Welcome to Class (0:05)

Friday, September 27, 2019 4:47 PM

Class Objectives

- Students will be able to add, commit, and push code up to GitHub from the command line.
- Students will be able to create and use Python dictionaries.
- Students will be able to read data in from a dictionary.
- Students will be able to use list comprehensions.
- Students will be able to write and re use Python function.
- Students will have a firm understanding of coding logic and reasoning.

Writing CSV Files

Monday, September 23, 2019 5:07 PM

Python can write data into CSV files too. This allows users to easily modify and/or create datasets based up previous data

Open write.py

- UNC-CHA-DATA-PT-09-2019-U-C
- 03-Python
- 2
- Activities
- 09-Ins_WriteCSV
- Solved

- The syntax for writing into a CSV file is similar to that used to read data in from an external file.
- First, the code references the path that will point into the CSV file the user would like to write to.

```
# Specify the file to write to
output_path = os.path.join('output', 'new.csv')
```

```
# Open the file using "write" mode. Specify the variable to hold the contents
with open(output_path, 'w', newline='') as csvfile:

    # Initialize csv.writer
    csvwriter = csv.writer(csvfile, delimiter=',')

    # Write the first row (column headers)
    csvwriter.writerow(['First Name', 'Last Name', 'SSN'])

    # Write the second row
    csvwriter.writerow(['Caleb', 'Frost', '505-80-2901'])
```

- Next, the **with open()** statement is used once more but with one significant difference. The parameter **'w'** is passed instead to inform Python to write to the file.

```
# Specify the file to write to
output_path = os.path.join('output', 'new.csv')
```

```
# Specify the file to write to
output_path = os.path.join('output', 'new.csv')

# Open the file using "write" mode. Specify the variable to hold the contents
with open(output_path, 'w', newline='') as csvfile:

    # Initialize csv.writer
    csvwriter = csv.writer(csvfile, delimiter=',')

    # Write the first row (column headers)
    csvwriter.writerow(['First Name', 'Last Name', 'SSN'])

    # Write the second row
    csvwriter.writerow(['Caleb', 'Frost', '505-80-2901'])
```

- Instead of **csv.reader()**, **csv.writer()** is used to inform Python that this application will be writing code into an external CSV file.

```
# Specify the file to write to
output_path = os.path.join('output', 'new.csv')

# Open the file using "write" mode. Specify the variable to hold the contents
with open(output_path, 'w', newline='') as csvfile:

    # Initialize csv.writer
    csvwriter = csv.writer(csvfile, delimiter=',')

    # Write the first row (column headers)
    csvwriter.writerow(['First Name', 'Last Name', 'SSN'])

    # Write the second row
    csvwriter.writerow(['Caleb', 'Frost', '505-80-2901'])
```

- To write a new row into a CSV file, simply use the **csv.writerow(<DATA LIST>)** function and pass in an array of data as the parameter

```
# Specify the file to write to
output_path = os.path.join('output', 'new.csv')

# Open the file using "write" mode. Specify the variable to hold the contents
with open(output_path, 'w', newline='') as csvfile:

    # Initialize csv.writer
    csvwriter = csv.writer(csvfile, delimiter=',')

    # Write the first row (column headers)
    csvwriter.writerow(['First Name', 'Last Name', 'SSN'])

    # Write the second row
    csvwriter.writerow(['Caleb', 'Frost', '505-80-2901'])
```

Zipping Lists

Monday, September 23, 2019 5:07 PM

It is possible to write new rows of data into a CSV file using a bunch of `csv.writerow()` statements, Python users can far more efficiently write data into a new CSV file by using the `zip()` function.

`zip()` takes in a series of lists as its parameters and joins them together into a stack.

