtering:	125
Turning Data into Information:.....	126
Aggregating and Plotting Average Ratings by Day:.....	128
Downsampling and Plotting Average Ratings by Week:	129
Downsampling and Plotting Average Ratings by Month:	130
Average Ratings by Course by Month:.....	130
What Day of the Week are People the Happiest?	132
Other Types of Plots:.....	133

Section 21: App 3 (Part 2): Data Analysis and Visualization – in-Browser Interactive Plots:.....	134
Intro to the Interactive Visualization Section:	134
Making a Simple Web App:.....	134
Making a Data Visualization Web App:.....	136
Changing Graph Labels in the Web App:	140
Adding a Time-Series Graph to the Web App:.....	141
Exercise: Monthly Time-Series:.....	142
Multiple Time-Series Plots:	144
Multiple Time-Series Streamgraph:	146
Exercise: Interactive Chart to Find the Happiest Day of the Week:	147
Adding a Pie Chart to the Web App:.....	148
Section 22: App 4: Web Development with Flask – Build a Personal Website:.....	149
Building Your First Website:.....	149
Preparing HTML Templates:	151
Adding a Website Navigation Menu:	152
Note on Browser Caching:	152
Improving the Website Frontend with CSS:.....	153
Creating a Python Virtual Environment:.....	153
How to Use the PythonAnywhere Service:.....	154
Deploying the Flask App on PythonAnywhere:.....	155
Section 23: Building Desktop Graphical User Interface (GUI) with Python:	156
Introduction to the Tkinter Library:	156
Creating a GUI Window and Adding Widgets:	156
Connecting GUI Widgets with Functions:	159
Exercise: Create a Multi-Widget GUI:	160
My Attempt:.....	161
Solution: Create a Multi-Widget GUI:.....	163
Section 24: Interacting with Databases:	164
How Python Interacts with Databases #sql:	164
Connecting to an SQLite Database with Python:	164
(SQLite) Selecting, Inserting, Deleting, and Updating SQL Records:.....	168
PostgreSQL Database with Python:	169
(PostGreSQL) Selecting, Inserting, Deleting, and Updating SQL Records:	170

Section 25: App 5: GUI Apps and SQL: Build a Book Inventory Desktop GUI Database App:	172
Demo of the Book Inventory App:	172
Designing the User Interface:	173
Coding the Frontend Interface:.....	174
Labels:	175
Entries:	175
Listbox:	176
Buttons:.....	176
Coding the Backend:	177
Connecting the Frontend with the Backend, Part 1:	179
Connecting the Frontend with the Backend, Part 2:	182
Full Frontend Code, Connected to Backend:	186
Exercise: Fixing a Bug in Our Program:	189
Solution: Fixing a Bug in Our Program:.....	191
Creating .exe and .app Executables from the Python Script:	192
Section 26: Object-Oriented Programming (OOP):.....	193
What is Object-Oriented Programming (OOP)?.....	193
Using OOP in a Program, Part 1:	194
Using OOP in a Program, Part 2:	195
Frontend with Object-Oriented Changes:.....	196
Backend with Object-Oriented Changes:.....	199
Creating a Bank Account Class:.....	200
Bank Account, Full Code So Far:.....	203
Creating Classes Through Inheritance:	204
OOP Glossary:	206
Section 27: App 6: Mobile App Development: Build a Feel-Good App:	208
Demo of the Mobile App:	208
Creating a User Login Page:	208
Creating a User Sign-Up Page:	212
Capturing User Input:.....	214
Full Python Code, so far:	215
Full Kivy Code, so far:	216
Processing User Sign Ups:	217

Creating a Sign Up Success Page:.....	219
Switching Between Pages:	221
Processing User Login Credentials:	222
Displaying Output to the User:	225
Full Python Program, so far:	227
Full Kivy Code, so far:	229
Stylizing the Login Page:.....	231
• Padding & Spacing:	231
• Button Size & Position:	232
• Stylizing Lower Buttons:.....	233
• Lower Button Colors & Opacity:	234
• Password Stars:	235
Stylizing the Sign Up Page:.....	236
Making the Buttons Interactive:	237
Adding Spacing:.....	237
Making Hoverable:.....	237
Switching to Logout Button Images:.....	238
Resizing Logout Button:	240
Repositioning Logout Button:	241
Making a Scrollable Area:	242
Section 28: Making an Android APK File from the Kivy App:	244
Note:	244
Preparing the Environment for Deploying the App to an Android Phone:.....	245
Creating an APK File for Android:	246
Troubleshooting Process:.....	247
Streamlined Process, from Start to Finish:	248
Installing the APK File for Android:	250
Deploying to iOS:.....	250
Section 29: Web Scraping with Python & Beautiful Soup:	251
How Web Scraping Works:	251
Link Update:	251
Web Scraping Example with Python:.....	252
Section 30: App 7: Web Scraping – Scraping Properties for Sale from the Web:.....	254

Demo of the Web Scraping App:	254
Note: The New Website URL:	254
Loading the Webpage in Python:.....	255
Extracting <div> Elements:.....	256
Scraping the Addresses of the Properties:.....	258
Scraping Special Elements:	261
Saving the Extracted Data in CSV Files:.....	263
Crawling Through Multiple Web Pages:	265
Section 31: App 8: Flask and PostgreSQL – Build a Data Collector Web App:.....	269
Demo of the Data Collector Web App:	269
Steps of Creating a PostgreSQL Database Web App with Flask:	270
Creating a Page with HTML:.....	271
Full index.html Code, So Far:	272
Stylizing the HTML Page with CSS:	274
Full CSS Code:.....	278
Capturing User Input:.....	280
Creating the PostgreSQL Database Model:.....	284
Our app.py Code, So Far:	288
Storing User Data to the Database:	289
Gmail App Password:	294
Emailing Database Values Back to User:.....	296
Our app.py Code, So Far:	298
Our send_email.py Code, So Far:.....	299
Emailing Data Statistics to Users:.....	300
Deploying the Database Web App Online:	304
Creating a Download-Upload Feature:	307
Section 32: App 9: Django & Bootstrap Blog and Translator App:.....	315
Demo of the App:.....	315
A Comparison of Python Web Frameworks:.....	317
Flask:	317
Django:	318
JustPy:	318
Setting Up a Virtual Environment:	319

Creating a Django Project:	321
Creating a Superuser for the Project:	324
Setting Up an Empty Django Blog App:.....	325
Creating a Database Model for the Blog App:	327
Overview of the Web App Architecture:	329
HTML Templates:	331
Django Views:.....	333
URL Patterns:	334
Creating Admin Interface Views:	336
Creating a Homepage:	338
Creating an “About” Page:.....	340
Listing Blog Posts on the Homepage:.....	341
Creating Links:.....	343
Adding Bootstrap to Django:	344
• Bootstrap Installation & Setup:.....	345
• Using Bootstrap to Modify Webpage:	347
• container:.....	347
• card:	348
• card-body & card-title:.....	348
• card-text:.....	350
• btn:	350
“index.html” code, so far:.....	353
Django Template Filters:.....	354
Template Inheritance:.....	355
Applying Bootstrap Styling to the Navigation Menu:	360
Demo of the Django Translation App:	362
The Steps of Django App Development:.....	363
Creating an Empty App Structure of the Translator:	364
Creating an HTML Form in Django:.....	365
Configuring the URLs:.....	367
Creating a Form:.....	368
Getting and Processing User Input Through a Form:.....	370
Completing the Translator App:.....	373

Section 33: App 10: Build a Geography Web App with Flask and Pandas:	376
Demo of the Geography Web App:.....	376
Solo Attempt, Notes:	377
Initial Setup:	377
Writing HTML and CSS:	377
Pandas Dataframes and Geocoding:.....	377
Solution, Part 1:	378
index.html:	378
main.css:.....	379
app.py:	380
download.html:.....	383
• Back to “app_ver2.py”:.....	383
Solution, Part 2:	384
app_ver4.py:	385
Section 34: Bonus Exercises:.....	386
Section 35: Bonus App: Building an English Thesaurus:.....	387
Demo of the Interactive English Dictionary App:.....	387
Know Your Dataset:	388
Loading JSON Data:	388
Returning the Definition of a Word:	389
Non-Existing Words:	390
Dealing with Case-Sensitive Words:	391
Calculating the Similarity Between Words:	392
Best Matches Out of a List of Words:	393
Finding the Most Similar Word from a Group of Words:	394
Getting Confirmation from the User:.....	395
Optimizing the Final Output:	397
Section 36: Bonus App: Building a Website Blocker:.....	399
Demo of the Website Blocker:.....	399

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 done 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.
- To 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:

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

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

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

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

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

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

```
1. with open("file.txt", "a+") as file:  
2.     content = file.write("Even more sample text")  
3.     file.seek(0)  
4.     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 Mac 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 **Jupyter 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[“Longitude”]=df[“Coordinates”].apply(lambda y: y.longitude 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 →
- 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:

```
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]]])
```

- 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.

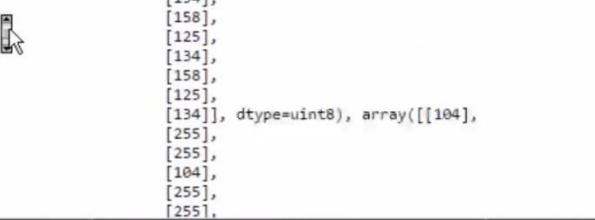
h-split:

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

[[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],
 [184],
 [255],
 [184],
 [255], dtype=uint8)]]
```



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:

```
1. import folium
2. import pandas
3.
4. data = pandas.read_csv("Volcanoes.txt")
5. lat = list(data["LAT"])
6. lon = list(data["LON"])
7. elev = list(data["ELEV"])
8.
9. html = """<h4>Volcano information:</h4>
10. Height: %s m
11. """
12.
13. map = folium.Map(location=[38.58, -99.09], zoom_start=5, tiles="Mapbox
    Bright")
14. fg = folium.FeatureGroup(name = "My Map")
15.
16. for lt, ln, el in zip(lat, lon, elev):
17.     iframe = folium.IFrame(html=html % str(el), width=200, height=100)
18.     fg.add_child(folium.Marker(location=[lt, ln], popup=folium.Popup(iframe),
        icon = folium.Icon(color = "green")))
19.
20.
21. map.add_child(fg)
22. 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:

```
1. import folium
2. import pandas
3.
4. data = pandas.read_csv("Volcanoes.txt")
5. lat = list(data["LAT"])
6. lon = list(data["LON"])
7. elev = list(data["ELEV"])
8. name = list(data["NAME"])
9.
10.html = """
11.Volcano name:<br>
12.<a href="https://www.google.com/search?q=%22%s%22"
    target="_blank">%s</a><br>
13.Height: %s m
14. """
15.
```

```
16. map = folium.Map(location=[38.58, -99.09], zoom_start=5, tiles="Mapbox
   Bright")
17. fg = folium.FeatureGroup(name = "My Map")
18.
19. for lt, ln, el, name in zip(lat, lon, elev, name):
20.     iframe = folium.IFrame(html=html % (name, name, el), width=200,
   height=100)
21.     fg.add_child(folium.Marker(location=[lt, ln], popup=folium.Popup(iframe),
   icon = folium.Icon(color = "green")))
22.
23. map.add_child(fg)
24. 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.: **I = lambda x: x**2**
 - **I(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>
%s<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:

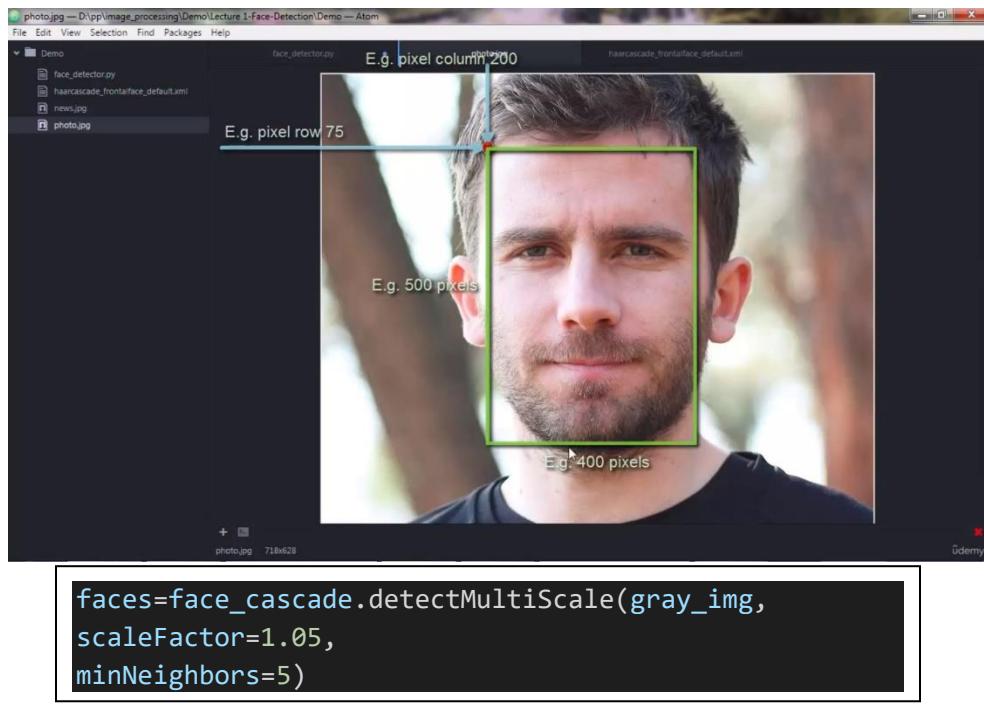
Note: Resources for this lecture include links to “Cv2 Documentation” and “Glob Documentation”.

- 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:

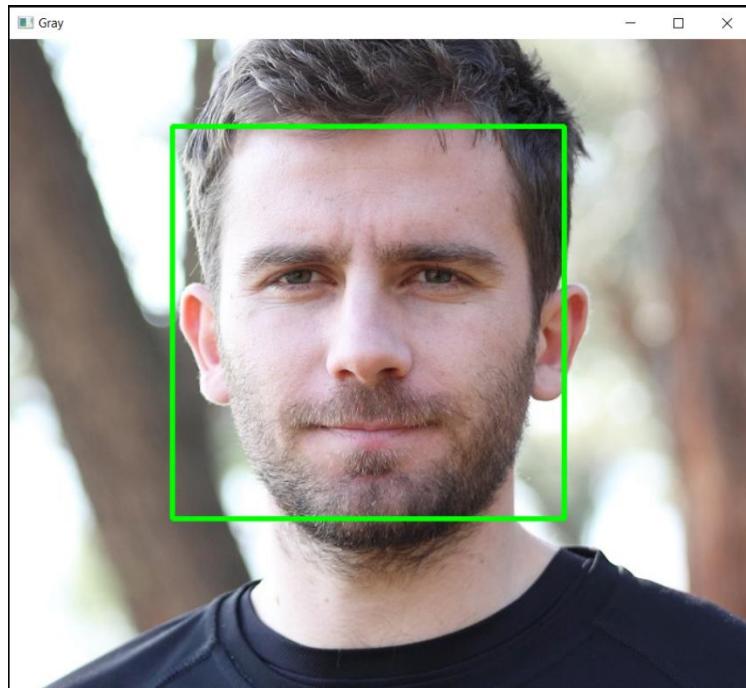
Note: Resources for this lecture include "Files.zip" and link to "Cv2 Documentation".

- 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:



- 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)

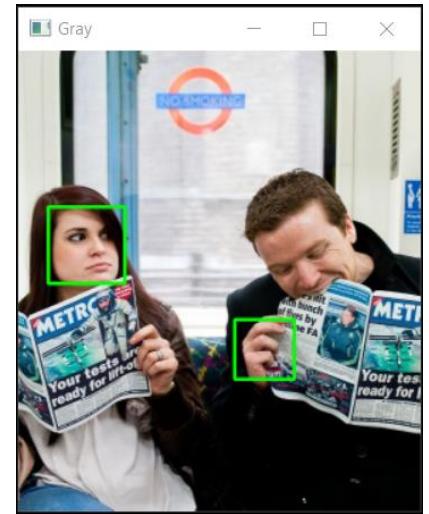
# Creates 'faces' numpy array
faces=face_cascade.detectMultiScale(gray_img,
scaleFactor=1.05,
minNeighbors=5)

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 use **glob**s 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 **verlegenhukens.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("verlegenhukens_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

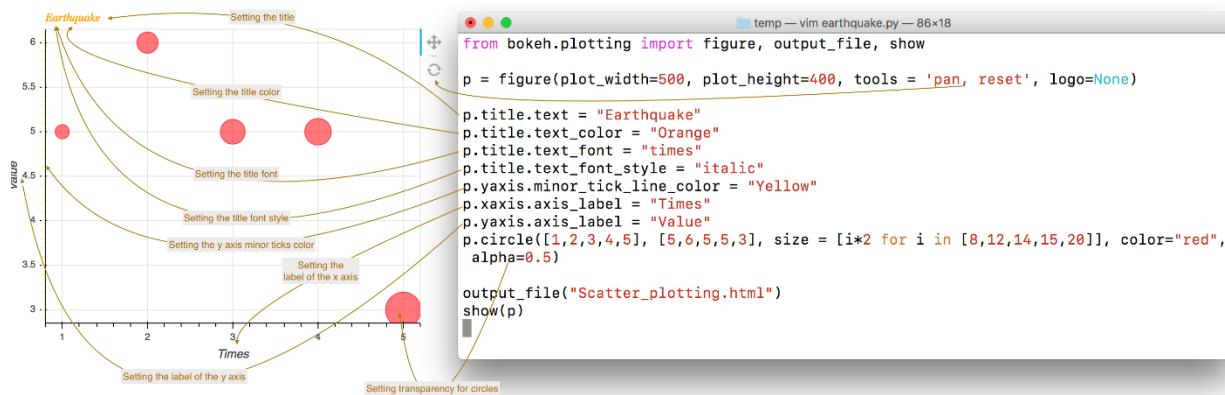
df=pandas.read_excel("verlegenhukens_resave.xlsx",sheet_name=0)
df["Temperature"] /= 10
df["Pressure"] /= 10

# 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 (a *streamgraph*).

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 at 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:

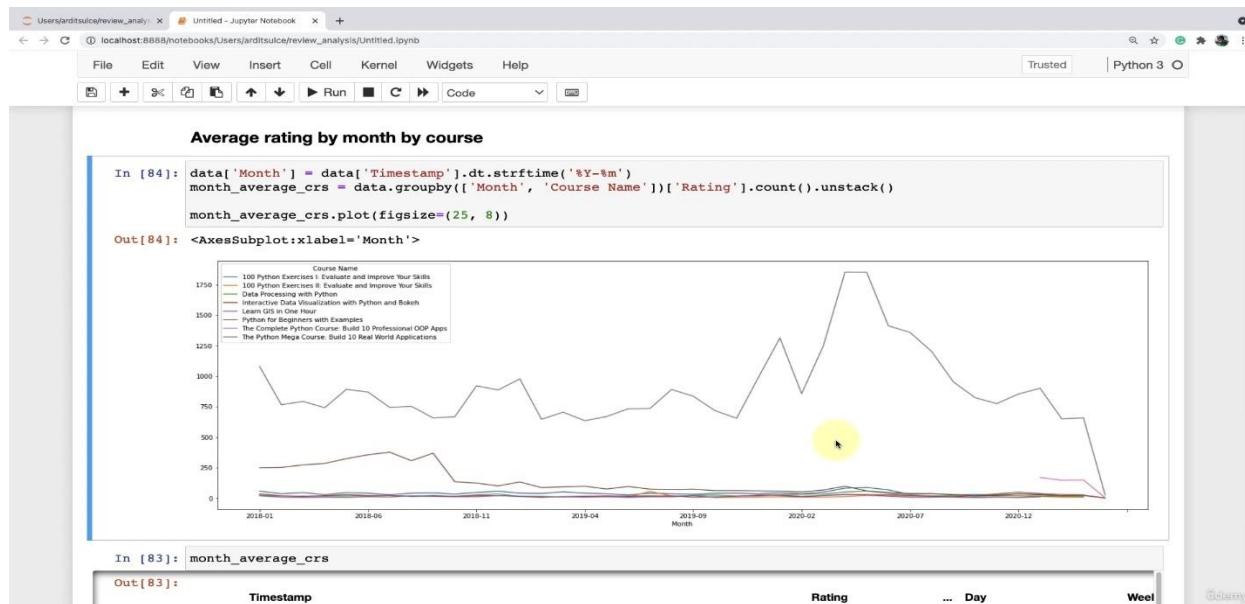
- In this lecture we're going to generate a plot with different lines, and each line is going to represent the **average rating per course**.
- Starting out, we can reuse our `data['Month'] =` setup from the last lecture.
- However, we need a new variable `month_average_crs` to group by both 'Month'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

Month	Course Name	Rating
2018-01	100 Python Exercises I: Evaluate and Improve Your Skills	4.353448
	Data Processing with Python	4.500000
	Interactive Data Visualization with Python and Bokeh	4.285714
	Learn GIS in One Hour	4.236842
	Python for Beginners with Examples	4.355422
...
2021-03	The Python Mega Course: Build 10 Real World Applications	4.632018
2021-04	100 Python Exercises I: Evaluate and Improve Your Skills	4.500000
	Interactive Data Visualization with Python and Bokeh	5.000000
	The Complete Python Course: Build 10 Professional OOP Apps	4.250000
	The Python Mega Course: Build 10 Real World Applications	4.576923

262 rows × 1 columns

- 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:

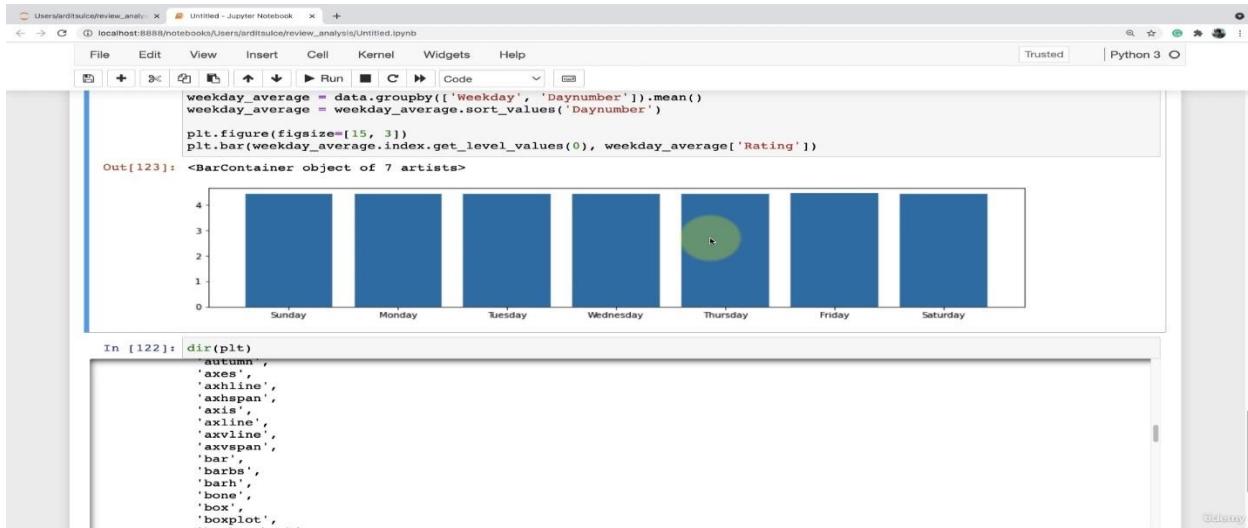
```
data['Weekday'] = data['Timestamp'].dt.strftime('%A')
data['Daynumber'] = data['Timestamp'].dt.strftime('%w') < < <

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

plt.figure(figsize=[15, 3])          WWW
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:

- Looks like we're going to be working with the same dataset as last section.
- He showed off interactive graphs of average ratings *by day*, *by week*, *by month*, *by month by course*, a cool looking *streamgraph* of “Analysis of Course Reviews”, “When are People the Happiest?”, and an interactive pie chart called “Number of Ratings by Course”.
- We're mainly going to be using **Highcharts** (a Python library) through **JustPy** in this section.
 - **JustPy** is a framework for using Python to generate modern-looking websites without writing any HTML, JavaScript, or CSS.

Making a Simple Web App:

- He noted towards the beginning of the video that Jupyter Notebook won't work here, so one should use an IDE such as VSCode. I prefer that over Jupyter anyway, personally.
- We start by installing **JustPy** (pip3.10 install justpy), then we import it into our new **0-simple-app.py** program that we're now creating.
- We then create a function **app()** and set a variable **wp** (for “webpage”, but we can name it whatever). We add **JavaScript functionality** with **Quasar**, which is a JS library.

```
import justpy as jp ← ← ←

def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    return wp
```

- Now if we run this code on its own, nothing will happen yet because no instance is calling this **app()** function that we've created. Do call the function, we just add:

```
import justpy as jp

def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    return wp

jp.justpy(app) ← ← ←
```

- Running this code will return a message such as:
 - “**JustPy ready to go on http://127.0.0.1:8000**”
 - which is the IP address you can copy/paste to your browser or **CTRL + Click** to see your webpage.
- At this stage, our text is just plaintext because we haven’t added any *style* to it yet. To add style, we can add extra arguments to our **QDiv()** callouts. We can look up styles by doing a google search for “**quasar style**”.
 - We added the argument **classes=”text-h1”** to our **h1** variable in our app function.
 - He noted that just re-running the Python code while it’s already running won’t do anything, you need to type **CTRL + C** to stop the old code first.
- Here’s our code at the end:

```
import justpy as jp

def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews",
    classes="text-h3 text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    return wp

jp.justpy(app)
```

Making a Data Visualization Web App:

Buckle up, this is a long one.

- We started by creating a new file, **1-av-rate-day.py** and we copied all our code from the previous file; we're going to add **Highcharts** functionality to it.
- To decide what kind of chart we need for the data, we can go read the **Highcharts documentation** ("Highcharts docs" in google). Like Quasar, Highcharts is a JavaScript library. Python is getting both these libraries/frameworks together.
- On the Highcharts documentation, we clicked on "**Chart and series types**" on the left, then scrolled down to "Spline chart" as an example. He showed that there are icons on the top-right corner of the chart for "**jsFiddle**" and "**CodePen**", which are two online code editors we can access the chart/code from.
- Once inside the code in **jsFiddle**, he showed that we need to copy all of the code after the comma after '**container**', :

The screenshot shows the jsFiddle interface with the following details:

- Fiddle meta:** Spline with inverted axes, author(s): Torstein Hals, Private fiddle.
- Resources:** URL: cdnjs, Async requests, Other (links, license).
- HTML:** Contains the Highcharts JS code for a spline chart with inverted axes, titled "Atmosphere Temperature by Altitude".
- CSS:** Contains styles for Highcharts figures and data tables.
- Output:** Shows the resulting Highcharts chart titled "Atmosphere Temperature by Altitude". The chart has an inverted Y-axis (Altitude) ranging from 0 km to 75 km and an X-axis (Temperature) ranging from -90° to -10°. It displays two data series: a blue line with circular markers and a red line with diamond markers, showing temperature decreasing with increasing altitude.

- And we copy up to the final curly bracket () at the bottom of the code. We then go to Python, and we create a string variable:
 - **chart_def = " " ",** and then we paste our copied JS code inside here.
- The full code for this variable will look like this:

```
chart_def = """
{
    chart: {
        type: 'spline',
        inverted: true
```

```
},
title: {
    text: 'Atmosphere Temperature by Altitude'
},
subtitle: {
    text: 'According to the Standard Atmosphere Model'
},
xAxis: {
    reversed: false,
    title: {
        enabled: true,
        text: 'Altitude'
    },
    labels: {
        format: '{value} km'
    },
    accessibility: {
        rangeDescription: 'Range: 0 to 80 km.'
    },
    maxPadding: 0.05,
    showLastLabel: true
},
yAxis: {
    title: {
        text: 'Temperature'
    },
    labels: {
        format: '{value}°'
    },
    accessibility: {
        rangeDescription: 'Range: -90°C to 20°C.'
    },
    lineWidth: 2
},
legend: {
    enabled: false
},
tooltip: {
    headerFormat: '{series.name}  
',
    pointFormat: '{point.x} km: {point.y}°C'
},
plotOptions: {
    spline: {
        marker: {
            enable: false
```

```

        }
    },
    series: [{
        name: 'Temperature',
        data: [[0, 15], [10, -50], [20, -56.5], [30, -46.5], [40, -22.1],
               [50, -2.5], [60, -27.7], [70, -55.7], [80, -76.5]]
    }]
}
"""

```

- We now go down and add a new **hc** (for Highcharts) variable in our **app()** function:

```

def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews", classes="text-h3
text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    hc = jp.HighCharts(a=wp, options=chart_def) ← ← ←
    return wp

jp.justpy(app)

```

- Now when we run the code and follow the output IP address, the chart we copy/pasted into **chart_def** shows up on our webpage. JustPy is converting the JS code (JSON) into a **Python dictionary**, but a special type of dictionary ('**addict.addict.Dict**' type). This also allows us to access keys/values using dot-notation now:
 - **title.text** for example gives us the data from the "text" key.
 - **hc.options.title.text = "Average Rating by Day"** will change the title of our graph.
- We can change any data we like in the template graph, which is important because we want to show our data in a graph. In particular, the graph has a **series** key, which is where our data points will go. This **series key** has two sub-keys: **name: 'Temperature'** and **data: [[data]]**.
 - Note that **series**' value is a **list** of one item, which is a dictionary containing **name** and **data**, so we have to treat it like a list. **Data** itself is a list of lists.
 - We're going to change this data in our **app()** function:

```

hc.options.title.text = "Average Rating by Day"
hc.options.series[0].data = [[3, 4], [6, 7], [8, 9]] ← ← ←
return wp

```

- Note also that the template graph we brought in was set as an inverted graph, and we can change this by going up to the top and setting "inverted" to "false":

```

chart_def = """
{
    chart: {
        type: 'spline',
        inverted: false ← ← ←

```

- We can also change our data directly within the JSON code, but later we're going to learn to "inject" DataFrames into this data's place anyway.
- Now, we're not usually going to type a list of lists as our new data like we have so far. More likely, we're going to separate x and y and then **list**/**zip** them up before passing them:

```

hc.options.title.text = "Average Rating by Day"
x = [3, 6, 8] ← ← ←
y = [4, 7, 9] ← ← ←
hc.options.series[0].data = list(zip(x, y)) ← ← ←

```

- Now that brings us to our DataFrame, which we can pull over from our Jupyter Notebook version (**plotting.ipynb** or **plotting.py**) with copy/paste if we wish:

```

import justpy as jp
import pandas ← ← ←
from datetime import datetime ← ← ←
from pytz import utc ← ← ←

data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp']) ← ← ←

```

- We can also bring over our "Average/count rating by day", which is a parsed/aggregated version of our DataFrame:
- We can then go back down to **app()** and replace 'x' and 'y' with "**day_average.index**" and

```

data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])
data['Day'] = data['Timestamp'].dt.date ← ← ←
day_average = data.groupby(['Day']).mean() ← ← ←

```

`'day_average['Rating']':`

```

hc.options.title.text = "Average Rating by Day"
hc.options.series[0].data = list(zip(day_average.index, ← ← ←
day_average['Rating'])) ← ← ←

```

- However, running this as-is produces a blank graph and no error message. This is because Highcharts considers dates in the format ‘**2018-01-01**’ to be *categories* and not *numbers*. To fix this, we need to add a “**categories**” sub-key to the **xAxis** key:

```
hc.options.xAxis.categories = list(day_average.index) ← ← ←  
hc.options.series[0].data = list(day_average['Rating'])
```

- Note that we just have to pass a list of **day_average['Rating']** now.
- At this point we still have some label tooltips and some axis titles from the template chart, but we can change them. In this case, he suggests just changing everything directly in the JavaScript code.

Changing Graph Labels in the Web App:

- He started off by talking about changing the template chart’s default labels, and mentions that you can change them with dot-notation, but that it’s better in this case to just change them in the JavaScript code. We/I already did some of this at the end of the last lecture. Nothing new, really.

Adding a Time-Series Graph to the Web App:

- We started off by creating a new file **2-av-rate-week.py** and copy/pasting our base code from **0-simple-app.py**. Then we created a new Highcharts object **hc**:
 - **hc = jp.HighCharts(a=wp, options=chart_def)**
- To create our new **chart_def**, we can use an example from HighCharts docs again. However, in this case it makes sense to just reuse the spline chart. We can copy it over from jsFiddle again, or copy it from our previous code.
- We also want to, again, copy the various **import** lines and the **data =** line from the code we used in Jupyter Notebook.

```
import justpy as jp
import pandas < < <
from datetime import datetime < < <
from pytz import utc < < <

data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp']) < < <
```

- We also need to parse our data by 'Week' this time:

```
data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])
data['Week'] = data['Timestamp'].dt.strftime('%Y-%U')
week_average = data.groupby(['Week']).mean()
```

- Then we have to add to our **app()** function down below, to overwrite the template JS code:

```
hc = jp.HighCharts(a=wp, options=chart_def)
hc.options.title.text = "Average Rating by Day"

hc.options.xAxis.categories = list(week_average.index)
hc.options.series[0].data = list(week_average['Rating'])
```

Full **app()** code looks like this:

```
def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews", classes="text-h3 text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    hc = jp.HighCharts(a=wp, options=chart_def)
    hc.options.title.text = "Average Rating by Day"

    hc.options.xAxis.categories = list(week_average.index)
    hc.options.series[0].data = list(week_average['Rating'])

    return wp
```

Exercise: Monthly Time-Series:

- Time to do the same thing we just did, but for months. Pretty much just swapped some things out. Here are some comparisons of parts of all three:

DataFrame Setup, Daily:

```
data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])
data['Day'] = data['Timestamp'].dt.date
day_average = data.groupby(['Day']).mean()
```

DataFrame Setup, Weekly:

```
data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])
data['Week'] = data['Timestamp'].dt.strftime('%Y-%U')
week_average = data.groupby(['Week']).mean()
```

DataFrame Setup, Monthly:

```
data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])
data['Month'] = data['Timestamp'].dt.strftime('%Y-%m')
month_average = data.groupby(['Month']).mean()
```

app(), Daily:

```
def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews", classes="text-h3 text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    hc = jp.HighCharts(a=wp, options=chart_def)
    hc.options.title.text = "Average Rating by Day"

    hc.options.xAxis.categories = list(day_average.index)
    hc.options.series[0].data = list(day_average['Rating'])

    return wp
```

app(), Weekly:

```
def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews", classes="text-h3 text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    hc = jp.HighCharts(a=wp, options=chart_def)
    hc.options.title.text = "Average Rating by Week"

    hc.options.xAxis.categories = list(week_average.index)
    hc.options.series[0].data = list(week_average['Rating'])

    return wp
```

app(), Monthly:

```
def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews", classes="text-h3 text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")
    hc = jp.HighCharts(a=wp, options=chart_def)
    hc.options.title.text = "Average Rating by Month"

    hc.options.xAxis.categories = list(month_average.index)
    hc.options.series[0].data = list(month_average['Rating'])

    return wp
```

Multiple Time-Series Plots:

- In this lecture we're going to recreate our graph for **Average rating by month by course**, similar to the version we created in Jupyter Notebook.
- We're going to start off by going to **HighCharts docs** again to choose a chart type. For this one, we'll be using one called **Areaspline chart**, which can plot multiple splines.
- Once he copied in some of our template code and then the JS code from HighCharts, he mentioned that it's a good idea to try testing out the unmodified chart first.
 - When we test out the unmodified JS code, we get an **error** in both Python terminal and in the webpage. I noticed this myself when I tried getting ahead of the video, so it's good to know it's not just me.
 - The error happens because of this line in **backgroundColor**:
 - `Highcharts.defaultOptions.legend.backgroundColor || '#FFFFFF'`
 - Python doesn't recognize this because it's JavaScript, so we want to delete everything in the line except for: **'#FFFFFF'**. Removing this line allows the code/graph to work.
- We can also get rid of the **area shading** that comes in with this type of chart by default by *changing* type: '**areaspline**' to '**spline**' up top.
-
- Once again, we then copy our **import** and **DataFrame** code from our Jupyter Notebook version and paste that up top. We then copy our **aggregation** code just below that:

```
import justpy as jp
from calendar import month
import pandas
from datetime import datetime
from pytz import utc

data = pandas.read_csv("reviews.csv", parse_dates=[ 'Timestamp' ])
data['Month'] = data['Timestamp'].dt.strftime('%Y-%m')
month_average_crs = data.groupby(['Month', 'Course Name']).mean().unstack()
```

- Now down in the **app()** function, we need to create the list found in **series**: dynamically, that will replace both **name** and **data** within it.
 - Ref: **series: [{ name: 'John', data: [] }, { name: 'Jane', data: [] }]** is more or less the default series entry on his screen. HighCharts has updated mine to be about 'moose' and 'deer'.
 - We need to use **list comprehension** to create a new dynamic variable **hc_data**:

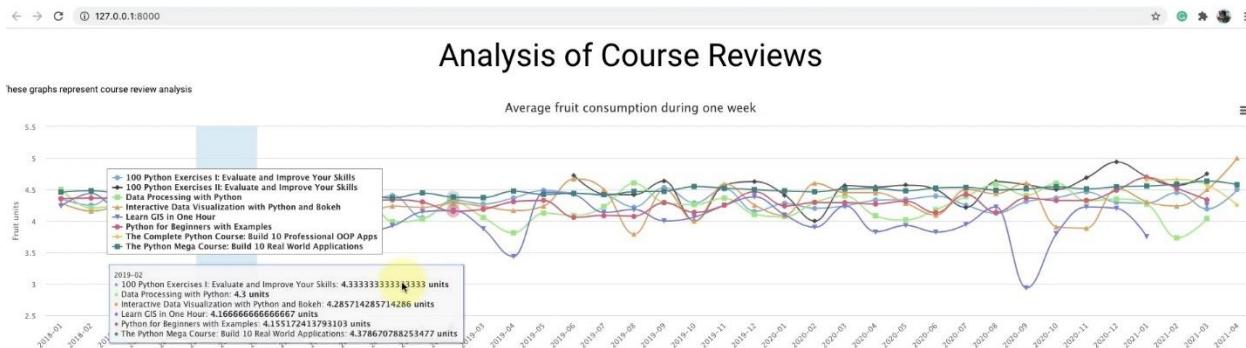
```
def app():
    wp = jp.QuasarPage() # wp stands for webpage
    h1 = jp.QDiv(a=wp, text="Analysis of Course Reviews",
                 classes="text-h3 text-center q-pa-md")
    p1 = jp.QDiv(a=wp, text="These graphs represent course review analysis")

    hc = jp.HighCharts(a=wp, options=chart_def)
    hc.options.xAxis.categories = list(month_average_crs.index)

    # creates list comprehension of Course Names, average ratings column
    hc_data = [{"name": v1, "data": [v2 for v2 in month_average_crs[v1]]} for
    v1 in month_average_crs.columns] < < <

    hc.options.series = hc_data < < <

    return wp
jp.justpy(app)
```



- All the course charts are shown together, but as you can see, the legend is floating over the charts to the left. We can change that by looking at our JS code for "**legend:**" and then setting "**floating:**" to '**false**'.
- Ended up playing around with some JS options that were causing issues for me but weren't present for his version.
- In the next lecture we're going to decide on the best type of chart to show the **rating count per month per course**, since this graph doesn't present it well. We'll use a **Stream Graph**.

Multiple Time-Series Streamgraph:

- We're going to start by duplicating our **4-av-rate-crs-month.py** code to create **5-av-rate-month-stream.py**, because we can reuse almost all of our code from the previous lecture. All we need to change is the [**JS code for the graph**](#).
- Running the code with the template graph (a stream chart about Olympic medal wins) as-is gives us an error due to the JSON. We need to delete all mention of “**colors**” variables from the JS code.
- Running it after deleting “colors” gives us another error, “**Ampezzo**”, which appears to be in the list of cities the Olympics were held in. This JS syntax is found in the [xAxis categories](#), and since we’re going to replace this when we inject our data anyway, we can just delete the whole [categories](#) section that it’s in. We end up leaving these categories as an [empty list](#).
- Playing around fixes things, and we would need to change some of the labeling and tooltips as usual, but I didn’t feel like doing this at this point.

Exercise: Interactive Chart to Find the Happiest Day of the Week:

- Starting off, I'm assuming this will be more copy/paste shenanigans from when we did this in our Jupyter Notebook graphs in the last section, adding that to our more recent code.
- Changing the code up top was easy:

```
data = pandas.read_csv("reviews.csv", parse_dates=['Timestamp'])
data['Weekday'] = data['Timestamp'].dt.strftime('%A')
data['Daynumber'] = data['Timestamp'].dt.strftime('%w')

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

- Translating data from using `plt.plot()` format to **HighCharts** format was a bit difficult, especially when I had to split what was formerly the X- and Y- data in a single line into two separate lines:
 - Here's the former `plt` version from **Jupyter plotting.py**:

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

- And here's the new version:

```
hc = jp.HighCharts(a=wp, options=chart_def)
hc.options.xAxis.categories =
list(weekday_average.index.get_level_values(0)) ← ← ←
hc.options.series[0].data = list(weekday_average['Rating']) ← ← ←
```

- Then I had to go through and delete some stuff from the JS code because the graph wasn't coming in properly. The range was still based on the Deer/Moose data even though our new data should've been injected and overwritten it.

Adding a Pie Chart to the Web App:

- We're going to start by going to HighCharts documentation and pulling a template for a pie chart from there.
- We're going to be recreating our Jupyter Notebook analysis from “**Number of Ratings by Course**” and applying this pie chart to it.
- We start by creating a new file, **7-rate-crs-pie.py** and copy/pasting our basic template.
- At the top, we imported some libraries as usual, brought in our data, and copy/pasted the pie chart template:

```
data = pandas.read_csv("reviews.csv", parse_dates=[ 'Timestamp' ])
share = data.groupby(['Course Name'])['Rating'].count()
```

- ~~He noted that within the **series** section of our pie chart template, we need to be careful because within series, the **data** section is a **list** of dictionaries for our data.~~
- Instead of ~~hc.options.series[0].data~~, this time we use ~~hc.options.series.data~~ to set our options in the graph (note, see note additions below):

```
hc.options.series.data =
```

- Since we're working with a list, we'll need to do a **list comprehension** on our “**share**” variable, which contains the names of courses as well as their ratings:

```
hc = jp.HighCharts(a=wp, options=chart_def)
hc_data = [{"name": v1, "y": v2} for v1, v2 in zip(share.index, share)]
hc.options.series.data = hc_data
```

- Just running this gives us an error, **AttributeError: 'list' object has no attribute 'data'**. Turns out we did need that **hc.options.series[0].data** after all.
- Adding that back in fixes the program, though I (and several other students in the Q&A section) had to delete a few JS lines in order to get the pie chart to actually show up on the page. The lines to delete were:
 - **plotBackgroundColor: null,**
 - **plotBorderWidth: null,**
 - **plotShadow: false,**
- After deleting these, my pie chart showed up perfectly.

Section 22: App 4: Web Development with Flask – Build a Personal Website:

Building Your First Website:

Note: Resource for this lecture is Flask Documentation.

- Start with **pip install flask** to install it.
- Then we created a **script1.py** file to write our website with.
- Inside the script we wrote:

```
from flask import Flask

app=Flask(__name__)

@app.route('/')
def home():
    return "Website content goes here!"

if __name__ == "__main__":
    app.run(debug=True)
```

- We then ran it, then went to our browser and typed in **localhost:5000** to see it. Our message shows at the top.
- To understand our 7 lines of code:
 - We **import** the **Flask** class **object** from the overall **flask library**.
 - We **instantiate** the Flask object as a variable called **app** and give it a special variable that will give it the **name** (**__name__**) of the Python script.
 - When we execute the Python script, it assigns the name “**__main__**” to the script.
 - Case 1 – Script executed: **__name__ = “__main__”**. This only happens if we manually execute the script.
 - Case 2 – Script imported: **__name__ = “script1”**. If imported, the conditional is not done.
 - He then changed **@app.route(‘/’)** to **@app.route(‘/about/’)** and reran the script. Now when we go to **localhost:5000** we get a 404 error, but if we go to **localhost:5000/about/** we get a page. If we still want a home page, we would add another decorator with **(‘/’)** in it.
 - Finished code after these changes looks like this:

```
from flask import Flask

app=Flask(__name__)

@app.route('/')
```

```
def home():
    return "Homepage goes here!"

@app.route('/about/') < < <
def about(): < < <
    return "'About' information goes here!" < < <

if __name__ == "__main__":
    app.run(debug=True)
```

Preparing HTML Templates:

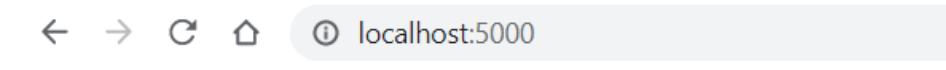
- From flask we also import **render_template** method of the Flask library.
- We then change our @app.route('/') contents to this:

```
@app.route('/')
def home():
    return render_template("home.html")
```

- We need to create a **home.html** file for this to point to now, and we need to store any and all html files for our program in a folder called **templates**, within the same folder as script1.py.
- Inside our newly created **home.html** file we wrote:

```
<!DOCTYPE html>
<html>
    <body>
        <h1>My homepage</h1>
        <p>This is a test website</p>
    </body>
</html>
```

- We then reloaded our homepage and got this:



A screenshot of a web browser window. The address bar shows 'localhost:5000'. The page content is displayed below the address bar.

My homepage

This is a test website

- We also duplicated our **home.html** to create a new **about.html** file for our 'About' page.
 - Don't forget to point `@app.route('/about/')` to this HTML file like we did with the `@app.route('.')` one.
- Checking **localhost:5000/about/** shows these changes.

Adding a Website Navigation Menu:

- For this one he just copy/pasted some code that he'd already written for **layout.html** in the templates folder. As such, I downloaded the existing version from the resources dropdown and pasted it over to my own templates folder. I opened it and replaced his name with mine.
- Now in our other HTML files, since they will be child files, we can delete the DOCTYPE declaration, the **<html>** tag, and we can replace the **<body>** tag with **<div class="home">**. We also use the templating syntax to tag the file up top with **{%extends "layout.html"%}** and **{%block content%}**, and **{%endblock%}** down at the bottom:

```
{%extends "layout.html"%}  
{%block content%}  
<div class="home">  
    <h1>My homepage</h1>  
    <p>This is a test website</p>  
</div>  
{%endblock%}
```

- We also give our **about.html** file a similar treatment.
- Our webpage now has a navigation menu.

Note on Browser Caching:

In the next lecture, we will add CSS styling to the webpage. Sometimes, when you make a change to the CSS file and reload the webpage, the changes are not shown because the browser uses the previous cached styling. If this happens, open the browser in private (incognito) mode and load the webpage there.

Improving the Website Frontend with CSS:

- We want to add CSS styling to make our website look nicer. To do this, we need to do two things: create a CSS file, and link to it from our layout.html file.
- Since he just copy/pasted an existing CSS file (**main.css**), I'm just going to use the one in the resources again. We put this in a folder called css inside of a folder called static.
- In our layout.html, we need to add another <head> above the other one:

```
<head>
    <title>Flask App</title>
    <link rel="stylesheet" href="{{url_for('static',
filename='css/main.css')}}">
</head>
```

- Now when we reload our website, it looks much, much nicer, based on the settings included in **main.css**.
- And now we're ready to deploy this website online, on the cloud.

Creating a Python Virtual Environment:

Note: Resources for this lecture include Flask Documentation and Heroku Documentation. Update: Heroku is no longer used in the course and has been replaced by PythonAnywhere.

- So far, this website is only visible to localhost. To deploy it, we need to run it on a webserver or on the cloud, so we're going to use a service called **Heroku**.
- Before deploying to Heroku, we need to do something that we should've done at the beginning of this section and set up a virtual environment for Python. It's good practice to use a "clean install" of Python rather than running from the main installation. This is because we don't need all those extra libraries from the main install if we're running a fast application.
- We install the virtual environment with **pip install virtualenv**. We also need to create a new folder *at the same level* as the folder our website code is contained in. In this case, I'm naming it **flask_app**. Note that he was using Atom, which works a little different from VSCode, so basically you want VSCode's command line running from the same level of the folder your website files are stored in.
- We then run the command **python -m venv virtual**, which creates our virtual environment and a folder named "**virtual**". This can take a few seconds to go through. If you go inside the "virtual" folder and into "scripts", you can see that "python.exe" is in there.
- Now in the terminal we can run **virtual/Scripts/python** ("\\virtual\\Scripts\\python" if using Windows command line), which will trigger our virtual Python environment's interactive shell. However, we're not interested in using the shell, so we run CTRL+Z to get out.
- We want to install Flask on our virtual environment Python, so we need to use the **pip** library from our virtual Python. To do this we run **virtual/Scripts/pip install flask**.
- With that installed, we want to run our web app from our virtual Python environment. To do that, we run **virtual/Scripts/python flask_app/script1.py**. Going to the website and refreshing double-checks that it's working correctly.

How to Use the PythonAnywhere Service:

Note: Resource for this lecture includes PythonAnywhere Documentation. Previously in the course, he taught students how to use Heroku, which is less user-friendly.

- Signed up for the free version of PythonAnywhere, confirmed email, etc.
- **Dashboard:** The starting place once we've signed in. Pretty self-explanatory.
- **Consoles:** He gave a brief tour starting with the **Consoles** tab, which we clicked and chose to open up a **bash** console. We can use the bash console to access our own server where we'll deploy our web app. We can also open up a Python interactive shell with **python 3**.
 - In bash, we can also use **pwd** to see our current working directory (duh), and we can use **nano** as a text editor (i.e. **nano example.py** to create and edit a Python file).
- **Files:** The next tab in the tour is **Files**, where the files for our current directory are created/stored.
 - Creating a new file here is easy, such as typing **example2.py** and clicking "New File". This opens up a more modern-looking text editor. There are even two buttons down below: "**>>> Run this file**" and "**\$ Bash console here**".
- **Web:** The **Web** tab is where we will create/store our web app in the next lecture.
- **Tasks:** In the **Tasks** tab, we can enter in the PATH of one of our PythonAnywhere Python files and set a time to run the program every day.
- **Databases:** The **Databases** menu allows us to create a **MySQL** or **Postgres** database.

Deploying the Flask App on PythonAnywhere:

- **Web tab:** From Dashboard we're going to switch over to the **Web tab**, then click on “Add a new web app”. PythonAnywhere lets us know what the domain name for the website will be (custom domain names are only allowed in paid versions). You also need to see a *domain provider* in order to *buy a domain name*. He recommends **namecheap.com**. We click “Next” at the bottom of this message.
- The next page asks us to “Select a Python Web framework”, so we select “Flask” and hit “Next”.
- Next we select the version of Python we want (he chose 3.9, I chose **3.10**) and hit “Next”.
- The next page tells us the **name of the app** and its **Path**. It seems to come in as “flask_app.py”, which is coincidentally what I named my local folder containing my web app. (Our **script1.py** file is going to become this.) Click “Next” to create our app.
- **Files tab:** After the app is created, we want to go to the **Files tab**. On the left we can see “mysite/”, which we want to click on. Inside that is **flask_app.py**. We click on this to open it, and we delete the default code that was put inside and replace it with code from our **script1.py**.
- Just like in our local directory, we want **static** and **templates** in **mysite/** as well. Type them each into the “New directory” field to create those. Don’t forget to create a **css** folder inside of the static folder.
- Upload the **template** files **layout.html**, **home.html**, and **about.html**, and the **main.css** file into **static**.
- Once everything is uploaded, we open **flask_app.py** and—since our app is now in “production”—at the bottom we change from `app.run(debug=True)` to `app.run(debug=False)`. We do this because leaving it in debug mode can make our app vulnerable to hackers.
- **Web tab:** Once we've made this final change, we're ready to see our app run. However, first we need to go to the **Web tab** and click on the “**Reload**” button.
- Then we click on the URL above that in order to see our website.
 - It worked fine for me (and him), but he noted that if it doesn't work, you scroll down in the Web tab and look for the **Error log**. Clicking on that will show you all the errors running on the Python app.

Section 23: Building Desktop Graphical User Interface (GUI) with Python:

Introduction to the Tkinter Library:

Note: Resource for this lecture is Tkinter Documentation.

- He showed off a GUI called “Bookstore” that he built with **Tkinter**, which communicates with a backend database.

Creating a GUI Window and Adding Widgets:

- Created a new Python file called **script1.py**.
- You don’t have to pip install tkinter because it’s a built-in library, so all we have to do is import it. However, it’s good practice to import *everything* from tkinter with:
 - **from tkinter import ***
 - This is because we’re going to use a lot of objects from the tkinter library, so it makes sense to just *load them all* instead of using dot-notation like `tkinter.button()`. Instead we can just say **button()**.
 - ~~After coding along in the video, I noticed that he used “`window = tk()`”, but when I typed that, “`tk`” wasn’t recognized. I decided to go back to the top and “`import tkinter as tk`”. I think he just forgot.~~
 - Turns out “`tk()`” just needed to be capitalized as “`Tk()`”. This may be a holdover from Python 2 since you need to “import Tkinter” in that version.
- From tkinter, the main things we work with are the **window** and the **widgets**.
- We create an empty window variable with:
 - **window = Tk()**
 - After that, we type below that:
 - **window.mainloop()**
 - Everything for our window (widgets, etc) will go between the lines “`window = Tk()`” and “`window.mainloop()`”. If we don’t do this, the window will open and then instantly close, instead of allowing you to click the X in the top corner:

```
from tkinter import *

window = Tk()

window.mainloop()
```

- Running our code as-is brings up an empty window. Now we just need to add widgets such as buttons.
- We’re going to start by adding a button with **b1 = Button()**. The “`Button()`” function takes some arguments, so we can see what arguments it takes by going into a Python interactive shell (in

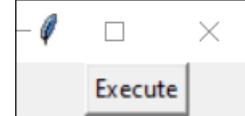
this case, IPython), and running “`from tkinter import *`” followed by “`Button?`” which brought up a list of “STANDARD OPTIONS”, or parameters. The first parameter we want to pass is “`window`”, followed by ‘text=“Execute”’. If we run as-is, the button still doesn’t show up, because we need to use the `b1.pack()` method:

```
from tkinter import *

window = Tk()

b1 = Button(window, text="Execute")
b1.pack()

window.mainloop()
```



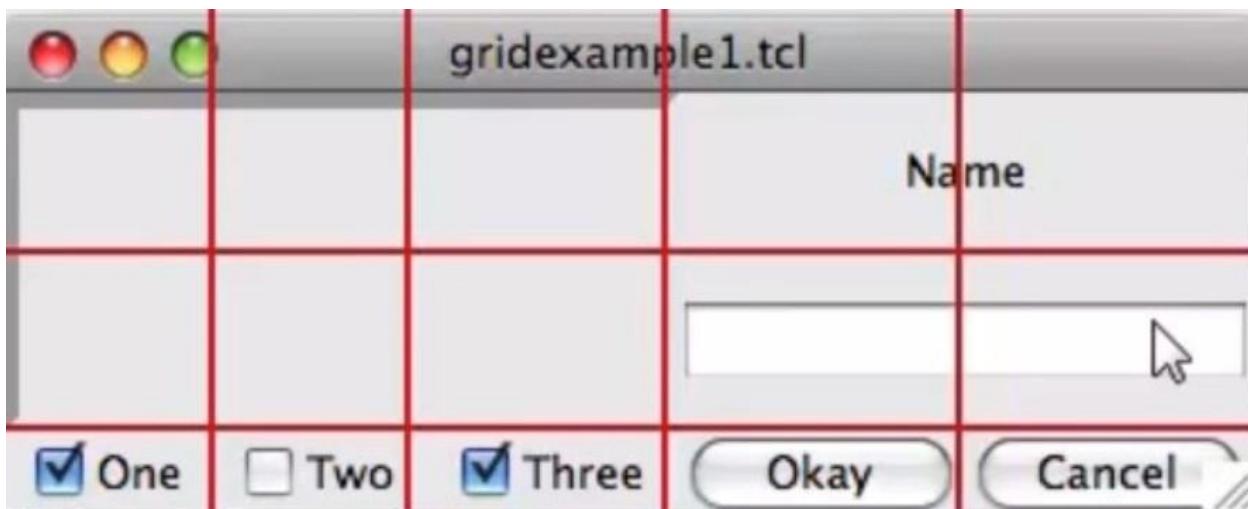
- Now when we run it, we get a window with a clickable button labeled “Execute”. It doesn’t do anything yet though.
- However, there’s another way besides `.pack()` to put your buttons in a window, and that’s with the `.grid()` method.
- Often it’s a matter of preference whether to use `.pack()` or `.grid()`, but with the `.grid()` method you have more control over the position of your buttons or your widgets in general. This is because you can specify the `row` and `column` where you want to put your button:
 - `b1.grid(row=0, column=0)`

```
from tkinter import *

window = Tk()

b1 = Button(window, text="Execute")
b1.grid(row=0, column=0)

window.mainloop()
```



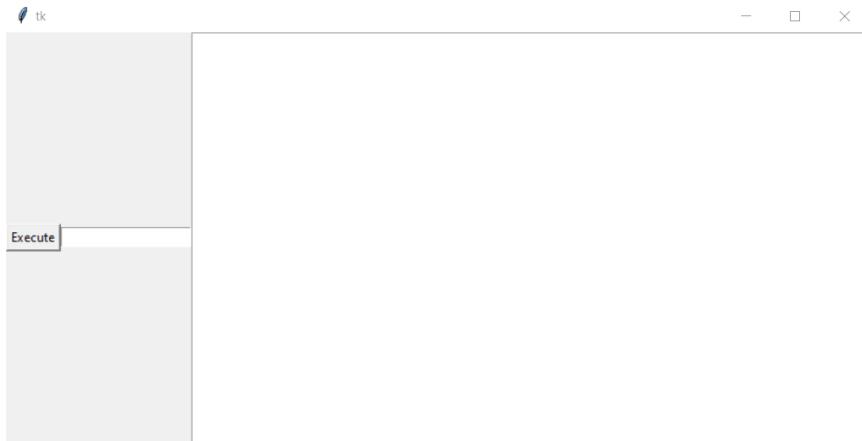
- You can also add a parameter, **rowspan**, to create a button or widget that spans two rows.
 - For example, **b1.grid(row=0, column=0, rowspan=2)** would span **row 0** and **row 1** in this case. I tried it out and the window looked exactly the same as the last example though, so I decided not to include a picture.
- We can also add another widget, let's say an **entry** in this case:

```
e1 = Entry(window)
e1.grid(row=0, column=1)
```

- This adds a text-box entry field to our window → → →
- We can also add a **text widget** with **t1 = Text(window)**:

```
t1 = Text(window)
t1.grid(row=0, column=2)
```

- This gives us a very large text box because that's the default height and width of a **Text** widget:



-
- To fix this we add two more arguments to **Text(window)**, the height and width by *cell*:

```
t1 = Text(window, height=1, width=20)
t1.grid(row=0, column=2)
```



Connecting GUI Widgets with Functions:

- So far our window and its widgets don't do anything.
- What we want at the end of this lecture is, when we type something into the **Entry** file and click the **Execute button**, we want it to print the *miles conversion from kilometers* in the **text field**.
- We want to add a “**command=**” parameter to our **b1 Button**, which takes a *function as an argument* and performs that function when the button is pressed:

```
def km_to_miles(): < < <
    print("Success!") # for testing

b1 = Button(window, text="Execute", command=km_to_miles) < < <
b1.grid(row=0, column=0)
```

- Now when we run our code and click the **Execute button**, our console prints out “Success!”.
- In our **e1 Entry**, we also want to use the “**textvariable=**” parameter to pull in a **StringVar()** string variable from *whatever we type into the Entry*:

```
e1_value = StringVar() < < <
e1 = Entry(window, textvariable=e1_value) < < <
e1.grid(row=0, column=1)
```

- To test, we want to add to our **km_to_miles** function:

```
def km_to_miles():
    print(e1_value.get()) # for testing
```

- Using the **.get()** method pulls the value so it can be printed. Now when we type in a value such as “10” and click on **Execute**, then “10” is printed out in the console.
- Now we want to be able to *insert that value* into our **Text widget**. To do this, we use:

```
def km_to_miles():
    print(e1_value.get()) # for testing
    t1.insert(END, e1_value.get()) < < <
```

- Now every time we type something and click Execute, the string gets tacked onto the end in the Text widget.
- Now we want to make our function actually convert **kilometers to miles**, so we add:

```
def km_to_miles():
    print(e1_value.get()) # for testing
    miles = e1_value.get() * 1.6 < < <
    t1.insert(END, miles)
```

- However, this throws an error. (See next page for solution).
-
-
-

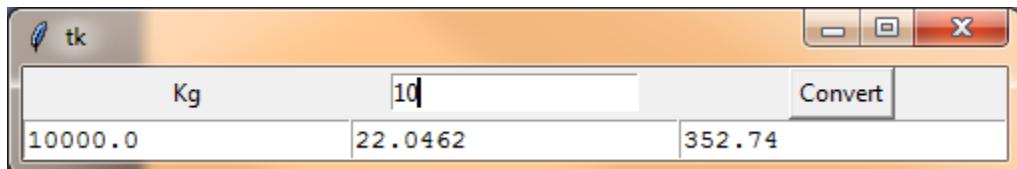
- However, this threw an error, since `e1_value.get()` returns our number as a string. When we initialize our variable `miles`, we need to convert this string into a `float` in order to get things to work:

```
def km_to_miles():
    print(e1_value.get()) # for testing
    miles = float(e1_value.get()) * 1.6 ← ← ←
    t1.insert(END, miles)
```

Exercise: Create a Multi-Widget GUI:

Create a Python program that expects a kilogram input value and converts that value to grams, pounds, and ounces when the user pushes the *Convert* button.

