ROYAUME DU MAROC

مكتب التكوين المهني وإنعاش الشغل

Office de la Formation Professionnelle et de la Promotion du Travail

OFPPT

INSTITUT SPECIALISE DE TECHNOLOGIE APPLIQUEE NTIC2 SIDI MAAROUF CASABLANCA

RAPPORT PROGRAMMATIN ORIENTE OBJET

GROUPE 2

REALISE PAR :

AMINA NASSIR

OUMAIMA MAGHFOUR

OTHMANE LAKRICHI

KENZA HRAGUE

SOMMAIRE

VARIANT1

1) QUESTION DE COURS

Enoncé

1. Quelle est la différence entre une interface et une classe abstraite en programmation orientée objet ?

2. Quelle est la définition d'une classe abstraite en programmation orientée objet ?

Solution

Différence entre une interface et une classe abstraite en programmation orientée objet :

- **Interface** : Une interface en programmation orientée objet définit un contrat qui spécifie les méthodes qu'une classe doit implémenter. Une interface ne fournit pas d'implémentation de ces méthodes ; elle déclare simplement leur signature. Les classes peuvent implémenter plusieurs interfaces, permettant ainsi une forme de polymorphisme.

- **Classe abstraite** : Une classe abstraite est une classe qui ne peut pas être instanciée elle-même. Elle peut contenir à la fois des méthodes abstraites (qui n'ont pas

de corps et doivent être implémentées par les classes filles) et des méthodes concrètes (avec une implémentation). Les classes filles doivent fournir des implémentations pour toutes les méthodes abstraites de la classe abstraite, sinon elles doivent aussi être déclarées comme abstraites.

2. Définition d'une classe abstraite en programmation orientée objet :

- **Une classe abstraite** est une classe qui ne peut être instanciée elle-même et qui est souvent utilisée comme modèle pour d'autres classes. Elle peut contenir des méthodes avec des implémentations concrètes ainsi que des méthodes abstraites, c'est-à-dire des méthodes déclarées mais sans implémentation. Les classes filles doivent fournir des implémentations pour toutes les méthodes abstraites, sinon elles doivent également être déclarées comme abstraites. Les classes abstraites sont souvent utilisées pour partager du code commun entre les classes filles et pour définir un ensemble de méthodes que les classes filles doivent implémenter.

2) ETUDE DE CAS :

PARTIE N° 1 :

Enoncé

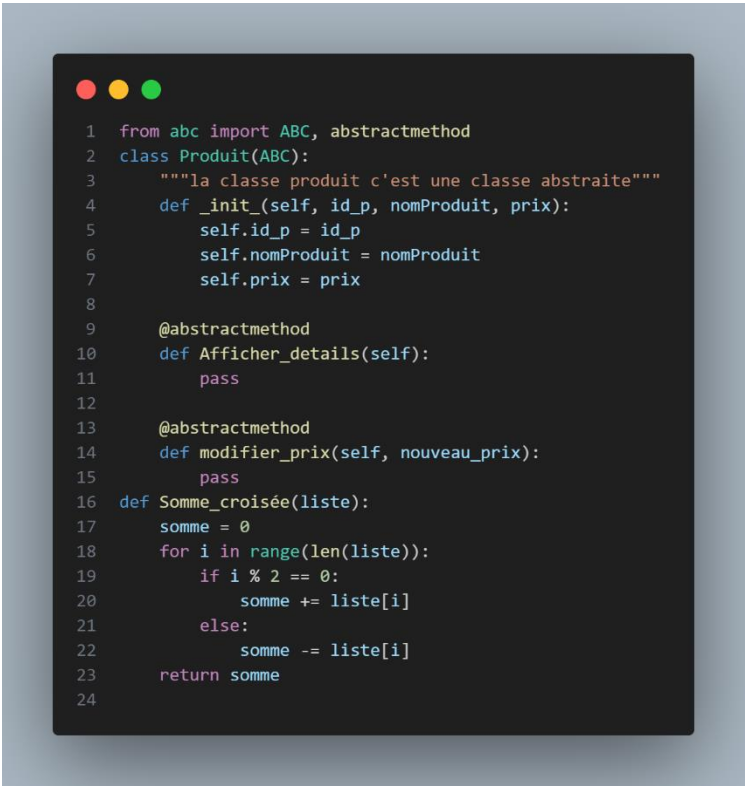
1. Définissez une classe abstraite **Produit** avec les attributs **id_p**, **nomProduit**, et **prix**, incluant deux méthodes Implémentez abstraites **Afficher_ détails ()** et **modifier_ prix (),** et ajoutez un constructeur pour initialiser les attributs.

