Warmup Problems

CS 1301 - Intro to Computing

Important

- Resources:
  - TA Helpdesk
  - Email TA's or use Ed Discussion
  - Textbook: How to Think Like a Computer Scientist
  - CS 1301 YouTube Channel
  - Handouts (in Canvas Files)
- Comment out or delete all function calls. Only import statements, global variables, and comments are okay to be outside of your functions.

Purpose

Warmup questions are the perfect questions to attempt when you begin studying for an exam or have just learned the relevant material. You may reinforce the concepts taught in CS 1301 by attempting these questions, which will be a step lower in complexity and test fewer concepts than our usual practice problems. We will not be providing a skeleton file to assist you with completing these warmup problems. Please name your file `warmupProblems.py` before submitting to Gradescope!

Grading

You are given the opportunity to earn 1% of extra credit applied to your final grade by successfully completing all of the warmup problems.

```
extraCreditPercentage = autogradedPoints / 150
```

Functions and Expressions

Tech Libs

Function Name: techLibs()
Parameters: N/A
Returns: None
Description: Welcome to Tech Mad Libs! Write a function called `techLibs()` that completes a funny Mad Lib. The function should ask the user for their **name**, a **friend's name**, and their **favorite study spot**. Print out each line of the completed Mad Lib, which looks like:

```
The other day, I saw {name} and {friendName} up to no good.
They were loudly gossiping at {studySpot} instead of studying.
```

```
>>> techLibs()
Enter your name: Lasya
Enter your best friend's name: Ramya
Enter your favorite study spot: Klaus
The other day, I saw Lasya and Ramya up to no good.
They were loudly gossiping at Klaus instead of studying.
```

```
>>> techLibs()
Enter your name: Harshith
Enter your best friend's name: Paige
Enter your favorite study spot: the Student Center
The other day, I saw Harshith and Paige up to no good.
```

Dining Dollars Calculator

Function Name: diningDollars()
Parameters: N/A
Returns: None
Description: Your friend has no idea how many dining dollars they'll have left at the end of the semester. Write a function called `diningDollars()` that asks the user how many **dining dollars** they currently have, how much they **spend weekly**, and how many **weeks are left** in the semester. Print out a string of the form `"You will have ${leftOver} left over."`. Round the final amount left over to two decimal places.

```
>>> diningDollars()
How many dining dollars do you currently have? 500.01
How many dining dollars do you spend per week? 20.00
How many weeks are left? 13
You will have $240.01 left over.
```

```
>>> diningDollars()
How many dining dollars do you currently have? 5.05
How many dining dollars do you spend per week? 1.00
How many weeks are left? 5
You will have $0.05 left over.
```

Sphere Volume Calculator

Function Name: sphereVol()
Parameters: N/A
Returns: None
Description: Write a function called `sphereVol()` that calculates the volume of a sphere. The function should ask the user for the **radius of their sphere** and print out a string of the form `"The volume of your sphere is {volume}."`. Round the volume to two decimal places.

Note: The volume of a sphere is `V = (3.14) * (r ** 3) * (4/3)`. Use 3.14 for pi.

```
>>> sphereVol()
Enter the radius of your sphere: 12.0
The volume of your sphere is 7234.56.
```

Conditionals

Italian Dinner Night

Function Name: italianNight()
Parameters: cookTime (`float`)
Returns: selectedDish (`str`)
Description: You decide to do a Italian dinner night with your friends! Using the table below, write a function called `italianNight()` that takes in the total time available for cooking, in minutes. Return the name of the dish that takes the longest to make without exceeding the given time limit. If you don't have enough time to make any dish, then return the string `"We'll cook some other time."`.

Dish	Cook Time (min)
Breadsticks	20
Pasta	35
Lasagna	50

```
>>> italianNight(85.0)
'Lasagna'
```

```
>>> italianNight(35.0)
'Pasta'
```

To Cook Or Not To Cook

Function Name: toCook()
Parameters: numClasses (`int`), diningDollars (`float`)
Returns: None (`NoneType`)
Description: You are trying to decide whether you should eat out or not given your busy schedule. Write a function called `toCook()` that takes in the number of classes you have each day and how many dining dollars you have left. If you have more than 3 classes a day **and** more than $10 in dining dollars, then print out `"Let's get Panda Express!"`. If you have 3 or fewer classes a day **and** have at least $50 in dining dollars, then print out `"Let's splurge on Chick-fil-A!"`. Otherwise, print out `"Guess I'll have to cook myself."`.

```
>>> toCook(4, 15.0)
Let's get Panda Express!
```

```
>>> toCook(2, 5.0)
Guess I'll have to cook myself.
```

Cooking Class

Function Name: cookingClass()
Parameters: date (`int`), isWeekend (`bool`)
Returns: availability (`str`)
Description: You are planning to go to a cooking class with your friend and want to figure out their availability. Given the table below, write a function called `cookingClass()` that takes in two parameters: the date (`int`) and whether that date is on the weekend (`bool`). Return the availability of your friend.

