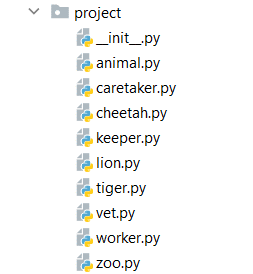
# Exercise: Encapsulation

Please, submit your source code solutions for the described problems to the [Judge System](https://alpha.judge.softuni.org/Contests/Encapsulation-Exercise/1939).

**Ask** **your** **questions** here: [https://www.slido.com](https://www.slido.com/) by entering the course code **#python-advanced**

## Wild Cat Zoo

Create a separate file for each class as shown below and submit a zip file containing all files (zip the whole project folder/module) - it is important to include all files in the project module to make proper imports.



The **Animal** class is a **base class** for any type of animal in the zoo. Upon initialization, it should receive **four public attributes**: a **name** (string), a **gender** (str), an **age** (int), and a **money\_for\_care** (int).  
The **Animal** class should also have **1 additional method**:

* **\_\_repr\_\_()** - returns string representation of the animal in the format: **"Name: {name}, Age: {age}, Gender: {gender}"**

The **Lion**, the **Tiger**, and the **Cheetah** classes should **inherit** from the **Animal** class. Each of these animals costs a certain **amount of money to be cared for**:

* A lion needs **50**
* A tiger needs **45**
* A cheetah needs **60**

The **Worker** class is a **base class** for any type of employee in the zoo. It should receive three public attributes - a **name** (string), an **age** (int), and a **salary** (int) upon initialization.

The **Worker** class should also have **one method**:

* **\_\_repr\_\_()** - returns string representation of the workers in the format: **"Name: {name}, Age: {age}, Salary: {salary}"**

The **Keeper**, the **Caretaker**,and the **Vet** classes should **inherit** from the **Worker** class.

The **Zoo** class should receive 4 attributes upon initialization:

* **Public** attribute **name: string**
* **Private** attribute **budget: int**
* **Private** attribute **animal\_capacity: int**
* **Private** attribute **workers\_capacity: int**

It should also have 2 instance attributes:

* Public attribute **animals: list** -(empty upon initialization)
* Public attribute **workers: list** - (empty upon initialization)

The **Zoo** class should also have **8 methods**:

* **add\_animal(animal, price)**
* If you have **enough budget** and **capacity** **add** the animal (instance of Lion/Tiger/Cheetah) to the **animals' list**, **reduce** the **budget**, and **return** **"{name} the {type of animal (Lion/Tiger/Cheetah)} added to the zoo"**
* If you have the capacity, but **no budget**, return **"Not enough budget"**
* In any other case, you **do not have space**, and you should return **"Not enough space for animal"**
* **hire\_worker(worker)**
* If you have **not exceeded** the capacity of workers in the zoofor the worker (instance of Keeper/Caretaker/Vet), **add him** to the workers and return **"{name} the {type(Keeper/Vet/Caretaker)} hired successfully"**
* Otherwise, return **"Not enough space for worker"**
* **fire\_worker(worker\_name)**
* If there **is a worker** with that name in the workers' list, **remove** him and return **"{worker\_name} fired successfully"**
* Otherwise, return **"There is no {worker\_name} in the zoo"**
* **pay\_workers()**
* If you have **enough budget** to pay the workers (sum their salaries) **pay them** and return **"You payed your workers. They are happy. Budget left: {left\_budget}"**
* Otherwise, return **"You have no budget to pay your workers. They are unhappy"**
* **tend\_animals()**
* If you have **enough budget** to take care of the animals, **reduce the budget** and return **"You tended all the animals. They are happy. Budget left: {left\_budget}"**
* Otherwise, return **"You have no budget to tend the animals. They are unhappy."**
* **profit(amount)**
* **Increase the budget** with the given amount of profit
* **animals\_status()**
  + Returns the following string (***Hint***: use the **\_\_repr\_\_** methods of the animals to print them on the console):