Open Zipper.py

- UNC-CHA-DATA-PT-09-2019-U-C
- 03-Python
- 2
- Activities
- 10-Ins_Zip
- Solved

- This application has three lists, all of which pertain to each other and are of the same length. By zipping these lists together, there is now a single joined list whose indexes reference all three of the lists inside

```
import csv
import os

# Three Lists
indexes = [1, 2, 3, 4]
employees = ["Michael", "Dwight", "Meredith", "Kelly"]
department = ["Boss", "Sales", "Sales", "HR"]

# Zip all three lists together into tuples
roster = zip(indexes, employees, department)

# save the output file path
output_file = os.path.join("..", "Files","output","output.csv")

# open the output file, create a header row, and then write the zipped object to the csv
with open(output_file, "w", newline="") as datafile:
    writer = csv.writer(datafile)

    writer.writerow(["Index", "Employee", "Department"])

    writer.writerows(roster)
```

Exercise: Udemy Zip

Monday, September 23, 2019 5:07 PM

Overview

Students will take a large dataset from Udemy, clean it up, and create a new CSV file that is far easier to comprehend

	A	B	C	D	E	F
1	Title	Course Price	Subscribers	Reviews Left	Percent of Reviews	Length of Course
2	Learn Web Designing & HTML5/CSS3 Essentials in 4-Hours	75	43285	525	0.01	4
3	Learning Dynamic Website Design - PHP MySQL and JavaScript	50	47886	285	0.01	12.5
4	ChatBots: Messenger ChatBot with API.AI and Node.JS	50	2577	529	0.21	4.5
5	Projects in HTML5	60	8777	206	0.02	15.5
6	Programming Foundations: HTML5 + CSS3 for Entrepreneurs 2015	20	23764	490	0.02	5.5
7	How To Make A Wordpress Website 2017 | Divi Theme Tutorial	40	3541	202	0.06	4
8	Build Your Own Backend REST API using Django REST Framework	50	2669	112	0.04	5.5
9	Angular and Firebase - Build a Web App with Typescript	150	1966	359	0.18	5
10	Web Development Masterclass - Complete Certificate Course	200	4090	178	0.04	19.5
11	Spring Boot Tutorial For Beginners	40	2578	210	0.08	5.5
12	The Complete Bootstrap Masterclass Course - Build 4 Projects	195	24978	540	0.02	7

Instructor Demo [web_starter.csv, web_solved.py]

Files: File in slack, web_starter.csv

Instructions

- Create a Python application that reads the data on Udemy Web Development offerings.
- Then store the contents of the Title, Price, Subscriber Count, Number of Reviews, and Course Length into Python Lists.
- Then zip these lists together together.
- Finally, write the contents of your extracted data into a CSV. Make sure to include the titles of these columns in your CSV.

Notes

- Windows user may get an UnicodeDecodeError, to avoid this file pass in encoding="utf8" as an additional parameter when reading in the file
- As, with many datasets, the file does not include the header line. Use the below as a guide on the columns: "id,title,url,isPaid,price,numSubscribers,numReviews,numPublished Lectures,instructionalLevel,contentInfo,publishedTime"

Review: Udemy Zip

Monday, September 23, 2019 5:08 PM

Key Points

- There are six empty lists created at the start of this application, all of which will be used to hold specific data taken from within the original CSV and ultimately be zipped together before being written to a new CSV file.
- For every new row read in from the original CSV file, new data is appended into the lists from earlier. In the cases of percent and length, the data is being altered before being placed into their respective list.

```
# Lists to store data
title = []
price = []
subscribers = []
reviews = []
review_percent = []
length = []

with open(udemy_csv, newline="") as csvfile:
    csvreader= csv.reader(csvfile, delimiter=",")
    for row in csvreader:
        # Add title
        title.append(row[1])

        # Add price
        price.append(row[4])

        # Add number of subscribers
        subscribers.append(row[5])

        # Add amount of reviews
        reviews.append(row[6])

        # Determine percent of review left to 2 decimal places
        percent = round(int(row[6])/int(row[5]), 2)
        review_percent.append(percent)

        # Get length of the course to just a number
        new_length = row[9].split(" ")
        length.append(new_length[0])
```

- Once all data has been read, the lists are zipped together and written into a new CSV file with a header row being written in beforehand.

```
# Zip lists together
cleaned_csv = zip(title, price, subscribers, reviews, review_percent,length)

# Set variable for output file
output_file = os.path.join("web_final.csv")

#  Open the output file
with open(output_file, "w", newline="") as datafile:
    writer = csv.writer(datafile)

    # Write the header row
    writer.writerow(["Title", "Course Price", "Subscribers", "Reviews Left",
                     "Percent of Reviews", "Length of Course"])

    # Write in zipped rows
    writer.writerows(cleaned_csv)
```