2. une méthode statique Somme croise qui prend en arguments une liste, et qui renvoie la somme croisée des éléments de cette liste.

Exemple : liste = [4,7,0,2,3,10,15]

Somme croisée = 4-7+0-2+3-10+15=3

Solution



```python
from abc import ABC, abstractmethod
class Produit(ABC):
    """la classe produit c'est une classe abstraite"""
    def _init_(self, id_p, nomProduit, prix):
        self.id_p = id_p
        self.nomProduit = nomProduit
        self.prix = prix

    @abstractmethod
    def Afficher_details(self):
        pass

    @abstractmethod
    def modifier_prix(self, nouveau_prix):
        pass
def Somme_croisée(liste):
    somme = 0
    for i in range(len(liste)):
        if i % 2 == 0:
            somme += liste[i]
        else:
            somme -= liste[i]
    return somme

```

PARTIE N° 2 :

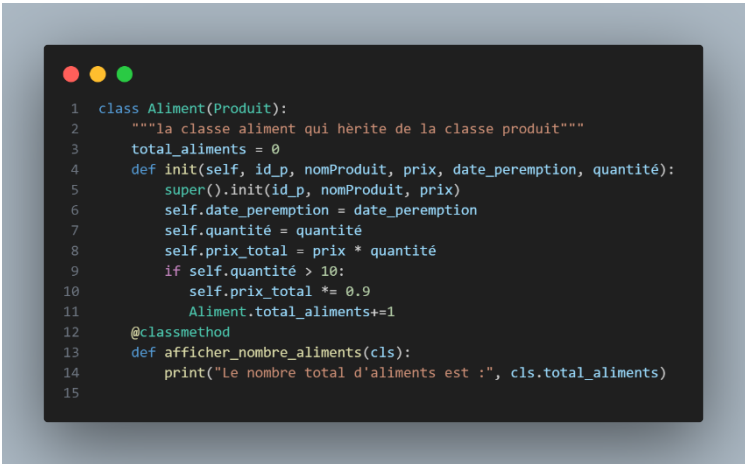
Enoncé

1. une classe dérivée **Aliment** qui n'est pas abstraite et qui hérite de la classe Produit avec les attributs **date_ péremption**, **quantité**, et **prix _ total** qui bénéficie d'une réduction de 10% lorsque la quantité d'aliments dépasse

10, et ajoutez un constructeur pour initialiser les attributs sans passer **prix_ total** en argument.

2. Implémentez une méthode de classe pour afficher le nombre total des aliments

Solution



```python
class Aliment(Produit):
    """la classe aliment qui hèrite de la classe produit"""
    total_aliments = 0
    def init(self, id_p, nomProduit, prix, date_peremption, quantité):
        super().init(id_p, nomProduit, prix)
        self.date_peremption = date_peremption
        self.quantité = quantité
        self.prix_total = prix * quantité
        if self.quantité > 10:
            self.prix_total *= 0.9
            Aliment.total_aliments+=1
    @classmethod
    def afficher_nombre_aliments(cls):
        print("Le nombre total d'aliments est :", cls.total_aliments)

```

PARTIE N° 3 :

Enoncé

1. Créez une nouvelle classe **panier** avec des attributs **id_ panier**, **aliments** (liste d'instances d'aliment), et **prix_ total_ panier** qui subit une argumentation de 17% de TVA, et ajoutez un

constructeur pour initialisez les attributs sans passer prix_ total_ un panier en argument.

2. Des Implémentez méthodes **getters** et **setters** pour accéder et modifier l'attribut Prix_ total_ panier de manière sécurisée.
3. Modifiez la classe panier dans le but d'afficher la moyenne des prix total de deux paniers. Par exemple : {panier P1, panier P2 = print (P1 + P2), afficher la moyenne des prix total de deux paniers P1 et P2}.
4. Implémentez une méthode de classe **chère_ panier ()** qui afficher les détails du panier qui a le plus grand prix_ total_ panier.