Date	Weekend	Availability
Odd	True	Unsure
Even	True	Yes
Odd	False	Yes
Even	False	No

```
>>> cookingClass(14, True)
"Yes"
```

```
>>> cookingClass(5, True)
"Unsure"
```

Iteration

Capture the Flag

Function Name: captureTheFlag()
Parameters: directions (`str`)
Returns: None (`NoneType`)
Description: You are minding your own business in the woods of Camp Half-Blood while an intense game of Capture the Flag occurs. Suddenly, you are attacked by a child of Ares and have to run away. You've run so far that you are lost, so you decide to retrace your steps.

Write a function called `captureTheFlag()` that takes in one parameter: a set of directions (`str`) of arbitrary length. Each character represents the directions you took: `'L'` for left and `'R'` for right. To retrace your steps you'll need to iterate **backwards** through the directions. **Print** out each direction in the format `'Turn {right/left}!'`.

```
>>> captureTheFlag('LLRL')
Turn left!
Turn right!
Turn left!
Turn left!
```

```
>>> captureTheFlag('LRR')
Turn right!
Turn right!
Turn left!
```

Accidentally Vaporized My Pre-Algebra Teacher...

Function Name: demystifyMessage()
Parameters: jumbledMessage (`str`)
Returns: unscrambledMessage (`str`)
Description: You are on a school trip to New York's Metropolitan Museum of Art and are starting to feel a little weird... You are drawn to the statues of the Greek Gods, but the text on the plaques underneath the statues are scrambled! Write a function called `demystifyMessage()` that takes in one parameter: a jumbled message (`str`). You should return an unscrambled message that contains only the **letters**.

```
>>> demystifyMessage("P1E3R5C7Y9")
'PERCY'
```

```
>>> demystifyMessage("**D%E^M7*;I&GOD")
'DEMIGOD'
```

Garden Gnome Emporium

Function Name: gnomeAccounting()
Parameters: orderNumbers (`str`)
Returns: None (`NoneType`)
Description: Tired of escaping Alecto, you take a break with Auntie Em amid vaguely familiar garden decorations resembling your Uncle Ferdinand... She needs accounting help with the sudden popularity of terrified garden gnomes in high New York society.

Write a function called `gnomeAccounting()` that takes in a long, continuous string of order numbers (`str`), where each order number is exactly 6 digits long. Separate the order numbers from each other, and print them out one by one.

```
>>> gnomeAccounting('291267145621689235')
291267
145621
689235
```

```
>>> gnomeAccounting('100010111002')
100010
111002
```

Strings and Lists

Case Count of Love Letter

Function Name: countCase()
Parameters: loveLetter (`str`)
Returns: difference (`str`)
Description: You're crafting a love letter, and you want every sentence to be as harmonious as our hearts – with an equal number of uppercase and lowercase letters! Write a function called `countCase()` that takes in one parameter: a sentence from a love letter (`str`).

- If the uppercase letters outnumber the lowercase letters, then return `"{difference} more uppercase letter(s)."`
- If the lowercase letters outnumber the uppercase letters, then return `"{difference} more lowercase letter(s)."`
- If both cases are equal, then return the string `"Perfectly balanced!"`.

Note: Only consider letters, and not digits, blank spaces, nor special characters.

```
>>> countCase("You are my MOON!")
'2 more lowercase letter(s).'
```

```
>>> countCase("U + m3 = <3")
'Perfectly balanced!'
```

Match Finder

Function Name: findLove()
Parameters: candidates (`list`), favCity (`str`)
Returns: matches (`list`)
Description: Love is in the air, and you're on a quest to find the perfect match! Write a function called `findLove()` that takes in a list of potential candidates (`list`) and your favorite city (`str`). Each candidate is represented as a list containing their name (`str`) and the city they reside in (`str`). Return a sorted list of the names of potential matches who live in your favorite city.

```
>>> candidates = [["Joey", "NYC"], ["Chandler", "NYC"]]
>>> findLove(candidates, "NYC")
['Chandler', 'Joey']
```

```
>>> candidates = [["Eric", "Denver"], ["Kyle", "Denver"],
                  ["Stan", "Atlanta"]]
>>> findLove(candidates, "Atlanta")
['Stan']
```

