# Python OOP Exam – 2021

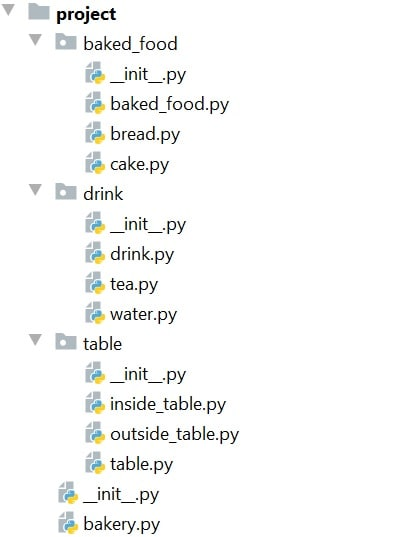
# Bakery



*As we all love baked delicacies, today you were chosen to build a simple bakery software system. This system must have support for baked foods, drinks and tables in the bakery. The project will consist of classes, which manage the interaction between the baked foods, the drinks and the tables.*

You will be provided with a **skeleton**, which includes all the folders and files that you will need.

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



# Judge Upload

For the **first 2 problems**, create a **zip** file with the name **project** and upload it to the judge system

For the **last problem**, create a **zip** file with the name **tests** and upload it to the judge system

# Structure (Problem 1) and Functionality (Problem 2)

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

You are **free to add additional attributes** (instance attributes, class attributes, methods, dunder methods, etc.) to simplify your code and increase readability as long as it does not change the project's final result according to the requirements and the program works properly.

### BakedFood

In the **baked\_food.py** file the class BakedFood should be implemented. It is a **base class** of any **type of baked food,** and it **should not be able to be instantiated**.

### Structure

The class should have the following attributes:

* **name**: **string** - passed upon **initialization.**
  + If the name **is an empty string or whitespace,** raise a **ValueError** with the message**: "Name cannot be empty string or white space!"**
* **portion**: **float** - passed upon **initialization.** It represents the size of the baked food in grams.
* **price**: **float** -passed upon **initialization.**
  + If the price **is less than or equal to 0,** raise a **ValueError** with the message: **"**Price cannot be less than or equal to zero!**"**

### Methods

#### \_\_init\_\_(name: str, portion: float, price: float)

The **\_\_init\_\_** method should have a **name, portion,** and **price**.

**\_\_repr\_\_()**

Override the repr method so it returns a string with information about each food in the following format:

**" - {baked\_food\_name}: {portion}g - {price}lv"**

The portion size and the price should be formatted to the **second decimal point**.

### Bread

In the file **bread.py** the class **Bread** should be implemented.

**Structure**

The class should **inherit** from the BakedFood class.

**Methods**

**\_\_init\_\_(name: str, price: float)**

Еach **bread** **portion** has initial size of 200 grams.

### Cake

In the file **cake.py** the class **Cake** should be implemented.

**Structure**

The class should **inherit** from the BakedFood class.

**Methods**

**\_\_init\_\_(name: str, price: float)**

Еach **cake** **portion** has initial size of 245 grams.

### Drink

In the **drink.py** file the class Drink should be implemented. **Drink** is a **base class** for any **type of drink,** and it **should not be able to be instantiated**.

### Structure

The class should have the following attributes:

* **name**: **string** - passed upon **initialization.**
  + If the name **is an empty string or whitespace,** raise a **ValueError** with the message **"Name cannot be empty string or white space!"**
* **portion:** **int** - passed upon **initialization.** It represents the size of the drink in milliliters.
  + If the portion **is less than or equal to 0,** raise a **ValueError** with message **"**Portion cannot be less than or equal to zero!**"**
* **price:** **float** - passed upon **initialization.**
* **brand:** **string** - passed upon **initialization.** 
  + If the brand name **is an empty string or whitespace,** raise a **ValueError** with the message **"Brand cannot be empty string or white space!"**

### Methods

#### \_\_init\_\_(name: str, portion: int, price: float, brand: str)

The **\_\_init\_\_** method should have a **name**, a **portion**,a **price** and a **brand.**

##### \_\_repr\_\_()

Override the repr method so it returns a **string** with the information about **each drink** in the following format:

" - {drink\_name} {brand\_name} - {portion}ml - {price}lv"

The portion size and the price should be formatted to the **second decimal point**.