Everyone Do: Cereal Cleaner (0:15)

Friday, September 27, 2019 4:57 PM

Files: cereal.csv and cereal_solved.py

Exercise Overview: Reads through the cereal.csv file and finds the cereals that contain five grams or more, prints the data from those rows to the terminal

- Importing of dependencies is done before anything else in the code. This is done to keep the code clean and makes it easier to read/understand in the future.

```
# Importing Modules
import os
import csv
```

```
# Path to csv file
cereal_csv = os.path.join("C:\\Users\\dedwa\\OneDrive\\Desktop\\DataViz\\Class 3.3\\Data\\cereal.csv")

# Open and read csv
with open(cereal_csv, newline="") as csvfile:
    csvreader = csv.reader(csvfile, delimiter=",")

    # Read the header row first (skip this part if there is no header)
    # Next() function returns the next row of an iterable object as a list
    csv_header = next(csvfile)
    print(f"Header: {csv_header}")

    # Read through each row of data after the header
    for row in csvreader:

        # Convert row to float and compare to grams of fiber
        # row[7] is referring to the 8 columns in the csv file that is read in
        if float(row[7]) >= 5:
            print(row)
```

- **`cereal_csv_path`** is named in a very self-explanatory way. This is to ensure that future developers know precisely what this variable references.

```
# Importing Modules
import os
import csv

# Path to csv file
cereal_csv = os.path.join("C:\\Users\\dedwa\\OneDrive\\Desktop\\DataViz\\Class 3.3\\Data\\cereal.csv")

# Open and read csv
with open(cereal_csv, newline="") as csvfile:
    csvreader = csv.reader(csvfile, delimiter=",")

    # Read the header row first (skip this part if there is no header)
    # Next() function returns the next row of an iterable object as a list
    csv_header = next(csvfile)
    print(f"Header: {csv_header}")

    # Read through each row of data after the header
    for row in csvreader:

        # Convert row to float and compare to grams of fiber
        # row[7] is referring to the 8 columns in the csv file that is read in
        if float(row[7]) >= 5:
            print(row)
```

- Same deal for **`csv_reader`** which is used to hold the data read in from the CSV file.

```
# Importing Modules
import os
import csv

# Path to csv file
cereal_csv = os.path.join("C:\\Users\\dedwa\\OneDrive\\Desktop\\DataViz\\Class 3.3\\Data\\cereal.csv")

# Open and read csv
with open(cereal_csv, newline="") as csvfile:
    csvreader = csv.reader(csvfile, delimiter=",")

    # Read the header row first (skip this part if there is no header)
    # Next() function returns the next row of an iterable object as a list
    csv_header = next(csvfile)
    print(f"Header: {csv_header}")

    # Read through each row of data after the header
    for row in csvreader:

        # Convert row to float and compare to grams of fiber
        # row[7] is referring to the 8 columns in the csv file that is read in
        if float(row[7]) >= 5:
            print(row)
```

- The **`next()`** function returns the next row of an iterable object, which in this case in our header

```
# Importing Modules
import os
import csv

# Path to csv file
cereal_csv = os.path.join("C:\\Users\\dedwa\\OneDrive\\Desktop\\DataViz\\Class 3.3\\Data\\cereal.csv")

# Open and read csv
with open(cereal_csv, newline="") as csvfile:
    csvreader = csv.reader(csvfile, delimiter=",")

    # Read the header row first (skip this part if there is no header)
    # Next() function returns the next row of an iterable object as a list
    csv_header = next(csvfile)
    print(f"Header: {csv_header}")

    # Read through each row of data after the header
    for row in csvreader:

        # Convert row to float and compare to grams of fiber
        # row[7] is referring to the 8 columns in the csv file that is read in
        if float(row[7]) >= 5:
            print(row)
```

