Atelier_4_corr

N'oubliez pas d'utiliser des commentaires pour expliquer le fonctionnement de chaque partie de votre code.

Activité 1

Vous devez créer une classe Rectangle qui représente un rectangle et qui offre les fonctionnalités suivantes :

Un constructeur qui prend en entrée la longueur et la largeur du rectangle et les initialise en tant qu'attributs de la classe.

Une méthode calculer_perimetre() qui calcule et renvoie le périmètre du rectangle.

Une méthode calculer_aire() qui calcule et renvoie l'aire du rectangle.

Une méthode est_carre() qui vérifie si le rectangle est un carré en comparant la longueur et la largeur, et renvoie True si c'est le cas, sinon renvoie False.

Solution :

```
class Rectangle:
    def __init__(self, longueur, largeur):
        # Initialisation des attributs de la classe
        self.longueur = longueur
        self.largeur = largeur

    def calculer_perimetre(self):
        # Calcul et renvoi du périmètre du rectangle
        return 2 * (self.longueur + self.largeur)

    def calculer_aire(self):
        # Calcul et renvoi de l'aire du rectangle
        return self.longueur * self.largeur

    def est_carre(self):
        # Vérification si le rectangle est un carré
        return self.longueur == self.largeur
```

Activité 2

Définir une classe Point contenant deux attributs x et y.

Définir 2 instances de la classePoint, c'est-à-dire 2 objets de typePoint. Pour chacun d'eux, les attributs prennent des valeurs qui sont propres à l'instance.

Cette classe possède une méthode : deplace() qui prend deux paramètres dx et dy

x=x+dx

y=y+dy

Indiquer les arguments entre parenthèse. Le premier argument d'une méthode doit être self

Solution :

```
class Point:

    def __init__(self, x, y):

        # Initialisation des attributs de la classe

        self.x = x

        self.y = y


    def deplace(self, dx, dy):

        # Déplacement du point

        self.x += dx

        self.y += dy
```

Activité 3 :

Gestion d'une bibliothèque

Imaginez que vous travaillez sur un programme de gestion pour une bibliothèque. Vous devez créer des classes pour représenter les livres, les membres de la bibliothèque et la bibliothèque elle-même.

Créez une classe Livre avec les attributs suivants :

Titre

Auteur

Année de publication

ISBN (numéro d'identification unique)

Créez une classe Membre pour représenter les membres de la bibliothèque avec les attributs suivants :

Nom

Numéro de membre

Livres empruntés (une liste)

Créez une classe Bibliotheque avec les méthodes suivantes :

Ajouter un livre à la bibliothèque.

Retirer un livre de la bibliothèque lorsqu'il est emprunté.

Emprunter un livre à un membre (ajouter le livre à la liste des livres empruntés par le membre).

Retourner un livre (retirer le livre de la liste des livres empruntés par le membre et le remettre à la bibliothèque).

Assurez-vous que les classes interagissent correctement entre elles. Vous pouvez également ajouter des fonctionnalités supplémentaires si vous le souhaitez.

Solution :

```
class Livre:
    def __init__(self, titre, auteur, annee_publication, isbn):
        self.titre = titre
        self.auteur = auteur
        self.annee_publication = annee_publication
        self.isbn = isbn


class Membre:
    def __init__(self, nom, numero_membre):
```

```
        self.nom = nom
        self.numero_membre = numero_membre
        self.livres_empruntes = []

class Bibliotheque:
    def __init__(self):
        self.livres = []

    def ajouter_livre(self, livre):
        self.livres.append(livre)

    def retirer_livre(self, livre):
        self.livres.remove(livre)

    def emprunter_livre(self, membre, livre):
        membre.livres_empruntes.append(livre)
        self.retirer_livre(livre)
 def retourner_livre(self, membre, livre):
     membre.livres_empruntes.remove(livre)
     self.ajouter_livre(livre)
```

Activité 4 (Encapsulation)

Vous devez créer une classe CompteBancaire qui permettra de gérer les opérations de base d'un compte bancaire, telles que le dépôt, le retrait et la consultation du solde.