**"You have {total\_animals\_count} animals****----- {amount\_of\_lions} Lions:  
{lion1}  
…**

**{lionN}  
----- {amount\_of\_tigers} Tigers:  
{tiger1}  
…**

**{tigerN}  
----- {amount\_of\_cheetahs} Cheetahs:  
{cheetah1}  
…**

**{cheetahN}"**

* **workers\_status()**
  + Returns the following string (***Hint***: use the **\_\_repr\_\_** methods of the workers to print them on the console):

**"You have {total\_workers\_count} workers  
----- {amount\_of\_keepers} Keepers:  
{keeper1}  
…**

**{keeperN}  
----- {amount\_of\_caretakers} Caretakers:  
{caretaker1}  
…**

**{caretakerN}  
----- {amount\_of\_vetes} Vets:  
{vet1}  
…**

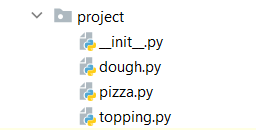
**{vetN}"**

### Examples

|  |
| --- |
| **Test Code** |
| from project.caretaker import Caretaker  from project.cheetah import Cheetah  from project.keeper import Keeper  from project.lion import Lion  from project.tiger import Tiger  from project.vet import Vet  from project.zoo import Zoo  zoo = Zoo("Zootopia", 3000, 5, 8)  # Animals creation  animals = [Cheetah("Cheeto", "Male", 2), Cheetah("Cheetia", "Female", 1), Lion("Simba", "Male", 4), Tiger("Zuba", "Male", 3), Tiger("Tigeria", "Female", 1), Lion("Nala", "Female", 4)]  # Animal prices  prices = [200, 190, 204, 156, 211, 140]  # Workers creation  workers = [Keeper("John", 26, 100), Keeper("Adam", 29, 80), Keeper("Anna", 31, 95), Caretaker("Bill", 21, 68), Caretaker("Marie", 32, 105), Caretaker("Stacy", 35, 140), Vet("Peter", 40, 300), Vet("Kasey", 37, 280), Vet("Sam", 29, 220)]  # Adding all animals  for i in range(len(animals)):  animal = animals[i]  price = prices[i]  print(zoo.add\_animal(animal, price))  # Adding all workers  for worker in workers:  print(zoo.hire\_worker(worker))  # Tending animals  print(zoo.tend\_animals())  # Paying keepers  print(zoo.pay\_workers())  # Fireing worker  print(zoo.fire\_worker("Adam"))  # Printing statuses  print(zoo.animals\_status())  print(zoo.workers\_status()) |
| **Output** |
| Cheeto the Cheetah added to the zoo  Cheetia the Cheetah added to the zoo  Simba the Lion added to the zoo  Zuba the Tiger added to the zoo  Tigeria the Tiger added to the zoo  Not enough space for animal  John the Keeper hired successfully  Adam the Keeper hired successfully  Anna the Keeper hired successfully  Bill the Caretaker hired successfully  Marie the Caretaker hired successfully  Stacy the Caretaker hired successfully  Peter the Vet hired successfully  Kasey the Vet hired successfully  Not enough space for worker  You tended all the animals. They are happy. Budget left: 1779  You payed your workers. They are happy. Budget left: 611  Adam fired successfully  You have 5 animals  ----- 1 Lions:  Name: Simba, Age: 4, Gender: Male  ----- 2 Tigers:  Name: Zuba, Age: 3, Gender: Male  Name: Tigeria, Age: 1, Gender: Female  ----- 2 Cheetahs:  Name: Cheeto, Age: 2, Gender: Male  Name: Cheetia, Age: 1, Gender: Female  You have 7 workers  ----- 2 Keepers:  Name: John, Age: 26, Salary: 100  Name: Anna, Age: 31, Salary: 95  ----- 3 Caretakers:  Name: Bill, Age: 21, Salary: 68  Name: Marie, Age: 32, Salary: 105  Name: Stacy, Age: 35, Salary: 140  ----- 2 Vets:  Name: Peter, Age: 40, Salary: 300  Name: Kasey, Age: 37, Salary: 280 |

## Pizza Maker

Create a separate file for each class as shown below and submit a zip file containing all files (zip the whole project folder/module) - it is important to include all files in the project module to make proper imports.