- In order to check through all of the rows in the CSV file and find those that have more than five grams of fiber in them, a `for` loop containing an `if` statement is used.
- After looking through the CSV file, we know that the **`fiber`** column is stored within the 8th column. This means the **`if`** statement must check values stored at index 7.


```
# Importing Modules
import os
import csv

# Path to csv file
cereal_csv = os.path.join("C:\\Users\\dedwa\\OneDrive\\Desktop\\DataViz\\Class 3.3\\Data\\cereal.csv")

# Open and read csv
with open(cereal_csv, newline="") as csvfile:
    csvreader = csv.reader(csvfile, delimiter=",")

    # Read the header row first (skip this part if there is no header)
    # Next() function returns the next row of an iterable object as a list
    csv_header = next(csvfile)
    print(f"Header: {csv_header}")

    # Read through each row of data after the header
    for row in csvreader:

        # Convert row to float and compare to grams of fiber
        # row[7] is referring to the 8 columns in the csv file that is read in
        if float(row[7]) >= 5:
            print(row)
```

Dictionaries (0:05)

Friday, September 27, 2019 4:58 PM

Files: dictionaries.py

Another data type that is commonly used in Python the dictionary, a.k.a. "dict".

- Dictionaries can contain multiple values and types of data within them.
- Dictionaries store data in key-value pairs. The key in a dictionary is a string that can be referenced in order to collect the value it is associated with.

- A pair of curly braces is used to create a dictionary: **`someVariable = {}`**
- Values can be added to dictionaries at declaration by
  - creating a key that is stored within a string, (“name”)
  - following it with a colon, (:)
  - and then placing the value desired afterwards. (“Tom Cruise”)
- Referencing a value within a dictionary is as simple as calling the dictionary **(actor)** and following it up with a pair of brackets containing the key **(“name”)** for the value desired.

```
# Unlike lists, dictionaries store information in pairs
# ------------------------------------------------------------

# A list of actors
actors = ["Tom Cruise", "Angelina Jolie","Kristen Stewart","Denzel Washington"]

# A dictionary of an actor
actor = {"name": "Tom Cruise"}
print(f'{actor["name"]}')
# ------------------------------------------------------------
```

- Dictionaries can hold multiple pieces of information by following up each key-value pairing with a comma and then placing another key-value pair afterwards.

- Keys are “name”, “genre”, “nationality”

```
# A dictionary can contain multiple pairs of information
actress = {
    "name": "Angelina Jolie",
    "genre": "Action",
    "nationality": "United States"
}
# ----------------------------------------------------------------

# A dictionary can contain multiple types of information
another_actor = {
    "name": "Sylvester Stallone",
    "age": 62,
    "married": True,
    "best movies": ["Rocky","Rocky 2","Rocky 3"]
}
print(f'{another_actor["name"]} was in {another_actor["best movies"][0]}')
# ----------------------------------------------------------------

# A dictionary can even contain another dictionary
film = {
    "title": "Interstellar",
    "revenues": {
        "United States": 360,
        "China": 250,
        "United Kingdom": 73
    }
}
print(f'{film["title"]} made {film["revenues"]["United States"]}'" in the US.")
# ----------------------------------------------------------------
```

- Dictionaries can also store lists. They can be accessed by first calling the key and then indexing the list.

```python
# A dictionary can contain multiple pairs of information
actress = {
    "name": "Angelina Jolie",
    "genre": "Action",
    "nationality": "United States"
}
# ------------------------------------------------------------

# A dictionary can contain multiple types of information
another_actor = {
    "name": "Sylvester Stallone",
    "age": 62,
    "married": True,
    "best movies": ["Rocky","Rocky 2","Rocky 3"]
}
print(f'{another_actor["name"]} was in {another_actor["best movies"][0]}')
# ------------------------------------------------------------

# A dictionary can even contain another dictionary
film = {
    "title": "Interstellar",
    "revenues": {
        "United States": 360,
        "China": 250,
        "United Kingdom": 73
    }
}
print(f'{film["title"]} made {film["revenues"]["United States"]}'" in the US.")
# ------------------------------------------------------------
```

