# Python OOP Regular Exam - 10 August 2024

[**Link to Judge**](https://alpha.judge.softuni.org/contests/python-oop-regular-exam-10-august-2024/4797)



*As a new enterprise dedicated to crafting high-quality wood products through specialized manufacturing processes, our primary focus is on producing exceptional furniture and toys, tailored to meet the unique demands of our diverse clientele. With a state-of-the-art factory and a commitment to excellence, we ensure that every product, from elegant chairs to delightful toys, embodies superior craftsmanship and innovative design.*

You are provided with a **skeleton** that includes all folders and files you need.

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

A screenshot of a computer

Description automatically generated

**Judge Upload**

For the **first two 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 **test folder** and **upload it** to the Judge system.

Do not 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)

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

You can **add additional attributes** (instance attributes, class attributes, methods, dunder methods, etc.) to simplify your code and increase readability if it does not change the project's final result following its requirements and proper workflow.

## Class BaseProduct

In the **base\_product.py** file, the class **BaseProduct** should be implemented. It serves as a **base class** for **any type** of **product** and should **not be instantiated directly**.

### Structure

The class should have the following attributes:

* **model:** str
  + The value represents the **model** of the **product**.
  + If the model contains **less than 3 characters** or **white spaces only**, raise a ValueError with the message: **"Product model must be at least 3 chars long!"**
* **price:** float
  + The value represents the **price** of the **product**.
  + If the price is **less than or equal to** **0.0**, raise a ValueError with the message: **"Product price must be greater than zero!"**
* **material:** str
  + The value represents the **material** of the **product**.
* **sub\_type: str**
  + **The value represents the sub-type of the product.**

### Methods

#### \_\_init\_\_(model: str, price: float, material: str, sub\_type: str)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### discount()

* **Decreases** the **product's price** by applying a **discount**. Keep in mind that **each type of product** implements the method **differently**.

## Class Chair

In the **chair.py** file, the class **Chair** should be implemented. The **chair** is a **type** of[**BaseProduct**](#_Class_BaseProduct). Each **chair** is made of wood (the **material** property must be set to **'Wood'**) and has a **sub-type** of **'Furniture'**.

### Methods

#### \_\_init\_\_(model: str, price: float)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### discount()

* The method **applies** a **discount of 10%** to the **chair's price**.

## Class HobbyHorse

In the **hobby\_horse.py** file, the class **HobbyHorse** should be implemented. The **hobby horse** is a type of [**BaseProduct**](#_Class_BaseFactory). Each **hobby horse** is made of wood and plastic parts (the **material** property must be set to **'Wood/Plastic'**) and has a **sub-type** of **'Toys'**.

### Methods

#### \_\_init\_\_( model: str, price: float)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### discount()

* The method **applies** a **discount of 20%** to the **hobby horse's price**.

## Class BaseStore

In the **base\_store.py** file, the class **BaseStore** should be implemented. It serves as a **base class** for **any type** of **store** and should **not be instantiated directly**.

### Structure

The class should have the following attributes:

* **name:** str
  + The value represents the **name of the store**.
  + If the name is an **empty string** or **contains only white spaces**, **raise** a **ValueError** with the message: **"Store name cannot be empty!"**
* **location:** str
  + The value represents the **abbreviation** of the **store location**. Example: 'LA2'
  + The location **must** **contain exactly 3 characters** and **should have no white spaces**. If it does not, raise a **ValueError** with the message: **"Store location must be 3 chars long!"**
* **capacity:** int
  + The value represents the **store's capacity**.
  + If the capacity **drops below** **zero**, raise a **ValueError** with the message: **"Store capacity must be a positive number or 0!"**
* **products:** list
  + A **list** containing **products** (objects) each **store** has for sale.
  + **Initially** set to an **empty list**.

### Methods

#### \_\_init\_\_(name: str, location: str, capacity: int)

* In the **\_\_init\_\_** method, all needed attributes must be set.

