Exercise: Polymorphism and Abstraction

This document defines the exercises for the "Python OOP" course at @Software University.

Please, submit your source code solutions for the described problems to the Judge System.

1. Vehicle

Create an **abstract class called `Vehicle`** that should have abstract methods **`drive`** and **`refuel`**. Create **2 vehicles** that **inherit the `Vehicle`** class (a **`Car`** and a **`Truck`**) and simulate **driving** and **refueling** them. **`Car`** and **`Truck`** both receive **`fuel_quantity`** and **`fuel_consumption`** in liters per **km** upon initialization. They both can be driven a given **distance**: **`drive(distance)`** and refueled with a given amount of fuel: **`refuel(fuel)`**. It is summer, so both vehicles use air conditioners, and their fuel consumption per **km** when **driving** is **increased by 0.9 liters** for the **car** and **1.6 liters** for the **truck**. Also, the **`Truck`** has a tiny hole in its tank, and when it is refueled, it keeps only **95% of the given fuel**. The car has no problems and adds all the given fuel to its tank. If a vehicle **cannot travel** the given distance, its fuel **does not change**.

Note: Submit all your classes and imports in the judge system

Examples

Test Code	Output

`car = Car(20, 5)`
`car.drive(3)`
`print(car.fuel_quantity)`
`car.refuel(10)`
`print(car.fuel_quantity)`
	`2.299999999999997`
	`12.299999999999997`

`truck = Truck(100, 15)`
`truck.drive(5)`
`print(truck.fuel_quantity)`
`truck.refuel(50)`
`print(truck.fuel_quantity)`
	`17.0`
	`64.5`

2. Groups

Create a class called **`Person`**. Upon initialization, it will receive a **`name`** (str) and a **`surname`** (str). Implement the needed **magic methods** so that:

- Each person could be represented by their **names, separated by a single space**.
- When you concatenate two people, you should return a **new instance** of a person who will take **the first name from the first person and the surname from the second person**.

Create another class called **`Group`**. Upon initialization, it should receive a **`name`** (str) and **`people`** (list of Person instances). Implement the needed **magic methods** so that:

- When you access the **length of a group instance**, you should receive the **total number of people** in the group.
- When you **concatenate two groups**, you should return a **new instance** of a group which will have a name-string in the format **`"{first_name} {second_name}"`** and **all the people** in the two groups will participate in the new one too.

- Each group should be represented in the format **"Group {name} with members {members' names separated by comma and space}"**
- You could **iterate over a group**, and **each person** (element of the group) should be represented in the format **"Person {index}: {person's name}"**

Examples

Test Code	Output

`p0 = Person('Aliko', 'Dangote')`
`p1 = Person('Bill', 'Gates')`
`p2 = Person('Warren', 'Buffet')`
`p3 = Person('Elon', 'Musk')`
`p4 = p2 + p3`
 	
`first_group = Group('__VIP__', [p0, p1, p2])`
`second_group = Group('Special', [p3, p4])`
`third_group = first_group + second_group`
 	
`print(len(first_group))`
`print(second_group)`
`print(third_group[0])`
 	
`for person in third_group:`
`  print(person)`
	`3`
	`Group Special with members Elon Musk, Warren Musk`
	`Person 0: Aliko Dangote`
	`Person 0: Aliko Dangote`
	`Person 1: Bill Gates`
	`Person 2: Warren Buffet`
	`Person 3: Elon Musk`
	`Person 4: Warren Musk`

3. Account

Create a single class called **Account**. Upon initialization, it should receive an **owner** (str) and a starting **amount** (int, **optional**, 0 by default). It should also have an attribute called **_transactions** (**empty list**). Create the following **methods**:

- **handle_transaction(transaction_amount)**
  - If the balance becomes **less than zero**, raise **ValueError** with the message **"sorry cannot go in debt!"** and **break the transaction**.
  - Otherwise, **complete it**, **save it,** and **return** a message **"New balance: {account_balance}"**
- **add_transaction(amount)**
  - if the amount is **not an integer**, raise **ValueError** with the message **"please use int for amount"**.
  - Otherwise, **check what the balance will be with the new transaction**
    - If the balance becomes **less than zero**, raise **ValueError** with the message **"sorry cannot go in debt!"** and **break the transaction**.
    - Otherwise, **complete it** and **return** a message **"New balance: {account_balance}"**
- **balance()** - a property that returns the **sum** between the **amount** and all the **transactions**

Implement the correct **magic methods** so the code in the example below works properly:

- When you **print** an account instance, the output should be in the format **"Account of {owner} with starting amount: {amount}"**.