- Dictionaries can also contain other dictionaries. In order to access the values inside nested dictionaries, simply add another key to the reference.

```python
# A dictionary can contain multiple pairs of information
actress = {
    "name": "Angelina Jolie",
    "genre": "Action",
    "nationality": "United States"
}
# ----------------------------------------------------------------

# A dictionary can contain multiple types of information
another_actor = {
    "name": "Sylvester Stallone",
    "age": 62,
    "married": True,
    "best movies": ["Rocky","Rocky 2","Rocky 3"]
}
print(f'{another_actor["name"]} was in {another_actor["best movies"][0]}')
# ----------------------------------------------------------------

# A dictionary can even contain another dictionary
film = {
    "title": "Interstellar",
    "revenues": {
        "United States": 360,
        "China": 250,
        "United Kingdom": 73
    }
}
print(f'{film["title"]} made {film["revenues"]["United States"]}'" in the US.")
# ----------------------------------------------------------------
```

Exercise: Hobby-Book (0:12)

Friday, September 27, 2019 4:58 PM

Starter File(s):

Instructions

- Create a dictionary that stores the following
  - Your name
  - Your age
  - A list of a few of your hobbies
  - A dictionary of a few days and the time you wake up on those days
- Print out
  - Your name
  - How many hobbies you have
  - A Time you get up during the week

Review: Review Hobby-Book (0:05)

Friday, September 27, 2019 4:59 PM

Key Points

- A variable called **my_info** stores the primary dictionary as noted by the curly braces.
- The keys are "name", "occupation", "age", "hobbies" and "wake-up". Their values are what is stored after the colons with each new key-value pair being separated by a comma.

```
# Dictionary full of info
my_info = {"name": "Dominique",
           "age": 30,
           "hobbies": ["Work", "School", "Hanging out with GF"],
           "wake-up": {"Mon": 6, "Friday": 6, "Saturday": 7, "Sunday": 8}}

# Print out results are stored in the dictionary
print(f'Hello I am {my_info["name"]}')
print(f'I have {len(my_info["hobbies"])} hobbies!')
print(f'On the weekend I get up at {my_info["wake-up"]["Saturday"]}')
```

- In order to find the number of values stored within the "hobbies" key, the `len()` function is used

```
# Dictionary full of info
my_info = {"name": "Dominique",
           "age": 30,
           "hobbies": ["Work", "School", "Hanging out with GF"],
           "wake-up": {"Mon": 6, "Friday": 6, "Saturday": 7, "Sunday": 8}}

# Print out results are stored in the dictionary
print(f'Hello I am {my_info["name"]}')
print(f'I have {len(my_info["hobbies"])} hobbies!')
print(f'On the weekend I get up at {my_info["wake-up"]["Saturday"]}')
```

List Comprehensions (0:10)

Friday, September 27, 2019 5:00 PM

Files: comprehensions.py

- We can treat `word variable` like an array, and use a **list comprehension** to turn it into a list of its own letters.

```
word = "halibut"

# Loop through each letter in the string
# and push to an array
letters = []
for letter in word:
    letters.append(letter)

print(letters)

# List comprehensions provide concise syntax for creating lists
letters = [letter for letter in word]

print(letters)

capital_letters = []
for letter in word:
    capital_letters.append(letter.upper())

print(capital_letters)

# List Comprehension for the above
capital_letters = [letter.upper() for letter in word]

print(capital_letters)

# We can also add conditional logic (if statements) to a list comprehension
july_temperatures = [87, 85, 92, 79, 106]
hot_days = []
for temperature in july_temperatures:
    if temperature > 90:
        hot_days.append(temperature)
print(hot_days)

# List Comprehension with conditional
hot_days = [temperature for temperature in july_temperatures if temperature > 90]

print(hot_days)
```

- We can create a new list of capitalized letters, by using a comprehension and calling `upper` on each letter.

```
word = "halibut"

# Loop through each letter in the string
# and push to an array
letters = []
for letter in word:
    letters.append(letter)

print(letters)

# List comprehensions provide concise syntax for creating lists
letters = [letter for letter in word]

print(letters)

# We can manipulate each element as we go
capital_letters = []
for letter in word:
    capital_letters.append(letter.upper())

print(capital_letters)

# List Comprehension for the above
capital_letters = [letter.upper() for letter in word]

print(capital_letters)

# We can also add conditional logic (if statements) to a list comprehension
july_temperatures = [87, 85, 92, 79, 106]
hot_days = []
for temperature in july_temperatures:
    if temperature > 90:
        hot_days.append(temperature)
print(hot_days)

# List Comprehension with conditional
hot_days = [temperature for temperature in july_temperatures if temperature > 90]

print(hot_days)
```

