Asie - juin 2025 - sujet 2

Exercice 1 (Algorithmique et POO - 6 points)
Pour travailler sur des dates, on a créé la classe `Date` dont le code est écrit ci-dessous :

```
1  class Date:
2      def __init__(self, jour, mois, annee):
3          self.jour = ...
4          self.mois = ...
5          self.annee = ...
6          self.nb_jours_par_mois = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
7
8
9
10     def get_jour(self):
11         return self.jour
12
13     def get_mois(self):
14         return self.mois
15
16     def get_annee(self):
17         return ...
18
19     def set_jour(self, jour):
20         self.jour = jour
21
22     def set_mois(self, mois):
23         self.mois = ...
24
25     def set_annee(self, annee):
26         self.annee = annee
27
28     def est_bissextile(self):
29         ...
```

Partie A : accès et modification des données
Le constructeur de la classe `Date` prend en paramètres trois entiers représentant le jour, le mois et l'année, puis les affecte respectivement aux attributs `jour`, `mois` et `annee`.

1. Recopier et compléter les lignes 3 à 5 du code précédent.
2. Indiquer à quelle date correspond l'instance de la classe `Date` suivante : `d = Date(1, 5, 2000)`
3. Ecrire le code permettant de créer une instance `d` de la classe `Date` qui représente la date du 19 juin 2024.
4. La méthode `get_annee` renvoie la valeur de l'attribut `annee`. Recopier et compléter les lignes 16 et 17 du code précédent.
5. La méthode `set_mois` modifie l'attribut `mois` en lui affectant la valeur passée en argument. Recopier et compléter les lignes 22 et 23 du code précédent.

L'attribut `nb_jours_par_mois` contient une liste qui correspond au nombre de jours pour chaque mois. Le mois de février contient généralement 28 jours mais lors des années bissextiles il en contient 29.

6. La classe `Date` dispose d'une méthode `est_bissextile`, qui utilise uniquement l'attribut `annee`, et qui renvoie `True` si l'année de l'instance courante est bissextile et `False` sinon. On veut compléter la méthode `__init__` pour ajuster le nombre de jours par mois pour les années bissextiles. Recopier et compléter les lignes 7 et 8 suivantes :

```
self.nb_jours_par_mois = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
if ... :
    self.nb_jours_par_mois[...] = 29
```

Partie B : sur l'année de l'instance courante

Pour déterminer le nombre de jours au cours d'une année, il faut savoir si elle est bissextile. On rappelle qu'une année est bissextile si elle est divisible par 4 mais pas par 100 ou si elle est divisible par 400.

7. Ecrire le code de la méthode `est_bissextile`.
   On rappelle qu'un entier `a` est divisible par l'entier `n` si `a%n == 0`.

On dote la classe `Date` de la méthode `nb_jours_passes` qui renvoie le nombre de jours passés dans l'année de l'instance courante.

```
1 def nb_jours_passes(self):
2     nb_jours = self.jour
3     mois = self.mois - 2
4     while mois >= 0:
5         nb_jours = nb_jours + self.nb_jours_par_mois[mois]
6         mois = mois - 1
7     return nb_jours
```

8. Indiquer quel sera l'affichage en console après l'exécution des deux instructions suivantes :

```
>>> d1 = Date(20, 3, 2001)
>>> d1.nb_jours_passes()
```

On dote la classe `Date` de la méthode `nb_jours_restants` qui renvoie le nombre de jours restants dans l'année de l'instance courante, soit 366 ou 365 moins le nombre de jours déjà passés, selon que l'année est bissextile ou non.

9. Recopier et compléter le script de la méthode `nb_jours_restants` ci-après :

```
1 def nb_jours_restants(self):
2     j = 365
3     if ...:
4         j = 366
5     return j - ...
```

Partie C : entre deux dates

On dote la classe `Date` de la méthode `nb_jours_depuis` qui prend en paramètre une autre instance `other` de la classe `Date` et qui renvoie le nombre de jours écoulés entre la date de l'instance `other` et la date de l'instance courante.

```
1  def nb_jours_depuis(self, other):
2      if other.get_annee() > self.get_annee():
3          return -1
4      if other.get_annee() == self.get_annee():
5          if other.nb_jours_passes() > self.nb_jours_passes():
6              return -1
7          if other.nb_jours_passes() == self.nb_jours_passes():
8              return 0
9      nb_jours = self.nb_jours_passes() + other.nb_jours_restants()
10     for annee in range(other.get_annee()+1, self.get_annee()):
11         d_suivant = date(1, 1, annee)
12         if d_suivant.est_bissextile():
13             nb_jours += 366
14         else:
15             nb_jours += 365
16     return nb_jours
```