Solution

```python
#partie n°3
class Panier:
    def __init__(self, id_panier):
        self.id_panier = id_panier
        self.aliments = []
        self.prix_total_panier = 0
        Panier.panier.append(self)

    def ajouter_aliment(self, aliment):
        self.aliments.append(aliment)
        self.__prix_total_panier += aliment.prix_total

    def calculer_prix_total_panier(self):
        self.prix_total_panier *= 1.17

    def get_prix_total_panier(self):
       return self.__prix_total_panier
    def set_prix_total_panier(self,nv_prix):
        self.__prix_total_panier = nv_prix
    def add(self, p):
        if isinstance(p, Panier):
            total_prix = self.prix_total_panier + p.prix_total_panier
            moyenne_prix = total_prix / 2
            return moyenne_prix
    def ajouter_aliment(self, aliment):
        self.aliments.append(aliment)
        self.prix_total_panier += aliment.prix_total

    @classmethod
    def chere_panier(cls):
        panier_cher = max(cls.paniers, key=lambda p: p.prix_total_panier)
        print("Le panier le plus cher est le panier {}.".format(panier_cher.id_panier))
        print("Détails du panier :")
        for aliment in panier_cher.aliments:
            print(aliment.nom, "- Prix total :", aliment.prix_total)

```

VARIANT2

1) QUESTION DE COURS

Enoncé

1 Citez deux caractéristiques d'une classe abstraite en programmation orientée objet ?

2 Citez deux caractéristiques d'une méthode abstraite en programmation orientée objet ?

Solution

1. Deux caractéristiques d'une classe abstraite en programmation orientée objet :

- Une classe abstraite nécessite des sous-classes qui fournissent des implémentations pour les méthodes abstraites sinon ces sous-classes sont déclarées abstraites
- Une classe abstraite hérite de la classe abstraite de base– **ABC**

2. deux caractéristiques d'une méthode abstraite en programmation orientée objet

- Pour définir une méthode abstraite dans la classe abstraite, on utilise un décorateur @abstractmethod
- Une méthode abstraite est une méthode qui ne contient pas de corps. Elle possède simplement une signature de définition (pas de bloc d'instructions)

1) ETUDE DE CAS :

PARTIE N° 1 :

Enoncé

1. Définissez une classe abstraite **Produit** avec les attributs **id_ produit**, **nom**, et **prix**, incluant deux méthodes abstraites **Afficher_ détails ()** et **Appliquer_ réduction (),** et ajoutez un constructeur pour initialiser les attributs.

2. Implémentez une méthode statique **Somme_ inverse** qui prend en arguments une liste, et qui renvoie la somme inverse des éléments de cette liste.

Exemple : liste [4,7,5,2,3,10,17]

Somme inverse = -4+7-5+2-3+10-17 =-10

Solution

```python
#etude de cas
#partie n° 1
from abc import ABC, abstractmethod
class Produit(ABC):
    """ la class produit c'est un classe abstarite"""
    def __init__(self, id_produit, nom, prix):
        self.id_produit = id_produit
        self.nom = nom
        self.prix = prix

    @abstractmethod
    def afficher_details(self):
        pass

    @abstractmethod
    def appliquer_reduction(self):
        pass

@staticmethod
def somme_inverse(liste):
        somme_inverse = 0
        for index, element in enumerate(liste):
            if index % 2 == 0:
                somme_inverse -= element
            else:
                somme_inverse += element
                return somme_inverse
#partie n°2

```

PARTIE N° 2 :

Enoncé

1.Créez dérivée une classe **Machine** qui n'est pas abstraite et qui hérite de la classe **Produit** avec les attributs **marque**, **quantité_ machine**, et **total_ prix** qui bénéficie d'une réduction de lorsque la quantité des machines dépasse 3, et ajoutez un constructeur pour initialiser

Les attributs sans passer total prix en argument.

2. Implémentez une méthode de classe pour afficher le nombre total des machines.

Solution

```python
#partie n°2
class Machine(Produit):
    total_machines = 0

    def __init__(self, id_produit, nom, prix, marque, quantite_machine):
        super().__init__(id_produit, nom, prix)
        self.marque = marque
        self.quantite_machine = quantite_machine
        Machine.total_machines += 1
        self.total_prix = self.calculer_prix_total()

    def calculer_prix_total(self):
        if self.quantite_machine > 3:
            reduction = 0.07 * self.prix * self.quantite_machine
            self.appliquer_reduction(reduction)
        return self.prix * self.quantite_machine

    def afficher_details(self):
        pass

    def appliquer_reduction(self):
        pass


    @classmethod
    def afficher_nombre_total_machines(cls):
        print("Nombre total de machines:", cls.total_machines)

```

PARTIE N° 3 :

Enoncé

1. une nouvelle classe **commande** avec des attributs Créez **id_ commande**, **machines** (une liste d'instances de machine), et **prix_ total_ commande** qui subit une argumentation de 20 % de TVA, et ajoutez un

constructeur pour initialisez les attributs sans passer prix_ total_ panier en argument.