- We can filter data in addition to changing it.
    - Adding conditional logic, such as `if statements`, to a list comprehension allows us to select a certain value or range of values.

```python
word = "halibut"

# Loop through each letter in the string
# and push to an array
letters = []
for letter in word:
    letters.append(letter)

print(letters)

# List comprehensions provide concise syntax for creating lists
letters = [letter for letter in word]

print(letters)

# We can manipulate each element as we go
capital_letters = []
for letter in word:
    capital_letters.append(letter.upper())

print(capital_letters)

# List Comprehension for the above
capital_letters = [letter.upper() for letter in word]

print(capital_letters)
```

```python
# We can also add conditional logic (if statements) to a list comprehension
july_temperatures = [87, 85, 92, 79, 106]
hot_days = []
for temperature in july_temperatures:
    if temperature > 90:
        hot_days.append(temperature)
print(hot_days)

# List Comprehension with conditional
hot_days = [temperature for temperature in july_temperatures if temperature > 90]

print(hot_days)
```

Exercise: List Comprehensions (0:10)

Friday, September 27, 2019 5:00 PM

Starter File(s): comprehensions.py (different file from previous example)

Overview: Use list comprehensions to compose wedding invitations to send to 5 people

Instructions

- Create a list that prompts the user for the names of five people that you know
- Run the provided program. Note that nothing forces you to write the name "properly"—e.g., as "Jane" instead of "jAnE". You will use list comprehensions to fix this.
    - First, use list comprehensions to create a new list that contains the lowercase version of each of the names your user provided.
    - Then, use list comprehensions to create a new list that contains the title-cased versions of each of the names in your lower-cased list.

Review: List Comprehensions (0:08)

Friday, September 27, 2019 5:01 PM

- First code block simply declares a list to store names in, and then collects 5 names from the user.

```
#Declares a list to store names in
names = []

#Foor Loop that collects 5 names
for _ in range(5):
    name = input("Please enter the name of someone you know. ")
    names.append(name)


#Takes all the names from input, makes them lower case
lowercased = [name.lower() for name in names]

#Takes the lower case names, makes them title (upper case first letter)
titlecased = [name.title() for name in lowercased]

#Add each invitation to a list
invitations = [
    f"Dear {name}, please come to the wedding this Saturday!" for name in titlecased]

#Prints the list of invitations
for invitation in invitations:
    print(invitation)
```

- Used a list comprehension that calls `lower` on each name from the input.

```
#Declares a list to store names in
names = []

#Foor Loop that collects 5 names
for _ in range(5):
    name = input("Please enter the name of someone you know. ")
    names.append(name)


#Takes all the names from input, makes them lower case
lowercased = [name.lower() for name in names]

#Takes the lower case names, makes them title (upper case first letter)
titlecased = [name.title() for name in lowercased]

#Add each invitation to a list
invitations = [
    f"Dear {name}, please come to the wedding this Saturday!" for name in titlecased]

#Prints the list of invitations
for invitation in invitations:
    print(invitation)
```

- Used a list comprehension that calls `title` on each name in the lowercase list.

```
#Declares a list to store names in
names = []

#Foor Loop that collects 5 names
for _ in range(5):
    name = input("Please enter the name of someone you know. ")
    names.append(name)


#Takes all the names from input, makes them lower case
lowercased = [name.lower() for name in names]

#Takes the lower case names, makes them title (upper case first letter)
titlecased = [name.title() for name in lowercased]

#Add each invitation to a list
invitations = [
    f"Dear {name}, please come to the wedding this Saturday!" for name in titlecased]

#Prints the list of invitations
for invitation in invitations:
    print(invitation)
```

