# Exercise: Classes and Objects

Problems for exercise and homework for the [Python Fundamentals Course @SoftUni](https://softuni.bg/trainings/3368/python-fundamentals-may-2021).

Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1734>.

***Note: Submit only the classes in the judge system for all tasks. Test your classes with your own code to see if they work properly.***

## Storage

Create a **class Storage**. The **\_\_init\_\_** method should accept **one parameter** - the **capacity** of the storage. The Storage class should also have an attribute called **storage**, where all the items will be stored. The class should have two additional methods:

* **add\_product(product)** - adds the product in the storage **if there is space** for it
* **get\_products()** - returns the storage list

class Storage:  
 def \_\_init\_\_(self, capacity):  
 self.capacity = capacity  
 self.storage = []  
  
 def add\_product(self, name):  
 if len(self.storage) < self.capacity:  
 self.storage.append(name)  
  
 def get\_products(self):  
 return self.storage

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| storage = Storage(4)  storage.add\_product("apple")  storage.add\_product("banana")  storage.add\_product("potato")  storage.add\_product("tomato")  storage.add\_product("bread")  print(storage.get\_products()) | ['apple', 'banana', 'potato', 'tomato'] |

## Weapon

Create a **class Weapon**. The **\_\_init\_\_** method should receive a number of **bullets** (**integer**). Create an attribute called **bullets** to store them. The class should also have the following methods:

* **shoot()** - if there are bullets in the weapon, **reduce** them **by 1** and return a message **"shooting…"**. If there are **no bullets** left, return: **"no bullets left"**.
* **\_\_repr\_\_()** - returns **"Remaining bullets: {amount\_of\_bullets}"**. You can read more about the **\_\_repr\_\_** method here: [link](https://www.journaldev.com/22460/python-str-repr-functions).

class Weapon():  
  
 def \_\_init\_\_(self, bullets):  
 self.bullets = bullets  
  
 def shoot(self):  
 if self.bullets > 0:  
 self.bullets -= 1  
  
 result = "shooting..."  
 else:  
 result = "no bullets left"  
 return result  
  
 def \_\_repr\_\_(self):  
 result = f"Remaining bullets: {self.bullets}"  
 return result

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| weapon = Weapon(5)  print(weapon.shoot())  print(weapon.shoot())  print(weapon.shoot())  print(weapon.shoot())  print(weapon.shoot())  print(weapon.shoot())  print(weapon) | shooting…  shooting…  shooting…  shooting…  shooting…  no bullets left  Remaining bullets: 0 |

## Catalogue

Create a **class Catalogue**. The **\_\_init\_\_** method should accept the **name** of the catalogue. Each catalogue should also have an **attribute** called **products** and it should be an empty **list**. The class should also have **three more methods**:

* **add\_product(product\_name)** - add the product to the products' list
* **get\_by\_letter(first\_letter)** - returns a **list** containing only the products that start with the given letter
* **\_\_repr\_\_** - returns the catalogue info in the following format:   
  **"Items in the {name} catalogue:  
  {item1}  
  {item2}**

**…**

**{itemN}**"  
The items should be **sorted alphabetically in ascending order**.

class Catalogue:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.products = []  
  
 def add\_product(self, product\_name):  
 self.products.append(product\_name)  
  
 def get\_by\_letter(self, first\_letter):  
 return [product for product in self.products if product[0] == first\_letter]  
  
 def \_\_repr\_\_(self):  
 result = f"Items in the {self.name} catalogue:\n"  
 result += '\n'.join(sorted(self.products))  
 return result

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| catalogue = Catalogue("Furniture")  catalogue.add\_product("Sofa")  catalogue.add\_product("Mirror")  catalogue.add\_product("Desk")  catalogue.add\_product("Chair")  catalogue.add\_product("Carpet")  print(catalogue.get\_by\_letter("C"))  print(catalogue) | ['Chair', 'Carpet']  Items in the Furniture catalogue:  Carpet  Chair  Desk  Mirror  Sofa |

## Town

Create a **class Town**. The **\_\_init\_\_** method should receive the **name** of the town. It should also have **3 more methods**:

* **set\_latitude(latitude)** - sets an attribute called **latitude** to the given one
* **set\_longitude(longitude)** - sets an attribute called longitude to the given one
* **\_\_repr\_\_** - returns representation of the object in the following string format:   
  **"Town: {name} | Latitude: {latitude} | Longitude: {longitude}"**

class Town:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.latitude = ''  
 self.longitude = ''  
  
 def set\_latitude(self, latitude):  
 self.latitude = latitude  
  
 def set\_longitude(self, longitude):  
 self.longitude = longitude  
  
 def \_\_repr\_\_(self):  
 return f"Town: {self.name} | Latitude: {self.latitude} | Longitude: {self.longitude}"

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| town = Town("Sofia")  town.set\_latitude("42° 41\' 51.04\" N")  town.set\_longitude("23° 19\' 26.94\" E")  print(town) | Town: Sofia | Latitude: 42° 41' 51.04" N | Longitude: 23° 19' 26.94" E |

## Class

Create a **class Class**. The **\_\_init\_\_** method should receive the **name** of the class. It should also have **2 lists** (**students** and **grades**). Create a **class attribute \_\_students\_count** equal to **22**. The class should also have **3 additional methods**:

* **add\_student(name, grade)** - if there is **space** in the class, **add** the **student** and the **grade** in the two lists
* **get\_average\_grade()** - returns the **average** of all existing **grades** formatted to the **second decimal** point (as a **number**)
* **\_\_repr\_\_** - returns the string (**single line**): **"The students in {class\_name}: {students}. Average grade: {get\_average\_grade()}".** The students must be seperated by **", "**

class Class:  
 \_\_students\_count = 22  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.students = []  
 self.grades = []  
  
 def add\_student(self, name, grade):  
 if len(self.students) < Class.\_\_students\_count:  
 self.students.append(name)  
 self.grades.append(grade)  
  
 def get\_average\_grade(self):  
 return round(sum(self.grades) / len(self.grades), 2)  
  
 def \_\_repr\_\_(self):  
 return f"The students in {self.name}: {', '.join(self.students)}. Average grade: {self.get\_average\_grade()}"

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| a\_class = Class("11B")  a\_class.add\_student("Peter", 4.80)  a\_class.add\_student("George", 6.00)  a\_class.add\_student("Amy", 3.50)  print(a\_class) | The students in 11B: Peter, George, Amy. Average grade: 4.77 |

## Inventory

Create a **class Inventory**. The **\_\_init\_\_** method should accept only the **\_\_capacity** (private attribute) of the inventory. You can read more about private attributes [here](https://www.tutorialsteacher.com/python/private-and-protected-access-modifiers-in-python). Each inventory should also have an attribute called **items**, where all the items will be stored. The class should also have **3 methods**:

* **add\_item(item)** - adds the item in the inventory if there is space for it. Otherwise, returns   
  **"not enough room in the inventory"**
* **get\_capacity()** - returns the value of **\_\_capacity**
* **\_\_repr\_\_()** - returns **"Items: {items}.\nCapacity left: {left\_capacity}"**. The items should be separated by **", "**

class Inventory:  
  
 def \_\_init\_\_(self, capacity):  
 self.items = []  
 self.\_\_capacity = capacity  
  
 def add\_item(self, item):  
 if len(self.items) < self.\_\_capacity:  
 self.items.append(item)  
 else:  
 return "not enough room in the inventory"  
  
 def get\_capacity(self):  
 return self.\_\_capacity  
  
 def \_\_repr\_\_(self):  
 result = f"Items: {', '.join(self.items)}.\nCapacity left: {self.\_\_capacity - len(self.items)}"  
 return result

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| inventory = Inventory(2)  inventory.add\_item("potion")  inventory.add\_item("sword")  print(inventory.add\_item("bottle"))  print(inventory.get\_capacity())  print(inventory) | not enough room in the inventory  2  Items: potion, sword.  Capacity left: 0 |

## Articles

Create a **class Article**. The **\_\_init\_\_** method should accept **3 arguments**: **title**, **content**, **author**. The class should also have **4 methods**:

* **edit(new\_content)** - changes the old content to the new one
* **change\_author(new\_author)** - changes the old author with the new one
* **rename(new\_title)** - changes the old title with the new one
* **\_\_repr\_\_()** - returns the following string **"{title} - {content}: {author}"**

class Article:  
 def \_\_init\_\_(self, title, content, author):  
 self.title = title  
 self.content = content  
 self.author = author  
  
 def edit(self, new\_content):  
 self.content = new\_content  
  
 def change\_author(self, new\_author):  
 self.author = new\_author  
  
 def rename(self, new\_title):  
 self.title = new\_title  
  
 def \_\_repr\_\_(self):  
 return f"{self.title} - {self.content}: {self.author}"

