# Exercise: Polymorphism and Abstraction

Problems for exercise and homework for the Python OOP Course @SoftUni.
Submit your solutions in the SoftUni judge system at https://judge.softuni.bg/Contests/1943.

## 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 simulates **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)`<br>`car.drive(3)`<br>`print(car.fuel_quantity)`<br>`car.refuel(10)`<br>`print(car.fuel_quantity)` | `2.299999999999997`<br>`12.299999999999997` |
| `truck = Truck(100, 15)`<br>`truck.drive(5)`<br>`print(truck.fuel_quantity)`<br>`truck.refuel(50)`<br>`print(truck.fuel_quantity)` | `17.0`<br>`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')`<br>`p1 = Person('Bill', 'Gates')`<br>`p2 = Person('Warren', 'Buffet')`<br>`p3 = Person('Elon', 'Musk')`<br>`p4 = p2 + p3`<br><br>`first_group = Group('__VIP__', [p0, p1, p2])`<br>`second_group = Group('Special', [p3, p4])`<br>`third_group = first_group + second_group`<br><br>`print(len(first_group))`<br>`print(second_group)`<br>`print(third_group[0])`<br><br>`for person in third_group:`<br>`  print(person)` | `3`<br>`Group Special with members Elon Musk, Warren Musk`<br>`Person 0: Aliko Dangote`<br>`Person 0: Aliko Dangote`<br>`Person 1: Bill Gates`<br>`Person 2: Warren Buffet`<br>`Person 3: Elon Musk`<br>`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**:

- **add_transaction(amount)** - if the amount is **not an integer**, raise **ValueError** with the message **"please use int for amount"**. Otherwise, **add the amount** to the transactions
- **balance()** - a property that returns the **sum** between the **amount** and all the **transactions**
- **validate_transaction(account: Account, amount_to_add)**
  - 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}"**

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 **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)`<br>`acc2 = Account('john')`<br>`print(acc)`<br>`print(repr(acc))`<br>`acc.add_transaction(20)`<br>`acc.add_transaction(-20)`<br>`acc.add_transaction(30)`<br>`print(acc.balance)`<br>`print(len(acc))`<br>`for transaction in acc:`<br>`  print(transaction)`<br>`print(acc[1])`<br>`print(list(reversed(acc)))`<br>`acc2.add_transaction(10)`<br>`acc2.add_transaction(60)`<br>`print(acc > acc2)`<br>`print(acc >= acc2)`<br>`print(acc < acc2)`<br>`print(acc <= acc2)`<br>`print(acc == acc2)`<br>`print(acc != acc2)`<br>`acc3 = acc + acc2`<br>`print(acc3)`<br>`print(acc3._transactions)` | `Account of bob with starting amount: 10`<br>`Account(bob, 10)`<br>`40`<br>`3`<br>`20`<br>`-20`<br>`30`<br>`-20`<br>`[30, -20, 20]`<br>`False`<br>`False`<br>`True`<br>`True`<br>`False`<br>`True`<br>`Account of bob&john with starting amount: 10`<br>`[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)`<br>`print(owl)`<br>`meat = Meat(4)`<br>`print(owl.make_sound())`<br>`owl.feed(meat)`<br>`veg = Vegetable(1)`<br>`print(owl.feed(veg))`<br>`print(owl)` | `Owl [Pip, 10, 10, 0]`<br>`Hoot Hoot`<br>`Owl does not eat Vegetable!`<br>`Owl [Pip, 10, 11.0, 4]` |
| `hen = Hen("Harry", 10, 10)`<br>`veg = Vegetable(3)`<br>`fruit = Fruit(5)`<br>`meat = Meat(1)`<br>`print(hen)`<br>`print(hen.make_sound())`<br>`hen.feed(veg)`<br>`hen.feed(fruit)`<br>`hen.feed(meat)`<br>`print(hen)` | `Hen [Harry, 10, 10, 0]`<br>`Cluck`<br>`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, as 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 sound, **`"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")`<br>`print(dog.make_sound())`<br>`print(dog)`<br>`tomcat = Tomcat("Tom", 6)`<br>`print(tomcat.make_sound())`<br>`print(tomcat)` | `Woof!`<br>`This is Rocky. Rocky is a 3 year old Male Dog`<br>`Hiss`<br>`This is Tom. Tom is a 6 year old Male Tomcat` |
| `kitten = Kitten("Kiki", 1)`<br>`print(kitten.make_sound())`<br>`print(kitten)`<br>`cat = Cat("Johnny", 7, "Male")`<br>`print(cat.make_sound())`<br>`print(cat)` | `Meow`<br>`This is Kiki. Kiki is a 1 year old Female Kitten`<br>`Meow meow!`<br>`This is Johnny. Johnny is a 7 year old Male Cat` |