The program will look similar to the one in the following picture:



Tip:

1 kg = 1000 grams

1 kg = 2.20462 pounds

1 kg = 35.274 ounces

My Attempt:

- After some initial trouble bonding 3 different functions to my **b1 Button**, I googled for a bit and decided to use a **lambda function**:
 - Button Command Parameter:
 - **command=lambda: [kg_to_grams(), kg_to_pounds(), kg_to_ounces()]**

```
from tkinter import *
window = Tk()

def kg_to_grams():
    print(e1_value.get()) # for testing
    grams = float(e1_value.get()) * 1000
    t1.insert(END, grams)

def kg_to_pounds():
    print(e1_value.get()) # for testing
    pounds = float(e1_value.get()) * 2.20462
    t2.insert(END, pounds)

def kg_to_ounces():
    print(e1_value.get()) # for testing
    ounces = float(e1_value.get()) * 35.274
    t3.insert(END, ounces)

l1 = Label(window, text="Kg")
l1.grid(row=0, column=0)

b1 = Button(window, text="Convert", command=lambda: [kg_to_grams(),
kg_to_pounds(), kg_to_ounces()])
b1.grid(row=0, column=2)

e1_value = StringVar()
e1 = Entry(window, textvariable=e1_value)
e1.grid(row=0, column=1)

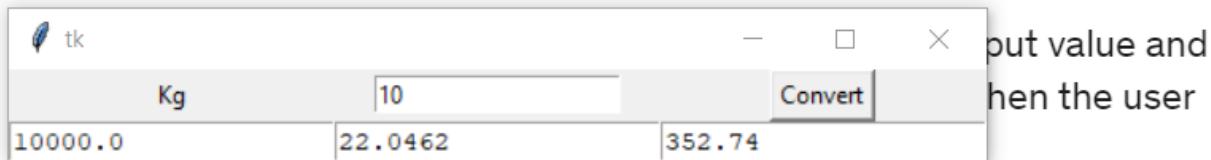
t1 = Text(window, height=1, width=20)
t1.grid(row=1, column=0)

t2 = Text(window, height=1, width=20)
t2.grid(row=1, column=1)

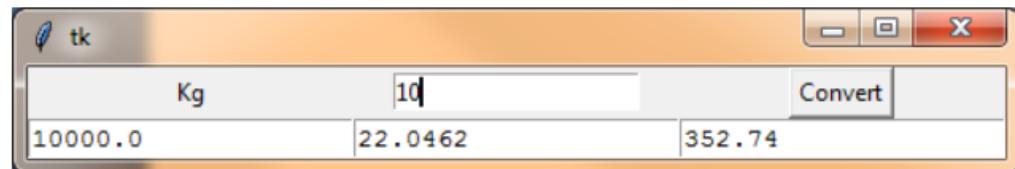
t3 = Text(window, height=1, width=20)
t3.grid(row=1, column=2)

window.mainloop()
```

- I also had to look up how to add the “Kg” to the top left like in the example, and found that I needed to use the **Label** object: **l1 = Label(window, text=“Kg”)**
- After that, it was just a matter of arranging the widgets so they’d end up in the right places, but I figured everything out in a little over 5 minutes:



The program will look similar to the one in the following picture:



Solution: Create a Multi-Widget GUI:

- His solution, for comparison. It seems he used a single function to calculate all three conversions, and also added some functionality to clear out the text boxes if they're filled. I like that second part, so I think I'll add it to my code as well.

```
1. from tkinter import *
2.
3. # Create an empty Tkinter window
4. window=Tk()
5.
6. def from_kg():
7.     # Get user value from input box and multiply by 1000 to get kilograms
8.     gram=float(e2_value.get())*1000
9.
10.    # Get user value from input box and multiply by 2.20462 to get pounds
11.    pound=float(e2_value.get())*2.20462
12.
13.    # Get user value from input box and multiply by 35.274 to get ounces
14.    ounce=float(e2_value.get())*35.274
15.
16.    # Empty the Text boxes if they had text from the previous use and fill them again
17.    t1.delete("1.0", END) # Deletes the content of the Text box from start to END
18.    t1.insert(END, gram) # Fill in the text box with the value of gram variable
19.    t2.delete("1.0", END)
20.    t2.insert(END, pound)
21.    t3.delete("1.0", END)
22.    t3.insert(END, ounce)
23.
24. # Create a Label widget with "Kg" as label
25. e1=Label(window,text="Kg")
26. e1.grid(row=0,column=0) # The Label is placed in position 0, 0 in the window
27.
28. e2_value=StringVar() # Create a special StringVar object
29. e2=Entry(window, textvariable=e2_value) # Create an Entry box for users to enter the
   value
30. e2.grid(row=0,column=1)
31.
32. # Create a button widget
33. # The from_kg() function is called when the button is pushed
34. b1=Button(window,text="Convert", command=from_kg)
35. b1.grid(row=0,column=2)
36.
37. # Create three empty text boxes, t1, t2, and t3
38. t1=Text(window, height=1,width=20)
39. t1.grid(row=1,column=0)
40.
41. t2=Text(window, height=1,width=20)
42. t2.grid(row=1,column=1)
43.
44. t3=Text(window, height=1,width=20)
45. t3.grid(row=1,column=2)
46.
47. # This makes sure to keep the main window open
48. window.mainloop()
```

Section 24: Interacting with Databases:

How Python Interacts with Databases #sql:

Note: Resources for this lecture include Sqlite3 Documentation and Psycopg Documentation.

- SQLite and PostgreSQL are different in that SQLite is not a client-server database; instead it is embedded into the end program.
- All of the data in his example BookStore app are stored in a **.db filetype**. This makes SQLite portable as long as another user has SQLite installed.
- PostgreSQL would be more appropriate for web applications.
- The library to work with an SQLite database is **sqlite3**.
- The library to work with a PostgreSQL database is **psycopg2**.

Connecting to an SQLite Database with Python:

- SQLite3 is a built-in Python library, so you don't need to install it.
- Now, the standard process for creating a database consists of 5 steps:
 - First you connect to the database.
 - Second you create a cursor object; the cursor object is like a pointer to access rows from a table for the database.
 - Third is you apply an SQL query; you may want to insert data into the database or select data from a table in the database.
 - Fourth you commit your changes to the database.
 - Fifth, you close the connection.

```
import sqlite3  
  
conn = sqlite3.connect("lite.db") ← ← ←
```

- You create a variable **conn** and you pass in your database file ("lite.db" here). If you don't have an existing database file with that name, this line will create one for you and a connection will be established. If you do have a database with this name, a connection will be established to the existing database.
- You then create a cursor object with **curr=conn.cursor()**, then use that cursor to **execute** some SQL code "**CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)**".

- We then want to *commit* these changes to the database with `conn.commit()`, then *close the connection* with `conn.close()`.

```
import sqlite3

conn = sqlite3.connect("lite.db")
cur=conn.cursor()
cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
conn.commit()
conn.close()
```

- Executing our code as-is creates the **lite.db** file in our directory, filled with the information we just committed.
- If we run the program again now, we get an error because the *table already exists*. To fix this we just need to apply a simple trick by *adding some code into our SQL code*:
 - Add “**IF NOT EXISTS**” before the table name “**store**”:

```
import sqlite3

conn = sqlite3.connect("lite.db")
cur=conn.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT, quantity INTEGER,
price REAL)")
conn.commit()
conn.close()
```

- Now we want to add some data to our SQL table by adding the following line underneath our (“**CREATE TABLE...**”) line:

```
cur.execute("INSERT INTO store VALUES ('Wine Glass', 8, 10.5)")
```

- Now if we wanted to insert another item, *we could* insert a similar line below that, but then we’d get a duplicate of ‘**Wine Glass**’. So what we could do in this case is, we could use a **function** to wrap around our **separate SQL statements**.

```

import sqlite3

def create_table(): < < <
    conn = sqlite3.connect("lite.db")
    cur=conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT, quantity
INTEGER, price REAL)")
    conn.commit()
    conn.close()

def insert(item, quantity, price): < < <
    conn = sqlite3.connect("lite.db")
    cur=conn.cursor()
    cur.execute("INSERT INTO store VALUES (?, ?, ?)", (item, quantity, price))
    conn.commit()
    conn.close()

insert("Water Glass", 10, 5) < < <
insert("Coffee Cup", 10, 5) < < <

```

- Now you may want to see the data that you're inserting, so let's create a function for that. To do this, we use SQL's common "SELECT * FROM" syntax:

```

def view():
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM store") < < <
    rows = cur.fetchall() < < <
    conn.close()
    return rows < < <

print(view()) < < <

```

- This will use SQL syntax to **SELECT * FROM store**, then use sqlite3 syntax to set a variable **rows = cur.fetchall()**, then return the variable rows, then print out the function's return value with **print(view())**. Note that this returns a Python *list*.
 - Returns "[('Wine Glass', 8, 10.5), ('Water Glass', 10, 5.0), ('Coffee Cup', 10, 5.0), ('Coffee Cup', 10, 5.0)]". We ended up with a duplicate of 'Coffee Cup' because we left in the "insert" function when we ran the program again.
- Full code, next page:

Full Lecture Code:

```
import sqlite3

def create_table():
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT, quantity INTEGER,
price REAL)")
    conn.commit()
    conn.close()

def insert(item, quantity, price):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("INSERT INTO store VALUES (?, ?, ?)", (item, quantity, price))
    conn.commit()
    conn.close()

insert("Coffee Cup", 10, 5)

def view():
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM store")
    rows = cur.fetchall()
    conn.close()
    return rows

print(view())
```

(SQLite) Selecting, Inserting, Deleting, and Updating SQL Records:

- Now we're going to create functions for deleting and updating SQL records.
- We started by copy/pasting our `view()` function because we're going to tweak it and reuse most of its code. We're going to *delete* the row that contains "Wine Glass". Since we're making changes, we want to make sure to `.commit()` those changes to our database:

```
def delete(item): < < <
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM store WHERE item=?", (item)) < < <
    conn.commit() < < <
    conn.close()
```

- We then ran `delete("Wine Glass")` along with the whole program, but we got an error. It turns out that `(item)` needed a trailing comma or else you get an `sqlite3.ProgrammingError`:
"Incorrect number of bindings supplied. The current statement uses 1, and there are 10 supplied":

```
def delete(item):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM store WHERE item=?", (item,)) < < <
    conn.commit()
    conn.close()

delete("Wine Glass")
```

- Adding this comma fixes the error and allows "Wine Glass" to be deleted from the `store` table.
Note that this kind of error happens with **SQLITE**, but *may not be true of other types of databases*.
- We're now going to add another function `update()`. We want to change the *quantity* of 'Water Glass' from **10** to **11**:

```
def update(quantity, price, item):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("UPDATE store SET quantity=?, price=? WHERE item=?", 
    (quantity, price, item))
    conn.commit()
    conn.close()

update(11, 6, "Water Glass")
```

- Note that when we have more than one parameter in our `.execute()` method, we don't need the trailing comma.
- Now when we run the code, it returns our updated values for "Water Glass": ('Water Glass', 11, 6.0).
- Note that it could be more useful to use an **ID** in this situation rather than the **item name**. To do this, we'd want to change the table at the beginning and add a **PRIMARY KEY** that auto-increments. However, we're going to learn how to do this in the next section of the course.

PostgreSQL Database with Python:

- In this next lecture, we're going to use the same script as the previous lectures, but we're going to modify it so that it's compatible/interactive with PostgreSQL databases. Luckily, most the code is the same with some slight changes. We'll also be using a library called **psycopg2**. This isn't a built-in library, so we'll need to install both that and **PostgreSQL**.
- We downloaded and ran the setup .exe.
 - During setup, we were assigned the *superuser name* **postgres**, and I chose the same *password* as **postgres123**.
- It also offered to download various extensions and drivers, we just kind of chose some spatial one at random. These can probably be added later by searching "**Stack Builder**" in the Windows search bar. We can also find "**pgAdmin**" and "**Application Stack Builder**" instead our PostgreSQL folder.
- However, since we want to access PostgreSQL through Python, we don't need to use any of these options. The way we access it through Python is by **installing psycopg2** by running:
 - **pip install psycopg2**
 - In older versions (this issue didn't crop up for me), you may get an error/warning saying that "psycopg is written in C, so you need a C compiler" in Windows. They must have fixed this after the lecture was filmed.
 - He noted after this that you can install Visual Studio to get around this, and I've been doing this whole course in VSCode, so maybe that's why it worked.
 - Second solution is to use pre-compiled Python libraries. You can find them through a Google search, download them, and use them. He put his downloaded .whl file into the same directory as the Python scripts we're working on.
- In the next lecture, we'll actually use all this.

(PostGreSQL) Selecting, Inserting, Deleting, and Updating SQL Records:

Note: Resources for this lecture include Sqlite3 Documentation, Psycopg2 Documentation, and PostgreSQL Documentation.

- In this lecture, we're going to go through our previous code and replace *only a few lines* in order to get it to work with PostgreSQL.
- The main difference is that the main database that we'll be working with *will not be stored as a .db file*. It will be a **database embedded in our PostgreSQL installation**. This requires there to be an existing database, and PostgreSQL comes with a database called **Postgres**, so we can just pass the database there. Or, you can create your own database:
 - To create a new database, we go into **pgAdmin**.
 - Once open, pgAdmin lists our **servers** and our **default database (Postgre)**.
 - Either way, you *need to connect to your database* and input your admin password.
 - We **created a new database ("database1")**, then closed pgAdmin. Now we go back to Python.
- First off we changed our **import** statement from **sqlite3** to **psycopg2**. We also did a **batch-replace** of all mentions of **sqlite3** to **psycopg2**. Most of the rest of the code/syntax will remain the same due to commonalities between SQL.
- We also replaced all mention of "lite.db" with:
 - (**"dbname='database1' user='postgres' password= 'postgres123' host='localhost' port='5432'"**)

```
import psycopg2

def create_table():
    conn = psycopg2.connect("dbname='database1' user='postgres'
password='postgres123' host='localhost' port='5432'")
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT,
    quantity INTEGER, price REAL)")
    conn.commit()
    conn.close()
```

- To test it out, we ran **create_table()** at the bottom, saved and ran the script. To check this table, we can go into **pgAdmin** and look for it in **database1 >>> schema >>> public >>> tables**.
-
- We then copy/pasted our new connection string into all the locations where "lite.db" previously was.
-
-
-
-

- We can also replace all the “?” placeholders from sqlite3 with **string formatting**: ‘%s’, followed by a tuple of the parameters:

```
def insert(item, quantity, price):
    conn = psycopg2.connect("dbname='database1' user='postgres'
password='postgres123' host='localhost' port='5432'")
    cur = conn.cursor()
    cur.execute("INSERT INTO store VALUES ('%s', '%s', '%s')" % (item,
quantity, price))
    cur.execute("INSERT INTO store VALUES (%s, %s, %s)" % (item, quantity,
price))
    conn.commit()
    conn.close()
```

- However, he noted that this is risky because it makes the database vulnerable to SQL injection attacks from hackers.
- An alternative is to pass our parameters a second variable to the **.execute** method:

```
#cur.execute("INSERT INTO store VALUES ('%s', '%s', '%s')" % (item,
quantity, price))
cur.execute("INSERT INTO store VALUES (%s, %s, %s)", (item, quantity,
price))
```

-
- Now we'll look at the **view()** function, then the **delete()** function. With slight adjustments, they work exactly the same as they did with sqlite3. He accidentally inserted 3 copies of “Orange”, so he ran **delete(“Orange”)** and it got rid of all rows containing “Orange”.
- Next up is **update()**. Same, same.

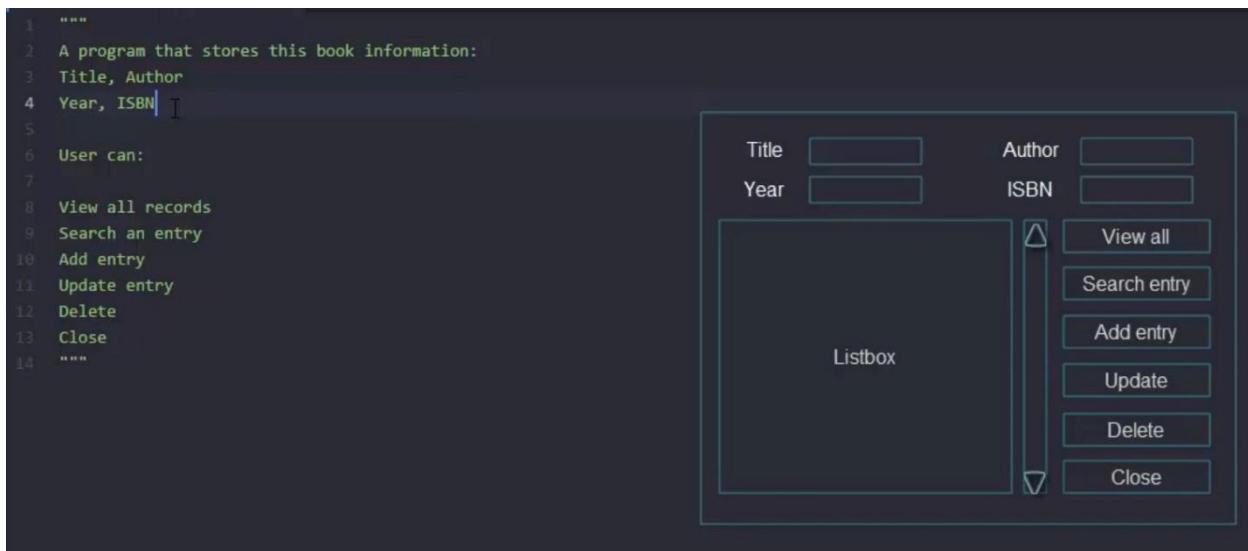
Section 25: App 5: GUI Apps and SQL: Build a Book Inventory Desktop GUI Database App:

Demo of the Book Inventory App:

- He started off by showing his .exe file for this section, **BookStore.exe**. In the same directory was an SQLite3 .db file, **lite1.db**, so it seems like we'll be using that to start with.
 - This will be the first .exe file I've ever created, so that's exciting.
- BookStore.exe was created in **tkinter** for the GUI and **SQLite3** for the backend.
- It contains four entry fields:
 - **Title**
 - **Author**
 - **Year**
 - **ISBN**
- Below that, there's a **text box** that *selects/displays* books from the database, and six buttons:
 - **View All**
 - **Search Entry**
 - **Add Entry**
 - **Update Selected**
 - **Delete Selected**
 - **Close**

Designing the User Interface:

- We want to start by defining some requirements for the GUI.
- **Requirements:** We want our program to be able to **show a list** of current records, so when we press “**View All**”, the current list of books is show in a text box. We also want to be able to **Search** for an entry, **Add** a new entry, **Update** an existing entry, **Delete** a current entry, and **Close** the window. These are our **requirements**.
- **Backend:** Now if we want we can choose to start *writing the backend*, so we can write a function that **selects** all the data in the database and return it in text. This fulfills the **View All** button. Other functions can be written to support the other buttons, and once this is complete we can start writing the frontend GUI so that a user can use these buttons.
- **Frontend:** Or, we can start *writing the frontend* first. So we build the **GUI** and place the buttons, but since we haven’t written backend functions yet, these buttons won’t do anything. Once the GUI is complete, we can write the backend to support the buttons and functionality of the program.
- In our case, we’re going to *start with the frontend GUI* first and then we’re going to build the backend after. We’ll build a GUI that will do nothing for a while, until we connect it to the backend.



- Note: Above we start off our program with some human-readable text outlining what our program should be able to do. On the left is a simple graphical representation of some of the features we might want to add to our program, such as the four **entry fields** up top, the **list-box** where our text list of books will be displayed, a **scrollbar**, and our **function buttons**.
 - Having a sketch/drawing of what we want our GUI to look like is important to help us visualize both our finished product, and how we might go about giving our program the required functionality.

Coding the Frontend Interface:

- To put widgets in a GUI using tkinter, we can use either the `.pack()` method or the `.grid()` method. Since we'll be using the `.grid()` method, it's a good idea to draw an actual grid on our sketch of the GUI. This way we have an idea of which row number and column number to pass to our widgets:



- At this point he decided to delete the **docstring** because we don't really need it for our purposes, and it saves space on our screen.

Labels:

- So we start off by importing ALL from tkinter (**from tkinter import ***), and then we create our window, **window=Tk()**. From there, we create our four **label objects**:

```
from tkinter import *

window = Tk() ← ← ← Create window

l1 = Label(window, text="Title") ← ← ← Create labels
l1.grid(row=0, column=0)

l2 = Label(window, text="Author") ← ← ←
l2.grid(row=0, column=2)

l3 = Label(window, text="Year") ← ← ←
l3.grid(row=1, column=0)

l4 = Label(window, text="ISBN") ← ← ←
l4.grid(row=1, column=2)

window.mainloop()
```

- Running our Python script at this point yields a small window with a two-by-two grid showing just our four labels.

Entries:

- Next up, we add our **entries**, but first we need to create a **StringVar** special object to pass to our **textvariable** argument in our **Entry()** function:

```
title_text = StringVar() ← ← ← StringVar
e1=Entry(window, textvariable=title_text) ← ← ← pass StringVar to Entry
e1.grid(row=0, column=1)

author_text = StringVar()
e2=Entry(window, textvariable=author_text) ← ← ←
e2.grid(row=0, column=3)

year_text = StringVar()
e3=Entry(window, textvariable=year_text) ← ← ←
e3.grid(row=1, column=1)

isbn_text = StringVar()
e4=Entry(window, textvariable=isbn_text) ← ← ←
e4.grid(row=1, column=3)
```

Listbox:

- Next is the **Listbox**. We need to create our **list1** Listbox object, and we want to use the **rowspan** and **columnspan** arguments to make it span several rows and columns, to get everything in our grid to line up nicely.
 - We also want to create a **Scrollbar** object to the right of our Listbox. To do this, we create our scrollbar object, and *then we tell our scrollbar and our Listbox about each other so they can interact*:

```
list1 = Listbox(window, height=6, width=35) ← ← ← Create listbox
list1.grid(row=2, column=0, rowspan=6, columnspan=2)

sb1 = Scrollbar(window) ← ← ← Create scrollbar
sb1.grid(row=2, column=2, rowspan=6)

list1.configure(yscrollcommand=sb1.set) ← ← ← Configure the two together
sb1.configure(command=list1.yview) ← ← ←
```

Buttons:

- Now we just need to add our six button widgets to the bottom-right:

```
b1 = Button(window, text="View All", width=12, <command=>) ← ← ←
b1.grid(row=2, column=3)

b2 = Button(window, text="Search Entry", width=12)
b2.grid(row=3, column=3)

b3 = Button(window, text="Add Entry", width=12)
b3.grid(row=4, column=3)

b4 = Button(window, text="Update Selected", width=12)
b4.grid(row=5, column=3)

b5 = Button(window, text="Delete Selected", width=12)
b5.grid(row=6, column=3)

b6 = Button(window, text="Close", width=12)
b6.grid(row=7, column=3)
```

- To get these buttons to work, we'd need to add a **command=** parameter, but for now we're just trying to get the buttons to display. The **command=** parameter will be used later when we attach our GUI to the **backend**.
- In the next lecture, we'll code the **backend** and then we'll connect the GUI to that.

Coding the Backend:

- So now we have our GUI. Now we need to create functions for all of its buttons, and connect it to our SQL database. We're going to create this with SQLite3, and we need to create a table.
- In this case, we're going to create a new script, **backend.py**, and then import that into **script1.py**. We're also going to rename **script1.py** to **frontend.py**.
 - Once we have some functions in **backend.py**, we can attach **command=** variables to our GUI buttons using the syntax “**command=backend.<etc>**”.
- All of our functions are going to be very similar to our SQLite3 script from the last section, but with **title**, **author**, **year**, and **isbn** instead of things like price and quantity.

```
def search(title="", author="", year="", isbn=""): ← ← ← need empty strings
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM book WHERE title=? OR author=? OR year=? OR
isbn=?", (title, author, year, isbn))
    rows = cur.fetchall()
    conn.close()
    return rows

def delete(id):
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM book WHERE id=?", (id,)) ← ← ← need trailing
comma
    conn.commit()
    conn.close()
```

- Note that in the **search()** function we need to pass some *empty strings as defaults* to prevent an error. This is because our **.execute()** method expects four parameters.
- Remember also that when passing a tuple of *one parameter* such as **id** (as is the case in the **delete()** function), we need a trailing comma.
- Once we have all our functions (**connect()**, **insert()**, **view()**, **search()**, **delete()**, and **update()**) we can work on connecting these functions to our GUI buttons.
- Full **backend.py** code follows:

```
import sqlite3

def connect():
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS book (id INTEGER PRIMARY KEY, title
TEXT, author TEXT, year INTEGER, isbn INTEGER)")
    conn.commit()
    conn.close()
```

```

def insert(title, author, year, isbn):
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("INSERT INTO book VALUES (NULL, ?, ?, ?, ?)", (title, author,
year, isbn))
    conn.commit()
    conn.close()

def view():
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM book")
    rows = cur.fetchall()
    conn.close()
    return rows

def search(title="", author="", year="", isbn ""):
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM book WHERE title=? OR author=? OR year=? OR
isbn=?", (title, author, year, isbn))
    rows = cur.fetchall()
    conn.close()
    return rows

def delete(id):
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM book WHERE id=?", (id,))
    conn.commit()
    conn.close()

def update(id, title, author, year, isbn):
    conn = sqlite3.connect("books.db")
    cur = conn.cursor()
    cur.execute("UPDATE book SET title=?, author=?, year=?, isbn=? WHERE id=?",
(title, author, year, isbn, id))
    conn.commit()
    conn.close()
connect()

```

Connecting the Frontend with the Backend, Part 1:

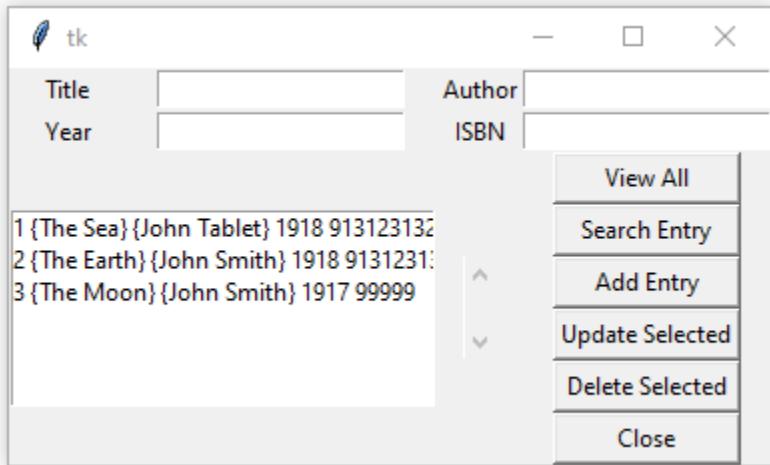
- Now we need to connect our frontend GUI with our backend so that it can fetch our backend function outputs and output or change the desired data.
- The first thing we do in our **frontend.py** script is to add **import backend**. This will allow us to access the functions in backend.py.
- We're going to start off by connecting the “View All” button to backend's **view()** function. When we click this button, we want the backend function to *output a list* of our book data *into the listbox*:

```
b1 = Button(window, text="View All", width=12, command=view_command) ← ← ←  
b1.grid(row=2, column=3)
```

- Note: When we pass commands to buttons, we leave out the parenthesis “()”. This is to prevent Python from executing the function; we want the function to execute *only when* the user presses the button.
- After adding this command, we need to go back to the top and write a function “**view_command**” that will take the output of **backend.view** and print it to the listbox:

```
def view_command(): ← ← ←  
    for row in backend.view():  
        list1.insert(END, row)
```

- Note here that we used the argument **END** in our insert method. This causes items to be added to the end of the listbox.
-
- Running our **frontend.py** and clicking “View All” results in this:



-
- Our books in the listbox should be the same as when we were testing our the backend by printing them out in the console. However, at this point if we press “View All” multiple times, it just keeps adding the list over and over. We want it to clear out the listbox first thing:

```
def view_command():
    list1.delete(0, END) ← ← ←
    for row in backend.view():
        list1.insert(END, row)
```

-
- Now we'll work on the “**Search Entry**” button, which we want to return matching entries to the listbox. We'll create a “**search_command**” **wrapper function** for this similar to with the “View All” button.

```
b2 = Button(window, text="Search Entry", width=12, command=search_command) ←
b2.grid(row=3, column=3)
```

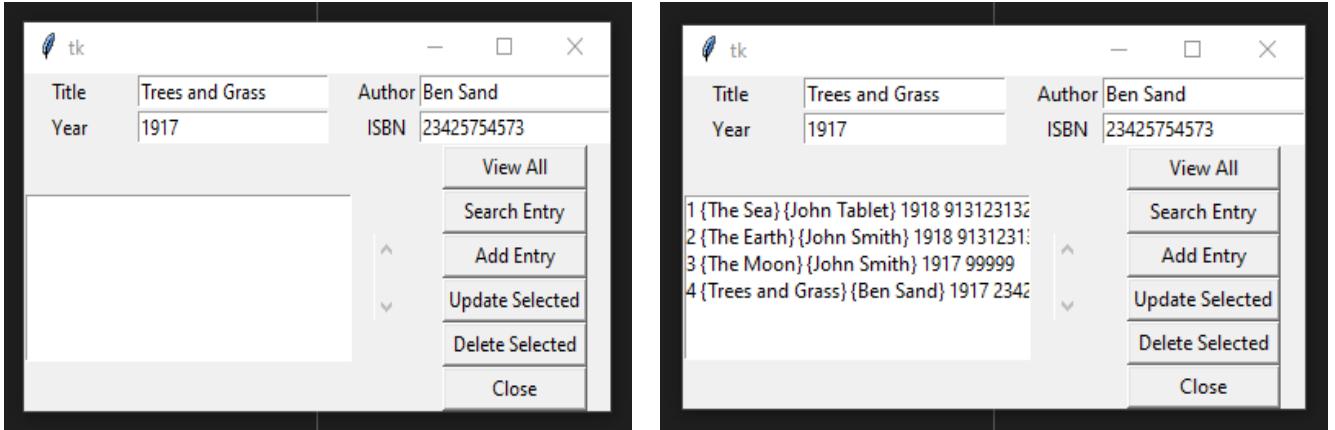
- A **wrapper function** is especially useful in cases like this because we'll be passing parameters to the function.
- Our backend **search()** function has four parameters. We can't put parenthesis in our **command=** parameter, so we put them in our wrapper function.
- We'll be getting these parameters from our **entry widgets**, such as **title_text = StringVar()**, etc:

```
def search_command():
    list1.delete(0, END)
    for row in backend.search(title_text.get(), author_text.get(),
year_text.get(), isbn_text.get()):
        list1.insert(END, row)
```

- Note that we used the **.get()** method to convert all our **StringVar()** variables into **strings**.
- Our GUI now allows us to search for books based on those four variables.
-
- Next up we want to work on the “**Add Entry**” button. This means that we'll be calling the **insert(title, author, year, isbn)** function in the backend. This will be a simple function since we're just going to *insert* the same four **.get()** strings as before:

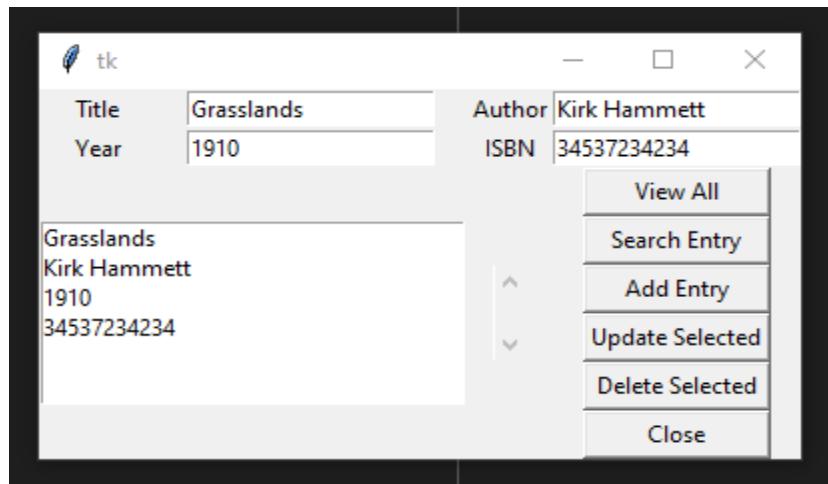
```
def add_command():
    backend.insert(title_text.get(), author_text.get(), year_text.get(),
isbn_text.get())
```

- And that's it. Filling out entries in our GUI and selecting “**Add Entry**” will now add entries to our database in the backend. We tested by adding some entries:



- It would be good if, when an entry is added, it automatically populates in the listbox so that the user knows that it was successfully added:

```
def add_command():
    backend.insert(title_text.get(), author_text.get(), year_text.get(),
isbn_text.get())
    list1.delete(0, END) < < <
    list1.insert(END, title_text.get(), author_text.get(), year_text.get(),
isbn_text.get()) < < <
```



- Almost there. The information populates the listbox, but it's in the wrong format. We can fix this simply by passing the book information as a single tuple:

```
def add_command():
    backend.insert(title_text.get(), author_text.get(), year_text.get(),
isbn_text.get())
    list1.delete(0, END)
    list1.insert(END, (title_text.get(), author_text.get(), year_text.get(),
isbn_text.get())) < < <
```

Connecting the Frontend with the Backend, Part 2:

- We still have a few buttons to work on: **Delete**, **Update**, and **Close**.
- First off, **Delete**. We need to think about what the user might expect to happen. Say we want to delete one of the rows: the entry should be deleted from the database, and the updated list of entries should populate the listbox.
 - As an aside, perhaps when they click on a row, they may want to see that same data populate in the **entries**. We'll do that too.
- Going to look at the **backend**, the **delete(id)** function takes an **id** as its parameter. So when a user clicks on an item in the listbox, we want to *grab* that *id* and send it to the **delete()** function of the backend script.
- There is a **bind()** function in the **tkinter** library which is used to *bind* a function to a *widget event*. We're going to *bind* a method to the *listbox widget* **list1**:
 - **list1.bind()**
 - This **.bind()** takes two arguments: an *event type* and a *function* that we want to bind to the event type:

```
list1.bind('<<ListboxSelect>>', get_selected_row)
```

- Now we need to go up above and write our **get_selected_row** function at the beginning of our program:

```
def get_selected_row(event):
    index = list1.curselection()
    print(index)
```

- We added a temporary **print** just so we can check what datatype this returns as. It returns a **tuple** such as **(2,)**.
- Doing this instead...

```
def get_selected_row(event):
    index = list1.curselection()[0] ← ← ←
    print(index)
```

- ...prints them out as simple integers, such as **2**. So we're very close now. We can use the index to **.get()** the selected tuple from the database:

```
def get_selected_row(event):
    index = list1.curselection()[0]
    selected_tuple = list1.get(index)
    print(selected_tuple)
```

- Now when we select items from our listbox, the tuple containing the item's information is printed to the console.
-
- We can use all of this to pass the **id** of an item to the **delete(id)** function in the backend of the script.

- So we go down to our “**Delete Selected**” button code and add a wrapper function command:

```
b5 = Button(window, text="Delete Selected", width=12, command=delete_command)
b5.grid(row=6, column=3)
```

- Now we go up top and add our wrapper function:

```
def delete_command():
    backend.delete(get_selected_row()[0])
```

- However, if we run this as-is, we get an error because `get_selected_row()` is expecting an `event` argument. What we need in this case is to declare `selected_tuple` as a **global variable** inside our `get_selected_row()` function. We also no longer need our return statement in the function:

```
def get_selected_row(event):
    global selected_tuple < < <
    index = list1.curselection()[0]
    selected_tuple = list1.get(index)
```

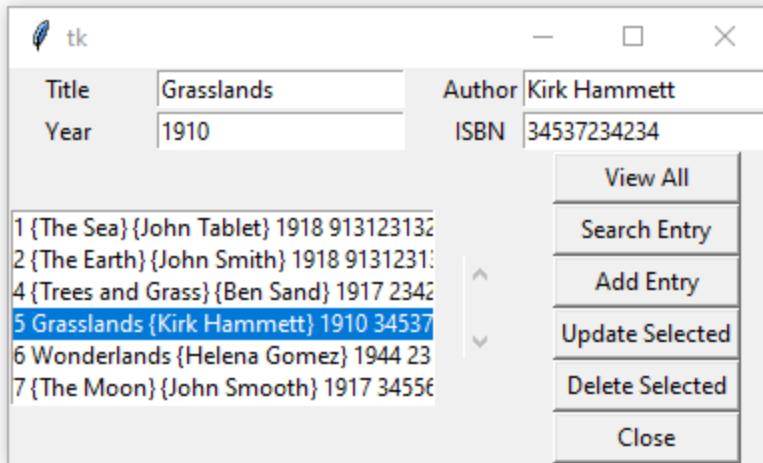
- We also want to modify our `delete_command()` function to use `selected_tuple`:

```
def delete_command():
    backend.delete(selected_tuple[0]) < < <
```

- We’re going to test this out by deleted our entry “**The Moon**”. However, at this stage we don’t see it disappear from the list unless we click “**View All**”, so we may want to change this to update the listbox instantly.
 - I ended up adding this just to my own code, as he didn’t address it at this stage. I reused some code from our `view_command` wrapper function.
 -
 - Next up, we want to populate our entry fields whenever we select an item from the list. We can go ahead and write that functionality directly inside our `get_selected_row()` function:

```
def get_selected_row(event):
    global selected_tuple
    index = list1.curselection()[0]
    selected_tuple = list1.get(index)
    e1.delete(0, END)
    e1.insert(END, selected_tuple[1])
    e2.delete(0, END)
    e2.insert(END, selected_tuple[2])
    e3.delete(0, END)
    e3.insert(END, selected_tuple[3])
    e4.delete(0, END)
    e4.insert(END, selected_tuple[4])
```

- This inserts the indexed values from an item’s tuple into the desired `entry`:



-
- Next up, we need to update our “**Update Selected**” button. In our `backend.py` script, our `update()` function gets passed `id`, `title`, `author`, `year`, and `isbn`.
- The `update_command` function will be similar to the delete one, which also got passed a variable (`id`). But this time, we’ll be passing a total of five variables, all indexes of `selected_tuple`:

```
def update_command():
    backend.update(selected_tuple[0], selected_tuple[1], selected_tuple[2],
    selected_tuple[3], selected_tuple[4]) ←←←
```

- And of course we need to add `command=update_command` down in the button definition. We tested out our new updating capabilities by changing an ISBN value, but we found that nothing was being updated. The reason for this is that we actually want to send `title_text.get()`, etc to our backend `update()` function:

```
def update_command():
    backend.update(selected_tuple[0], title_text.get(), author_text.get(),
    year_text.get(), isbn_text.get()) ←←←
```

- Note: I noticed that, while my program functions correctly now, if I drag-select text in any of the entries, my console will say that there’s an “`IndexError: tuple index out of range`”. It doesn’t affect my program, so I’ll leave it alone for now, but it’s interesting to note.
-
- Now we just need to work on our “**Close**” button. For this, we just need to add `command>window.destroy` to the “Close” button’s parameters:

```
def update_command():
    backend.update(selected_tuple[0], title_text.get(), author_text.get(),
    year_text.get(), isbn_text.get()) ←←←
```

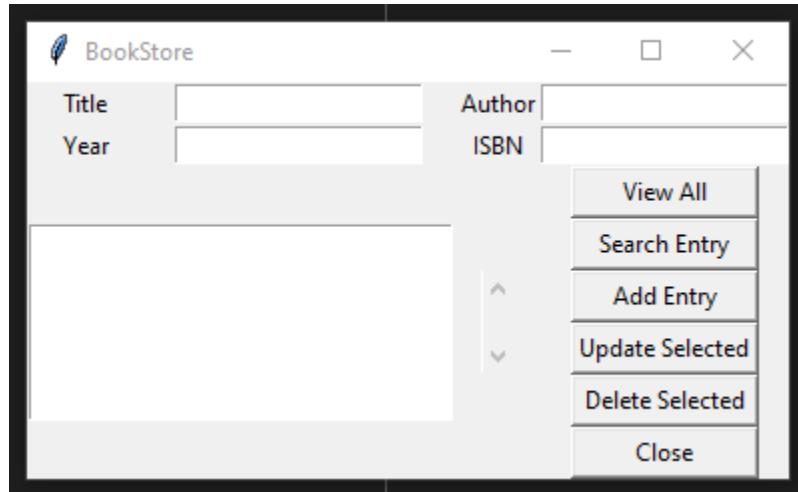
- And it works.

- We may also want to add a title to our window:

```
# Create window
window = Tk()

# Gives window a title
window.wm_title("BookStore")
```

- And now our window has a title:



-
- And we're finished.

Full Frontend Code, Connected to Backend:

```
from tkinter import *
import backend

# Creates function to get information from selecting items in listbox
# Also populates entry fields with selected listbox item's information
def get_selected_row(event):
    global selected_tuple
    index = list1.curselection()[0]
    selected_tuple = list1.get(index)
    # Inserts tuple index items into desired entry fields
    e1.delete(0, END)
    e1.insert(END, selected_tuple[1])
    e2.delete(0, END)
    e2.insert(END, selected_tuple[2])
    e3.delete(0, END)
    e3.insert(END, selected_tuple[3])
    e4.delete(0, END)
    e4.insert(END, selected_tuple[4])

# Wrapper functions for buttons:
# Creates wrapper function for "View All" button
def view_command():
    list1.delete(0, END)
    for row in backend.view():
        list1.insert(END, row)

# Creates wrapper function for "Search Entry" button
def search_command():
    list1.delete(0, END)
    for row in backend.search(title_text.get(), author_text.get(),
year_text.get(), isbn_text.get()):
        list1.insert(END, row)

# Creates wrapper function for "Add Entry" button
def add_command():
    backend.insert(title_text.get(), author_text.get(), year_text.get(),
isbn_text.get())
    list1.delete(0, END)
    list1.insert(END, (title_text.get(), author_text.get(), year_text.get(),
isbn_text.get()))

# Creates wrapper function for "Delete Selected" button
```

```

def delete_command():
    backend.delete(selected_tuple[0])
    list1.delete(0, END)
    for row in backend.view():
        list1.insert(END, row)

# Creates wrapper function for "Update Selected" button
def update_command():
    backend.update(selected_tuple[0], title_text.get(), author_text.get(),
year_text.get(), isbn_text.get())
    list1.delete(0, END)
    for row in backend.view():
        list1.insert(END, row)

# Create window
window = Tk()

# Gives window a title
window.wm_title("BookStore")

# Create labels
l1 = Label(window, text="Title")
l1.grid(row=0, column=0)

l2 = Label(window, text="Author")
l2.grid(row=0, column=2)

l3 = Label(window, text="Year")
l3.grid(row=1, column=0)

l4 = Label(window, text="ISBN")
l4.grid(row=1, column=2)

# Create entry fields/boxes
title_text = StringVar()
e1=Entry(window, textvariable=title_text)
e1.grid(row=0, column=1)

author_text = StringVar()
e2=Entry(window, textvariable=author_text)
e2.grid(row=0, column=3)

year_text = StringVar()
e3=Entry(window, textvariable=year_text)
e3.grid(row=1, column=1)

```

```

isbn_text = StringVar()
e4=Entry(window, textvariable=isbn_text)
e4.grid(row=1, column=3)

#Create listbox
list1 = Listbox(window, height=6, width=35)
list1.grid(row=2, column=0, rowspan=6, columnspan=2)

# Create scrollbar
sb1 = Scrollbar(window)
sb1.grid(row=2, column=2, rowspan=6)

# Configure listbox and scrollbar to interact
list1.configure(yscrollcommand=sb1.set)
sb1.configure(command=list1.yview)

list1.bind('<<ListboxSelect>>', get_selected_row)

# Create buttons
# After "backend.py" created, button commands can reference backend functions
b1 = Button(window, text="View All", width=12, command=view_command)
b1.grid(row=2, column=3)

b2 = Button(window, text="Search Entry", width=12, command=search_command)
b2.grid(row=3, column=3)

b3 = Button(window, text="Add Entry", width=12, command=add_command)
b3.grid(row=4, column=3)

b4 = Button(window, text="Update Selected", width=12, command=update_command)
b4.grid(row=5, column=3)

b5 = Button(window, text="Delete Selected", width=12, command=delete_command)
b5.grid(row=6, column=3)

b6 = Button(window, text="Close", width=12, command=window.destroy)
b6.grid(row=7, column=3)

# Mainloop window
window.mainloop()

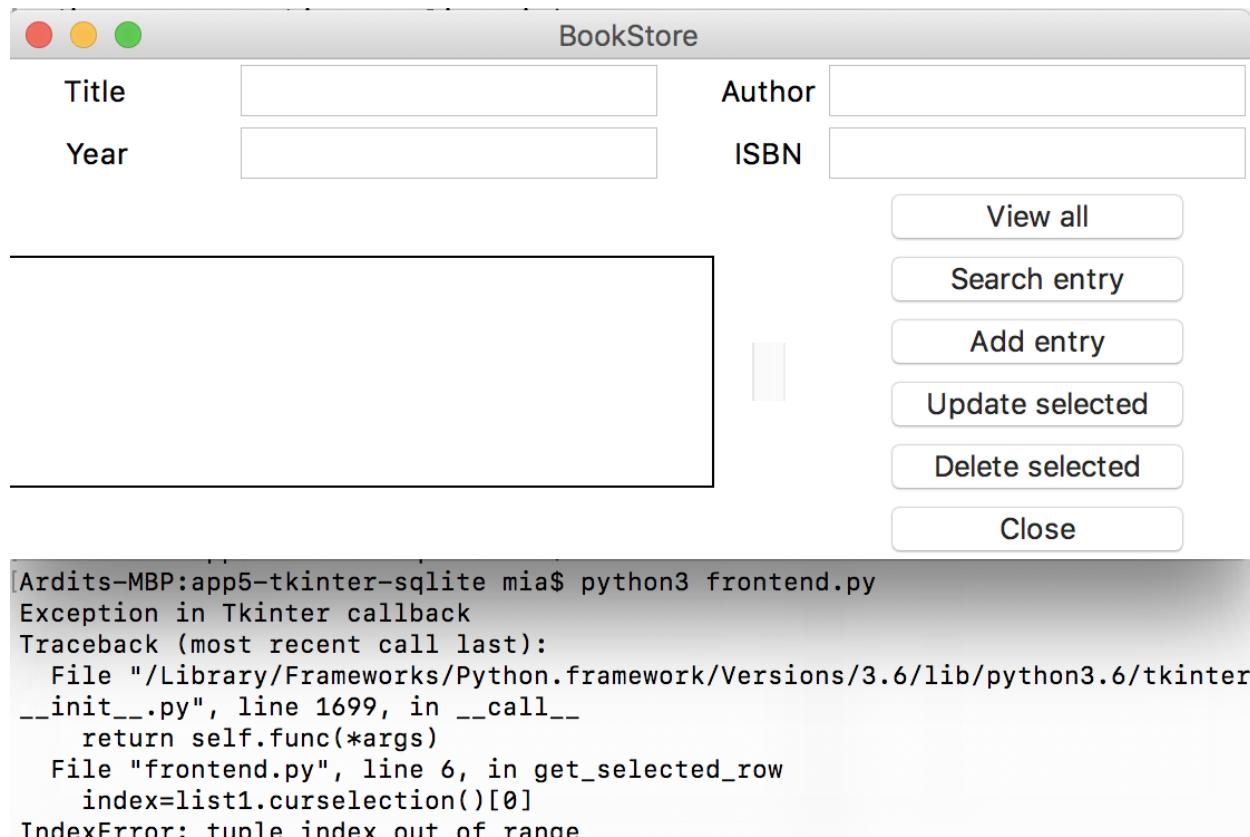
```

Exercise: Fixing a Bug in Our Program:

- Sounds like this will address that IndexError I noted earlier.

Exercise

If you haven't already noticed, the program has a bug. When the listbox is empty and the user clicks the listbox, an *IndexError* is generated in the terminal:



Why does this error happen?

Well, everything starts with the user clicking on the listbox. Clicking the listbox executes the following code:

```
list1.bind('<<ListboxSelect>>',get_selected_row)
```

That code calls the `get_selected_row` function:

```
1. def get_selected_row(event):
2.     global selected_tuple
3.     index=list1.curselection()[0]
4.     selected_tuple=list1.get(index)
```

```
5.     e1.delete(0,END)
6.     e1.insert(END, selected_tuple[1])
7.     e2.delete(0,END)
8.     e2.insert(END, selected_tuple[2])
9.     e3.delete(0,END)
10.    e3.insert(END, selected_tuple[3])
11.    e4.delete(0,END)
12.    e4.insert(END, selected_tuple[4])
```

Since the listbox is empty, `list1.curselection()` will be an empty list with no items. Trying to access the first item on the list with `[0]` in line 3 will throw an error because there is no first item in the list.

Try to fix that bug. The next lecture contains the solution.

Solution: Fixing a Bug in Our Program:

Solution

```
1. def get_selected_row(event):
2.     try:
3.         global selected_tuple
4.         index=list1.curselection()[0]
5.         selected_tuple=list1.get(index)
6.         e1.delete(0,END)
7.         e1.insert(END, selected_tuple[1])
8.         e2.delete(0,END)
9.         e2.insert(END, selected_tuple[2])
10.        e3.delete(0,END)
11.        e3.insert(END, selected_tuple[3])
12.        e4.delete(0,END)
13.        e4.insert(END, selected_tuple[4])
14.    except IndexError:
15.        pass
```

Explanation

The error was fixed by simply implementing a `try` and `except` block. When the `get_selected_row` function is called, Python will execute the indented block under `try`. If there is an *IndexError*, none of the lines under `try` will be executed; the line under `except` will be executed, which is `pass`. The `pass` statement means "do nothing". Therefore, the function will do nothing when there's an empty listbox.

Creating .exe and .app Executables from the Python Script:

- Our program now works well, and consists of three files: **frontend.py**, **backend.py**, and **books.db**. We can execute the scripts by running frontend.py in Python, but we want to wrap everything up in a standalone .exe file to be more user-friendly.
- This can be done whether it be on Windows, Mac, or Linux.
 - To do this, we run **pip install pyinstaller**.
- With **pyinstaller**, it's incredibly easy to create .exe files from Python programs. To do so, we go to the console/terminal and run:
 - **pyinstaller frontend.py**
 - and that's it. If we leave it like this, it will give us a **.exe** file if we're running it on Windows, a **.app** file on Mac, and together with those files we'll get a lot of other files. This can help us troubleshoot errors.
 - If we just want our single executable .exe (or .app) file, we pass another parameter:
 - **pyinstaller --onefile frontend.py**
 - In this form, we'll still get a terminal or command line displayed in the background of our GUI, but we can prevent that with:
 - **pyinstaller --onefile --windowed frontend.py**
- Double-clicking our new .exe file will cause it to pop up in a window AND will create an empty **books.db** database file.
- In situations where you'd be working with an existing database, you'd want to give the user both the .exe file AND the existing database for the program to connect to.

Section 26: Object-Oriented Programming (OOP):

What is Object-Oriented Programming (OOP)?

- OOP is a way to organize code; one could easily just create a program using definitions, but using OOP helps to organize things.
- It's generally accepted that if you use *more than two* functions in your code, you could benefit from using OOP.
- He opened the video with a screenshot of our **backend.py** program from the previous section. We should organize our callback functions into something called a **class**.
- Our object in this case is our *entire window* (tkinter) with its buttons and widgets. We can put all the attributes that define this object (window) such as the titles of buttons, the labels, and the callback functions, *inside of a class*.
- We can start by putting the callback functions from **backend.py** *inside of a class*, and then we're going to do the same to **frontend.py**.

Using OOP in a Program, Part 1:

- We indented everything in **backend.py** and put it inside of a class:
 - **class Database:**
- We also changed our **connect()** function to an initialization function:
 - **def __init__():**
- We then went into **frontend.py** and replaced “import backend” with:
 - **from backend import Database**
 - then we added **database = Database()** below that.
 - We also had to go through frontend.py and replace all instances of “backend” with “**database**”, such as “**backend.insert()**” with “**database.insert()**”.
- Running the program as-is produces an error, saying “**__init__()** takes 0 positional arguments but 1 was given”. To fix this, we need to perform a trick in **backend.py**’s **__init__()**:
 - **__init__(self):**
 - So we can name our database in the **frontend.py**, we also passed into **backend.py**:
 - **__init__(self, db):**

```
def __init__(self, db): < < <
    conn = sqlite3.connect(db) < < <
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS book (id INTEGER PRIMARY KEY,
title TEXT, author TEXT, year INTEGER, isbn INTEGER)")
    conn.commit()
    conn.close()
```

- Now in our **frontend.py**, we can name the database:

```
from tkinter import *
from backend import Database

database = Database("books.db") < < <
```

- However, since we reference “books.db” in several places in **backend.py**, we need to add the **self** parameter to our functions in order to get our program to work.
-
- However, OOP is about more than just putting functions inside of classes, and so far we’re not fully leveraging the advantages of using classes. We can button these up more nicely; each function has some repeated actions, so we can leverage **inheritability**.