2. Implémentez des méthodes **getters** et **setters** pour accéder et modifier l'attribut
   Prix_ total_ commande de manière sécurisée.
3. Modifiez la classe panier dans le but d'afficher la moyenne des prix total de deux paniers.
   Par exemple : {commande C1, commande C2 = print (C1 + C2), afficher la moyenne des prix total de deux paniers C1 et C2}.
4. Implémentez une méthode de classe **chère_ commande ()** qui afficher les détails du panier qui a le plus grand prix_ total_ commande

Solution

```python
#partie n°3
class Commande:
    commandes = []
    def __init__(self, id_commande, date, machines):
        self.id_commande = id_commande
        self.date = date
        self.machines = machines
        self._prix_total_commande = self.calculer_prix_total_commande()
        Commande.commandes.append(self)



    def calculer_prix_total_commande(self):
        prix_total = sum(machine.total_prix for machine in self.machines)
        prix_total_tva = prix_total * 1.20
        return prix_total_tva

    def get_prix_total_commande(self):
        return self._prix_total_commande

    def set_prix_total_commande(self, nouveau_prix):
        raise ValueError("Impossible de modifier le prix total de la commande.")

    def add(self, other):
        if isinstance(other, Commande):
            total_prix = self._prix_total_commande + other._prix_total_commande
            return total_prix

    @classmethod
    def chere_commande(cls):
        if not cls.commandes:
            print("Aucune commande n'a été enregistrée.")
            return
        chere = max(cls.commandes, key=lambda commande: commande._prix_total_commande)
        print("Détails de la commande la plus chère:")
        print(f"ID: {chere.id_commande}")
        print(f"Date: {chere.date}")
        print("Machines:")
        for machine in chere.machines:
            machine.afficher_details()
        print(f"Prix total avec TVA: {chere._prix_total_commande:.2f}")

```

PACKAGE

Codage d'une classe

Apport du langage Python

Ajout d'un attribut d'instance

Il est possible d'ajouter un attribut uniquement pour une instance donnée via la syntaxe suivante :

Instance. Nouvel _ attribut = valeur

L'attribut spécial __dict__

Cet attribut spécial donne les valeurs des attributs de l'instance

L'attribut spécial **__doc__** affiche la documentation de la classe

La fonction **dir** donne un aperçu des méthodes de l'objet

Utilisation de @property

@property est un décorateur qui évite d'utiliser la fonction getter explicite

@attribut. Setter est un décorateur qui évite d'utiliser la fonction setter explicite

```python
from src.variant1 import *
from src.variant2 import *
liste = [3,4,5,6,78,7]
res = Somme_croisée(liste)
print(res)
aliment1 = Aliment(1, "Pomme", 1.5, "2024-04-10", 12)
aliment2 = Aliment(2, "Banane", 2, "2024-04-08", 8)
print(aliment1.afficher_details())
print(aliment1.afficher_nombre_aliment())
aliment1.color = "rouge"
print(aliment1.__dict__)
print(aliment1.__doc__)
print(dir(aliment1))
#del aliment1
#print(aliment1)
panier1 = Panier(1, [aliment1, aliment2])
panier2 = Panier(2, [aliment2])
panier1.prix_total_panier = 5
print(panier1.prix_total_panier)
panier2.prix_total_panier = 8
print(panier2.prix_total_panier)
p3 =(panier1+panier2)
print(p3)
print(Panier.cher_panier(paniers=[panier1,panier2]))
machine1 = Machine(1, "Machine 1", 1000, "Marque A", 5)
machine2 = Machine(2, "Machine 2", 1500, "Marque B", 2)
print(machine1.afficher_details())
machine1.module ="hp"
print(machine1.__dict__)
print(machine1.__doc__)
print(dir(machine1))
print(machine1.afficher_nombre_total_machines())
#del machine1
#print(machine1)
Commande1 = Commande(101, [machine1, machine2])
Commande2 = Commande(102, [machine3])
Commande1.prix_total_commande = 399
print(Commande1.prix_total_commande)
Commande2.prix_total_commande = 200
print(Commande2.prix_total_commande)
c3 = Commande1 + Commande2
print(c3)
Commande.chere_commande(Commande1)


```

➢ **Formation par :**

Mr. ÔTHMAN HAQQAY