On crée les instances de la classe `Date` suivantes :

```
>>> 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)
```

10. Indiquer quels seront alors les affichages en console après l'exécution de chacune des instructions suivantes (on précise que l'année 2024 est bissextile)

```
>>> d1.nb_jours_depuis(d2)
>>> d1.nb_jours_depuis(d3)
>>> d1.nb_jours_depuis(d4)
>>> d1.nb_jours_depuis(d5)
```

Le *timestamp* est le nombre de secondes qui se sont écoulées depuis le 1er janvier 1970 à 00h00. Il s'agit de la date de la mise en marche du système d'exploitation UNIX. Par exemple, le 01/01/2024 à 00 :00 :00 correspond au *timestamp* de `1704063600`.

11. Recopier et compléter le code de la méthode `timestamp` qui renvoie le nombre de secondes qui se sont écoulées depuis le 1er janvier 1970.

```
1 def timestamp(self):
2     d = ...
3     return self.nb_jours_depuis(d) * 24 * 3600
```

Exercice 2 (Programmation python et gestion de processus - 6 points)
On souhaite élaborer un programme système permettant de gérer l'ordre d'exécution des processus sur le processeur.

1. Donner le nom de ce type de programme.
2. Donner les différents états possibles d'un processus.

Chaque processus dispose d'une valeur de priorité. Un processus est prioritaire sur un autre processus si sa valeur de priorité est plus petite. Ainsi pour rendre un processus moins prioritaire, il faut augmenter sa valeur de priorité, par exemple en la faisant passer de 2 à 3.

Fonctionnement du programme gérant l'ordre d'exécution des processus :

- ⋆ On dispose d'une liste dont les éléments sont des files de processus. La première file contient les processus ayant la valeur de priorité la plus élevée 0, la seconde ceux ayant la valeur de priorité 1, etc. A l'arrivée d'un nouveau processus :
  - Attribuer au nouveau processus la valeur de priorité la plus élevée 0 ;
  - Placer le nouveau processus dans la file d'attente correspondant à sa valeur de priorité (c'est-à-dire la première file de la liste).
- ⋆ À chaque cycle d'horloge :
  - S'il n'y a pas de processus en cours d'exécution et s'il reste des processus en attente :
    — sélectionner un processus avec la priorité la plus élevée dans l'une des files d'attente non vides ;
    — élire ce processus comme nouveau processus en cours d'exécution ;
  - Sinon si un processus est en cours d'exécution :
    — Si le processus a terminé son exécution, le retirer du processeur ;
    — Sinon,
      + incrémenter le temps d'utilisation du processus ;
      + Si des processus de priorité supérieure ou égale attendent :
        * retirer le processus en cours d'exécution du processeur ;
        * réduire sa priorité de 1 et le mettre dans la file d'attente correspondant à sa priorité ;
        * élire un processus dont la priorité est la plus élevée parmi les processus des files d'attente non vides ;
      + sinon, réduire sa priorité de 1 et continuer à exécuter le processus en cours d'exécution.

3. Parmi les propositions suivantes, donner la structure la plus adaptée pour stocker les processus d'une même priorité :
   - ⋆ Proposition 1 : liste
   - ⋆ Proposition 2 : file
   - ⋆ Proposition 3 : pile

Pour représenter le processus, on utilise une classe `Processus` qui possède les variables d'instances `PID` (l'identifiant du processus), `priorite` (la priorité du processus), `temps_utilisation` sur le CPU et le temps nécessaire à son exécution `temps_CPU`.

