# Exercise: Polymorphism

Problems for exercise and homework for the [Python OOP Course @SoftUni](https://softuni.bg/courses/python-oop). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1943>

## 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 have fuel\_quantity, fuel\_consumption in liters per **km** and 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 **with 1.6 liters** for the **truck**. Also, the Truck has a tiny hole in its tank and when it's 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 |

## Groups

Create a class called **Person**. Upon initialization it will receive a **name** (str) and a **surname** (str). 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 the test code below works

### 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 |

## Account

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

* **add\_transaction(amount)** – if the amount is **not an integer**, raise **ValueError** with message **"please use int for amount"**, otherwise, **add the amount** to the transactions
* **balance()** – property that returns **sum** of the **amount** and all the **transactions**
* **validate\_transaction(account: Account, amount\_to\_add)** – if the transaction is **possible**, **add** it. If the balance becomes **less than zero**, raise **ValueError** with message **"sorry cannot go in debt!"** and **break the transaction**. Otherwise, **complete it** and **return** a message **"New balance: {account\_ballance}"**

Implement the correct **magic methods**, so the code in the example bellow works properly

### 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] |

## Wild Farm

Create the following project structure:



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

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

* Food - quantity (int) - **abstract class**
  + Vegetable
  + Fruit
  + Meat
  + Seed

In the animal.py file implement the Animal, Bird and Mammal classes. In the birds.py file implement the bird classes and in the mammals.py file implement the mammal classes

* Animal - name (string), weight (float), food\_eaten (attribute, 0 upon initialization) - **abstract class**
  + Bird - wing\_size (float) - **abstract class**
    - Owl
    - Hen
  + Mammal - living\_region (string) - **abstract class**
    - Mouse
    - Dog
    - Cat
    - Tiger

All **animals** should also have the ability to ask for food by producing a sound. 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 instance of some of the food classes.

**Animals** will only eat a certain 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] |

## Animals

Your task is to create a class **hierarchy** like the described below.

The **Animal** class (**abstract**) should take, as attributes, **a name, age and gender**. It should **have 2 methods**: **repr()** and **make\_sound().**

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

The **Cat** class should **inherit** and **implement** the **Animal** class. Her repr should **return "This is {name}. {name} is a {age} year old {gender} {class}"**. The **make\_sound()** method should return **"Meow meow!"**

The **Kitten** class should **inherit** and **implement** the **Cat** class. The kitten shouldn't accept "**gender**" as a parameter but it should have its value set to **"Female".** The **make\_sound()** method should return **"Meow"**

The **Tomcat** class should **inherit** and **implement** the **Cat** class. The tomcat shouldn't accept "**gender**" as a parameter but it should have its value set to **"Male".** The **make\_sound()** method should return **"Hiss"**