Mutual Interests

Function Name: mutualInterests()
Parameters: yourInterests (`list`), theirInterests (`list`)
Returns: interests (`list`)
Description: You're looking to see if you are compatible with a new match! Write a function called `mutualInterests()` that takes in two parameters: your list of interests (`list`) and your match's list of interests (`list`). Return a sorted list containing the interests you both share. If you have no mutual interests, return an empty list.

```
>>> yourInterests = ["dance", "movies", "music"]
>>> theirInterests = ["movies", "music"]
>>> mutualInterests(yourInterests, theirInterests)
['movies', 'music']
```

```
>>> yourInterests = ["basketball", "music"]
>>> theirInterests = ["music", "basketball", "swimming"]
>>> mutualInterests(yourInterests, theirInterests)
['basketball', 'music']
```

Tuples and Modules

104 Days of Summer Vacation But Even More Inventions!

Function Name: numInventions()
Parameters: inventions (`list`)
Returns: completed (`int`)
Description: Phineas and Ferb are working on a project that involves organizing information about their inventions. Write a function called `numInventions()` that takes in a list of inventions. Each invention is represented as a tuple with the invention name (`str`) and completion status (`bool`). The function should return the number of completed inventions.

```
>>> inventions = [("Time Machine", True), ("Anti-Gravity Ray", False),
                  ("Giant Rollercoaster", True), ("Shrinkinator", False)]
>>> numInventions(inventions)
2
```

```
>>> inventions = [("Super Robot", True), ("Invisible Cloak", True),
                  ("Teleportation", True), ("Mind-Reading Helmet", True),
                  ("Laser Beam Glasses", True)]
>>> numInventions(inventions)
5
```

Whatcha Doin?

Function Name: helpPhineas()
Parameters: badge (`str`), badges (`list`)
Returns: canHelp (`bool`)
Description: Isabella's crush on Phineas is no secret, but the other Fireside Girls worry she's devoting too much time to helping him. Write a function called `helpPhineas()` that takes in two parameters: the name of a badge (`str`) and a list of all badges. Each badge in the list is represented as a tuple containing the badge name (`str`) and the number of times Isabella has earned it (`int`). If Isabella has more than 8 or fewer than 1 of the specified badge, then she can't help Phineas. Return `True` if Isabella can help Phineas, and `False` otherwise. Also, return `False` if the specified badge is not in the list.

```
>>> badges = [("Enthusiasm Patch", 6), ("Honesty Patch", 5),
              ("Beekeeping Patch", 10)]
>>> helpPhineas("Honesty Patch", badges)
True
```

```
>>> badges = [("Aeronautic Patch", 6), ("Marshmallow Patch", 5),
              ("I Just Saw A Cute Boy Patch", 64)]
>>> helpPhineas("I Just Saw A Cute Boy Patch", badges)
False
```

Emergency! Pick an Agent!

Function Name: agentDispatch()
Parameters: coordinates (`tuple`), agents (`list`)
Returns: secretAgent (`str`)
Description: Your friend Carl, the usual operations manager at OWCA, is ill and needs your assistance in selecting agents to dispatch. Write a function called `agentDispatch()` that takes in two parameters: the coordinates of a villain (`tuple`) and a list OWCA's available agents. Each agent in the list is represented as a tuple containing their name and current coordinates (`tuple`). The function should return the name of the closest agent to the villain.

Hint: Take a look at the math.dist(p, q) function to quickly calculate the distance between points *p* and *q*.
Note: You may assume no agents will be equally close.

```
>>> agents = [('Agent P', (2, 5)),
              ('Agent C', (4, 2)),
              ('Agent H', (-2, 12)),
              ('Agent G', (205, 300)),
              ('Agent M', (1, 1))]
>>> agentDispatch((6, 5), agents)
'Agent C'
```

```
>>> agents = [('Agent P', (2, 5)),
              ('Agent C', (4, 2)),
              ('Agent H', (-2, 12)),
              ('Agent G', (205, 300)),
              ('Agent M', (1, 1))]
>>> agentDispatch((1, 1), agents)
'Agent M'
```

Dictionaries

Grand Prix Voting

Function Name: bestCity()
Parameters: voteDict (`dict`)
Returns: bestCity (`str`)
Description: As the newly promoted CEO of Formula 1, you're tasked with deciding the location for the next Grand Prix. Write a function called `bestCity()` that takes in a dictionary of votes (`dict`), containing cities that are mapped to a list of board members who voted for each city. Return the city with the highest number of votes.