4. Compléter le constructeur de la classe `Processus` :

```
1 class Processus:
2     ...(self, ..., priorite, temps_CPU):
3         ... priorite = priorite
4         ... PID = ...
5         self.temps_utilisation = 0
6         self.temps_CPU = temps_CPU
```

5. On considère les trois processus suivants :

```
P1 = Processus(PID=1,priorite=0,temps_CPU=10)
P2 = Processus(PID=2,priorite=0,temps_CPU=7)
P3 = Processus(PID=3,priorite=0,temps_CPU=5)
```

On a donc `liste_files=[[P3, P2, P1], [], []]`.
Compléter la simulation suivante, dans laquelle la variable `CPU` contient le processus en cours d'exécution :

```
Cycle 1: CPU=P1 liste_files=[[P3, P2], [], []]
Cycle 2: CPU=P2 liste_files=[[P3],[P1],[]]
Cycle 3: CPU=P3 liste_files=[[], [...],[]]
Cycle 4: CPU=P3 liste_files=[[], [...], [...]]
Cycle 5: CPU=... liste_files=[[], [...], [...]]
```

Dans les questions 6 et 7, on dispose :

- ⋆ d'un processus qui nécessite un temps d'utilisation de 1000 pour terminer ;
- ⋆ d'un nombre important de processus dont le temps d'utilisation pour terminer est de 4 où l'on suppose de plus que chaque processus terminé est remplacé par un nouveau processus similaire.

6. Expliquer pourquoi le processus qui nécessite un long temps d'utilisation du CPU risque de ne jamais terminer avec le programme de gestion de l'ordre des processus ci-dessus (indiquer notamment la priorité du processus long au bout de quelques temps).

Pour régler ce phénomène, on décide d'ajouter la variable d'instance `temps_d_attente` au processus, et on définit une constante appelée `Max_Temps` qui correspond au temps maximum qu'un processus attend avant de remonter sa priorité. L'idée est qu'à chaque cycle, le `temps_d_attente` augmente. Ainsi, si sa valeur dépasse `Max_Temps`, alors sa priorité augmente.

7. Expliquer pourquoi le processus qui nécessite un temps long d'utilisation du CPU ne risque plus de ne jamais terminer avec ce nouveau programme de gestion de l'ordre des processus.
8. Ecrire une fonction `meilleur_priorite` qui renvoie `None` s'il n'y a plus de processus et la priorité de l'un des processus les plus prioritaires de la liste des files d'attente dans le cas contraire.

```
1 def meilleur_priorite(liste_files):
2     ...
```

Exemple :

```
# p1, p2 et p3 sont des instances de la classe 'Processus'
>>> liste_files = [[], [p2], [p3, p1]]
>>> meilleur_priorite(liste_files)
1
```

9. Ecrire une fonction `prioritaire` qui renvoie `None` si aucune des files d'attente de la liste ne contient un processus et qui renvoie l'un des processus parmi les plus prioritaires sinon (dans ce cas la fonction `prioritaire` supprimera le processus choisi de la file d'attente dans laquelle il se trouvait).

```
1 def prioritaire(liste_files):
2     ...
```

On pourra utiliser `liste.pop(i)` pour renvoyer l'élément de la liste à l'indice `i`, tout en le supprimant de la liste.

10. Ecrire une fonction `gerer` qui récupère le processus en cours d'exécution `p` ainsi que la liste des files d'attente `liste_files` et qui implémente le programme donné en début d'énoncé pour gérer les processus.

```
1 def gerer(p, liste_files):
2     ...
```

Exercice 3 (Dictionnaires, récursivité, spécification, POO, bases de données et arbres binaires - 8 points)
Cet exercice est composé de trois parties indépendantes.

Partie A
Dans cette partie, on s'intéresse à la gestion de la base de données d'un hôpital. On pourra utiliser les mots-clés SQL suivants : `AND`, `FROM`, `INSERT`, `INTO`, `JOIN`, `ON`, `SELECT`, `SET`, `UPDATE`, `VALUES`, `WHERE`. On utilisera également la fonction d'agrégation `COUNT` qui renvoie le nombre d'enregistrements correspondant à une requête.

La table `Patient` possède les attributs suivants :

- `nom_patient` de type TEXT (clé primaire) ;
- `prenom` de type TEXT ;
- `numero_secu` de type INT ;
- `age` de type INT.

Patient			
nom_patient	prenom	numero_secu	age
Heartman	Alice	207053523800187	17
Douglas	Bob	100017500155572	24
Woods	Caroll	258125930610747	65

La table `Symptome` possède les attributs suivants :

- `nom_patient` de type TEXT (clé primaire et clé étrangère) ;
- `toux` de type TEXT ;
- `fievre` de type TEXT ;
- `nausee` de type TEXT ;
- `anosmie` de type TEXT.