- When you print a **representational string** of an account instance, the output should be in the format **`"Account({owner}, {amount})"`**.
- When you access the **length of an account instance**, you should receive the **total number of transactions** made.
- You should **iterate over** an account instance and **receive each transaction** as a result.
- You should be able to **reverse the order of transactions** by reversing an account instance.
- You should be able to **compare `(>, <, >=, <=, ==, !=)`** two account instances **by their balance amount**.
- When you **concatenate two accounts**, you should return a **new account** with a **name-string** in the format **`"{first_owner}&{second_owner}"`** and the **starting amount** - the sum between their two. Both their transactions should be added to the new account.

Examples

Test Code	Output

`acc = Account('bob', 10)`
`acc2 = Account('john')`
`print(acc)`
`print(repr(acc))`
`acc.add_transaction(20)`
`acc.add_transaction(-20)`
`acc.add_transaction(30)`
`print(acc.balance)`
`print(len(acc))`
`for transaction in acc:`
`  print(transaction)`
`print(acc[1])`
`print(list(reversed(acc)))`
`acc2.add_transaction(10)`
`acc2.add_transaction(60)`
`print(acc > acc2)`
`print(acc >= acc2)`
`print(acc < acc2)`
`print(acc <= acc2)`
`print(acc == acc2)`
`print(acc != acc2)`
`acc3 = acc + acc2`
`print(acc3)`
`print(acc3._transactions)`
	`Account of bob with starting amount: 10`
	`Account(bob, 10)`
	`40`
	`3`
	`20`
	`-20`
	`30`
	`-20`
	`[30, -20, 20]`
	`False`
	`False`
	`True`
	`True`
	`False`
	`True`
	`Account of bob&john with starting amount: 10`
	`[20, -20, 30, 10, 60]`

4. Wild Farm

Create the following project structure:

Your task is to create a class **hierarchy** like the one described below. The **`Animal`**, **`Bird`**, **`Mammal`**, and **`Food`** classes should be abstract:

In the **`food.py`** file, implement the following classes:

- **`Food`** - the class should be **abstract** and should receive **`quantity`** (int) upon **initialization**
- **`Vegetable`**, **`Fruit`**, **`Meat`**, and **`Seed`** classes should **inherit** from the **`Food`** class

In the **`animal.py`** file, implement the following classes:

- **`Animal`** - the class should be **abstract** and should have the following attributes:
  - **`name`** (string) - passed upon **initialization**
  - **`weight`** (float) - passed upon **initialization**
  - **`food_eaten`** - 0 by default
- **`Bird`** - should **inherit** from the **`Animal`** class. The class should be **abstract** and should have **`wing_size`** (float) as an additional attribute passed upon initialization.
- **`Mammal`** - should **inherit** from the **`Animal`** class. The class should be **abstract** and should have **`living_region`** (str) as an additional attribute passed upon initialization.

In the **`birds.py`** file, implement the following classes:

- **`Owl`**
- **`Hen`**

In the **`mammals.py`** file, implement the following classes:

- **`Mouse`**
- **`Dog`**
- **`Cat`**
- **`Tiger`**

All **animals** also can ask for food by producing a sound. Create a **`make_sound()`** method that returns the sound:

- **`Owl`** - **`"Hoot Hoot"`**
- **`Hen`** - **`"Cluck"`**
- **`Mouse`** - **`"Squeak"`**
- **`Dog`** - **`"Woof!"`**
- **`Cat`** - **`"Meow"`**
- **`Tiger`** - **`"ROAR!!!"`**

Now use the classes that you have created to instantiate some animals and feed them. Add method **`feed(food)`** where the food will be an instance of some food classes.

