DÉCOUVREZ
LES FORMATIONS
DE L'OFPPT

OFPPT

FILIÈRES
CONDITIONS D'ACCÈS
PROGRAMME
DIPLÔMES

Rapport De contrôle {v1, v2}

Module : programmation orienté objet

❖ ***Realise par :***

- ✓ *Ilham barakat*
- ✓ *Hatim harchane*
- ✓ *Adam el mosaoui*
- ✓ *Assiya hamilou*

Annee scolaire ***2023 2024***

Formateur *othmane haqay*

Filiere *devloppement digital 104*

Centre de formation *ntic 2 sidi maarouf*

OFPPT

PLAN :

INTRODUCTION :

- La programmation orientée objet est l'une des méthodologies récentes de programmation, couramment utilisée par les langages de programmation les plus répandus (python, JavaScript, C#.Net, ...). Cette méthodologie succède à la programmation impérative en lui ajoutant les notions d'objets et de classes.

L'orientation objet présente de nombreux avantages, sinon elle ne serait pas le paradigme dominant du développement logiciel moderne. L'un des principaux avantages est l'encapsulation de la logique et des données dans des classes individuelles. Cela améliore la maintenabilité et rend le code plus facile à étendre.

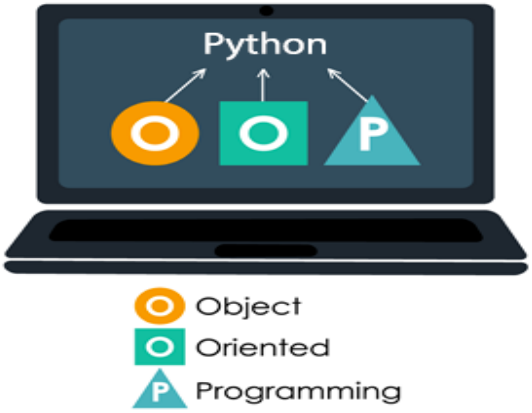
DÉFINITION D'UN PACKAGE :

❖ En programmation orientée objet, un package (aussi appelé module ou bibliothèque) est une unité d'organisation qui regroupe un ensemble de classes, d'interfaces, de fonctions ou d'autres éléments de code connexes. Les packages sont utilisés pour organiser et structurer le code source d'une application de manière logique et hiérarchique

❖ Voici quelques caractéristiques et fonctionnalités importantes des packages en programmation orientée objet :

- Encapsulation
- Organisation
  - Portée
- Réutilisation
- Gestion des dépendances
- Découpage fonctionnel

❖ En résumé, les packages sont des composants essentiels de la programmation orientée objet qui permettent d'organiser, de structurer et de gérer efficacement le code source d'une application. Ils favorisent la modularité, la réutilisabilité et la maintenance du code, tout en facilitant la gestion des dépendances et la collaboration entre les développeurs.



CONTRÔLE VERSION 2 :

Enonce :

Question de cours:

1-Citez deux caractéristiques d'une classe abstraites en poo

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

Correction :

1. Caractéristiques d'une classe abstraite en POO :

a. **Ne peut pas être instanciée directement** : Une classe abstraite ne peut pas être utilisée pour créer des objets directement. Elle sert de modèle pour les sous-classes et fournit une structure commune pour ces sous-classes.

b. **Peut contenir des méthodes concrètes** : En plus de déclarer des méthodes abstraites, une classe abstraite peut également contenir des méthodes concrètes avec une implémentation par défaut. Ces méthodes peuvent être héritées et utilisées par les sous-classes.

2. Caractéristiques d'une méthode abstraite en programmation orientée objet :

a. **Doit être redéfinie par les sous-classes** : Une méthode abstraite est déclarée dans une classe parente mais n'a pas d'implémentation concrète. Chaque sous-classe doit fournir sa propre implémentation de la méthode abstraite.

b. **Contribue à la polymorphie** : Les méthodes abstraites permettent de définir un comportement commun pour différentes classes, tout en laissant chaque classe spécifique définir sa propre logique. Cela favorise la polymorphie, où une même méthode peut être appelée sur des objets de différentes classes, produisant un comportement différent en fonction de la classe réelle de l'objet

Enonce :

Etude de cas:

Partie1:

1-Definissez une classe abstraite Produit avec les attributs id_p,nomProduits et prix,incluant deux méthodes abstraites Afficher_details() et Appliquer_réduction() et ajoutez un constructeur pour initialiser les attributs.