Using OOP in a Program, Part 2:

- Since we want to keep the connection open while the **class Database** is in use, we can delete the **conn.close()** line in our **__init__** function.
- Since we're staying connected to the database through our **__init__** function, we can also delete the line **conn = sqlite3.connect("books.db")** in all of our other functions.
- We can also delete our cursor object line **cur = conn.cursor()**, because we're creating that in our **__init__** function.
- However, we're still referencing the **cur** and **conn** variables in various functions, to **.execute()**, **.commit()**, and **.close()**. The **cur** and **conn** variables currently exist now only as local variables within the **__init__** function.
 - In order to fix this, we need to add "self." in front of several things, i.e. "**self.cur**" and "**self.conn**" in various places in the code.
 - Adding "self." to the beginning of a *local variable* like this converts it into a "**attribute**" of the class object.

```
def __init__(self, db):
    self.conn = sqlite3.connect(db) ← ← ←
    self.cur = self.conn.cursor() ← ← ←
    self.cur.execute("CREATE TABLE IF NOT EXISTS book (id INTEGER PRIMARY
KEY, title TEXT, author TEXT, year INTEGER, isbn INTEGER)") ← ← ←
    self.conn.commit() ← ← ←
```

- We can also delete the **conn.close()** lines in all of our functions. We'll deal with closing the database connection later. Deleting these lines prevents a **connection error** from happening when we use one of the functions and it closes the connection prematurely.
- Now we're going to add a **close()** method as a function of its own. There's another special method we can add to our class to do this: **__del__(self)**. This method will be called when the **frontend.py** script **exits**, or when we exit out by clicking "Close" or the "X" corner button.

```
def __del__(self):
    self.conn.close()
```

- He noted that there's no standard way to write objects like this. We could also include a "**Commit**" button to our GUI and a "**commit()**" method to our backend, so that we can store all of the **self.commit()** commands in one place.
- He also noted that **__del__** isn't always used, and that you can usually get away with just using an **__init__** function at the beginning. But in this case, it's useful since you always want to close the connection at the end of using the program.
- He also noted that we can change our **Frontend** to be **Object-Oriented** with **classes** as well.

Frontend with Object-Oriented Changes:

```
from tkinter import *
from backend import Database

database = Database("books.db") ← ← ←

# Creates function to get information from selecting items in listbox
# Also populates entry fields with selected listbox item's information
def get_selected_row(event):
    try:
        global selected_tuple
        index = list1.curselection()[0]
        selected_tuple = list1.get(index)
        # Inserts tuple index items into desired entry fields
        e1.delete(0, END)
        e1.insert(END, selected_tuple[1])
        e2.delete(0, END)
        e2.insert(END, selected_tuple[2])
        e3.delete(0, END)
        e3.insert(END, selected_tuple[3])
        e4.delete(0, END)
        e4.insert(END, selected_tuple[4])
    except:
        pass

# Wrapper functions for buttons:
def view_command():
    list1.delete(0, END)
    for row in database.view():
        list1.insert(END, row)

def search_command():
    list1.delete(0, END)
    for row in database.search(title_text.get(), author_text.get(),
year_text.get(), isbn_text.get()):
        list1.insert(END, row)

def add_command():
    database.insert(title_text.get(), author_text.get(), year_text.get(),
isbn_text.get())
    list1.delete(0, END)
```

```

list1.insert(END, (title_text.get(), author_text.get(), year_text.get(),
isbn_text.get()))

def delete_command():
    database.delete(selected_tuple[0])
    list1.delete(0, END)
    for row in database.view():
        list1.insert(END, row)

def update_command():
    database.update(selected_tuple[0], title_text.get(), author_text.get(),
year_text.get(), isbn_text.get())
    list1.delete(0, END)
    for row in database.view():
        list1.insert(END, row)

window = Tk()

window.wm_title("BookStore")

l1 = Label(window, text="Title")
l1.grid(row=0, column=0)

l2 = Label(window, text="Author")
l2.grid(row=0, column=2)

l3 = Label(window, text="Year")
l3.grid(row=1, column=0)

l4 = Label(window, text="ISBN")
l4.grid(row=1, column=2)

title_text = StringVar()
e1=Entry(window, textvariable=title_text)
e1.grid(row=0, column=1)

author_text = StringVar()
e2=Entry(window, textvariable=author_text)
e2.grid(row=0, column=3)

year_text = StringVar()
e3=Entry(window, textvariable=year_text)

```

```

e3.grid(row=1, column=1)

isbn_text = StringVar()
e4=Entry(window, textvariable=isbn_text)
e4.grid(row=1, column=3)

list1 = Listbox(window, height=6, width=35)
list1.grid(row=2, column=0, rowspan=6, columnspan=2)

sb1 = Scrollbar(window)
sb1.grid(row=2, column=2, rowspan=6)

list1.configure(yscrollcommand=sb1.set)
sb1.configure(command=list1.yview)

list1.bind('<<ListboxSelect>>', get_selected_row)

b1 = Button(window, text="View All", width=12, command=view_command)
b1.grid(row=2, column=3)

b2 = Button(window, text="Search Entry", width=12, command=search_command)
b2.grid(row=3, column=3)

b3 = Button(window, text="Add Entry", width=12, command=add_command)
b3.grid(row=4, column=3)

b4 = Button(window, text="Update Selected", width=12, command=update_command)
b4.grid(row=5, column=3)

b5 = Button(window, text="Delete Selected", width=12, command=delete_command)
b5.grid(row=6, column=3)

b6 = Button(window, text="Close", width=12, command=window.destroy)
b6.grid(row=7, column=3)

window.mainloop()

```

Backend with Object-Oriented Changes:

```
import sqlite3

class Database:
    def __init__(self, db):
        self.conn = sqlite3.connect(db)
        self.cur = self.conn.cursor()
        self.cur.execute("CREATE TABLE IF NOT EXISTS book (id INTEGER PRIMARY KEY, title TEXT, author TEXT, year INTEGER, isbn INTEGER)")
        self.conn.commit()

    def insert(self, title, author, year, isbn):
        self.cur.execute("INSERT INTO book VALUES (NULL, ?, ?, ?, ?)", (title, author, year, isbn))
        self.conn.commit()

    def view(self):
        self.cur.execute("SELECT * FROM book")
        rows = self.cur.fetchall()
        return rows

    def search(self, title="", author="", year="", isbn ""):
        self.cur.execute("SELECT * FROM book WHERE title=? OR author=? OR year=? OR isbn=?", (title, author, year, isbn))
        rows = self.cur.fetchall()
        return rows

    def delete(self, id):
        self.cur.execute("DELETE FROM book WHERE id=?", (id,))
        self.conn.commit()

    def update(self, id, title, author, year, isbn):
        self.cur.execute("UPDATE book SET title=?, author=?, year=?, isbn=? WHERE id=?", (title, author, year, isbn, id))
        self.conn.commit()

    def __del__(self):
        self.conn.close()
```

Creating a Bank Account Class:

- We're going to create a Bank Account app, **acc.py**, and a balance storage file, **balance.txt**.
- For simplicity, we'll be working on only one account.
- We start by creating a **class Account**, then run our usual **__init__(self)** function at the beginning. We're also inputting into our **balance.txt** file an initial value of **1000**.
- To create an **Account** object, we first want to pull the initial value from **balance.txt** into it.
- We read the data from **balance.txt** into a *temporary variable* “**file**”, and then set **self.balance = file.read()**, where **self**. Is the object and **balance** is the *instance variable*:

```
class Account:  
    def __init__(self, filepath):  
        with open(filepath, 'r') as file:  
            self.balance = int(file.read())
```

- Now that we have our **__init__** function, but before we create a *withdraw* or *deposit* function, he says that he likes to start by **calling** the class to see if it works:

```
class Account:  
    def __init__(self, filepath):  
        with open(filepath, 'r') as file:  
            self.balance = int(file.read())  
  
account = Account("balance.txt") ← ← ←  
print(account)
```

- Running this returns “**<__main__.Account object at 0x0000022A761A4400>**” to the terminal, showing the memory storage location of the Account object.
 - **__main__** is the name assigned to the *module* by Python automatically.
 - If we open an *interactive shell* and run **from account import acc**, then the output gets renamed to: “**<account.acc.Account object at {location}>**”
 - Didn't work on my end. Might be because I have my file structure set up or named differently.
 - In his version, “**acc**” is the name of the program (without its extension) and “**account**” is the name of the **package**, which I think means the fact that he put all of this into a folder named “**account**” instead of “**Bank_Account_Program**”. I'll try changing my folder name and see what happens.
 - Nope, didn't work.
 - Short version: “**account**” = *package*, “**acc**” = *module*, “**Account**” = *class*.
-
-
-
-
-

- So far, we've printed out the **object namespace**, as opposed to the **balance**. To print out the balance, we do this:

```
class Account:
    def __init__(self, filepath):
        with open(filepath, 'r') as file:
            self.balance = int(file.read())

account = Account("balance.txt")
print(account.balance) ← ← ←
```

- This returns “**1000**” to the terminal.
-
- Now we want to write our **withdraw** function, which will take *self*, an **amount** variable, and it will subtract that amount from *self.balance*:

```
def withdraw(self, amount):
    self.balance = self.balance - amount
```

- We also would want to update our balance in our **balance.txt** file, and we could do that again with the **open()** function like we did it `__init__`.
 - However, it might be more constructive to create a **separate method** focused entirely on updating our **balance.txt** file.
- We create a similar **deposit** function:

```
def deposit(self, amount):
    self.balance = self.balance + amount
```

- We then run a test where we withdraw **100** from our account and print out the *new balance*:

```
print(account.balance)
account.withdraw(100)
print(account.balance)
```

- This prints out **1000** and then **900** below that. However, if you run it again, you'll notice that a *new object instance* is being created, and the same two numbers are printed out.
 - You don't have 900 as a current balance in the beginning, you still have 1000. This is because the **balance.txt** file that's being read from isn't being updated.
 - We need to *write* the changes to the file if we want to keep these changes.
-
- We'll create a **commit** function:

```
def commit(self):
    with open(filepath)
```

- You'll notice that we want to use a *filepath* similar to in our `__init__` function, but that variable **filepath** is local to the `__init__` function and can't be seen in this scope.

- One way around this is to pass a new path variable “**path**” into the **commit(self, path)** function. But that’s not the most elegant way to do things.
- Instead, we’ll make the **filepath** local variable in `__init__` into an **instance variable**:

```
def __init__(self, filepath):
    self.filepath = filepath < < <
    with open(filepath, 'r') as file:
        self.balance = int(file.read())
```

- Now we can go back to our **commit** function and do this:

```
def commit(self):
    with open(self.filepath, 'w') as file: < < <
        file.write(str(self.balance))
```

- We also need to *call* the **commit** method at the bottom in order to get it to actually update **balance.txt**:

```
account = Account("balance.txt")
print(account.balance)
account.withdraw(100)
print(account.balance)
account.commit() < < <
```

- Now the value updates!

Bank Account, Full Code So Far:

```
class Account:  
    def __init__(self, filepath):  
        self.filepath = filepath  
        with open(filepath, 'r') as file:  
            self.balance = int(file.read())  
  
    def withdraw(self, amount):  
        self.balance = self.balance - amount  
  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
  
    def commit(self):  
        with open(self.filepath, 'w') as file:  
            file.write(str(self.balance))  
  
account = Account("balance.txt")  
print("Old balance: ", account.balance)  
account.withdraw(100)  
print("New balance: ", account.balance)  
account.commit()
```

Creating Classes Through Inheritance:

- **Inheritance:** Inheritance is the process of creating a new class out of a base class. It has all of the properties and methods of the base class, plus some of its own special properties.
- So we've created our **Account** class. Now, what if we want to transfer money from this bank account to another bank account? We would have to add a **transfer** method in the Account class.
- However, in the case he's presenting, you can directly transfer money from your *checking account*, but not your *savings account*, so a **transfer** method would only make sense for a *checking account*.
- So one solution is to create a completely separate class.
- Or,
- We can use **inheritance** to create a new **subclass**, "**Checking**":

```
class Account:  
    def __init__(self, filepath):  
        self.filepath = filepath  
        with open(filepath, 'r') as file:  
            self.balance = int(file.read())  
  
    def withdraw(self, amount):  
        self.balance = self.balance - amount  
  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
  
    def commit(self):  
        with open(self.filepath, 'w') as file:  
            file.write(str(self.balance))  
  
  
class Checking(Account): ← ← ← syntax to inherit attributes  
    def __init__(self, filepath):  
        Account.__init__(self, filepath)  
  
    def transfer(self, amount): ← ← ← special method  
        self.balance = self.balance - amount  
  
  
checking = Checking("balance.txt")  
checking.transfer(110)  
print(checking.balance)  
checking.commit()
```

-

- We can also add unique variables to our subclass: suppose when you transfer money, it comes with a fee.

```

class Account:
    def __init__(self, filepath):
        self.filepath = filepath
        with open(filepath, 'r') as file:
            self.balance = int(file.read())

    def withdraw(self, amount):
        self.balance = self.balance - amount

    def deposit(self, amount):
        self.balance = self.balance + amount

    def commit(self):
        with open(self.filepath, 'w') as file:
            file.write(str(self.balance))

class Checking(Account):
    def __init__(self, filepath, fee): < < <
        Account.__init__(self, filepath)
        self.fee = fee < < <

    def transfer(self, amount):
        self.balance = self.balance - amount - self.fee < < <

checking = Checking("balance.txt", 1) < < <
checking.transfer(100)
print(checking.balance)
checking.commit()

```

OOP Glossary:

- **Class:** Like an object blueprint or prototype.
- **Object Instance:** Created when you call the class.
- **Instance Variable:** Defined inside the methods of the class, and these are accessible by the object instance.
- **Class Variable:** We didn't create one before, but we'll create one for our **Checking** Account now: `type = "checking"`. Class variables are shared by *all the instances* of that class:

```
class Checking(Account):  
  
    type = "checking" ← ← ←  
  
    def __init__(self, filepath, fee):  
        Account.__init__(self, filepath)  
        self.fee = fee
```

- For example, our single *object instance* for our checking account has this **Class variable**. If we create more instances of checking accounts, they will all have that **Class variable**.

```
jacks_checking = Checking("jack.txt", 1)  
jacks_checking.transfer(100)  
print(jacks_checking.balance)  
jacks_checking.commit()  
print(jacks_checking.type) ← ← ←  
  
johns_checking = Checking("john.txt", 1)  
johns_checking.transfer(100)  
print(johns_checking.balance)  
johns_checking.commit()  
print(johns_checking.type) ← ← ←
```

- This will print "checking" for each account.
-
- **Doc Strings:** A Docstring is usually passed right under the **class** keyword:

```
class Checking(Account):  
    """  
    This class generates checking account objects. ← ← ←  
    """  
    type = "checking"
```

- Now if we run `print(johns_checking.__doc__)` at the bottom of our code, it will print the Docstring, "**This class generates checking account objects.**"
 - This is very useful when importing classes from modules. If you can't see the class itself, you can still print out the Docstring to get some information.

- **Data Member:** Class variables or Instance variables (i.e. `type = "checking"` or `self.fee = fee`).
- **Constructor:** The `__init__` function. The constructor constructs the class. A special method.
- **Methods:** We create methods by defining them as functions that we can apply to our object instance.
- **Instantiation:** The process of creating *objects instances*, or *instances of a class*.
- **Inheritance:** When we create a **subclass** out of a **base class**. This subclass shares the methods of the base class, plus it has its own methods that are specific to that subclass.
- **Attributes:** When you access class variables or instance variables, you can say you're accessing the **attributes** of your object.

Section 27: App 6: Mobile App Development: Build a Feel-Good App:

Demo of the Mobile App:

- A single app will be accessible to Android, iOS, Windows, macOS, and Linux. It will have a login and will take information about people's emotional state.

Creating a User Login Page:

Note: Resources for this lecture includes "kivy documentation".

- We start by creating a new file, "**main.py**". This is standard practice for executable apps such as ours, and naming it something else can cause errors later. This is where the logic of the program will be written, in the Python language.
- We also created a *kivy file*, "**design.kv**".
- We also kept a screenshot of the desired GUI up on the screen for reference (this could be a screenshot or even just a sketch on paper of what you want the eventual GUI to look like):



- There are two ways to create a mobile app using Python and Kivy.
 - The first is to write everything in Python.
 - The second is to write a GUI in the .kv file. This is the method we're going to use.
- In **main.py** we write:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.screenmanager import ScreenManager, Screen

Builder.load_file('design.kv') ← ← ← points to our "design.kv" file
```

- The way that Kivy works is that it's hierarchical. For now we're only interested in having one screen for our login, so we type:
 - <LoginScreen>:
 - If we were to also include a sign-up screen, we would type <SignUpScreen>: below that.
 - We also installed a Kivy extension so that VSCode will highlight our Kivy syntax.
 - Our screen has **widgets**. Our first widget is **Label**, but even before that we want to include an *invisible widget*, **GridLayout**, in order to organize our screen. Our GridLayout will have one column and an undefined number of rows by default, but we can set "**cols: 2**" if we want 2 columns. We can also nest GridLayouts:

```
<LoginScreen>:
    GridLayout:
        GridLayout:
            Label:
                text: "User Login"
            TextInput:
                hint_text: "Username"
            TextInput:
                hint_text: "Password"
            Button:
                text: "Login"
        GridLayout:
            cols: 2
            Button:
                text: "Forgot Password?"
            Button:
                text: "Sign Up"
```

- We also want to end our Kivy file with an "invisible widget", <RootWidget>:, which will keep track of all of our app's screens:

```
<RootWidget>:  
    LoginScreen:  
        name: "login_screen"
```

- So far we only have one screen to keep track of.
-
- Now that we have our Kivy file set up, we want to go back to our Python file and create some **empty classes** that have the same names as our *rules* in Kivy.

```
class LoginScreen(Screen):  
    pass  
  
class RootWidget(ScreenManager):  
    pass
```

- We also want to create a **class MainApp(App)**: that references the **build** method from **App** (running **dir(App)** will show this among its methods).
 - This class needs to return **RootWidget()** (the *object*, not the *class*).
- The last thing we need to do in our Python script is to *call* our **MainApp** class. It's a good practice to nest this inside a conditional such that **__name__ == "__main__"**.

```
from kivy.app import App  
from kivy.lang import Builder  
from kivy.uix.screenmanager import ScreenManager, Screen  
  
Builder.load_file('design.kv')  
  
class LoginScreen(Screen):  
    pass  
  
class RootWidget(ScreenManager):  
    pass  
  
class MainApp(App):  
    def build(self):  
        return RootWidget()  
  
if __name__ == "__main__":  
    MainApp().run()
```

- Note that **.run()** is also a method of **App**.
-
- At this point we ran **main.py** to test for errors. We got no errors in our console, but the GUI doesn't look how we want it to look.

- The grid layout isn't behaving how it was meant to, and this is because we didn't specify how many columns it should have. Adding those in makes our Kivy file look like this:

```

<LoginScreen>:
    GridLayout:
        cols: 1 ← ← ←
        GridLayout:
            cols: 1 ← ← ←
            Label:
                text: "User Login"
            TextInput:
                hint_text: "Username"
            TextInput:
                hint_text: "Password"
            Button:
                text: "Login"
        GridLayout:
            cols: 2
            Button:
                text: "Forgot Password?"
            Button:
                text: "Sign Up"

<RootWidget>:
    LoginScreen:
        name: "login_screen"

```

- Running **main.py** now gives us a screen that is set up more or less how we want it. We just need to add some padding and formatting to improve it later.
- He ended the video by pointing out that the above fields such as “**GridLayout**” are Python classes found within Kivy. We could've also, in Python, done “**from kivy.uix.gridlayout import GridLayout**” in order to do everything in Python. However, by using a Kivy file, these classes are imported implicitly.
- In terms of Kivy hierarchy, the highest thing is **App** (represented by **MainApp**), followed by **ScreenManager** (which is represented by **RootWidget**), followed by **Screen** (which is represented by **LoginScreen**).
 - App (MainApp) → ScreenManager (RootWidget) → Screen (LoginScreen)**
- Because of this hierarchy, we also could've nested our entire Kivy file underneath **RootWidget** and it would work the same way:
 - App (MainApp) → ScreenManager (RootWidget) → Screen (LoginScreen) →**
 - **GridLayout → GridLayout → TextInput, Button**
- However, the first way we wrote our Kivy script is easier to keep organized when we add more screens later.

Creating a User Sign-Up Page:

- Currently when we run **main.py**, our GUI looks pretty good, but nothing happens when we press any of the buttons.
- We're going to start by granting functionality to the “**Sign Up**” button.
- We start by going into our **design.kv** code and adding another attribute to our “Sign Up” Button. Below “text:”, we add “**on_press: root.sign_up()**”:

```
Button:  
    text: "Sign Up"  
    on_press: root.sign_up() < < <
```

- Here, “**root**” is a special name that can be used inside of a Kivy file, and “**root**” is the class of the rule widget, “**<LoginScreen>**”, or “**class LoginScreen**” in Python. So “**root**” refers to our Python **LoginScreen** class.
- It’s using the method “**.sign_up()**” from **LoginScreen**, which we haven’t added yet, but we can create it:

```
class LoginScreen(Screen):  
    def sign_up(self): < < <  
        print("Sign up button pressed")
```

- Now when we run our Python file and press the “Sign Up” button, the message “Sign up button pressed” prints out to the terminal. However, this message is just for testing purposes. What we really want to do is **switch the screen**.
- But first, we need to **create the Sign Up screen**. We do this in the Kivy file:

```
<SignUpScreen>: < < <  
    GridLayout:  
        cols: 1  
        Label:  
            text: "Sign Up for a space journey!"  
        TextInput:  
            hint_text: "Username"  
        TextInput:  
            hint_text: "Password"  
    Button:  
        text: "Submit"
```

- Submitting a new entry to this screen will save this information to a **writable** and **readable** **database** that we’ll create. This database will be a **JSON file**.
-
-
-
-

- At this stage, if we save this and run the Python script, nothing is going to happen yet when we click on “Sign Up”. This is because we haven’t linked our new screen to the “Sign Up” button yet; it will still just print the test message.
 - We need to list the **SignUpScreen** in our **<RootWidget>** section:

```
<RootWidget>:
    LoginScreen:
        name: "login_screen"
    SignUpScreen: < < <
        name: "sign_up_screen" < < <
```

- Now in the Python file we modify our **def sign_up(self)**:

```
class LoginScreen(Screen):
    def sign_up(self):
        self.manager.current = "sign_up_screen" < < <
```

- However, when we ran our Python script, we got an error message saying “**Unknown class <SignUpScreen>**”. This is because there isn’t a built-in class named **SignUpScreen** in Kivy, and we haven’t created a class by that name in Python. To fix this, we add a new class of that name to our Python script, inheriting from **Screen**:

```
class RootWidget(ScreenManager):
    pass

class SignUpScreen(Screen): < < <
    pass
```

- And now the new screen pops up when the “Sign Up” button is clicked.
- Explaining “**self.manager.current**”:
 - “**self**” refers to the instance that has been created from this current class “**LoginScreen**”.
 - “**manager**” is a property of “**Screen**” (which “**LoginScreen**” is inheriting from). “**manager**” has a property known as “**current**”.
 - “**current**” attribute will get the name of the screen we want to switch to, which is “**sign_up_screen**”, which is a widget we have created in our Kivy program.

Capturing User Input:

- At this stage, we want to be able to take input for a username and password on the Sign Up page and store it in the database when we click “Submit”.
 - Note: Once we have the functionality of our program working, then we’ll play around with the design of the app to make it look better.
- We’re going to add functionality to the “Submit” button now. We start by adding a line to that button in Kivy:

```
Button:  
    text: "Submit"  
    on_press: root.add_user() < < <
```

- Then we add a function in Python to our **SignUpScreen** class called **add_user**:

```
class SignUpScreen(Screen):  
    def add_user(self): < < <  
        pass
```

- This method is going to get our **username** and **password** values and it’s going to store them in a **JSON file**.
- We can access the username using **root.ids.username**:

```
TextInput:  
    id: username < < <  
    hint_text: "Username"  
TextInput:  
    id: password < < <  
    hint_text: "Password"  
Button:  
    text: "Submit"  
    on_press: root.add_user(root.ids.username, root.ids.password) <  
< <
```

- To use this in our Python script, we need to add some parameters to the function:

```
class SignUpScreen(Screen):  
    def add_user(self, uname, pword): < < <  
        print(uname, pword) < < < for testing
```

- However, the test print here just prints out two Kivy TextInput objects at a location in memory rather than the actual text entries. If we want the actual text, we need to use **root.ids.username.text** and **root.ids.password.text**. Now we get the actual text printing out.
- He then broke down the entire **on_press** line:
-
-

- He then broke down the entire `on_press` line:
 - `root` refers to `<SignUpScreen>`.
 - `add_user` is the *method* in our Python `SignUpScreen` class.
 - class `ids` is a property of the `SignUpScreen` object, which is inherited from `Screen`. This actually gives a sort of *dictionary* with all the IDs within the `SignUpScreen` rule. It gives us access to the usernames, the passwords, etc. `text` accesses the actual text from this dictionary.

Full Python Code, so far:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.screenmanager import ScreenManager, Screen

Builder.load_file('design.kv')

class LoginScreen(Screen):
    def sign_up(self):
        self.manager.current = "sign_up_screen"

class RootWidget(ScreenManager):
    pass

class SignUpScreen(Screen):
    def add_user(self, uname, pword): ← ← ←
        print(uname, pword) ← ← ←

class MainApp(App):
    def build(self):
        return RootWidget()

if __name__ == "__main__":
    MainApp().run()
```

Full Kivy Code, so far:

```
<LoginScreen>:
    GridLayout:
        cols: 1
        GridLayout:
            cols: 1
            Label:
                text: "User Login"
            TextInput:
                hint_text: "Username"
            TextInput:
                hint_text: "Password"
            Button:
                text: "Login"
        GridLayout:
            cols: 2
            Button:
                text: "Forgot Password?"
            Button:
                text: "Sign Up"
                on_press: root.sign_up()

<SignUpScreen>:
    GridLayout:
        cols: 1
        Label:
            text: "Sign Up for a space journey!"
        TextInput:
            id: username < < <
            hint_text: "Username"
        TextInput:
            id: password < < <
            hint_text: "Password"
        Button:
            text: "Submit"
            on_press: root.add_user(root.ids.username.text,
root.ids.password.text) < < <

<RootWidget>:
    LoginScreen:
        name: "login_screen"
    SignUpScreen:
        name: "sign_up_screen"
```

Processing User Sign Ups:

- In this lecture, we're going to implement the Sign-Up Screen such that it stores the inputs in a **JSON file**.
- He included his version of **users.json** as a download, but we can also start with an empty one. I chose to download his version.
 - By default this was formatted as a single line.
 - You can reformat it to be more readable by using:
 - **SHIFT + ALT + 'F'** on Windows.
 - **SHIFT + OPTION + 'F'** on Mac.
 - **CTRL + SHIFT + 'I'** on Linux.
 - It now looks like this:

```
{  
    "user1": { ← ← ← key  
        "username": "user1", ← ← ← key-value pair in sub-dictionary  
        "password": "password1",  
        "created": "2020-07-07 00:16:28.293901"  
    }, ← ← ← smaller dictionary is value of key "user1"  
    "user2": {  
        "username": "user2",  
        "password": "password2",  
        "created": "2020-07-07 00:28:07.315781"  
    }  
}
```

- The idea is the app will add new users here when a user presses the “Submit” button on the Sign-Up Page. It acts as a big Python dictionary made up of smaller dictionaries.
- Up top we need to **import json** to our Python file.
- Then we modify our **add_user** method:

```
class SignUpScreen(Screen):  
    def add_user(self, uname, pword):  
        with open("users.json") as file: ← ← ← opens users.json file  
            users = json.load(file) ← ← ← loads json file into Python dict  
        print(users) ← ← ← prints Python dict (for testing)
```

- Running this as-is just prints “users.json” to our console as a Python dictionary. No new information is added at this stage, even though I input “u1” and “p1” into the Sign-Up page and pressed “Submit”.
-
- We can add a new **key** and **value** to this dictionary:

```
def add_user(self, uname, pword):
    with open("users.json") as file:
        users = json.load(file)
    users[uname] = {'username': uname, 'password': pword,
                   'created': datetime.now().strftime("%Y-%m-%d %H:%M:%S")} ← ← ←
    print(users)
```

- This adds new user information to the dictionary AND formats the datetime to be similar the existing entries in the dictionary. However, these additions aren't permanently added yet.
- Now we just need to open "users.json" in **write** mode in order to save the changes:

```
def add_user(self, uname, pword):
    with open("users.json") as file:
        users = json.load(file)

    users[uname] = {'username': uname, 'password': pword,
                   'created': datetime.now().strftime("%Y-%m-%d %H:%M:%S")}
    with open("users.json", 'w') as file: ← ← ←
        json.dump(users, file) ← ← ←
```

- This will create a new **empty JSON file** called "users.json" and will fill it with our dictionary, which includes both the old and new information.
 - Note: The updated JSON file seems to have trouble formatting using **SHIFT + ALT + 'F'** for some reason.

Creating a Sign Up Success Page:

- Now that we can add new users by typing them into the text boxes and clicking “Submit”, let’s create another page to show when “Submit” is pressed.
- First we need to add a line to the end of our **add_user** method in Python:

```
def add_user(self, uname, pword):  
    with open("users.json") as file:  
        users = json.load(file)  
  
        users[uname] = {'username': uname, 'password': pword,  
                       'created': datetime.now().strftime("%Y-%m-%d %H:%M:%S")}  
    with open("users.json", 'w') as file:  
        json.dump(users, file)  
    self.manager.current = "sign_up_screen_success" ← ← ←
```

- This will point to a screen with the same name that we’ll create in our Kivy file:

```
<RootWidget>:  
    LoginScreen:  
        name: "login_screen"  
    SignUpScreen:  
        name: "sign_up_screen"  
    SignUpScreenSuccess: ← ← ←  
        name: "sign_up_screen_success" ← ← ←
```

- Now we just need to create/format the actual screen up above that:

```
<SignUpScreenSuccess>  
    GridLayout:  
        cols: 1  
        Label:  
            text: "Sign up successful!"  
        Button:  
            text: "Login Page"
```

- Running as-is lead to a “FactoryException”. We forgot to create a **class** in our Python program for this new screen:

```
class SignUpScreenSuccess(Screen):  
    pass
```

- Now when we run the script and sign up as “u3” and “p3”, we successfully get to the “Sign up success” page. However, clicking the “Login Page” button doesn’t do anything yet, as we haven’t added an **on_press** line to it.
- As an exercise, he had us try and figure out what code to add so that this “**Login Page**” button takes us back to the main “User Login” page. I did that with the following:

- In Python, I added a **log_in** method that changes screens and that Kivy can reference:

```
class SignUpScreenSuccess(Screen):  
    def log_in(self): < < <  
        self.manager.current = "login_screen" < < <
```

- And then in Kivy, I added an **on_press** line referencing the **log_in** method in Python:

```
<SignUpScreenSuccess>  
    GridLayout:  
        cols: 1  
        Label:  
            text: "Sign up successful!"  
        Button:  
            text: "Login Page"  
            on_press: root.log_in() < < <
```

- And it worked!

Switching Between Pages:

- He started by implementing the “Login Page” button in `<SignUpScreenSuccess>` just like he had us do as an exercise at the end of the last section.
- However, he used the name `go_to_login` for his method, so I’ll change mine to be consistent in case it would’ve otherwise become an issue in future lectures.
- Python:

```
class SignUpScreenSuccess(Screen):  
    def go_to_login(self): ← ← ←  
        self.manager.current = "login_screen"
```

- Kivy:

```
<SignUpScreenSuccess>  
    GridLayout:  
        cols: 1  
        Label:  
            text: "Sign up successful!"  
        Button:  
            text: "Login Page"  
            on_press: root.go_to_login() ← ← ←
```

- After implementing this, he pointed out that the *default transition between screens* is called “left”, which means that every time we switch to another screen, it swipes in to the left.
- However, maybe we want to change the transition to “right”.
- We do that with:

```
class SignUpScreenSuccess(Screen):  
    def go_to_login(self):  
        self.manager.transition.direction = "right" ← ← ←  
        self.manager.current = "login_screen"
```

- Now when we click on “Login Page” in the success page, the screen swipes *right*.

Processing User Login Credentials:

- He started the video by showing a version of the program with a lot more functionality, what we're still trying to accomplish.
- We need to add another screen, **LoginScreenSuccess**, in Kivy:

```
<RootWidget>:  
    LoginScreen:  
        name: "login_screen"  
    SignUpScreen:  
        name: "sign_up_screen"  
    SignUpScreenSuccess:  
        name: "sign_up_screen_success"  
    LoginScreenSuccess: < < <  
        name: "login_screen_success" < < <
```

- We also need to format this new screen:

```
<LoginScreenSuccess>:  
    GridLayout:  
        cols: 1  
        Button:  
            text: "Logout"  
            on_press: root.log_out() < < < we still need to create this  
        Label:  
            text: "How do you feel?"  
        TextInput:  
            hint_text: "Things to try: happy, sad, unloved..."  
        Button:  
            text: "Enlighten me"  
        Label:  
            text: ""
```

- Then we need to create a new **class** for this screen in Python and give it **logout** functionality:

```
class LoginScreenSuccess(Screen):  
    def log_out(self):  
        self.manager.transition.direction = "right"  
        self.manager.current = "login_screen"
```

- However, at this stage we still haven't implemented the **Login button** in our main Login Page. To do that, we need to go back into Kivy and give that button an **on_press** line in the Login Screen section:

```
Button:  
    text: "Login"  
    on_press: root.login() < < <
```

- We also need to add a similar method as our `add_user` method from our `SignUpScreen` class to our `LoginScreen` class. It will check the input credentials against the JSON file.
- Adding to the above Kivy section:

```
TextInput:  
    id: username < < <  
    hint_text: "Username"  
TextInput:  
    id: password < < <  
    hint_text: "Password"  
Button:  
    text: "Login"  
    on_press: root.login(root.ids.username.text,  
root.ids.password.text) < < <
```

- This will get the input values and pass them to the Python method.
- We need the program to check login credentials against the JSON file, and if they're correct, we want to switch the screen to `LoginSuccessScreen`:

```
class LoginScreen(Screen):  
    def sign_up(self):  
        self.manager.current = "sign_up_screen"  
  
    def login(self, uname, pword):  
        with open("users.json") as file: < < <  
            users = json.load(file)  
        if uname in users and users[uname]['password'] == pword: < < <  
            self.manager.current = "login_screen_success" < < <
```

- We still need to add an “else” conditional, but we’re going to run it here to see if it needs any debugging.
 - So far, so good. I get to the `LoginSuccessScreen` (though nothing here except for the Logout Button works at the moment).
- At the moment, if we input any incorrect data, nothing happens. So we’re going to add a label just under the Login button, with empty text for now. This adds a space under the “Login” button. This will give us space for our “else” conditional.
- We also need to add an `id` to this label, `login_wrong`:

```
Button:  
    text: "Login"  
    on_press: root.login(root.ids.username.text,  
root.ids.password.text)  
Label:  
    id: login_wrong ← ← ←  
    text: "" ← ← ←
```

- We can now point to this label in our “else” conditional:

```
def login(self, uname, pword):  
    with open("users.json") as file:  
        users = json.load(file)  
    if uname in users and users[uname]['password'] == pword:  
        self.manager.current = "login_screen_success"  
    else:  
        self.ids.login_wrong.text = "Wrong username or password!" ← ← ←
```

- And now this label pops up in that empty space! And that’s how we access elements in the Kivy file.

Displaying Output to the User:

Note: Resources for this lecture include a zip file filled with three text files we need for this lecture: Files.zip. They are titled ‘happy’, ‘sad’, and ‘unloved’.

- Download the zip file, ‘**Files.zip**’. Then extract the files ‘**happy.txt**’, ‘**sad.txt**’, and ‘**unloved.txt**’, and place them in a folder called ‘**quotes**’ in the same directory as **main.py**.
- What we’re going to do in this lecture is, in the program that we have we’re going to write one of the three supported feelings (the ‘**hint_text**’ of “Things to try: happy, sad, unloved...”) and set it such that if we write “**happy**” then it will show one of the quotes from the “**happy.txt**” file, etc.
 - If the user enters an unsupported feeling, the output will be “Try another feeling”.
- First off, we’re going to locate the “**Enlighten me**” button in our **<LoginScreenSuccess>** and add an **on_press** line:

```
Button:  
    text: "Enlighten me"  
    on_press: root.get_quote() ← ← ←
```

- Now this **get_quote()** method needs to be added to class **LoginScreenSuccess** so that **root** can access it. However, we also need to add an id, ‘**feeling**’ to the **TextInput** and we need to run the method on that:

```
TextInput:  
    id: feeling ← ← ←  
    hint_text: "Things to try: happy, sad, unloved..."  
Button:  
    text: "Enlighten me"  
    on_press: root.get_quote(root.ids.feeling.text) ← ← ←
```

- Now we can pass this to our Python method:

```
def get_quote(self, feels):  
    print(feels)
```

- We’re going to start by printing “feels” just as a test. So far, so good.
 - Note: He (and I) added a username-password combination that was just “” and “” (empty strings), just to speed things up for testing. Now you just have to click buttons.
- Now that that works, we may want to convert any input feeling into all lowercase, to correspond to our .txt files. We also want to **import glob** up top, because we’re going to use it:

```
def get_quote(self, feel):  
    feel = feel.lower() ← ← ←  
    available_feelings = glob.glob("quotes/*txt") ← ← ← creates a list  
    print(available_feelings)
```

- When we run this, a list gets printed out to the console that includes all the available files, all of the available feelings.

- Now we want to extract only the stuff from the file matching the feeling. A way to do that is by using the **Path object** of the **pathlib** library.
 - Using Path, if you give Path a filepath and then extract the **.stem** from that, it gives you the name of the file without the extension or the higher part of the filepath.
 - Using **.name** would give you the full name of the file, including the extension, i.e. “sad.txt”.
 - We want to apply this to all the file names with a **list comprehension**.
- In Python, we’re adding:

```
def get_quote(self, feel):
    feel = feel.lower()
    available_feelings = glob.glob("quotes/*txt")

    available_feelings = [Path(filename).stem for filename in
                          available_feelings] ← ← ←
    print(available_feelings) ← ← ← for testing
```

- Note: Don’t forget **from pathlib import Path** up top to get this to work.
- Running this gives us a list of **['happy', 'sad', 'unloved']**.
- Having trouble after adding the conditional.
 - After looking up the error on Stack Overflow, managed to fix it by adding some more attributes to the **open()** method:
 - encoding='utf-8', errors='ignore'**:

```
available_feelings = [Path(filename).stem for filename in
                      available_feelings]

if feel in available_feelings:
    with open(f"quotes/{feel}.txt", encoding='utf-8', errors='ignore')
as file: ← ← ←
        quotes = file.readlines()
        self.ids.quote.text = random.choice(quotes)
```

- Now I just have the “AttributeError: ‘super’ object has no attribute” error like the instructor currently does in the video. He explained that this is a common error if the Python code can’t find the ID in Kivy, in this case: “id: quote”. He said that sometimes this is caused when one doesn’t save their Kivy file after making a change.
- Now that it’s working, clicking “**Enlighten me**” more than once will display new quotes with every press.
- Sometimes you’ll get a long quote that won’t fit into the label. In the future, we’re going to add a scrollable area so you can scroll to read the entire quote.
- Now we just need to **stylize the app**.

Full Python Program, so far:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.screenmanager import ScreenManager, Screen
import json, glob, random ← ← ←
from datetime import datetime
from pathlib import Path ← ← ←

Builder.load_file('design.kv')

class LoginScreen(Screen):
    def sign_up(self):
        self.manager.current = "sign_up_screen"

    def login(self, uname, pword):
        with open("users.json") as file:
            users = json.load(file)
        if uname in users and users[uname]['password'] == pword:
            self.manager.current = "login_screen_success"
        else:
            self.ids.login_wrong.text = "Wrong username or password!"

class RootWidget(ScreenManager):
    pass

class SignUpScreen(Screen):
    def add_user(self, uname, pword):
        with open("users.json") as file:
            users = json.load(file)

        users[uname] = {'username': uname, 'password': pword,
                       'created': datetime.now().strftime("%Y-%m-%d %H:%M:%S")}
        with open("users.json", 'w') as file:
            json.dump(users, file)
        self.manager.current = "sign_up_screen_success"

class SignUpScreenSuccess(Screen):
    def go_to_login(self):
        self.manager.transition.direction = "right"
        self.manager.current = "login_screen"

class LoginScreenSuccess(Screen):
    def log_out(self):
        self.manager.transition.direction = "right"
```

```
self.manager.current = "login_screen"

def get_quote(self, feel): ← ← ←
    feel = feel.lower() ← ← ←
    available_feelings = glob.glob("quotes/*txt") ← ← ←

    available_feelings = [Path(filename).stem for filename in
                           available_feelings] ← ← ←

    if feel in available_feelings: ← ← ←
        with open(f"quotes/{feel}.txt", encoding='utf-8', errors='ignore') as
            file: ← ← ←
                quotes = file.readlines() ← ← ←
                self.ids.quote.text = random.choice(quotes) ← ← ←

    else:
        self.ids.quote.text = "Try another feeling." ← ← ←

class MainApp(App):
    def build(self):
        return RootWidget()

if __name__ == "__main__":
    MainApp().run()
```

Full Kivy Code, so far:

```
<LoginScreen>:
    GridLayout:
        cols: 1
        GridLayout:
            cols: 1
            Label:
                text: "User Login"
            TextInput:
                id: username
                hint_text: "Username"
            TextInput:
                id: password
                hint_text: "Password"
            Button:
                text: "Login"
                on_press: root.login(root.ids.username.text,
root.ids.password.text)
            Label:
                id: login_wrong
                text: ""
        GridLayout:
            cols: 2
            Button:
                text: "Forgot Password?"
            Button:
                text: "Sign Up"
                on_press: root.sign_up()

<SignUpScreen>:
    GridLayout:
        cols: 1
        Label:
            text: "Sign Up for a space journey!"
        TextInput:
            id: username
            hint_text: "Username"
        TextInput:
            id: password
            hint_text: "Password"
        Button:
            text: "Submit"
            on_press: root.add_user(root.ids.username.text,
root.ids.password.text)
```

```
<SignUpScreenSuccess>
    GridLayout:
        cols: 1
        Label:
            text: "Sign up successful!"
        Button:
            text: "Login Page"
            on_press: root.go_to_login()

<LoginScreenSuccess>:
    GridLayout:
        cols: 1
        Button:
            text: "Logout"
            on_press: root.log_out()
        Label:
            text: "How do you feel?"
        TextInput:
            id: feeling ← ← ←
            hint_text: "Things to try: happy, sad, unloved..."
        Button:
            text: "Enlighten me"
            on_press: root.get_quote(root.ids.feeling.text) ← ← ←
        Label:
            id: quote ← ← ←
            text: ""

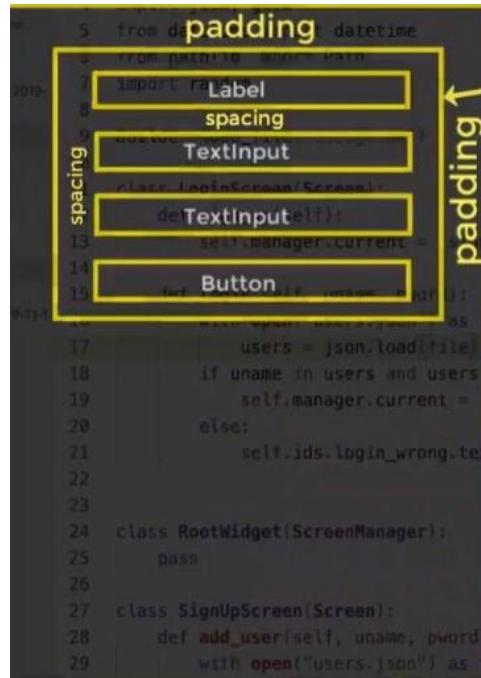
<RootWidget>:
    LoginScreen:
        name: "login_screen"
    SignUpScreen:
        name: "sign_up_screen"
    SignUpScreenSuccess:
        name: "sign_up_screen_success"
    LoginScreenSuccess:
        name: "login_screen_success"
```

Stylizing the Login Page:

- Now that we have our app's functionality working (except for the "Forgot Password?" button), we can work on styling our app to look nicer.
- Since it's easy to drag-resize our app window (and it automatically resizes), we can make it the size and shape of a phone screen. This is nice since it gives us an idea of what the finished mobile app will look like.
-
- We're not doing any work in the Python file this time, just in the Kivy file.
- **Padding & Spacing:** The first thing we want to add is some padding and some spacing. Currently, all of our buttons and text boxes are stretched out to the edges of the screen. To change this, we add some fields to our **<LoginScreen> GridLayout**:

```
<LoginScreen>:  
    GridLayout:  
        cols: 1  
        padding: 15, 15 ← ← ←  
        spacing: 20, 20 ← ← ←
```

- “**Padding**” refers to the border *around* the GridLayout, so it adds padding between the grid and the outermost border of the window.
- “**Spacing**” refers to the area *between* widgets such as buttons, text boxes, etc.



-
-

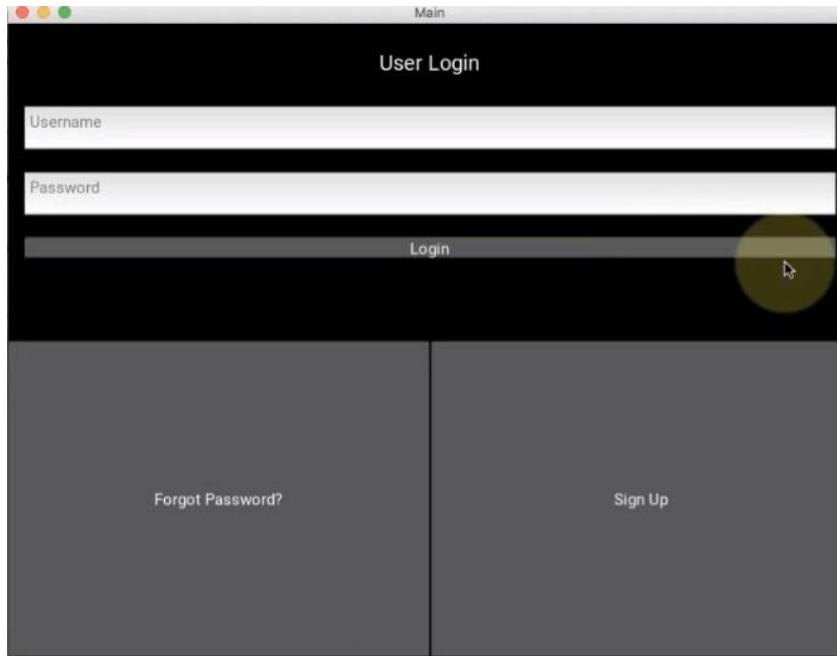
- We can also do things like increase the ***font size*** of “User Login”.

```
Label:
    text: "User Login"
    font_size: '20sp' ← ← ←
```

- This ***font_size*** field needs to be a *string*. The ‘sp’ portion of it stands for “*space-independent pixels*”.
- **Button Size & Position:** We can also **change the size** of the **Login button**. We do this with an attribute called “***size_hint***”. By default, the area is proportionally divided between the widgets. For example, if you have two GridLayout widgets as children of a main one, each will get 50% of the area. The same happen for sub-widgets within each of those GridLayout widgets.

```
Button:
    text: "Login"
    on_press: root.login(root.ids.username.text,
root.ids.password.text)
    size_hint: 0.3, 0.5 ← ← ←
```

- However, while this shrinks the button in the Y-direction, the button still fills up all the space in the X-direction:

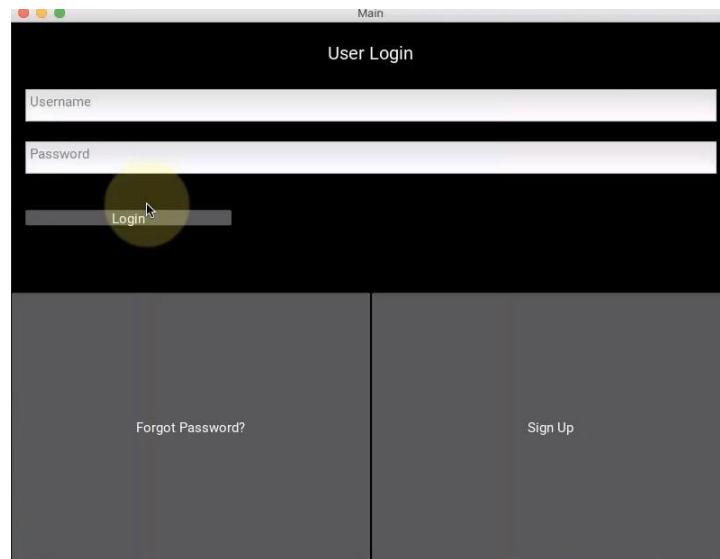


- In order to fix this, we actually need to indent the entire Button widget for “Login” and put it inside of a **RelativeLayout** widget:
-
-
-

- In order to fix this, we actually need to indent the entire Button widget for “Login” and put it inside of a **RelativeLayout** widget:

```
RelativeLayout: < < <
    Button:
        text: "Login"
        on_press: root.login(root.ids.username.text,
root.ids.password.text)
        size_hint: 0.3, 0.5
```

- This results in this:



- The “0.3” value gives it 1/3 of the available space in the RelativeLayout.
- In order to get this button in the center of the RelativeLayout, we need to use “**pos_hint**”:

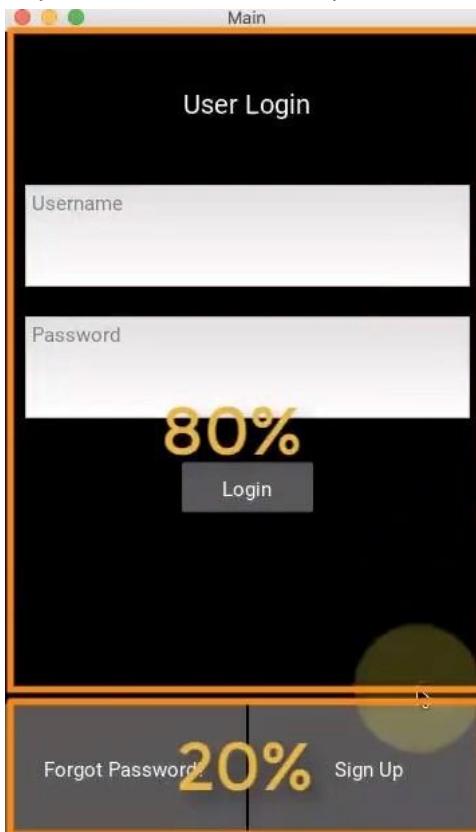
```
RelativeLayout:
    Button:
        text: "Login"
        on_press: root.login(root.ids.username.text,
root.ids.password.text)
        size_hint: 0.3, 0.5
        pos_hint: {'center_x': 0.5, 'center_y': 0.6} < < <
```

- I ended up playing around with my **size_hint** values a bit more to make it look nicer to me.
-
-
- **Styling Lower Buttons:** Now we still need to stylize the “**Forgot Password**” and “**Sign Up**” buttons down below.
-
-

- Now we still need to stylize the “Forgot Password” and “Sign Up” buttons down below:

```
GridLayout:
    cols: 2
    size_hint: 0.2, 0.2 < < <
    Button:
        text: "Forgot Password?"
    Button:
        text: "Sign Up"
        on_press: root.sign_up()
```

- This gives the lower GridLayout 20% of the overall space:



- We also added some padding and spacing.
-
- [Lower Button Colors & Opacity](#): Now, let's make these two lower buttons look like **links**.
- We'll start by changing the background color to **black** in **RGB** with an opacity of 0:

```
Button:
    text: "Forgot Password?"
    background_color: 1, 1, 1, 0 < < <
```

- Now the button has a black background with white text.
- However, when we click on the button now, we can't visually tell if anything is happening with it.

- To fix this, we add an **opacity** attribute:

```
Button:
    text: "Forgot Password?"
    background_color: 1, 1, 1, 0
    opacity: 1 if self.state == 'normal' else 0.5 ← ← ←
```

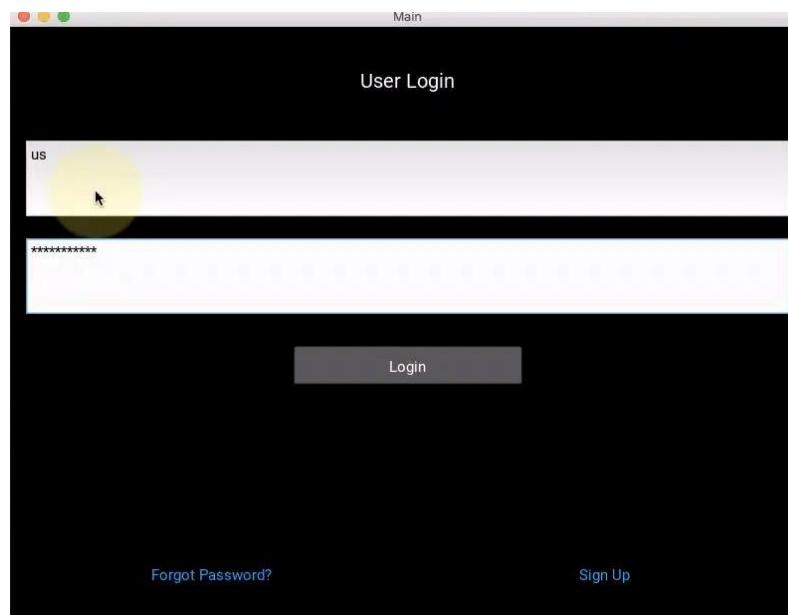
- What this does is, the text is 100% (1) opacity normally, but will go to $\frac{1}{2}$ opacity when clicked.
- Lastly, we change the color of the button (the text, in this case) to *blue*.