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| article = Article("some title", "some content", "some author")  article.edit("new content")  article.rename("new title")  article.change\_author("new author")  print(article) | new title - new content: new author |

## \* Vehicle

Create a **class Vehicle**. The **\_\_init\_\_** method should receive a **type** (str), a **model** (str), and a **price** (int). You should also set an **owner** to **None**. The class should have the following methods:

* **buy(money, owner)**
  + If the person **has enough money** and the vehicle has **no owner**, **sets** the **owner** to the given one and returns: **"Successfully bought a {type}. Change: {change}"**. Change should be formatted to the **second decimal place**.
  + If the **money is not enough**, return: **"Sorry, not enough money"**
  + If the car **already has** an **owner**, return: **"Car already sold"**
* **sell()** - if the car **has an owner**, set it to **None** again. Otherwise, return: **"Vehicle has no owner**"
* **\_\_repr\_\_()** - returns **"{model} {type} is owned by: {owner}"** if the vehicle **has** an **owner**. Otherwise, return: **"{model} {type} is on sale: {price}"**

class Vehicle:  
 def \_\_init\_\_(self, type, model, price):  
 self.owner = None  
 self.type = type  
 self.model = model  
 self.price = price  
  
 def buy(self, money, owner):  
 if self.owner:  
 return "Car already sold"  
 elif self.price > money:  
 return "Sorry, not enough money"  
 else:  
 self.owner = owner  
 return f"Successfully bought a {self.type}. Change: {money - self.price:.2f}"  
  
 def sell(self):  
 if not self.owner:  
 return "Vehicle has no owner"  
 self.owner = None  
  
 def \_\_repr\_\_(self):  
 if self.owner is not None:  
 return f"{self.model} {self.type} is owned by: {self.owner}"  
 return f"{self.model} {self.type} is on sale: {self.price}"

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| vehicle\_type = "car"  model = "BMW"  price = 30000  vehicle = Vehicle(vehicle\_type, model, price)  print(vehicle.buy(15000, "Peter"))  print(vehicle.buy(35000, "George"))  print(vehicle)  vehicle.sell()  print(vehicle) | Sorry, not enough money  Successfully bought a car. Change: 5000.00  BMW car is owned by: George  BMW car is on sale: 30000 |

## \* Movie

Create a **class Movie**. The **\_\_init\_\_** method should receive a **name** and a **director**. It should also have default value of an attribute **watched** set to **False**. There should also be a class attribute **\_\_watched\_movies** which will keep track of the number of all the watched movies. The class should have the following methods:

* **change\_name(new\_name)** - changes the name of the movie
* **change\_director(new\_director)** - changes the director of the movie
* **watch()** - change the **watched** attribute to **True** and **increase** the **total watched** movies class attribute (if the movie **is not already watched**)
* **\_\_repr\_\_()** - returns **"Movie name: {name}; Movie director: {director}. Total watched movies: {\_\_watched\_movies}"**

class Movie:  
 \_\_watched\_movies = 0  
  
 def \_\_init\_\_(self, name, director):  
 self.name = name  
 self.director = director  
 self.watched = False  
  
 def change\_name(self, new\_name):  
 self.name = new\_name  
  
 def change\_director(self, new\_director):  
 self.director = new\_director  
  
 def watch(self):  
 if not self.watched:  
 self.watched = True  
 Movie.\_\_watched\_movies += 1  
  
 def \_\_repr\_\_(self):  
 return f"Movie name: {self.name}; Movie director: {self.director}. Total watched movies: {Movie.\_\_watched\_movies}"

### Example

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| first\_movie = Movie("Inception", "Christopher Nolan")  second\_movie = Movie("The Matrix", "The Wachowskis")  third\_movie = Movie("The Predator", "Shane Black")  first\_movie.change\_director("Me")  third\_movie.change\_name("My Movie")  first\_movie.watch()  third\_movie.watch()  first\_movie.watch()  print(first\_movie)  print(second\_movie)  print(third\_movie) | Movie name: Inception; Movie director: Me. Total watched movies: 2  Movie name: The Matrix; Movie director: The Wachowskis. Total watched movies: 2  Movie name: My Movie; Movie director: Shane Black. Total watched movies: 2 |