Python Object Oriented

Guizani Ahmed

```python
# oriente objet
# class and instance attribute

class item:
    pay_rate = 0.8 # 20% discount defined in the class level
    def __init__(self, electronic: str, price: float, quantity: int):

        # run validation to received argument
        assert price >= 0, f"Price {price} is not greater or equal than 0!"
        assert quantity >= 0, f"Quantity {quantity} is not equal or greater than zero!"

        # les variable a definir pour l'objet
        self.electronic = electronic
        self.price = price
        self.quantity = quantity

    # methode to calculate the total of price quantity
    def calculate_total_price (self):
        return self.quantity * self.price

    # here the discount are token from the class level if not defined as entry
    def apply_discount (self):
        self.price = self.price * self.pay_rate

# now we need to have 20% discount for laptop and 30% for phone
# pour item 1 le discount est pris en charge directement de class level pay_rate
item1= item ("phone", 200, 12)
item1.apply_discount()
print(item1.price)
# pour item 2 j'ai forcer la definition de par_rate pour discount de 30%
item2= item ("laptop", 600, 20)
item2.pay_rate=0.7
item2.apply_discount()
print(item2.price)
```

Purely object-oriented, and for the discount made by class level and exception

Here to store the base to give in a list

```python
class item:
    pay_rate = 0.8 # 20% discount defined in the class level
    list = [] # the list of all data
    def __init__(self, electronic: str, price: float, quantity: int):

        # run validation to received argument
        assert price >= 0, f"Price {price} is not greater or equal than 0!"
        assert quantity >= 0, f"Quantity {quantity} is not equal or greater than

        # les variable a definir pour l'objet
        self.electronic = electronic
        self.price = price
        self.quantity = quantity
        # ici on va ajouter les data a la list
        self.list.append(self)

    # methode to calculate the total of price quantity
    def calculate_total_price (self):
        return self.quantity * self.price

    # here the discount are token from the class level if not defined as entry
    def apply_discount (self):
        self.price = self.price * self.pay_rate

    # here the function fo the visualisation of the all data
    def __repr__(self) -> str:
        return f"Item ('{self.electronic}', {self.price}, {self.quantity})"

# now we need to have 20% discount for laptop and 30% for phone
# pour item 1 le discount est pris en charge directement de class level pay_rate
item1= item ("phone", 200, 12)
item1.apply_discount()
print(item1.price)
# pour item 2 j'ai forcer la definition de par_rate pour discount de 30%
item2= item ("laptop", 600, 20)
item2.pay_rate=0.7
item2.apply_discount()
print(item2.price)
# pour voir tout les item
print(item.list)
```

Use of csv as data base with a class method

```python
import csv
class item:
    pay_rate = 0.8 # 20% discount defined in the class level
    list = [] # the list of all data
    def __init__(self, electronic: str, price: float, quantity: int):

        # run validation to received argument
        assert price >= 0, f"Price {price} is not greater or equal than 0!"
        assert quantity >= 0, f"Quantity {quantity} is not equal or greater than zero!"

        # les variable a definir pour l'objet
        self.electronic = electronic
        self.price = price
        self.quantity = quantity
        # ici on va ajouter les data a la list
        self.list.append(self)

    # methode to calculate the total of price quantity
    def calculate_total_price (self):
        return self.quantity * self.price

    # here the discount are token from the class level if not defined as entry
    def apply_discount (self):
        self.price = self.price * self.pay_rate
    # to have the data in csv file and use it
    @classmethod
    def instanciate_from_csv(classmethod):
        # to open the file
        with open("table.csv", 'r') as f:
            # convert to python dictionnary
            reader = csv.DictReader (f)
            items = list(reader)
        # visualising the dictionnary
        for i in items:
            print(i)
    # here the function fo the visualisation of the all data
    def __repr__(self) -> str:
        return f"Item ('{self.electronic}', {self.price}, {self.quantity})"

item.instanciate_from_csv()
```

```
{'name': 'Phone', 'price': '100', 'quantity': '1'}
{'name': 'laptop', 'price': '1000', 'quantity': '3'}
{'name': 'cable', 'price': '10', 'quantity': '2'}
{'name': 'Mouse', 'price': '50', 'quantity': '5'}
{'name': 'keyboard', 'price': '75', 'quantity': '5'}
```

How to initiate instances in class method

```python
    @classmethod
    def instanciate_from_csv(classmethod):
        # to open the file
        with open("table.csv", 'r') as f:
            # convert to python dictionnary
            reader = csv.DictReader (f)
            items = list(reader)
        # initiate instances
        for i in items:
            item(
                electronic=i.get('electronic'),
                price=float(i.get('price')),
                quantity=int(i.get('quantity')),
            )
    # here the function fo the visualisation of the all data
    def __repr__(self) -> str:
        return f"Item ('{self.electronic}', {self.price}, {self.quantity})"

item.instanciate_from_csv()
print(item.list)
```

```
[Item ('Phone', 100.0, 1), Item ('laptop', 1000.0, 3), Item ('cable', 10.0, 2), Item ('Mouse', 50.0, 5), Item ('keyboard', 75.0, 5)]
```