Symptome				
nom_patient	toux	fievre	nausee	anosmie
Heartman	Oui	Non	Non	Oui
Douglas	Non	Oui	Oui	Non
Woods	Oui	Oui	Non	Non

La table `Maladie` possède, entre autres, l'attribut `nom_maladie` de type TEXT, qui est la clé primaire. Les autres attributs de cette table ne sont pas représentés car ils ne sont pas utiles pour l'exercice.

Maladie
nom_maladie
Covid-19
Gastroentérite

La table `Diagnostic` possède les attributs suivants :

- `nom_patient` de type TEXT (clé primaire et clé étrangère) ;
- `nom_maladie` de type TEXT (clé étrangère).

Diagnostic	
nom_patient	nom_maladie
.....	.....
.....	.....

1. Ecrire une requête SQL permettant d'obtenir les noms et prénoms des patients ayant strictement plus de 60 ans.
2. Alice Heartman ne tousse plus. Ecrire une requête SQL permettant de mettre à jour la base de données avec cette information.
3. On souhaite effectuer des statistiques sur les symptômes des patients atteints de Covid-19. Écrire une requête SQL permettant de connaître le nombre de patients avec un diagnostic de Covid-19 qui toussent.

Un employé de l'hôpital saisit la requête suivante :

```
INSERT INTO Patients VALUES ('Douglas', 'Patrick', 168077230253829, 55)
```

4. Expliquer pourquoi cette requête produit une erreur.
5. Proposer une modification du schéma relationnel qui permettrait de résoudre ce problème.

Partie B

On s'intéresse maintenant à l'automatisation du diagnostic à partir des symptômes. Cette automatisation se fait à l'aide d'un arbre de décision binaire, tel que celui illustré sur la figure 1.

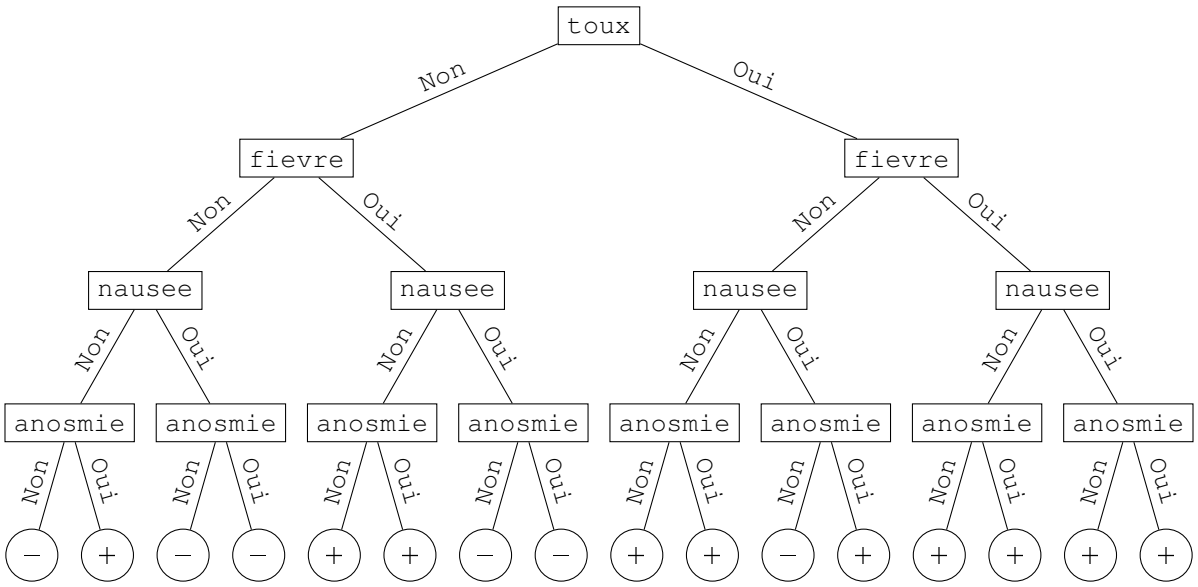

FIGURE 1 – Exemple d'arbre de décision binaire

Chaque nœud interne de l'arbre est étiqueté par un symptôme, et chaque feuille est étiquetée par un diagnostic. Pour établir un diagnostic, on se place à la racine et on parcourt l'arbre de la manière suivante :

- si on arrive à une feuille, le diagnostic est l'étiquette de cette feuille ;
- sinon, on regarde si l'étiquette du nœud est un des symptômes du patient. Si oui, on continue le parcours dans le sous-arbre droit, sinon, on continue le parcours dans le sous-arbre gauche.

