Exercices type bac

Programmation orientée objet

Présentation.
Prérequis : cours sur la POO, exercices sur la POO, TP sur les classes, récursivité
Ces trois exercices reprennent des sujets de l'épreuve écrite du baccalauréat : Asie 2024 (jour 1), centres étrangers 2024 (jour 2) et Asie 2025 (jour 2). Les questions ont été adaptées au travail sur machine : le résultat d'un programme ne se devine plus, il se vérifie en l'exécutant. Elles ont aussi été prolongées : recherche et correction des erreurs des programmes du sujet, justification des algorithmes, coût, mémoïsation.

Exercice 1 *Antennes relais (Asie 2024, jour 1)*

Une entreprise doit placer des antennes relais le long d'une rue rectiligne. Une antenne de rayon p couvre toutes les maisons situées à une distance inférieure ou égale à p de l'antenne. Connaissant les positions des maisons, l'entreprise veut couvrir toutes les maisons avec le moins d'antennes possible.

La rue est représentée par un axe gradué : chaque maison et chaque antenne est repérée par son abscisse. Une antenne placée en 2,5 avec un rayon 1 couvre ainsi les maisons dont l'abscisse est comprise entre 1,5 et 3,5.

Les maisons et les antennes sont des objets des classes suivantes.

```
class Maison:
    def __init__(self, position):
        self._position = position

    def get_pos_maison(self):
        return self._position

class Antenne:
    def __init__(self, position, rayon):
        self._position = position
        self._rayon = rayon

    def get_pos_antenne(self):
        return self._position

    def get_rayon(self):
        return self._rayon
```

Partie A. Modélisation

1. Ajouter à chacune des deux classes une méthode `__repr__` : une maison en 3,5 s'affiche `Maison(3.5)` et une antenne en 2,5 de rayon 1 s'affiche `Antenne(2.5, 1)`.
2. Exécuter le code suivant, qui reprend la fonction `creation_rue` du sujet : elle reçoit une liste de positions et renvoie la liste des objets `Maison` correspondants, rangés par abscisse croissante. Expliquer ce que montre l'affichage.

```
def creation_rue_sujet(pos):
    pos.sort()
    maisons = []
    for p in pos:
        m = Maison(p)
        maisons.append(m)
    return maisons

positions = [9, 2, 5]
rue = creation_rue_sujet(positions)
print(positions)
```

3. Écrire une fonction `creation_rue(positions)` qui renvoie la même liste de maisons sans modifier la liste `positions` reçue.
4. Ajouter une méthode `couvre(self, maison)` à la classe `Antenne` : elle renvoie `True` si l'antenne couvre la maison et `False` sinon. La méthode ne doit pas accéder directement aux attributs d'une maison : elle passe par `get_pos_maison`. On rappelle que la valeur absolue s'obtient avec `abs`.
5. Écrire une fonction `toutes_couvertes(maisons, antennes)` qui renvoie `True` si chaque maison de la liste `maisons` est couverte par au moins une antenne de la liste `antennes`, et `False` sinon.

Partie B. Placement des antennes

6. Lire la fonction `strategie_1` suivante, utilisée par l'entreprise. Elle reçoit une liste de maisons triée par abscisse croissante et le rayon des antennes, et renvoie la liste des antennes placées.

```
def strategie_1(maisons, rayon):
    antennes = [Antenne(maisons[0].get_pos_maison(), rayon)]
    for m in maisons[1:]:
        if not antennes[-1].couvre(m):
            antennes.append(Antenne(m.get_pos_maison(), rayon))
    return antennes
```

   Expliquer pourquoi les antennes renvoyées par `strategie_1` couvrent toujours toutes les maisons. Vérifier ensuite avec `toutes_couvertes` sur la rue `creation_rue([0, 2, 3, 4, 5, 7, 9, 10.5, 11.5])` avec un rayon 2.
7. Écrire la fonction `strategie_2(maisons, rayon)` qui applique une stratégie améliorée : on parcourt les maisons par abscisse croissante et, dès qu'une maison d'abscisse x n'est pas couverte, on place une antenne en $x + r$, où r est le rayon. Comparer ensuite le nombre d'antennes des deux stratégies sur la rue de la question 6.
8. Justifier qu'aucune autre façon de placer les antennes n'en utilise moins que `strategie_2`.
9. Donner, en fonction du nombre n de maisons et du nombre k d'antennes, le coût de `strategie_1`, de `strategie_2` et de `toutes_couvertes` dans le pire des cas. On admet que `append` a un coût constant.
10. Écrire une fonction `toutes_couvertes_rapide(maisons, antennes)` qui donne le même résultat que `toutes_couvertes` lorsque les antennes ont toutes le même rayon et sont rangées par abscisse croissante, comme les maisons : c'est le cas des antennes renvoyées par les deux stratégies. La fonction ne doit parcourir qu'une seule fois chacune des deux listes : son coût doit être de l'ordre de $n + k$.