```
Button:
    text: "Forgot Password?"
    background_color: 1, 1, 1, 0
    opacity: 1 if self.state == 'normal' else 0.5
    color: 0.1, 0.7, 1, 1 ← ← ←
```

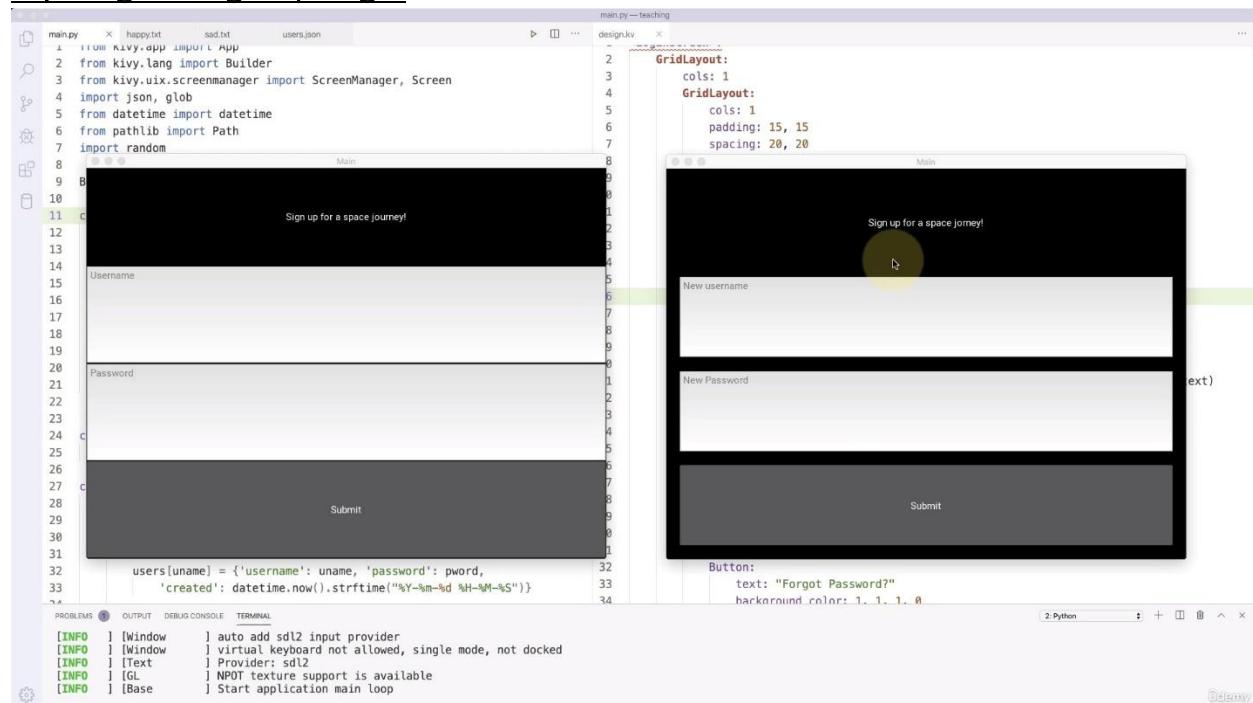
- This gives it 10% of *red*, 70% of *green*, **100%** of *blue*, and 100% opacity.
- We also add all these same attributes (copy/paste) to the “Sign Up” button.
-
- **Password Stars:** One other thing we might want to do in the Login page is this: currently when we type in a password it’s visible in the text box, but it would be better if it **showed stars** in place of characters. We do this by adding the **password** attribute and setting it to **True**:

```
TextInput:
    id: password
    password: True ← ← ←
    hint_text: "Password"
```

- This sets all input characters in the Password text-box to appear as stars:



Styling the Sign Up Page:



```
main.py
1  happy.txt    sad.txt    users.json
2  from kivy.lang import Builder
3  from kivy.uix.screenmanager import ScreenManager, Screen
4  import json, glob
5  from datetime import datetime
6  from pathlib import Path
7  import random
8
9  B
10 c
11 c
12     Sign up for a space journey!
13
14 Username
15
16
17
18
19
20 Password
21
22
23
24
25
26
27 c
28     Submit
29
30
31
32     users[uname] = {'username': uname, 'password': pword,
33     'created': datetime.now().strftime("%Y-%m-%d %H-%M-%S")}
34
```

```
design.kv
1
2 GridLayout:
3     cols: 1
4     GridLayout:
5         cols: 1
6         padding: 15, 15
7         spacing: 20, 20
8
9
10 Mains
11
12
13
14
15 New username
16
17
18
19
20 New Password
21
22
23
24
25
26
27
28
29
30
31
32     Button:
33         text: "Forgot Password?"
34         background_color: 1, 1, 1, 0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[INFO] [Window] auto add sdl2 input provider
[INFO] [Window] virtual keyboard not allowed, single mode, not docked
[INFO] [Text] Provider: sdl2
[INFO] [GL] NPOT texture support is available
[INFO] [Base] Start application main loop

Python

Odemy

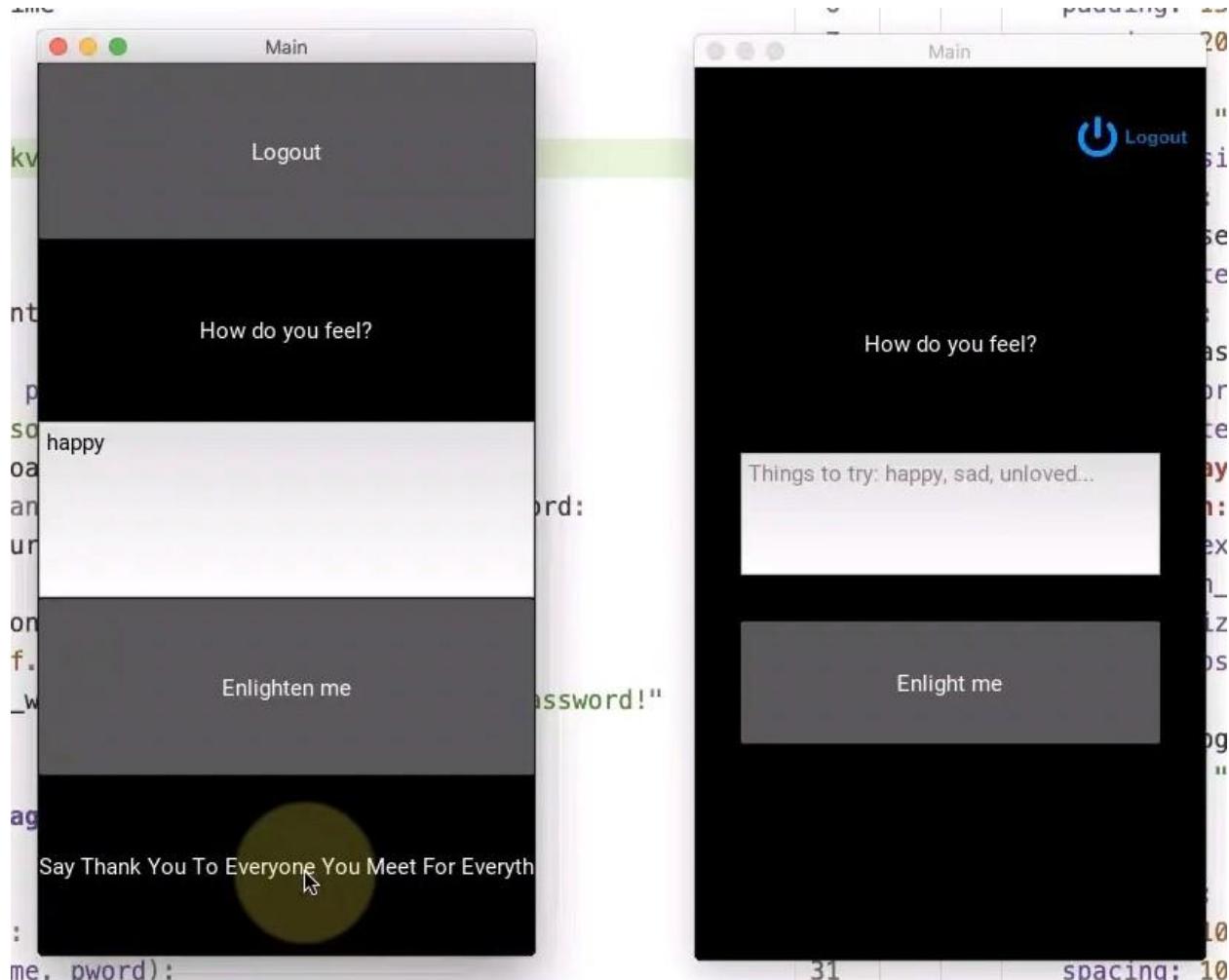
- Now we want to add some padding and spacing to our Sign Up Page.

```
<SignUpScreen>:
    GridLayout:
        cols: 1
        padding: 20, 20 < < <
        spacing: 20, 20 < < <
```

Making the Buttons Interactive:

Note: Included in the resources for this lecture is a file called "**hoverable.py**", which we'll need.

Adding Spacing:



- In this lecture, we want to add some padding, stylize the logout button (and enable it to turn red when hovered over), and add a scroll-bar to our quote output area.
- We added **padding** and **spacing** of 30, 30 each.

Making Hoverable:

- He noted that there isn't a native function for Kivy to implement a hover function, or that there isn't an easy way to do so.
 - Luckily, someone developed a helpful sub-module called **hoverable.py** which we can put in the same directory as our **main.py** program.
- We don't have to change anything in the "hoverable.py" file, we just need to import from it:

```
from hoverable import HoverBehavior < < <
```

- This imports the **HoverBehavior class** from the hoverable.py file.

- Then we create a *new class ImageButton()* in our **main.py** file...

```
class ImageButton(): < < <
```

- ...and we change our “button” callout for said button in **Kivy** to **ImageButton**.

```
ImageButton:
    text: "Logout"
    on_press: root.log_out()
```

- We set our **class ImageButton()** to inherit from **HoverBehavior**, **Image**, **ButtonBehavior**, and we just need to set this to **pass**, because it just gets the three parents working together (*note, see end of lecture for fix*):

```
class ImageButton(HoverBehavior, Image, ButtonBehavior): < < <
    pass < < <
```

- This means we also need to import these **parents** into our program up top:

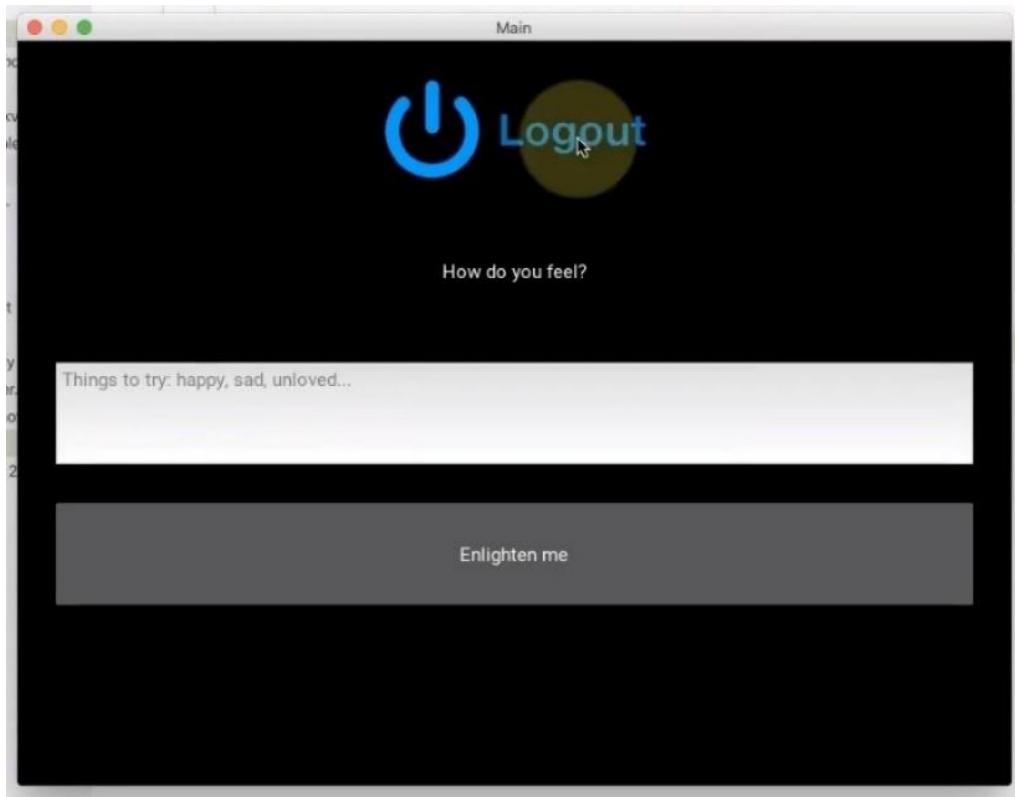
```
from hoverable import HoverBehavior
from kivy.uix.image import Image < < <
from kivy.uix.behaviors import ButtonBehavior < < <
```

Switching to Logout Button Images:

- In **Kivy** where we changed our callout to **ImageButton**, we no longer need the **text: “Logout”** field, because we’ll be using our created image instead, and this image contains the text “Logout”.
- We also will add a **source** field to control the hovering function:

```
ImageButton:
    on_press: root.log_out()
    source: 'logout_hover.png' if self.hovered else
    "logout_nothover.png" < < <
```

- This tests a Boolean condition “hovered” found in the **hoverable.py** sub-module, and it points to our two .png images to be used. It now looks like this:

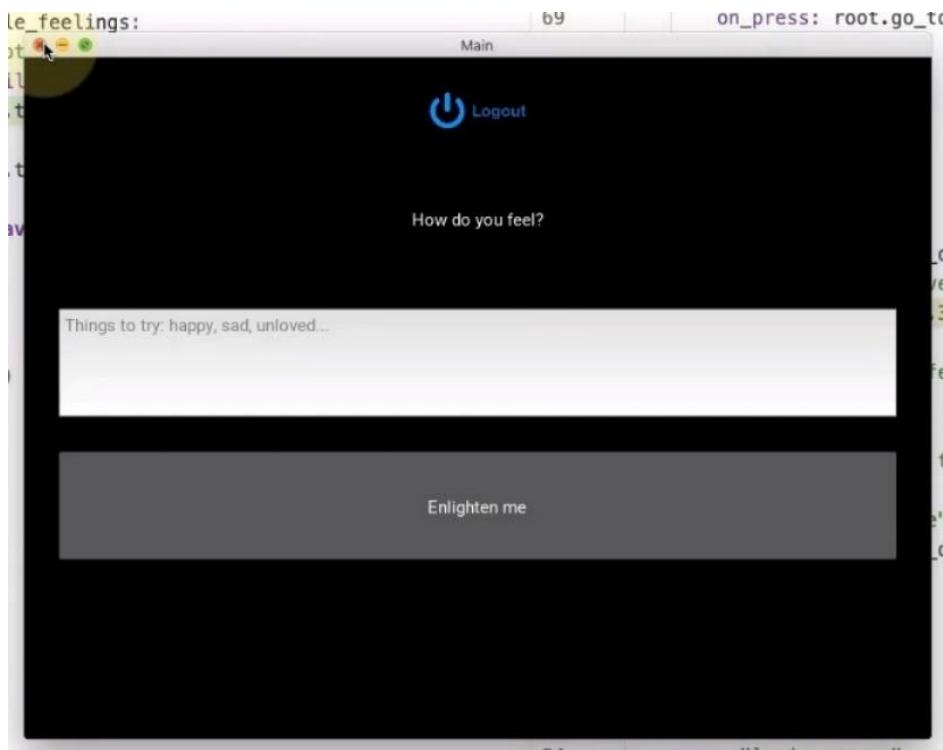


Resizing Logout Button:

- Currently our Logout button is too big. This is because we have 5 widgets total on this page, so each of them is getting 20% of the available area. So we're going to do some size-hinting:

```
ImageButton:  
    on_press: root.log_out()  
    source: 'logout_hover.png' if self.hovered else  
"logout_nothover.png"  
    size_hint: 0.35, 0.35 ← ← ←
```

-
- This makes it look like this:



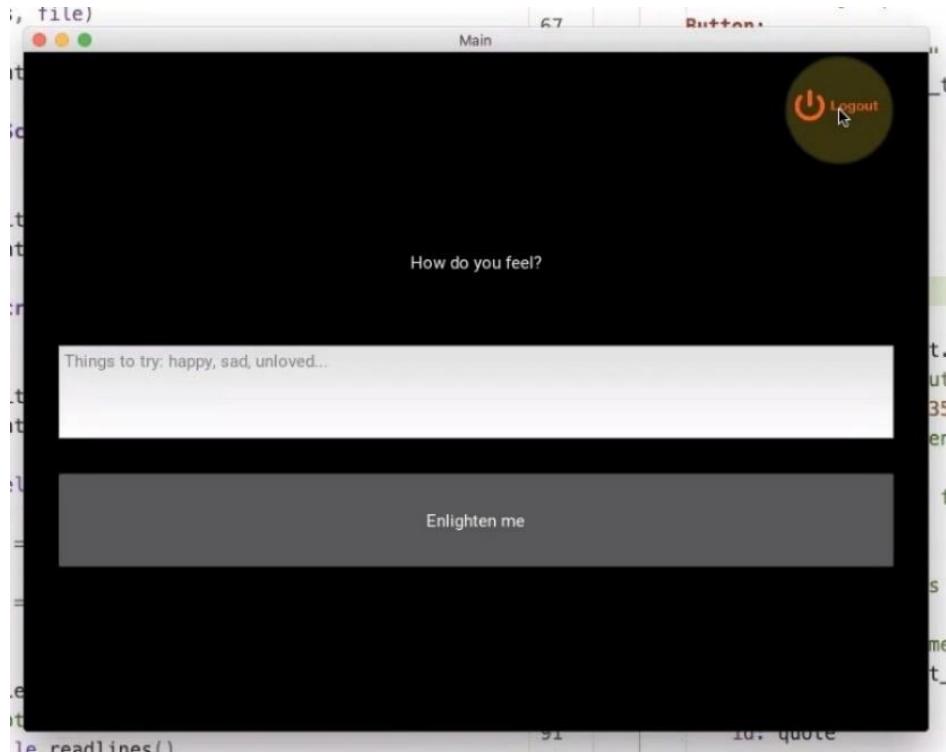
-

Repositioning Logout Button:

- It's still in the center, so if we want to push it to the right, we can use the **pos_hint** attribute. In order to use this correctly, we also have to place our **ImageButton** *inside* a **RelativeLayout**:

```
RelativeLayout: < < <
    ImageButton:
        on_press: root.log_out()
        source: 'logout_hover.png' if self.hovered else
"logout_nothover.png"
        size_hint: 0.35, 0.35
        pos_hint: {'center_x': 0.93, 'center_y': 0.8} < < <
```

- Now our window looks like this:



- However, so far when we click this **Logout** image, *it doesn't work*. It turns out that it's because of how we *ordered* the parents in our **ImageButton** class in Python; **ButtonBehavior** should've been first:

```
class ImageButton(ButtonBehavior, HoverBehavior, Image):
    pass
```

- And now it works. In the next lecture, we'll focus on the scrollable area.

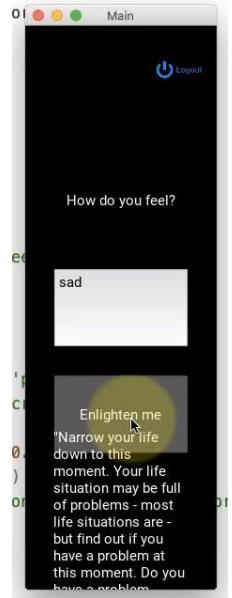
Making a Scrollable Area:

- The display for our quote isn't optimized at this point; a long quote will be cut off. Or, we can wrap the text within the borders of the widget.
- We do that by adding a **text_size** attribute to the Label widget for our quote in .kv:

```
Label:  
    id: quote  
    text: ""  
    text_size: self.width, self.height ← ← ←
```

- Now our quote just barely fits inside of our widget, no matter what size we make our window.
- However, we might want to give our text more room to stretch vertically. To do that, we set self.height to **None** instead:

```
Label:  
    id: quote  
    text: ""  
    text_size: self.width, None ← ← ←
```



- Now the height of the text isn't bound by the widget. However, for thin windows and long quotes, this text can now overlap other widgets →→→
- So how do we fix this? We can *change the height of the label to match the height of the text*:
- We start by changing the Label's **size_hint_y** to **None**, which allows it to ignore the standard 20% of the total area to be **100 pixels high by default**:

```
Label:  
    id: quote  
    text: ""  
    text_size: self.width, None  
    size_hint_y: None ← ← ←
```

- This is the same as adding an attribute **height: 100**. Setting height to 400 almost completely covers up the widgets up above. However, this isn't what we want; we want the height to be dynamic, based on the length of the quote:

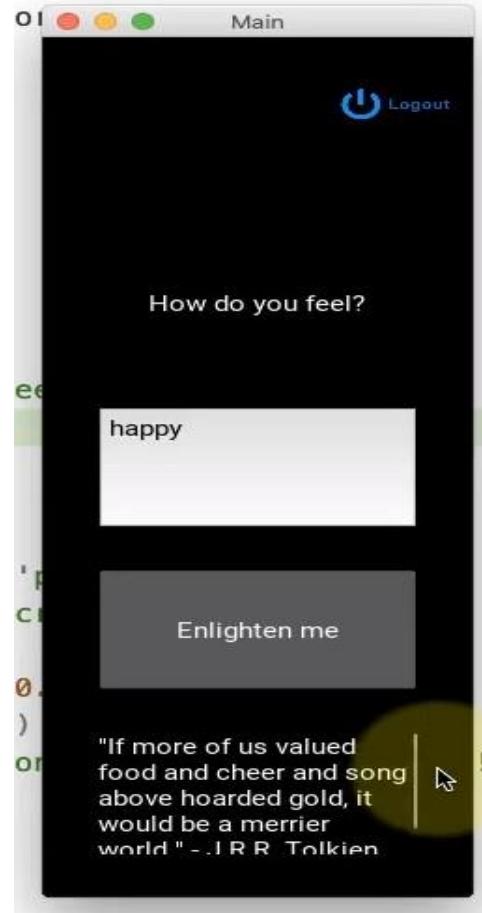
```
Label:  
    id: quote  
    text: ""  
    text_size: self.width, None  
    size_hint_y: None  
    height: self.texture_size[1] ← ← ←
```

- This **texture_size** is a **tuple** containing the **width** and the **height** of the text, so taking the second element of the tuple **[1]** sets this relative to the **height of the text**.
- Our Label is now **dynamic**, changing size based on both the size of the window *and* the length of the quote.

- However, there's still one more thing left to do. We need to add space for a **ScrollView** widget:

```
ScrollView: < < <
    Label:
        id: quote
        text: ""
        text_size: self.width, None
        size_hint_y: None
        height: self.texture_size[1]
```

- This causes the widget size to appear static (other buttons don't move around when quote sizes change). It also adds a scrollbar to the right side of the widget, which reacts to mouse clicks, mouse wheel, and I'm also guessing touchscreen →→→
- When the text is short, there is no scrollbar, and when the text is long, the scrollbar appears.



Section 28: Making an Android APK File from the Kivy App:

Note:

From course instructor:

""This section is a bonus section that covers the instructions on how to convert the Python code to an APK file, which is a file that can be installed in Android devices.

Important note:

As always, you will see the step-by-step instructions in the videos, but please note that creating an APK file involves other third-party tools, and the process also depends on other system configurations that are not related to Python. Therefore, it can sometimes be difficult for me or David to troubleshoot student issues related to the creation of the APK file.

That said, feel free to post a question in the Q&A. Keep in mind that our support might not be as good as it is in the case of issues with Python code."""

Preparing the Environment for Deploying the App to an Android Phone:

- **py → apk**
- Currently, our app has a lot of files associated with it:
 - The **main.py** file that controls the main functions of our app.
 - The **design.kv** file that controls **styling**.
 - The **hoverable.py** file that we use for our “logout” button.
 - Our **.png** files for the “logout” button
 - The **users.json** for storing login information.
 - Our folder full of **quotes**.
- All of these files will be **bundled** into one **APK file**.
- **Buildozer:** In order to make an APK file, we need to use a Python library called **buildozer** which we can install with a pip install command.
- However, the problem is that making APK files on a Windows or Mac machine is very hard. It is best to use a Linux operating system such as **Ubuntu**.
 - We’re going to use a **Virtual Machine** to accomplish this, specifically **Virtual Box** (which I already have from my Linux class).
 - We set up a virtual Ubuntu machine using Virtual Box, but if I create mobile apps in the future I’ll probably just work with my Ubuntu laptop instead.
 - We set it up with **4GB RAM, 20GB storage**, though he did say that students could go with less if their computer didn’t have that many resources available.
- Once it was set up and all updated were downloaded, we continued.
- He noted that setting the VM’s advanced settings to “Bidirectional” doesn’t usually work (I couldn’t get it to work either; never have gotten that to work on a VM). So instead, we’re going to use **Dropbox** to transfer our files over.

Creating an APK File for Android:

- Now that we have our files inside Ubuntu, we're going to use the **Buildozer Python library** to create an **APK file**.
- But before we do that, we need to install some dependencies inside of the Ubuntu operating system. Buildozer needs them in order to create this APK file.
- **First step**, we want to be inside the folder where our files reside, and we want to *right click and choose “Open In Terminal”*.
 - And remember, we have to have our main file named exactly as “**main.py**” because **Buildozer** needs it to be *named as such* in order to work.
 - He used the **rm** Linux command to *remove* a screenshot file that he had. I wonder if the file would've cause an issue. He also has a file called “vsc_here.cmd” in his directory which I don't have, though later in the video he mentioned that we can ignore/delete it because it was just some notes.
- Among these files, there's one called “**kivy-buildozer-installer.sh**” that we can see by running the Linux command **cat kivy-buildozer-installer.sh**.
 - Running this shows a list of *dependencies* as well as instructions on where to properly clone the Buildozer repository.
 - So far, our Linux system doesn't have Kivy installed either.
 - We run **bash kivy-buildozer-installer.sh** and enter our Ubuntu password that we set up; this will download and install the necessary packages to our Ubuntu machine.
 - He noted that this is not a stable process because there are so many dependencies involved, so sometimes we'll get errors during installation or during APK creation.
 - ~~I got a few errors during installation; let's hope they don't become a problem.~~
Had problems.
 - Had to run **vim .bashrc** in the Home/root folder and add the line:
 - **export PATH=\$PATH:~/local/bin/** to the end of .bashrc.
 - After that, I could run “**buildozer init**” (see below).
- We ran **python3 main.py** to open and test out our app. It ran fine just like when we created/tested it on our Windows machine.
- Next we want to do is to create a **Buildozer specification file**: we run **buildozer init** (see above for troubleshooting I had to do).
- This creates a file called “**buildozer.spec**”, which we can open in a text editor by double-clicking. In here, we can change some parameters, such as “title =” to change what our app is called.
 - We changed the name of this to “**title = How Do You Feel?**”.
 - We can also change the package name, but we left it as “**package.name = myapp**”.
 - Most importantly, we went down to the section “**Source files to include**” and we added “**json**” at the end of the list so we can include our **users.json** file. We also need to add **txt** (see error later on), which will cover the file including all of the quotes.
 - Lower in the file, **application versioning** is set to “**0.1**” by default.
 - To “**Application requirements**”, we might have to add **third party Python libraries** that are called out in our **main.py** program in some cases, but in this case all of our libraries are either in the same local directory (such as “**hoverable**”) or they're standard Python libraries (like “**json**”, “**glob**”, “**random**”, “**pathlib/Path**”, and “**datetime**”).
 - We also went down to “**Supported orientation**” and set “**orientation = all**”.

- He also changed “`android.arch = armeabi-v7a`” to “`android.arch = armeabi-v8a`”, but mine seemed to have already been updated to include both.
 - We then saved and closed the file.
- After configuring the `buildozer.spec` file, we want to go back to the terminal and run the command that will *create* the APK file:
 - **`buildozer android debug`**
 - It prompted me to run `sudo apt-get install zlib1g-dev` and `sudo apt-get install git`, and then install a **Java compiler** first. After fumbling around on Google and typing some listed commands for that, it seemed to work. Still running as of typing this.
 - At some point it will prompt you whether or not you want to **accept the license [Y/n]**, and you say yes.
 - I got an error “**error: no acceptable C compiler found in \$PATH**”,
 - so after a quick Google search, I ran the command “`sudo apt-get install build-essential`”.
 - He got an error “**Build failed: Asked to compile for no Archs, so failing**” followed by a second error message that I got as well (“**command failed... pythonforandroid.toolchain create...**” etc).
 - He said that he thought it was because we specified Android **version 8** (see above), so we replace “`android.arch = armeabi-v8a`” with “`arm64-v8a`”.
 - We also may have forgotten to add the `.txt` extension (see `buildozer.spec` notes above)
 - After fixing these, we’re going to try running:
 - **`buildozer android clean debug`**. This will ensure that it won’t mess with previous efforts to install the APK package.

Troubleshooting Process:

- **Note:** I can’t get it to compile. Might have to try this on my actual Ubuntu laptop and see if that helps.
- **Update:** Still had the same problems on my Ubuntu machine. Noticed that part of the error said that “**autoreconf not found**”, so after some searching, ran:
 - `sudo apt-get install autoconf` (“for the ‘autoreconf’ binary”)
 - `sudo apt-get install automake` (“for the ‘aclocal’ binary”; Linux said this was already present)
 - `sudo apt-get install libltdl-dev` (“which defines the ‘LT_SYS_SYMBOL_USCORE’ macro”)
 - `sudo apt-get install libffi-dev` (to handle “`_ctypes not found`” error)
 - One forum post/answer mentioned that one might need to reinstall Python from source to be able to read these changes, else you might run into dependency issues.
- **Update (a week or two later):** Still having issues, getting an error saying “**WARNING: pip is configured with locations that require TLS/SSL, however the ssl module in Python is not available**”.

- Searched, tried `sudo apt-get install libreadline-gplv2-dev lib32readline-dev libncursesw5-dev libssl-dev libsdl2-dev tk-dev libgdbm-dev libc6-dev libbz2-dev` and still nothing.
- ChatGPT suggested running `sudo apt-get install python-openssl`, so let's see if that does the trick. This command worked on my Ubuntu machine, but I had to run `sudo apt-get install python3-openssl` to get it to run on my VM.
- I may have been looking in the wrong direction here. The issue might be Buildozer's inability to fetch URL <https://pypi.org/simple/cython/> as there's also an error saying "**Could not find a version that satisfies the requirement Cython (from versions: none)**".
- A search led me to try running `sudo apt-get install libssl-dev`.
- **It worked!!!!**
- There is now a file in the "**bin**" directory called "**myapp-0.1-arm64-v8a-debug.apk**".
- Now to do it on my VM to test it.

Streamlined Process, from Start to Finish:

- Some things that were necessary before running Buildozer:
 - **Install:**
 - `sudo apt-get install zlib1g-dev`, `sudo apt-get install git` and a **Java compiler** will be necessary to get Buildozer to run.
 - `sudo apt-get install build-essential` to get an acceptable C compiler in \$PATH.
 - `sudo apt-get install autoconf` ("for the 'autoreconf' binary")
 - `sudo apt-get install automake` ("for the 'aclocal' binary"; Linux said this was already present)
 - `sudo apt-get install libltdl-dev` ("which defines the 'LT_SYS_SYMBOL_USCORE' macro")
 - `sudo apt-get install libffi-dev` (to handle "_ctypes not found" error)
 - `sudo apt-get install libssl-dev` (to handle "_ctypes not found" error)
 - The other packages I tried during troubleshooting may or may not be necessary. Try them if needed.
 - A new machine might also need to install **vim** if it doesn't have it.
 - **Configure:**
 - Run `vim .bashrc` in the Home/root folder and add the line: `export PATH=$PATH:~/local/bin/` to the end of the file.
- Next, open a **terminal** from the project directory, where all our necessary files such as **main.py** and **design.kv** are.
- **Buildozer Installation and Init:**
 - Run `bash kivy-buildozer-installer.sh` to install the necessary Buildozer packages on a new machine.
 - Run `python3 main.py` to test your app one last time if you wish.
 - Run `buildozer init` to create the **buildozer.spec** file.

- **Buildozer.spec:**
 - Double-click buildozer.spec to open and edit it.
 - Change “**title = How Do You Feel?**”.
 - Keep “**package.name = myapp**” (this will go into our APK file name).
 - Go down to “**Source files to include**” and add **json, txt**.
 - Keep “**application versioning**” at “**0.1**” (this will go into our APK file name).
 - To “**Application requirements**”, we might have to add **third party Python libraries** that are called out in our **main.py** program in some cases, but in this case all of our libraries are either in the same local directory (such as “hoverable”) or they’re standard Python libraries (like “**json**”, “**glob**”, “**random**”, “**pathlib/Path**”, and “**datetime**”).
 - We also went down to “**Supported orientation**” and set “**orientation = all**”, but I seemed to get errors unless I chose “**orientation = portrait**”. The .spec file in my Ubuntu machine didn’t even include a version for “all”.
 - Change “**android.arch =** ” to be “**android.arch = arm64-v8a**” (this will go into our APK file name).
- **Run Buildozer:**
 - Run either **buildozer android debug** (or **buildozer android clean debug** if you’ve been doing some troubleshooting after the initial try).
 - The **clean** version will ensure that it won’t mess with previous efforts to install the APK package.
 - However, for a super clean install, I ended up running **rm -rf .buildozer** to delete a hidden file at one point. This forced me to go back to using **buildozer android debug** again as though it were the first time.
 - If it runs to completion without any errors, it’ll give you your APK file:
 - “**myapp-0.1-arm64-v8a-debug.apk**”

Installing the APK File for Android:

- He started by saying that this entire process is easier if you have a dedicated or dual-boot Ubuntu machine (way ahead of you, man).
- To transfer the APK file to our mobile device, it would be useful to be able to connect the phone to the Ubuntu machine with a USB cord and transfer the file that way.
- However, he wanted to find a solution that would work for everyone, not just people with dedicated Ubuntu machines. So he's using a file transfer website called "**Gofile**".
- He uploaded the file here, and it gave back a **download link** which we can put into the browser of a mobile device.
 - I wonder if I could just do all this with the version I already saved to Dropbox...
- From here, he showed how to download the file on a phone. His browser was already navigated to the download link, so he hit "download" a few times to get the file.
- Once the APK file downloads, it'll ask if you want to **install the application**. He pressed **Install**, and then Android prompted with a security warning, asking if you trust the download source. He hit "**Install Anyway**".
- He then showed the app in action on his phone, and everything seemed to work perfectly.

Deploying to iOS:

Note from Instructor:

"Unfortunately, converting a kivy app to an iOS app requires you to have access to a Mac computer. The conversion cannot be done on a Windows or a Linux computer.

It's also currently not possible to do this with Python 3. You need to use Python 2.

I will replace these notes with a video once there's a stable method to convert kivy apps to iOS. Meanwhile, you can try the official instructions found here: <https://kivy.org/doc/stable/guide/packaging-ios.html>

or if you prefer video, you can watch this YouTube video from Erik Sandberg: <https://www.youtube.com/watch?v=UAI3PG-qN2k>"

Section 29: Web Scraping with Python & Beautiful Soup:

How Web Scraping Works:

- An example would be seen if you go into **inspection mode** for a website. In his example, he's using a real example website called **example.com**. Going into inspection mode, we can see the HTML tags, the CSS, etc.
- The HTML elements are key for web scraping.
- Let's say we want to extract the text of the **<h1>** tag (in this case, "**<h1>Example Domain</h1>**") out of all the *divisions*, or **<div>** tags.
- So what you do is, you tell Python to go to *all* the **<h1>** tags and extract the text of these tags.
- But first we have to *load* all of the HTML text to Python to read, and we do that with the **request library**. Using **request** we can give Python the URL, and Python will grab all of the HTML text for that URL.
- Once we have this text, we'll use the **beautiful soup** library to extract all of the elements from the text.

Link Update:

IMPORTANT NOTE

In the next video we will scrape a webpage which was previously stored at:

<https://pythonhow.com/example.html>.

Please note that the webpage has now moved to this link: <https://pythonizing.github.io/data/example.html>

So, please use the second link instead of the first one.

Web Scraping Example with Python:

- We're starting with a very simple webpage describing three big cities: London, Paris, and Tokyo. Each city name functions as a **header** for its block, followed by a short descriptive **paragraph**.
- We're going to extract the **city name** for each.
- He's going to be using **Jupyter Notebook** for this one.
- So from a working directory we opened the **Windows Command Line** (or **Powershell**) and entered **Jupyter Notebook**. When it loaded, we chose **New** and **Python3**.
-
- Now, the first thing we want to do is to load the desired source HTML code into Python. So we want to **import requests** (pip install it if you don't already have it), and then we also want to **pip install bs4** to get Beautiful Soup 4 (the latest version at the time the video was created). We then **import** that to Python as well:

```
import requests  
from bs4 import BeautifulSoup
```

- We then want to use **requests** to get and store our webpage in a variable **r**:

```
r=requests.get("https://pythonizing.github.io/data/example.html")
```

- If we run **type(r)** here, it returns that it's a "**requests**" **datatype**, **requests.models.Response**.
- We want to grab the **content** of requests and store it in a variable **c**:

```
r=requests.get("https://pythonizing.github.io/data/example.html")  
c=r.content ↵ ↵ ↵
```

- If we run **type(c)** here, it returns "**bytes**".
- We can also **print(c)**. It doesn't look very nice, but it's a single line version of the HTML script, the source code. If we want to make this "beautiful" and extract the elements and text from it, we want to use Beautiful Soup. It will parse this source code and give us what we want.
- To do this, we create a storage variable **soup** and run **BeautifulSoup** on (**c**). We may also want to pass a second argument to BeautifulSoup to *specify which parser* we want to use:

```
soup=BeautifulSoup(c,"html.parser") ↵ ↵ ↵
```

- This "**html.parser**" is the one we'll almost always use.
- (In **Jupyter Notebook**): If we then run **print(soup.prettify())**, it will print out the *source code* as it appears in *inspection mode*. This is really just for demonstration though.
- Looking through the source code (either through *prettyfify* or *inspection mode*) we see that the cities are all tagged under the **<h2>** tag. Naturally we begin to want to **iterate** through the boxes of the website, or the *divisions* (**<div>** tags).
- We want to perform a *method* called **find_all()** on **soup**:

```
all=soup.find_all("div")
```

- Now, this would find ALL the “div” tags, but we only want to focus on the ones with **class=“cities”** in the source code. So to do this, we pass in a dictionary with a key of “class” and a value of “cities”. We also want to save this to a variable, **all**:

```
all=soup.find_all("div", {"class": "cities"}) ← ← ←
```

- In Jupyter Notebook if we run/print “all”, we’ll see that these divisions have been extracted from the source code:

The screenshot shows a Jupyter Notebook interface with several code cells. Cell 5 contains the HTML source code. Cell 6 imports BeautifulSoup. Cell 8 runs the command `all=soup.find_all("div", {"class": "cities"})`. Cell 9 prints the variable `all`, which is a list containing three `<div>` objects. Each object has its `text` attribute expanded, showing the content of the `<h2>` and `<p>` tags it contains.

```
+ 8% 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 279 280 281 282 283 284 285 286 287 288 289 289 290 291 292 293 294 295 296 297 298 299 299 300 301 302 303 304 305 306 307 308 309 309 310 311 312 313 314 315 316 317 318 319 319 320 321 322 323 324 325 326 327 328 329 329 330 331 332 333 334 335 336 337 338 339 339 340 341 342 343 344 345 346 347 348 349 349 350 351 352 353 354 355 356 357 358 359 359 360 361 362 363 364 365 366 367 368 369 369 370 371 372 373 374 375 376 377 378 379 379 380 381 382 383 384 385 386 387 388 389 389 390 391 392 393 394 395 396 397 398 399 399 400 401 402 403 404 405 406 407 408 409 409 410 411 412 413 414 415 416 417 418 419 419 420 421 422 423 424 425 426 427 428 429 429 430 431 432 433 434 435 436 437 438 439 439 440 441 442 443 444 445 446 447 448 449 449 450 451 452 453 454 455 456 457 458 459 459 460 461 462 463 464 465 466 467 468 469 469 470 471 472 473 474 475 476 477 478 478 479 480 481 482 483 484 485 486 487 488 489 489 490
```

- Each `<div>` is separated by a comma, so we have a ***list*** of three elements.
 - If we only wanted to extract the first element, we could’ve used `.find()` instead of `.find_all()`, or:
 - We can use ***list indexing*** such as `all=soup.find_all("div", {"class": "cities"})[0]`, or just `all[0]`.
- Now we want to narrow it down to the `<h2>` tags. So, we can take our **all** object and apply `.find_all()` again. However, this can only be pointed at specific elements of the list:
 - `all[0].find_all("h2")` → gives “[`<h2>London</h2>`]”, a ***list***.
 - `all[0].find_all("h2")[0]` → gives “`<h2>London</h2>`”, with ***tags***.
 - `all[0].find_all("h2")[0].text` → gives “`London`”, just the text.
- So this extracts just the first city, London. But what about Paris and Tokyo? Or a longer list? We’re going to need to use a ***for loop***:

```
for item in all:
    print(item.find_all("h2")[0].text)
```

- And this lists all three cities!
- We can also pass `p` instead (“`item.find_all("p")[0].text`”) and we’ll get the paragraphs instead.

Section 30: App 7: Web Scraping – Scraping Properties for Sale from the Web:

Demo of the Web Scraping App:

*Note: It appears that we'll be using **Jupyter Notebook** for this section.*

- There's an increase in demand for getting data from the web.
- For example, one might need house data from a **real estate website**, or business data from a site like **yellow pages**, or perhaps information from **Wikipedia**.
- However, websites like this don't offer things like **.csv files**. What you want to do in cases like this is to **scrape data** from these websites.
- In **Python Web Scraping**, you get this data, and you store it in a **pandas dataframe**. From there, you can **export the data as a .csv**.
-
- We'll be working with **Century21 real estate website data**.
- With Python, we'll be performing a **search query** on a location.
- Python will search for data, get the data, and store it in a **dataframe**.
- Lastly, Python will export the data as a **.csv file**.

Note: The New Website URL:

Important note:

Please note that the URL of the website we are scraping in the videos is outdated.
Please use this new URL instead:

<https://pythonizing.github.io/data/real-estate/rock-springs-wy/LCWYROCKSPRINGS/>

Loading the Webpage in Python:

Note: Work for this app primarily done in **Jupyter Notebook**, but I've also made a version on VSCode for consistency in notes.

- We start by *importing requests* and **BeautifulSoup**, then we load in our website (see note on previous page) and set its *content* to a variable **c**:

```
import requests
from bs4 import BeautifulSoup

r=requests.get("https://pythonizing.github.io/data/real-estate/rock-springs-
wy/LCWYROCKSPRINGS/")
c=r.content
```

- We also printed out the content of **c** to make sure everything was working.
- We then set the variable **soup** and ran an **html.parser** on it, then printed **soup.prettify** to see it formatted like an HTML:

```
soup=BeautifulSoup(c,"html.parser")
print(soup.prettify())
```

- This resulted in:

```
In [1]: import requests
        from bs4 import BeautifulSoup

In [3]: r=requests.get("http://www.century21.com/real-estate/rock-springs-wy/LCWYROCKSPRINGS/")
        c=r.content

In [4]: soup=BeautifulSoup(c,"html.parser")
        print(soup.prettify())
<script src="/js/respond-custom.js"></script>
<![endif]-->
<!--[if lte IE 8]>
<link rel="stylesheet" href="/css/ie8fix.css">
<![endif]-->
<!--[if lte IE 7]>
<link rel="stylesheet" href="/css/ie7fix.css">
<![endif]-->
<!--[if gt IE 8]><!-->
<meta content="width=device-width,initial-scale=1,maximum-scale=1" name="viewport">
<link href="/css/responsiveLiquidMap.css" rel="stylesheet">
<script>
var g_responsiveEnabled = true;
</script>
<!--<![endif]-->
<script charset="utf-8" src="http://www.googleadservices.com/pagead/conversion_async.js" type="text/javascript">
</script>
</link>
```

- He searched the text for an address that pops up in the website to make sure everything came in correctly, then *deleted* the **prettyify** since we were just using it to test.
- Next up, we want to *understand the structure* of the webpage. We'll be using the **inspect** feature to do this.

Extracting <div> Elements:

- Next we're going to use **inspect** to learn the structure of the webpage.
- The logic of what we want to do is, we want to iterate over all of the boxes containing property information on the webpage.
- Hovering over and highlighting these boxes will also highlight their section of code in the **inspect** window and vice versa.
- We're looking for elements containing **<div class="propertyRow">** that represent the entire row/box. Inside here we can find data about the property, including the **price**.
- We want to scrape for **price**, **address**, **number of beds**, **number of baths**, and the **square footage** and **lot size** when included. Later we're going to apply these to **pandas dataframes** to send them to a **.csv file**.
- Once we understand the structure of the website, we can begin extracting this data with **.find_all**:

```
all=soup.find_all("div", {"class": "propertyRow"})
```

- He then printed **all** in a Jupyter Notebook line, showing a “ResultSet” similar to a **list**. Printing **len(all)** gave a result of **10**, because we have 10 results per page on the webpage.
- Printing **all[0]** shows the HTML code for the *first element* (or first property on the page). We can run **.find_all** on this just like we ran it on **soup**:

```
all[0].find_all("h4", {"class": "propPrice"})
```

- This finds the **property price (propPrice)** for the *first element*. To find the prices for all, we would iterate through the list.
- Since each property only has *one price*, we can use the **.find** method instead of **.find_all**. We can also use the **.text** method to pare it down to just the price:

```
all[0].find("h4", {"class": "propPrice"}).text
```

```
In [1]: import requests
from bs4 import BeautifulSoup

In [3]: r=requests.get("http://www.century21.com/real-estate/rock-springs-wy/LCWYROCKSPRINGS/")
c=r.content

In [17]: soup=BeautifulSoup(c,"html.parser")

In [6]: all=soup.find_all("div", {"class": "propertyRow"})

In [22]: all[0].find("h4", {"class": "propPrice"}).text[]

Out[22]: '\n          \n          \n          $725,000\n          \n          \n          \n'
```

```
In [ ]:
```

- This gives us some **\n** new-line markers as well though. Luckily, the object that we're creating here is a **string**, which means we can apply *string methods* to remove the new-lines:

```
all[0].find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(" ", "")
```

- This replaces both the **\n** instances and the whitespace with *empty strings*, giving us just the price on its own.

- Now that we have the bones of this process, we can write the code to *iterate through all property entries*.
- Before going further, we decided to organize our code. He taught us a trick to do this in Jupyter Notebook:
 - First, we want to be in **command mode**, so hit **esc** to get to that.
 - Next, we click on the *first cell*, then hit **Shift + J** repeatedly to select subsequent cells.
 - We want to merge all of these into a single cell; we do that with **Shift + M**.

Scraping the Addresses of the Properties:

- He started the lecture off by printing **soup** again, which is our *source code*, and then printed **all**, which is the source code separated at our chosen divider. So far, we've only extracted the *property price* of the first division, the *first house entry* on the webpage. To get all of the boxes, we need to iterate:

```
for item in all:  
    print(item.find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(  
        " ", ""))
```

- This prints out each price for a home on the first page of the webpage.
- Looking into the **address** data, we find that it's divided into two parts: the main address and the city/state information. These are tagged with ****, and we can use **[0]** and **[1]** to get to each of them. We'll also need to use some *string methods*.

```
In [31]: for item in all:  
    print(item.find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(" ", ""))  
    print(item.find_all("span", {"class": "propAddressCollapse"}))
```

\$725,000
[0 Gateway, Rock Springs, WY 82901]
\$452,900
[1003 Winchester Blvd., Rock Springs, WY 82901]
\$379,900
[3239 Spearhead Way, Rock Springs, WY 82901]
\$379,000
[600 Talladega, Rock Springs, WY 82901]

```
for item in all:  
    print(item.find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(  
        " ", ""))  
    print(item.find_all("span", {"class": "propAddressCollapse"})[0].text)  
    print(item.find_all("span", {"class": "propAddressCollapse"})[1].text)  
    print(" ")
```

- Now we need to extract the **number of beds** and **number of baths**. These also have "**span**" tags: **** and ****. However, we have to be careful here because the actual number of beds and baths in each is held within **** tags:

```
...  
    <div class="secondaryDetails" style="background-color: #f0f0f0; padding: 10px; border-radius: 5px;">  
        ...  
        <div class="infoLine1" style="margin-bottom: 10px;">  
            <span class="infoBed" style="font-weight: bold;">4 /<span class="infoValueFullBath" style="font-weight: bold;">4 Full Baths  
        
```

- Running a `.find_all` directly could cause some problems because we have multiple `` tags. The same applies to the `area` of the house.
- There are two potential solutions here:
 - You can assume that `beds` is always the first one and pass an index of `[0]` to the `ResultSet` list (then `[1]` for baths and so on).
 - But we're going to do it in a more constructive way. We're going to extract the `` tags first, and then go inside those and extract the info in the `` tags:

```
for item in all:
    print(item.find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(
        " ", ""))
    print(item.find_all("span", {"class": "propAddressCollapse"})[0].text)
    print(item.find_all("span", {"class": "propAddressCollapse"})[1].text)
    print(item.find("span", {"class": "infoBed"})) ← ← ←
    print(" ")
```

- Note that some of the results will be `None` (such as a property with no house on it yet, and therefore no beds). We can't run a `.text` method on this or we'll get an error. We can account for this situation using a `try/except` statement:

```
for item in all:
    print(item.find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(
        " ", ""))
    print(item.find_all("span", {"class": "propAddressCollapse"})[0].text)
    print(item.find_all("span", {"class": "propAddressCollapse"})[1].text)
    try: ← ← ←
        print(item.find("span", {"class": "infoBed"}).text)
    except: ← ← ←
        pass
    print(" ")
```

- Now we get the text for the `` tag as well as the text for the `` tag underneath that (i.e. "4 Beds"). We could just leave it like that, but he said he'd like to have the plain number only. To do that, we can apply the `.find` method again:

```
try:
    print(item.find("span", {"class": "infoBed"}).find("b").text) ← ← ←
except:
    pass
```

- Now that that's working, we can do the same for **number of baths**, **area**, and **half-baths**.
-
-
- Full iteration code so far on next page:

```
for item in all:
    print(item.find("h4", {"class": "propPrice"}).text.replace("\n", "").replace(
        "", ""))
    print(item.find_all("span", {"class": "propAddressCollapse"})[0].text)
    print(item.find_all("span", {"class": "propAddressCollapse"})[1].text)
    try:
        print(item.find("span", {"class": "infoBed"}).find("b").text)
    except:
        print(None)

    try:
        print(item.find("span", {"class": "infoSqFt"}).find("b").text)
    except:
        print(None)

    try:
        print(item.find("span", {"class": "infoValueFullBath"}).find("b").text)
    except:
        print(None)

    try:
        print(item.find("span", {"class": "infoValueHalfBath"}).find("b").text)
    except:
        print(None)

    print(" ")
```

Scraping Special Elements:

- So far we've scraped **price**, **address**, **beds**, **full-baths**, **half-baths**, and **square footage**.
- We still need to get the **Lot Size** for each entry. However, this one will be a little trickier to get. The reason for this is, if you look at the *source code*, you see that the entry "Lot Size" () and the entry for its size "x.xx Acres" () are both under a <div class="columnGroup">.
- When you look at another property with more attributes, you'll see that <div class="columnGroup"> is repeated multiple times for all of the different attributes that property might have (such as "**Age**", "**Amenities**", "**Appliances**", "**Architectural Style**"). Iterating through properties to extract just the **Lot Size** might lead you to accidentally extracting one of these other attributes.
 - In one example the instructor showed, you may be expecting to extract the **Lot Size**, but instead you'll extract the house's **Age** because that's the first attribute listed in that column.
- Instead, let's try **looping** through all of the **columnGroup** entries and checking if the **featureGroup** is named **Lot Size**. If it contains "Lot Size", we'll have it give us the **.text** value inside of **featureName** (such as **0.34 Acres**).
- We're going to place *another loop* inside of our *big loop*. We're also going to **print(column_group)** to test that we're getting the data we want:

```
for column_group in item.find_all("div", {"class": "columnGroup"}):
    print(column_group)
```

- This will search through all instances of "**columnGroup**":

```
except:
    print(None)
for column_group in item.find_all("div", {"class": "columnGroup"}):
    print(column_group)

    print(" ")
Rock Springs, WY 82901
None
None
None
None
<div class="columnGroup propFeatureHeader">FEATURES:</div>
<div class="columnGroup">
<span class="featureGroup">Architecture Style: </span><span class="featureName">Other</span>
</div>
<div class="columnGroup">
<span class="featureGroup">Roof Type: </span><span class="featureName">Unknown</span>
</div>
```

- Next we'll want to narrow it down and only search for ones where the **featureGroup** is named **Lot Size** if a given house has that attribute. We'll do this by iterating through all **featureGroup** instances. We'll also print "**feature_group.text**" and "**feature_name.text**" to test:

```
for column_group in item.find_all("div", {"class": "columnGroup"}):
    # print(column_group)
    for feature_group, feature_name in zip(column_group.find_all("span", {"class": "featureGroup"}), column_group.find_all("span", {"class": "featureName"})): ← ← ←
        print(feature_group.text, feature_name.text) ← ← ←
```

- This prints out all attributes of a given property. One property in particular has a lot of them:

```

for column_group in item.find_all("div", {"class": "columnGroup"}):
    # print(column_group)
    for feature_group, feature_name in zip(column_group.find_all("span", {"class": "featureGroup"}), column_group.find_all("span", {"class": "featureName"})):
        print(feature_group.text, feature_name.text)
    print("")

4
None
4
None
Age: New Construction
Appliances: Dishwasher,
Basement: Finished
Bath Features: Wall Shower and Tub,
Cooling: Central A/C
Exterior: Thermal Windows / Doors
Exterior Description: Other,
Exterior Living Space: Deck
Fireplace Count: 2 Fireplaces
Fireplace Description: Gas
Flooring: Hardwood,
Garage Count: 3 Car Garage
Garage Description: Attached Garage
Heating - Fuel Type: Gas
Heating Type: Forced Air
Interior: Walk-in Closet

```

- He pointed out that in the second property, a lot of attributes show up that are **hidden in the source code** on the webpage. You normally wouldn't be able to see them from the page where all the properties are listed together; you see them once you click on that property and go to its specific listing.
- Now we can check each **featureGroup** to see if it's named **Lot Size**:

```

for column_group in item.find_all("div", {"class": "columnGroup"}):
    # print(column_group)
    for feature_group, feature_name in zip(column_group.find_all("span", {"class": "featureGroup"}), column_group.find_all("span", {"class": "featureName"})):
        # print(feature_group.text, feature_name.text)
        if "Lot Size" in feature_group.text: ← ← ←
            print(feature_name.text) ← ← ←

```

- In the next lecture, we'll get rid of our various print statements and actually store all of our data in a **Pandas Dataframe**.

Saving the Extracted Data in CSV Files:

- We'll need an 8-column Pandas dataframe to contain all of our desired attributes.
- One solution is to *iterate through the dataframe*, but that's a costly way to do it in terms of time efficiency; dataframes weren't built to handle iteration efficiently.
- It's better to *create a dataframe from a Python dictionary or list of dictionaries*. This is how we'll do it.
- For each iteration, we'll add our values to a *dictionary*. Our first one will be the **Price**, with "Price" as the key and the price as the value (such as {"Price": "\$750,000"}), then **Address** (so now we have {"Price": "\$750,000", "Address": "0 Gateway"}) and so on. At the end we'll have {"Price": "\$750,000", "Address": "0 Gateway", "Locality": "Rock Springs WY 82901", "Beds": None, "Area": 3706, "Full Baths": None, "Half Baths": None, "Lot Size": 0.27} for one house. For the next iteration, we'll have the same keys again for another property's attributes. By the end, we'll end up with 10 dictionaries (10 properties per page on the website).
- We'll need somewhere to *store these dictionaries*. We'll store these dictionaries in a **list**.
- We also want to start each iteration with an empty dictionary, **d={}**.
- We replace **print** statements with, for example, **d["Price"] =** in order to set our keys and values on the fly:

```
for item in all:  
    d={} ← ← ←  
    d["Address"]=item.find_all("span",{"class":"propAddressCollapse"})[0].text  
    d["Locality"]=item.find_all("span",{"class":"propAddressCollapse"})[1].text  
    d["Price"]=item.find("h4",{"class":"propPrice"}).text.replace("\n","").replace(" ", "") ← ← ←  
    try:  
        d["Beds"]=item.find("span",{"class":"infoBed"}).find("b").text  
    except:  
        d["Beds"]=None
```

- By the end of the *first iteration* of our *big for-loop*, we'll have our **first dictionary**. We want to store that dictionary somewhere, so we'll create a list. We want to create this list *outside of the loop*:

```
l=[] ← ← ←  
for item in all:  
    d={} ← ← ←
```

- We then want to append our dictionary to this list: **l.append(d)** at the end of the for-loop:

```
        if "Lot Size" in feature_group.text:  
            d["Lot Size"]=feature_name.text  
        l.append(d) ← ← ←
```

- This will store a dictionary for each property in our main **list**.
- Printing **len(l)** gives us **10**, the number of properties on the page.