Note: Assume there will never be a tie.

```
>>> voteDict = {'Atlanta': ['Collin', 'Harshith', 'Chris'],
                'LA': ['Matias', 'Paige'],
                'Seoul': ['Rod']}
>>> bestCity(voteDict)
'Atlanta'
```

```
>>> voteDict = {'Atlanta': ['Collin'], 'LA': ['Matias', 'Paige'],
                'Seoul': ['Rod']}
>>> bestCity(voteDict)
'LA'
```

Constructor's Championship Winner

Function Name: winningTeam()
Parameters: raceWinners (`dict`)
Returns: winners (`list`)
Description: The FIA recently announced that last season's winner was mistakenly crowned due to errors in tallying the points. The FIA acknowledged the mistake, and now they need your help to ensure it doesn't happen again.

Write a function called `winningTeam()` that takes in a dictionary of teams (`dict`), containing team names mapped to a tuple containing their driver who scored the most points in the format `(driver, points)`. The function should return a sorted list of teams who scored at least 25 points.

```
>>> raceWinners = {'Red Bull': ('Max Verstappen', 575),
                   'Mercedes': ('Lewis Hamilton', 234),
                   'Aston Martin': ('Fernando Alonso', 206)}
>>> winningTeam(raceWinners)
['Aston Martin', 'Mercedes', 'Red Bull']
```

```
>>> raceWinners = {'Haas': ('Kevin Magnussen', 3),
                   'AlphaTauri': ('Yuki Tsunoda', 17),
                   'Sauber': ('Valterri Bottas', 10)}
>>> winningTeam(raceWinners)
[]
```

Fantasy F1 Team

Function Name: fantasyF1()
Parameters: driverCatalog (`dict`), myPicks (`list`)
Returns: cost (`int`)
Description: It's never too early to plan your fantasy F1 driver lineup! Write a function called `fantasyF1()` that takes in a catalog of drivers (`dict`) and your fantasy driver picks (`list`). The driver catalog dictionary maps the racing team names to another dictionary containing driver names (`str`) mapped to their respective cost (`int`). Return the total cost of your desired fantasy F1 team.

Note You may assume that drivers you plan to draft will always be found in the driver catalog.

```
>>> driverCatalog = {'Ferarri': {'Charles LeClerc': 500, 'Carlos Sainz': 400},
                     'McLaren': {'Lando Norris': 500, 'Oscar Piastri': 300}}
>>> myPicks = ['Charles LeClerc', 'Lando Norris', 'Oscar Piastri']
>>> fantasyF1(driverCatalog, myPicks)
1300
```

```
>>> driverCatalog = {'Mercedes': {'Lewis Hamilton': 700, 'George Russell': 400},
                     'Red Bull': {'Max Verstappen': 1000, 'Sergio Perez': 600},
                     'Aston Martin': {'Fernando Alonzo': 500, 'Lance Stroll': 200}}
>>> myPicks = ['Lewis Hamilton', 'Max Verstappen', 'Sergio Perez']
>>> fantasyF1(driverCatalog, myPicks)
2300
```

File I/O

For this section, you will be reading and writing to text files that pertain to the carbon emissions of different forms of transportation.

swiftieFlights.txt

The `swiftieFlights.txt` file being read from will be formatted as shown below. Be sure to download the provided file from Canvas, and **move it into the same folder as your `warmupProblems.py` file**. Recall that if you are using a Mac, do not place warmupProblems.py and `swiftieFlights.txt` in your Downloads or Desktop folder. To open the `.txt` file, you can use the built-in Notepad for Windows, TextEdit for Mac, or any other text editor of your choice.

You will be working with a text file that contains data about Taylor Swift's flights. The data will contain the cities she is flying between, the flight time, the number of gallons of jet fuel used, the cost of fuel used, and the number of tons of carbon dioxide equivalent emissions. Each flight's data will be separated by a newline. The code below shows how the text file will be formatted. See the provided `swiftieFlights.txt` file as an example.

Note: `swiftieFlights.txt` will never contain duplicate flights.

```
Taylor Swift Flight Log

airport1 to airport2
flight time
gallons of jet fuel
cost of fuel
tons of carbon emmissions

airport2 to airport3
flight time
gallons of jet fuel
cost of fuel
tons of carbon emmissions

...
```

Taylor's Carbon Emissions

Function Name: taylorEmissions()
Parameters: route (`str`)
Returns: carbonEmissions (`float`)
Description: In order to comply with aviation regulations, Taylor must be able to track the carbon emissions for all of her flights. Write a function called `taylorEmissions()` that takes in a route and returns the number of tons of carbon emitted from that route, as a float. You may assume that the passed in route will always be present in `swiftieFlights.txt`.

```
>>> taylorEmissions('CPS to SUS')
0.8342
```

```
>>> taylorEmissions('MMU to MKC')
11.0
```

Taylor Stop Flying!