### Tea

In the file **tea.py** the class **Tea** should be implemented.

**Structure**

The class should **inherit** from the Drink class.

**Methods**

**\_\_init\_\_(name: str, portion: int, brand: str)**

Each tea costs exatly 2.50.

### Water

In the file **water.py** the class **Water** should be implemented.

**Structure**

The class should **inherit** from the Drink class.

**Methods**

**\_\_init\_\_(name: str, portion: int, brand: str)**

Each water costs exatly 1.50.

### Table

In the **table.py** file the class Table should be implemented. It is a **base** **class** for every type of **table** and **should not be able to be instantiated.**

### Structure

The class should have the following attributes:

* **table\_number: int** -passed upon **initialization.**
* **capacity: int** -passed upon **initialization.** It represents the table's seat capacity.
  + If it is **less than or equal to zero,** raise a **ValueError** with the message "Capacity has to be greater than 0!"
* **food\_orders: empty list containing every food order made from the table.**
* **drink\_orders: empty list containing every drink order made from the table.**
* **number\_of\_people:** int - the count of people who sit at the table. **0 by default**.
* **is\_reserved: Bool - False by default. Returns True if the table is reserved.**

### Methods

#### \_\_init\_\_(table\_number: int, capacity: int)

The **\_\_init\_\_** method should have a **table\_number**, a **capacity**, **food\_orders**, **drink\_orders**, **number\_of\_people** and **is\_reserved**.

##### reserve(number\_of\_people: int)

Reserves the table with the count of people given.

##### order\_food(baked\_food: BakedFood)

Orders the provided food.

##### order\_drink(drink: Drink)

Orders the provided drink.

##### get\_bill()

Returns the bill for all the ordered drinks and food.

##### clear()

Removes all the ordered drinks and food and finally frees the seats at the table.

##### free\_table\_info()

**Only if the table is** **free**, returns a string in the following format:

"Table: {table\_number}"

"Type: {table\_type}"

"Capacity: {table\_capacity}"

### InsideTable

In the file **inside\_table.py** the class **InsideTable** should be implemented.

**Structure**

The class should **inherit** from the Table class.

**Methods**

**\_\_init\_\_(table\_number: int, capacity: int)**

The inside table's number could only be between **1** and **50** **inclusive**. Otherwise,raise a **ValueError** with message "Inside table's number must be between 1 and 50 inclusive!".

### OutsideTable

In the file **outside\_table.py** the class **OutsideTable** should be implemented.

**Structure**

The class should **inherit** from the Table class.

**Methods**

**\_\_init\_\_(table\_number: int, capacity: int)**

The **outside** **table's number** could only be between **51** and **100** **inclusive**. Otherwise, raise a **ValueError** with message "Outside table's number must be between 51 and 100 inclusive!".

### Bakery

In the file **bakery.py** the class **Bakery** should be implemented.

**Structure**

The class should have the following attributes:

* **name: str** -passed upon **initialization**.
  + If the name **is an empty string or whitespace,** raise a **ValueError** with a message **"Name cannot be empty string or white space!"**
* **food\_menu: empty list that will contain every type of food in the bakery's menu.**
* **drinks\_menu: empty list that will contain every type of drink in the bakery's menu.**
* **tables\_repository: empty list containing every table at the bakery**.
* **total\_income -** the total income from all the completed bills. **0 by default**.

**Methods**

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

The **\_\_init\_\_** method should have a **name**, a **food\_menu**, **drinks\_menu**, **tables\_repository**, and **total\_income**.

