Devoir sur table

Durée de l'épreuve : 1h
L'usage de la calculatrice n'est pas autorisé.

Exercice 1 (2 points)

Faire correspondre chaque type de test à sa définition.

	Les tests
a	Unitaires
b	D'intégration
c	Fonctionnels (boîte noire)
d	Structurels (boîte blanche)
e	De performance
f	D'utilisabilité

	Permettent de valider
1	Plusieurs parties du programme ensemble
2	Par rapport à la spécification
3	Une partie du programme
4	L'ergonomie du programme
5	Par rapport à la taille des données
6	Par rapport à l'implémentation

Exercice 2 (2 points)

QCM sur la Programmation Orientée Objet. Une seule réponse possible.

Question 1 :

Quelle est la méthode spéciale utilisée pour initialiser un objet dans une classe Python ?

A. `__start__`
B. `__init__`
C. `__main__`
D. `__create__`

Question 2 :

En Python, comment appelle-t-on une méthode d'instance depuis un objet ?

A. `ClassName.method_name()`
B. `object.method_name()`
C. `self.method_name()`
D. `instance.method_name(self)`

Question 3 :

Dans une classe Python, que représente l'attribut `self` ?

A. Une copie de l'objet actuel
B. Un lien vers la classe mère
C. L'objet courant de la classe
D. Une variable temporaire

Question 4 :
Quelle est l'utilité des méthodes statiques dans une classe Python ?

A. Elles ne nécessitent pas de paramètre `self` et peuvent être appelées sans créer d'instance
B. Elles modifient les attributs de la classe
C. Elles permettent de surcharger les opérateurs
D. Elles sont automatiquement appelées à chaque instanciation

Exercice 3 (3 points)

Vous êtes chargé de créer une classe appelée **Rectangle** pour représenter des rectangles. Chaque rectangle est défini par sa **largeur** et sa **hauteur**. Votre objectif est de permettre la comparaison de rectangles entre eux en utilisant les opérateurs d'égalité, d'infériorité, de supériorité et de taille.

Implémentez la classe avec les surcharges demandées :

1. Implémentez la classe Rectangle avec :
    - Un constructeur (__init__)
    - Une méthode surface() qui retourne la surface du rectangle (largeur * hauteur)
2. Surchargez les opérateurs suivants :
    - **Égalité (==)** : Deux rectangles sont égaux si leurs surfaces sont identiques.
    - **Inférieur (<)** : Un rectangle est plus petit qu'un autre si sa surface est inférieure.
    - **Supérieur (>)** : Un rectangle est plus grand qu'un autre si sa surface est supérieure.
    - **Taille (len())** : La taille d'un rectangle correspond à sa surface.

Rappels :
Pour surcharger un opérateur, il faut redéfinir une méthode spéciale dans la classe :

- __eq__(self, other) pour ==
- __lt__(self, other) pour <
- __gt__(self, other) pour >
- __len__(self) pour len()

Le code suivant teste la classe :

```
r1 = Rectangle(4, 5)  # Surface : 20
r2 = Rectangle(3, 7)  # Surface : 21
r3 = Rectangle(2, 10) # Surface : 20

print(r1 == r2)  # False, car 20 != 21
print(r1 < r2)   # True, car 20 < 21
print(r3 > r2)   # False, car 20 > 21 est faux
print(len(r1))   # 20, la surface du rectangle r1
```

Exercice 4 (4 points)

Une classe pour gérer une classe d'élèves

Vous êtes chargé de créer une application pour gérer une classe d'élèves dans un lycée. Chaque élève possède un nom, un prénom et une moyenne générale. Une classe d'élèves est composée de plusieurs élèves, et l'objectif est de pouvoir effectuer certaines opérations comme ajouter un élève, afficher tous les élèves ou encore calculer la moyenne générale de la classe.

Compléter le code suivant :

```
class Eleve:
    def __init__(self, nom, prenom, moyenne):
        """Initialise un élève avec son nom, prénom et moyenne."""
        ...
        ...
        ...

    def __str__(self):
        """Affiche les informations de l'élève sous une forme lisible."""
        return f"{self.prenom}{self.nom} - Moyenne : {self.moyenne:.1f}"


class Classe:
    def __init__(self):
        """Initialise une classe vide (sans élèves)."""
        self.eleves = ...

    def ajouter_eleve(self, eleve):
        """Ajoute un élève à la liste des élèves."""
        if isinstance(eleve, Eleve):
            ...
        else:
            print("Erreur : vous devez ajouter un objet de type Eleve.")

    def afficher_classe(self):
        """Affiche tous les élèves de la classe."""
        if not self.eleves:
            print("La classe est vide.")
        else:
            for eleve in self.eleves:
                print(eleve)

    def moyenne_classe(self):
        """Calcule et retourne la moyenne générale de la classe."""
        if not self.eleves:
            return 0.0  # Aucun élève, donc moyenne de 0
        total_moyennes = ...
        for eleve in self.eleves:
            total_moyennes += ...
        return ... / len(...)
```

Exercice 5 (4 points)

Nous avons l'habitude de noter les expressions arithmétiques avec des parenthèses : (2 + 3) × 5.
Il existe une autre notation utilisée par certaines calculatrices, appelée notation postfixe, qui n'utilise pas de parenthèses. L'expression arithmétique précédente est alors obtenue en saisissant successivement 2, puis 3, puis l'opérateur +, puis 5, et enfin l'opérateur ×.

