Exercice 1

File et programmation fonctionnelle

Bac 2023 Centres Etrangers J1 : Exercice 3

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.

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 :

```
1 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 :

Fonction	Description
`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'element en tête de `f` et le renvoie
`taille(f)`	renvoie le nombre d'elements 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).

```
def ajout(f):
    couleurs = ["bleu","rouge","jaune","vert"]
    indice = randint(...)
    enfiler(...,...)
    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 :

```
def affiche_seq(sequence):
    stock = ceer_file_vide()
    ajout(sequence)
    while not est_vide(sequence):
        c = ...
        ...
        time.sleep(0.5)
    while ... :
        ...
```

4. 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 :

- Complétez 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):
```

```
def tour_de_jeu(sequence):
    ZONE A
    stock = creer_file_vide()
    while not est_vide(sequence):
        c_joueur = saisie_joueur()
        c_seq = ZONE B
        if c_joueur ... c_seq:
            ZONE C
        else:
            ZONE D
    ZONE E
        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

Exercice 2

Liste chainée en Programmation Objet : Etudiants et Université

(exercice sur le modèle de celui `https://inf104.wp.imt.fr/tp5-malloc-et-listes-chainees/` écrit pour le langage C)

Une Université utilise un fichier avec la liste des étudiants ainsi que des informations les concernant. On suppose que ces renseignements sont mis sous la forme suivante :

```
D = {nom1: information1, nom2: information2, ... }
```

Les variables `nom1`, `nom2`, `nom3`, ... sont supposées être des variables globales.

Les renseignements sont mis sous la forme d'un dictionnaire `D`.

Un exemple de ce dictionnaire :

```
nom1 = "Aristote"
nom2 = "Badim"
nom3 = "Bossis"
D = {nom1: "L1 informatique", nom2: "L1 Chimie", ...}
```

On souhaite créer une *liste chainée* avec ces renseigments. Les étudiants sont alors les *maillons* d'une chaine. Les maillons s'appeleront `Etudiant`. Les valeurs contenues sont `nom` et `information`. Un attribut appelé `suiv` stocke le *pointeur* vers l'étudiant suivant.

2.1 Classe etudiant

Ecrire le script Python qui définit la classe `Etudiant`

2.2 Le 3e étudiant

Ecrire le script qui affecte les renseignements du dictionnaire au troiseme maillon `etudiant3`.

2.3 Le 2e étudiant

Ecrire le script qui affecte les renseignements du dictionnaire au 2e maillon `etudiant2`.

Pointer aussi vers le 3e étudiant en renseignant l'attribut `suiv` du deuxieme étudiant vers le troisieme.

2.4 Le premier étudiant

Ecrire le script qui affecte les renseignements du dictionnaire au premier maillon `etudiant1`.

Pointer aussi vers le 2e étudiant en renseignant l'attribut `suiv` du premier étudiant vers le second.

2.5 La classe Universite

Cette classe est une liste chainée des étudiants. Cette liste ne comporte qu'un seul attribut, `premier`, qui pointe vers le premier étudiant de la chaine.

a. Ecrire le script qui définit cette classe.

b. Ecrire l'instruction qui définit l'instance de la classe `Universite`. On appelera cette instance de classe : `formation`. L'élement de tête de `formation` sera l'`etudiant1`.

c. Cet objet, `formation` hérite des méthodes de la classe `Universite`. L'une de ces méthodes, `effectif` retourne le nombre d'étudiants d'une formation donnée. On appelle cette méthode en plaçant un seul paramètre, le nom de la formation (`str`). Quelle instruction va retourner le nombre d'étudiants de la formation 'L1 informatique' ?

2.5.1 Questions de cours

a. Quel est l'avantage de stocker les étudiants dans une liste chainée plutôt que dans un dictionnaire

b. Quel est l'avantage de la liste chainée par rapport à d'autres structures lineaires, comme le tableau par exemple. Illustrez votre propos avec le cas du depart ou de l'arrivée d'étudiants en cours d'année dans cette université.

Exercice 3

COURS - Listes

3.1 Principe

Une liste chaînée est une structure linéaire, constituée de cellules. Une cellule (ou *maillon* contient :

- une valeur (de n'importe que type),
- et un pointeur qui dirige soit vers la cellule suivante, soit vers `None`, auquel cas la cellule est la *dernière* de la séquence.

Il s'agit d'une structure *linéaire*, mais dans laquelle les éléments n'occupent pas *à priori*, des positions contigües dans la mémoire :

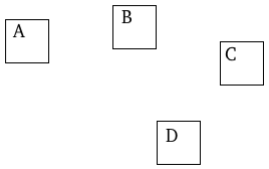

FIGURE 1 – elements dans une liste

Pour relier ces éléments ensembles, dans une même structure de données, il faut alors utiliser des *pointeurs*.

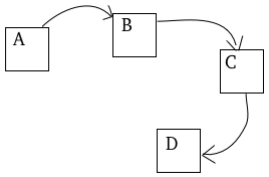

FIGURE 2 – liste chainée grâce aux pointeurs

L'élément **A** pointe sur **B**, qui pointe sur **C**, qui pointe sur **D**. **D** ne pointe sur ... **rien** !