Function Name: taylorFlights()
Parameters: carbonTons (`int`)
Returns: flights (`list`)
Description: Taylor Swift has a fixed concert schedule and cannot make changes to it. However, she is committed to reducing her carbon emissions and needs your help.

Write a function called `taylorFlights()` that takes in one parameter: the maximum amount of carbon emissions, in tons per flight, that Taylor is willing to tolerate (`int`). Return a list of flights **sorted in alphabetical order** that exceed Taylor's specified limit.

```
>>> taylorFlights(10)
['MMU to MKC']
```

```
>>> taylorFlights(1)
['BNA to MKC', 'BUR to LAS', 'BWI to BNA', 'MKC to MMU',
 'MMU to BNA', 'MMU to MKC', 'SUS to CPS']
```

Sweden's Transportation

Function Name: transportationModes()
Parameters: cityTransport (`list`)
Returns: None (`NoneType`)
Description: You're planning a summer trip to Europe and aiming to explore Sweden, which is known for its low carbon emissions. Fortunately, Europe boasts excellent public transportation, allowing you to travel conveniently without relying solely on flights.

Write a function called `transportationModes()` that takes in a list of tuples containing a mode of Sweden's transportation (`str`) and the amount of carbon that is emitted in tons (`int`). Write to a file named `transportationModes.txt` that displays each mode of transportation and its carbon emissions in the format:

```
Transportation Modes

transport1: carbonEmissions1
transport2: carbonEmissions2
...
```

Note: Be sure no newline characters (`\n`) are included at the end of `transportationModes.txt` .

```
>>> cityTransport = [("Bus", 6), ("Train", 7), ("Plane", 2)]
>>> transportationModes(cityTransport)
```

Contents of `transportationModes.txt` after the function above is executed:

```
Transportation Modes

Bus: 6
Train: 7
Plane: 2
```

APIs

In this section, we will be using the Studio Ghibli API.

If you make a valid request with the URL: https://ghibliapi.vercel.app/locations, you will receive the following JSON formatted response:

```
[
  {
    "id": "11014596-71b0-4b3e-b8c0-1c4b15f28b9a",
    "name": "Irontown",
    "climate": "Continental",
    "terrain": "Mountain",
    "surface_water": "40",
    "residents": [
      "https://ghibliapi.vercel.app/people/ba924631-068e-4436-b6de-f3283fa848f0",
      "https://ghibliapi.vercel.app/people/030555b3-4c92-4fce-93fb-e70c3ae3df8b"
    ],
    "films": [
      "https://ghibliapi.vercel.app/films/0440483e-ca0e-4120-8c50-4c8cd9b965d6"
    ],
    "url": "https://ghibliapi.vercel.app/locations/11014596-71b0-4b3e-b8c0-
           1c4b15f28b9a"
  },
  ...
]
```

If you make a request with an invalid URL, you will receive the following response:

```
{}
```

Studio Ghibli API endpoints correspond to films, people, locations, etc. and are searchable by ID:

```
https://ghibliapi.vercel.app/locations/11014596-71b0-4b3e-b8c0-1c4b15f28b9a
```

This retrieves location information of `Irontown` with ID of `11014596-71b0-4b3e-b8c0-1c4b15f28b9a`.

You will have to use this form of querying to retrieve specific information for questions.

Movie Picker

Function Name: letterLover()
Parameters: name (`str`)
Returns: movieList (`list`)
Description: As a picky viewer, your friend is struggling to choose a Studio Ghibli movie, insisting it must start with a specific letter. Write a function called `letterLover()` that takes in one letter. Return a **sorted** list of every Studio Ghibli movie that starts with that letter (case insensitive).

Hint: Use the https://ghibliapi.vercel.app/films endpoint to retrieve data on all Studio Ghibli films.

```
>>> letterLover('p')
['Pom Poko', 'Ponyo', 'Porco Rosso', 'Princess Mononoke']
```

```
>>> letterLover('r')
[]
```

Species Finder

Function Name: speciesFinder()
Parameters: eyeColor (`str`)
Returns: exists (`bool`)
Description: You are looking for species with a specific eye color. Write a function called `speciesFinder()` that takes in an eye color. Return `True` if it is possible for a species in the Ghibli world to have the provided eye color, and `False` otherwise.

```
>>> speciesFinder("Black")
True
```

```
>>> speciesFinder("Purple")
False
```

Film Characters

Function Name: characters()
Parameters: filmName (`str`)
Returns: charList (`list`)
Description: Your friends are curious about Studio Ghibli movies and want to learn about the characters in your favorite film before watching it with you. Write a function called `characters()` that takes in a film name (`str`). Return a **sorted** list of character names in the movie.