On modélise cette saisie par le tableau [2, 3, '+', 5, '*'].
Autre exemple, la notation postfixe de 3 × 2 + 5 est modélisée par le tableau : [3, 2, '*', 5, '+'].

D'une manière plus générale, la valeur associée à une expression arithmétique en notation postfixe est déterminée à l'aide d'une pile en parcourant l'expression arithmétique de gauche à droite de la façon suivante :

- si l'élément parcouru est un nombre, on le place au sommet de la pile ;
- si l'élément parcouru est un opérateur, on récupère les deux éléments situés au sommet de la pile et on leur applique l'opérateur. On place alors le résultat au sommet de la pile.
- à la fin du parcours, il reste alors un seul élément dans la pile qui est le résultat de l'expression arithmétique.

Dans le cadre de cet exercice, on se limitera aux opérations × et +.
Pour cet exercice, on dispose d'une classe Pile qui implémente les méthodes de base sur la structure de pile.
Compléter le script de la fonction eval_expression qui reçoit en paramètre une liste python représentant la notation postfixe d'une expression arithmétique et qui renvoie sa valeur associée.

```
class Pile:
    """Classe définissant une structure de pile."""
    def __init__(self):
        self.contenu = []

    def est_vide(self):
        """Renvoie un booléen indiquant si la pile est vide."""
        return self.contenu == []

    def empiler(self, v):
        """Place l'élément v au sommet de la pile"""
        self.contenu.append(v)

    def depiler(self):
        """
        Retire et renvoie l'élément placé au sommet de la pile,
        si la pile n'est pas vide. Produit une erreur sinon.
        """
        assert not self.est_vide()
        return self.contenu.pop()

def eval_expression(tab):
    p = Pile()
    for ... in tab:
        if element != '+' ... element != '*':
            p.empiler(...)
        else:
            if element == ...:
                resultat = ... + ...
            else:
                resultat = ...
            p.empiler(...)
    return ...
```

Exemples :

```
>>> eval_expression([2, 3, '+', 5, '*'])
25
>>> eval_expression([1, 2, '+', 3, '*'])
9
>>> eval_expression([1, 2, 3, '+', '*'])
5
```

Exercice 6 (1.5 points)

Question 1 :
Quelle opération n'est pas possible dans une pile (structure LIFO) ?
A. Ajouter un élément en haut de la pile
B. Retirer l'élément du haut de la pile
C. Accéder directement à l'élément du bas de la pile
D. Vérifier si la pile est vide

Question 2 :
Quelle est la principale différence entre une file et une pile ?
A. La file ne permet pas de supprimer d'élément
B. La file suit un ordre FIFO (First In First Out), alors que la pile suit un ordre LIFO (Last In First Out)
C. La pile est limitée en taille, mais pas la file
D. La file utilise des listes chaînées, et la pile des tableaux

Question 3 :
Dans une liste chaînée, que se passe-t-il lorsqu'on supprime un nœud situé au milieu de la liste ?
A. Tous les nœuds suivants sont décalés vers la gauche
B. Le pointeur du nœud précédent est mis à jour pour pointer vers le nœud suivant
C. Le nœud suivant devient la tête de la liste
D. La liste est recréée entièrement

Exercice 7 (3.5 points)

Exercice : Méthodes d'une liste chaînée
Consigne :
Complétez les méthodes suivantes pour manipuler une liste chaînée :

1. **taille** : Retourne le nombre de cellules dans la liste.
2. **chercher_element** : Retourne True si un élément donné est présent dans la liste, sinon False.
3. **ajouter_debut** : Ajoute une nouvelle cellule au début de la liste.

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

class ListeChainee:
  def __init__(self):
    self.tete = None

  def est_vide(self):
    return self.tete is None

  def taille(self):
    """Calcule la taille de la liste."""
    courant = self.tete
    compteur = 0
    while courant:
      compteur = ...
      courant = ...  # Avancer dans la liste
    return ... # Retourner la taille

  def chercher_element(self, valeur):
    """Vérifie si un élément est présent dans la liste."""
    courant = self.tete
    while courant:
      if ... == valeur:  # Comparer la valeur de la cellule courante avec 'valeur'
        return ...
      courant = ...  # Avancer dans la liste
    return False  # Retourner False si l'élément n'est pas trouvé

  def ajouter_debut(self, valeur):
    """Ajoute une cellule au début de la liste."""
    nouvelle_cellule = ...  # Créer une nouvelle cellule
    self.tete = ...  # Mettre la nouvelle cellule en tête de liste
```