Animals will only eat a specific type of food, as follows:

- Hens eat **everything**

- Mice eat **vegetables** and **fruits**
- Cats eat **vegetables** and **meat**
- Tigers, Dogs, and Owls eat only **meat**

If you try to give an animal a **different type** of food, it will not eat it, and you should return:

- **`"{AnimalType} does not eat {FoodType}!"`**

The weight of an animal will increase with every piece of food it eats, as follows:

- Hen - **0.35**
- Owl - **0.25**
- Mouse - **0.10**
- Cat - **0.30**
- Dog - **0.40**
- Tiger - **1.00**

Override the **`__repr__()`** method to print the information about an animal in the formats:

- Birds - **`"{AnimalType} [{AnimalName}, {WingSize}, {AnimalWeight}, {FoodEaten}]"`**
- Mammals - **`"{AnimalType} [{AnimalName}, {AnimalWeight}, {AnimalLivingRegion}, {FoodEaten}]"`**

Note: Submit all your classes and your imports in the judge system

Examples

Test Code	Output

`owl = Owl("Pip", 10, 10)`
`print(owl)`
`meat = Meat(4)`
`print(owl.make_sound())`
`owl.feed(meat)`
`veg = Vegetable(1)`
`print(owl.feed(veg))`
`print(owl)`
	`Owl [Pip, 10, 10, 0]`
	`Hoot Hoot`
	`Owl does not eat Vegetable!`
	`Owl [Pip, 10, 11.0, 4]`

`hen = Hen("Harry", 10, 10)`
`veg = Vegetable(3)`
`fruit = Fruit(5)`
`meat = Meat(1)`
`print(hen)`
`print(hen.make_sound())`
`hen.feed(veg)`
`hen.feed(fruit)`
`hen.feed(meat)`
`print(hen)`
	`Hen [Harry, 10, 10, 0]`
	`Cluck`
	`Hen [Harry, 10, 13.15, 9]`

5. Animals

Your task is to create a class **hierarchy** like the one described below. Submit in judge a **zip file** named **`project`**, containing a **separate file for each of the classes**.

The **Animal** class (**abstract**) should take, attributes, a **name**, an **age**, and a **gender**. It should **have 2 methods**: **repr()** and **make_sound()**.

The **Dog** class should **inherit** and **implement** the **Animal** class. Its **repr()** method should return **"This is {name}. {name} is a {age} year old {gender} {class}"**. The dog sound is **"Woof!"**.

The **Cat** class should **inherit** and **implement** the **Animal** class. Its **repr()** method should **return "This is {name}. {name} is a {age} year old {gender} {class}"**. The cat sounds, **"Meow meow!"**.

The **Kitten** class should **inherit** and **implement** the **Cat** class. Its gender is **"Female"**, and its sound is **"Meow"**.

The **Tomcat** class should **inherit** and **implement** the **Cat** class. Its gender is **"Male"**, and its sound is **"Hiss"**.

Examples

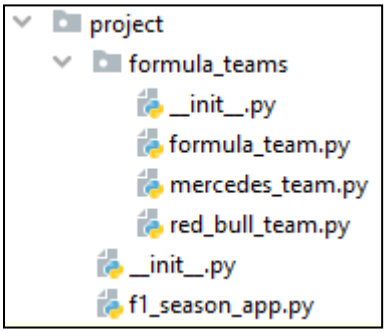
Test Code	Output

`dog = Dog("Rocky", 3, "Male")`
`print(dog.make_sound())`
`print(dog)`
`tomcat = Tomcat("Tom", 6)`
`print(tomcat.make_sound())`
`print(tomcat)`
	`Woof!`
	`This is Rocky. Rocky is a 3 year old Male Dog`
	`Hiss`
	`This is Tom. Tom is a 6 year old Male Tomcat`

`kitten = Kitten("Kiki", 1)`
`print(kitten.make_sound())`
`print(kitten)`
`cat = Cat("Johnny", 7, "Male")`
`print(cat.make_sound())`
`print(cat)`
	`Meow`
	`This is Kiki. Kiki is a 1 year old Female Kitten`
	`Meow meow!`
	`This is Johnny. Johnny is a 7 year old Male Cat`

*6. Formula 1 Manager

For this task, you will be provided with a **skeleton** that includes all the folders and files you need.



Note: You cannot change the folder and file structure and their names!

Judge Upload

Create a **zip** file with the **project 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**.

Description

You are the F1 manager of the two biggest teams in F1, "Red Bull" and "Mercedes". Your task is to create a program that calculates the revenue after every race for both teams. Your app should have the following structure and functionality.

1. Class FormulaTeam

In the **`formula_team.py`** file, the class **`FormulaTeam`** should be implemented. It is a **base class** for any **type of formula team,** and it **should not be able to be instantiated**.