L'arbre de la figure 1 donne un diagnostic pour la Covid-19. Par exemple, un patient qui ne tousse pas et n'a pas d'anosmie, mais a de la fièvre et des nausées est diagnostiqué négatif si on suit cet arbre de décision.

6. Donner le diagnostic pour un patient qui tousse et qui a de la fièvre, mais n'a pas de nausée ni d'anosmie, d'après l'arbre de la figure 1.

On décide d'implémenter les arbres binaires à l'aide de la classe `Noeud` ci-dessous :

```
class Noeud:
    def __init__(self, valeur, gauche = None, droit = None):
        """valeur correspond au symptome si le noeud est
        interne ou au diagnostic si le noeud est une feuille"""
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

    def est_feuille(self):
        """renvoie vrai si le noeud est une feuille faux sinon"""
        return self.gauche == None and self.droit == None

    def symptome(self):
        assert not self.est_feuille()
        return self.valeur

    def diagnostic(self):
        assert self.est_feuille()
        return self.valeur
```

7. Préciser la signification de l'assertion de la méthode `symptome`.
8. Nommer un attribut et une méthode de la classe `Noeud`.

On représente les symptômes d'un patient en Python par un dictionnaire dont les clés sont les symptômes possibles, et les valeurs sont `True` si le patient présente ce symptôme et `False` sinon. Par exemple, les symptômes du patient de la question 7 sont représentés par le dictionnaire suivant :

```
patient = {'toux' : True, 'fievre' : True, 'nausee' : False, 'anosmie' : False}
```

9. Compléter la fonction `applique` suivante, définie récursivement, qui renvoie le diagnostic établi en utilisant un arbre de décision binaire implémenté à l'aide de la classe `Noeud` précédente.

```
def applique(arbre, patient):
    if arbre.est_feuille():
        ...
    else:
        if patient[arbre.symptome()]:
            ...
        else:
            ...
```

10. Donner la taille de l'arbre représenté en figure 1. On considère que la taille d'un arbre constitué d'une unique feuille est 1.

On souhaite réduire la taille de cet arbre en utilisant l'observation suivante : un nœud dont les deux sous-arbres sont des feuilles correspondant au même diagnostic peut être remplacé par une feuille correspondant à ce diagnostic, comme illustré en figure 2.

FIGURE 2 – Règle de réduction pour les arbres de décision binaire

11. Appliquer cette règle à l'arbre de la figure 1 pour le réduire et dessiner le nouvel arbre.

12. Compléter la méthode `reduire` qui permet d'appliquer cette règle récursivement pour réduire la taille d'un arbre de décision binaire.

```
def reduire(self):
    """fonction récursive qui réduit la taille d'un arbre de
    décision sans changer les décisions prises"""
    if self.est_feuille():
        return ...
    self.gauche.reduire()
    self. ...
    if self.gauche.est_feuille() and ... and ... == ... :
        self.valeur = ...
        self.gauche = ...
        self.droite = ...
```

Partie C

Dans cette partie, on s'intéresse à l'intégrité et à la sécurité des données.

Sur les 15 chiffres du numéro de sécurité sociale, 2 servent à détecter les erreurs : étant donné le nombre n formé des 13 premiers chiffres, le nombre k formé les 2 derniers chiffres, appelé la clé, est choisi pour que $n + k$ soit un multiple de 97.

Par exemple, 207053523800187 est bien formé car :

$$2070535238001 + 87 = 97 \times 21345724104.$$

On rappelle que les opérateurs `%` et `//` permettent en Python d'obtenir respectivement le reste et le quotient dans une division euclidienne. Par exemple : `13%3` renvoie `1` et `13//3` renvoie `4` (car $13 = 3 \times 4 + 1$). On peut donc vérifier qu'un nombre entier n est un multiple de p en testant si le reste de la division de n par p vaut zéro.

13. Recopier et compléter la fonction `verifie` suivante qui renvoie un booléen indiquant si un numéro de sécurité sociale représenté par un entier (type int) est bien formé.

```
def verifie(num_secu):
    n = num_secu // 100
    k = num_secu % 100
    return ...
```

14. Compléter la fonction `cle` qui permet de renvoyer la clé `k` d'un numéro de sécurité sociale en prenant pour paramètre le nombre `n` formé des 13 premiers chiffres du numéro de sécurité sociale.

```
def cle(n):
    ...
```