Exercice 2 *Itinéraires dans une grille (centres étrangers 2024, jour 2)*

On se déplace dans une grille rectangulaire, de la case en haut à gauche jusqu'à la case en bas à droite. Les seuls déplacements autorisés sont d'une case vers la droite, noté `D`, ou d'une case vers le bas, noté `B`. Un itinéraire est une chaîne de caractères formée de ces deux lettres.

Le nombre de `D` d'un itinéraire est sa **longueur**, le nombre de `B` est sa **largeur**. L'itinéraire `"DDBDBBDDDDB"` a pour longueur 7 et pour largeur 4 ; il se dessine dans une grille de 5 lignes et 8 colonnes, où `S` marque la case de départ, `E` la case d'arrivée et `*` les cases visitées :

```
S * * . . . . .
. . * * . . . .
. . . * . . . .
. . . * * * * *
. . . . . . . E
```

Une case est repérée par le couple `(i, j)` de son numéro de ligne et de son numéro de colonne : le départ est en `(0, 0)`, un déplacement `D` augmente `j` de 1 et un déplacement `B` augmente `i` de 1.

Un itinéraire est représenté par un objet de la classe `Chemin` suivante. L'attribut `grille` est une liste de listes : `grille[i][j]` est le caractère de la case `(i, j)`, `'.'` au départ.

```python
class Chemin:

    def __init__(self, itineraire):
        self.itineraire = itineraire
        longueur, largeur = 0, 0
        for direction in self.itineraire:
            if direction == "D":
                longueur = longueur + 1
            if direction == "B":
                largeur = largeur + 1
        self.longueur = longueur
        self.largeur = largeur
        self.grille = [['.' for i in range(longueur + 1)] for j in range(largeur + 1)]
```

Partie A. Programmation orientée objet

1. Exécuter les instructions ci-dessous, dont le sujet demandait le résultat. Expliquer le message d'erreur obtenu, puis corriger la ligne fautive pour afficher la largeur et la longueur de `chemin_1`.

```python
chemin_1 = Chemin("DDBDBBDDDDB")
a = chemin_1.largeur
b = chemin_1.longueur
print(f"largeur : {a}, longueur : {b}")
```

2. Modifier le constructeur pour qu'il refuse, par une assertion accompagnée d'un message, un itinéraire qui contient un autre caractère que `D` ou `B` : en l'état, `Chemin("DDXB")` est accepté sans erreur.
3. Écrire une méthode `remplir_grille` qui parcourt l'itinéraire et modifie la grille : la case de départ reçoit `'S'`, chaque case visitée ensuite reçoit `'*'` et la case d'arrivée reçoit `'E'`.
4. Écrire une méthode `get_dimensions` qui renvoie le couple `(longueur, largeur)` de l'itinéraire.
5. Écrire une méthode `dessin` qui remplit la grille, puis renvoie la chaîne de caractères qui la représente : les cases d'une ligne sont séparées par une espace et les lignes par un retour à la ligne `"\n"`, comme dans le dessin ci-dessus. Écrire ensuite une méthode `tracer_chemin` qui affiche ce dessin.

6. Écrire une méthode `cases` qui renvoie la liste des couples `(i, j)` des cases visitées, dans l'ordre du parcours, du départ jusqu'à l'arrivée comprises. Pour `Chemin("DBB")`, la méthode renvoie `[(0, 0), (0, 1), (1, 1), (2, 1)]`.
7. Écrire une méthode `croise(self, autre)` qui reçoit un autre objet `Chemin` de mêmes dimensions et renvoie `True` si les deux itinéraires passent par une même case autre que le départ et l'arrivée, et `False` sinon. Par exemple, `"DBDB"` et `"BDBD"` passent tous deux par la case `(1, 1)`, alors que `"DDBB"` et `"BBDD"` ne se croisent pas.