Structure

The class should have the following attributes:

- **`budget: int`**
    - An integer that represents the **budget of the team**.
    - If the budget is **less than 1 000 000**, raise **`ValueError`** with the message: **`"F1 is an expensive sport, find more sponsors!"`**

Methods

__init__(budget: int)

- In the **`__init__`** method, all the needed attributes must be set.

calculate_revenue_after_race(race_pos: int)

- Each team should be able to calculate their revenue
- Each team has its unique sponsors
    - Sponsors give the team money if they finish in a certain position or better
- Each team has a different amount of expenses

2. Class RedBullTeam

In the **`red_bull_team.py`**, the class **RedBullTeam** should be implemented.

Methods

__init__(budget: int)

- In the **`__init__`** method, all the needed attributes must be set.

calculate_revenue_after_race(race_pos: int)

- Red Bull sponsors:
    - Oracle:
        - 1st place – 1 500 000$
        - 2nd place – 800 000$
    - Honda:
        - 8th place – 20 000$
        - 10th place – 10 000$
- Red Bull expenses per race – 250 000$
- To **calculate the revenue** from the race, **sum the earned money** from the sponsors depending on the position in the race and **subtract the expenses**

- After that, **add the result** to the team's budget and **return** the following message: **`"The revenue after the race is { revenue }$. Current budget { current budget }$"`**

Note: Each sponsor gives the money for the best position only. If you are 1st and the sponsor gives money for 1st and 2nd positions, you get the money only for the 1st position!

3. Class MercedesTeam

In the **`Mercedes_team.py`**, the class **MercedesTeam** should be implemented.

Methods

__init__(budget: int)

- In the **`__init__`** method, all the needed attributes must be set.

calculate_revenue_after_race(race_pos: int)

- Mercedes sponsors:
  - Petronas:
    - 1st place – 1 000 000$
    - 3rd place – 500 000$
  - TeamViewer:
    - 5th place – 100 000$
    - 7th place – 50 000$
- Mercedes expenses per race – 200 000$
- To **calculate the revenue** from the race, **sum the earned money** from the sponsors depending on the position in the race and **subtract the expenses**
- After that, **add the result** to the team's budget and **return** the following message: **`"The revenue after the race is { revenue }$. Current budget { current budget }$"`**

Note: Each sponsor gives the money for the best position only. If you are 1st and the sponsor gives money for 1st and 2nd positions, you get the money only for the 1st position!

4. Class F1SeasonApp

In the **`f1_season_app.py`** file, the class **F1SeasonApp** should be implemented. It will contain all the functionality of the project.

Structure

The class should have the following attributes:

- **`red_bull_team`: `RedBullTeam`**
  - It should be **set to None on initialization**.
- **`mercedes_team`: `MercedesTeam`**
  - It should be **set to None on initialization**.

Methods

__init__()

- In the **`__init__`** method, all the needed attributes must be set.

register_team_for_season(team_name: str, budget: int)

- Valid team names: **"Red Bull", "Mercedes"**

- If a **team name is valid**, register the team with the corresponding name and **return** the following message: **`"{ team name } has joined the new F1 season."`**
- If a **team name is invalid**, raise **`ValueError`** with the message: **`"Invalid team name!"`**

Note: There won't be a case where a valid team tries to register for a second time.

new_race_results(race_name: str, red_bull_pos: int, mercedes_pos: int)

- If **Red Bull or Mercedes haven't registered yet**, raise an **`Exception`** with the following message: **`"Not all teams have registered for the season."`**
- Otherwise, find which team has the better position in the race, calculate every team's revenue, update their budget, and return the following message: **`"Red Bull: { Red Bull revenue message }. Mercedes: { Mercedes revenue message }. { team with better position } is ahead at the { race name } race."`**
- ***Note: Teams' positions will always be valid.***

Examples

Input

`from project.f1_season_app import F1SeasonApp`
 	
`f1_season = F1SeasonApp()`
 	
`print(f1_season.register_team_for_season("Red Bull", 2000000))`
`print(f1_season.register_team_for_season("Mercedes", 2500000))`
`print(f1_season.new_race_results("Nurburgring", 1, 7))`
`print(f1_season.new_race_results("Silverstone", 10, 1))`

Output

`Red Bull has joined the new F1 season.`
`Mercedes has joined the new F1 season.`
`Red Bull: The revenue after the race is 1270000$. Current budget 3270000$. Mercedes: The revenue after the race is -150000$. Current budget 2350000$. Red Bull is ahead at the Nurburgring race.`
`Red Bull: The revenue after the race is -240000$. Current budget 3030000$. Mercedes: The revenue after the race is 900000$. Current budget 3250000$. Mercedes is ahead at the Silverstone race.`

"Into turn 9, Verstappen stays ahead!..."