- Now we want to throw this list `l` to a **dataframe**. To do that, we need to **import pandas** and pass the list `l` into a dataframe:

```
import pandas < < < (actually up top, but he wrote it down below in Jupyter)
df=pandas.DataFrame(l) < < <
```

- Now when we print `df`:

	Address	Area	Beds	Full Baths	Half Baths	Locality	Lot Size	Price
0	0 Gateway	None	None	None	None	Rock Springs, WY 82901	NaN	\$725,000
1	1003 Winchester Blvd.	None	4	4	None	Rock Springs, WY 82901	0.21 Acres	\$452,900
2	3239 Spearhead Way	3,076	4	3	1	Rock Springs, WY 82901	Under 1/2 Acre,	\$379,900
3	600 Talladega	3,154	5	3	None	Rock Springs, WY 82901	NaN	\$379,000
4	3457 Brisol Avenue	3,236	5	3	None	Rock Springs, WY 82901	0.34 Acres	\$349,900
5	234 Vía Spoleto	2,688	4	3	None	Rock Springs, WY 82901	Under 1/2 Acre,	\$330,000
6	2425 Cripple Creek	8,263	4	35	None	Rock Springs, WY 82901	NaN	\$279,900
7	522 Emerald Street	1,172	3	3	None	Rock Springs, WY 82901	Under 1/2 Acre,	\$254,000
8	1302 Veteran's Drive	1,932	4	2	None	Rock Springs, WY 82901	0.27 Acres	\$252,900
9	343 Vía Rucce	None	3	2	1	Rock Springs, WY 82901	0.16 Acres	\$219,900

In []:

- Pandas even assigned **NaN** values for cases where there was no **Lot Size** in the dictionary.
- Now we just need to save the data to a .csv file with: `df.to_csv("Output.csv")`:

```
df=pandas.DataFrame(l)
#print(df)

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

- And now we have our data saved.
- In the next lecture, we'll work on **iterating through multiple pages**. The Century21 webpage has 3 pages worth of properties for Rock Springs, WY.

Crawling Through Multiple Web Pages:

- Rock Springs, WY has a total of **28** listings (actually **37 listings**, see note later in this lecture) spread out through 3 pages. We've only extracted **10** of the properties, those on the first page. This is a small town so we only have 28 listings, but *a large city will have many more*.
- What we need to do is load the *other pages* in Python as well. We'll do this using **requests** again, just as we did in the beginning of the program so far.
- We can either go manually through all of the pages (3 pages, or **300 pages** for a larger city) and copy the URL, or we can try to figure out if there is a **rule** that changes the URL when we switch through the pages. Clicking on Page 2 gives us the URL for this page.
 - **Note:** the cached version of the URL for page 2 is:
 - <https://pythonizing.github.io/data/real-estate/rock-springs-wy/LCWYROCKSPRINGS/t=0&s=10.html>
 - And the URL for page 3 is:
 - <https://pythonizing.github.io/data/real-estate/rock-springs-wy/LCWYROCKSPRINGS/t=0&s=20.html>
 - The added portions, “**t=0&s=10.html**” and “**t=0&s=20.html**”, give us an idea of how the website is constructed. Going back to the first page again shows us:
 - <https://pythonizing.github.io/data/real-estate/rock-springs-wy/LCWYROCKSPRINGS/t=0&s=0.html>
 - The added portion here for the first page is “**t=0&s=0.html**”.
 - So it goes “**t=0&s=0.html**”, “**t=0&s=10.html**”, “**t=0&s=20.html**”.
- So let's go into Jupyter and store a **base_url**:

```
base_url="https://pythonizing.github.io/data/real-estate/rock-springs-wy/LCWYROCKSPRINGS/t=0&s="
```

- This is the portion of the URL that's shared by all the pages (the number at the end, **0/10/20**, has been removed).
- To test that iterating through these is going to work, let's use print statements within a for-loop. We'll be using **range(0, 30, 10)** here, starting at the beginning (**0**), stepping up by **10**, and then **30** is an arbitrary number we've chosen because it's high enough to include all **3** pages for now. Later we'll want to change this to a variable that can adapt to a city with many more pages:

```
for page in range(0,30,10):
    print(base_url + str(page) + ".html")
```

- Running this gives us:

```
In [38]: base_url="http://www.century21.com/real-estate/rock-springs-wy/LCWYROCKSPRINGS/#t=0&s="
for page in range(0,30,10):
    print(base_url+str(page))

http://www.century21.com/real-estate/rock-springs-wy/LCWYROCKSPRINGS/#t=0&s=0
http://www.century21.com/real-estate/rock-springs-wy/LCWYROCKSPRINGS/#t=0&s=10
http://www.century21.com/real-estate/rock-springs-wy/LCWYROCKSPRINGS/#t=0&s=20
```

```
In [ ]:
```

- The URLs are now stepping up from **0** to **10** to **20** at the end, and they can be opened by clicking.

- Now we want to (again) set up our **requests (r)**, **content (c)**, and **soup**:

```
for page in range(0,30,10):
    print(base_url+str(page)+".html")
    r=requests.get(base_url+str(page)+".html") < < <
    c=r.content < < <
    soup=BeautifulSoup(c, "html.parser") < < <
    print(soup.prettify()) < < <
```

- We ran **print(soup.prettify())** just to make sure we were getting all the source code, but now we want to replace that with **all=soup.find_all("div",{"class": "propertyRow"})**:

```
for page in range(0,30,10):
    print(base_url+str(page)+".html")
    r=requests.get(base_url+str(page)+".html")
    c=r.content
    soup=BeautifulSoup(c, "html.parser")
    # print(soup.prettify())
    all=soup.find_all("div", {"class": "propertyRow"}) < < <
```

- He ran **print(all)** to show that it worked, but it was still a lot of data.
- Since he wrote this section of code at the bottom of the Jupyter Notebook session, we want to copy all of that code and paste it up above our **big for-loop**. We also want to indent everything in our **big for-loop** to the right so that it becomes part of the code we just wrote:

```
l=[]
base_url=https://pythonizing.github.io/data/real-estate/rock-springs-
wy/LCWYROCKSPRINGS/t=0&s= < < < place all the code we just wrote up here
for page in range(0,30,10):
    print(base_url+str(page)+".html")
    r=requests.get(base_url+str(page)+".html")
    c=r.content
    soup=BeautifulSoup(c, "html.parser")
    # print(soup.prettify())
    all=soup.find_all("div", {"class": "propertyRow"})

    for item in all: < < < indent so it runs for each page
        d={}
        d["Address"]=item.find_all("span", {"class": "propAddressCollapse"})[0].tex
t
        d["Locality"]=item.find_all("span", {"class": "propAddressCollapse"})[1].te
xt
        d["Price"]=item.find("h4", {"class": "propPrice"}).text.replace("\n", "").re
place(" ", "")
        try:
            d["Beds"]=item.find("span", {"class": "infoBed"}).find("b").text
        except:
```

```

d[ "Beds"]=None

try:
    d[ "Area"]=item.find("span", {"class":"infoSqFt"}).find("b").text
except:
    d[ "Area"]=None

try:
    d[ "Full
Baths"]=item.find("span", {"class":"infoValueFullBath"}).find("b").text
except:
    d[ "Full Baths"]=None

try:
    d[ "Half
Baths"]=item.find("span", {"class":"infoValueHalfBath"}).find("b").text
except:
    d[ "Half Baths"]=None

for column_group in item.find_all("div", {"class":"columnGroup"}):
    # print(column_group)
    for feature_group, feature_name in
zip(column_group.find_all("span", {"class":"featureGroup"}), column_group.find_all(
"span", {"class":"featureName"})):
        # print(feature_group.text, feature_name.text)
        if "Lot Size" in feature_group.text:
            d[ "Lot Size"]=feature_name.text
    l.append(d)

```

- **Note:** This may be due to changes in the cached website (they've apparently gone through at least two iterations of this), but after making the changes to crawl through multiple pages, I get a DataFrame that includes entries from **Rocksprings, TX** and **Black Canyon City, AZ**. Not sure why this is happening, but apparently one other student had the same issue and it never got resolved.
 - **Update:** This does appear to be due to the way the cached website is set up. For some reason, properties from those other cities are mixed into the 3 pages that are supposed to be for **Rock Springs, WY**. Not really a good way to fix it, but at least it's not because there's anything wrong with my code.
- **Note:** There are actually **37 listings** from the cached website due to the above weirdness.
- In the instructor's case, he was missing a few listings towards the end. Looking into the cause, it turns out that the property it failed on had a **Locality** listed, but no Address. So we'll add an exception to handle this case.
- I went ahead and just added **try/except** statements to **all** of my keys. The only ones that were missing it were **Address**, **Locality**, and **Price**. Covered all my bases that way.

- The last thing we want to do is to replace our **range** value of **30** with a variable.
- We find out what's controlling the page by running **inspect** on, say "**3**" in the page links (we want to do this while we're **not** on Page 3, so it'll be a link). This leads us to:
 - **<a class="Page"**
 - In this case, Page 3 is the **last page**, so we want to keep that in mind when calculating our variable (we'll use [-1] here):

```
page_nmbr=soup.find_all("a", {"class": "Page"})[-1].text ← ← ←
print(page_nmbr)
```

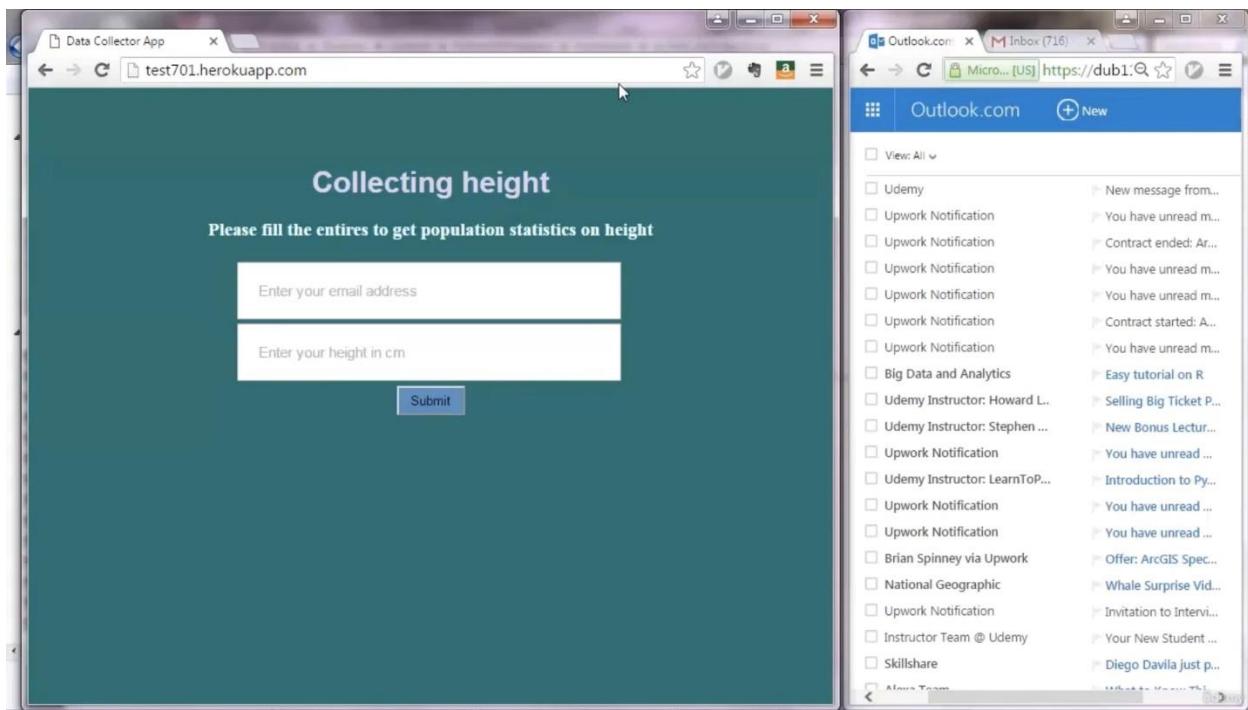
- Now we can plug this into our **range** function directly and multiply it by 10 to get **30** (note: remember that **page_nmbr** is currently a **string**, so convert it to **int**):

```
for page in range(0, int(page_nmbr)*10, 10): ← ← ←
    print(base_url+str(page)+".html")
    r=requests.get(base_url+str(page)+".html")
    c=r.content
    soup=BeautifulSoup(c, "html.parser")
    # print(soup.prettify())
    all=soup.find_all("div", {"class": "propertyRow"})
```

- And we're pretty much done.
- Some things we can modify or add are:
 - We can try this with **other localities** besides Rock Springs, WY.
 - We could also add some sort of **user input** to search for a different locality.

Section 31: App 8: Flask and PostGreSQL – Build a Data Collector Web App:

Demo of the Data Collector Web App:



- What this application does is it asks users for their email and their height.
- Once it gets this pair of data from the user, it sends that data to the database on the server, then calculates the average height of all users that have submitted so far. Then it sends that average to the user that just submitted their email address.
 - Note: Apparently this is in **cm**.
- When you submit an email and height, it sends you to a success page, saying "**Thank you for your submission! You will receive an email with the survey results shortly.**"
- He used one of his email accounts to send the emails out, and another one to test that it's being sent out properly. The email contains **your height**, the **average height**, and **how many people** have submitted so far.

Steps of Creating a PostGreSQL Database Web App with Flask:

- Remember, almost every web application has a **front-end** and a **back-end**.
 - The **front-end** is made up of **HTML**, **CSS**, and **JavaScript** when a website has JavaScript. In our case, we will just have HTML and CSS. The HTML is responsible for the building blocks of our website; the title/heading, text, input boxes, and submit button. The CSS is responsible for *styling* the website, adding color, animations, fonts, etc.
 - The **back-end** handles the input data *using Python*, which *grabs* the two pieces of data, *analyzes* them if needed, and then sends the data to a **database**. You can also send data back to the front-end if needed (such as the success page).
- **The Steps:**
 - **HTML and CSS:** The instructor prefers to build the **frontend** first and then move on to the **backend** later.
 - Some people do it the other way around, but he prefers to think about the **user requirements** first, which we build the interface around. So in the next lectures, we'll build the **HTML** part and the **CSS** part really quickly.
 - **HTML and CSS:** We'll build those two files locally and then test them on our browser.
 - **Flask:** Then we'll build the basic structure of a **Flask** application.
 - **Test Python:** Once we have the basic structure built out, we'll make sure Python is receiving the input data correctly, the email addresses and the height. We'll *use some print statements* to test that we're getting the data correctly without using a database yet.
 - **PostGreSQL Database:** Then we'll create our database and create a **table** for the two fields, email and height. Once our table is built, we'll *send our data to the database* from the inputs.
 - **Query Data and Run Calculations:** We also want to write some code to **query data** from the database (the height) and **calculate the average**. We'll use *print statements* at this stage to make sure the calculations are being done correctly before we handle the email portion of the application.
 - **Email Code:** Implement the portion of the application that sends out emails.
 - **Launch:** Launch the web application so anyone can use it.

Creating a Page with HTML:

- Remember from the last time we made a **Flask webpage** that we need to structure our folders for it.
 - We need a **templates** folder and a **static** folder, and then our **Python file** will also go in this location.
- We're going to build the **HTML file** first.
 - Inside our **templates folder** we create a file, **index.html**.
 - Inside our **static folder** we create a file, **main.css**.
- To create our HTML file, we write (note: end tags not shown here, for space-saving):

```
<!DOCTYPE html>
<html lang="en">
    <title>Data Collector App</title>
    <head>
        <link href="../static/main.css" rel="stylesheet">
    </head>
    <body>
        <div class="container">
            <h1>Collecting Height</h1>
            <h3>Please fill the entries to get population statistics on
height</h3>
```

- We set the DOCTYPE to "html", we set the language to English inside the <html> tag.
- Inside of <head> we set a <link> tag, <link href="../static/main.css" rel="stylesheet"> to tell the HTML code where to look for the CSS file. The "../" sets the relative file path (".." means "go up one level" in many command line systems).
- Down in the <body> section we use a <div> tag because it's a good practice to keep things nice and organized in HTML code. Inside here we have <h1> and <h3> tags communicating with our user what they're supposed to do with the web application.
- We tested the HTML out in our browser, and the basic structure seems to be working.
- Inside of **templates** we created another HTML file to reference, **success.html** and copy/pasted

```
<!DOCTYPE html>
<html lang="en">
    <title>Data Collector App</title>
    <head>
        <link href="../static/main.css" rel="stylesheet">
    </head>
    <body>
        <div class="container">
            <p> Thank you for your submission <br>
            You will receive an email with the survey results shortly.
        </p>
```

most of our code from index.html to use as a template. **Note:** Both HTML pages will reference the *same main.css* for styling.

- Now we're going to add ***two inputs*** and ***a button*** to our **index.html** file:

```
<form action="success.html" method="POST">
    <input title="Your email will be safe with us"
placeholder="Enter your email address" type="email" name="email_name"
required> <br>
    <input title="Your data will be safe with us"
placeholder="Enter your height in cm" type="number" min="50" max="300"
name="height_name" required>
    <button type="submit"> Submit </button>
</form>
```

- We do this inside of a **<form>** tag that sends users to success.html when accomplished.
- We start with our first **<input>**, setting it to **type="email"**, giving it a **name** variable/attribute that Python can search for later, and setting it to **required**.
- We add another **<input>** setting it to **type="number"**, which restricts what kind of input the user can enter, meaning we don't need to handle cases in our Python code. We also set **min="50"** and **max="300"**, add a **name** attribute for Python, and set it to **required**.
- We then add a **<button>** and set its **type="submit"**. Between the tags we just add the text **"Submit"** that we want to appear on the button.
- Testing index.html in our webpage now shows the two entry fields and the button have been added. He showed that if we pass something that doesn't look like an email into the email field, the web app gives a warning saying that it needs to include an **@**. Similarly, entering anything that's not a number or is outside of the range, you get a warning in the height entry.
 - If you enter something with an **@** and a valid number within the range, it sends you to **success.html**.
- In the next lecture, we'll write the **CSS** code, **main.css**, to stylize our webpage and make it look nice and professional.

Full index.html Code, So Far:

```
<!DOCTYPE html>
<html lang="en">
    <title>Data Collector App</title>
    <head>
        <link href="../static/main.css" rel="stylesheet">
    </head>
    <body>
        <div class="container">
            <h1>Collecting Height</h1>
            <h3>Please fill the entries to get population statistics on
height</h3>
            <form action="success.html" method="POST">
```

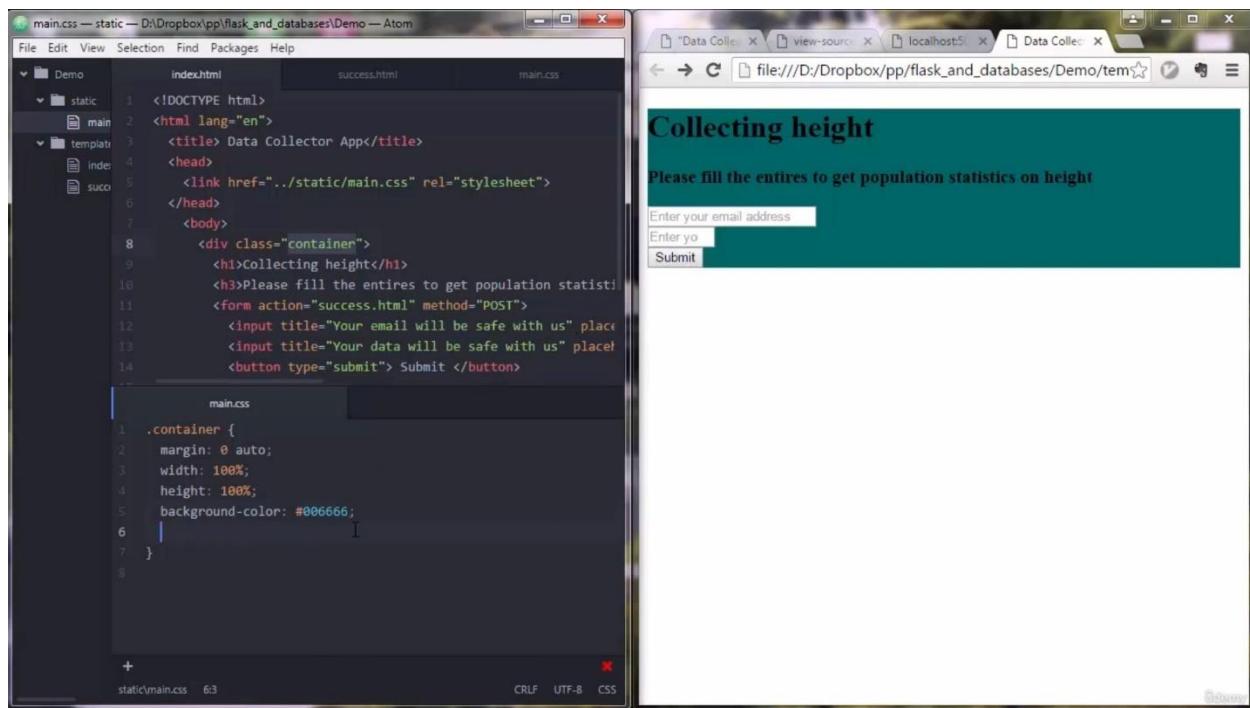
```
        <input title="Your email will be safe with us"  
placeholder="Enter your email address" type="email" name="email_name" required>  
<br>            <input title="Your data will be safe with us"  
placeholder="Enter your height in cm" type="number" min="50" max="300"  
name="height_name" required> <br>  
                <button type="submit"> Submit </button>  
            </form>  
        </div>  
    </body>  
</html>
```

Stylizing the HTML Page with CSS:

- So we have our plain webpage. Time to spruce it up with CSS.
- We started off with just:

```
.container {  
    margin: 0 auto;  
    width: 100%;  
    height: 100%;  
    background-color: #006666;  
}
```

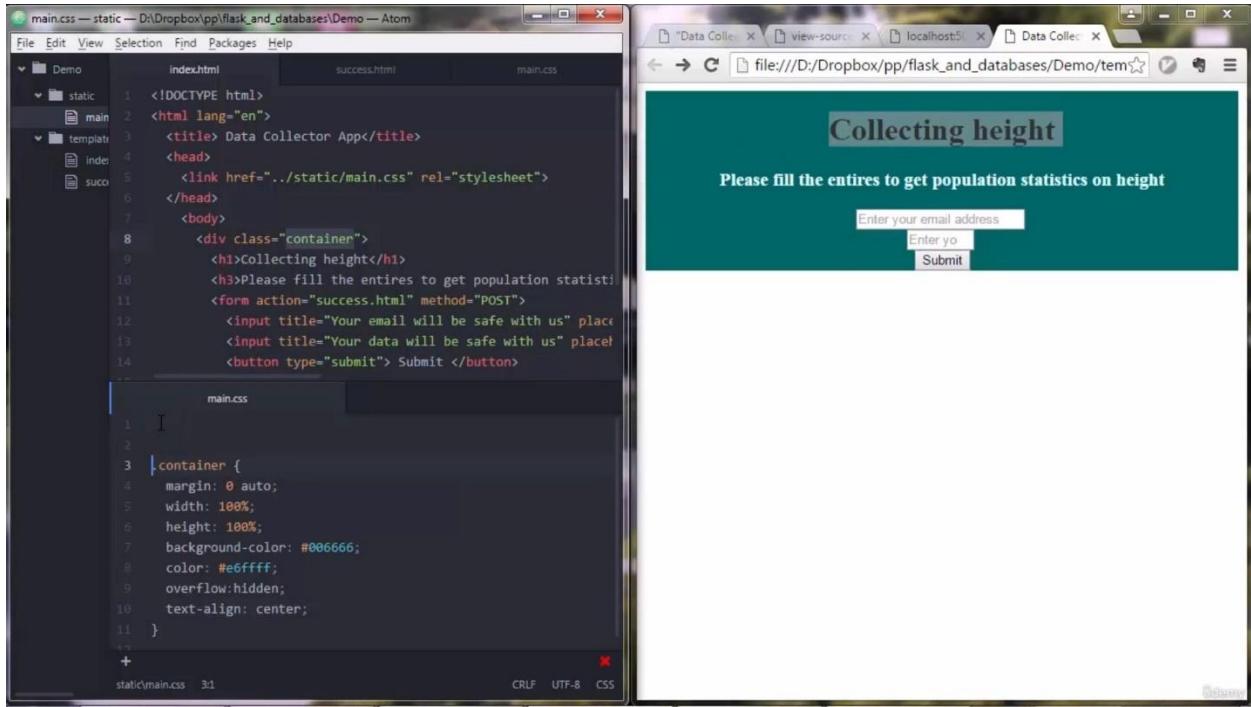
- Where **.container** references `<div class="container">` in `index.html`. Saving this file and reloading our webpage shows that just the container is now teal:



- Let's add a few more options:

```
.container {  
    margin: 0 auto;  
    width: 100%;  
    height: 100%;  
    background-color: #006666;  
    color: #e6ffff; ← ← ←  
    overflow: hidden; ← ← ←  
    text-align: center; ← ← ←  
}
```

- Now the text is a different color and everything is aligned in the center:

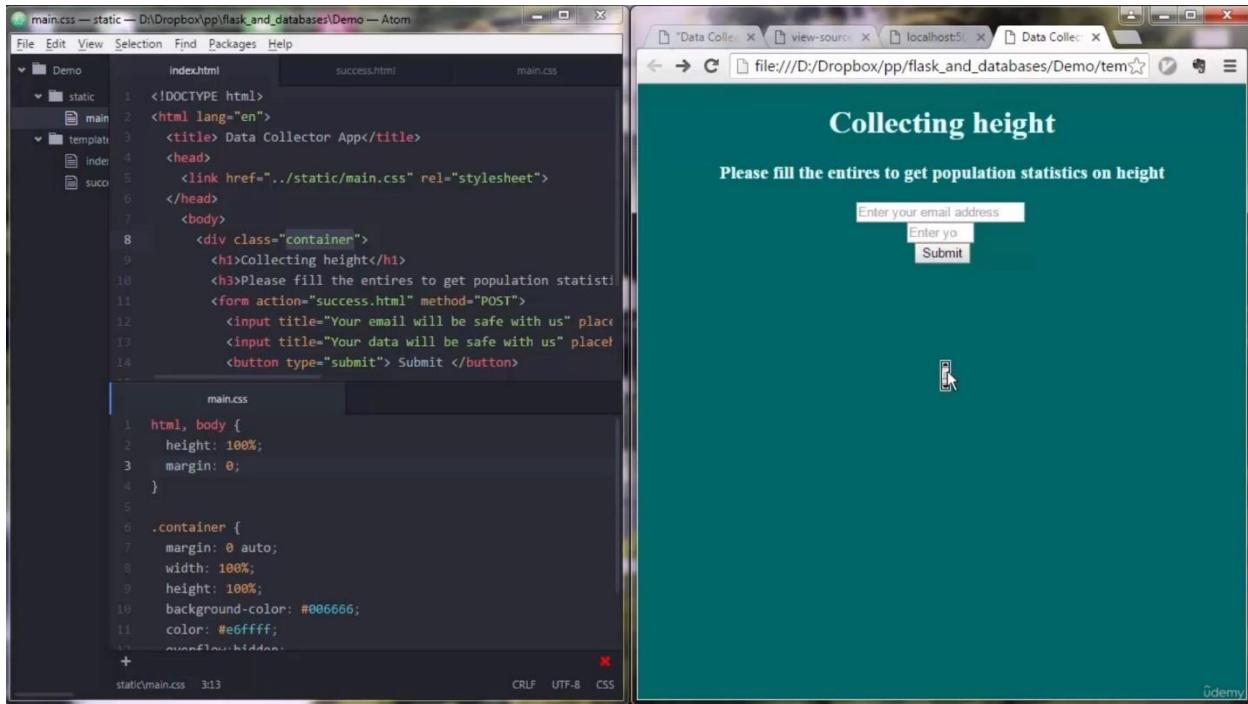


- Now we want to get rid of the margins and also color the empty area. We do that by adding the **html** tag and the **body** tag up at the top:

```
html, body { ← ← ←
    height: 100%; ← ← ←
    margin: 0; ← ← ←
}

.container {
    margin: 0 auto;
    width: 100%;
    height: 100%;
    background-color: #006666;
    color: #e6ffff;
    overflow: hidden;
    text-align: center;
}
```

- And now the whole webpage is looking much better (shown on next page):



- Now what else does it need?
- If we want, we can specifically point to tags such as `<h1>` or `<h3>`. We'll leave `<h3>` as-is, but let's point to `<h1>` with `.container h1 {`:

```
.container h1 {
    font-family: Arial, sans-serif;
    font-size: 30px;
    color: #DDCCCE;
    margin-top: 80px;
}
```

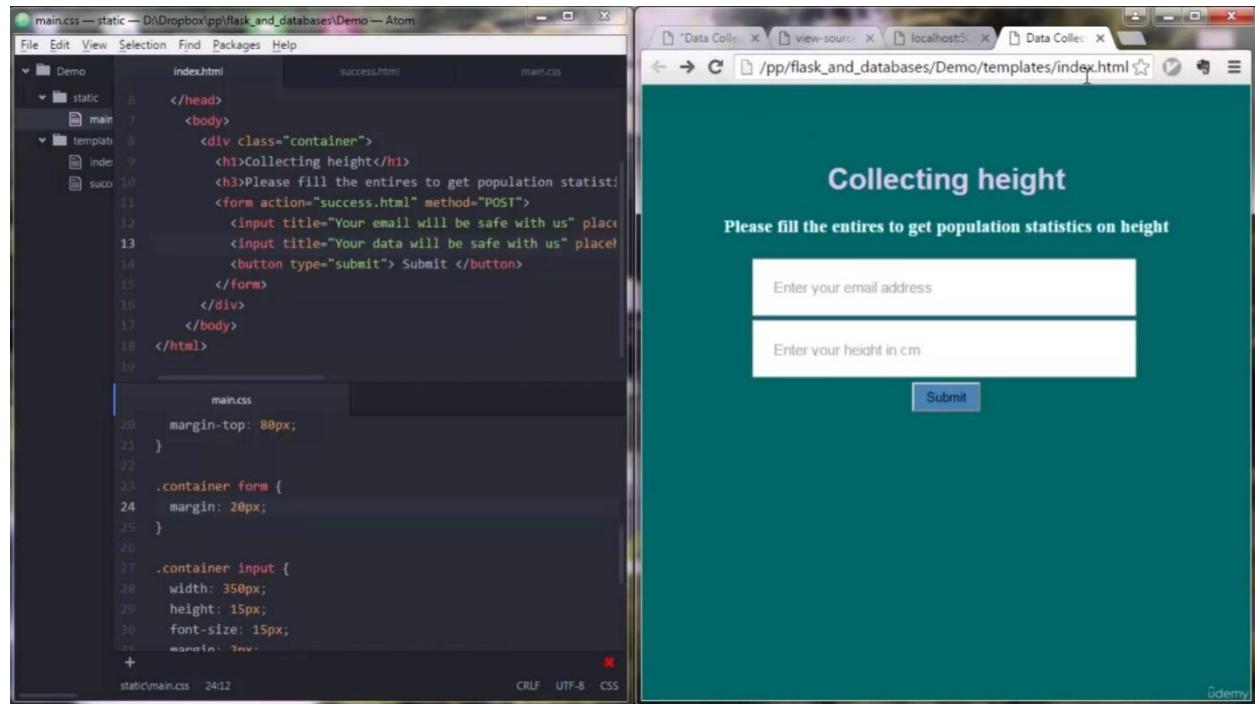
- This changes the font type, font size, and color. Adding **margin-top 80px** also lowers everything in the webpage by a bit.
- Let's also make changes with **.container form {**, **.container input {**, and **.container button {**.

```
.container form {
    margin: 20px;
}

.container input {
    width: 350px;
    height: 15px;
    font-size: 15px;
    margin: 2px;
    padding: 20px;
    transition: all 0.2s ease-in-out;
}

.container button {
    width: 70px;
    height: 30px;
    background-color: steelblue;
    margin: 3px;
}
```

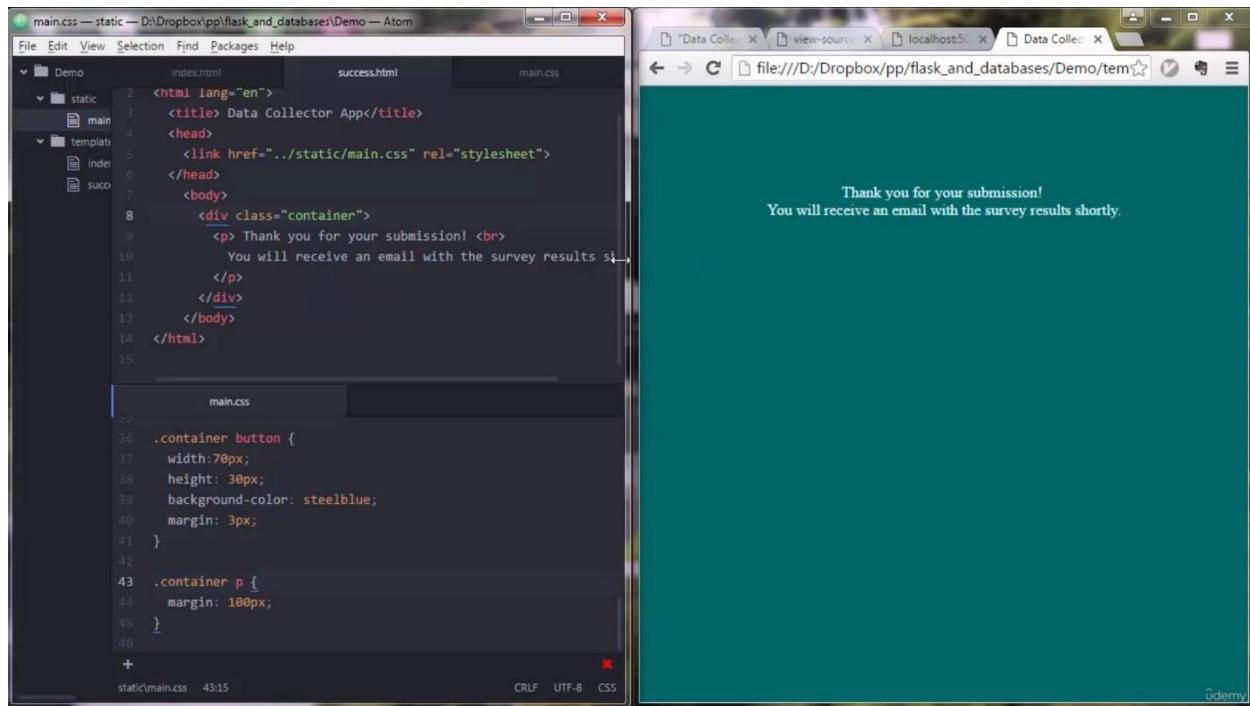
- This gives us this result:



- We also loaded **success.html** to see that similar changes were rendered there with color, alignment, and margins. We can also specifically target the text in **success.html** to place it further down on the page; we do this by adding **.container p{}**, which will look for the **<p>** tag in **success.html**:

```
.container p {
    margin: 100px;
}
```

- Which results in:



- And with that we're done with this lecture. In the next lecture, we'll focus on capturing the user input.

Full CSS Code:

```
html, body {
    height: 100%;
    margin: 0;
}
```

```
.container {  
    margin: 0 auto;  
    width: 100%;  
    height: 100%;  
    background-color: #006666;  
    color: #e6ffff;  
    overflow: hidden;  
    text-align: center;  
}  
  
.container h1 {  
    font-family: Arial, sans-serif;  
    font-size: 30px;  
    color: #DDCCEE;  
    margin-top: 80px;  
}  
  
.container form {  
    margin: 20px;  
}  
  
.container input {  
    width: 350px;  
    height: 15px;  
    font-size: 15px;  
    margin: 2px;  
    padding: 20px;  
    transition: all 0.2s ease-in-out;  
}  
  
.container button {  
    width: 70px;  
    height: 30px;  
    background-color: steelblue;  
    margin: 3px;  
}  
  
.container p {  
    margin: 100px;  
}
```

Capturing User Input:

- In this lecture, we'll finally be focusing on our **Python script** to capture the user input.
- We want to start by creating a **virtual environment**, similar to the last time we used Flask.
 - It's not strictly necessary to build in a virtual environment, and one could work on this program without one, but it helps you *isolate your environment in a fresh Python installation*, which later you can use for *deploying your web application on the cloud or on a web server*.
- **Creating the Virtual Environment:** We start by opening a command line in the directory we want to save our Python program to (the same directory that contains the sub-directories **static** and **templates**).
 - We make sure we have **virtualenv** installed by running **pip install virtualenv**.
 - Running this gave me the message "Requirement already satisfied", so we're good.
 - Next we run **python -m venv virtual** where "virtual" is the name of the folder where you want you want your *virtual environment to be created*.
 - If running on a **bash**-based system (Linux, Mac): **python -m virtualenv virtual** instead.
 - Running these commands can take a while, but wait for the command line to reappear.
 - A folder called **virtual** has been created in the same directory as **static** and **templates**. It includes a **clean installation of Python** as well as **pip**. Inside of it are:
 - **Include** directory.
 - **Lib** directory.
 - **Scripts** directory, where *Python* and *pip* are installed, among other things.
 - A configuration file, **pyvenv.cfg**.
 - Once our virtual environment is configured, we run:
 - **virtual\Scripts\pip install flask** in the command line to install Flask for the virtual environment. That's one way to do it.
 - Or, we could also run:
 - **virtual\Scripts\activate** which is a .bat or batch file. That will trigger your virtual environment in the *console*.
 - Once that's triggered, we can just run **python** and the interactive Python shell of our virtual environment will be triggered. Use CTRL+Z to get out.
 - Now we can run **python app.py** in our *virtual console*, or we can do it the old way with **virtual\python app.py**.
 - Throughout these lectures, we'll mainly be using **activate** because it will help us to be more focused on the main things we're working on.
 - Now we run:
 - **pip install flask** to install Flask on our clean virtual installation.
- Once Flask is installed, we open our main app directory (the one containing the templates, static, and virtual) in our IDE. This is the location where we'll be creating our **Python file**. We're going to name it **app.py**.
- Now we begin building our Python program:
- We start with **from flask import Flask, render_template, request**, so the Flask class and two methods.

- We also create an `app` variable and set it to `Flask(__name__)`.

```
from flask import Flask, render_template, request

app=Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

if __name__ == '__main__':
    app.debug=True
    app.run()
```

- We also use the **decorator**, `@app.route("/")`, or home page.
- The rest is pretty much just how you set up any Flask-based website. The `index()` function uses the `render_template` method to render `index.html` for us.
- Inside the `.run()` method we can specify the **port** if we want: `app.run(port=5001)` for example. But we'll leave it empty.
- Now in our **console** (his built-in console for Atom was giving him issues, so we're doing this section in the Windows console), we run `python app.py` in our active virtual environment.
 - This gave me a message "Running on http://127.0.0.1:5000", showing the **localhost IP address** and the **column of 5000**.
 - It also gave me a warning message: "WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI serve instead". I suppose that's helpful information.
 - Pasting "Running on http://127.0.0.1:5000" or "**localhost:5000**" into the browser now **shows our web page**.
 - However, we still have quite a lot of things to do to get everything working together. Currently if we type in a test email and height (such as `a@a.com` and `176`) and click submit, we get an **error page**: "Not Found. The requested URL was not found" etc.
 - The reason for this is that we haven't yet **rendered success.html** in our Python program. It needs its own rendering function:

```
@app.route("/")
def index():
    return render_template("index.html")

@app.route("/success") ← ← ←
def success(): ← ← ←
    return render_template("success.html") ← ← ←
```

- Now it works, but only if we visit the URL `localhost:5000/success` manually in our browser. It still doesn't work if we try and submit an email and a height and then click on the Submit button.

- The reason for this is that in the `index.html` file, it's just rendering a file (`success.html`) which Flask isn't able to find. So we need to add a **placeholder** here with double brackets:
`{url_for('success')}`:

```
<form action="{{url_for('success')}}" method="POST"> ← ← ←
```

- Now when we enter data and hit "Submit", we get a new error page: "**Method Not Allowed**, The method is not allowed for the requested URL." This would be the **POST method** called out in the tag.
- However, when you use Python decorators here (`@app.route()`), these *implicitly declare a get method*. If we want to change this to a **post method**, we declare that in the decorator as a *list*:

```
@app.route("/success", methods=['POST']) ← ← ←
def success():
    return render_template("success.html")
```

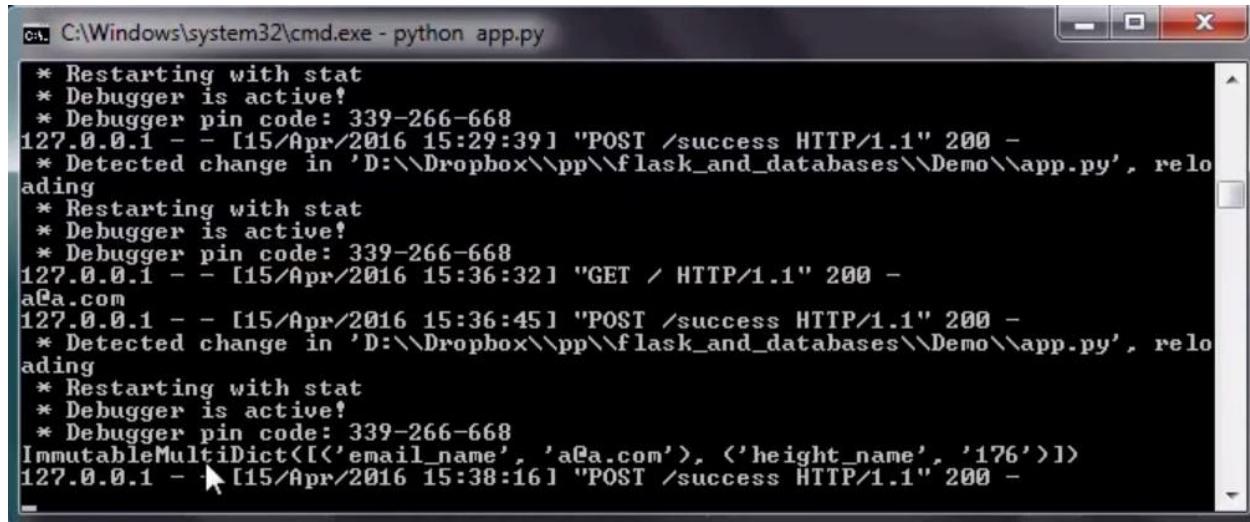
- And now hitting the "Submit" button works just fine.
- Now where are the **email** and **height** values? They're being *passed to the server* as a *post method*, but we need to capture them. We'll do that inside of the **success function**.
 - Sometimes you might get a **get method** through this URL, so for this reason, you'll want to make sure that you check that the method is for **POST**, using an **if statement**. This helps differentiate whether the page was reached manually or through the POST request of clicking the "Submit" button.

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST': ← ← ← checks we got here by clicking button
        email=request.form["email_name"] ← ← ← sets email from email_name
        print(email) ← ← ← prints email to PowerShell console to test
        height=request.form["height_name"] ← ← ← sets height
        print(height) ← ← ← prints height to PowerShell console to test
        return render_template("success.html") ← ← ← renders success.html
```

- Now we can see those values printed to our console. We also see both GET and POST requests, depending on what we're doing in the web app:

```
* Restarting with stat
* Debugger is active!
* Debugger PIN: 121-162-857
127.0.0.1 - - [12/Feb/2023 13:28:08] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Feb/2023 13:28:08] "GET /static/main.css HTTP/1.1" 304 -
a@a.com
176
127.0.0.1 - - [12/Feb/2023 13:28:12] "POST /success HTTP/1.1" 200 -
127.0.0.1 - - [12/Feb/2023 13:28:13] "GET /static/main.css HTTP/1.1" 304 -
```

- If we want to see more of what's going on behind the scenes, we can change our `print` statement to be `print(requests.form)`:



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe - python app.py'. The window displays log output from a Flask application. It includes messages about restarting with static files, debugger status, and a POST request from '127.0.0.1' containing an 'email_name' key with value 'a@a.com' and a 'height_name' key with value '176'. The log ends with a successful POST response.

```
* Restarting with stat
* Debugger is active!
* Debugger pin code: 339-266-668
127.0.0.1 - - [15/Apr/2016 15:29:39] "POST /success HTTP/1.1" 200 -
* Detected change in 'D:\\Dropbox\\pp\\flask_and_databases\\Demo\\app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger pin code: 339-266-668
127.0.0.1 - - [15/Apr/2016 15:36:32] "GET / HTTP/1.1" 200 -
a@a.com
127.0.0.1 - - [15/Apr/2016 15:36:45] "POST /success HTTP/1.1" 200 -
* Detected change in 'D:\\Dropbox\\pp\\flask_and_databases\\Demo\\app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger pin code: 339-266-668
ImmutableMultiDict([('email_name', 'a@a.com'), ('height_name', '176')])
127.0.0.1 - - [15/Apr/2016 15:38:16] "POST /success HTTP/1.1" 200 -
```

- Running `print(requests.form)` gives us something called an `ImmutableMultiDict`. This is like a *dictionary*, holding our data as keys and values.
-
- In the next lecture, we'll be taking our `email` and `height` values and sending them to a **PostgreSQL database**.

Creating the PostGreSQL Database Model:

- So now we have access to the **email** and **height** values that the user is entering into our website interface. Now we're going to store these values in a **PostGreSQL database**.
- We need to make sure that a **PostGreSQL** database is *installed in our computer*, and we need to create a **table** in that database to store our values. Then we need to create an **email column** and a **height column** in that table.
- Once we have all of that set up, instead of printing the two values, we'll enter them into the table as a **row**.
-
- So to start off, we need to make sure **PostGreSQL is installed**. We've gone over this once in a previous lecture (when we built our **Bookstore Application**).
- One way to create database tables and to access PostGreSQL in general is to use **pgAdmin III** (**pgAdmin 4** in my case; this video is from 2016). This was installed with PostGreSQL during the Bookstore App section. *Note: I had to download an updated version, which took about 30ish minutes.*
 - When you install PostGreSQL, it will ask you to create a password.
 - We set our *super user account* to **postgres** and our *password* to **postgres123** during that lecture.
- Open **pgAdmin**, then *right-click* on "PostgreSQL" and click "**Connect**". *Once I'd updated and relaunched pgAdmin, I seemed to already be connected to the database.*
- I currently have 2 databases listed: "database1" (from "Bookstore" app) and "postgres" (present by default). Instructor has 4 databases total.
- We're going to **create a new database** for our application. *Right-click* on "Databases" → "New Object" → "New Database...". We're going to name our new database "**height_collector**" and specify the "Owner" as "**postgres**", then click OK.
- So our database has been created, but we don't have any tables in there yet (these would be found under "height_collector" → "Schemas" → "Tables").
 - We're going to **create our table** in **Python**.
 - We built our Bookstore app with **tkinter** and **psycopg2**. Tkinter handled the GUI and psycopg2 allows us to access our PostGreSQL database. It allows us to send SQL statements to our database.
 - We can use the same *library* (psycopg2) with Flask, but a more commonly used library for integrating PostGreSQL with Flask is **SQLAlchemy**.
 - **SQLAlchemy** is based on, but more high-level than psycopg2. We apply operations with less lines of code; we don't need to *establish a connection* to the database, *commit the changes*, and then *close the connection*.
-
- First off, we *quit* out of our running Flask operation in cmd/PowerShell (CTRL+C). Now we're back in just our **virtual environment**.
 - Now we run **pip install psycopg2**. *Note: Instructor had issues, tried upgrading pip (didn't help) and then had to search for a precompiled library online for psycopg2.*
 - Next we run **pip install Flask-SQLAlchemy**.
 - Once that's successfully installed, we go to our Python script, **app.py**.

- We're going to *import SQLAlchemy* to our Python program:

```
from flask import Flask, render_template, request
from flask_sqlalchemy import SQLAlchemy < < <
```

- Note: In the video (from 2016) he ran “**from flask.ext.sqlalchemy import SQLAlchemy**” instead. This doesn’t seem to be the correct command anymore.
- He explained that doing this references a library that is stored in the “**virtual**” folder in the app directory, aka your *virtual environment*.
 - His “**from flask.ext.sqlalchemy import SQLAlchemy**” line pointed to “Virtual” → “Lib” → “site-packages” → “flask” → “ext” → “**__init__.py**”.
 - My “**from flask_sqlalchemy import SQLAlchemy**” line points to “Virtual” → “Lib\site-packages” → “flask_sqlalchemy” → probably “**__init__.py**”.
 - Opening *this __init__.py* shows a line “**from .extension import SQLAlchemy**”, so apparently they chose to streamline things.
- Now, once we have a database, we want to create a **model** for our database, a **table** with **columns**. We specify what types of columns we want: int, float, str, etc. We’ll write this model with a **Python class**.
- We’re going to create a **class Data**, and this class inherits its type from SQLAlchemy (see below, after configuration). Before this, we create a variable **db** and set it to a function from SQLAlchemy, inputting the name of our app inside:

```
app=Flask(__name__)
db=SQLAlchemy(app) < < <
```

- However, at this point we’re still not specifying the connection to our database, so our Flask application still doesn’t know what database to connect with. We want to use **.config()** to do this:

```
app=Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='postgresql://postgres:postgres123@localhost/height_collector' < < <
db=SQLAlchemy(app)
```

- By doing this, we’re *setting the value* of the dictionary key “**SQLALCHEMY_DATABASE_URI**” to our database at localhost. This looks similar to a web page address, with the “**postgresql**” server followed by the username “**postgres**”, the password “**postgres123**” AT localhost, followed by the database name **height_collector**.
 - Doing this lets the app know the *location* of the database.
- Now that we’re configured our app to find our database, we can get started on creating our new **class Data**. We’ll be inheriting from the **db.Model class** from SQLAlchemy:
 -
 -
 -
 -

- Now that we're configured our app to find our database, we can get started on creating our new **class Data**. We'll be inheriting from the **db.Model class** from SQLAlchemy:

```
class Data(db.Model):
    __tablename__="data"
    id=db.Column(db.Integer, primary_key=True)
    email_=db.Column(db.String(120), unique=True)
    height_=db.Column(db.Integer)
```

- Inherits object attributes from **db.Model**.
- We set the table name to “**data**”.
- We set an **ID** for each entry, only accept integers, and set it as **primary_key**.
- We create the variable **email_**, only accepts strings 120 characters or less, and only accepts **unique** emails. The underscore at the end just helps us differentiate with the “email” variable elsewhere in the Python program.
- We create the variable **height_** and set it to only accept integers. The underscore at the end just helps us differentiate with the “height” variable elsewhere in the Python program.
- Then we set up the **initialization function**, **__init__**. This is the first thing that executes when you *call a class instance*:

```
class Data(db.Model):
    __tablename__="data"
    id=db.Column(db.Integer, primary_key=True)
    email_=db.Column(db.String(120), unique=True)
    height_=db.Column(db.Integer)

    def __init__(self, email_, height_): < < <
        self.email_ = email_ < < <
        self.height_ = height_ < < <
```

-
- Now let's go ahead and call an **instance of this class**. But where do we call the class?
 - Now, we could put it lower down in our program (say, after “def success”), but this would execute the Flask app.
 - So a better idea would be to go to our *Command Line* (in our virtual environment) and trigger a *Python session* where we'll *import the db object from this script*. This is where our last three lines...

```
if __name__ == '__main__':
    app.debug=True
    app.run()
```

- ...come in handy. When we import something from the script, the script is not executed, because the name of the script will not be “**__main__**”, but it will be “**app**”.
- So in our *command line*, we run **python**, and then:

- So in our *command line*, we run **python**, and then:
 - Run **from app import db**.
 - Run **db.create_all()**. *Note: See below for three solutions.* This will create the tables/columns that we need using the model class **Data** that we created.
 - *Note: I kept getting errors for this.*
 - Taken from Stack Overflow:
<https://stackoverflow.com/questions/73961938/flask-sqlalchemy-db-create-all-raises-runtimeerror-working-outside-of-applicat>
 - Apparently, as of Flask-SQLAlchemy 3.0, all access to db.engine (and db.session) requires an active Flask application context. So, db.create_all() uses db.engine, so it requires an app context.
 - Here are three fixes:
 - Run the following in the Python script:
 - **with app.app_context():**
 - **db.create_all()**.
 - Instead of calling “create_all” in the script do it manually in the shell. Use **flask shell** to start a Python shell that already has an app context and the **db** object imported:
 - **\$ flask shell**
 - **>>> db.create_all()**
 - Or push a context manually if using a plain Python shell:
 - **\$ python**
 - **>>> from project import app, db**
 - **>>> app.app_context().push()**
 - **>>> db.create_all()**
 - The **third one** worked for me.
 - Now when you go to **pgAdmin**, the table is now present in the database.
 - At the moment the table has *zero entries*, so if we run **SELECT * FROM data** we get nothing but the column titles.
 -
 - In the next lecture, we’ll want to send some values to the database. So we’ll be running some SQL queries based on data entered into our web app.

Our app.py Code, So Far:

```
from flask import Flask, render_template, request
from flask_sqlalchemy import SQLAlchemy < < <

app=Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='postgresql://postgres:postgres123@localhost/height_collector' < < <
db=SQLAlchemy(app) < < <

class Data(db.Model):
    __tablename__="data"
    id=db.Column(db.Integer, primary_key=True)
    email_=db.Column(db.String(120), unique=True)
    height_=db.Column(db.Integer)

    def __init__(self, email_, height_): < < <
        self.email_ = email_ < < <
        self.height_ = height_ < < <

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        return render_template("success.html")

if __name__ == '__main__':
    app.debug=True
    app.run()
```

Storing User Data to the Database:

- So far, when we enter an **email** and **height** into our web app, we have it set up so that each of them is printed out in our console (for testing purposes).
- Now we want to take these two values and **send them to the database**.
- Our database/table currently has no values, and we can prove this by running the SQL query: **SELECT * FROM data**. This returns nothing but the column names.
- So let's go ahead and add some data.
-
- To be able to add rows with SQLAlchemy, you should point to the SQLAlchemy object **db** (in our case). To do this, we add this line (where our test print statements are):

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        db.session.add(email, height) ← ← ←
        db.session.commit() ← ← ←
    return render_template("success.html")
```

- This is a bit tricky, as it's not *values* we're passing, but something else. But let's try with values first and see what happens. We input “**email**” and “**height**”, which are two *plain Python strings* which were previously being printed out in the *console*. We also need to perform a **.commit()** method. Note: So far we've only had to use two lines of code to do what used to take us many more (MySQL-style).
- Let's go ahead and see what happens so far if we run our script.
 - So what happens so far is, we'll be sent to the **/success** URL by the **@route** decorator, which is defined in **index.html** in the **<form action="{{url_for('success')}}">** action-button. That will execute the **success** function in **app.py** which will try and send the data to the database.
 - However, when we try at this point, we get an **error**:
“sqlalchemy.orm.exc.UnmappedInstanceError: Class ‘builtins.str’ is not mapped.”
 - Note: If we had **app.debug=False** we wouldn't get this error.
 - When we *deploy* our application, we want to set this to False because we don't want the user to see these errors because that could become a **security issue**.
 - This error suggests that we need to **map** these strings (our input data) to something else, i.e. the **database model (db.Model)**.
-
- So we're going to start by creating an *object instance* of our Data class: **data=Data(email, height)**. Our Data class will take our values and create a **database model object** which will be recognized by the **.add()** method of the SQLAlchemy object:
-

- So we're going to start by creating an *object instance* of our Data class: `data=Data(email, height)`. Our Data class will take our values and create a **database model object** which will be recognized by the `.add()` method of the SQLAlchemy object:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        data=Data(email, height) ← ← ←
        db.session.add(data) ← ← ←
        db.session.commit()
    return render_template("success.html")
```

- Let's try inputting data into the webpage again.
 - Now we get sent to the **Success** page AND our *data gets input* into our **table**.
 - Next we want to send the **user's height** to the **user's email address**.
 - We also need to calculate an **average height** from the "height_" column of our table. Every time a user inputs a new height, this will be recalculated. The average height also needs to be sent to the user's email.
- However, before we add those things, let's make sure our application is running smoothly. Let's test and see what happens if we try and reuse the **same email address**:
 - We get an error: "**IntegrityError**", "**DETAIL: Key (email_)=(a@a.com) already exists.**"
 - So our *unique email constraint* is working like it's supposed to.
 - However, eventually we don't want this error to be displayed and break the user page. We'll add some error handling for it.
 - Looking at the error line "[SQL: 'INSERT INTO data (email_, height_)'" etc tells us that the error is happening near `data=Data(email, height) → db.session.add(data)`.
 - So what we need to do is to check the pair of values (**email, height**) against the table:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"] ← ← ← check these
        height=request.form["height_name"] ← ← ←
        print(email)
        print(height)
        data=Data(email, height) ← ← ← against these
        db.session.add(data) ← ← ←
        db.session.commit()
    return render_template("success.html")
```

- This means we need to apply some sort of ***conditional*** to the rows:
 -

- This means we need to apply some sort of ***conditional*** to the rows. Now, before we add the actual conditional statement to this, the instructor wanted to show us what the line he's adding for the conditional does on its own by *printing* it:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        print(db.session.query(Data).filter(Data.email_ == email)) ← ← ←
        data=Data(email, height)
        db.session.add(data)
        db.session.commit()
    return render_template("success.html")
```

- Running the code with this line added gives us the same webpage error so far (because we still haven't applied the conditional) as well as a *long traceback* printed to the **console**. Scrolling up through the traceback to find the *print* statement produces the **SQL query**:

```
SELECT data.id AS data_id, data.email_ AS data_email_, data.height_
FROM data
WHERE data.email_ = :email_1
```

- This is making a **query** to the database to search for the *existing email*. We can simplify our Python code for this by adding a **.count()** method:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        print(db.session.query(Data).filter(Data.email_ == email).count()) ←
        data=Data(email, height)
        db.session.add(data)
        db.session.commit()
    return render_template("success.html")
```

- Adding this **.count()** method produces the same webpage error, but now when we scroll up through the traceback message, this line prints out "1" denoting that there's 1 already-existing email that matches the one we just put in. If we print the **type()**, it tells us that it's a simple **integer**. We can make use of that fact to apply our ***conditional*** there.

- So we have one scenario where we'll get **1** (for an email that already exists in the database) and a scenario where we'll get **0** (when there *isn't* an email that already exists in the database). Let's build our **if-statement** off of that:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        if db.session.query(Data).filter(Data.email_ == email).count() == 0:
            data=Data(email, height)
            db.session.add(data)
            db.session.commit()
            return render_template("success.html")
    return render_template("index.html") ← ← ← reloads login page
```

- Note: The HTML of the webpage still reads as “localhost:5000/**success**” even if we enter an existing email.
- Our app is working almost perfectly now. However, this isn't the best **user experience** because the user might think that things are broken if they enter an existing email. They have no way of knowing what went wrong yet.
- So if the user adds an existing email, we could add a **red message** above the email input box. To do that, we're first want to add a **text attribute** to our lower **render_template()** function:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
        print(email)
        print(height)
        if db.session.query(Data).filter(Data.email_ == email).count() == 0:
            data=Data(email, height)
            db.session.add(data)
            db.session.commit()
            return render_template("success.html")
    return render_template("index.html",
                           text="Seems like you've input an already-existing email.") ← ← ←
```

- Now we need to define in **index.html** where that text will go, as well as its other attributes:
-

- Now we need to define in **index.html** where that text will go, as well as its other attributes. We're going to do that right above the `<form action="{{url_for('success')}}" button:`

```

<h1>Collecting Height</h1>
<h3>Please fill the entries to get population statistics on
height</h3>
<div class="message"> ← ← ←
    {{text | safe}} ← ← ← placeholder
</div>
<form action="{{url_for('success')}}" method="POST">
```

- Now when we input an existing email (**a@a.com**) again, we get:

The screenshot shows a dark teal header with the title "Collecting Height". Below it is a sub-header "Please fill the entries to get population statistics on height". A red error message "Seems like you've input an already-existing email." is displayed. There are two input fields: one for "Enter your email address" containing "a@a.com" and another for "Enter your height in cm". A blue "Submit" button is at the bottom.

- We can modify the style of this text in our **main.css** file. We add this to the bottom:

```
.message {
    font-family: Arial, sans-serif;
    font-size: 15px;
    color: #ff9999;
}
```

- The instructor noted that he found that color code online.
- Now when we reload the page and input an existing email again, we get:

The screenshot shows the same page as before, but the error message "Seems like you've input an already-existing email." is now displayed in a larger, red font. The rest of the interface remains the same.

- The font size and color of the warning message has been updated.
- In the next lecture we'll implement sending the user the results: their height and the average.

Gmail App Password:

From instructor:

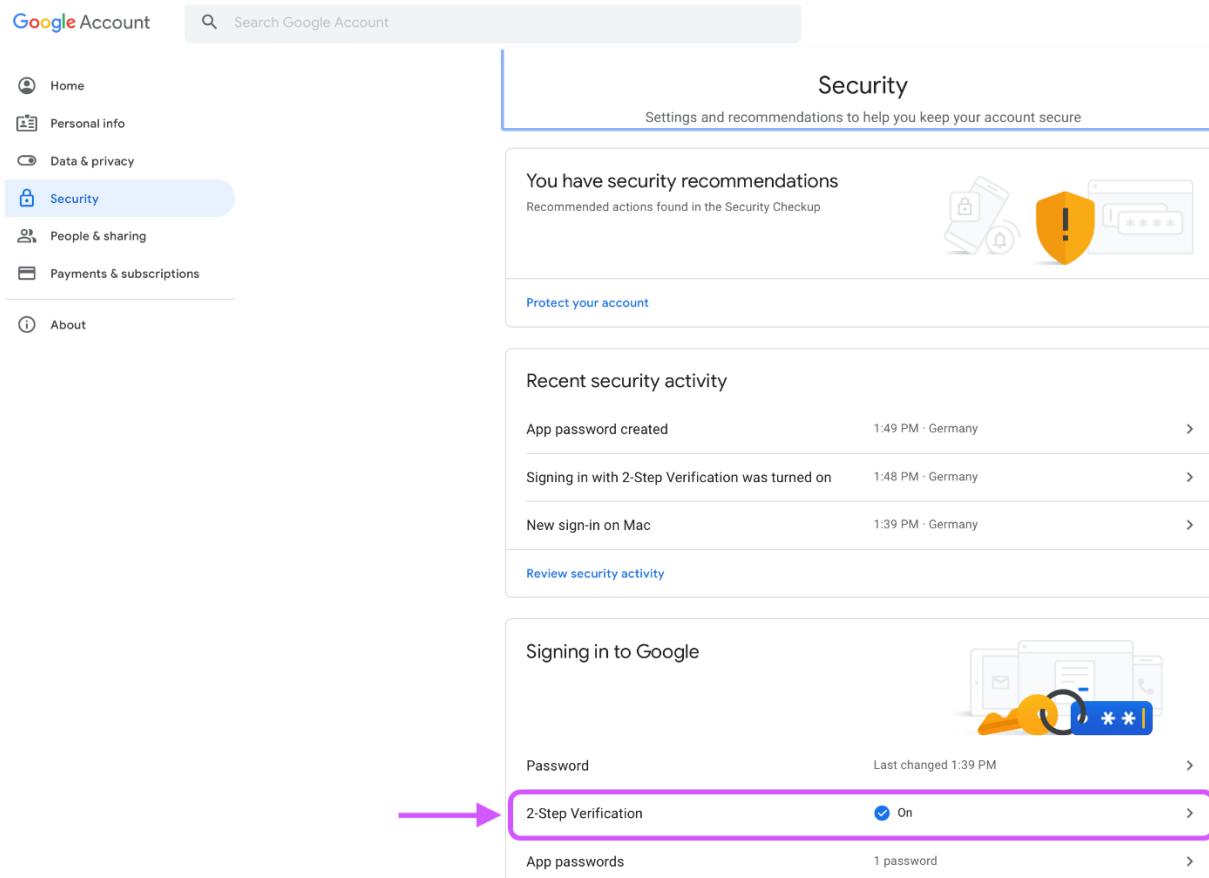
“Hi everyone!

In the following video, you will learn how to send emails to users via Python. We will use a Gmail address as the sender's email address, and we will use **the regular Gmail password** to authenticate. Please note that starting from **May 30, 2022**, Google does not allow apps to use the regular Gmail password anymore.

The solution is to use a Gmail **app password** instead. Please follow the steps below to create an app password for your Gmail account, so you are ready for the next video:

1. Go to the **Security** page on Gmail (direct link here):
<https://myaccount.google.com/u/3/security>

2. On the Security page, find and set up the **2-Step Verification** option. That option is a requirement for creating an app password.



The screenshot shows the Google Account Security page. The left sidebar has links for Home, Personal info, Data & privacy, Security (which is selected and highlighted in blue), People & sharing, Payments & subscriptions, and About. The main content area is titled "Security" with the subtitle "Settings and recommendations to help you keep your account secure". It features a section titled "You have security recommendations" with a shield icon and a "Protect your account" button. Below this is a "Recent security activity" section listing three events: "App password created" at 1:49 PM · Germany, "Signing in with 2-Step Verification was turned on" at 1:48 PM · Germany, and "New sign-in on Mac" at 1:39 PM · Germany. A "Review security activity" button is also present. At the bottom is a "Signing in to Google" section with a "Password" field last changed at 1:39 PM, a "2-Step Verification" field set to "On" (indicated by a checked checkbox), and an "App passwords" section showing 1 password. A purple arrow points to the "2-Step Verification" field.

- Once you set up 2-Step Verification, go to the Security page again and then go to App passwords which can be found just below the 2-Step Verification option.
- On the App password page, create an app password and choose Other (Custom Name) as the app and device options:

← App passwords

App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it. [Learn more](#)

The screenshot shows the 'Your app passwords' section of the Google Account security settings. It displays a table with one row of data:

Name	Created	Last used
Repl	1:49 PM	-

Below the table, there is a section titled "Select the app and device you want to generate the app password for." It includes dropdown menus for "Select app" (with options: Mail, Calendar, Contacts, YouTube, and Other (Custom name), where "Other (Custom name)" is highlighted) and "Select device". A "GENERATE" button is located to the right of the device selection dropdown.

- Press the Generate button, and you see your app password. That is the password (without the spaces) you should use in the Python script.

Emailing Database Values Back to User:

- At the start of this video, the instructor has 4 windows open: **Atom** (IDE), **Gmail**, **Outlook** (he still referred to it as “Hotmail”), and the **Command Line** (I’ll be using PowerShell).
- He’ll be using his **Gmail** account as the address from which the app will be sending emails.
- He’ll be using his **Outlook** account to input into our web app to test the application.
-
- Currently, our program is set to print out the email and height entered into the web app. We want to replace “print” with something like **send_email(email, height)**. This will be the name of a function we’ll create in just a minute.
 - We’re actually going to create an entirely new Python script (or **module**) in order to keep our code organized; we’ll call this **send_email.py**; it should be in the same directory level as **app.py**.
 - Inside of **send_email.py**, we’ll write a function: **def send_email()**.
 - Then at the beginning of **app.py** we’ll say **from send_email import send_email**:

```
from flask import Flask, render_template, request
from flask_sqlalchemy import SQLAlchemy
from send_email import send_email ← ← ← import function from module
```

- Now down in our **success** function in **app.py**, we’re using our *imported function*:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
    #
        print(email, height)
        send_email(email, height) ← ← ← use imported function
```

- One tip to remember: don’t name a module “email.py”. Then if you tried “from email import ...” it can cause problems because there’s also a *built-in module* in Python called “email”. By using “from email import ...”, you’re reserving this namespace for your *custom module*, but then if you try and run “import email” later, it can cause problems because the namespace is occupied.
 - In general, don’t name custom modules the same as built-in modules.
- Now we want to create our **send_email** function inside our **send_email** module:

```
import email

def send_email(email, height):
    from_email = "((email-here))@gmail.com" # test email
    from_password = "((password-here))" # app password
    to_email = email

    subject="Height data"
    message="Hey there, your height is %s." % height
```

- Note that the **message** will be sent as *plain text* at this point. If we want to make it **HTML-enabled**, we'll want to use **mime text**:

```
from email.mime.text import MIMEText < < <

def send_email(email, height):
    from_email = "((email-here))@gmail.com" # test email
    from_password = "((password-here))" # app password
    to_email = email

    subject="Height data"
    message="Hey there, your height is <strong>%s</strong>." % height < < <
```

- We can now use HTML code such as **** to make the height value come in bold.
- We also want to add:

```
msg=MIMEText(message, 'html')
msg['Subject'] = subject
msg['To'] = to_email
msg['From'] = from_email
```

- This converts the **message** variable to '**html**', sets the subject line of the email to **subject**, and sets the "**to**" and "**from**" addresses.
- Next we want to set up some **SMTP** (Simple Mail Transfer Protocol) stuff through Gmail. We'll have to **import smtplib** for this to work. We also want to **send the message** at this point:

```
from email.mime.text import MIMEText
import smtplib < < <

...
...
...

gmail = smtplib.SMTP('smtp.gmail.com', 587) < < <
gmail.ehlo()
gmail.starttls()
gmail.login(from_email, from_password)
gmail.send_message(msg) < < <
```

- The instructor noted that we also have to go to our Google account and set "**Allow less secure apps: ON**".
 - I'm not sure this option still exists, so I might be able to skip it, or it might've been handled by switching to using an "app password" (see previous lecture/note), or I might need to do some troubleshooting.
- Now we **run app.py** to see what we get.
- And it looks like we're successful:
-

- Now we *run app.py* to see what we get.
- And it looks like we're successful:



- So the email sending feature is working perfectly.
- In the next lecture we'll calculate the **average of all the heights** of users in the database, and we'll send that **average** to new users via this email system.

