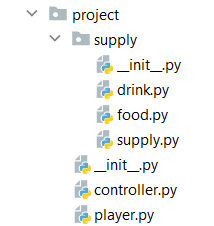
# Python OOP Exam - Medieval Games

*You and your friends gathered to play Medieval Games in your backyard!*

You will be provided with a **skeleton** that includes all the folders and files you will need.

***Note: You are not allowed to change the folder and file structure and change their names!***



# Judge Upload

For the **first 2 problems**, create a **zip** file with the **project** **folder** and **upload it** to the judge system.

For the **last problem**, create a **zip** file with the **tests** **folder** and **upload it** to the judge system.

You do not need to include **in the zip file** your **venv**, **.idea**, **pycache**, and **\_\_MACOSX** (for Mac users), so you do not exceed **the maximum allowed size** of **16.00 KB**.

# Structure (Problem 1) and Functionality (Problem 2)

Our first task is to implement the **structure and functionality** of all the classes (properties, methods, inheritance, etc.)

You are **free to add additional attributes** (instance attributes, class attributes, methods, dunder methods, etc.) to simplify your code and increase readability as long as it does not change the project's final result according to the requirements and the program works properly.

## Class Supply

In the file **supply.py,** the class **Supply** should be implemented. It is a **base class** of any **type of supply,** and it **should not be able to be instantiated**.

### Structure

The class should have the following attributes:

* **name: str**
  + If it is **an empty string,** raise **ValueError** with the message **"Name cannot be an empty string."**
* **energy: int**
  + If it is a **negative number,** raise **ValueError** with the message **"Energy cannot be less than zero."**

### Methods

**\_\_init\_\_(name: str, energy: int)**

The **\_\_init\_\_** method should receive a **name** and **energy**.

**details()**

Return the supply's **type**, **name** and **energy** in the format: **"{type}: {name}, {energy}"**.

The type of the supply is either **"Food"** or **"Drink"**.

Hint: override the method in the child classes.

## Class Food

In the **food.py** file, the class **Food** should be implemented. The food is a **type of supply**. A food has **25 units of energy** as an **optional** parameter.

## Class Drink

In the **drink.py** file, the class **Drink** should be implemented. The drink is a **type of supply**. Each drink has **15 initial units of energy**.

## Class Player

In the **player.py** file, the class **Player** should be implemented. It will store the info of each player.

### Structure

The class should have the following attributes:

* **name: str**
  + If it's set to an **empty string**, raise **ValueError** with the message **"Name not valid!"**
  + **There should not be two players with the same name (they all should be unique). If a second player is created with the same name,** raise **Exception** with the message **"Name {name} is already used!"**
* **age: int**
  + If the player is under **12 years old**, raise **ValueError** with the message **"The player cannot be under 12 years old!"**
* **stamina: int**
  + An optional parameter, **100** by default
  + **Stamina's max value is 100, and its min value is 0**
  + If it is **less than zero or more than 100**, raise **ValueError** with the message **"Stamina not valid!"**
* **need\_sustenance: bool**
  + Returns **if** the player's **stamina** is **less than 100. It is read-only, and it should not be able to be set**

### Methods

**\_\_init\_\_(name: str, age: int, stamina: int)**

Upon initialization, all the needed attributes must be set.

**\_\_str\_\_()**

Override the method so that its return the player's data in the format:

**"Player: {player\_name}, {age}, {stamina}, {need\_sustenance}"**

## Class Controller

In the **controller.py** file, the class **Controller** should be implemented. It will contain **all the functionality** of the project.

### Structure

This class will have the following **attributes**:

* **players: list**
  + An **empty** list that will contain **all the players** (objects)
* **supplies: list**
  + An **empty** list that will contain **all the supplies** (objects)

### Methods

**\_\_init\_\_()**

Upon initialization, all the needed attributes must be set.