2-implimenez une méthode statique Somme_inverse qui prend en arguments une liste, et qui renvoi la somme inverse des éléments de cette liste.

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

Somme croisée= -4+7-5+2-3+10-17=-10

Correction :

```python
from abc import ABC,abstractmethod
class Produit(ABC):
    def __init__(self,id_produit,nom,prix):
        self.id_produit = id_produit
        self.nom = nom
        self.prix = prix
    @abstractmethod
    def Afficher_detail(self):
        pass
    @abstractmethod
    def appliquer_reduction(self):
        pass
    @staticmethod
    def somme_inverse(liste):
        somme = 0
        for i, nombre in enumerate(liste):
            if i % 2 == 0:
                somme -= nombre
            else:
                somme += nombre
        return somme
```

Enonce :

Partie2:

1-Créez une classe dérivée 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 7% lorsque la quantité des machines dépasse 3,et ajoutez un constructeur pour initialiser les attributs sans passer total_prix en argument

2-Implimentez une méthode de classe pour afficher le nombre total des machines

Correction :

```
class Machine(Produit):
    nbr_machine = 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
        self.total_prix = self.prix * self.quantite_machine
        self.total_prix = self.appliquer_reduction(7)
        Machine.nbr_machine += 1
    def Afficher_detail(self):
        return f"id_produit: {self.id_produit}\nnom: {self.nom}\nprix:
{self.prix}\nmarque:
        {self.marque}\nquantite_machine: {self.quantite_machine}\ntotal_prix:
{self.total_prix}"
    def appliquer_reduction(self, poucentaReduire):
        if self.quantite_machine > 3:
            self.total_prix = self.total_prix - (self.total_prix*(poucentaReduire/100))
        return self.total_prix
    @classmethod
    def nombreMachine(cls):
        return f"le nombre total des aliment est: {cls.nbr_machine}"
m = Machine(123, "sabone" ,8, "qsdfghu", 12)
print(f"le somme inverse: {Produit.somme_inverse([4, 7, 5, 2, 3, 10, 17])}")
```

Enonce :

Partie3:

1-Créez une nouvelle classe Commande avec les attributs id_commande,date,et Machines(une liste d'instance de Machine) et prix_total_commande en argument.

2-Implimentez des méthodes getters et setters pour accéder et modifier l'attribut prix_total_commande de maniére sécurisée

3-Modifier la classe Commande dans le but d'afficher la moyenne des prix total de deux commandes par exemple: {Commande C1,Commande C2 -->print(C1+C2),afficher la moyenne des prix total des deux commandes C1 et C2}

4-Implimentez une méthode de classe chère_Commande() qui affiche les détails dee la commande qui à le plus grand prix_total_commande