Our app.py Code, So Far:

```
from flask import Flask, render_template, request
from flask_sqlalchemy import SQLAlchemy
from send_email import send_email <--<

app=Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']='postgresql://postgres:postgres123@localhost/height_collector'
db=SQLAlchemy(app)

class Data(db.Model):
    __tablename__="data"
    id=db.Column(db.Integer, primary_key=True)
    email_=db.Column(db.String(120), unique=True)
    height_=db.Column(db.Integer)

    def __init__(self, email_, height_):
        self.email_ = email_
        self.height_ = height_

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
```

```

height=request.form["height_name"]
#     print(email, height)
send_email(email, height) ← ← ← ← ← ← ← ← ←
if db.session.query(Data).filter(Data.email_ == email).count() == 0:
    data=Data(email, height)
    db.session.add(data)
    db.session.commit()
    return render_template("success.html")
return render_template("index.html",
text="Seems like you've input an already-existing email.")

if __name__ == '__main__':
    app.debug=True
    app.run()

```

Our send_email.py Code, So Far:

```

from email.mime.text import MIMEText
import smtplib

def send_email(email, height):
    from_email = "((email-here))@gmail.com"    # sending email
    from_password = "((password-here))"    # app password
    to_email = email

    subject="Height data"
    message="Hey there, your height is <strong>%s</strong>." % height

    msg=MIMEText(message, 'html')    # sets to HTML code
    msg['Subject'] = subject    # subject line
    msg['To'] = to_email    # target email
    msg['From'] = from_email    # sending email

    gmail = smtplib.SMTP('smtp.gmail.com', 587)
    gmail.ehlo()
    gmail.starttls()
    gmail.login(from_email, from_password)
    gmail.send_message(msg)

```

Emailing Data Statistics to Users:

- This can be considered the *last lecture* about actually *building* our web application.
- We'll still have to **deploy our web application** on a **live server** so that we can visit that through a URL.
- So now we're going to **calculate the average height** of the user and **send it to their email address**.
- So to get all of the currently stored *heights* in our database, we'll need to **run a query** on the database. But **where should we run the query** in our program?
 - We want to **calculate the average**, and we want to **include the current user's height** in that average.
 - So we want to calculate the average just after the **db.session.commit()** statement.
 - We also need to add **from sqlalchemy.sql import func**:

```
from flask import Flask, render_template, request
from flask_sqlalchemy import SQLAlchemy
from send_email import send_email
from sqlalchemy.sql import func < < <
```

- Then we add the line calculating the average. We also want to **print** it right after to test that it's working:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
    #
        print(email, height)
        send_email(email, height)
        if db.session.query(Data).filter(Data.email_ == email).count() == 0:
            data=Data(email, height)
            db.session.add(data)
            db.session.commit()
            average_height = db.session.query(func.avg(Data.height_)) < < <
            print(average_height) < < <
        return render_template("success.html")
```

- The we submitted some *dummy data*, “**b@b.com**” and “**176**”, and what was printed out to the console was:

```
SELECT avg(data.height_) AS avg_1
FROM data
```

- So this isn't a number, which we probably weren't expecting. We need to do a trick here and use the **.scalar()** method:
 -

- So this isn't a number, which we probably weren't expecting. We need to do a trick here and use the `.scalar()` method:

```

        data=Data(email, height)
        db.session.add(data)
        db.session.commit()
→ → → average_height = db.session.query(func.avg(Data.height_)).scalar()
        print(average_height)
        return render_template("success.html")

```

- Now when we run dummy data “`c@b.com`” and “`174`” we get:
 - `167.33333333333333`, a float with quite a few decimal points. We probably don't want to send that many decimal points to the user, so we'll want to apply a **round function** and pass `1` as an attribute for “1 decimal place”:

```

        db.session.commit()
        average_height = db.session.query(func.avg(Data.height_)).scalar()
        average_height = round(average_height, 1) ← ← ←
        print(average_height)
        return render_template("success.html")

```

- A third dummy data entry gave us a resulting average height of `172.0` printed to the console.
-
- Now we want to send the **average height value** to the `send_email` function/module as well. We add `average_height` as an input variable in the function, and we add more text and another placeholder to our message. We also replace “height” after the message with a **tuple** containing “height” and “average_height”:

```

def send_email(email, height, average_height): ← ← ←
    from_email = "((email-here))@gmail.com"    # sending email
    from_password = "((password-here))"    # app password
    to_email = email

    subject="Height data"
    message="Hey there, your height is <strong>%s</strong>. Average height of
all users is %s." % (height, average_height) ← ← ←

```

- Now we're almost done, except that in `app.py`, our function `send_email(email, height)` is being called before we even check the database:

```

        send_email(email, height) ← ← ←
→ → → if db.session.query(Data).filter(Data.email_ == email).count() == 0:

```

- If we try and pass `average_height` in here, it wouldn't make much sense because it hasn't been calculated yet. We also don't want to send a user the average without getting their data as well.
- So let's take that statement and move it down to where our **test print statement** is instead:
-

- So let's take that statement and move it down to where our ***test print statement*** is instead:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"]
        height=request.form["height_name"]
    #    print(email, height)
        if db.session.query(Data).filter(Data.email_ == email).count() == 0:
            data=Data(email, height)
            db.session.add(data)
            db.session.commit()
            average_height = db.session.query(func.avg(Data.height_)).scalar()
            average_height = round(average_height, 1)
            send_email(email, height, average_height) ← ← ←
    return render_template("success.html")
```

- We may also want to add a variable **count** here:

```
average_height = round(average_height, 1)
count = db.session.query(Data.height_).count() ← ← ←
send_email(email, height, average_height)
```

- This counts how many height entries there are in the database. This is because the user might want to know that number to know how ***statistically significant*** the average is. To send that to the user, we'll just *add count to send_email* (both the function call here, and the module):

```
average_height = round(average_height, 1)
count = db.session.query(Data.height_).count()
send_email(email, height, average_height, count) ← ← ←
```

- And:

```
def send_email(email, height, average_height, count):
    from_email = "((email-here))@gmail.com"    # sending email
    from_password = "((password-here))"      # app password
    to_email = email

    subject="Height data"
    message="Hey there, your height is <strong>%s</strong>. Average height of
all users is <strong>%s</strong> and that is calculated out of
<strong>%s</strong> people" % (height, average_height, count)
```

- Now we should test our program to see if the added functionality is working.
- If we want to use the same email we used last time (say, someone without as many email addresses as me), we'll need to delete that email entry from the database first.

- Now we should test our program to see if the added functionality is working.
- If we want to use the same email we used last time (say, someone without as many email addresses as me), we'll need to delete that email entry from the database first.
- We do this in pgAdmin with the SQL command:
 - **`DELETE FROM data WHERE email_ = '({email-here})'`**
- We can double check that it's been deleted by running:
 - **`SELECT * FROM data`**
- Now we can go ahead and test the web app with that email again.
- When we run the test, we get an email:

Height data ➔ Inbox

 [REDACTED]test@gmail.com
Hey there, your height is 173.3:21 PM (1 hour ago) ⭐

 [REDACTED]test@gmail.com
to me ▾5:16 PM (0 minutes ago) ⭐ ↗ ⋮

Hey there, your height is **173**. Average height of all users is **172.0** and that is calculated out of **7** people

- And we're done with our application!!
-
- In the next lecture, we'll learn how to deploy our web app and our database onto a live server.

Deploying the Database Web App Online:

- We'll be deploying our app using **PythonAnywhere**. We can upload our files here, and we'll get a domain name where we can access the web app so that anyone can use our web app.
 - We can deploy apps for free here.
-
- **Web:** From the upper tabs, we're going to choose "Web".
 - Then we click "**Add a new web app**".
 - Looks like I still have the one from the last Flask app that we did. The free version of PythonAnywhere only allows one website, so I might as well delete that one.
 - In the "Create new web app" window, click "**Next**".
 - In the next page, we want to choose "**Flask**" as our *Python Web Framework*, and then "**Python 3.10**".
 - It gives us a default path of "**/home/((username))/mysite/flask_app.py**". The instructor said to just leave it as-is.
 - PythonAnywhere gives us a message that if there's an existing file with that name, it will be overwritten. Guess I'll just overwrite my old one, it wasn't very exciting anyway.
 - **Files:** Once our new domain is created, go back to the upper tabs again and choose "Files".
 - In "**Directories**" to the left, you'll notice that a directory called "**mysite/**" was created for you. Click on that and you'll see "**flask_app.py**", but we want to change that to our own code. This is the *entry point* to our web application.
 - So we want to copy all our code from our local **app.py** and paste it over the code in **flask_app.py**. Keep the file named "**flask_app.py**" though, or else things won't work.
 - Click "**Save**", then go back into **mysite/**.
 - Now we need to upload the other files: **send_email.py** into **mysite/**.
 - Next, we want to create the "**static**" and "**templates**" folders by typing these into the "**Enter new directory name**" field to the left.
 - Upload the **templates** files ("*index.html*" and "*success.html*") into the online **templates** directory and the **static** files ("*main.css*") to the online **static** directory.
 - Now we can try out the app as though we were a visitor to see if it works so far. But first we need to reload the app:
 - Go to "**Web**" again.
 - Click the "**Reload**" button.
 - Click the link of the web app URL above that. This brings us to the live version of our web app.
 - However, as it is, the web app isn't going to work; you'll get an **Internal Server Error**. This is because the database isn't hosted yet.
 - If you go to **mysite/** and the **flask_app.py**, you'll see that **app.config** is still connected to our **local PostGreSQL database**.
 - To fix this, we want to create a database inside the PythonAnywhere server.
 -
 -
 -

- **Databases:** In the upper tabs, choose “**Databases**”.
 - On this page, there are two choices to the left: “**MySQL**” and “**PostGres**”.
 - For some reason, we’re going with the **MySQL** one. It asks you to set a password, then you click “**Initialize MySQL**”. I set mine to “**\$DatabasePassword1**” because it’ll be easy to remember for adding to my code, and because I don’t really care about security here.
 - PythonAnywhere creates a database named “**((username))\$default**”, but we want to create our own database with the same name as our *local one*: **height_collector**.
 - The full name of the database will now be “**((username))\$height_collector**”.
 -
 - Once we have a database, we want to *create a table*. In our local test version with PostGreSQL, we had created/populated one with our *virtual environment Python integrated console*.
 - However, here we want to create the table using **SQL queries**. The place we want to run those queries is on this page, so we click on “**((username))\$height_collector**” to “Start console” on it.
 - We input into the console:

```
mysql> CREATE TABLE data (id SERIAL PRIMARY KEY, email VARCHAR(120), height_ INT);
```

- And we get “**Query OK, 0 rows affected**”. We’ve created our table.
- Now we enter “**exit**” to get out of this console, then hit “Back” in the browser.
- Next, we need to change something in our **flask_app.py** file.
 - We go in and delete the string:
“`postgresql://postgres:postgres123@localhost/height_collector`”.
 - We want to replace it with a string of the format:
 - ‘**mysql+mysqlconnector://((username)):((database password))@((database host address))/((database name))**’.
 - Database host address is: `((username)).mysql.pythonanywhere-services.com`
 -
 - After that’s set up, go to the upper tabs and go to “**Web**” and click the “**Reload**” button.
 - Once the web app is reloaded, we visit it again by [clicking the link above](#).
 - Ran some dummy data through it with multiple emails, both real and non-existent. The app seems to be working perfectly.
 - We can also go back to the “**Databases**” tab, open the **console**, and run **SELECT * FROM data** to check that all the rows are being populated:

```
mysql> SELECT * FROM data;
+---+ | id | email_ | height_ | +---+
| 1 | [REDACTED]@gmail.com | 173 |
| 2 | [REDACTED]@gmail.com | 180 |
| 3 | a@b.com | 150 |
| 4 | b@c.com | 200 |
| 5 | d@d.com | 184 |
| 6 | [REDACTED].test@gmail.com | 170 |
| 7 | [REDACTED]@outlook.com | 173 |
+---+
7 rows in set (0.00 sec)
```

- If anything went wrong, you'll want to go up to “**Web**” and check the **Error log**. This brings up any and all errors in a *traceback* format.
-
- In the next lecture, we'll be talking about creating a Download/Upload feature.

Creating a Download-Upload Feature:

Note: Attachment for this lecture includes “**Sample.csv**” file for uploading.

- This is sort of a bonus lecture because we’re not going to work on our web application or improve it further. However, it’s still related to web applications in general, particularly Flask web applications: this lecture will show us what to do when a user wants to upload or download files from a web app.
- We’ll start by learning how to add an **input** where a user can submit a file, and then how to handle that file in Flask, then how to let the user download the *product of that file*. So, **uploading, processing, and then downloading**.
- Included in this lecture’s attachments was a file called “**Sample.csv**” (he also had a “**Sample.txt**” on his screen, but that wasn’t included in the attachments).
 - Since he’s using **Atom** as his IDE, it knows to highlight the file names in **green** to show that they haven’t been uploaded to a Git repository yet.
-
- The first thing we need to do is *modify* our **index.html** file. For comparison’s sake, he brought up our webpage as it currently is; what we’re going to do is *replace the text entry* for **email address** with an **input for a file**, which he stated is quite easy.
- Our current file looks like:

```
<form action="{{url_for('success')}}" method="POST" >
    <input title="Your email will be safe with us" placeholder="Enter your
email address" type="email" name="email_name" required> <br>
    <input title="Your data will be safe with us" placeholder="Enter your
height in cm" type="number" min="50" max="300" name="height_name" required>
<br>
    <button type="submit"> Submit </button>
```

- We want to delete both inputs in this case and replace them with...

```
<form action="{{url_for('success')}}" method="POST"
encryption="multipart/form-data"> ← ← ←
    <input type="file" name="file"> <br> ← ← ←
    <button type="submit"> Submit </button>
```

- ...where the **input type** is “**file**” (as is the name if we want), and we set an **encryption type**.
- Now when we reload our webpage, it looks like this:



- You can see a “Choose File” button here now. Selecting this opens up a window, from which we can pass a **file input**.

- However, if we pass this file as-is, *we'll get an error*. This is because we still need to change our **app.py** Python code to be able to *handle data from a CSV file*.
- In our **app.py** file, we go down to where we have a **POST** method request (these POST requests send data to our web application). We'll still be changing code in our **success** function here.
 - What we want to change here is, instead of requesting **.form**, we want to request **.files** when setting the *email* and *height*:
 - Old:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        email=request.form["email_name"] ← ← ←
        height=request.form["height_name"] ← ← ←
```

- New:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"] ← ← ←
#        height=request.files["height_name"] ← ← ← (no longer needed)
```

- We're more-or-less just using our existing app.py code as a structure to build to.
- We can now delete all of the **database code (“db”)**:

```
26 @app.route("/success", methods=['POST'])
27 def success():
28     if request.method=='POST':
29         email=request.files["file"]
30         if db.session.query(Data).filter(Data.email_==email).count() == 0:
31             data=Data(email,height)
32             db.session.add(data)
33             db.session.commit()
34             average_height=db.session.query(func.avg(Data.height_)).scalar()
35             average_height=round(average_height,1)
36             count=db.session.query(Data.height_).count()
37             send_email(email, height, average_height, count)
38             return render_template("success.html")
```

- This gives us this:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        return render_template("success.html")
```

- What we're doing is checking the POST request, and if it's storing our “**file**” from the HTML inside another variable named “**file**”, then we **render_template** to **success.html**. We can also delete the other *return* line to keep things simple in this case; we'll handle duplicate emails another way.
- To see what we're doing, let's add a few *test print statements*:

- To see what we're doing, let's add a few test print statements:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        print(file) ← ← ←
        print(type(file)) ← ← ←
    return render_template("success.html")
```

- Now when we *reload our web application* and *upload the file*, we get:
 - <FileStorage: 'Sample.csv' ('application/vnd.ms-excel')>, and:
 - <class 'werkzeug.datastructures.FileStorage'> printed out to the terminal.
 - This second part is a **special storage** file-type. We can handle this as a *Python object* and apply the `.read()` method to it:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        content=file.read() ← ← ← storage in a variable, "content"
        print(content) ← ← ← print content
        #print(type(file))
    return render_template("success.html")
```

- Applying `.read()` to the file content gives us the data we got from web-scraping last section:

```
C:\Windows\system32\cmd.exe - python app.py
overhead and will be disabled by default in the future. Set it to True to suppress this warning.
  warnings.warn('SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled by default in the future. Set it to True to suppress this warning.')
* Debugger is active!
* Debugger pin code: 142-457-940
127.0.0.1 - - [28/May/2016 19:17:20] "GET / HTTP/1.1" 200 -
b'.Address,Area,Beds,Full Baths,Half Baths,Latitude,Lot Size,Price\r\n0,0 Gatewa
y,...,"Rock Springs, WY 82901",,$725,000 "\r\n1,1003 Winchester Blvd.,,4.4,"Ro
ck Springs, WY 82901",0.21 Acres,$452,900 "\r\n2,3239 Spearhead Way,"3,076",4.3
1,"Rock Springs, WY 82901","Under 1/2 Acre, ",,$379,900 "\r\n3,600 Talladega,"3
,154",5.3,"Rock Springs, WY 82901",,$379,000 "\r\n4,3457 Brisol Avenue,"3,236"
5.3,"Rock Springs, WY 82901",0.34 Acres,$349,900 "\r\n5,234 Via Spoleto,"2,68
8",4.3,"Rock Springs, WY 82901","Under 1/2 Acre, ",,$330,000 "\r\n6,2425 Cripple Creek,"8,263",4.35,"Rock Springs, WY 82901",,$229,900 "\r\n7,522 Emerald Street,"1,172",3.3,"Rock Springs, WY 82901","Under 1/2 Acre, ",,$254,000 "\r\n8,13
02 Veteran's Drive,"1,932",4.2,"Rock Springs, WY 82901",0.27 Acres,$252,900 "
\r\n9,343 Via Rucco,,3,2,1,"Rock Springs, WY 82901",0.16 Acres,$219,900 "\r\n10
,913 Madison Dr,"1,344",3,2,"Rock Springs, WY 82901","Under 1/2 Acre, ",,$209,0
00 "\r\n'
<FileStorage: 'Sample.csv' ('application/vnd.ms-excel')>
<class 'werkzeug.datastructures.FileStorage'>
127.0.0.1 - - [28/May/2016 19:17:24] "POST /success HTTP/1.1" 200 -
```

- Note that this is just an example of uploading/reading a file; this data won't work for the current script because we need an **email** and a **height**. Now for writing a file:

- To do that, we'll need to save our file to our directory first using the `.save()` method. We also use the `.filename` to name it (same name). Since this is a string, we can concatenate it with "uploaded_". This will be saved to the same directory:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        file.save("uploaded_" + file.filename) ← ← ←
        return render_template("success.html")
```

- In this case, "uploaded_Sample.csv" will be the same as "Sample.csv" when opened.
- Something to know: it's not safe to use the `.save` expression. It's better to also add "`secure_filename()`":

```
from werkzeug.utils import secure_filename
...
...
...
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        file.save(secure_filename("uploaded_" + file.filename))
        return render_template("success.html")
```

- Note: This keeps users from using "/" in file names to track down your server root and injecting harmful bash scripting.
- Now that we've saved a renamed copy of our file, let's **write** to it:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        file.save(secure_filename("uploaded_" + file.filename))
        with open("uploaded_" + file.filename, "a") as f: ← ← ←
            f.write("This was added later") ← ← ←
        return render_template("success.html")
```

- Now when we check our file, the new line is written in the lowest cell:

```
9  7,522 Emerald Street,"1,172",3,3,,"Rock Springs, WY 82901","Under 1/2 Acre, ","$254,000 "
10 8,1302 Veteran's Drive,"1,932",4,2,,"Rock Springs, WY 82901",0.27 Acres,$252,900 "
11 9,343 Via Rucce,,3,2,1,"Rock Springs, WY 82901",0.16 Acres,$219,900 "
12 10,913 Madison Dr,"1,344",3,2,,"Rock Springs, WY 82901","Under 1/2 Acre, ","$209,000 "
13 This was added later! |
```

- Now let's see how you *let the user download* your files.
- So instead of our “/success” page, we’re going to *render a download button* below the “submit” button. We’ll still be rendering the index.html page, but we’ll be inserting a download button. So how do we do that?
- For starters, we pass “index.html” into our **render_template**, as well as a “**btn**” parameter pointing to a new “**download.html**” that we’ll create:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        file.save(secure_filename("uploaded_" + file.filename))
        with open("uploaded_" + file.filename, "a") as f:
            f.write("This was added later")
    return render_template("index.html", btn = "download.html") ← ← ←
```

- We create the following “**download.html**” script:

```
<!DOCTYPE html>
<html lang="en">
    <div class="download">
        <a href="{{url_for('download')}}" target="blank"> <button class="btn">
Download </button></a>
    </div>
</html>
```

- When we press the button named “Download”, the code will execute the Python function “**download**”, which we haven’t created yet.
- We also still haven’t told our “index.html” page about our download button. We do this in a **jinja** tag under the **</form>** line:

```
</form>
{% include btn %} ← ← ←
</div>
```

- However, if we reload our webpage now, we’ll get an error saying “**jinja2.exceptions.TemplatesNotFound: Tried to select from an empty list of templates**”. So why is Python not able to find a template?
- The reason this happens is because when we reload the webpage, what we’re visiting is “index.html” (URL “/”), and the **def index** function is executed:

```
@app.route("/")
def index():
    return render_template("index.html")
```

- So it's trying to render the "index.html" page, but when you look at the script for index.html, the index.html page can't find the empty template "btn". This is because the **btn** field isn't being passed in **def index**, it's being passed at the end of **def success**:

```
@app.route("/success", methods=['POST'])
def success():
    if request.method=='POST':
        file=request.files["file"]
        file.save(secure_filename("uploaded_" + file.filename))
        with open("uploaded_" + file.filename, "a") as f:
            f.write("This was added later")
    return render_template("index.html", btn = "download.html") ← ← ←
```

- So we need to find a way for the index.html code to *ignore* the **btn** variable we put in the jinja tag. To do that, we simply add "ignore missing":

```
</form>
{% include btn ignore missing %} ←←←
</div>
```

- Note: Had to do some troubleshooting because this wasn't working. Found that someone had previously asked about this on Stack Overflow, and it even seemed like they were someone in the same class. (<https://stackoverflow.com/questions/63750379/jinja2-exceptions-undefinederror-btn-is-undefined>). The fix was:

```
</form>
{% if btn %}
    {% include btn %}
{% endif %}
</div>
```

- So now when the user presses the "Submit" button, the **def success** function is executed, and the "index.html" template is rendered again, this time with a "download.html" button!
- However, now when we try and load a file and hit "Submit", we run into another error: **"werkzeug.routing.exceptions.BuildError Could not build url for endpoint 'download'. Did you mean 'index' instead?"**
 - This is because when Flask tries to run the download.html template, it tries to access the **download** function. But we haven't written it yet.
 - Let's get past this problem for now by just writing a *dummy function*:

```
@app.route("/download")
def download():
    pass
```

- This will trick Flask into thinking we have a real **def download** function it can access.
- Now when we run it, the **"Download"** button pops up right under "Submit":
-

- Now when we run it, the “**Download**” button pops up right under “Submit”:



- Now when the user presses “**Download**”, then the **download** function will be executed.
- We’ll also need to **import send_file** from Flask to use in our function:

```
from flask import Flask, render_template, request, send_file
...
...
...
@app.route("/download")
def download():
    return send_file("uploaded_" + file.filename,
attachment_filename="yourfile.csv", as_attachment=True)
```

- However, we currently can’t access the variable **file** because it’s a *local variable* inside of the success function. To fix this, we need to add a **global** variable callout inside of success:

```
from flask import Flask, render_template, request, send_file
...
...
...
@app.route("/success", methods=['POST'])
def success():
    global file < < <
    if request.method=='POST':
        file=request.files["file"]
        file.save(secure_filename("uploaded_" + file.filename))
        with open("uploaded_" + file.filename, "a") as f:
            f.write("This was added later")
        return render_template("index.html", btn = "download.html")

@app.route("/download")
def download():
    return send_file("uploaded_" + file.filename,
attachment_filename="yourfile.csv", as_attachment=True)
```

- Note: I kept getting an error when trying to download (“**flask send_file()** got an unexpected keyword argument ‘attachment_filename’”). This is because “attachment_filename” was replaced by “**download_name**”:

- Note: I kept getting an error when trying to download (“**flask send_file() got an unexpected keyword argument 'attachment_filename'**”). This is because “attachment_filename” was replaced by “**download_name**”:

```
@app.route("/download")
def download():
    return send_file("uploaded_" + file.filename,
download_name="yourfile.csv", as_attachment=True) < < <
```

- Opening our downloaded data shows us that it's the same data but with the extra line added at the end.

Section 32: App 9: Django & Bootstrap Blog and Translator App:

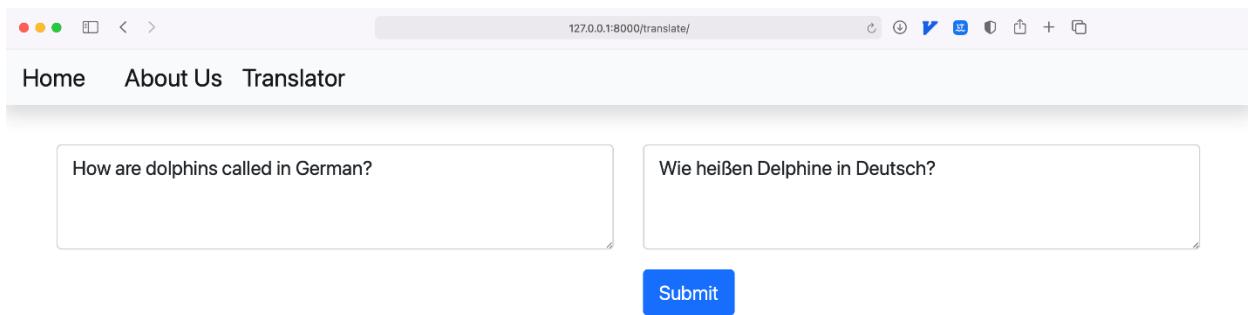
Demo of the App:

Welcome to a new Python app! This one will be a Django website which has two features. It has a blog feature with an admin interface through which content creators can write and publish new blog posts to the website. The website also has a translator app where people can translate text from one language to another. Here is how the website will look like after you code it:

The screenshot shows a web browser window displaying a Django-based blog application. The URL in the address bar is 127.0.0.1:8000. The page has a header with navigation links for Home, About Us, and Translator. Below the header, there are three blog post cards, each with a title, a brief description, the author's name (Ardit), and a 'Read More' button.

- Dolphins**
Dolphin is the common name of aquatic mammals within the infraorder Cetacea. The term dolphin usually refers to the extant families Delphinidae (the oceanic dolphins), Platanistidae (the Indian river dolphins), named Iniidae (the New World river dolphins), and Pontoporiidae (the brackish dolphins), and the extinct Lipotidae (baiji or Chinese river dolphin). There are 40 extant species named as dolphins. Dolphins range in size from the relatively small 1.7-metre-long (5 ft 7 in) long and 50-ki
Author: Ardit
[Read More](#)
- Armadillos**
They are prolific diggers. Many species use their sharp claws to dig for food, such as grubs, and to dig dens. The nine-banded armadillo prefers to build burrows in moist soil near the creeks, streams, and arroyos around which it lives and feeds. The diets of different armadillo species vary, but consist mainly of insects, grubs, and other invertebrates. Some species, however, feed almost entirely on ants and termites. Paws of a hairy and a giant armadillo Armadillos have very poor eyesight
Author: Ardit
[Read More](#)
- Frogs**
A frog is an anuran of the suborder Ranae, characterized by its large head, short body, and long hind legs, which are adapted for leaping. Frogs are typically terrestrial, but some are arboreal, and others are semi-aquatic. They are found in all continents except Antarctica. Frogs are often considered to be indicators of environmental health, as they are sensitive to changes in their surroundings. They play a vital role in the ecosystem, serving as both predators and prey for many other organisms.
Author: Ardit
[Read More](#)

And here is how the finished translator tool:



Let's get started!

A Comparison of Python Web Frameworks:

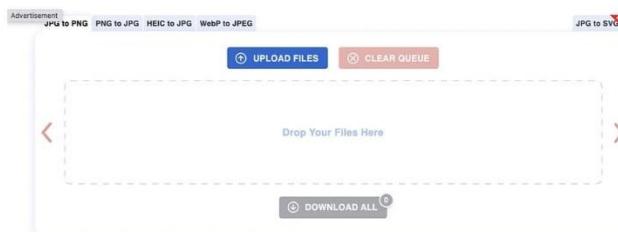
- What is a web framework?
 - One could code just in straight Python.
 - Or, one could build a web application using a **web framework**, which are essentially *Python libraries* that you install with pip. These web frameworks make it easier to write web applications using Python.
 - Creating a web app without using one of these web frameworks is like *reinventing the wheel*.
 - You *should* use a web framework.
- Which web framework should you use?
 - There are many web frameworks for Python, but we're only going to focus on three of them:
 - **Flask**
 - **Django**
 - **JustPy**

Flask:

- **Flask** is a web framework usually targeted for building *small web applications*.
 - For example, let's say you just want to build a small translator that translates from one language to another. Think of "Google Translator".



- Another example would be an app that converts *image file types*, such as "JPG to PNG", "PNG to JPG", "HEIC to JPG", "WebP to JPEG".
 - In this case you might also need to install an image processing library like OpenCV/cv2.
 - However, actually *serving* the web app will be done with Flask:



- You can, of course, still create *bigger* websites with Flask. You can have a blog with multiple entries, and all of these will have to be saved to a database. Flask can do this, but Django would do this better.

Django:

- By comparison, Django has a better level of *abstraction*. While a blog/database could be done with Flask, you have to do *more routine tasks*.
- Django also allows you to add **more apps** to that website project.
 - In the example of a blog, perhaps you also want to add a translator app. We can build a translator on top of the existing website.
 - However, if we just want a translator website alone, Django would be overkill for this. Django projects take more work to set up in the beginning.

JustPy:

- JustPy is a very new framework. It's also a very special case.
 - With Flask and Django, you also need to know HTML and CSS to be able to build web apps. These are the building blocks of a *front-end*.
 - JustPy allows you to build web apps without a single line of HTML or CSS. It has some specific **Python objects** instead, such as **div** objects or **buttons** objects.
 - The downside to this is that you're not very flexible doing this compared to using Flask and Django.
 - JustPy is also better suited for *small web applications*.
 - It also doesn't work as well with *databases* as the other two.
-
- If you want to purely be a **web developer**, it's useful to learn both Flask and Django.
- However, if for example you're mainly interested in **data science**, then you could use a simple web app designed in JustPy to present your data on a website.

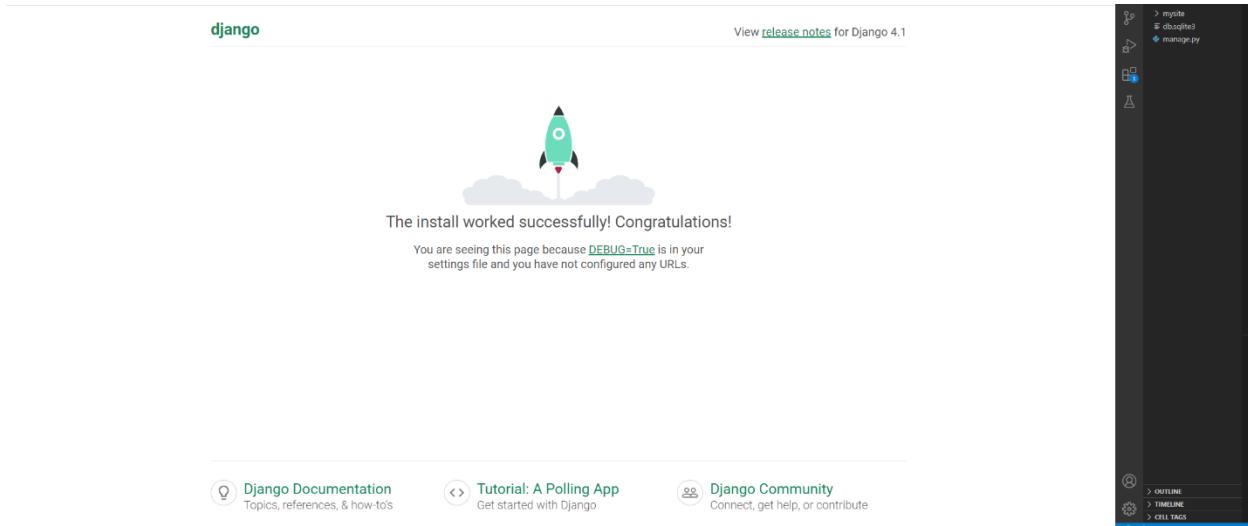
Setting Up a Virtual Environment:

- A Python virtual environment is just a ***copy of your installation***, or a ***copy of your Python interpreter***.
-
- Why do we make a copy?
 - With Python, we might need to write different programs. And for each of those programs, we might need to install *different packages*, or *different libraries*.
 - Usually, we just install these libraries in our *main Python installation* (also known as a “*global interpreter*”).
 - But by making a copy of each program that we built, then we only have to install libraries for *that copy*.
 - The benefit of this, especially for web applications, is that we’ll likely deploy them later to online servers.
 - When you make a copy of the interpreter and then you install the libraries that you need on that copy/virtual environment, then you have a clear list of all the libraries that you have installed.
 - You can generate this list, and then you can give the list to the server when you deploy your web application, so that the server can take the list of libraries and install all of the libraries.
 - The benefit here is that if you used your global Python installation, you’d have many, many libraries to deal with. The server would have to install *all of them*, which just isn’t a good idea. Why not have a separate interpreter for every project?
 - Now that we’re going to create a **Django** app, we’re going to create a virtual environment first.
-
- We start by ***opening an empty folder*** we created for this project in VSCode (I named mine “Django_Blog_Translator”).
- We then ***open a terminal***. We run a quick check by entering *our* version of Python in the terminal line (he typed “**Python3.9**”, I ran “**Python3.10**”) in order to open the integrated interpreter command line. We ran **exit()** right after since we’re just testing.
- Next, we run **`python3.10 -m venv env`** (for Windows) or **`python -m virtualenv virtual`** (for Linux, Mac) to set up our virtual environment.
 - The **-m** flag tells Python that we’ll be executing the following library **venv**, and then **env** is the folder where the virtual environment will be placed.
- Once that’s installed, we want to select a *default Python shell* for this particular VSCode project. We can do this with shortcuts:
 - On Mac: **Cmd + Shift + P**
 - On Windows: **CTRL + Shift + P**
 - This will cause a box to appear at the top of VSCode; search for “**select interpreter**”. We want to choose the one with “env: venv” in the name and “env” in the filepath.
 - Now when we close/refresh our terminal, the entry line now starts with **(env)** before anything else. This means that our chosen Python virtual environment is being used, so if we simply enter **python**, we get the interactive shell with the Python version that we selected from our virtual environment folder.

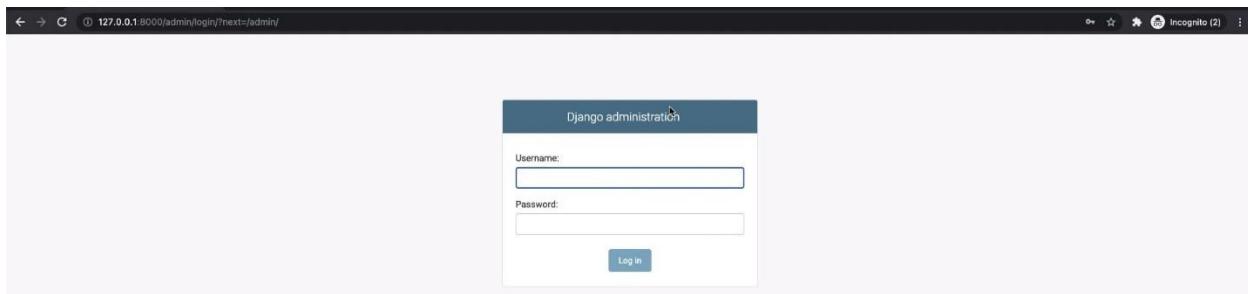
- Note: When creating a new file at this stage, be careful where it's being placed. He created a quick test file called "hello.py", and it got put in the "env" folder. We don't want to mess with the "env" folder.
 - To avoid placing the file inside the "env" folder, we first want to press **esc** to take the focus off of the "env" folder.
-
- To wrap it up, the virtual environment was used when we:
 - Opened the Python interactive shell.
 - Executed our test Python file.
 - And it's also used when we install third-party packages.
 - So let's install Django!
-
- Run **pip install django**.
- Now Django will be installed for *this particular installation of Python*, not for our Global Python.
-
- Now, if we close this project and then go back to it again, you may have to close the current terminal and open a new one in order to have our virtual environment active in the terminal and get the **(env)** at the front of the line again.
-
- Next up, we'll begin creating our Django project.

Creating a Django Project:

- When you create a *web app* with **Django**, first you want to create a **Django project**.
 - This is a set of files which are generated automatically by Django.
 - Then you can **add apps** to this project. So you'll have *one project*, but *several apps*.
 - In our case, we're going to have one project, but then also add the **blog app** and the **translator app**.
-
- To start a project, we'll first need to **install Django** and then also (preferably) set up our **virtual environment**. We've already done this so far.
- So far in our *root directory* (our app's main directory), we have our *env folder* and a test "hello.py" file. We can delete "hello.py" because we don't need it.
- So, from our *virtual environment's terminal* we run:
 - **django-admin startproject mysite .** :
 - We're using the Django command "**startproject**" and we're passing a name for the folder this will be stored in, "**mysite**".
 - The dot ". " at the end means that the **current folder** ("Django_Blog_Translator") is the **project folder**.
 -
 - Running this command creates two things in the project folder:
 - **mysite** folder.
 - **manage.py** file.
 -
 - The **mysite** folder contains:
 - An empty **__init__.py** file. We can place things inside here if we want them to be executed at the beginning of running the app, when the app starts.
 - An **asgi.py** file: this is a configuration file that we may want to change later on when we deploy the app on an ASGI server, which is one type of server. The other type of server is WSGI. ASGI: "Asynchronous Server Gateway Interface".
 - Per a quick Google search: "Unlike WSGI, ASGI allows multiple, asynchronous events per application".
 - An **wsgi.py** file: a configuration file we may want to change if we deploy on a WSGI server. WSGI: "Web Server Gateway Interface".
 - A **settings.py** file. This will contain all of the settings, such as the time zone, the place of the static files (static URL, /static/) such as images and CSS, etc.
 - A **urls.py** file. This will have the URLs of the project, how the URLs will be rooted.
 - **manage.py** is a file we don't normally change. This is useful when you start a new Python website: we can start the app by running (in our env terminal):
 - **python manage.py runserver**, and we get a warning at first, saying:
 - "**You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, content types, sessions. Run 'python manage.py migrate' to apply them.**" We're going to fix that in just a bit.
 - We also get the website's URL : **http://127.0.0.1:8000/**. If we go to that URL, we get our starter website:

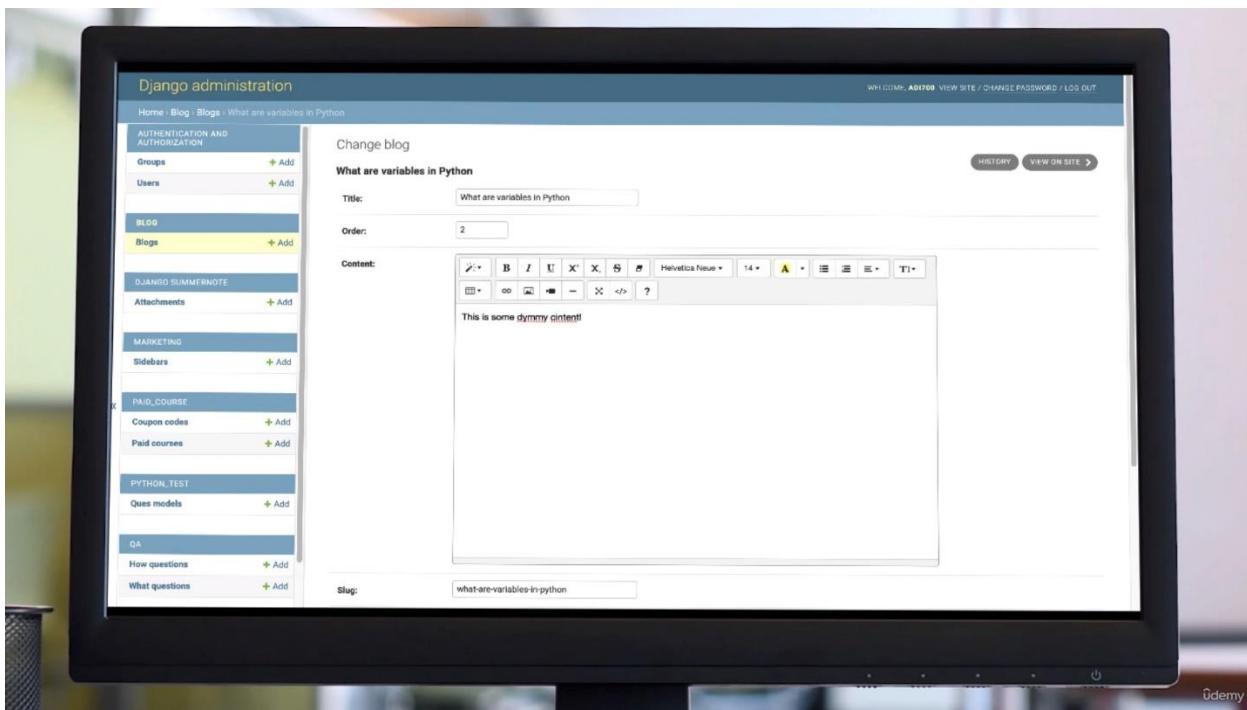


- The website doesn't have any custom apps running at the moment, but this tells us that it's working. “**CTRL + C**” will quit out of the app.
- Now back to that error message from earlier. All we have to do here is run the suggested command **python manage.py migrate**, and Django will execute the necessary ASQL commands for us. These are queries that create some *default database tables* for us.
- You'll also notice that a new **db.sqlite3** database has been created. Django uses sqlite3 by default, but you can also set it to use PostGreSQL on the production server. Locally, it's good to use sqlite3.
- We may want to change/create tables later on, and it's easier for us to do so using Python. When we make changes, we'll re-run the command **python manage.py migrate** to translate the Python code into SQL queries.
- Next up, the instructor decided to show us what tables were created by the command. He used a program called “**DB Browser for SQLite**” to open the sqlite3 file. Opening the sqlite3 file in this program lists the current tables. Currently, most of these tables are empty.
- One of these tables was “**django_admin_log**”. So what is this? To answer this, he started the server again with **python manage.py runserver**, then went up to the URL bar and typed “**http://127.0.0.1:8000/admin**”. This leads to a login page:



- Currently we don't have an admin user created yet. There isn't anything in the table for that yet. But why do we need an admin interface anyway?

- We said that we were going to build a website that contains blogs. Who's going to add content to the blogs? You, the developer? Authors?
- There are two ways to write content:
 - Either they go to your VSCode project and manually change files that will write the content, or:
 - A smarter way is to have a *friendlier interface*, where you have *content boxes*, and *options to change the font*. You'll have buttons, tools, toolboxes, and things like that. A real word processing app similar to Microsoft Word. This will make it easier for authors to add content to the website.
 - You would give authors an **admin password** so they can log in and post content.
 - We're going to look at the admin interface in the next lecture.
- Example of friendlier interface, from the lecture:



Creating a Superuser for the Project:

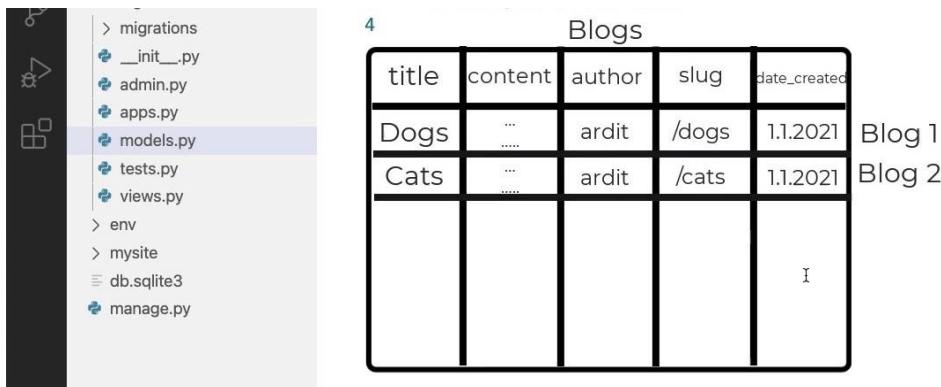
- In the previous lecture, we ran commands to start our Django project, and a bunch of starter files were automatically created for us. We also ran our website locally, including showing our admin interface.
- Now we want to create a *new user* to login to the admin interface.
- If we currently have our app running, we want to quit with **CTRL + C** to get back to our **env** terminal line first.
- Next we want to run: **python manage.py createsuperuser**.
 - It prompts us for a **username**, an **email address**, a **password** (repeat). If the password is too short, you'll get some warnings, but you can ignore those by typing **Y**. After this, you should get a message saying "Superuser created successfully".
- Once we've created our superuser, we can run **python manage.py runserver** again to start our webpage.
- Now if we go to "**http://127.0.0.1:8000/admin**", we can log in with the username and password that we just created. Once we've logged in, we get this screen:



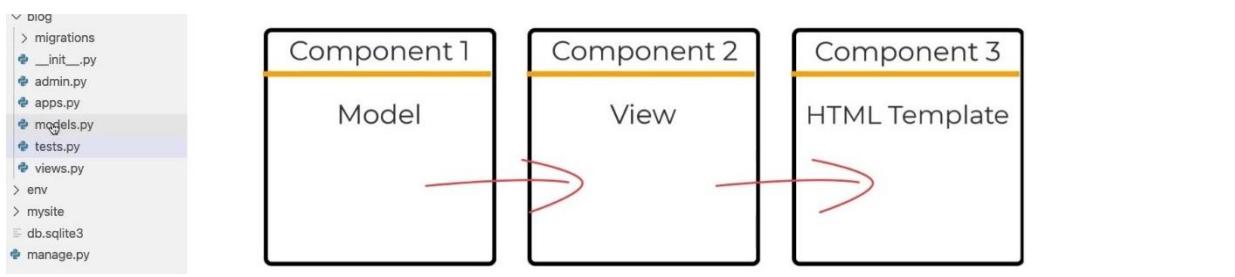
- Note that we have options to add **groups** and **users** now. So for example, in our blog application, if you want to *hire an author* to create more content for the blog, we'll want to create a new username for them. You create a username and password, then send them their credentials.
- In this same "Django administration" panel, later on when we create our blog app, we'll see other sections here that allow the admins to create new blogs/content to add to the website.
-
- In the next lecture, we're going to create the **blog** app to add to our Django project.

Setting Up an Empty Django Blog App:

- In this lecture we're going to create our **Blog App**. First we need to create an **empty app**, then we need to *add code to that app*.
- We need to start by creating a **structure** for our app by running:
 - **python manage.py startapp blog**, where "blog" is the chosen **name** for our app.
 - Running this line creates a new **folder** in our root project directory, "**blog**".
- Inside of this new "**blog**" app folder, we see **multiple files** and also a "**migrations**" directory.
 - "**migrationsmigration** is a **change** in the **database**, so this will handle the execution of SQL queries for us. The code that actually makes changes to the SQL database will be "**models.py**".
 - Inside "migrations" is an "**__init__.py**" file specific to "migrations".
 - "**__init__.pyautomatically execute when the app starts**.
 - "**admin.pyadmin interface**.
 - "**apps.pyconfiguration file** for your app, which you also don't want to change.
 - "**models.pythree components** from a Django perspective. The **first component** is the **model**: a **model** contains the **database fields** of that webpage, such as "**title**", "**content**", "**author**", "**slug**", "**date_created**", etc:



- "**views.pynext component** is the **view**. A view could be either a **Python function** or a **Python class**. It acts as a **middle man** between the **model of the webpage** (a blog post in our case) and **the HTML code**, or **template**.
 - The **model** is just a **database table**, and to serve this table to a **browser**, you need **HTML code** as well:



○

- “**tests.py**”: This is to write **tests** for testing the app once it’s ready. You want to make sure everything is working properly before putting it into **production**.
- So that covers our “**blog**” folder, but we need to do one more thing before we end this lecture.
-
- We need to go into our “**mysite**” folder and make a change inside the **settings.py** file.
 - If you scroll down inside that file, you’ll see a list labeled “**INSTALLED_APPS**”. These are all **default apps** required by Django, so we don’t want to change the existing ones.
 - However, after the last one in the list, there’s a **comma** afterwards. We want to press enter after that comma and add a new line, ‘**blog**’:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog', ↵ ↵ ↵
]
```

- Note that the name here is the same as the **directory/folder name** for the app, “**blog**”.
- This tells Django about the new app that was added.

Creating a Database Model for the Blog App:

- In this lecture, we're going to add some `code` to our `model.py` file (inside of our “`blog`” folder).
- Inside this, we're going to create a `class Post(models.Model)`.
 - This isn't a simple class, it inherits from the `models` class provided by Django, making this a **Model** class.
 - **Model** is a specific class created by the Django authors, and it's designed to **contain fields of data**.
 - Note that when we use the `Post` class anywhere, a **new table** will be created in our SQLite3 database.
- Inside this `class Post`, we create variables for our table contents:
 - `title = models.CharField(max_length=200)` sets a post's title and limits it to 200 characters.
 - `content = models.TextField()` stores the written content of the post.
 - `date_created = models.DateTimeField(auto_now_add=True)` gives each post a time tag when it gets posted.
 - `slug = models.SlugField(max_length=200, unique=True)` gives a “slug” field, or the section of the URL after the slash after the main page. We also limit it to 200 characters and we ensure that it's unique.
 - `author = models.ForeignKey(to=User, on_delete=models.CASCADE)` creates an author field. It also sets the behavior to “cascade” if the author is deleted, meaning that if an author is deleted, all of the author's blog posts will also be deleted.
 - For this one we also needed to add `from django.contrib.auth.models import User` to the top of our `models.py` file.
 - `status = models.IntegerField(choices=STATUS, default=0)` gives us the status of the post. On the `admin interface`, there will be a “status” dropdown list which will contain two options: “draft” and “publish”. So the content creator could save their article as either “draft” or “published”. This would make it easier for content creators to organize blogs because they could save articles as drafts, and then in our Django app we could make a conditional note to publish such articles that have their “status” as “publish”.

- We also have to create our own variable **STATUS** above our class, setting it to a tuple of two tuples: **STATUS = ((0, 'Draft'), (1, 'Publish'))**. We set the “**default**” field in our **status** variable to **0** for “Draft”.

