Interrogation (2 h)

(Calculatrice non autorisée)

Exercice 1 *(8 points)*

Cet exercice traite du thème « structures de données », et principalement des piles.

La classe `Pile` utilisée dans cet exercice est implémentée en utilisant des listes Python et propose quatre éléments d'interface :

- Un constructeur qui permet de créer une pile vide, représentée par `[]` ;
- La méthode `est_vide()` qui renvoie `True` si l'objet est une pile ne contenant aucun élément, et `False` sinon ;
- La méthode `empiler` qui prend un objet quelconque en paramètre et ajoute cet objet au sommet de la pile. Dans la représentation de la pile dans la console, cet objet apparaît à droite des autres éléments de la pile ;
- La méthode `depiler` qui renvoie l'objet présent au sommet de la pile et le retire de la pile.

Exemples :

```
>>> mapile = Pile()
>>> mapile.empiler(2)
>>> mapile
[2]
>>> mapile.empiler(3)
>>> mapile.empiler(50)
>>> mapile
[2, 3, 50]
>>> mapile.depiler()
50
>>> mapile
[2, 3]
```

La méthode `est_triee` ci-dessous renvoie `True` si, en dépilant tous les éléments, ils sont traités dans l'ordre croissant, et `False` sinon.

```
1 def est_triee(self):
2   if not self.est_vide() :
3      e1 = self.depiler()
4      while not self.est_vide():
5          e2 = self.depiler()
6          if e1 ... e2 :
7              return False
8          e1 = ...
9    return True
```

1. Recopier sur la copie les lignes 6 et 8 en complétant les points de suspension.

On créé dans la console la pile `A` représentée par `[1, 2, 3, 4]`.

2. a. Donner la valeur renvoyée par l'appel `A.est_triee()`.
   b. Donner le contenu de la pile `A` après l'exécution de cette instruction.

On souhaite maintenant écrire le code d'une méthode `depileMax` d'une pile non vide ne contenant que des nombres entiers et renvoyant le plus grand élément de cette pile en le retirant de la pile.
Après l'exécution de `p.depileMax()`, le nombre d'éléments de la pile `p` diminue donc de 1.

```
1   def depileMax(self):
2       assert not self.est_vide(), "Pile vide"
3       q = Pile()
4       maxi = self.depiler()
5       while not self.est_vide() :
6          elt = self.depiler()
7          if maxi < elt :
8             q.empiler(maxi)
9             maxi = ...
10         else :
11            ...
12      while not q.est_vide():
13         self.empiler(q.depiler())
14      return maxi
```

3. Recopier sur la copie les lignes 9 et 11 en complétant les points de suspension.

On créé la pile `B` représentée par [9, -7, 8, 12, 4] et on effectue l'appel `B.depileMax()`.

4. a. Donner le contenu des piles `B` et `q` à la fin de chaque itération de la boucle `while` de la ligne 5.
   b. Donner le contenu des piles `B` et `q` avant l'exécution de la ligne 14.
   c. Donner un exemple de pile qui montre que l'ordre des éléments restants n'est pas préservé après l'exécution de `depileMax`.

On donne le code de la méthode `traiter()` :

```
1   def traiter(self):
2       q = Pile()
3       while not self.est_vide():
4           q.empiler(self.depileMax())
5       while not q.est_vide():
6           self.empiler(q.depiler())
```

5. a. Donner les contenus successifs des piles `B` et `q`
   - avant la ligne 3,
   - avant la ligne 5,
   - à la fin de l'exécution de la fonction `traiter`

   lorsque la fonction `traiter` est appliquée sur la pile `B` contenant [1, 6, 4, 3, 7, 2].

   b. Expliquer le traitement effectué par cette méthode.

Exercice 2 *(12 points)*

Cet exercice porte sur les structures de Files

Simon *est un jeu de société électronique de forme circulaire comportant quatre grosses touches de couleurs différentes : rouge, vert, bleu et jaune.*
Le jeu joue une séquence de couleurs que le joueur doit mémoriser et répéter ensuite. S'il réussit, une couleur parmi les 4 est ajoutée à la fin de la séquence. La nouvelle séquence est jouée depuis le début et le jeu continue. Dès que le joueur se trompe, la séquence est vidée et réinitialisée avec une couleur et une nouvelle partie commence.

Source Wikipédia

Exemple de séquence jouée : rouge→bleu→rouge→jaune→bleu

Dans cet exercice nous essaierons de reproduire ce jeu.
Les quatre couleurs sont stockées dans un tuple nommé `couleurs` :

```
couleurs = ("bleu", "rouge", "jaune", "vert")
```

Pour stocker la séquence à afficher nous utiliserons une structure de file que l'on nommera `sequence` tout au long de l'exercice.
La file est une structure linéaire de type FIFO (First In First Out). Nous utiliserons durant cet exercice les fonctions suivantes :

Structure de données abstraite : File	
`creer_file_vide()`	: renvoie une file vide
`est_vide(f)`	: renvoie `True` si `f` est vide, `False` sinon
`enfiler(f, element)`	: ajoute `element` en queue de `f`
`defiler(f)`	: retire l'élément en tête de `f` et le renvoie
`taille(f)`	: renvoie le nombre d'éléments de `f`

En fin de chaque séquence, le Simon tire au hasard une couleur parmi les 4 proposées. On utilisera la fonction `randint(a,b)` de la bibliothèque `random` qui permet d'obtenir un nombre entier compris entre **a** inclus et **b** inclus pour le tirage aléatoire.
Exemple : `randint(1,5)` peut renvoyer 1, 2, 3, 4 ou 5.