- Injected each name in `titlecased` into our wedding invitation format.

```
#Declares a list to store names in
names = []

#Foor Loop that collects 5 names
for _ in range(5):
    name = input("Please enter the name of someone you know. ")
    names.append(name)


#Takes all the names from input, makes them lower case
lowercased = [name.lower() for name in names]

#Takes the lower case names, makes them title (upper case first letter)
titlecased = [name.title() for name in lowercased]

#Add each invitation to a list
invitations = [
    f"Dear {name}, please come to the wedding this Saturday!" for name in titlecased]

#Prints the list of invitations
for invitation in invitations:
    print(invitation)
```

- Used a `for` loop to print every invitation in the `invitations` list.
    - we use a `for` loop instead of a list comprehension because we are *not* using `titlecased` to create a new list—rather, we are simply performing an action for each item in `titlecased`.

```
#Declares a list to store names in
names = []

#Foor Loop that collects 5 names
for _ in range(5):
    name = input("Please enter the name of someone you know. ")
    names.append(name)


#Takes all the names from input, makes them lower case
lowercased = [name.lower() for name in names]

#Takes the lower case names, makes them title (upper case first letter)
titlecased = [name.title() for name in lowercased]

#Add each invitation to a list
invitations = [
    f"Dear {name}, please come to the wedding this Saturday!" for name in titlecased]

#Prints the list of invitations
for invitation in invitations:
    print(invitation)
```

Functions (0:15)

Friday, September 27, 2019 5:02 PM

DRY (Don't Repeat Yourself)

There are disadvantages of writing code that does the same things in three different places

- If we write the same code in different places, and expect it to behave the same everywhere, we then have to update in several places when we make a change
- In large code bases, copying code in multiple places would often require us to was time make the same change in several and force us to keep track of duplicate code
- This is where Functions can come into play

- Functions are a way for us to give a name to a set of instructions we want to be able to repeat.

- Basic Anatomy of a function

```
# Basic Definition
def name(parameters):
    # code goes here
    return
```

  - Keyword def is used to define a function
  - "name" is what the functions is called
    - Parentheses indicate that "name" is a function
  - Colon at the end of the line
  - We can pass data to functions through parameters and arguemnts

- **`message`** is the data that we give the function, with which it *does* something.
- Here we just defined the function, but did not run it

```python
# Simple function with one parameter
def show(message):
    print(message)
```

- We would run the function using the code below
- Calling a function is a synonym for running it. When we say call function, it means we are running or executing it

```python
# Think of the parameter `message` as a variable
# You assign the string "Hello, World!" when you call the function
# This is like saying `message = "Hello, World!"`
show("Hello, World!")
```

- Consider a recipe for quesadillas. Parts of the recipe are *always* the same but we can choose to make the recipe with, for instance, either chicken *or* beef (our "arguments").
- Arguments are positional and position matters!
    - `make_quesadilla("sour cream", "beef")` will return `"Here is a sour cream quesadilla with beef"`.

```python
# Functions can have more than one parameter
def make_quesadilla(protein, topping):
    quesadilla = f"Here is a {protein} quesadilla with {topping}"
    print(quesadilla)
```

- We can make parameters optional if we specify a default parameter.
- `topping="sour cream"` makes "sour cream" our default `topping`. That is, if no topping is specified as an argument when the function is called, the function will supply "sour cream" as the `topping`.

```python
# We can also specify default values for parameters
def make_quesadilla(protein, topping="sour cream"):
    quesadilla = f"Here is a {protein} quesadilla with {topping}"
    print(quesadilla)
```

- make_quesadilla("chicken")
    - Would return "Here is a chicken quesadilla with sour cream"

- Make_quesadilla("beef","guacamole")
    - Would return "Here is a beef quesadilla with guacamole"

- We can also return data with return statement
- We often calculate values inside of functions

```
# Functions can return a value
def square(number):
    return number * number
```

- We can save a return value to a variable
- We can also print the return value

```
# You can save the value that is returned
squared = square(2)
print(squared)

# You can also just print the return value of a function
print(square(2))
print(square(3))
```

Exercise: Functions (0:10)

Friday, September 27, 2019 5:06 PM

Starter File(s): main.py

Instructions

- Write a function called "average" that accepts a list of numbers as a single argument
    - The function "average" should return the average for a list of numbers
- Test your function by calling it with different values and printing the results