```
from django.db import models
from django.contrib.auth.models import User

STATUS = ((0, 'Draft'), (1, 'Published'))
# Create your models here.

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
    slug = models.SlugField(max_length=200, unique=True)
    author = models.ForeignKey(to=User, on_delete=models.CASCADE)
    status = models.IntegerField(choices=STATUS, default=0)
```

-
- Once we have our **class Post** finished, we want to be sure to **save** the **file**, and then we want to **apply it to our database**. We'll need to run some SQL queries to do this, but remember that we don't need to do this explicitly, we just need to run:
 - **python manage.py makemigrations**. Running this will add some files and folders to our **blog/migrations** folder, and the terminal lists one of them: “**0001_initial.py**”.
 - Next we need to run:
 - **python manage.py migrate**. This will **apply SQL queries** based on the “**Post**” **model**. Now if we look at our **db.sqlite3** file in DB Browser and hit “refresh” in the **second tab**, we'll see that a table “**blog_post**” has been added, containing the fields from our model.
-
- That completes our section about making a **model**.
- In the next lecture, we'll work on creating a **view**.

Overview of the Web App Architecture:

- We won't be doing any coding in this lecture, but we will be learning the architecture of a Django web app.

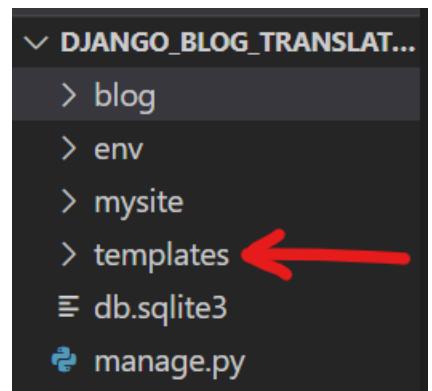


- So far we've learned how to **create an empty Django project**, and we've learned about the **models.py** file, but to better understand what's going on, it's time to look at the bigger picture and how a **Django web app** is **structured**.
-
- **User Interface/Experience:** Let's start by talking about the **web app from the user's perspective** ("example.com/dogs" above). The user may see a **title**, **content**, **author**, and the **date**. The users see this, but as developers, we see the **HTML template**.
- **HTML Template:** The above "**HTML Template**" file is rendered by the browser and translates to the user's view of the webpage. Within the HTML template, we'll work with both **HTML tags** (those inside "`<>`") and **Django tags** (those inside curly brackets, or "`{ }`").
 - Inside of these curly there will be a "**title**" variable, a **content** variable in another tag, an **author** variable, a **date** variable, etc. Now what are these variables, the **title**, the **author**, etc.? These will come from the **database**, "db.sqlite3" inside the **blog_post** table.

- **Database/Tables:** Each row inside of the **database's blog_post table** corresponds with one **blog post**. So for the post “Dogs”, the **title** value “Dogs” (and the other values: **content**, **author**, **date**, **slug**) will be injected into the **Django tag** in the **HTML template**.
- **models.py:** Next, we looked at **models.py** already, where we saw that we should place some **classes** there which define the **structure of the database table** (the variables such as “title” and “content” correspond to the columns of the table). We’re not providing any particular values for the rows of the tables at this time, we’re just defining the structure.
- **urls.py:** Next we have the **urls.py** file. This is the file that takes care of the **URL rooting**. Django will look at this **urls.py** file and see that the URL is, for example, “example.com/<slug>”, where “slug” is another field from our **database**.
 - Say the user entered “/dogs” at the end of the URL: this **urls.py** file will look for the “/dogs” slug among the database table’s rows.
 - If it finds one, then it **notifies the corresponding view**.
 - The **urls.py** file will have **two arguments**: the **URL Pattern** (“example.com/<slug>”), and **views.Blog**.
 - This argument **views.Blog** points to a **class** within the **views.py** file.
- **views.py:** This class will take care of the user request. In our example, there is a **class BlogView** and inside that is a variable **model = Post** and a variable **template=“blog.html”**. In some cases we may also need a class for a **form**, such as a contact form that a user can fill out.
- **Admin Interface:** The admin interface is something optional. We could enter data into the rows of our database table by entering it as a post in the admin interface (this is the method we’ve been working towards). However, there’s also other ways we could populate our table rows:
 - For example, we could have a server that gathers weather data for us from some other servers, and this server is inserting data into our tables through some other scripts. In that case, the structure of our website covered in those upper 6 boxes in the diagram would take care of rendering that data for the user.

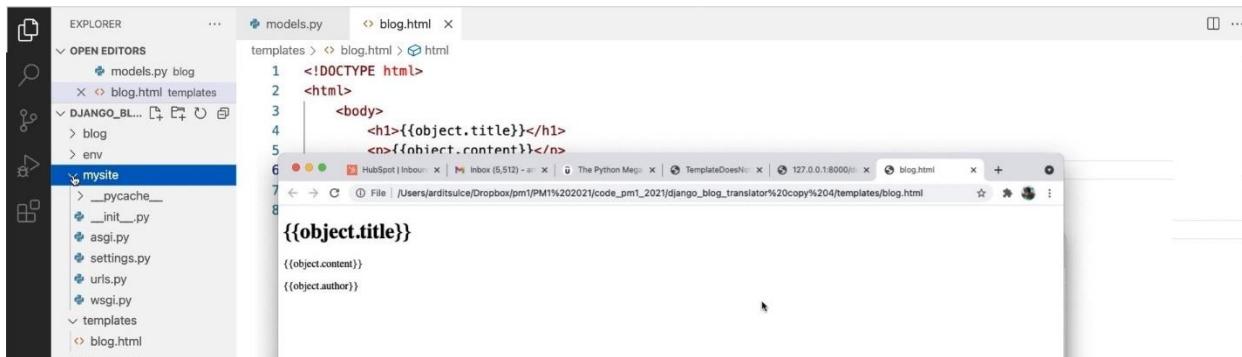
HTML Templates:

- Previously we created a “**Post**” class within `models.py`, which defines the “Post” database table. **Each row in the table will be able to be injected into an HTML template**. That’s what we’re going to work on in this lecture.
- The question is, where do we put the template? Currently, we have our **root project directory**, and inside of that we have:
 - **mysite** folder, which contains the configuration of our project.
 - **blog** folder, which contains the configuration and the files for our **blog app**.
 - **HTML templates** should go **directly under our project root directory**.
 - Now, **inside of templates**, we want to **insert a new file and type “`blog.html`”**.
- **`blog.html`:** Now we’re going to fill out our `blog.html` file with tags that correspond to the columns in our database table. Rather than using *static text* here like we have in simpler web pages, we want to use **Django Variables**:



```
<!DOCTYPE html>
<html>
  <body>
    <h1>{{object.title}}</h1> ← ← ←
    <p>{{object.content}}</p> ← ← ←
    <p>{{object.author}}</p> ← ← ←
  </body>
</html>
```

- However, if we open this `.html` file as-is in our browser, it still acts like *static text* at this point:



- This is because we haven’t made any connections yet. This **template** needs to be **connected to our `models.py` file somehow**.
 - The way we connect this **blog.html** with `models.py` is through a “**view**”. We’re going to do this in the next lecture, but for now it’s worth mentioning that these **views** are **classes**, and they’ll be written inside the `views.py` file in the **blog folder**.

- **`settings.py`:** Before that, we need to complete something regarding the **template**. We need to tell our **Django project's `settings.py`** file (inside “mysite”) where our **templates** are **located**.
- We do this by going to the bottom of `settings.py` and declaring a variable **TEMPLATES_DIR** (don’t forget to **import os**) at the beginning of `settings.py`):

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

- “**BASE_DIR**” is a variable declared by default at the beginning of the file, at it’s basically the **project’s root directory**.
- What our “**TEMPLATES_DIR**” variable is doing is, it’s **joining** the **base directory** with ‘**templates**’.
-
- Now we just need to save **settings.py**, and we’ll talk about **views.py** in the next lecture.
-
-
-
-
- **Note:** We got an **error** in the later “**URL Patterns**” lecture and needed to place “**TEMPLATES_DIR**” earlier in this file, below the “**BASE_DIR**” path in **settings.py**:

```
from pathlib import Path
import os

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates') ← ← ←
```

Django Views:

- We still need to connect our **blog.html** templates file to our **models.py** file and to our **database**.
 - The blog.html file has variables that are still waiting for input from the database table in **db.sqlite3**.
 - Since our **Post** class from models.py controls the structure of our database table, we need to connect blog.html to this class.
 - This is where the *middleman* **views.py** comes in handy.
- **views.py**: We're going to go into **views.py** and create a **class "BlogView"**. This is a simple class, and will be filled with two variables: **model** and **template_name**:
 - We set model to **model = Post**.
 - To do this we need to import Post from models.py by adding "**from .model import Post**" to the beginning of the file.
 - We set template_name to **template_name = 'blog.html'**.

```
from django.shortcuts import render
from .models import Post < < <

# Create your views here.

class BlogView: < < <
    model = Post < < <
    template_name = 'blog.html' < < <
```

- Once that file is saved, we're going to run **python manage.py runserver** in our virtual environment.
- This gets us to our homepage, but so far we don't have any webpages other than this. If we try and go to **/dogs** we'll get an error because it doesn't exist.
- In the next lecture, we're going to enter some *dummy data* into our **database table** using the program "**db Browser for SQLite**".
 - Normally we want to enter data through **admin**, but for testing things out, we quickly want some data to test with *now*.
- **Note:** We forgot to inherit from **generic**, from **django.views**, and it caused problems in the next lecture. Should look like this:

```
from django.shortcuts import render
from .models import Post
from django.views import generic < < <

# Create your views here.

class BlogView(generic.DetailView): < < <
    model = Post
    template_name = 'blog.html'
```

URL Patterns:

- In the **previous lecture** we created a class BlogView that **connects** the **model** (“Post”) with the **template** (“blog.html”).
-
- Now we’re going to use db Browser for SQLite to enter some **dummy data** to test our web app out on.
 - Using db Browser for SQLite, *open db.sqlite3*.
 - Click on the “**Browse Data**” tab and find the “**blog_post**” table.
 - Time to manually enter a new table row.
 - Click the button with the pop-up tool-tip saying “**Insert a new record in the current table**” (for me, it was to the right of the “Print” button). Note that “id” auto-increments ('1' in this case).
 - Into “**title**”, enter “**Dogs**”.
 - Into “**content**”, enter “**Dogs are good!**”.
 - For “**date_created**”, it should be in a format such as **YYYY-MM-DD**.
 - Into “**slug**”, enter “**dogs**”. Django will add the ‘/’ for us.
 - Into “**status**”, choose “**1**” for ‘published’ (remember we chose “0” for ‘draft’ and “1” for ‘published’).
 - For “**author_id**”, set this to “**1**”; this will be the **first user of the database**, or the **superuser**.
 - Once all the fields are filled, click on “**Write Changes**” up top. This saves the changes into a row in the database table.
 - Now we can **close** the **database**.
-
- **urls.py, ‘blog’ version:** (In the video, looking at “models.py”), now let’s connect the **slug** to **views.py**, in class **BlogView**.
 - To do that, we need to go to our “**blog**” **folder**, right-click, and choose “add new file”. We’re going to create a urls.py file.
 - We want to start by *importing from the current location* (“.”) “views”. We also want to import “path” from “django.urls”
 - We can then create a **list “urlpatterns”**:

```
from . import views
from django.urls import path

urlpatterns = [
    path('<slug:slug>', views.BlogView.as_view(), name='blog_view')
]
```

- The meaning of “<slug:slug>” is that, for “example.com/dogs”, Django will look through the database table, and it will search for “dogs” in **each row** of the “slug” field/column. If it finds “dogs” in a row, it’s going to **execute** **views.BlogView()** as the **view**. Including the argument **name “blog_view”** can come in handy later.
- We also need to do something else regarding URLs. You’ll see that there’s **another urls.py file inside** the folder **mysite**.

- **`urls.py`, ‘mysite’ version:** We also need to do something else regarding URLs. You’ll see that there’s *another* `urls.py` file *inside* the folder “**mysite**”, which contains the configurations of the Django project, not the Django apps such as “blog”.
- Inside *that* `urls.py` file, another **urlpatterns** list is being used.
 - This “**mysite**” `urls.py` file wants to know about the one inside of the “**blog**” app.
 - For every app that we create, we want to **declare it** here. So after the comma, we add:

```
from django.contrib import admin
from django.urls import path, include ← ← ←

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls')) ← ← ←
]
```

- Note that the use of an *empty string* here (“”) denotes the **home directory / project root directory**.
-
- **Error, see Note in “Django Views” lecture:** At this point, after saving all our changes, we got an **error message** while running the website, saying that our **object** doesn’t have a **method** `.as_view()`. This was because we forgot to **inherit “generic.DetailView”** in `BlogView` earlier.
 - It turns out that “**DetailView**” is an empty class that just **bundles/inherits** from other classes. Following this back shows that it inherits from “**BaseDetailView**”, which inherits from “**View**”. This is a *chain of inheritance*.
 - It’s the “**View**” class that has the method `.as_view()`.
-
- **Testing:** Now it’s time to **test out our app**.
 - We get a “**Page not found**” error now. This is because our homepage isn’t configured yet. We don’t have any **URL Pattern** that tells Django what to do when people visit the **homepage**.
 - For now, what we should look for instead is one of our blogs, i.e. “`127.0.0.1:8000/dogs`”. However, we get an **error message** saying “**TemplateDoesNotExist** at `/dogs`”. This is an easy error to fix.
 - Django **recognized** that there is indeed a “**dogs**” entry in the **database**; if we tried “`/cats`”, we would get a “Page not found” error instead.
 - This error is usually a **miscommunication error**, where Django is not able to find the **path of blog.html**.
 - Inside of `settings.py`, look for the “**TEMPLATES**” section. Inside of that is an *empty list* called “**DIRS**”, and this expects a value to be declared. We need to declare a list of **templates directories**, though in our case we only need **one**.
 - We also need to cut our “**TEMPLATES_DIR**” variable and paste it earlier in the `settings.py` file, right below “**BASE_DIR**” (**see Note in “HTML Templates” lecture**).
 - After this change, “`127.0.0.1:8000/dogs`” works as expected at this point.

Creating Admin Interface Views:

- Now we're going to **add** the **option** to the **Admin Interface** to be able to **add blog posts**. For that, we're going to need a *username* and *password* (which we already set up previously).
- So let's start off by going to "127.0.0.1:8000/admin" and logging in.
- On the admin page, you'll see that at the moment, you can only add new **users** and **groups**, not **posts**.
- **admin.py:** To add this functionality, we need to go to our "**blog**" folder and open **admin.py**.
 - The default comment in the script prompts us to "register our models here". The model they mean is our **Post** model, or our Post class from inside **models.py**.
 - We have to add **from .models import Post** at the top, then we register our model by adding **admin.site.register(Post)**:

```
from django.contrib import admin
from .models import Post < < <

# Register your models here.
admin.site.register(Post) < < <
```

- Now when we refresh our admin page, we should see a window labeled "**Blog**", and inside that we can click on "Add" for "**Posts**".
- Let's add the following:
 - Title: **Cats**
 - Content: "**Cats are also good!**"
 - Slug: **cats**
 - Author: <**author name**>
 - Status: **Publish**
- Then click on "**Save**", and you should get a message saying "**The post 'Post object (2)' was added successfully**". Both of our posts so far are also listed under the "**Posts**" section.
- If we go to "127.0.0.1:8000/cats" (or click on the post), it takes us to a new page with our "**cats**" entry.
-
- The instructor also noted that, while this is optional, you can **add some columns** to the list of posts giving more information about each post. For example, the name of each blog so far is very generic ("Post object (1)", "Post object (2)'), which makes it difficult to find the post that you want.
 - So how do we display the **Title** of the post here instead of "Post object"? Well first of all, let's talk about where the name "Post object (1)" comes from in the first place.
 - Since "Post" is our *model class*, this is actually the name of a particular *Post object* created from that *class*. Hence, "**Post object (#)**". In Python, if you were to just print out a class, it will print out the *name of that class*, "**<class '__main__.Post'>**". Django is trying to get a string representation of this class Post.

- However, we can change that using a *magic method*, `def __str__(self):`:

```
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
    slug = models.SlugField(max_length=200, unique=True)
    author = models.ForeignKey(to=User, on_delete=models.CASCADE)
    status = models.IntegerField(choices=STATUS, default=0)

    def __str__(self): < < <
        return self.title < < <
```

- Now when we refresh the page, the posts have been *renamed* to their `title`.
-
- If we want to add more information as columns to this list of posts, we'll want to go to `admin.py`, and we'll want to create a **new class**, `class PostAdmin`, which inherits from `admin.ModelAdmin`:

```
class PostAdmin(admin.ModelAdmin):
    list_display = ('date_created', 'author')
```

- Note here that “`ModelAdmin`” is a special version of “`Model`” for Admins. We also pass a *tuple* consisting of the fields we want to display on the page, in this case “`date_created`” and “`author`”.
- However, at the moment this is just a class “floating around”; Django doesn't know about this class. We want to tell Django somehow that the Django project needs to consider this class. So we just add it in below:

```
from django.contrib import admin
from .models import Post

class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'date_created', 'author') < < <

# Register your models here.
admin.site.register(Post, PostAdmin) < < <
```

- We also decided to add ‘`title`’ to the tuple, because otherwise our admin screen was skipping it now and going straight to ‘`date_created`’.
-
- In the next lecture, we'll **format** the **homepage**, because currently it's nonexistent.

Creating a Homepage:

- For now, we're going to **create** a **simple homepage** that just says "This is a **homepage**." However, later on we're going to **replace** it with one that automatically **lists all the blog posts**.
-
- So what do you **start** with **first**: an **HTML file**? A new **model**?
- Here's the instructor's general approach: **always think about the user**, so solve the problem starting with the user. What the **user sees** is the **HTML**.
 - From there, you go deeper and deeper, through the **View**, the **Model**, and eventually the **backend**.
- So the **first thing we need to do is create a new HTML template**.
-
- **index.html:** So we go to our "templates" folder and create a new file, naming it "**index.html**" (this is a pretty standard name for a homepage, but you could choose almost any name). Inside, we write a static message for now:

```
<!DOCTYPE html>
<html>
  <body>
    This is the homepage!
  </body>
</html>
```

- **urls.py, blog:** Now we want to point a URL to this page. We need to do this inside of our **blog** app folder, inside of the **urls.py** file found there. Inside of here is a **list of path function calls**:

```
urlpatterns = [
    path('<slug:slug>', views.BlogView.as_view(), name='blog_view')
]
```

- So we need to add another call to the list:

```
urlpatterns = [
    path('<slug:slug>', views.BlogView.as_view(), name='blog_view'),
    path('', views.HomeView.as_view(), name='home_view') ← ← ←
]
```

- **views.py:** Note that we don't have a **HomeView** class created in our **views.py** file yet. We'll need to add this:

```
class BlogView(generic.DetailView):
    model = Post
    template_name = 'blog.html'

class HomeView(generic.DetailView): ← ← ←
```

```
class BlogView(generic.DetailView):
    model = Post
    template_name = 'blog.html'

class HomeView(generic.DetailView): < < <
```

- The instructor noted that we **don't really want** to inherit from **DetailView**, but he left it there because he wanted to show us something later.
- He also noted that unlike with **BlogView**, we **don't really have** a **model** in this case. Our **index.html** is just a static page, it's **not getting any data from a database** like **blog.html** is. However, we don't actually *need* to input a model; we can just input **template_name**:

```
class BlogView(generic.DetailView):
    model = Post
    template_name = 'blog.html'

class HomeView(generic.DetailView):
    template_name = 'index.html' < < <
```

- Now when a user visits the home URL, the view "**HomeView**" inside the blog folder's [**urls.py**](#) will call the template "**index.html**" template in [**views.py**](#), and be rendered in the user's browser.
-
- **Error:** However, we got an error when trying out the page, saying that "**HomeView is missing a QuerySet**", saying that we need to **define the HomeView model**. The problem here is that we don't *have* a **HomeView** model.
 - It turns out that this is why he wanted to show us what would happen if we **inherited** from **generic.DetailView**. This is **not the correct view-type to use for views that simply render a template instead of getting data from a model**.
 - Instead, we want to use a **TemplateView**:

```
class BlogView(generic.DetailView):
    model = Post
    template_name = 'blog.html'

class HomeView(generic.TemplateView): < < <
    template_name = 'index.html'
```

- And now the homepage renders correctly!!
- So now we know the difference between a few different **view types**, including the **DetailView** for getting more complex data from a model, vs **TemplateView** that just renders a static page.

Creating an “About” Page:

- So far we've used **DetailView** to show **blog pages**, **TemplateView** for the **index page**, and now we want to use *another type of view* to show a *list of blogs*.

```
class BlogView(generic.DetailView):  
    model = Post  
    template_name = 'blog.html'  
  
class HomeView(generic.TemplateView):  
    template_name = 'index.html'
```

- Now, **TemplateView** isn't well-suited for showing a list of blogs, so we can either **delete** our entire **HomeView** class and replace it with this other view-type, or we can **repurpose it to use for an “About page”**.
 - First, we **renamed** “index.html” to “**about.html**”.
 - Then we went into [views.py](#) and replaced “index.html” with “**about.html**” and renamed “HomeView” to “**AboutView**”:

```
class AboutView(generic.TemplateView): ← ← ←  
    template_name = 'about.html' ← ← ←
```

- Then we go to [urls.py](#) and change it to point to **AboutView**, and change the name variable. We also want to pass a URL pattern in the beginning, which will be “**about/**”:

```
urlpatterns = [  
    path('<slug:slug>', views.BlogView.as_view(), name='blog_view'),  
    path('about/', views.AboutView.as_view(), name='about_view') ← ← ←  
]
```

- And now, going to “127.0.0.1:8000/**about/**” (you need the trailing slash “/”) gets you to your static webpage (the old homepage).
- In the next lecture, we're going to **create** the **index.html** page again, with new content showing a **list of blogs**.

Listing Blog Posts on the Homepage:

- Now to create a homepage that lists all of the blog posts, we're going to start by thinking about it from the user's point of view.
- So we'll start by creating a new file in "templates", [index.html](#).
 - We're going to start off like we usually do for an HTML file, with the `<!DOCTYPE html>`, the `<html>` and `<body>` tags.
 - Inside of the `body` tags, we're going to use something called a **template tag**, which follows the pattern "`{% %}`". We're going to put a **for-loop** inside of this:

```
<!DOCTYPE html>
<html>
    <body>
        {% for post in post_list %} ← ← ←
            {{post.title}}
            {{post.author}}
        {% endfor %} ← ← ←
    </body>
</html>
```

- Now, what is "`post_list`"? It's a **view**, which we need to define in [views.py](#).
- In [views.py](#), we need to create a class **PostList**:

```
class BlogView(generic.DetailView):
    model = Post ← ← ← note: "Post" down in queryset doesn't refer to this
    template_name = 'blog.html'

class AboutView(generic.TemplateView):
    template_name = 'about.html'

class PostList(generic.ListView): ← ← ←
    queryset = Post.objects.filter(status=1).order_by('date_created')
    template_name = 'index.html'
```

- In this case, **PostList** will inherit from [generic.ListView](#). ListView is specialized in rendering **multiple data rows**.
- It also needs a **queryset** variable, **Post**.
 - Now in this case, "Post" is the **model**, not the **view** (above in the same file). The "Post" **model** is found in [models.py](#).
 - [models.py](#) is where the **queryset** can find things like the **title**, **author**, etc.
 - We want to go through the **objects** (`Post.objects`) and **filter** for those with **status=1**, which are our **Published posts**.
 - We also want to **order_by** "date_created".
- We also set a **template_name** variable to "[index.html](#)".
-
- Now that we have a **view** for **PostList/post_list**, we need to set the path in [urls.py](#):

- Now that we have a **view** for **PostList/post_list**, we need to set the path in **urls.py** (the one inside the “**blog folder**”):

```
urlpatterns = [
    path('<slug:slug>', views.BlogView.as_view(), name='blog_view'),
    path('about/', views.AboutView.as_view(), name='about_view'),
    path('', views.PostList.as_view(), name='home') ← ← ←
]
```

- Now we can test out our website’s homepage. Make sure to save all the files we’ve just changed, then relaunch or reload the homepage.
 - In a way, it’s working:
 - The homepage lists blogs in the following pattern:
 - “Dogs <> author <> Cats <> author <>”
 - Not very appealing, but we can change that in **index.html**. For example, we may want to put the **title** inside of **<h2></h2>** tags and the **author** inside of **<p></p>** tags:

```
<!DOCTYPE html>
<html>
  <body>
    {% for post in post_list %}
      <h2>{{post.title}}</h2> ← ← ←
      <p>{{post.author}}</p> ← ← ←
    {% endfor %}
  </body>
</html>
```

- Now when we save the file and reload the homepage, we get something a bit nicer:



- We may want to modify the **ordering** of these posts. Currently, the oldest post is on top, but in the case of a blog, we may want the **newer posts shown at the top**. To do that, we just need to go to **views.py** and **adding a minus sign** to our **order_by** method:

```
class PostList(generic.ListView):
    queryset = Post.objects.filter(status=1).order_by('-date_created') ← ← ←
    template_name = 'index.html'
```

- And now the **newest blog post** (“Cats”) will be **on top**.
- At this point, we’re only **missing two things**: one is that the listed blog posts **should have a link** to said blog post, and the second is that we could use some **styling** to make it **look better**.

Creating Links:

- Previously we built a list of blog posts for the homepage of our web app. Now we want to link listed **posts** to their **respective blog post pages**.
- Currently we have an **<h2></h2>** tag for the **blog post title**, and a **<p></p>** tag for the **author name**. These are to be found in **index.html**.
 - What we can do here is, instead of having the blog post title as *simple text*, we can **turn it into a link**.
 - To do that, instead of using **<h2></h2>** tags, we can use **<a>** tags:

```
<!DOCTYPE html>
<html>
    <body>
        {% for post in post_list %}
            <a>{{post.title}}</a> ← ← ←
            <p>{{post.author}}</p>
        {% endfor %}
```

- **<a>** tags are used for **links** and **buttons**, so **clickable** HTML elements.
- Therefore, these tags **expect a link**, which follows the pattern ****:

```
{% for post in post_list %}
    <a href="localhost:8000/{{post.slug}}">{{post.title}}</a> ← ← ←
    <p>{{post.author}}</p>
{% endfor %}
```

- This would be one way to do it, with “**“{{post.slug}}**” being a variable. However, this won’t be the **best way** to do it, because “**localhost:8000**” will be changed to something else later on when we deploy the app. A better way to go would be to **use a URL tag**:

```
{% for post in post_list %}
    <a href="{% url 'blog_view' post.slug %}">{{post.title}}</a> ← ←
    <p>{{post.author}}</p>
{% endfor %}
```

- Here we use the “**blog_view**” name that we gave the blog post in **urls.py**. We also have to pass the second argument “**post.slug**” or else we get an error.
- Now we have working clickable links to our blog posts!!
- We can also improve how this page looks by re-applying those **<h2>** tags around “**“{{post.title}}**”:

```
{% for post in post_list %}
    <a href="{% url 'blog_view' post.slug
%}"><h2>{{post.title}}</h2></a> ← ← ←
    <p>{{post.author}}</p>
{% endfor %}
```

Adding Bootstrap to Django:

- In this video, we're going to learn how to make our homepage look like this with **Bootstrap** CSS styling:

The image shows two cards side-by-side. The first card is titled 'Cats' and contains text about the domestic cat species. The second card is titled 'Dogs' and contains text about the domestic dog species. Both cards feature a blue 'Read more' button at the bottom.

- **Bootstrap CSS** is a CSS Library and front-end framework that works well with Django, among other web frameworks.
- In this lecture, we're going to mainly be working with our [index.html](#) file.
- The way we're going to do this is by **looking at the Bootstrap documentation**.
- A few words about **Bootstrap** before we look at the documentation:
 - CSS is usually kept in a folder ("static") in the **project root directory**. In this folder, we normally create a file called "**main.css**", and in that file you write some CSS code that looks something like this:

```
p {  
    font-size: 17px;  
    color: blue;  
}
```

- This code will get all of the `<p>` tags and then color them blue and set their font size.
 - Note: There's a way to directly integrate this into an HTML file as well, but usually we keep it in main.css in a "static" folder for organization's sake.
- However, when we use **Bootstrap**, what we're doing is, we're using **code that is already written: CSS code**.
 - So the Bootstrap library already has CSS code in it, and you just use that code by referring to something called a **class**.
- For example, if you wanted to use Bootstrap to change all `<p>` tags, you'd add this to (for example) the [index.html](#) file...

```
{% for post in post_list %}  
    <a href="{% url 'blog_view' post.slug  
%}"><h2>{{post.title}}</h2></a>  
        <p class="p-pretty">{{post.author}}</p> ← ← ←  
    {% endfor %}
```

- ...and this ‘`class="p-pretty"`’ callout would reference some CSS code in Bootstrap such as...

```
P p-pretty { ← ← ←
  font-size: 17px;
  color: blue;
}
```

-
- [Bootstrap Installation & Setup](#): Now, back to the **online Bootstrap documentation**. There are two ways to install Bootstrap:
 - Either **install it directly on your machine** (“`$ npm install bootstrap`” / “`$ gem install bootstrap -v 5.1.0`”), or:
 - Use a quicker way with “**jsDelivr**”.
- We clicked on the second option, which includes a *guide for how to use Bootstrap*.
 - In this case, we have to include (copy/paste) a CSS file inside of the `<head>` tag for our [index.html](#) file. We don’t currently have any `<head>` tags, so let’s add some:

```
<!DOCTYPE html>
<html>
  <head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-GLhLTQ8iRABdZLl603oVMWSktQOp6b7In1z13/Jr59b6EGGoI1aFkw7cmDA6j6gD"
crossorigin="anonymous"> ← ← ←
  </head>
  <body>
    {% for post in post_list %}
      <a href="{% url 'blog_view' post.slug
%}"><h2>{{post.title}}</h2></a>
      <p>{{post.author}}</p>
    {% endfor %}
  </body>
</html>
```

-
- We also have to place the **JavaScript file**. To do that, copy/paste that from the documentation right before the closing `</body>` tag:
 -
 -
 -
 -
 -
 -
 -

- We also have to place the **JavaScript file**. To do that, copy/paste that from the documentation right before the closing **</body>** tag:

```
<!DOCTYPE html>
<html>
    <head>
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-GLh1TQ8iRABdZLl603oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD" crossorigin="anonymous">
    </head>
    <body>
        {% for post in post_list %}
            <a href="{% url 'blog_view' post.slug %}"><h2>{{post.title}}</h2></a>
            <p>{{post.author}}</p>
        {% endfor %}
        <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundle.min.js" integrity="sha384-w76AqPfDkMBXo30js1Sgez6pr3x5M1Q1ZAGC+nuZB+EYdgRZgiwxhTBTkF7CXvN" crossorigin="anonymous"></script> ← ← ←
    </body>
</html>
```

- And that's it. Now **save** the [**index.html**](#) file.
- Now when we relaunch/reload our webpage, we can see that styling has been introduced:



- The links are a bit lighter blue, and the font is a bit bigger.
- So, that's how to **include Bootstrap** into your program. Now let's move on to modifying it.
-
- There's a few more things we want to add before we start modifying things.
 - We want to set the **<html>** tag language to English and add some **<meta>** tags:
 -
 -
 -
 -
 -

- There's a few more things we want to add before we start modifying things.
 - We want to set the `<html>` tag language to English and add some `<meta>` tags:

```
<!DOCTYPE html>
<html lang="en"> ← ← ←
  <head>
    <meta charset="utf-8"> ← ← ←
    <meta name="viewport" content="width=device-width, initial-scale=1">←
```

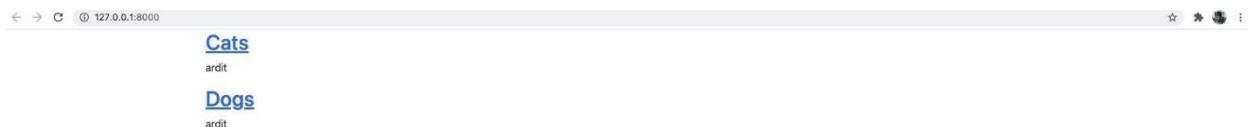
- The `<title>` in the Bootstrap starter example is optional, but we might as well add one:

```
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-GLhlTQ8iRABdZLl603oVMWSktQ0p6b7In1zI3/Jr59b6EGGoI1aFkw7cmDA6j6gD" crossorigin="anonymous">
  <title>Blog Posts</title> ← ← ←
</head>
```

- The documentation also explains what all the different tags do lower in the page.
- [Using Bootstrap to Modify Webpage](#):
- [container](#): Now we want to use the **actual features of Bootstrap**. To start, what you usually do is use a main `<div>` tag that wraps around your main content. We also want to add `class="container"` to the first `<div>` tag:

```
<body>
  <div class="container"> ← ← ←
    {% for post in post_list %}
      <a href="{% url 'blog_view' post.slug
    %}"><h2>{{post.title}}</h2></a>
      <p>{{post.author}}</p>
    {% endfor %}
  </div> ← ← ←
```

- “**container**” is one of the main classes of Bootstrap, and once we’ve applied it, our homepage now looks like this:



- Now we see a nice-looking margin added to the left of our content.

- What Bootstrap is doing with “**container**” is, “**container**” is some CSS code saved somewhere in the CSS file at “<https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css>” that we added in the first **<link>** tag.
- This “**container**” has a *margin* included in its attributes that adds space between the text and the left side of the window. The page also *adjusts* this margin depending on the width of the browser window (it’s dynamic).
- **card**: What we want to do now is, for each of the posts listed on the homepage, we want to create another **<div>** wrapper and assign that **<div>** tag the class of “**card**”:

```
<body>
  <div class="container">
    {% for post in post_list %}
      <div class="card"> ← ← ←
        <a href="{% url 'blog_view' post.slug
        %}"><h2>{{post.title}}</h2></a>
        <p>{{post.author}}</p>
      </div> ← ← ←
    {% endfor %}
  </div>
```

- Now when we reload our homepage, our listed posts are enclosed in “**card**” frames.
- The instructor then went through the web documentation and showed where “**container**” was, inside the “**Layout**” tab, and that “**card**” was inside the “**Components**” tab. The “**card**” page showed an example of what a card could look like, along with its **customization options**.
- One of the options the documentation said we could pass into “**card**” was a *margin*:

```
<body>
  <div class="container">
    {% for post in post_list %}
      <div class="card m-3"> ← ← ←
        <a href="{% url 'blog_view' post.slug
        %}"><h2>{{post.title}}</h2></a>
        <p>{{post.author}}</p>
      </div>
    {% endfor %}
  </div>
```

- This adds a margin of 3 (“level of spacing”, not pixels) around the card boxes:



- We can also add things like **images** here, but we’re not going to do that.
- **card-body & card-title**: next page:

- There's another `<div>` class, “**card-body**” that we can use to make our page look nicer.
- We also have a class called “**card-title**”, which in our case would match the blog post titles (i.e. “Cats” and “Dogs”):

```

<body>
    <div class="container">
        {% for post in post_list %}
            <div class="card m-3">
                <div class="card-body"> ← ← ←
                    <div class="card-title"> ← ← ←
                        {{post.title}} ← ← ←
                    </div>
                    <a href="{% url 'blog_view' post.slug
                %}"><h2>{{post.title}}</h2></a>
                        <p>{{post.author}}</p>
                    </div>
            </div>
        {% endfor %}
    </div>

```

- At the moment, we're duplicating “`{{post.title}}`” just to show what happens:



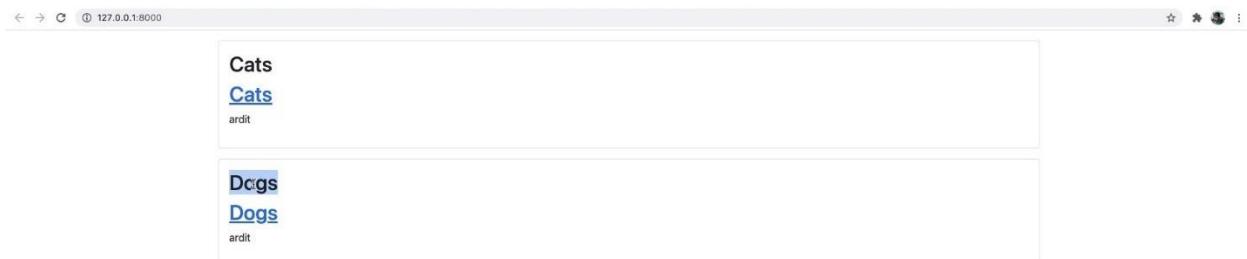
- He noted that it's probably better to use `<h2>` instead of `<div>` here, so we changed to that:

```

<div class="card-body">
    <h2 class="card-title"> ← ← ←
        {{post.title}}
    </h2> ← ← ←
    <a href="{% url 'blog_view' post.slug
    %}"><h2>{{post.title}}</h2></a>
        <p>{{post.author}}</p>
    </div>

```

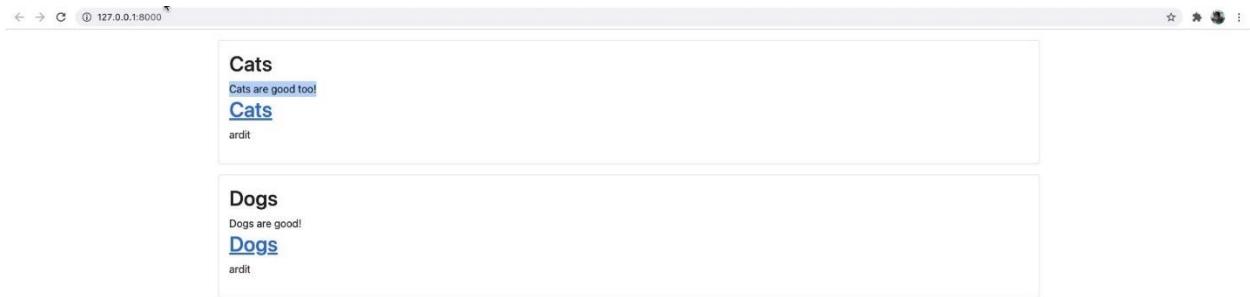
- Which led to this change:



- **card-text**: Next up, we have “**card-text**”, which we can use to display some of the post’s content by including “{{post.content}}” inside:

```
<h2 class="card-title">
    {{post.title}}
</h2>
<div class="card-text"> ← ← ←
    {{post.content}} ← ← ←
</div> ← ← ←
```

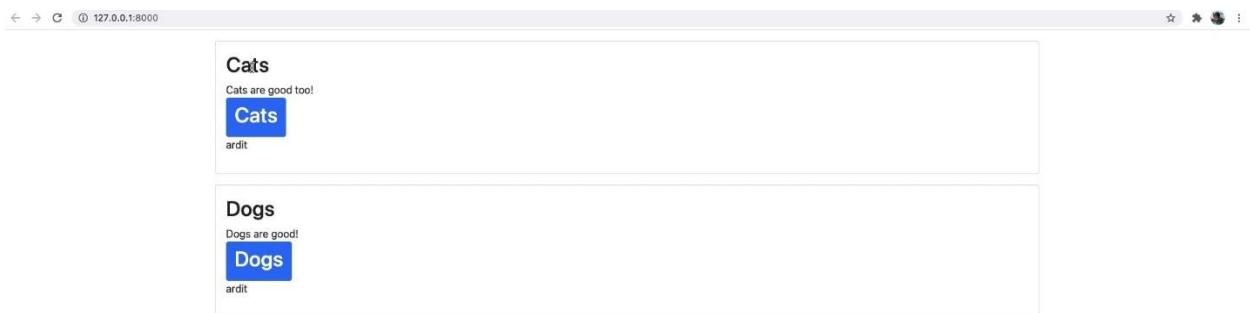
- This displays the page like this:



- **btn**: Next, let’s replace the *links* with **buttons**. This will be an `<a>` tag with `class="btn btn-primary"`. We’ll just be adding some of that into our *existing* `<a>` tag line:

```
<div class="card-body">
    <h2 class="card-title">
        {{post.title}}
    </h2>
    <div class="card-text">
        {{post.content}}
    </div>
    <a class="btn btn-primary" href="{% url 'blog_view' post.slug %}"><h2>{{post.title}}</h2></a> ← ← ←
        <p>{{post.author}}</p>
    </div>
```

- Now when we refresh the page, we get:



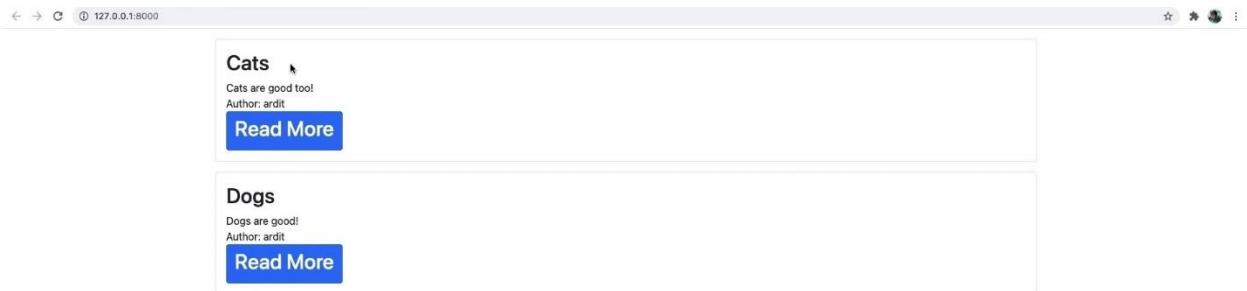
- Since this is a button now, we don't need to display the title on it anymore. We can just replace it with something like “Read More”:

```
<div class="card-body">
    <h2 class="card-title">
        {{post.title}}
    </h2>
    <div class="card-text">
        {{post.content}}
    </div>
    <a class="btn btn-primary" href="{% url 'blog_view' post.slug %}"><h2>Read More</h2></a> ← ← ←
        <p>{{post.author}}</p>
    </div>
```

- We can also rearrange where we put the “author” field, perhaps above the button, using another `<div>` with `class="card-text"`:

```
<div class="card-body">
    <h2 class="card-title">
        {{post.title}}
    </h2>
    <div class="card-text">
        {{post.content}}
    </div>
    <div class="card-text"> ← ← ←
        Author: {{post.author}} ← ← ←
    </div>
    <a class="btn btn-primary" href="{% url 'blog_view' post.slug %}"><h2>Read More</h2></a>
        <p>{{post.author}}</p> ← ← ←
    </div>
```

- Now our homepage looks like this:



- It still looks like a few of our `divs` are too close to each other, particularly the `content` div, the `author` div, and the `button` div.
- We could change this by replacing the `<div>` tags with `<p>` tags:

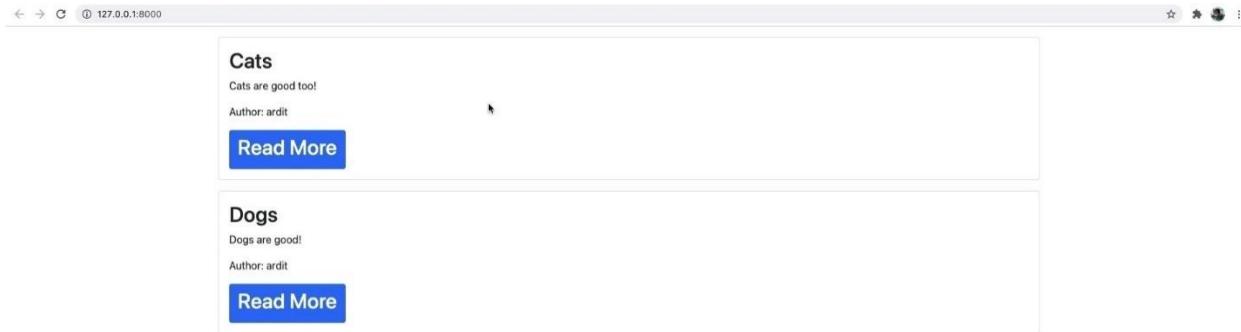
- We could change this by replacing the `<div>` tags with `<p>` tags:

```

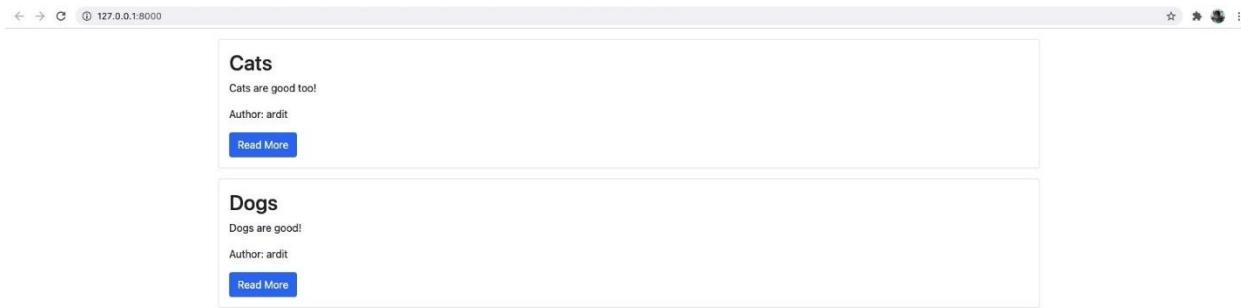
<div class="card-body">
    <h2 class="card-title">
        {{post.title}}
    </h2>
    <p class="card-text"> ← ← ←
        {{post.content}}
    </p> ← ← ←
    <p class="card-text"> ← ← ←
        Author: {{post.author}}
    </p> ← ← ←
    <a class="btn btn-primary" href="{% url 'blog_view'
post.slug %}"><h2>Read More</h2></a>
</div>

```

- This makes gives the elements of our cards more space:



- The buttons still look a little big relative to everything else; that's because the button text is wrapped in `<h2>` tags, so we can make the smaller *removing* the `<h2>` tags entirely:



- And that looks about right for now. Note that the **cards** are also **dynamic/responsive** to **changes in the window size**.
- In the next lecture, we're going to learn about **Django Template Tags**.

"index.html" code, so far:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-GLhTQ8iRABdZL1603oVMWSktQOp6b7In1z13/Jr59b6EGGoI1aFkw7cmDA6j6gD" crossorigin="anonymous">
      <title>Blog Posts</title>
  </head>
  <body>
    <div class="container">
      {% for post in post_list %}
        <div class="card m-3">
          <div class="card-body">
            <h2 class="card-title">
              {{post.title}}
            </h2>
            <p class="card-text">
              {{post.content}}
            </p>
            <p class="card-text">
              Author: {{post.author}}
            </p>
            <a class="btn btn-primary" href="{% url 'blog_view' post.slug %}">Read More</a>
          </div>
        </div>
      {% endfor %}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundle.min.js" integrity="sha384-w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBtkF7CXvN" crossorigin="anonymous"></script>
  </body>
</html>
```

Django Template Filters:

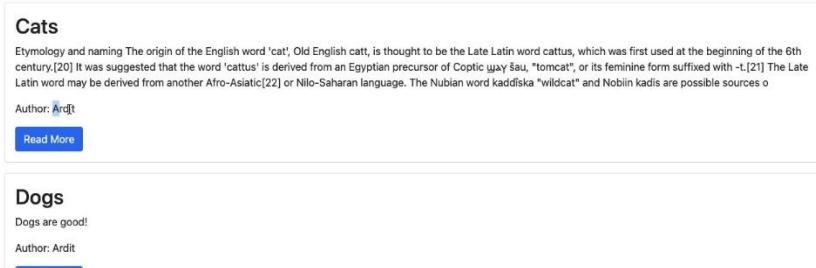
- Now that we've done some basic styling, it's time to take some other design choices into consideration.
- For example, what if the content of a blog post was a long article? We probably wouldn't want to display ALL of the content in the **card** in this situation. So how do we display just the first portion of the content, let's say the first 500 characters of any given blog post?
- Next, how do we show the **author name** in a *capital letter*? Currently it just displays an all-lowercase author name, identical to a given author's user account.
- The answer to both of these questions is **Template Filters**.
-
- Template filters** are a feature of Django which allows us to modify database table values from the HTML template (such as “{{post.title}}”, “{{post.content}}”, or “{{post.author}}”).
- We started by going to “**127.0.0.1:8000/admin**” and logging in to our user account. From there, we clicked on “**Posts**” and then our “**Cats**” blog post.
 - We're going to add some more content here, to make it a *longer article*.
 - We went to the Wikipedia article about cats and copy/pasted a bunch of content from there into our own **content**, then saved the changes. If we go to our homepage at this point, it's completely taken over by the content we just posted.
- We can now go to **index.html** and apply a **template filter** called “**slice**” to “{{post.content}}”. To use this function, we need to be able to use the “**pipe**” symbol (commonly used in Linux to chain commands together):

```
<p class="card-text">
    {{post.content | slice:"500"}}
</p>
```

- Doing a quick Google search for **Django template filters** gives a nice list of filters you can apply to your code. The next one we're going to use from the list is called “**title**”, which converts a string into **title-case**; we're going to apply that to “{{post.author}}”:

```
<p class="card-text">
    Author: {{post.author | title}}
</p>
```

- Now that both of those are done, our homepage should look like this:



Template Inheritance:

- Currently we have Bootstrap styling for our homepage, **index.html**, but not for **blog.html** (which renders posts) or **about.html** (which is just a static “about” page currently).
- We still need a way to **navigate** through our webpage. For example, **how can the visitor go from the homepage to the about page?**
 - Usually this is implemented through a **navigation menu bar**, perhaps at the top of the page. We’ll be creating one in this lecture.
 - Now, how do we implement that? The **navigation menu bar should be on every page**.
 - One way to do that is to code the HTML for the navigation menu into *every HTML template file*. However, that would be tedious.
 - D.R.Y principle = “**Don’t Repeat Yourself**”.
 - So what we’re going to do instead is to use **template inheritance**.
- **Template inheritance** works like this: the component that will be repetitive—in this case, our navigation toolbar—is going to stay in one single file. Then the other files (**index.html**, **blog.html**, and **about.html**) will **get** that other file where the navigation toolbar is coded. And that is template inheritance.
 - We start by creating a new file **base.html** inside our “**templates**” folder:

```
<nav> < < < “nav” tag
  <div>
    <ul> < < < unordered list tag
      <li> < < < listed item tag
        <a href="{% url 'home' %}"></a> < < <
      </li>
      <li>
        <a href="{% url 'about_view' %}"></a> < < <
```

- Note that we got the URL names used here from the **urls.py** file in the “**blog**” folder.
- Next we have to go to **index.html** and **cut** everything from the very first **<body>** tag to the top of the file...

```
templates > ◊ index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  ... <head>
4  .... <meta charset="utf-8">
5  .... <meta name="viewport" content="width=device-width, initial-scale=1">
6  .... <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css" rel="stylesheet" integrity='
7  .... <title>Blog Posts</title>
8  ... </head>
9  ... <body>
10 ...   <div class="container">
11 ...     &# for post in posts.list.all
```

- ...and paste them above the **<nav>** tag in **base.html**:
 -
 -
 -
 -
 -

- ...and paste them *above* the <nav> tag in **base.html**.
- Then go back to **index.html** and cut from the bottom </html> tag up to <script>...

```

18     {% endfor %}
19   </div>
20   <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-U1DkaT...>
21   </body>
22 </html>
23

```

- ...and paste that *below* the </nav> tag in **base.html**. Now, **base.html** is the **parent** file, and the other files will be the **children** files that will **get** from the parent file.
- This also means that it will install **Bootstrap** on every single child webpage.

