Sviluppo applicazioni e analisi dati con Python

Simone Capodivento

February 2026

Contents

1 Introduction

Questo documento ha l'obiettivo di esaminare i vari esercizi fatti durante il corso di Python.
A cura di Matteo Martinelli

2 Car Dealer Exercise

2.1 Documentazione della classe Car

We started by creating a small documentation and we looked about the .txt file.

2.2 Questo è il file dal quale prendiamo i dati

```
Toyota Yaris of 2017 with 110000 km.
brand: Toyota
model: Yaris
color: metallic grey
year: 2017
accessories: antifog lights, apple car play, rear camera
km: 110000
license_plate: ZZ999YY
vin: 1234567
price: 10000
currency: EUR
***
brand: Opel
model: corsa
color: black white
year: 2020
accessories: antifog lights, rear camera, level 2 adas
km: 60000
license_plate: AB123YZ
vin: 7654321
price: 8000
currency: EUR
***
brand: Ferrari
model: F40
color: red
year: 1989
```

```
accessories: coraggio
km: 5642
license_plate: M0401IT
vin: 1234
price: 100000000
currency: EUR
***
```

3 Class Car

We looked forward to creating a car class, which is a subclass of the Object class, the father of all classes. Inside this class, we declare that it is of type int, and we start creating all the attributes like brand, model, etc. The attributes need to be of type str, int, or set based on the content that they contain. After that, we wrote the error handling section, which shows for every attribute the type of error (exception raised) and the eventual checks to do. We then wrote:

```
class Car(object):
    _next_id: int = 1000

    def __init__(
            self,
            brand: str,
            model: str,
            color: str,
            year: int,
            accessories: set,
            km: int,
            license_plate: str,
            vin: str,
            price: int,
            currency: str
    ):
        pass
```

Note that `self` in Python is similar to `this` in Java. The getter section is where the program accesses the attributes of our car class, so after the collection of the data they return them. That is the structure of a getter for the attribute `brand`:

```
@property
def brand(self):
    return self._brand
```

We wrote the setters only for accessories, price and license_plate:

```
@accessories.setter
def accessories(self, value: str, remove=False):
    if not isinstance(value, str):
        raise TypeError(f"The accessory must be a string; received {value} of type "
                        f"{type(value)}")
    if not remove:
        self._accessories.add(value)
        print(f"Accessory {value} correctly added!")
    if remove:
        self._accessories.remove(value)
        print(f"Accessory {value} correctly removed!")
```

These methods add or remove values. We also wrote a `classmethod` decorator that checks the id. We create a static method for `license_plate` that validates it and returns a bool. With the slice operator, it checks the composition of the license plate, and if it does not follow the Italian format (AA111AA), it returns false.

We created a `warehouse.py` file. It does the following: first, it imports `datetime` and `Car` from `src.car`. We start declaring the `Warehouse` class, which is also a subclass of Object. We wrote the error handling section about the attributes of our warehouse, such as name, address, and phone number. We set the type of our variables, then created the `car_list` and `cars_collection`. We started writing the getters for all the attributes of our warehouse. Then, we have the error handling in case of wrong attribute types.

We created a `load_cars_from_file` method that loads data from our `cars_warehouse.txt` document. This section reads the document in a `for` loop, appends a new car to our collection, prints the message "New car added to the collection", and raises a `FileNotFoundError` if the file is missing.

```
def load_cars_from_file(self, path='../db/cars_warehouse.txt'):
    try:
        with open(path, 'r', encoding='utf-8') as file:
            car_info = dict()
            for line in file:
                if line == '***\n' or line == '***':
                    print(f"Loading the car {car_info['brand']} {car_info['model']}")
                    new_car = Car(
                        car_info['brand'],
                        car_info['model'],
                        car_info['color'],
                        int(car_info['year']),
                        car_info['accessories'],
                        int(car_info['km']),
                        car_info['license_plate'],
                        car_info['vin'],
                        int(car_info['price']),
                        car_info['currency'],
                    )
                    self._cars_collection.append(new_car)
                    car_info = dict()
                    print('New car added to the list')
                else:
                    line_info = line.split(':')
                    line_info[0] = line_info[0].strip()
                    line_info[1] = line_info[1].strip()
                    if line_info[0] == 'accessories':
                        accessories_list = line_info[1].split(',')
                        for i in range(len(accessories_list)):
                            accessories_list[i] = accessories_list[i].strip()
                        car_info[line_info[0]] = set(accessories_list)
                    else:
                        car_info[line_info[0]] = line_info[1]
    except FileNotFoundError as e:
        print('An error occurred!\n', e)
```