Partie B. Itinéraires aléatoires

8. Écrire une fonction `itineraire_aleatoire(m, n)` qui renvoie un itinéraire de largeur `m` et de longueur `n` tiré au hasard selon l'algorithme du sujet : tant que l'on n'a atteint ni la dernière ligne ni la dernière colonne, on tire au sort un déplacement `D` ou `B`, on l'ajoute à l'itinéraire et on met à jour `i` ou `j` ; on termine ensuite l'itinéraire par les déplacements qui mènent à la case d'arrivée. La fonction `choice` du module `random` renvoie un élément choisi au hasard dans une liste non vide.
9. Écrire une fonction `frequences(m, n, essais)` qui tire `essais` itinéraires avec `itineraire_aleatoire(m, n)` et renvoie un dictionnaire qui associe à chaque itinéraire obtenu son nombre d'apparitions. L'appeler avec `m = 2`, `n = 2` et 6 000 essais. Les six itinéraires possibles sont-ils obtenus aussi souvent les uns que les autres ? Expliquer les résultats en calculant la probabilité d'obtenir `"DDBB"`, puis celle d'obtenir `"DBDB"`.

Partie C. Nombre d'itinéraires

10. Justifier, en notant $\mathrm{N}(m, n)$ le nombre d'itinéraires différents de largeur m et de longueur n, que $\mathrm{N}(0, n) = \mathrm{N}(m, 0) = 1$ et que, pour $m \geqslant 1$ et $n \geqslant 1$, $\mathrm{N}(m, n) = \mathrm{N}(m-1, n) + \mathrm{N}(m, n-1)$. En déduire une fonction récursive `nombre_chemins(m, n)`.
11. Mesurer avec la fonction `time` du module `time` la durée de `nombre_chemins(k, k)` pour `k` allant de 10 à 13. Que constate-t-on ? Écrire une version `nombre_chemins_memo(m, n)` qui range dans un dictionnaire `memo`, à la clé `(m, n)`, chaque résultat déjà calculé, puis calculer `nombre_chemins_memo(30, 30)`.

Exercice 3 *Dates (Asie 2025, jour 2)*

La classe `Date` du sujet représente une date par trois entiers : le jour, le mois et l'année. L'attribut `nb_jours_par_mois` contient le nombre de jours de chaque mois, de janvier à décembre : février compte 28 jours, et 29 les années bissextiles. Une année est bissextile si elle est divisible par 4 mais pas par 100, ou si elle est divisible par 400.

La méthode `nb_jours_passes` renvoie le numéro du jour dans l'année : 1 pour le 1er janvier, 32 pour le 1er février. La méthode `nb_jours_restants` renvoie le nombre de jours qui restent dans l'année après la date. La méthode `nb_jours_depuis` renvoie le nombre de jours écoulés depuis la date `other` jusqu'à la date `self`, et `-1` si `other` est postérieure à `self`.

Le code ci-dessous est celui du sujet, questions résolues : il contient deux erreurs et un défaut de conception, que les questions font découvrir.

```
class Date:
    def __init__(self, jour, mois, annee):
        self.jour = jour
        self.mois = mois
        self.annee = annee
        self.nb_jours_par_mois = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
        if self.est_bissextile():
            self.nb_jours_par_mois[1] = 29
```

```
    def get_jour(self):
        return self.jour

    def get_mois(self):
        return self.mois

    def get_annee(self):
        return self.annee

    def set_jour(self, jour):
        self.jour = jour

    def set_mois(self, mois):
        self.mois = mois

    def set_annee(self, annee):
        self.annee = annee

    def est_bissextile(self):
        return (self.annee % 4 == 0 and self.annee % 100 != 0) or self.annee % 400 == 0

    def nb_jours_passes(self):
        nb_jours = self.jour
        mois = self.mois - 2
        while mois >= 0:
            nb_jours = nb_jours + self.nb_jours_par_mois[mois]
            mois = mois - 1
        return nb_jours

    def nb_jours_restants(self):
        j = 365
        if self.est_bissextile():
            j = 366
        return j - self.nb_jours_passes()

    def nb_jours_depuis(self, other):
        if other.get_annee() > self.get_annee():
            return -1
        if other.get_annee() == self.get_annee():
            if other.nb_jours_passes() > self.nb_jours_passes():
                return -1
            if other.nb_jours_passes() == self.nb_jours_passes():
                return 0
        nb_jours = self.nb_jours_passes() + other.nb_jours_restants()
        for annee in range(other.get_annee() + 1, self.get_annee()):
            d_suivant = date(1, 1, annee)
            if d_suivant.est_bissextile():
                nb_jours += 366
            else:
                nb_jours += 365
        return nb_jours
```