Définir la classe CompteBancaire avec les attributs suivants :

titulaire: le nom du titulaire du compte.

solde: le solde actuel du compte (initialisé à 0 au moment de la création) qui est déclaré comme privé

Définir les méthodes suivantes dans la classe CompteBancaire :

deposer(self, montant): Cette méthode permettra de déposer un montant sur le compte.

retirer(self, montant): Cette méthode permettra de retirer un montant du compte, mais vous devez vérifier si le solde est suffisant avant de réaliser le retrait.

consulter_solde(self): Cette méthode permettra de consulter le solde du compte.

Créer une instance de la classe CompteBancaire en prenant le nom du titulaire en entrée.

Effectuer quelques opérations de dépôt, de retrait et de consultation du solde pour vérifier le bon fonctionnement de la classe.

Solution :

```
class CompteBancaire:
    def __init__(self, titulaire):
        self.titulaire = titulaire
        self.__solde = 0

    def deposer(self, montant):
        self.__solde += montant

    def retirer(self, montant):
        if montant <= self.__solde:
            self.__solde -= montant
        else:
            print("Solde insuffisant")

    def consulter_solde(self):
        return self.__solde
```

Activité 6 (attributs statiques)

Ecrire en python un système de suivi des employés d'une entreprise. Vous devez créer une classe Employe avec les attributs suivants :

nombre_employes (attribut de classe) : Un attribut statique qui garde une trace du nombre total d'employés dans l'entreprise.

augmentation_salaire (attribut de classe) : Un attribut statique qui représente le pourcentage d'augmentation de salaire accordé à tous les employés lorsqu'il est modifié.

__init__ : Un constructeur pour initialiser les attributs d'instance tels que le nom, le salaire de base de l'employé.

afficher_informations : Une méthode d'instance qui affiche les informations de l'employé, y compris le salaire après l'augmentation.

Solution :

```
class Employe:

  nombre_employes = 0

  augmentation_salaire = 1.04


  def __init__(self, nom, salaire_base):


    self.nom = nom

    self.salaire_base = salaire_base

    Employe.nombre_employes += 1


  def afficher_informations(self):

    salaire_apres_augmentation = self.salaire_base * Employe.augmentation_salaire

    print(f"Nom: {self.nom}, Salaire: {salaire_apres_augmentation}")
```

Activité 7 (surchage des opérateurs)

Créez une classe Calculatrice qui peut effectuer des opérations arithmétiques de base (addition, soustraction, multiplication, division) en surchargeant les opérateurs correspondants. De plus, ajoutez la possibilité d'effectuer des opérations avancées telles que la puissance et le calcul de la racine carrée.

Surchargez les opérateurs +, -, *, / pour effectuer les opérations arithmétiques de base entre deux instances de la classe Calculatrice.

Ajoutez des méthodes pour effectuer des opérations avancées :

puissance(x, y): calcule la puissance de x à la puissance y.

racine_carree(x): calcule la racine carrée de x.

Assurez-vous de gérer les cas où la division par zéro peut se produire.

Solution :

```
class Calculatrice:
  def __init__(self, valeur):
    self.valeur = valeur

  def __add__(self, autre):
    return Calculatrice(self.valeur + autre.valeur)

  def __sub__(self, autre):
    return Calculatrice(self.valeur - autre.valeur)

  def __mul__(self, autre):
    return Calculatrice(self.valeur * autre.valeur)

  def __truediv__(self, autre):
    if autre.valeur == 0:
        raise ValueError("Division par zéro impossible.")
      return Calculatrice(self.valeur / autre.valeur)

    @staticmethod
    def puissance(x, y):
      return x ** y

    @staticmethod
    def racine_carree(x):
      if x < 0:
        raise ValueError("Calcul de racine carrée d'un nombre négatif impossible.")
      return math.sqrt(x)

  # Utilisation de la classe Calculatrice
  calc1 = Calculatrice(10)
  calc2 = Calculatrice(5)
```

```
# Opérations de base
addition = calc1 + calc2
soustraction = calc1 - calc2
multiplication = calc1 * calc2
try:
    division = calc1 / calc2
except ValueError as e:
    print(e)

# Opérations avancées
puiss = Calculatrice.puissance(2, 3)
try:
    racine = Calculatrice.racine_carree(9)
except ValueError as e:
    print(e)
```