1.

Recopier et compléter, sur votre copie, les `...` des lignes 3 et 4 de la fonction `ajout(f)` qui permet de tirer au hasard une couleur et de l'ajouter à une séquence.
La fonction `ajout` prend en paramètre la séquence `f` ; elle renvoie la séquence `f` modifiée (qui intègre la couleur ajoutée au format chaîne de caractères).

```
1  def ajout(f):
2      couleurs = ("bleu", "rouge", "jaune", "vert")
3      indice = randint(...,...)
4      enfiler(...,...)
5      return f
```

En cas d'erreur du joueur durant sa réponse, la partie reprend au début ; il faut donc vider la file `sequence` pour recommencer à zéro en appelant `vider(sequence)` qui permet de rendre la file `sequence` vide sans la renvoyer.

2.

Ecrire la fonction `vider` qui prend en paramètre une séquence `f` et la vide sans la renvoyer.

Le Simon doit afficher successivement les différentes couleurs de la séquence.
Ce rôle est confié à la fonction `affich_seq(sequence)`, qui prend en paramètre la file de couleurs `sequence`, définie par l'algorithme suivant :

- on ajoute une nouvelle couleur à `sequence` ;
- on affiche les couleurs de la séquence, une par une, avec une pause de 0,5 s entre chaque affichage.

Une fonction `affichage(couleur)` (dont la rédaction n'est pas demandée dans cet exercice) permettra l'affichage de la couleur souhaitée avec `couleur` de type chaîne de caractères correspondant à une des 4 couleurs.
La temporisation de 0,5 s sera effectuée avec la commande `time.sleep(0.5)`.
Après l'exécution de la fonction `affich_seq`, la file `sequence` ne devra pas être vidée de ses éléments.

3.

Recopier et compléter, sur la copie, les `...` des lignes 4 à 10 de la fonction `affich_seq(sequence)` ci-dessous :

```
1  def affich_seq(sequence):
2      stock = creer_file_vide()
3      ajout(sequence)
4      while not est_vide(sequence) :
5          c = ...
6          ...
7          time.sleep(0.5)
8          ...
9      while ... :
10         ...
```

4.

Cette question est indépendante des précédentes : bien qu'elle fasse appel aux fonctions construites précédemment, elle peut être résolue même si le candidat n'a pas réussi toutes les questions précédentes.

Nous allons ici créer une fonction `tour_de_jeu(sequence)` qui gère le déroulement d'un tour quelconque de jeu côté joueur. La fonction `tour_de_jeu` prend en paramètre la file de couleurs `sequence`, qui contient un certain nombre de couleurs.

- Le jeu électronique Simon commence par ajouter une couleur à la séquence et affiche l'intégralité de la séquence.
- Le joueur doit reproduire la séquence dans le même ordre. Il choisit une couleur via la fonction `saisie_joueur()`.
- On vérifie si cette couleur est conforme à celle de la séquence.
- S'il s'agit de la bonne couleur, on poursuit sinon on vide `sequence`.
- Si le joueur arrive au bout de la séquence, il valide le tour de jeu et le jeu se poursuit avec un nouveau tour de jeu, sinon le joueur a perdu et le jeu s'arrête.

La fonction `tour_de_jeu` s'arrête donc si le joueur a trouvé toutes les bonnes couleurs de `sequence` dans l'ordre, ou bien dès que le joueur se trompe.

Après l'exécution de la fonction `tour_de_jeu`, la file `sequence` ne devra pas être vidée de ses éléments en cas de victoire.

a. Afin d'obtenir la fonction `tour_de_jeu(sequence)` correspondant au comportement décrit ci-dessus, recopier le script ci-dessous et :

- Compléter le `...`
- Choisir parmi les propositions de syntaxes suivantes lesquelles correspondent aux ZONES A, B, C, D, E et F figurant dans le script et les y remplacer (il ne faut donc en choisir que six parmi les onze) :

```
vider(sequence)
defiler(sequence)
enfiler(sequence,c_joueur)
enfiler(stock,c_seq)
enfiler(sequence, defiler(stock))
enfiler(stock, defiler(sequence))
affich_seq(sequence)
while not est_vide(sequence):
while not est_vide(stock):
if not est_vide(sequence):
if not est_vide(stock):
```

```
1  def tour_de_jeu(sequence):
2      ZONE A
3      stock = creer_file_vide()
4      while not est_vide(sequence) :
5          c_joueur = saisie_joueur()
6          c_seq = ZONE B
7          if c_joueur ... c_seq:
8              ZONE C
9          else:
10             ZONE D
11     ZONE E
12         ZONE F
```

b. Proposer une modification pour que la fonction se répète si le joueur trouve toutes les couleurs de la séquence (dans ce cas, une nouvelle couleur est ajoutée) ou s'il se trompe (dans ce cas, la séquence est vidée et se voit ajouter une nouvelle couleur). On pourra ajouter des instructions qui ne sont pas proposées dans la question a.