Review: Functions (0:05)

Friday, September 27, 2019 5:08 PM

Key Points

- Define a function called `average` that accepts a single parameter called numbers.

```
# Write a function that returns the arithmetic average for a list of numbers
def average(numbers):
    #gives the total count of numbers
    length = len(numbers)

    #Initialize my total variable
    total = 0.0

    #for sums all the numbers within the list
    for number in numbers:
        total += number

    # Returns the sum divided by the length
    return total / length


# Test your function with the following:
print(average([1, 5, 9]))
```

- We can define variables inside of the function body that are typically only used inside of the function. `length = len(numbers)`

```
# Write a function that returns the arithmetic average for a list of numbers
def average(numbers):
    #gives the total count of numbers
    length = len(numbers)

    #Initialize my total variable
    total = 0.0

    #for sums all the numbers within the list
    for number in numbers:
        total += number

    # Returns the sum divided by the length
    return total / length


# Test your function with the following:
print(average([1, 5, 9]))
```

- Could have created another variable called `average` and returned that variable, but we can actually just return the results from `sum / length`.

```python
# Write a function that returns the arithmetic average for a list of numbers
def average(numbers):
    #gives the total count of numbers
    length = len(numbers)

    #Initialize my total variable
    total = 0.0

    #for sums all the numbers within the list
    for number in numbers:
        total += number

    # Returns the sum divided by the length
    return total / length


# Test your function with the following:
print(average([1, 5, 9]))
```

- Tested our code by calling the function with test data and printing the results.

```python
# Write a function that returns the arithmetic average for a list of numbers
def average(numbers):
    #gives the total count of numbers
    length = len(numbers)

    #Initialize my total variable
    total = 0.0

    #for sums all the numbers within the list
    for number in numbers:
        total += number

    # Returns the sum divided by the length
    return total / length

# Test your function with the following:
print(average([1, 5, 9]))
```

Everyone Do: Wresting With Functions (0:10)

Friday, September 27, 2019 5:08 PM

Files: WWE-Data-2016.csv and wrestling_functions.py

Exercise Overview: Reads in wrestling data and creates a function called print_percentages which takes in a parameter called wrestler_data. Calculates the percentage of matches the wrestler won, lost, and drew over the course of year. Prints out the stats for the wrestler in the terminal

Key Points

- Looking through the CSV data beforehand was key to figuring out how to calculate the total number of matches wrestled. Doing so would have told you what each index within a row referred to.
- Even though `row` is the variable being passed into the function, `wrestler_data` is still used within the function itself. The data within `row` is essentially moved into `wrestler_data` for usage within the function

Break (0:40)

Friday, September 27, 2019 5:10 PM

GitHub Presentation

Saturday, September 28, 2019 12:07 AM

_data-03-3-python...

Adding Files from Command Line

Saturday, September 28, 2019 12:07 AM

Log into your GitHub Account
Create a new repo
From repo, click the green box in the "Clone or Download"
Can you select "Use SSH"?

Cd ~ before using links

Generating SSH Keys

Adding a new SSH to GitHub Account

- Open terminal (or git-bash for Windows users) and navigate to the home folder using `cd ~`.
- Type in `git clone <repository link>` in the terminal to clone the repo to the current directory. Once this has run, everyone should now see a folder with the same name as the repo.

- Open the folder in VS Code and create two python script files named `script01.py` and `script02.py`.
- Once the files have been created, open up Terminal/git-bash and navigate to the repo folder. Run the following lines and explain each as you go through them.

```
# Displays that status of files in the folder
git status

# Adds all the files into a staging area
git add .

# Check that thr files were added correctly
git status

# Commits all the files to your repo and adds a message
git commit -m <add commit message here>

# Pushes the changes up to GitHub
git push origin master
```

- Finally navigate to the repo on Github.com to see that the changes have been pushed up.

Adding more to Repo

Saturday, September 28, 2019 9:31 AM

Instructions

Using the repo that just created, make or add the following changes:

- Add new lines of code to one of the python files.
- Create a new folder.
- Add a file to the newly created folder.
- Add, commit and push the changes.
- Delete the new folder.
- Add, commit and push the changes again.