#### add\_food (food\_type: str, name: str, price: float)

Creates a food with the correct type and adds it to the menu. The possible types of food are **"Bread"** and "**Cake**". If the food is created and added successfully, returns:

"Added {baked\_food\_name} ({baked\_food\_type}) to the food menu"

**If a baked food with the given name already exists in the food menu, raise an Exception with message** "{food\_type} {name} is already in the menu!"

#### add\_drink (drink\_type: str, name: str, portion: int, brand:str)

Creates a drink with the correct type and adds it to the menu. The possible types of drinks are **"Tea"** and "**Water**". If the drink is created and added successfully, returns:

**"Added {drink\_name} ({drink\_brand}) to the drink menu"**

**If a drink with the given name already exists in the drink repository, raise** Exception **with message** "{drink\_type} {name} is already in the menu!"

#### add\_table (table\_type: str, table\_number: int, capacity: int)

Creates a table with the correct type, adds it to the table respository. The possible types of tables are **"InsideTable"** and "**OutsideTable**". If the table is created and added successfully, returns:

"Added table number {table\_number} in the bakery"

**If a table with the given number already exists in the table repository, raise **Exception** with message "**Table {table\_number} is already in the bakery!"

#### reserve\_table (number\_of\_people: int)

Finds the first possible table which is not reserved, and its capacity is enough for the number of people provided. Then reserves the table and returns:

"Table {table\_number} has been reserved for {number\_of\_people} people"

Otherwise, returns:

"No available table for {number\_of\_people} people"

#### order\_food (table\_number: int, food\_name1: str, food\_name2: str …)

The **order\_food** method will receive a **table's number** and **different number** of **strings** with **food's names**.

Finds the table with that number. If there is no such table returns:

"Could not find table {table\_number}"

Otherwise, adds the food which could be ordered (are in the menu) in the table's orders, returns the information about the ordered food and the food that is not in the menu in the format:

"Table {table\_number} ordered:

- {baked\_food\_name1}: {portion1}g - {price1}lv

- {baked\_food\_name2}: {portion2}g - {price2}lv

...

- {baked\_food\_nameN}: {portionN}g - {priceN}lv

{bakery\_name} does not have in the menu:

{food\_name\_not\_in\_the\_menu1}

{food\_name\_not\_in\_the\_menu2}

...

{food\_name\_not\_in\_the\_menuN}"

#### order\_drink (table\_number: int, drinks\_name1: str, drink\_name2: str …)

The **order\_drink** method will receive a **table's number** and **different number** of **strings** with **drink's names**.

Finds the table with that number. If there is no such table, it returns:

"Could not find table {table\_number}"

Otherwise, adds the drinks which could be ordered (are in the menu) in the table's orders, returns orders of the drinks which are in the menu and the ones that are not:

"Table {table\_number} ordered:

- {drink\_name1} {brand\_name1} - {portion1}ml - {price1}lv

- {drink\_name2} {brand\_name2} - {portion2}ml - {price2}lv

...

- {drink\_nameN} {brand\_nameN} - {portionN}ml - {priceN}lv

{bakery\_name} does not have in the menu:

{drink\_name\_not\_in\_the\_menu1}

{drink\_name\_not\_in\_the\_menu2}

...

{drink\_name\_not\_in\_the\_menuN}"

#### leave\_table (table\_number: int)

Finds the table with the same table number. Gets the bill for that table and clears it. Finally returns:

"Table: {table\_number}"

"Bill: {table\_bill}"

The bill price should be formatted to the **second decimal point**.

#### get\_free\_tables\_info()

For each free table returns the table info. Each table info should start on a new row.

#### get\_total\_income()

Returns the total income in the format, formatted to the **second decimal point**:

"Total income: {income}lv"

# Problem 3. Unit Tests

You will **be provided with another skeleton** for this problem. You should **write tests** for the **PetShop** class. It will have some **properties**, **methods**, and a **constructor**, all of them **working properly**. You are **NOT ALLOWED** to change the class. Cover the whole class with unit tests to make sure that the class is working as intended. Submit **only the test** folder.