#### get\_estimated\_profit()

* **Computes** the **estimated future profit**, based on the **products** the **store has in stock** (**products** collection) and an **additional percentage** of **10%**.
* **Sum** the **prices for all products in stock**,then **compute** the **estimated profit** by **multiplying** the **sum and** thegiven **percentage** value of **10%**.

**Example**: If the total sum of all product prices is **100.0**, then the **estimated profit** is calculated as **100.0 \* 10% = 10.0**

* **Returns** a string in the format: **"Estimated future profit for {products\_count} products is {profit}"**
* Format the **profit value** to the **second decimal place**.

#### store\_type()

* **Returns** the **type** of the **store as a property**. Keep in mind that **each** type of **store** **implements** the method **differently**.

**Hint**: Use a **property decorator** to define this method as a property, making it **accessible** like **<store\_object>.store\_type**

#### store\_stats()

* Returns **detailed information** about the store. Keep in mind that **each** type of **store** **implements** the method **differently**.

## Class FurnitureStore

In the **furniture\_store.py** file, the class **FurnitureStore** should be implemented. A **furniture store** is a **type** of[**BaseStore**](#_Class_BaseFactory_1). The **furniture store** sells **only** **furniture products** and has an **initial capacity** of **50**.

### Methods

#### \_\_init\_\_(name: str, location: str)

* In the **\_\_init\_\_** method, all needed attributes must be set.

#### store\_type()

* Returns the **type** of the **store as a property**, set to: **"FurnitureStore"**

**Hint**: Use a **property decorator** to define this method as a property, making it **accessible** like **<store\_object>.store\_type**

#### store\_stats()

* The method **returns detailed information** about the **available products**, **prices**,and **capacity** in the following format (each row on a new line):

**"Store: {name}, location: {location}, available capacity: {capacity}**

**Estimated future profit for {products\_count} products is {profit}**

**\*\*Furniture for sale:**

**{model1}: {num\_of\_product\_pieces}pcs, average price: {avg\_price\_per\_model}**

**...**

**{modeln}: {num\_of\_product\_pieces}pcs, average price: {avg\_price\_per\_model}"**

* **Sort** the **available furniture** by their **model ascending** and provide the **number** of **products** **per model**.
* The **average price per model** is calculated based on the **prices** of the **products** under **this model** and **their number**.
* **Format** the **profit** and **average price** to the **second decimal place**.

**Hint:** You can use the[**get\_estimated\_profit()**](#_get_estimated_profit())method.

## Class ToyStore

In the **toy\_store.py** file, the class **ToyStore** should be implemented. A **toy store** is a **type** of[**BaseStore**](#_Class_BaseFactory_1). The **toy store** sells **only** **toy products** and has an **initial capacity** of **100**.

### Methods

#### \_\_init\_\_(name: str, location: str)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### store\_type()

* **Returns** the **type** of the **store** **as a property**, set to: **"ToyStore"**

**Hint**: Use a **property decorator** to define this method as a property, making it **accessible** like **<store\_object>.store\_type**

#### store\_stats()

* The method **returns detailed information** about the **available products**, **prices**,and **capacity** in the following format (each row on a new line):

**"Store: {name}, location: {location}, available capacity: {capacity}**

**Estimated future profit for {products\_count} products is {profit}**

**\*\*Toys for sale:**

**{model1}: {num\_of\_product\_pieces}pcs, average price: {avg\_price\_per\_model}**

**...**

**{modeln}: {num\_of\_product\_pieces}pcs, average price: {avg\_price\_per\_model}"**

* **Sort** the **available toys** by their **model ascending** and provide the **number** of **products** **per model**.
* The **average price per model** is calculated based on the **prices** of the **products** under **this model** and **their number**.
* **Format** the **profit** and **average price** to the **second decimal place**.

**Hint:** You can use the[**get\_estimated\_profit()**](#_get_estimated_profit()) method.

## Class FactoryManager

In the **factory\_manager.py** file, the class **FactoryManager** should be implemented. It will **manage** the **production** and **sales to stores**.