We also created a `save_car_in_file` method: This reads the txt document and writes to it. It also raises `FileNotFoundError` and prints whether a car was correctly written or if an error occurred.

```
def save_car_in_file(self, car_to_save: Car, path='../db/cars_warehouse.txt'):
    self._cars_collection.append(car_to_save)
    try:
        with open(path, 'a', encoding='utf-8') as file:
            accessories_str = ''
            accessories_list = list(car_to_save.accessories)
            for i in range(len(accessories_list)):
                if i+1 == len(accessories_list):
                    accessories_str += accessories_list[i]
                else:
                    accessories_str += accessories_list[i]
                    accessories_str += ', '

            list_to_write = [
                '\n'
                'brand: ' + str(car_to_save.brand) + '\n',
                'model: ' + str(car_to_save.model) + '\n',
                'color: ' + str(car_to_save.color) + '\n',
                'year: ' + str(car_to_save.year) + '\n',
                'accessories: ' + accessories_str + '\n',
                'km: ' + str(car_to_save.km) + '\n',
                'license_plate: ' + str(car_to_save.license_plate) + '\n',
                'vin: ' + str(car_to_save.vin) + '\n',
                'price: ' + str(car_to_save.price[:-1]) + '\n',
                'currency: ' + str(car_to_save.price[-1:]) + '\n',
                '***'
```

```
            ]
            file.writelines(list_to_write)
            print('Car written successfully!')
    except FileNotFoundError as e:
        print('An error occurred!\n', e)
```

We started to implement some utility methods, such as `average_car_value`:

```
def average_car_value(self):
    value = 0
    for car in self.cars_collection:
        value += int(car.price[:-1])
    average = value / len(self.cars_collection)
    return average
```

the `average_car_kilometers`:

```
def average_car_kilometers(self):
    km = 0
    for car in self.cars_collection:
        km += car.km
    average = km / len(self.cars_collection)
    return average
```

and the `average_car_age`:

```
def average_car_age(self):
    age = 0
    for car in self.cars_collection:
        car_age = datetime.now().year - car.year
        age += car_age
    average = age / len(self.cars_collection)
    return average
```

We take all of those operations, and we write them down into another txt file called `info.txt`. We also raise a `FileNotFoundError` just in case, read the file, and print out if an error occurred.

We created our `main` only after that. We started by importing both the `Warehouse` and `Car` classes. We gave a name to our warehouse instance along with a name, address, and phone number as an example. We print out the statistics of our warehouse, such as the average price, average km, and average age of cars. We also started creating a simple command-line menu (still to be completed):

```
# Menu da linea di comando
class Menu(object):
    __menu_instance = None
    __initialized = False

    def __new__(cls, *args, **kwargs):
        if cls.__menu_instance is not None:
            return cls.__menu_instance
        else:
            instance = super().__new__(cls)
            cls.__menu_instance = instance
            return instance

    def __init__(self):
        self.system_warehouse = Warehouse(
            name='Il Paradiso del Pistone',
            address='Via del carburante 10',
            phone_number='+39 399 123456'
        )
        self.options = {
            0: 'exit',
            1: '',
            2: '',
            3: '',
            4: '',
            5: '',
```

```
            6: '',
            7: '',
            8: '',
        }
```

In order to wrap up our program, we wrote unit tests for the `Car` class since it is the most important component. We start by importing it, and we declare that the result must be a boolean (pass or fail). We provide an example car, raise a general exception if something goes wrong, and append `False` if it does.