Hint: When there are no characters for a given movie, the only url will be `"https://ghibli-api.vercel.app/people/"` .

```
>>> characters("Kiki's Delivery Service")
['Jiji', 'Kiki', 'Madame', 'Osono', 'Tombo', 'Ursula']
```

```
>>> characters("Grave of the Fireflies")
[]
```

Recursion

How Many Fouls?

Function Name: foulCount()
Parameters: teamList(`list`)
Returns: totalFouls (`int`)
Description: You are a March Madness referee and need to tally up the fouls committed by each team.

Write a **recursive** function called `foulCount()` that takes in a list of team fouls. This list contains tuples of the team name (`str`) and number of fouls they have commited (`int`). Return the total number of fouls committed throughout the tournament.

```
>>> foulCount([('Kentucky', 15), ('Michigan St.', 18), ('Duke', 12)])
45
```

```
>>> foulCount([('Kansas', 11), ('UNC', 28), ('Arizona', 4),('UCLA', 19)])
62
```

NCAA Intern Mistake

Function Name: convertTeams()
Parameters: teamsTup (`tuple`)
Returns: teamsList (`list`)
Description: The NCAA interns made an error while gathering the March Madness team names, including statistics alongside the names. Write a **recursive** function called `convertTeams()` that takes in a tuple of team names and random data. Return a list of only the team names with the numeric data filtered out to ensure future data integrity.

```
>>> convertTeams(("Terps", "Jackets", 99, "Tar Heels", 10000))
['Terps', 'Jackets', 'Tar Heels']
```

```
>>> convertTeams((9024, "Blue Jays", "Retrievers", 13984, "Seagulls"))
['Blue Jays', 'Retrievers', 'Seagulls']
```

Team Merger

Function Name: mergeTeamNames()
Parameters: team1 (`str`), team2 (`str`)
Returns: finalTeamName (`str`)
Description: In an attempt to establish a new basketball league with combined teams, your task is to merge the names of two basketball teams into a single entity. Write a **recursive** function called `mergeTeamNames()` that takes in the names of two teams. Return the new team name that merges the two team names by alternating letters from each team's name, starting with the first letter of the first team name.

Note: If you exhaust the letters available for alternating between the team names, ensure to include the remaining letters from the non-empty team name in the final merged team name.

```
>>> mergeTeamNames("JACKETS", "BULLDOGS")
'JBAUCLKLEDTOSGS'
```

```
>>> mergeTeamNames("DEVILS", "TERPS")
'DTEEVRIPLSS'
```

Object Oriented Programming

In this section, you will be creating a simple spiderverse themed simulation! Due to the structure of the autograder test cases, we recommend implementing the `__init__()` methods for all classes before attempting the other methods. This would also help you understand the purpose of all class attributes, which is essential to know when implementing the other methods.

Spiderman

Attributes:

- name (`str`): the Spiderman's name.
- health (`float`): how much health this Spiderman has.
- damage (`float`): the amount of damage a Spiderman can exert on an opponent.

Methods:

- **__init__()**: Initializes a Spiderman's name, health, and damage.
- **webStrike()**
  - This method should take in one additional argument: a `Villain` object, `villain` , that the Spiderman is attacking.
  - The `Villain` object should have its `health` attribute decreased by the value of Spiderman's `damage` attribute.
  - Additionally, if the `Villain` object's `health` attribute is less than or equal to zero, then that means the villain is defeated.
    - If the villain is defeated, then the `Villain` object's `isDefeated` attribute should be updated to `True` . Print the message `"{Villain's name} has been defeated!"` .
- **__gt__()**
  - A `Spiderman` object is considered greater than another `Spiderman` object if it has more health.

Villain

Attributes:

- name (`str`): the Villain's name.
- health (`float`): how much health this Villain has.
- damage (`float`): the amount of damage a Villain can exert on an opponent.
- isDefeated (`bool`): whether the Villain has been defeated.

Methods:

- **__init__()**: Initializes the following attributes:
  - name
  - health
  - damage
  - isDefeated - should always be initalized to `False`
- **attack()**
  - This method should take in one additional argument: a `Spiderman` object, `spiderman`, that the Villain is attacking.
  - In this method, a Villain can only attack if its `isDefeated` attribute is `False`.
    - In the case that `isDefeated` is `True`, this method should immediately print the message `"{Villain's name} is already defeated!"` and do nothing.
  - Only if the Villain is able to attack, the `Spiderman` object should have its `health` attribute decreased by the value of Villain's `damage` attribute.
- **__str__()**
  - This method creates a string representation of the `Villain` object.
  - This method should return the message `"{Villain's name} has {Villain's health} health and {Villain's damage} damage."`.