### Structure

The class should have the following attributes:

* **name: str**
  + The value represents the name of the factory.
* **income: float**
  + The value represents the income the **factory generates** by **selling products**.
  + **Initially** set to **zero** (**0.0**).
* **products: list**
  + A list **containing all products** (objects) that the factory produces.
  + **Initially** an **empty list**.
* **stores: list**
  + A list **containing all stores** (objects) the factory has as partners.
  + **Initially** an **empty list**.

### Methods

#### \_\_init\_\_(name: str)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### produce\_item(product\_type: str, model: str, price: float)

The method produces(**creates**) a **product object** of the **given type**, **model**, and **price** and **adds** it to the **products** collection.

* If the **type** is **not valid**, raise an **Exception** with the following message:

**"Invalid product type!"**

* Otherwise, **create** the product, **add** it to the **products list**, and **return** the following message:

**"A product of sub-type {product\_sub\_type} was produced."**

* **Valid types** of **products** are: **"Chair"** and **"HobbyHorse"**

#### register\_new\_store(store\_type: str, name: str, location: str)

The method registers(**creates**) a **store object** of the **given type** with the given name and location and **adds** it to the **stores** collection.  
All **store names** will be **unique**.

* **First**, check if the **store type** is **valid**: **'FurnitureStore'** or **'ToyStore'**.

If not, **raise** an **Exception** with the following message:

**"{store\_type} is an invalid type of store!"**

* Otherwise, **create** the store object, **add** it to the **stores** list, and **return** the following message:

**"A new {store\_type} was successfully registered."**

#### sell\_products\_to\_store(store: BaseStore, \*products: BaseProduct)

The method initiates a **sale** ofthe **provided collection** of **product** objects (**one or many**)to the **given store** object. The **store and products** will **always exist**,and **product objects** will be **unique**.

* **First**, check if the store has **enough capacity** to **buy all items**.

If not, **return** the following message:

**"Store {name} has no capacity for this purchase."**

* **Then**, start **checking** the product **sub-types** to see if they **match** the **store type**:
  + **FurnitureStore** sells products of the **sub-type 'Furniture'**, while **ToyStore** sells only items with the **sub-type 'Toys'**.
  + **Filter** **only those** that **match** and **add** them to the **store products collection**.
* You should also **remove** **them** from the **factory collection of products**.
* **Decrease** the **store's available capacity** with the **number** of **purchased products**.
* **Sum up** the **prices** of the **purchased products** and **add** the **amount** to the **factory income**.
  + If there are **suitable items** that the **store can accept** and **sell**, **return** the following message:

**"Store {name} successfully purchased {num\_of\_purchased\_products} items."**

* If **no products** were sold due to **type mismatch**, **return**:

**"Products do not match in type. Nothing sold."**

#### unregister\_store(store\_name: str)

The method **removes the store** with the given **name** from the **factory stores collection**.

* **First**, check if there is a store with the given **name** in the **stores collection**.

If not, **raise an Exception** with thefollowing message:

**"No such store!"**

* **Then**, check if the store **has unsold products**.If so, **return** thefollowing message:

**"The store is still having products in stock! Unregistering is inadvisable."**

* If the store can be **unregistered successfully**, **remove** it from the factory **stores collection**, and **return** the following message:

**"Successfully unregistered store {store\_name}, location: {store\_location}."**

#### discount\_products(product\_model: str)

The method **decreases** the **prices** for the **unsold products** from the **factory products collection**, **filtered** by the **provided** **model** (which will always be an existing model, but can be of a sold item), and **returns** the following message:

**"Discount applied to {products\_count} products with model: {model}"**

* The discount **depends on** the **product sub-type**:
  + **Furniture products** get a **10% discount** while **toy products** get a **20% discount**.

**Hint**: Use the product's [**discount()**](#_discount()) method to decrease the prices for the different types of products properly.

#### request\_store\_stats(store\_name: str)

The method **returns** up-to-date **store statistics** for the **store** with the **given name**.