**add\_player(player1: Player, player2: Player, … playerN: Player)**

* **Add the players** to the players' list. You should **not** add a player who has **already been added.**
* In the end, **return a message** with the successfully added players' **names**, separated with a comma and a space (**", "**) in the format: **"Successfully added: {name1}, {name2}, … {nameN}"**

**add\_supply(supply1: Supply, supply2: Supply, … supplyN: Supply)**

* **Add all supplies** to the supplies list
* **A supply could be added multiple times**

**sustain(player\_name: str, sustenance\_type: str)**

* **Use the last supply** **added from the given type to sustain the player** (increase his stamina with the supply's energy value and remove the supply from the list) and **return** the message **"{player\_name} sustained successfully with {supply\_name}."**
* A player **always** uses the **whole amount** (units) of the given supply, but his **stamina cannot enhance** **above 100** (it should be **set** to 100).
* If the player **doesn't need sustenance**, it **won't be appropriate to waste a supply**. **Just return** the message **"{player\_name} have enough stamina."**
* If the given type is food, but there is **no food left**, raise an **Exception** with the message **"There are no food supplies left!"**
* If the given type is drink, but there are **no drinks left**, raise an **Exception** with the message **"There are no drink supplies left!"**
* The valid sustenance types are **"Food"** and **"Drink"**. In any other case, **ignore** the command.
* If the **player** is **not** in the players list, **ignore** the command.

**duel(first\_player\_name: str, second\_player\_name: str)**

* The two players participate in a duel, **each** **of them** could only **attack** **once**.
* If a player's stamina is 0, he could not participate in a duel. In that case, return a message **"Player {player\_name} does not have enough stamina."** and **discontinue the duel**. If both players' stamina is 0, return the message for both players on separate lines, starting from the first one given.
* If both players have a **positive value of stamina**, the duel begins:
  + The player with a **lower value** of stamina **attacks first**. He **reduces** the other player's stamina by a value equal to **one-half of his own** (the attacker's) **stamina**.
  + Next, the other player attacks **the same way** (reduces the first player's stamina by a value equal to one-half of his own (the second attacker's) stamina).
  + If, during the duel, a player's **stamina** **becomes equal to or less than 0**, it should be **set to 0**. The player **immediately loses** the duel, and the **other player becomes a winner.**
  + Otherwise, the winner is the player **who has left with more stamina**.
  + Return the winner's name in the format: **"Winner: {winner\_name}"**
* **Note**: there will be **no case** where both players will have **equal stamina values** at the beginning or in the end.
* **Note**: the players will **always exist** in the players list.

**next\_day()**

* First, the **stamina** of each added player **gets reduced** by the result of **multiplying** their **age** by **2**
* If a player's **stamina** **becomes less than 0**, it should be **set to 0**
* Then, you need to **sustain** each player by **giving** them **one food** (first) and **one drink** (second)

**\_\_str\_\_()**

Override the method so that its return the players' data and the supplies' data in the format:

**"Player: {player\_name\_1}, {age}, {stamina}, {need\_sustenance}**

**Player: {player\_name\_2}, {age}, {stamina}, {need\_sustenance}**

**...**

**Player: {player\_name\_N}, {age}, {stamina}, {need\_sustenance}**

**{supply\_type}: {name\_1}, {energy}**

**{supply\_type}: {name\_2}, {energy}**

**...**

**{supply\_type}: {name\_N}, {energy}"**

## Examples

|  |
| --- |
| **Test Code** |
| from project.controller import Controller  from project.player import Player  from project.supply.drink import Drink  from project.supply.food import Food  controller = Controller()  apple = Food("apple", 22)  cheese = Food("cheese")  juice = Drink("orange juice")  water = Drink("water")  first\_player = Player('Peter', 15)  second\_player = Player('Lilly', 12, 94)  print(controller.add\_supply(cheese, apple, cheese, apple, juice, water, water))  print(controller.add\_player(first\_player, second\_player))  print(controller.duel("Peter", "Lilly"))  print(controller.add\_player(first\_player))  print(controller.sustain("Lilly", "Drink"))  first\_player.stamina = 0  print(controller.duel("Peter", "Lilly"))  print(first\_player)  print(second\_player)  controller.next\_day()  print(controller) |
| **Output** |
| None  Successfully added: Peter, Lilly  Winner: Lilly  Successfully added:  Lilly sustained successfully with water.  Player Peter does not have enough stamina.  Player: Peter, 15, 0, True  Player: Lilly, 12, 82.5, True  Player: Peter, 15, 37, True  Player: Lilly, 12, 98.5, True  Food: cheese, 25  Food: apple, 22 |

# Problem 3. Unit Tests

You will **be provided with another skeleton** for this problem. **Open** the **new skeleton** as a **new project** and write tests for the **Movie** class. The class will have some methods, fields, and one constructor, all of them working properly. You are **NOT ALLOWED** to change any class. Cover the whole class with unit tests to make sure that the class is working as intended. Submit **only the test** folder.