Universe

Attributes:

- name (`str`): this Universe's name.
- spiderList (`list`): the Spiderman objects present in this universe.
- villainList (`list`): the Villain objects present in this universe.

Methods:

- **__init__()**: Initializes the following attributes:
  - name
  - spiderList - always initialized to `[]`
  - villainList - always initialized to `[]`
- **transportSpiderman()**: This method should take in one additional argument: a `Spiderman` object whose `name` should be added to the `spiderList` list. Ensure that the list is always sorted.
- **transportVillain()**: This method should take in one additional argument: a `Villain` object whose `name` should be added to the `villainList` list. Ensure that the list is always sorted.
- **battle()**
  - This method should take in two additional arguments: a `Spiderman` and `Villain` object that will battle each other.
  - If the Spiderman's name is not in `spiderList`, print `"{Spiderman's name} not found."`.
  - In addition, if the Villain's name is not in `villainList`, print `"{Villain's name} not found."`.
  - The battle will only start if both the Spiderman and the Villain are found in the Universe. Do not continue unless the battle is able to start.
  - While the Spiderman's health **and** the Villain's heatlh are both greater than zero:
    - First, call the Spiderman's `webStrike` method with the Villain as the parameter.
    - Second, call the Villain's `attack` method with the Spiderman as the parameter.
  - After the battle has finished, check who has been defeated.
    - If the Spiderman has been defeated (with health less than or equal to zero), print `"{Spiderman's name} lost the battle!"`
    - If the Villain has been defeated instead, print `"{Villain's name} lost the battle!"`

Object Oriented Programming Test Cases

Test 000: Create Peter Spiderman

Points: 1
Prerequisites: None
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
```

Tests:

- Check Attributes of `peter`:

```
peter (class Spiderman)
    name: 'Peter'
    health: 100
    damage: 50
```

Test 001: Create Miles Spiderman

Points: 0.5
Prerequisites: None
Objects Created and Methods Called:

```
miles = Spiderman("Miles", 80, 10)
```

Tests:

- Check Attributes of `miles`:

```
miles (class Spiderman)
    name: 'Miles'
    health: 80
    damage: 10
```

Test 002: Create Green Goblin Villain

Points: 1
Prerequisites: None
Objects Created and Methods Called:

```
greenGob = Villain("Green Goblin", 100, 20)
```

Tests:

- Check Attributes of `greenGob`:

```
greenGob (class Villain)
    name: 'Green Goblin'
    health: 100
    damage: 20
    isDefeated: False
```

Test 003: Create Doc Ock Villain

Points: 0.5
Prerequisites: None
Objects Created and Methods Called:

```
docOck = Villain("Doc Ock", 200, 80)
```

Tests:

- Check Attributes of `docOck`:

```
docOck (class Villain)
    name: 'Doc Ock'
    health: 200
    damage: 80
    isDefeated: False
```

Test 004: Create Universe

Points: 1
Prerequisites: None
Objects Created and Methods Called:

```
universe = Universe("Universe 1")
```

Tests:

- Check Attributes of `universe`:

```
universe (class Universe)
    name: 'Universe 1'
    spiderList: []
    villainList: []
```

Test 005: Spiderman Webstrike (Not Defeated)

Points: 1
Prerequisites: Test 000, Test 002
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
greenGob = Villain("Green Goblin", 100, 20)
peter.webStrike(greenGob)
```

Tests:

- Check Attributes of `peter`:

```
peter (class Spiderman)
    name: 'Peter'
    health: 100
    damage: 50
```

- Check Attributes of `greenGob`:

```
greenGob (class Villain)
    name: 'Green Goblin'
    health: 50
    damage: 20
    isDefeated: False
```

- Check Print Statements:

```

```

Test 006: Spiderman Webstrike (Defeated)

Points: 1
Prerequisites: Test 005
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
greenGob = Villain("Green Goblin", 100, 20)
peter.webStrike(greenGob)
peter.webStrike(greenGob)
```

Tests:

- Check Attributes of `peter`:

```
peter (class Spiderman)
    name: 'Peter'
    health: 100
    damage: 50
```

- Check Attributes of `greenGob`:

```
greenGob (class Villain)
    name: 'Green Goblin'
    health: 0
    damage: 20
    isDefeated: True
```

- Check Print Statements:

```
Green Goblin has been defeated!
```

Test 007: Spiderman Greater Than Method

Points: 1
Prerequisites: Test 000, Test 001
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
miles = Spiderman("Miles", 80, 10)
```