Use a static method (ne pend pas un objet, mais une valeur donner)

```python
    @staticmethod
    def is_integer(num):
        # i will check if the entry is a integer or not for i.e: 5.0, 10.0$
        if isinstance(num, float):
            # count out if the floats that are point zero
            return num.is_integer()
        elif isinstance(num, int):
            return True
        else:
            return True
    # to have the data in csv file and use it
    @classmethod
    def instanciate_from_csv(classmethod):
        # to open the file
        with open("table.csv", 'r') as f:
            # convert to python dictionnary
            reader = csv.DictReader (f)
            items = list(reader)
        # initiate instances
        for i in items:
            item(
                electronic=i.get('electronic'),
                price=float(i.get('price')),
                quantity=int(i.get('quantity')),
            )
    # here the function fo the visualisation of the all data
    def __repr__(self) -> str:
        return f"Item ('{self.electronic}', {self.price}, {self.quantity})"

item.instanciate_from_csv()
print (item.is_integer(5.0))
```

- Difference: in class methode there are a mandatory parameter but in staticmethod the parameter is a regular parameter not mandatory

```python
@classmethod
def instantiate_from_yaml_file(cls):
    '''
    This should also do something that has a relationship
    with the class, but usually, those are used to
    manipulate different structures of data to instantiate
    objects, like we have done with CSV.
    '''
```

```python
@staticmethod
def is_integer():
    '''
    This should do something that has a relationship
    with the class, but not something that must be unique
    per instance!
    '''
```

Benefit of inherence

```python
# we will look for the brokens phones that's why we will implement inherence of the class item into the child class phone
class Phone(item):
    list=[]
    # to have access to all attribute of the class parent we need to use super
    def __init__(self, electronic: str, price: float, quantity: int, brokens=0):
        super().__init__(electronic, price, quantity)
        assert brokens>=0, f'the {brokens} is not greater or equal than 0!'
        # assign to self object
        self.brokens=brokens
        # actions to execute
        Phone.list.append(self)

phone1 = Phone('sumsung', 500, 5, 1)
print (phone1.calculate_total_price())
```

To know which class did the job

In the child class no need to use list

```python
    def __repr__(self) -> str:
        # in order to know which class the super or the child we will use {self.__class__.__name__}
        return f"{self.__class__.__name__} ('{self.electronic}', {self.price}, {self.quantity})"
# we will look for the brokens phones that's why we will implement inherence of the class item into the child class phone
class Phone(item):
    # to have access to all attribute of the class parent we need to use super
    def __init__(self, electronic: str, price: float, quantity: int, brokens=0):
        super().__init__(electronic, price, quantity)
        assert brokens>=0, f'the {brokens} is not greater or equal than 0!'
        # assign to self object
        self.brokens=brokens

phone1 = Phone('sumsung', 500, 5, 1)
# the list is accessible from the child class because we did super init and in this one we have self.list.append(self)
print(Phone.list)
```

Use of property decorator: for read only the user cannot modify

```python
from item import item
from phone import Phone

item1 = item("iphonex", 750, 12)
print(item1.read_only_name)
```

```python
import csv

class item:
    pay_rate = 0.8 # 20% discount defined in the class level
    list = [] # the list of all data
    def __init__(self, electronic: str, price: float, quantity: int):...

    # methode to calculate the total of price quantity
    def calculate_total_price (self):...

    # here the discount are token from the class level if not defined as entry
    def apply_discount (self):...
    # use of static method
    @staticmethod
    def is_integer(num):...
    # to have the data in csv file and use it
    @classmethod
    def instanciate_from_csv(classmethod):...
    # here the function fo the visualisation of the all data
    def __repr__(self) -> str:...
    @property
    def read_only_name(self):
        return "AAA"
```

Encapsulation: restricting direct access to some attribute

```python
from item import Item

item1 = Item("MyItem", 750)

# Setting an Attribute
item1.name = "OtherItem"

# Getting an Attribute
print(item1.name)
```

```python
        self.__name = name
        self.price = price
        self.quantity = quantity

        # Actions to execute
        Item.all.append(self)

    @property
    # Property Decorator = Read-Only Attribute
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        if len(value) > 10:
            raise Exception("The name is too long!")
        else:
            self.__name = value

    def calculate_total_price(self):
        return self.price * self.quantity
```

Use of property decorator: for read only the user cannot modify and USE NAME.SETTER decorator to have the possibility to modify

abstraction: private methode that is not calleble from outsite of the instance

```python
def __connect(self, smpt_server):
    pass
def __prepare_body(self):
    return f"""
    Hello Someone.
    We have {self.name} {self.quantity} times.
    Regards, JimShapedCoding
    """

def __send(self):
    pass
def send_email(self):
    self.__connect()
    self.__prepare_body()
    self.__send()
```

polymorphism: to have different scenario as result from apply a specific function Like the len() function

```python
from phone import Phone
from keyboard import Keyboard

item1 = Keyboard("jscKeyboard", 1000, 3)

item1.apply_discount()

print(item1.price)
```

Knows how to handle diffrent kind of object that it receive in argument and return the result accordenly:
If we give list: amount of element in list
If we give string we will receive the entire caractere in the word already give it.

```python
name = "Jim"
print(len(name))

some_list = ["some","name"]
print(len(some_list))
# That's polymorphism in action, a single function does now
# how to handle different kinds of objects as expected!
```