Correction :

```
class Commande:
    TVA = 0.20
    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()
    def calculer_prix_total_commande(self):
        prix_total = sum(machine.prix_machine for machine in self.machines)
        return prix_total * (1 + self.TVA)
    @property
    def prix_total_commande(self):
        return self._prix_total_commande
    @prix_total_commande.setter
    def prix_total_commande(self, new_price):
        raise AttributeError("Le prix total de la commande ne peut pas être modifié
directement.")
    @classmethod
    def moyenne_prix_total(cls, commande1, commande2):
        return (commande1.prix_total_commande + commande2.prix_total_commande) / 2
    @classmethod
    def chere_Commande(cls, *commandes):
        chere_commande = max(commandes, key=lambda x: x.prix_total_commande)
        print("Détails de la commande la plus chère :")
        print(f"ID Commande: {chere_commande.id_commande}")
        print(f"Date: {chere_commande.date}")
        print("Machines:")
        for machine in chere_commande.machines:
            print(f" - {machine.nom_machine}: {machine.prix_machine}")
        print(f"Prix total: {chere_commande.prix_total_commande}")
```

```
 # Méthode spéciale pour l'addition de deux commandes
    def _add_(self, other):
        if not isinstance(other, Commande):
            raise TypeError("L'addition ne peut être effectuée qu'avec une autre instance
de Commande.")
        return Commande(-1, "", [])  # Nous renvoyons une nouvelle commande avec des
valeurs vides pour l'exemple

# Exemple d'utilisation
    if_name_ == "_main_":
        # Création de quelques machines
        machine1 = Machine(1, "Machine 1", 1000)
        machine2 = Machine(2, "Machine 2", 1500)
        machine3 = Machine(3, "Machine 3", 2000)

        # Création de deux commandes
        commande1 = Commande(1, "2024-04-02", [machine1, machine2])
        commande2 = Commande(2, "2024-04-03", [machine2, machine3])

        # Affichage de la moyenne des prix totaux
        moyenne_prix = Commande.moyenne_prix_total(commande1, commande2)
        print(f"Moyenne des prix totaux des commandes 1 et 2: {moyenne_prix}")

        # Affichage de la commande la plus chère
        Commande.chere_Commande(commande1, commande2)
        # Addition de deux commandes (pour l'exemple)
        try:
            resultat_addition = commande1 + commande2
            print("Résultat de l'addition de commande 1 et commande 2 :",
resultat_addition)
        except TypeError as e:print(e)
```

CONTRÔLE VERSION 1 :

Enonce :

Question de cours:

1-Quelle est la différence entre une interfaces et une classe abstraite ?

2-Quelle est la définition d'une clase abstraite en programmation orientée objet ?

Correction :

1- **Interface** : Toutes les méthodes déclarées dans une interface sont implicitement abstraites et publiques, Une classe peut implémenter plusieurs interfaces, permettant ainsi d'hériter du comportement de plusieurs sources différentes.

Classe abstraite :

Une classe abstraite est une classe qui ne peut pas être instanciée directement Les méthodes abstraites dans une classe abstraite définissent un comportement qui doit être implémenté par les sous-classes

2-Les classes abstraites sont souvent utilisées pour définir des types génériques ou des modèles de base dans une hiérarchie de classe Les méthodes abstraites déclarées dans une classe abstraite définissent un comportement que les sous-classes doivent implémenter.

Enonce :

Etude de cas:

Partie1:

1-Definissez une classe abstraite Produit avec les attributs id_p,nomProduits et prix, incluant deux méthodes abstraites Afficher_details() et modifier_prix() et ajoutez un constructeur pour initialiser les attributs.

2-implimenez une méthode statique Somme_croisé qui prend en arguments une liste, et qui renvoi 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

Correction :

```python
from abc import ABC, abstractclassmethod
class Produit(ABC):
    def __init__(self, id_p, nomProduit, prix):
        self.id_p = id_p
        self.nomProduit = nomProduit
        self.prix = prix
    @abstractclassmethod
    def Aficher_detail(self):
        pass
    @abstractclassmethod
    def modifier_prix(self):
        pass
    @staticmethod
    def somme_croisee(liste):
        somme=0
        for i in range(len(liste)):
            if i%2 ==0:
                somme+=liste[i]
            else:
                somme -=liste[i]
        return somme
l = [4, 7, 0, 2, 3, 10, 15]
print(f"la somme croise est: {Produit.somme_croisee(l)}")
```

Enonce :

Partie2:

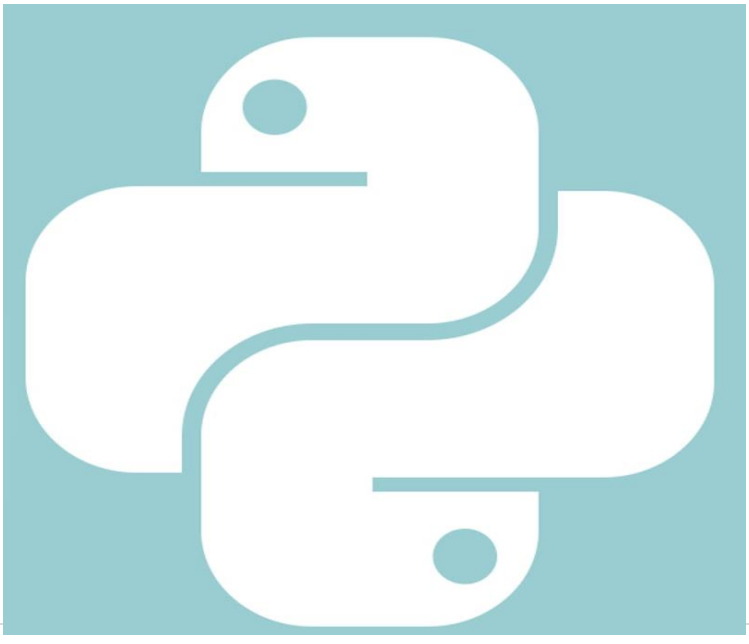
1-Créez une classe dérivée Aliment qui n'est pas abstraite et qui hérite de la classe Produit avec les attributs date_peremption,quantité et prix_total qui bénéficie d'une réduction de 10% lorsque la quantité d'aliments dépasse 10, et ajoutez un constructeur pour initialiser les attributs sans passer prix_total en argument

2-Implimentez une méthode de classe pour afficher le nombre total des aliments

Correction :

```
class Aliment(Produit):
    nbr_aliment = 0
    def __init__(self, id_p, nomProduit, prix, date_peremption, quantite):
        super().__init__(id_p, nomProduit, prix)
        self.date_peremption = date_peremption
        self.quantite = quantite
        self.prix_total = self.prix * self.quantite
        if self.quantite > 10:
            self.prix_total = self.prix_total - (self.prix_total*(10/100))
        Aliment.nbr_aliment += 1
    def Aficher_detail(self):
        return f"id_p: {self.id_p}\nnomProduit: {self.nomProduit}\nprix:
{self.prix}\ndate_peremption: {self.date_peremption}\nquantite:
{self.quantite}\nprixtotal: {self.prixtotal}"

    def modifier_prix(self, nouveauPrix):
        self.prix = nouveauPrix
    @classmethod
    def nombreAliment(cls):
        return f"le nombre total des aliment est: {cls.nbr_aliment}"
```



Enonce :

Partie3:

1-Créez une nouvelle classe Panier avec les attributs id_panier,Aliments(une liste d'instance d'Aliment) et prix_total_panier en argument.

2-Implimentez des méthodes getters et setters pour accéder et modifier l'attribut prix_total_panier de maniére sécurisée

3-Modifier la classe Panier dans le but d'afficher la moyenne des prix total de deux panier par exemple: {Panier P1,Panier P2 -->print(P1+P2),afficher la moyenne des prix total des deux panier P1 et P2}

4-Implimentez une méthode de classe chère_Panier() qui affiche les détails du panier qui à le plus grand prix_total_panier

Correction :

```
class panier:
    mell_p_t=0
    mell_panier=0
    mell_aliment=0
    nbr=0
    def _init_(self,id_panier,aliments):
        self.id_panier=id_panier
        self.aliments=aliments if not None else []
        self.prix_total_panier=0
        ptv=0
        panier.nbre+=1
        for x in aliments:
          ptv +=x.prix_total
        self.p_t=ptv
        if ptv > panier.mell_p_t:
             panier.mell_p_t=ptv
             panier.mell_aliment=aliments
             panier.mell_panier=id_panier
    def calculer_p_t(self):
        p_t=0
        for aliment in self.aliments:
            p_t+=aliment.prix_total
            self.prix_total_panier=p_t*(17/100)
    def get_prix_total_panier(self):
        return self.__prix_total_panier()
    def set_prix_total_panier(self,nv):
        self.__prix_total_panier=nv
    def _add_(self,other):
        return (self.prix_total_panier+other.prix_total_panier)/2
    @classmethod
    def chere_panier(cls):
     print ("le mielleur panier est:",{cls.mell_panier})
     cls.mell_aliment.afficher_detail()
     for aliment in cls.mell_aliment:
        aliment.afficher_detail()
```