Partie A. Cohérence de l'objet

1. Écrire une méthode `__repr__` qui renvoie la date au format `"19/06/2024"`, le jour et le mois sur deux chiffres. L'expression `f"{valeur:02d}"` écrit un entier sur deux chiffres, avec un zéro devant si nécessaire.

2. Exécuter le code suivant, puis comparer le résultat avec celui de `Date(1, 3, 2024).nb_jours_passes()`. Expliquer l'écart.

```
d = Date(1, 3, 2023)
d.set_annee(2024)
print(d.nb_jours_passes())
```

3. Ajouter une méthode `jours_dans_mois(mois)` qui renvoie le nombre de jours du mois `mois` (de 1 à 12) de l'année de la date, puis réécrire `nb_jours_passes` avec cette méthode et une boucle `for`. L'attribut `nb_jours_par_mois` et les lignes qui le calculent deviennent inutiles : les retirer. Expliquer pourquoi ce changement supprime le défaut de la question 2.
4. Ajouter une méthode `est_valide` qui renvoie `True` si la date existe et `False` sinon. Par exemple, `Date(29, 2, 2023)` et `Date(31, 4, 2024)` ne sont pas valides.

Partie B. Entre deux dates

5. Exécuter les instructions du sujet ci-dessous. Expliquer l'erreur obtenue et corriger la méthode `nb_jours_depuis`.

```
d1 = Date(15, 6, 2024)
d2 = Date(15, 6, 2024)
d3 = Date(15, 7, 2024)
d4 = Date(15, 6, 2025)
d5 = Date(15, 6, 2022)
print(d1.nb_jours_depuis(d2))
print(d1.nb_jours_depuis(d3))
print(d1.nb_jours_depuis(d4))
print(d1.nb_jours_depuis(d5))
```

6. Calculer à la main le nombre de jours entre le 1er mars 2024 et le 20 mars 2024, puis le comparer avec la valeur de `Date(20, 3, 2024).nb_jours_depuis(Date(1, 3, 2024))`, à afficher dans la cellule de la question 5. Ce cas, où `other` précède `self` dans la même année, n'est jamais testé par les instructions du sujet. Modifier ensuite la méthode `nb_jours_depuis` pour corriger ce défaut.
7. Écrire une méthode `timestamp` qui renvoie le timestamp de la date, c'est-à-dire le nombre de secondes écoulées entre le 1er janvier 1970 à 0 h et cette date à 0 h ; elle utilise `nb_jours_depuis`. Le sujet annonce 1704063600 pour le 1er janvier 2024 : comparer avec la valeur calculée par la méthode et expliquer l'écart, sachant que le sujet a été rédigé en France, où l'heure légale d'hiver est en avance d'une heure sur le temps universel.
8. Écrire une méthode `__eq__` qui permet de comparer deux dates avec `==`.
9. Écrire une méthode `lendemain` qui renvoie un nouvel objet `Date` pour le jour suivant, sans modifier la date de départ. Tester sur le 30 avril 2026, le 28 février 2024, le 28 février 2023 et le 31 décembre 2024.
10. Le 1er janvier 1970 était un jeudi. Écrire une méthode `jour_semaine` qui renvoie le nom du jour de la semaine d'une date postérieure au 1er janvier 1970, par exemple `"vendredi"` pour le 9 octobre 2026. Pourquoi la méthode ne convient-elle pas pour le 14 juillet 1789 ?
11. Écrire, en question bonus, une fonction `date_du_jour(numero, annee)`, réciproque de `nb_jours_passes`, qui renvoie l'objet `Date` correspondant au jour numéro `numero` de l'année `annee`. Par exemple, `date_du_jour(60, 2024)` renvoie la date du 29 février 2024.