Activité 8 Héritage

Créez une classe de base appelée Employe avec les attributs suivants :

nom: le nom de l'employé

salaire: le salaire de base de l'employé

annee_embauche: l'année d'embauche de l'employé

Ajoutez une méthode calculer_salaire_annuel() à la classe Employe qui calcule le salaire annuel en multipliant le salaire de base par 12.

Créez deux classes dérivées de Employe :

Manager: avec un attribut supplémentaire bonus et une méthode calculer_salaire_annuel() qui surcharge la méthode de la classe de base pour inclure le bonus dans le calcul.

Developpeur: avec un attribut supplémentaire langage et une méthode afficher_infos() qui affiche les informations spécifiques aux développeurs.

Créez des instances des classes Manager et Developpeur, et appelez leurs méthodes respectives.

Solution :

```
class Employe:
    def __init__(self, nom, salaire, annee_embauche):
        self.nom = nom
        self.salaire = salaire
        self.annee_embauche = annee_embauche

    def calculer_salaire_annuel(self):
        return self.salaire * 12

class Manager(Employe):
    def __init__(self, nom, salaire, annee_embauche, bonus):
        super().__init__(nom, salaire, annee_embauche)
        self.bonus = bonus

    def calculer_salaire_annuel(self):
        return super().calculer_salaire_annuel() + self.bonus

class Developpeur(Employe):
    def __init__(self, nom, salaire, annee_embauche, langage):
        super().__init__(nom, salaire, annee_embauche)
        self.langage = langage

    def afficher_infos(self):
        print(f"Nom: {self.nom}, Langage: {self.langage}")

# Création d'instances et appel des méthodes
manager = Manager("Alice", 5000, 2015, 1000)
print(manager.calculer_salaire_annuel())

developpeur = Developpeur("Bob", 4000, 2018, "Python")
developpeur.afficher_infos()
```

Activité 9 (méthodes abstraites)

Créez une classe abstraite appelée Forme avec une méthode abstraite calculer_surface().

Ajoutez deux classes dérivées de Forme :

Cercle: avec un attribut rayon et une méthode calculer_surface() qui calcule la surface du cercle.

Rectangle: avec des attributs longueur et largeur, et une méthode calculer_surface() qui calcule la surface du rectangle.

Créez des instances des classes Cercle et Rectangle, et appelez leur méthode calculer_surface().

Essayez de créer une instance de la classe abstraite Forme pour voir ce qui se passe.

Solution :

```
from abc import ABC, abstractmethod
import math


class Forme(ABC):
    @abstractmethod
    def calculer_surface(self):
        pass


class Cercle(Forme):
    def __init__(self, rayon):
        self.rayon = rayon

    def calculer_surface(self):
        return math.pi * self.rayon ** 2


class Rectangle(Forme):
    def __init__(self, longueur, largeur):
        self.longueur = longueur
        self.largeur = largeur

    def calculer_surface(self):
        return self.longueur * self.largeur
```

```
# Création d'instances et appel des méthodes
cercle = Cercle(5)
print(cercle.calculer_surface())

rectangle = Rectangle(4, 6)
print(rectangle.calculer_surface())

# Tentative de création d'une instance de Forme
# Cela va échouer car Forme est une classe abstraite
try:
    forme = Forme()
except TypeError as e:
    print(e)
```

Bon travail