Tests:

- Check the Comparison `peter > miles`:

```
True
```

Test 008: Villain Attack (Not Defeated)

Points: 1
Prerequisites: Test 001, Test 003
Objects Created and Methods Called:

```
miles = Spiderman("Miles", 80, 10)
docOck = Villain("Doc Ock", 200, 80)
docOck.attack(miles)
```

Tests:

- Check Attributes of `miles`:

```
miles (class Spiderman)
    name: 'Miles'
    health: 0
    damage: 10
```

- Check Attributes of `docOck`:

```
docOck (class Villain)
    name: 'Doc Ock'
    health: 200
    damage: 80
    isDefeated: False
```

- Check Print Statements:

```

```

Test 009: Villain Attack (Defeated)

Points: 1
Prerequisites: Test 001, Test 003
Objects Created and Methods Called:

```
miles = Spiderman("Miles", 80, 10)
docOck = Villain("Doc Ock", 200, 80)
docOck.isDefeated = True
docOck.attack(miles)
```

Tests:

- Check Attributes of `miles`:

```
miles (class Spiderman)
    name: 'Miles'
    health: 80
    damage: 10
```

- Check Attributes of `docOck`:

```
docOck (class Villain)
    name: 'Doc Ock'
    health: 200
    damage: 80
    isDefeated: True
```

- Check Print Statements:

```
Doc Ock is already defeated!
```

Test 010: Villain Str Method

Points: 1
Prerequisites: Test 002
Objects Created and Methods Called:

```
greenGob = Villain("Green Goblin", 100, 20)
```

Tests:

- Check Result of `print(greenGob)`:

```
Green Goblin has 100 health and 20 damage.
```

Test 011: Universe Transport Spiderman

Points: 1
Prerequisites: Test 000, Test 001, Test 004
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
miles = Spiderman("Miles", 80, 10)
universe = Universe("Universe 1")
universe.transportSpiderman(peter)
universe.transportSpiderman(miles)
```

Tests:

- Check Attributes of `universe`:

```
universe (class Universe)
    name: 'Universe 1'
    spiderList: ['Miles', 'Peter']
    villainList: []
```

Test 012: Universe Transport Villain

Points: 1
Prerequisites: Test 002, Test 003, Test 004
Objects Created and Methods Called:

```
greenGob = Villain("Green Goblin", 100, 20)
docOck = Villain("Doc Ock", 200, 80)
universe = Universe("Universe 1")
universe.transportVillain(greenGob)
universe.transportVillain(docOck)
```

Tests:

- Check Attributes of `universe`:

```
universe (class Universe)
    name: 'Universe 1'
    spiderList: []
    villainList: ['Doc Ock', 'Green Goblin']
```

Test 013: Universe Battle (People Not Found)

Points: 1
Prerequisites: Test 000, Test 002, Test 004
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
greenGob = Villain("Green Goblin", 100, 20)
universe = Universe("Universe 1")
universe.battle(peter, greenGob)
```

Tests:

- Check Attributes of `universe`:

```
universe (class Universe)
    name: 'Universe 1'
    spiderList: []
    villainList: []
```

- Check Print Statements:

```
Peter not found.
Green Goblin not found.
```

Test 014: Universe Battle (Spiderman Wins)

Points: 1
Prerequisites: Test 006, Test 008, Test 009, Test 011, Test 012
Objects Created and Methods Called:

```
peter = Spiderman("Peter", 100, 50)
greenGob = Villain("Green Goblin", 100, 20)
universe = Universe("Universe 1")
universe.transportSpiderman(peter)
universe.transportVillain(greenGob)
universe.battle(peter, greenGob)
```

Tests:

- Check Attributes of `universe`:

```
universe (class Universe)
    name: 'Universe 1'
    spiderList: ['Peter']
    villainList: ['Green Goblin']
```

- Check Print Statements:

```
Green Goblin has been defeated!
Green Goblin is already defeated!
Green Goblin lost the battle!
```

Test 015: Universe Battle (Villain Wins)

Points: 1
Prerequisites: Test 006, Test 008, Test 011, Test 012
Objects Created and Methods Called:

```
miles = Spiderman("Miles", 80, 10)
docOck = Villain("Doc Ock", 200, 80)
universe = Universe("Universe 1")
universe.transportSpiderman(miles)
universe.transportVillain(docOck)
universe.battle(miles, docOck)
```

Tests:

- Check Attributes of `universe`:

```
universe (class Universe)
    name: 'Universe 1'
    spiderList: ['Miles']
    villainList: ['Doc Ock']
```

- Check Print Statements:

```
Miles lost the battle!
```