3.2 Représentation d'une liste chainée

Exemple avec 3 éléments contenants les valeurs 'A', 'C', et 'D', mises dans 3 maillons M1, M2 et M3.

- M1 pointe vers M2
- M2 pointe vers M3
- M3 pointe sur ...rien

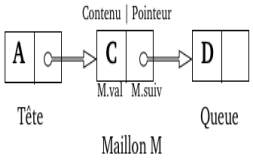

FIGURE 3 – illustration d'une liste chainée

3.3 Interface

On trouve en général les opérations suivantes pour l'interface d'une *Liste* :

- `est_vide` : renvoie `True` si Liste vide
- `taille` : renvoie le nombre de *maillons* de la séquence
- `get_dernier_maillon` :
- `get_maillon_indice` : n'existe pas avec listes python
- `ajouter_debut` : n'existe pas avec listes python
- `inserer_apres` : n'existe pas avec listes python
- `ajouter_fin` :
- `supprimer_apres` : n'existe pas avec listes python
- ...

3.4 Implémentation

L'implémentation est plus naturelle en *programmation orientée objet* (voir le chapitre POO).

Un maillon est une instance de la classe `Maillon`

```
class Maillon:
  def __init__(self):
    self.val = None
    self.suiv = None
```

Une Liste est une instance de la classe `Liste`

```
class Liste:
  def __init__(self):
    self.tete = None
```

Son attribut `tete` est de type `Maillon` , ou bien vaut `None` si la liste est vide.

3.5 Exemple de scripts

3.5.1 Creation d'une liste

```
ma_liste = Liste()
M1, M2, M3 = Maillon(), Maillon(), Maillon()
M1.val = 'A'
M2.val = 'C'
M3.val = 'D'
M1.suiv = M2
M2.suiv = M3
M3.suiv = None
ma_liste.tete = M1
```

3.5.2 Afficher le contenu de l'élément de tête de `ma_liste`

```
print(ma_liste.tete.val)
```

3.5.3 Afficher le 2e element

```
print(ma_liste.tete.suiv.val)
```

3.5.4 Afficher l'élément de rang N :

```
i = 1
M = ma_liste.tete
while i < 27:
  i += 1
  M = M.suiv
print(M.val)
```

3.5.5 Parcourir toute la liste chainée :

```
# script itératif
M = ma_liste.tete
while not M.suiv is None:
  <instruction>
  M = M.suiv
```

```
# script recursif
def parcours(M):
  if M.suiv is None:
    <exit>
  else:
    <instruction>
    parcours(M.suiv)

>>> parcours(ma_liste.tete)
```

3.6 Interêt et inconvenient par rapport aux listes en python

L'interface d'une liste chainée fournit des méthodes plus efficaces que la *Pile*, la *File* ou le *tableau*, lorsque l'on veut par exempke : **insérer**, ou **supprimer** un élément dans la séquence.

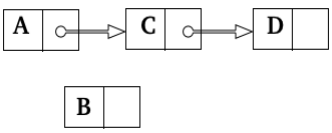

FIGURE 4 – insertion dans une liste

Cette opération est prévue par l'interface d'une liste chainée => O(n) : `inserer_apres(i)`, où `i` est l'indice de l'élément après lequel on veut *insérer*.

Avec une liste python, qui implémente la Pile (voir cours) cela necessite de décaler toutes les valeurs de rang supérieur à `i`. C'est une opération qui est évaluée en $O(n_2)$ pour sa complexité asymptotique.

Il s'agit du même problème lorsque l'on veut *supprimer après i*.

Un autre avantage est la possibilité de faire pointer le dernier élément sur le premier de la liste. On créé ainsi une liste *périodique*.

Inconvénient : Pour accéder à un *maillon* de rang `i` dans une liste chainée, il faut remonter la liste depuis le premier élément (celui de tête), jusqu'à celui de rang `i`. Et cela se fait avec une complexité asymtotique O(n).

3.7 Correction des exercices en ligne

```
# correction de l'exercice 1
class Maillon:
  def __init__(self):
    self.val = None
    self.suiv = None

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

ma_liste = Liste()
M1, M2, M3 = Maillon(), Maillon(), Maillon()
M1.val = 'Premier'
M2.val = 'Troisieme'
M3.val = 'Quatrieme'
M1.suiv = M2
M2.suiv = M3
M3.suiv = None
ma_liste.tete = M1

print(ma_liste.tete.val)
print(ma_liste.tete.suiv.val)
print(ma_liste.tete.suiv.suiv.val)

s = ma_liste.tete
s.val = '1'
print(ma_liste.tete.val)
print(M1.val)
```

```
# Correction de l'exercice 2 en ligne question 2.1
M = L.tete
while not M.suiv is None:
    print(M.val)
    M = M.suiv
print(M.val)
```

Pour le script recursif, on pourra s'aider de l'exemple du script recursif qui ajoute 2.

```
# Correction de l'exercice 1.2 en ligne

def affiche(M):
```

```
    if M.suiv is None:
        return M.val
    else:
        return str(M.val) + '=>' + affiche(M.suiv)


# Correction question 2.3 en ligne
>>> print(affiche(L.tete))
'A=>B=>C=>D'
```

```
# Correction Exercice 3 en ligne
M4 = Maillon()
M4.val='Deuxieme'
M4.suiv = M2
M1.suiv = M4
```