Create a class called **Topping**. Upon initialization, it should receive:

* **topping\_type: str** - if the topping is an **empty string**, raise a **ValueError** with the message **"The topping type cannot be an empty string"**
* **weight: float** - if the weight is **0 or less**, raise a **ValueError** with the message **"The weight cannot be less or equal to zero"**

Hint: Use **Getters** and **Setters.**

Create a class called **Dough**. Upon initialization, it should receive:

* **flour\_type: str** - if the flour type is an **empty string**, raise a **ValueError** with the message **"The flour type cannot be an empty string"**
* **baking\_technique: str** - if the technique is an **empty string**, raise a **ValueError** with the message **"The baking technique cannot be an empty string"**
* **weight: float** - if the weight is **0 or less**, raise a **ValueError** with the message **"The weight cannot be less or equal to zero"**

Create a class called **Pizza**. Upon initialization, it should receive:

* **name: str** - if the name is an **empty string**, raise a **ValueError** with the message **"The name cannot be an empty string"**
* **dough: Dough** - if the dough is **None**, raise a **ValueError** with the message **"You should add dough to the pizza"**
* **max\_number\_of\_toppings: int** – represents the maximum **number of toppings** **the pizza should have**. If it is **0 or less**, raise a **ValueError** with the message **"The maximum number of toppings cannot be less or equal to zero"**
* **toppings: dict** – empty dictionary upon initialization containing the **topping type** **as a key** and the **topping's weight as a value**.

The class should also have 2 instance methods:

* **add\_topping(topping: Topping)**
  + **Add** a new topping to the dictionary
* If there is **no space left** for **a new topping**, raise a **ValueError**: **"Not enough space for another topping"**
* If the topping is **already in the dictionary**, **increase the value of its weight**.
* **calculate\_total\_weight()** - returns the total weight of the pizza (dough's weight and toppings' weight)

### Examples

|  |
| --- |
| **Test Code** |
| from project.dough import Dough  from project.pizza import Pizza  from project.topping import Topping  tomato\_topping = Topping("Tomato", 60)  print(tomato\_topping.topping\_type)  print(tomato\_topping.weight)  mushrooms\_topping = Topping("Mushroom", 75)  print(mushrooms\_topping.topping\_type)  print(mushrooms\_topping.weight)  mozzarella\_topping = Topping("Mozzarella", 80)  print(mozzarella\_topping.topping\_type)  print(mozzarella\_topping.weight)  cheddar\_topping = Topping("Cheddar", 150)  pepperoni\_topping = Topping("Pepperoni", 120)  white\_flour\_dough = Dough("White Flour", "Mixing", 200)  print(white\_flour\_dough.flour\_type)  print(white\_flour\_dough.weight)  print(white\_flour\_dough.baking\_technique)  whole\_wheat\_dough = Dough("Whole Wheat Flour", "Mixing", 200)  print(whole\_wheat\_dough.weight)  print(whole\_wheat\_dough.flour\_type)  print(whole\_wheat\_dough.baking\_technique)  p = Pizza("Margherita", whole\_wheat\_dough, 2)  p.add\_topping(tomato\_topping)  print(p.calculate\_total\_weight())  p.add\_topping(mozzarella\_topping)  print(p.calculate\_total\_weight())  p.add\_topping(mozzarella\_topping) |
| **Output** |
| Tomato  60  Mushroom  75  Mozzarella  80  White Flour  200  Mixing  200  Whole Wheat Flour  Mixing  260  340  ValueError: Not enough space for another topping |

## Football Team Generator

Create a separate file for each class as shown below and submit a zip file containing all files (zip the whole project folder/module) - it is important to include all files in the project module to make proper imports.

Create a class called **Player**. Upon initialization, it should receive:

* Private attribute **name: string**
* Private attribute **sprint: int**
* Private attribute **dribble: int**
* Private attribute **passing: int**
* Private attribute **shooting: int**

You should create property only for the name of the player. The class should also have one additional method:

Override the **\_\_str\_\_()** method of the class so it returns**:**

**"Player: {name}**

**Sprint: {sprint}**

**Dribble: {dribble}**

**Passing: {passing}**

**Shooting: {shooting}"**

Create a class called **Team**. Upon initialization, it should receive:

* Private attribute **name: string**
* Private attribute **rating: int**

The class should also have a private instance attribute - **players: list** -empty list upon initialization that will contain all the players (objects)