* If there is **no such store**, **return** a message:

**"There is no store registered under this name!"**

* Otherwise, **return** the **string** that the **store statistics method** generates.

**Note**: Use the store's [**store\_stats()**](#_store_stats()) method to generate the statistics properly.

#### statistics()

**Returns** structured and **sorted** information about the **factory** and its partner **stores**. Each string is on a **new line**.

"Factory: {factory\_name}

Income: {factory\_income}

\*\*\*Products Statistics\*\*\*

Unsold Products: {factory\_products\_count}. Total net price: {products\_sum\_price}

{product\_model1}: {count}

...

{product\_modeln}: {count}

\*\*\*Partner Stores: {total\_stores\_count}\*\*\*

{store\_name1}

...

{store\_namen}"

* The factory income and the total products' net price should be formatted to the 2nd decimal place.
* Include a breakdown of unsold products by model, showing each product model and its count.
* Product models and store names are sorted in ascending order.

#### Examples

|  |
| --- |
| **Test Code** |
| ***# Initialize the FactoryManager***  factory\_manager = FactoryManager("Cool Factory")  ***# Produce some items***  print(factory\_manager.produce\_item("Chair", "Classic", 80.0))  print(factory\_manager.produce\_item("Chair", "Modern", 100.0))  print(factory\_manager.produce\_item("Chair", "Modern", 200.0))  print(factory\_manager.produce\_item("HobbyHorse", "Rocking Horse", 120.0))  print(factory\_manager.produce\_item("HobbyHorse", "Rocking Horse", 100.0))  print()  ***# Register new stores***  print(factory\_manager.register\_new\_store("FurnitureStore", "Furniture Outlet", "SOF"))  print(factory\_manager.register\_new\_store("ToyStore", "Toy World", "VAR"))  print()  ***# Sell products to stores***  chair1 = factory\_manager.products[0]  chair2 = factory\_manager.products[1]  chair3 = factory\_manager.products[2]  store1 = factory\_manager.stores[0]  store2 = factory\_manager.stores[1]  print(factory\_manager.sell\_products\_to\_store(store2, chair1, chair2))  print(factory\_manager.sell\_products\_to\_store(store1, chair1, chair2, chair3))  print()  ***# Unregister store***  print(factory\_manager.unregister\_store("Furniture Outlet"))  print()  ***# Discount products***  print(factory\_manager.discount\_products("Classic"))  print(factory\_manager.discount\_products("Rocking Horse"))  print()  ***# Request store statistics***  print(factory\_manager.request\_store\_stats("Furniture Outlet"))  print(factory\_manager.request\_store\_stats("Toy World"))  print()  ***# Factory statistics***  print(factory\_manager.statistics())  print()  ***# Unregister store***  print(factory\_manager.unregister\_store("Toy World")) |
| **Output** |
| A product of sub-type Furniture was produced.  A product of sub-type Furniture was produced.  A product of sub-type Furniture was produced.  A product of sub-type Toys was produced.  A product of sub-type Toys was produced.  A new FurnitureStore was successfully registered.  A new ToyStore was successfully registered.  Products do not match in type. Nothing sold.  Store Furniture Outlet successfully purchased 3 items.  The store is still having products in stock! Unregistering is inadvisable.  Discount applied to 0 products with model: Classic  Discount applied to 2 products with model: Rocking Horse  Store: Furniture Outlet, location: SOF, available capacity: 47  Estimated future profit for 3 products is 38.00  \*\*Furniture for sale:  Classic: 1pcs, average price: 80.00  Modern: 2pcs, average price: 150.00  Store: Toy World, location: VAR, available capacity: 100  Estimated future profit for 0 products is 0.00  \*\*Toys for sale:  Factory: Cool Factory  Income: 380.00  \*\*\*Products Statistics\*\*\*  Unsold Products: 2. Total net price: 176.00  Rocking Horse: 2  \*\*\*Partner Stores: 2\*\*\*  Furniture Outlet  Toy World  Successfully unregistered store Toy World, location: VAR. |

# Task 3: Unit Tests (100 points)

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