Current base.html Code:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-GLhLTQ8iRABdZLl6O3oVMWSktQ0p6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD" crossorigin="anonymous">
      <title>Blog Posts</title>
  </head>
  <body>

    <nav>
      <div>
        <ul>
          <li>
            <a href="{% url 'home' %}"></a>
          </li>
          <li>
            <a href="{% url 'about_view' %}"></a>
          </li>
        </ul>
      </div>
    </nav>

    <!-- Bootstrap Bundle with Popper -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundle.min.js" integrity="sha384-w76AqPfDkMBDXo30jS1Sgez6pr3x5M1Q1ZAGC+nuZB+EYdgRZgiwxhTBTkF7CXvN" crossorigin="anonymous"></script>

```

```
</body>  
</html>
```

- Now we need to tell each of our pages (such as [index.html](#)) that it has to look to [base.html](#) with the template tag “`{% extends 'base.html' %}`”:

```
{% extends 'base.html' %} ← ← ← add this to the top  
    <div class="container">  
        {% for post in post_list %}
```

- We need to do this to [about.html](#) and [blog.html](#), too. We also need to delete several `<html>` and `<body>` tags that are no longer needed.
 - New [about.html](#) file:

```
{% extends 'base.html' %} ← ← ←  
  
This is the about page!
```

- New [blogs.html](#) file:

```
{% extends 'base.html' %} ← ← ←  
  
<h1>{{object.title}}</h1>  
<p>{{object.content}}</p>  
<p>{{object.author}}</p>
```

- Now we need to go back to [base.html](#) and add something after the lower `</nav>` tag:

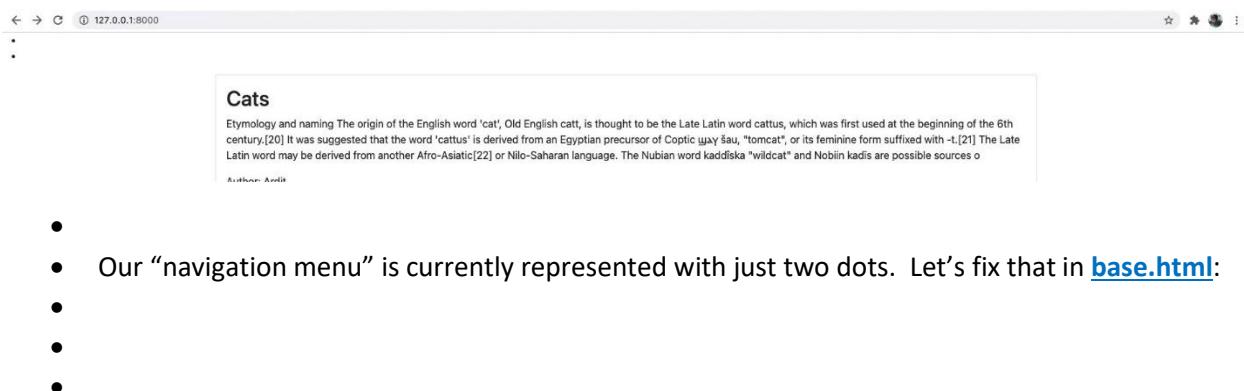
```
</nav>  
  
{% block content %} ← ← ←  
{% endblock content %} ← ← ←
```

- So what is this?
- From a user’s perspective, when they go to the website, they see the navigation menu bar (everything from `<nav>` to `</nav>` in [base.html](#)), and then below that they see the **content** of a given webpage. This could be the post about Cats, or the About page, etc.
- So the “block content”/“endblock content” represents a webpage’s actual content. We’ll need to remember this *variable*, “**content**”.
-
- Then we go to, say, [index.html](#) and then underneath “`{% extends 'base.html' %}`”, we add:
 -
 -
 -
 -

- Then we go to, say, [index.html](#) and then underneath “{% extends ‘base.html’ %}”, we add:

```
{% extends 'base.html' %}
{% block content %} < < < < < < < < <
    <div class="container">
        {% for post in post_list %}
            <div class="card m-3">
                <div class="card-body">
                    <h2 class="card-title">
                        {{post.title}}
                    </h2>
                    <p class="card-text">
                        {{post.content | slice:"500"}}
                    </p>
                    <p class="card-text">
                        Author: {{post.author | title}}
                    </p>
                    <a class="btn btn-primary" href="{% url 'blog_view' post.slug %}">Read More</a>
                </div>
            </div>
        {% endfor %}
    </div>
{% endblock content %} < < < < < < < < <
```

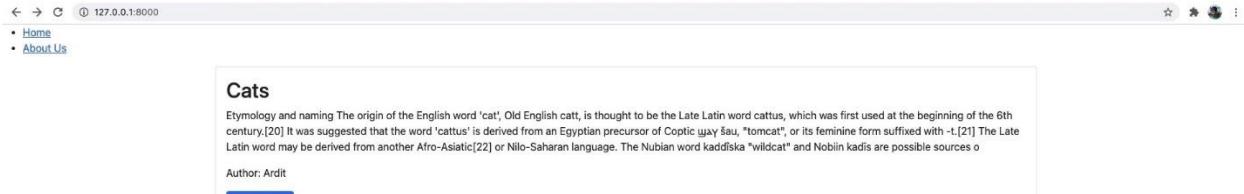
- We then do the same for [about.html](#) and [blog.html](#).
-
- **Error:** At this point we tried loading our homepage and got an error, saying “**Invalid block tag on line 1: ‘extend’.** Did you forget to register or load this tag?” This was just due to a **typo**:
 - We should’ve written “**extends**” instead of “extend”. I’ve gone through the notes previous to this and corrected it everywhere.
-
- Now when we get the homepage to load, we see this:



- Our “navigation menu” is currently represented with just two dots. Let’s fix that in [base.html](#); we added links in our tags, but no text:

```
<li>
    <a href="{% url 'home' %}">Home</a> ← ← ←
</li>
<li>
    <a href="{% url 'about_view' %}">About Us</a> ← ← ←
```

- Now when we refresh our homepage, we get:



- We also prettied up our [about.html](#) by putting it inside

tags with class="container". We can also put it inside

tags, and add additional content in another tag:

```
{% extends 'base.html' %}
{% block content %}
    <div class="container"> ← ← ←
        <h2>This is the about page!</h2> ← ← ←
        <div> ← ← ←
            This is the content! ← ← ←
        </div>
    </div>
{% endblock content %}
```

- Now, our **navigation menu bar** doesn’t look very pretty at the moment because we haven’t applied any **Bootstrap styling** to it.
 - It currently has **<nav>** with no **class**, **<div>** with no **class**, as well as ****, ****, and **<a>**.
 - We’ll add those in the next lecture.

Applying Bootstrap Styling to the Navigation Menu:

- Now to add styling to the tags in our [base.html](#) file.
- We started by indenting everything that creates the navigation menu bar by two indentations, to be ‘subordinate’ to the `<body>` tag:

```
<nav>
  <div>
    <ul>
      <li>
        <a href="{% url 'home' %}">Home</a>
      </li>
      <li>
        <a href="{% url 'about_view' %}">About Us</a>
      </li>
    </ul>
  </div>
</nav>
```

- Then we added classes to `<nav>` (note: Bootstrap elements can have [multiple classes](#)):

```
<nav class="navbar navbar-expand-lg navbar-light bg-light shadow"
id="mainNav">
```

- We gave it a **class="navbar"**, added **navbar-expand-lg** to make it a **large, collapsible** navbar, made it **light**, made the **background light**, and added **shadow** to it. We also gave it an ID/name: **id="mainNav"** to use elsewhere.
 - [Classes](#) like these can be found online in the **Bootstrap documentation**.
- Next we added classes to `<div>`:

```
<div class="collapse navbar-collapse" id="navbarResponsive">
```

- Next we added to ``:

```
<ul class="navbar-nav ml-auto">
```

- The “**ml**” stands for “**margin left**”.
- Next we have ``:

```
<ul class="navbar-nav ml-auto">
```

- We saved all this and refreshed our webpage:

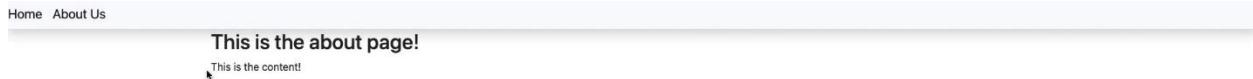


- It looks better, except that our `<a>` links aren't spaced very far part. However, the background and shadowing are nice.
- So let's add styling to our `<a>` tags now:

```

        <li class="nav-item">
            <a class="navbar-brand" href="{% url 'home' %}">Home</a> ← ← ←
        </li>
        <li class="nav-item">
            <a class="navbar-brand" href="{% url 'about_view' %}">About Us</a> ← ← ←
    
```

- Now let's see what that looks like:

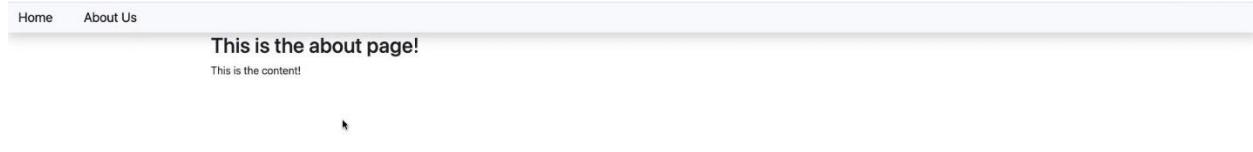


- Much better. I'm liking the understated black arial text better than the old blue link text.
- We should still put it a bit to the right. Adding the class `p-3` to the first `<a>` tag should do it:

```

        <a class="navbar-brand p-3" href="{% url 'home' %}">Home</a> ^ ^ ^
    </li>
    <li class="nav-item">
        <a class="navbar-brand p-3" href="{% url 'about_view' %}">About Us</a> ^ ^ ^
    
```

- The "`p-3`" means "let's give it some `padding` around the `<a>` tag. Let's see what that looks like:



- Nice. However, we probably want some more space between the **navigation bar** and the **content**. We *probably* want to do this in the `<nav>` element, by adding `mb-3` ("mb" stands for "margin bottom"):

```

<nav class="navbar navbar-expand-lg navbar-light bg-light shadow mb-3" id="mainNav"> ^ ^ ^
    
```

- Now there's some breathing room underneath the main menu bar.

Demo of the Django Translation App:

- This Django web app is going to be part of our Django website project (the “blog” app is already part of it).
- The **translation app** will be our **second app**.
 - Creating a second app will help us to see things from a **different perspective** and learn better. We’ll essentially be reviewing the **app creation process**.
 - Another reason is that we’ll be learning about **Django forms**. In this Django app, we’re going to have **two textboxes**:
 - In one, the **user** is going to **enter** words, terms, or sentences they want translated, and in the second they’re going to **see the translated text**.
 - Both boxes will be inside of a **form**, and when the user enters their text, it will be **posted** to a **server**. The server is going to **get** the text and then use a **Python library** to translate that text into another language. Then Django is going to **send** the data back to the user.
- So we’re going to:
 - 1) Review the app creation process.
 - 2) Learn about Django forms.
 - 3) Learn how to **process** content that the user sends to our Django app.

The Steps of Django App Development:

- The instructor started the video with a written list of the **steps you should take** to build this particular translator app:
 - 1) Research. Is it possible?
 - 2) Create an empty Django app.
 - 3) Top-bottom approach:
 - 1) Create **HTML**.
 - 2) Configure **URLs**.
 - 3) Create **views**.
 - 4) Create **models**.
 - 5) Connect the **processing** (translating) **part**.
- These steps also work for any other app you may want to build.
-
- 1) Is it possible? **VVV**
 - You want to input a **string**, have it **processed**, and return a string in **another language**.
 - So can you translate text with Python?
 - For that, you would want to do some research on the web until you find a **Python package** that allows you to do this.
 - Or, you could **build your own** Python package. However, this would be a lot of work for this particular task, because translating languages is not easy.
 - The instructor did some research to see if there was an existing package, and he found one called **googletrans**, which is built on top of the **Google Translate API**.
 - Check: **VVV**
- 2) Create an empty Django app.
 - We'll be doing this in the next lecture.
- 3) Top-bottom approach.
 - Think from the **user's point of view**. You want to create what the user wants to see.
 - **HTML**: The HTML controls what the user will see, so we create an **HTML template**.
 - **URL**: Then that template needs to be **rendered** to a particular **URL**. This will want to trigger a particular view.
 - **View**: Next we create our **views**. The view will get the HTML we created, and **serve** it to the URL.
 - **Model**: The **model** will contain the **data** that will be injected to the HTML template. At this step we should have built the basic **structure** of our web app, including the **textboxes** ready for text.
 - **Processing**: Once we have this set up/connected, we can get the input text, process/translate it, and return it to the user via the second textbox.
-
- In the next lecture, we will create our **empty Django web app**.

Creating an Empty App Structure of the Translator:

- Now we're going to create a new app inside our existing Django project.
- We started by going to our virtual environment's terminal and running **python manage.py runserver** (mine was already running at the time of writing these notes, but do this if it's not).
We're going to take another look at how our website currently looks:

The screenshot shows a simple website layout. At the top, there is a navigation bar with links for 'Home' and 'About Us'. Below the navigation bar is a main content area. In the content area, there is a single article card with the title 'Cats'. The card contains a short paragraph of text about the etymology of the word 'cat'. Below the text, it says 'Author: Ardit' and has a blue 'Read More' button.

- We're going to add another menu item, **Translator**, on our navbar, right where the mouse arrow is in the picture. This link will lead to the page that contains the **translator app**.
-
- To create the app, we first *stop the server from running* with CTRL + C.
- Next we run...
 - **python manage.py startapp translator** (note: most important command in this lecture)
- ...where we give it the name "translator". This causes a new folder, "translator", to appear in our **root project directory**, the same directory that the "blog" app folder is in.
- Once you create an app, you want to **register that app** in the "**mysite settings.py** file, in the "**INSTALLED_APPS**" section:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
    'translator', < < <
]
```

-
- In the next lecture, we're going to start building our **HTML template** for the **Translator app**.

Creating an HTML Form in Django:

Note: Resources for this lecture include a single file, “[translator.html](#)”, in case we want to simply copy/paste that code into our own translator.html file.

- Previously we created our empty Django app. The next step is to create the **HTML file**.
- We’re going to create a **new page** on the **menu bar**.
 - That page is going to be simple, with a textbox on the left, another textbox on the right, and then a button below.
 - User’s will be able to enter text in English on the left, then press the button and have their text printed in the right textbox in German. We can improve the app later on by adding an option to choose other languages, but for now we’re going to keep it simple and **focus on the forms**.
 - Let’s start by **creating an HTML file** in **templates**.
- Right-click on the “**templates**” folder in VSCode and create a file, “[translator.html](#)”.
 - He copy/pasted some code he’d already written for the file, and there was a copy of the file in the resources for this lecture, but I decided to manually type it from the video, for practice:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundle.min.js" integrity="sha384-w76AqPfDkMBDXo30js1Sgez6pr3x5M1Q1ZAGC+nuZB+EYdgRZgiwxhTBTkF7CXvN" crossorigin="anonymous"></link>

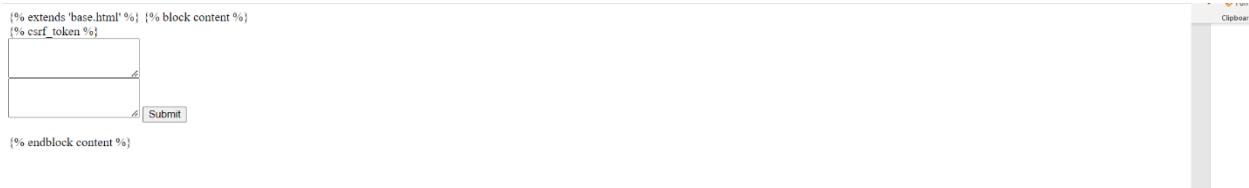
<div class="container">
    <form action="{% url 'get_translation' %}" method="post">
        <div class="row">
            <div class="col-sm-6 mt-3 left">
                <textarea class="form-control" rows="3" name="my_textarea"></textarea>
            </div>
            <div class="col-sm-6 mt-3 left">
                <textarea class="form-control" rows="3"></textarea>
                <input class="btn btn-primary ml-3 mt-3" type="submit" value="Submit">
            </div>
        </div>
    </form>
</div>
```

- We tested the HTML file out as-s by opening it in a browser. Currently the button doesn’t do anything, but you can type things into each box.
- At this point we deleted the Bootstrap line at the top, because we’re just going to inherit that from **base.py**.
-
-

- At this point we deleted the Bootstrap line at the top, because we're just going to inherit that from base.html.

```
{% extends 'base.html' %} ← ← ←
{% block content %} ← ← ←
<div class="container">
    <form action="{% url 'get_translation' %}" method="post">
        <div class="row">
            <div class="col-sm-6 mt-3 left">
                <textarea class="form-control" rows="3"
name="my_textarea"></textarea>
            </div>
            <div class="col-sm-6 mt-3 left">
                <textarea class="form-control" rows="3"></textarea>
                <input class="btn btn-primary ml-3 mt-3" type="submit"
value="Submit">
            </div>
        </div>
    </form>
</div>
{% endblock content %} ← ← ←
```

- Note: Although my code is the same as his, the Bootstrap styling isn't working when I open my translator.html file in a browser. This was true both when it had the full Bootstrap link and when we *extended* it from base.html. This is also true if I try and open the resources version of translator.html that I downloaded.
-
- Now we need to **define** a URL through which the user will access this particular HTML webpage.
 - Hopefully this will fix the above problem. Currently it looks like this:



Configuring the URLs:

- The first place we want to go for this is to our “mysite” folder and open [urls.py](#). Then we add our new page, [translator.html](#):

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls')),
    path('translate/', include('translator.urls')) ← ← ←
]
```

- Now, currently we don't have a “[urls](#)” file inside our **translator app folder**, so we need to create one. Right-click on the **translator app folder** and create a [urls.py](#) file there.
 - Inside of this file we'll define the URLs associated with the app. Some apps could have multiple URLs associated with them, but in our main **mysite >> urls.py** we only have the one, “[translate/](#)”.
 - We could also, for example, have “example.com/translate/whatever” or “example.com/translate/get”, etc. But every one of those pages would contain the “[translate/](#)” part, which we've defined in **mysite >> urls.py**.
 - The main “[translate/](#)” URL can be considered a **main door**. Any subsequent doors need to be defined in **translator >> urls.py**. Let's go to the version of [urls.py](#) inside of the “blog” folder and copy/paste all of that into our new file, as a starting point:

```
urlpatterns = [
    path('<slug:slug>', views.BlogView.as_view(), name='blog_view'),
    path('about/', views.AboutView.as_view(), name='about_view'),
    path('', views.PostList.as_view(), name='home')
]
```

- However, we can get rid of some unnecessary things here. For example, we won't be needing the “[slug](#)” lines, since we won't be posting articles. We can also delete the “[about/](#)” section.
 - Remember that the empty string line, “['](#)”, will render a URL with just “[translate/](#)” and nothing else after it.
 - When a user goes to “example.com/translate/”, one of the **views** we're going to create is going to handle that request. So let's call it:

```
urlpatterns = [
    path('', views.translator_view, name='translator_view')
]
```

- By default, Python works with views *as functions*, so in this case, we can also delete the [.as_view\(\)](#) method we've been using. The previous classes we've been using weren't views, so we were using “[.as_view\(\)](#)” to **convert them to views**.
- Now we're ready to create our **view**.

Creating a Form:

- Previously we configured both **mysite >> urls.py** and **translator >> urls.py**:

```
urlpatterns = [
    path('', views.translator_view, name='translator_view')
]
```

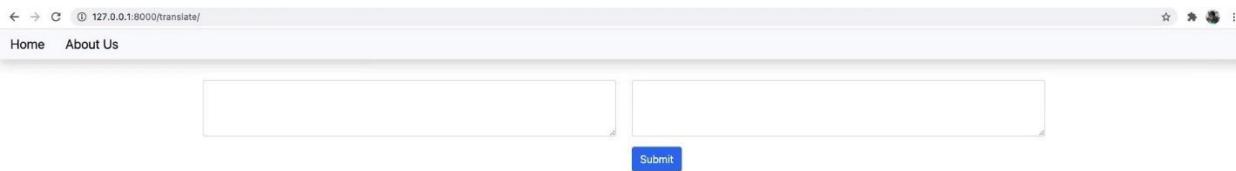
- We now need to *create* a view for “**translator_view**”. We’ll do this inside of **translator >> views.py**. As stated previously, this will be a **function**:

```
from django.shortcuts import render

# Create your views here.

def translator_view(request):
    return render(request, template_name='translator.html')
```

- The parameter “**request**” could be any name, but choosing this name is good practice. This function, on the background, expects a *request*, and **translator >> urls.py** will pass that request object when the user visits the website.
- We saved that file, so let’s see what will happen when we run this. If not already running, in the virtual environment console run **python manage.py runserver**. Now let’s visit the URL **“127.0.0.1:8000/translate/”**.
 - **Error:** At this point, we get an error, saying “Reverse for ‘get_translation’ not found: ‘get_translation’ is not a valid view function or pattern name.”
 - This ‘**get_translation**’ is actually in our “**translator.html**” file, inside **<form action=...>**. This has to correspond to one of the patterns (paths) inside of **translator >> urls.py**, so let’s try and change that **<form action=...>** to include “**translator_view**” instead.
 - Now we get this:



- As a side note, turns out I was right earlier when I suggested that linking **translator.html** to the other stuff would cause the Bootstrap styling to actually work for this page.
- However, if we press the “Submit” button, we get an **error “403 Forbidden: CSRF verification failed. Request aborted.”**
- **CSRF** is a security feature in Django that needs to be enabled, and it’s very easy to enable it. We just have to add it in our template:
 -
 -
 -
 -

- CSRF is a security feature in Django that needs to be enabled, and it's very easy to enable it. We just have to add it in our [translator.html](#) template, just under the <form action=...> line:

```
{% extends 'base.html' %}  
{% block content %}  
<div class="container">  
    <form action="{% url 'translator_view' %}" method="post">  
        {% csrf_token %} ← ← ←  
        <div class="row">  
            <div class="col-sm-6 mt-3 left">  
                <textarea class="form-control" rows="3"  
name="my_textarea"></textarea>  
            </div>  
            <div class="col-sm-6 mt-3 left">  
                <textarea class="form-control" rows="3"></textarea>  
                <input class="btn btn-primary ml-3 mt-3" type="submit"  
value="Submit">  
            </div>  
        </div>  
    </form>  
</div>  
{% endblock content %}
```

- Now if we go back to the [/translate/](#) webpage and click “Submit”, nothing happens (but hey, at least it’s not an error now).
 - However, we’re not handling any *post requests* at the moment.
 - In [views.py](#), we’re not handling any post requests or telling Django what to do when there’s a post request.
- In the next lecture we’ll set that up, and we’ll also handle *processing* user input.

Getting and Processing User Input Through a Form:

- So far we've built our "translator" webpage. It looks good, but currently it doesn't do anything.
- To understand what we need to do, we first have to understand **two HTTP requests** that happen here:
 - The first HTTP request happens when the user enters the URL.
 - This is a **get request**, and it's being handled by the URL pattern inside both **translator >> urls.py** and **mysite >> urls.py**. These two are merged into one URL, which is "**example.com/translate/**".
 - However, there's another HTTP request being made when the user presses the Submit button. Another URL is being accessed.
 - The other URL being accessed is the one seen in **translator.html...**

```
<div class="container">
    <form action="{% url 'translator_view' %}" method="post"> ← ← ←
        <div class="row">
```

- ...which is the same as in **translator >> urls.py**:

```
urlpatterns = [
    path('', views.translator_view, name='translator_view') ← ← ←
]
```

- Currently, visiting the URL through a **get request** and pressing the "Submit" button are both taking the user to the same URL location. We need to separate these: The user will be able to 1) visit the URL through a **get request**, or 2) hit the Submit button to make a **post request**.
 - To start, we need to distinguish between **get requests** and **post requests** in **views.py**:

```
def translator_view(request):
    if request.method == 'POST': ← ← ←
        original_text = request.POST['my_textarea'] ← ← ←
        print(original_text)
    else:
        return render(request, template_name='translator.html')
```

- Note that "my_textarea" came from **translator.html**.
- We're also going to **print** out "original_text" to make sure it's working right.
-
-
-
-
-
-
-
-

- When we type something into the textbox and hit “Submit”, whatever we typed gets printed out to the terminal now. However, we also get an *error*, because we’re not currently *returning* anything. So we add these lines to [views.py](#):

```
def translator_view(request):
    if request.method == 'POST':
        original_text = request.POST['my_textarea']
        output = original_text.upper() ← ← ←
        return render(request, 'translator.html', {'output_text': output}) ←
    ←
    else:
        return render(request, 'translator.html')
```

- Then to match the ‘**output_text**’ portion, we add a tag with that into our [translator.html](#) file:

```
<div class="col-sm-6 mt-3 left">
    <textarea class="form-control" rows="3"
name="my_textarea"></textarea>
</div>
<div class="col-sm-6 mt-3 left">
    <textarea class="form-control"
rows="3">{{output_text}}</textarea> ← ← ←
    <input class="btn btn-primary ml-3 mt-3" type="submit"
value="Submit">
</div>
```

- Now when we write something in the left textbox and hit Submit, we get the **capitalized version** in the right textbox. So that’s working.
- One issue we still have with our app is that, once we write and press “Submit”, our original text in the left textbox disappears. Let’s fix that.
 - First let’s understand *why* that’s happening, starting in [views.py](#):

```
def translator_view(request):
    if request.method == 'POST':
        original_text = request.POST['my_textarea']
        output = original_text.upper()
        return render(request, 'translator.html', {'output_text': output})
    else:
        return render(request, 'translator.html')
```

- So here when it’s a *post request*, we’re creating some output and then *injecting it* back into the page, but currently we’re not sending anything else.
- So, if we want the **original text** in the webpage, we just need to add that to our **render**:
-

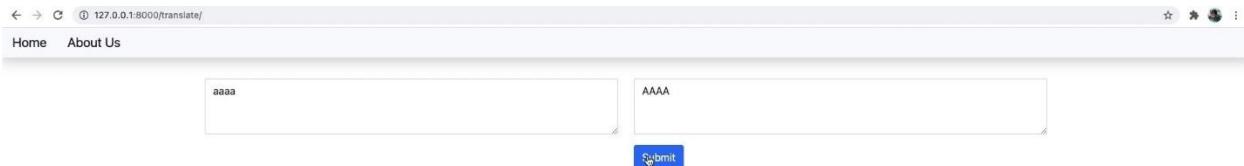
- So, if we want the ***original text*** in the webpage, we just need to add that to our **render**:

```
def translator_view(request):
    if request.method == 'POST':
        original_text = request.POST['my_textarea']
        output = original_text.upper()
        return render(request, 'translator.html', {'output_text': output,
'original_text': original_text}) < < <
    else:
        return render(request, 'translator.html')
```

- Then we want to copy that string '**original_text**' and add it to [translator.html](#):

```
{% extends 'base.html' %}
{% block content %}
<div class="container">
    <form action="{% url 'translator_view' %}" method="post">
        {% csrf_token %}
        <div class="row">
            <div class="col-sm-6 mt-3 left">
                <textarea class="form-control" rows="3"
name="my_textarea">{{original_text}}</textarea> < < <
            </div>
            <div class="col-sm-6 mt-3 left">
                <textarea class="form-control"
rows="3">{{output_text}}</textarea>
                <input class="btn btn-primary ml-3 mt-3" type="submit"
value="Submit">
            </div>
        </div>
    </form>
</div>
{% endblock content %}
```

- And now it works:



- That completes the Django part. Now all we need to do is **translate** using another package.

Completing the Translator App:

- In order to **translate** our text, we're going to start by opening a **new virtual environment terminal** and running **pip install googletrans**.
 - Note: Currently (as of 2016 when this video was recorded) there's an issue with the base library which can be fixed by running: **pip install googletrans==4.0.0-rc1**. Try either version, and if it doesn't work, install the other version and it will overwrite the first.
 - I had to use the "**pip install googletrans==4.0.0-rc1**" version to get the following interactive shell commands to work.
- Once installed, we start an **interactive Python shell** and run:
 - **>>> from googletrans import Translator**, to *import* it.
 - **>>> translation = Translator()**, to create an *instance* of the *Translator class*.
 - **>>> translation.translate(text="How did you learn Python?", dest='de').text**, where '**text**' is our 'input' text and '**dest**' is our **destination language**, which we've set to 'de' for 'Deutsch' or German—you can find language abbreviations on Google. Adding **.text** at the end outputs the text.

The screenshot shows a Jupyter Notebook interface with a terminal tab selected. The terminal window displays a Python session. The user has imported the googletrans module and created a Translator instance. They then used the translate method on this instance, passing in the English text "How did you learn Python?" and specifying the destination language as 'de'. The resulting German translation, "Wie hast du Python gelernt?", is printed to the console. The terminal also shows the Python version and build information.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
(env) arditsulce@Ardits-MacBook-Pro django_blog_translator % python
Python 3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from googletrans import Translator
>>> translation = Translator()
>>> translation.translate(text="How did you learn Python?", dest='de').text
'Wie hast du Python gelernt?'
>>>
```

- So the library is working for us. Now we want to incorporate it into our app.
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

- The best practice here is to create a ***new file*** inside of our “**translator**” app folder.
 - Let’s call it [translate.py](#):

```
from googletrans import Translator

def translate(text):
```

- This **translate** function will be used *inside* of [views.py](#). We’re going to call the function here, to translate instead of changing to uppercase:

```
from django.shortcuts import render
from . import translate < < < our function (from above)

# Create your views here.

def translator_view(request):
    if request.method == 'POST':
        original_text = request.POST['my_textarea']
        output = translate.translate(original_text) < < <
        return render(request, 'translator.html', {'output_text': output,
'original_text': original_text})
    else:
        return render(request, 'translator.html')
```

- Now let’s fill out our **translate** function in [translate.py](#) so it will work:

```
def translate(text):
    translator = Translator()
    translation = translator.translate(text=text, dest='de')
    return translation.text
```

- Save both files. Now let’s try out the app in the browser:

The screenshot shows a web page with two text input fields. The left field contains the German sentence "Wo hast du Python gelernt?". The right field contains the English sentence "Where did you learn Python?". Below the fields is a blue "Submit" button.

- And it seems to be working. We got a translation in German.
-
- Now let’s add a ***menu item*** to our navbar to add this translator app.
-
-

- Now let's add a **menu item** to our navbar to add this translator app.
- To do that, we're going to go to [base.html](#) and add a new to our navbar:

```
<li class="nav-item">
    <a class="navbar-brand p-3" href="{% url 'home' %}">Home</a>
</li>
<li class="nav-item">
    <a class="navbar-brand p-3" href="{% url 'about_view' %}">About Us</a>
</li>
<li class="nav-item">
    <a class="navbar-brand p-3" href="{% url 'translator_view' %}">Translator</a> ← ← ←
</li>
```

-
- And that concludes the Django blog/translator app.

Section 33: App 10: Build a Geography Web App with Flask and Pandas:

Demo of the Geography Web App:



- This is a **Flask application**.
- It expects a **CSV file** from the user, which should have a **column** named “**address**” or “**Address**”.
 - The user can upload such a file using the “Choose File” button.
 - Next to this button, it should either read the **file name**, or “**No file selected**”.
 - Once the file is chosen, the user can press the “**Submit**” button. A table of addresses similar to the following will then show up on the screen:

ID	Address	Name	Employees	Latitude	Longitude
0 1	3666 21st St San Francisco CA 94114 USA	Madeira	8	37.756489	-122.429343
1 2	735 Dolores St San Francisco CA 94119 USA	Bready Shop	15	NaN	NaN
2 3	332 Hill St San Francisco California 94114 USA	Super River	25	37.755725	-122.428601
3 4	3995 23rd St San Francisco CA 94114 USA	Ben's Shop	10	37.752965	-122.431714
4 5	1056 Sanchez St San Francisco California USA	Sanchez	12	37.752146	-122.429815

Download

- The **backend** of the Python script will read the file and add a **latitude** and a **longitude** and tack it onto the table. These are *calculated* from the “**address**” column. This is similar to the *geocoding* we’ve done previously.
- Lastly, you want to enable the user to **download** the resulting **table** as a **CSV file**.
-
- Note that you might have to add some **error handling**, such as when a user uploads a file that doesn’t contain a column with “address” in the name.
 - The example version *output a message* to the screen: “**Please make sure you have an address column in your CSV file!**”.

Solo Attempt, Notes:

Initial Setup:

- **Virtual environment** stuff from:
 - **Section 22** (App-4, “Web Development with Flask – Build a Personal Website”)
 - **Section 32** (App-9, “Django & Bootstrap Blog and Translator App”)
- Started off by creating a folder, “**my first attempt**”, just to give it a name.
- CD’d into that from my terminal and ran **python -m venv virtual** to create a virtual environment.
Closed and reopened terminal to get “**(env)**” message in command line.
- Ran:
 - **pip install flask,**
 - **pip install pandas,**
 - **pip install geopy,**
 - **pip install openpyxl** and **pip install xlrd**, (optional, in case I want to add functionality for .xlsx, .xls files in the future future)
- A

Writing HTML and CSS:

- Mainly from **Section 31** (App-8, “Flask and PostGreSQL – Build a Data Collector Web App”).
- Reused “**main.css**” code and some of “**index.html**” from “**Section-31, App-8 Data Collector**” as a starting point.
- Replaced sections as needed.
-
-
- a
- A

Pandas Dataframes and Geocoding:

- From **Section 13** (“Using Python with CSV, JSON, and Excel Files”).
 - From this section, ran **pip install pandas**, **pip install geopy**, and both **pip install openpyxl** and **pip install xlrd** from above.
-
- A
- A
- A
- A
- A
- A

Solution, Part 1:

- First off, let's create our **directory structure** so we can organize our files:
 - Create “**static**”, “**templates**”, “**uploads**”, and use **venv** to create “**virtual**”.
 - “**uploads**” is a folder where we will generate *intermediate files* that the application will make available for download for the user.
- Use **pip** to install:
 - **flask, pandas, geopy**.

index.html:

- As far as the actual app goes, let's start from the **front-end** first.
 - So let's create “**index.html**”:

```
<!DOCTYPE html>
<html lang="en">
<title> Super Geocoder </title>
<head>
    <link href="../static/main.css" rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1>Super Geocoder</h1>
        <h3>Please upload your CSV file. The values containing addresses should be in a column named <em>address</em> or <em>Address</em></h3>
        <form action="{{url_for('success_table')}}" method="POST"
        enctype="multipart/form-data">
            <input type="file" accept=".csv" name="file" />
            <button type="submit"> Submit </button>
        </form>
        <div class="output">
            {{text|safe}}
            {% include btn ignore missing %}
        </div>
    </div>
</body>
</html>
```

main.css:

- Next let's make our **front-end** look *prettier* by creating a “**main.css**” file:

```
html, body {  
    height: 100%;  
    margin: 0;  
}  
.container {  
    margin: 0 auto;  
    width: 100%;  
    height: 100%;  
    background-color: #006666;  
    color: #e6ffff;  
    overflow: hidden;  
    text-align: center;  
}  
.container form {  
    margin: 20px;  
}  
.container h1 {  
    font-family: Arial, sans-serif;  
    font-size: 30px;  
    color: #DDCCEE;  
    margin-top: 80px;  
}  
.container button {  
    width: 70px;  
    height: 30px;  
    background-color: steelblue;  
    margin: 3px;  
}  
.container input {  
    width: 200px;  
    height: 15px;  
    font-size: 15px;  
    margin: 2px;  
    padding: 5px;  
    transition: all 0.2s ease-in-out;  
}  
.output {  
    display: inline-block;  
}
```

- Remember to reference the CSS file in the **<head>** tag in “**index.html**”.

app.py:

- He started with just the bare bones, and named this initial version “[app_ver1.py](#)”:

```
from flask import Flask, render_template, request, send_file
from geopy.geocoders import ArcGIS
import pandas

app=Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route('/success-table', methods=['POST'])
def success_table():
    return render_template()

@app.route("/download-file/")
def download():
    return send_file()

if __name__=="__main__":
    app.run(debug=True)
```

- We start by bringing in the necessary *libraries*: [flask](#), [pandas](#), and [geopy](#).
 - We need “[render_template](#)” to render our various HTML files in the “templates” folder.
 - We need the “[request](#)” method because we know that users will be making requests.
 - We need the “[send_file](#)” method because we’re allowing users to download files.
 - Note that in his older version he brings in “[Nominatim](#)”, but in newer versions we need to bring in “[ArcGIS](#)”.
- We then create a *Flask instance* and set it to the variable “[app](#)”.
- We then create [three functions](#), which for now don’t do anything. All three functions will have a *decorator function* at the beginning; we want a decorator every time the user does something.
 - **def index:** The first will be the *homepage* (“[/](#)”), with **def index**. This will render the homepage, which is “[index.html](#)”.
 - **def success_table:** The next thing you might expect if the user is on your homepage is that, once a user presses the “**Choose File**” button, selects a CSV file, and presses “**Submit**”, what you want to do is to load that file into Python.
 - You want to read this file as a [pandas](#) **dataframe**, calculate **latitude** and **longitude** from the address column, **return a dataframe**, and **send that dataframe** via a [render_template](#) method down below the buttons on the homepage.
 - We want a function that does all of this, so we create **def success_table**. The decorator here will be `@app.route('/success-table', methods=['POST'])`, and it requires the ‘POST’ methods setting, because we’re expecting a POST request.

- **def download:** Since our **success_table** function will display the new table and show a “Download” button, we want to trigger a **download** function when the user clicks it. This will **return send_file()**, sending the file to the user via download.
-
-
- Next we'll add some more functionality in “app_ver2.py”, mainly in our “**success_table**” function, but also a few things in our “**download**” function:

```
from flask import Flask, render_template, request, send_file
from geopy.geocoders import ArcGIS
import pandas

app=Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route('/success-table', methods=['POST'])
def success_table():
    if request.method=='POST': ← ← ← (1)
        file=request.files['file'] ← ← ← (1)
        df=pandas.read_csv(file) ← ← ← (2)
        gc=ArcGIS() ← ← ← (2)
        df["coordinates"]=df["Address"].apply(gc.geocode) ← ← ← (2)
        df['Latitude'] = df['coordinates'].apply(lambda x: x.latitude if x != None else None) ← ← ← (3)
        df['Longitude'] = df['coordinates'].apply(lambda y: y.longitude if y != None else None) ← ← ← (3)
        df=df.drop("coordinates", 1) ← ← ← (3)
        df.to_csv("uploads/geocoded.csv", index=None) ← ← ← (4)
        return render_template("index.html", text=df.to_html(),
btn='download.html') ← ← ← (4)

@app.route("/download-file/")
def download():
    return send_file("uploads/geocoded.csv", attachment_filename='yourfile.csv',
as_attachment=True)

if __name__=="__main__":
    app.run(debug=True)

●
● (1) Now in success_table, it checks if there's a 'POST' request. If there is one, we get a [file].
○ This is the name of the input from index.html:
<input type="file" accept=".csv" name="file" />
```

- This is the *uploaded* CSV file. When the user presses the *button* “Submit”, a URL is triggered: “{{url_for('success_table')}}”, or the URL for the **success_table** function.
-
- (2) We then create a dataframe **df=pandas.read_csv(file)** that reads our new “file” storage variable.
 - Then we create a geocoder variable **gc=ArcGIS()** (“**gc=Nominatum()**” in the instructor’s older version).
 - We then create a temporary storage column in our dataframe, **df[‘coordinates’]** and apply our geocoding with **df[“Address”].apply(gc.geocode)**.
-
- (3) Then we get a **Latitude** column and a **Longitude** column out of our **coordinates** column using **.apply** and some *lambda functions*:

```
df['Latitude'] = df['coordinates'].apply(lambda x: x.latitude if x != None else None)
df['Longitude'] = df['coordinates'].apply(lambda y: y.longitude if y != None else None)
```

- Then we **drop** the **coordinates** column, since it was only needed temporarily.
-
- (4) Next we convert the **dataframe** to a **CSV** and use **df.to_html()** to render it on our homepage:

```
df.to_csv("uploads/geocoded.csv", index=None)
return render_template("index.html", text=df.to_html(),
btn='download.html')
```

- Note that in **index.html**, we have a **<div class= “output”>** that includes some **Jinja** code, including *text placement* “{{text | safe}}”, returning **html** as a string:

```
<div class="output">
  {{text|safe}}
  {% include btn ignore missing %}
</div>
```

- The code “**{% include btn ignore missing %}**” handles the “**btn='download.html'**” segment from **app.py**. It’s different from the *text* code above it, and it renders HTML similarly to when we *extend* some HTML from a base.html file.
- It’s different from *extending* in that *extending* will render the HTML code *as soon as* the user visits the page. We don’t want this to be displayed right away, which is why we’re saying “**ignore missing**”.
- It will *only be included* when the user is *visiting the /success-table* URL.
- Once the user has submitted a file and it’s been processed, then the **download.html** template will be rendered:

download.html:

```
<!DOCTYPE html>
<html lang="en">
<div class="download">
<a href="{{url_for('download')}}" target="blank"> <button class="btn"> Download
</button></a>
</div>
</html>
```

- When the file has been submitted and this HTML has been *rendered*, it will look like this:

The screenshot shows a web application titled "Super Geocoder". At the top, there is a message: "Please upload your CSV file. The values containing addresses should be in a column named **address** or *Address*". Below this is a file input field labeled "Choose File" with the placeholder "No fil...hosen" and a "Submit" button. A table displays five rows of geocoded data:

ID	Address	Name	Employees	Latitude	Longitude
0	3666 21st St San Francisco CA 94114 USA	Madeira	8	37.756489	-122.429343
1	735 Dolores St San Francisco CA 94119 USA	Bready Shop	15	NaN	NaN
2	332 Hill St San Francisco California 94114 USA	Super River	25	37.755725	-122.428601
3	3995 23rd St San Francisco CA 94114 USA	Ben's Shop	10	37.752965	-122.431714
4	1056 Sanchez St San Francisco California USA	Sanchez	12	37.752146	-122.429815

At the bottom right of the table, there is a blue "Download" button with a cursor pointing at it.

- Now when the user *presses* the “Download” button, the “{{url_for('download')}}”, i.e. it’s **function**, is triggered in [app.py](#).
-
- [Back to “app_ver2.py”](#): Now to our URL for @app.route("/download-file/"), we have our **def download function**:

```
@app.route("/download-file/")
def download():
    return send_file("uploads/geocoded.csv", attachment_filename='yourfile.csv',
as_attachment=True)
```

- When the user visits this URL, we want to use a **send_file()** method to send the file that we generated earlier in the **success_table** function.
 - We want to send this to the user under the name “**yourfile.csv**”, *as an attachment*.
-
- At this point, we’re basically down with the program, at least in terms of its *basic functionality*. However, we may still have some small issues with it as-is.
 - For example, what if we choose a CSV file that *doesn’t have an “Address” column*? When we try and create a new column called “coordinates” and try to access “Address”, we will get an error.
 - We may want to add some *checks* to make sure the necessary column exists.
 - If the column exists, we run as normal, but if it doesn’t we may want to return a message to the user saying that the file contains no “Address” column.

Solution, Part 2:

- For “[app_ver3.py](#)”, let’s start by putting a section of our code inside a **try-except** check:

```
from flask import Flask, render_template, request, send_file
from geopy.geocoders import ArcGIS
import pandas

app=Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route('/success-table', methods=['POST'])
def success_table():
    if request.method=='POST':
        file=request.files['file']
        try: ← ← ←
            df=pandas.read_csv(file)
            gc=ArcGIS()
            df["coordinates"]=df["Address"].apply(gc.geocode)
            df['Latitude'] = df['coordinates'].apply(lambda x: x.latitude if x != None else None)
            df['Longitude'] = df['coordinates'].apply(lambda y: y.longitude if y != None else None)
            df=df.drop("coordinates", 1)
            df.to_csv("uploads/geocoded.csv", index=None)
            return render_template("index.html", text=df.to_html(),
btn='download.html')
        except: ← ← ←
            return render_template("index.html", text="Please make sure you have
an address column in your CSV file!") ← ← ←

@app.route("/download-file/")
def download():
    return send_file("uploads/geocoded.csv", attachment_filename='yourfile.csv',
as_attachment=True)

if __name__=="__main__":
    app.run(debug=True)
```

app_ver4.py:

- Now, currently the downloadable file will always be named as the *string*, “**geocoded.csv**”.
 - That could cause some problems, because if *two or more users* are submitting data at the same time, you may have some **name clashes**.
 - One thing you could do is use a **datetime module** to generate unique names for every file:

```
from flask import Flask, render_template, request, send_file
from geopy.geocoders import ArcGIS
import pandas
import datetime < < <
...
...
@app.route('/success-table', methods=['POST'])
def success_table():
    global filename < < <
    if request.method=='POST':
        file=request.files['file']
        try:
            df=pandas.read_csv(file)
            gc=ArcGIS()
            df["coordinates"]=df["Address"].apply(gc.geocode)
            df['Latitude'] = df['coordinates'].apply(lambda x: x.latitude if x != None else None)
            df['Longitude'] = df['coordinates'].apply(lambda y: y.longitude if y != None else None)
            df=df.drop("coordinates", 1)
            filename = datetime.datetime.now().strftime("uploads/%Y-%m-%d-%H-%M-%S-%f" + ".csv") < < <
            df.to_csv(filename, index=None) < < <
            return render_template("index.html", text=df.to_html(),
btn='download.html')
        except:
            return render_template("index.html", text="Please make sure you have
an address column in your CSV file!")

@app.route("/download-file/")
def download():
    return send_file(filename, attachment_filename='yourfile.csv',
as_attachment=True) < < <
```

- We **import datetime** up top. We also pass a *global variable*, **global filename** so that we can generate it inside of **success_table** and then *pass it to download* later.
- We use string formatting and datetime to create a unique file name every time the code is run.

Section 34: Bonus Exercises: A series of 103 bonus exercises (as is Section 38).

Page intentionally left blank.

Section 35: Bonus App: Building an English Thesaurus:

Demo of the Interactive English Dictionary App:

- From the instructor's description of what we've learned in the course, and the fact that he said something along the lines of "this will be your first app", make me think that in an earlier version of this course, this really was the first application. As such, I have a feeling it will be somewhat simple compared to many of the other apps we've created before this.
- We're going to create a program where you can **input an English word**, and the program will **return the definition of that word in English**.
- In his file tree (in Atom), he has a folder titled "teaching", and two files inside of that:
 - "app1.py" and "data.json".
- He ran the program **app1.py** from the terminal, and a prompt line came up: "**Enter word:**"
- He then typed in "**mountain**". The program returned **two definitions of mountain**:
 - "A feature of the earth's surface that rises high above the base and has generally steep slopes and a relatively small summit area."
 - "A great number or large amount of things placed in a pile."
- He ran the program again and entered the word "**mathematics**" to show an example of a word with **only one meaning**:
 - "The science that deals with concepts such as quantity, structure, space, and change."
- The program also has other features, such as if you input "**rainn**" with two N's, the program will ask:
 - "**Did you mean rain instead? Enter Y if yes, or N if no:**"
- If you enter a very *random* word (he used the example "**ababaaaaaaab**"), you get a message saying:
 - "**The word doesn't exist. Please double check it.**"
- If you input a word with a random mix of uppercase and lowercase letters, you'll still get the definition for the word.
- At the end that one could *extend* the functionality of this program by turning it into a *web application*, with a web interface for users to use.
 - The fact that he mentioned web applications in the future tense once again makes me think that in an earlier version of the course, this really was one of the first apps you make.

Know Your Dataset:

Note: Resources for this lecture include the file “**data.json**”.

- We create a folder for our program (I named mine “English_Thesaurus”), and inside of that we place the file **data.json**.
 - This file contains all of the words and their definitions.
 - Any **.py** files will also be placed here.
- This dataset of words and definitions was found online. Its size is around 5mb.
- This is a JSON file. JSON is like a language, or rather a set of rules on how to format some data.
- It starts with **curly brackets** (“{”), followed by sets of **keys** and **values**, similar to a **Python dictionary**.
- The **key** is separated from the **value** by a **colon** (“:”).
- Looking closely at the **values** in this case, they are contained inside of **square brackets** (“[“]) similar to a **Python list**. This is useful for cases where a word has more than one definition; the various definitions can be *listed* inside of this *list*.
 - If we use **CTRL + F** inside of the interpreter and search for one of the example words from the previous lecture (let’s say, “rain”), it will take us to the **key** of “rain”, followed by its **value**, a list of definitions for “rain”.
- Now we need to figure out how to **load this data into Python**, as a specific Python datatype. So, what datatype will be most appropriate for a JSON file? We’ll cover that in the next lecture.

Loading JSON Data:

- We can treat this JSON file as a **Python dictionary**, and once we load it in, it will be very easy to access a **value** by inputting its **key**.
- So how do we load this into a Python dictionary?
- The instructor started off by opening a **Python interactive shell** by running **python3**.
 - He then ran **>>> import json**.
 - We then want to use a **json.load()** method. To show what this does, he ran **help(json.load)**, showing that it gets a “**file-like object**” (called “**fp**” here) containing a JSON document.
 - So next he ran **data = json.load(open("data.json", 'r'))**, passing the filepath of the file.
 - Since we should already be running our terminal from the same location, we just need to pass the file’s name.
 - The ‘**r**’ is for *read mode*, but is optional because *read* is the default.
 - If you run **type(data)** now, it will return ‘**dict**’. You can print this out if you want, and it’ll print the entire contents of the JSON file.
 - You can also run **data["rain"]** and it’ll output the *list* of definitions, or the **value**:
 - [‘Precipitation in the form of liquid water drops with diameters greater than 0.5 millimeters.’, ‘To fall from the clouds in drops of water.’]

Returning the Definition of a Word:

- We start by creating a very basic version of our app, that **loads the JSON file**, asks for a **user input**, then outputs the **value/definition** that corresponds to the input's key:

```
import json

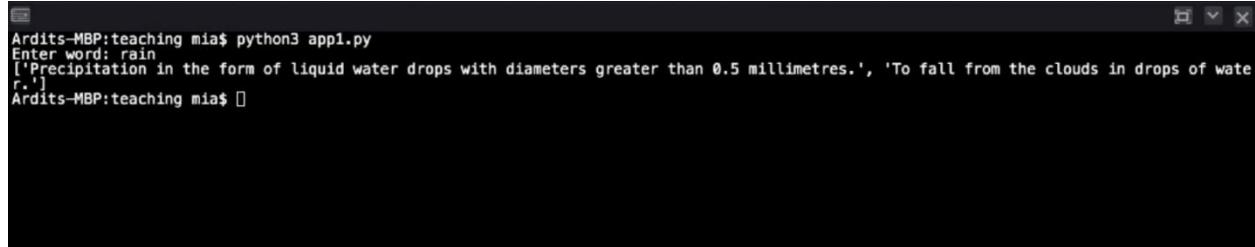
data = json.load(open("data.json"))

def define(word):
    return data[word]

word = input("Enter word: ")

print(define(word))
```

- When we run this program from the terminal and enter the word “**rain**”, the output is this:



A screenshot of a terminal window titled "Ardits-MBP:teaching mia\$". It shows the command "python3 app1.py" being run, followed by the user input "Enter word: rain". The program then prints two definitions: "'Precipitation in the form of liquid water drops with diameters greater than 0.5 millimetres.', 'To fall from the clouds in drops of water.'". The terminal window has a dark background and light-colored text.

- Notice that it outputs a **list** of definitions tied to the key “**rain**”:
 - “[‘Precipitation in the form of liquid water drops with diameters greater than 0.5 millimeters.’, ‘To fall from the clouds in drops of water.’]”
-
- However, inputting a non-existing word (such as “asdfaaaa”) returns a **key error**. We don’t want to show this to the user because it’s not very readable.
- In the next lecture, we’ll handle this error.

Non-Existing Words:

- So now instead of showing our users that ugly *key error* as output, we'll replace that with a message saying “The **word doesn't exist. Please double check it.**”
 - To show some behind-the-scenes logic here, he ran **>>> “rain” in data** in the interactive shell and it returned **True**.
 - So let's make use of this as a *conditional* inside of our **define** function:

```
import json

data = json.load(open("data.json"))

def define(word):
    if word in data: ←←←
        return data[word] ←←←
    else: ←←←
        return "The word doesn't exist. Please double check it." ←←←

word = input("Enter word: ")

print(define(word))
```

- Now when we input a nonsense word, we get a useful and human-readable error message.
- However, now if we enter “**RAIN**”, it checks for the all-caps version in **data** and says that it doesn't exist. So next up we're going to handle those cases.

Dealing with Case-Sensitive Words:

- Currently our program thinks that words with capital letters don't exist in the data. We need to handle that.
- The instructor ran **>>> dir(str)** to show the different *methods* that can be applied to a string. One of them is “**.lower()**”. So let's add that to our code:

```
import json

data = json.load(open("data.json"))

def define(word):
    if word.lower() in data: ← ← ←
        return data[word.lower()] ← ← ←
    else:
        return "The word doesn't exist. Please double check it."

word = input("Enter word: ")

print(define(word))
```

- Now we can run existing words in any combination of uppercase and lowercase, and still get the correct values.
-
- Now, what if instead of “rain”, the user types in an extra N by mistake: “rainn”. We may want to try and calculate what the user might've meant, so we'll handle that in the next lecture.

Calculating the Similarity Between Words:

- Now we want to take into consideration when a user *mistypes* something.
- If we don't know how to do something like this (or how to do other things with Python) a good best-practice approach is to do some research on Google. You might be able to find something like:
 - A **Python library** that does what you want, or
 - some **source code** or a **function** that you can use.
 - Most often you'll find a *library* that does what you need.
- In our case, we can use a *standard library* called "**difflib**" that does what we need.
-
- The instructor started out in the interactive shell, running:
 - **>>> import difflib**
 - **>>> from difflib import SequenceMatcher**
 - **>>> SequenceMatcher(None, "rainn", "rain").ratio()**
 - (Note: we pass "None" as the first argument; this is the **junk** argument, which uses a function as input for cases where we might have "junk" characters, new-lines, etc).
 - (Note: running this without ".ratio()" will just return a "SequenceMatcher" object and a memory address).
 - This returns "**0.8888888888888888**", which indicates a similarity between the two strings "rainn" and "rain" on a scale from 0 to 1.
-
-
- However, our app needs to take the input word and *compare it to every key in the dictionary*.
 - Instead we'll want to use a feature called **get_close_matches**, but we'll save this for the next lecture.

Best Matches Out of a List of Words:

- Now we want to get the ***most similar*** word out of a list.
- In the interactive shell, we run:
 - `>>> from difflib import get_close_matches`, then:
 - `>>> help(get_close_matches)` shows us that the arguments we can pass into it are: **word**, **possibilities**, **n=3**, and **cutoff=0.6**.
 - **word** is a sequence for which close matches are desired (typically a string).
 - **possibilities** is a list of sequences against which to match **word** (typically a *list of strings*).
 - Optional arg **n** (default 3) is the maximum number of close matches to return. **n** must be **> 0**.
 - Optional arg **cutoff** (default 0.6) is a float in [0,1]. Possibilities that don't score at least that similar to word are ignored.
 - The best (no more than **n**) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.
- We then ran:
 - `>>> get_close_matches("rainn", ["help", "pyramid", "rain"])`,
- which returns:
 - `['rain']`
-
- So now we're able to get a close match out of a list of words, but how do we get one out of our **keys**?
 - We still have our **data** variable loaded (or if you don't follow all the interactive shell commands from the "Loading JSON Data" lecture until now).
 - We can use a method of dictionaries, `data.keys()` to extract a list of all keys.
 - If we run `type(data.keys())`, it returns "`<class 'dict_keys'>`", but it *behaves like a list*.
- So now let's run:
 - `>>> get_close_matches("rainn", data.keys())`,
- and this returns:
 - `['rain', 'train', 'rainy']`
 - Note: If we pass the same command with an argument of **n = 5**, we get even more matches, such as:
 - `['rain', 'train', 'rainy', 'grain', 'drain']`
- It's important to note that the list is **ordered**, and the first word is the one with the highest ratio of similarity, so we're only interested in the first word. So we can run:
 - `>>> get_close_matches("rainn", data.keys())[0]`,
- to just get:
 - `'rain'`
-
- Now we just need to add this functionality to our program.

Finding the Most Similar Word from a Group of Words:

- Note that if you run something like `>>> get_close_matches("rainn", data.keys())`, you get an empty list ("[]") returned to you. We can use this to make our conditional statements smarter.
- We may also want to pass a higher cutoff value such as `cutoff=0.8` into `get_close_matches`. Here are two cases and their return outputs:
 - `>>> get_close_matches("cacacaa", data.keys(), cutoff=0.5),`
 - ['acacia', 'Sciacca', 'cacao']
 - `>>> get_close_matches("cacacaa", data.keys(), cutoff=0.8),`
 - []
- Now to our code:
- We should leave the first conditional, “`if word in data:`”, as-is, since it’s probably best for this conditional to be applied first. If the word *isn’t* in the list, then we can check for similar words.
- To do this, we’ll add an `elif` statement to check that the `length` of `get_close_matches > 0`:

```
import json
from difflib import get_close_matches

data = json.load(open("data.json"))

def define(word):
    word = word.lower()
    if word in data:
        return data[word]
    elif len(get_close_matches(word, data.keys())) > 0: ← ← ←
        return "Did you mean %s instead?" % get_close_matches(word,
data.keys(), cutoff=0.8)[0] ← ← ←
    else:
        return "The word doesn't exist. Please double check it."

word = input("Enter word: ")

print(define(word))
```

- This checks to make sure that `get_close_matches` isn’t returning an *empty list* (no matches), then asks the user if they meant the *highest similarity word* instead.
- However, at this point the program only asks them if they meant another word instead. We should add the ability for the user to respond.
- We’ll do this by *executing another `input function`* inside this conditional.

Getting Confirmation from the User:

- So now let's add an input function and a message to prompt the user to enter 'Y' or 'N':

```
import json
from difflib import get_close_matches

data = json.load(open("data.json"))

def define(word):
    word = word.lower()
    if word in data:
        return data[word]
    elif len(get_close_matches(word, data.keys())) > 0:
        return input("Did you mean %s instead? Enter 'Y' if yes, or 'N' if no." % get_close_matches(word, data.keys(), cutoff=0.8)[0]) ← ← ←
    else:
        return "The word doesn't exist. Please double check it."

word = input("Enter word: ")

print(define(word))
```

- With this, we have an input function, but it doesn't actually store the input as a variable anywhere. It's just data on-the-fly. So let's set it to a variable, **yn**:

```
def define(word):
    word = word.lower()
    if word in data:
        return data[word]
    elif len(get_close_matches(word, data.keys())) > 0:
        yn = input("Did you mean %s instead? Enter 'Y' if yes, or 'N' if no." % get_close_matches(word, data.keys(), cutoff=0.8)[0]) ← ← ←
    else:
        return "The word doesn't exist. Please double check it."
```

- Now either 'Y' or 'N' will be stored in that variable. Now we need to *process* that variable:

```
    yn = input("Did you mean %s instead? Enter 'Y' if yes, or 'N' if no: "
% get_close_matches(word, data.keys(), cutoff=0.8)[0])
    if yn == "Y" or "y": ← ← ←
        return data[get_close_matches(word, data.keys(), cutoff=0.8)[0]] ←
    else: ← ← ←
        return "The word doesn't exist. Please double check it." ← ← ←
```

-
- Now let's check what happens if you type in "rainn" but then select 'N':
-

- Now let's handle what happens if you type in "rainn" but then select 'N', or if they select anything other than 'Y' or 'N':

```

elif len(get_close_matches(word, data.keys())) > 0:
    yn = input("Did you mean %s instead? Enter 'Y' if yes, or 'N' if no: "
% get_close_matches(word, data.keys(), cutoff=0.8)[0])
    if yn.upper() == "Y":
        return data[get_close_matches(word, data.keys(), cutoff=0.8)[0]]
    elif yn.upper() == "N": ← ← ←
        return "The word doesn't exist. Please double check it." ← ← ←
    else: ← ← ←
        return "We didn't understand your entry." ← ← ←

```

- Note that I added some `.upper()` methods on `yn` to handle cases where the user might just type in the lowercase version of each letter.
- We've now more-or-less handled all the normal possibilities our program might encounter.
 - However, there's one more thing we can improve upon: we can format the output (the definitions) to appear nicer on the screen.
 - Currently our definitions are being output as a *list*.
 - Instead of that, we want our definitions to be *listed* without the square brackets and quotation marks, with *new lines* in between.

Optimizing the Final Output:

- Now we want the output to be a little more user friendly. Currently it's a *Python list*.
- We want it to be presented as separate lines (without square brackets or quotation marks) in the terminal.
- We can do that by iterating through the list.
-
- Instead of just *printing*, let's start by *storing* our list object in a variable, **output**:

```
word = input("Enter word: ")  
  
output = define(word) ← ← ←
```

- Now we can use a **for-loop** to *iterate* through the list and print each item:

```
output = define(word)  
  
for item in output: ← ← ←  
    print(item) ← ← ←
```

- So this lists our items on new lines. However, in cases where we were returning a message for users, we now have a problem: sometimes it will print each letter of a string message on a new line:

```
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE  
  
d  
o  
e  
s  
n,  
t  
  
e  
x  
i  
s  
t  
. 
```

- To fix this, let's separate the two cases by **datatype**; *list* vs *string*.

```
output = define(word)  
  
if type(output) == list: ← ← ←  
    for item in output:  
        print(item)  
else: ← ← ←  
    print(output) ← ← ←
```

- Now our program can handle both *list* and *string* outputs.
-
- Now, our program works well, but it's user interface is still **command line**.

- Now, our program works well, but it's user interface is still **command line**.
 - It doesn't have a **GUI**, and it's not a **web application** either.
 - We can *extend* this program to turn it into a **web application** if we wish. On a live website, we can have the user input their word into the webpage and have the output displayed via HTML.
 - We can also use a **database** instead of our JSON file. A database is a better choice for datasets that are very big (as opposed to our 5mb JSON file).

Section 36: Bonus App: Building a Website Blocker:

Demo of the Website Blocker:

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