The **Team** class has the following methods:

* **add\_player(player: Player)**
* If the player **is already in the team**, return **"Player {name} has already joined"**
* **Otherwise, add the player** to the team and return **"Player {name} joined team {team\_name}"**
* **remove\_player(player\_name: str)**
* **Remove the player** and **return him**
* If the player **is not in the team**, return **"Player {player\_name} not found"**

### Examples

|  |
| --- |
| **Test Code** |
| from project.player import Player  from project.team import Team  p = Player("Pall", 1, 3, 5, 7)  print("Player name:", p.name)  print("Points sprint:", p.\_Player\_\_sprint)  print("Points dribble:", p.\_Player\_\_dribble)  print("Points passing:", p.\_Player\_\_passing)  print("Points shooting:", p.\_Player\_\_shooting)  print("\ncalling the \_\_str\_\_ method")  print(p)  print("\nAbout the team")  t = Team("Best", 10)  print("Team name:", t.\_Team\_\_name)  print("Teams points:", t.\_Team\_\_rating)  print("Teams players:", len(t.\_Team\_\_players))  print(t.add\_player(p))  print(t.add\_player(p))  print("Teams players:", len(t.\_Team\_\_players))  print(t.remove\_player("Pall"))  print(t.remove\_player("Pall")) |
| **Output** |
| Player name: Pall  Points sprint: 1  Points dribble: 3  Points passing: 5  Points shooting: 7  calling the \_\_str\_\_ method  Player: Pall  Sprint: 1  Dribble: 3  Passing: 5  Shooting: 7  About the team  Team name: Best  Teams points: 10  Teams players: 0  Player Pall joined team Best  Player Pall has already joined  Teams players: 1  Player: Pall  Sprint: 1  Dribble: 3  Passing: 5  Shooting: 7  Player Pall not found |

## Restaurant **[Solve with AI]**

Create a **restaurant** with the following classes and hierarchy:



Submit in judge a **zip file** containing a separate file for each of the classes using the structure shown below:



The **Product** class should have the following **private** **attributes** and subsequent **getters**:

* **name: string**
* **price: float**

**Beverage** and **Food** classes are **products**:

* The **Beverage** class should have an additional **private** **attribute –** **milliliters: float** and its subsequent **getter**
* The Food class should have an additional private **attribute – grams: float** and its subsequent **getter**

**HotBeverage** and **ColdBeverage** are **beverages.**

**Coffee** and **Tea** are **hot beverages:**

* The Coffee class should have an additional **private** **attribute – caffeine: float** and its subsequent **getter**. Itshould also have the following **class** **attributes**, which should apply to all coffees made:
* **MILLILITERS = 50 (constant)**
* **PRICE = 3.50 (constant)**

Starter, MainDish, and Dessert are **food**:

* The Dessert class should have an additional **private** **attribute - calories - float** and its subsequent **getter**

**Salmon** is the **main dish**. Also, it must have the following class attribute, which should apply to all salmons:

* **GRAMS = 22 (constant)**

**Soup** is a **starter**.

**Cake** is a **dessert**. Also, it must have the following **class attributes** which should apply to all cakes made:

* **GRAMS = 250 (constant)**
* **CALORIES = 1000 (constant)**
* **PRICE = 5 (constant)**

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| product = Product("coffee", 2.5)  print(product.\_\_class\_\_.\_\_name\_\_)  print(product.name)  print(product.price)  beverage = Beverage("coffee", 2.5, 50)  print(beverage.\_\_class\_\_.\_\_name\_\_)  print(beverage.\_\_class\_\_.\_\_bases\_\_[0].\_\_name\_\_)  print(beverage.name)  print(beverage.price)  print(beverage.milliliters)  soup = Soup("fish soup", 9.90, 230)  print(soup.\_\_class\_\_.\_\_name\_\_)  print(soup.\_\_class\_\_.\_\_bases\_\_[0].\_\_name\_\_)  print(soup.name)  print(soup.price)  print(soup.grams) | Product  coffee  2.5  Beverage  Product  coffee  2.5  50  Soup  Starter  fish soup  9.9  230 |