Autotest n°3 – Correction

Exercice 1 : Dans cet exercice, on considère une pile d'entiers positifs. On suppose que les quatre fonctions suivantes ont été programmées préalablement en langage Python :

- `empiler(P, e)` : ajoute l'élément `e` sur la pile `P` ;
- `depiler(P)` : enlève le sommet de la pile `P` et renvoie la valeur de ce sommet ;
- `est_vide(P)` : renvoie `True` si la pile est vide et `False` sinon ;
- `creer_pile()` : renvoie une pile vide.

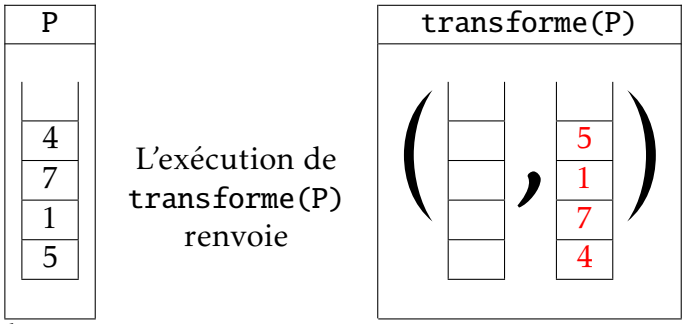
Dans cet exercice, seule l'utilisation de ces quatre fonctions sur la structure de données pile est autorisée.

1) Compléter le schéma ci-dessous en exécutant les appels de fonctions donnés. On écrira ce que renvoie la fonction utilisée dans chaque cas, et on indiquera `None` si la fonction ne renvoie aucune valeur.

	Étape 0 Pile d'origine `P`	Étape 1 `empiler(P, 9)`	Étape 2 `depiler(P)`	Étape 3 `est_vide(P)`
	4 7 1 5	9 4 7 1 5	4 7 1 5	4 7 1 5
Valeur renvoyée		`None`	9	`False`

2) On propose la fonction ci-dessous, qui prend en argument une pile `P` et renvoie un couple de piles :

```
def transforme(P) :
    Q = creer_pile()
    while not est_vide(P) :
        v = depiler(P)
        empiler(Q, v)
    return (P, Q)
```



Compléter sur votre copie le schéma ci-dessus.

3) Écrire une fonction `maximum` en langage Python recevant une pile `P` non vide comme argument et qui renvoie la valeur maximale de cette pile. On ne s'interdit pas qu'après exécution de la fonction, la pile soit vide.

```
def maximum(P):
    maxi = depiler(P)
    while not est_vide(P):
        m = depiler(P)
        if m > maxi:
            maxi = m
    return maxi
```

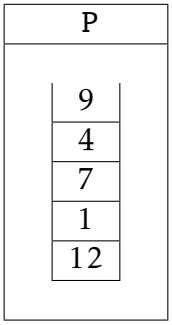
4) On souhaite connaître le nombre d'éléments d'une pile à l'aide de la fonction `taille(P)`. Après l'appel de cette fonction, la pile est dans son état initial.

a) Proposer une stratégie écrite en langage naturel et/ou expliquée à l'aide de schémas, qui permette de mettre en place une telle fonction.

Solution :

On dépile tous les éléments un à un en les comptant et en les plaçant dans une nouvelle pile. On dépile ensuite la nouvelle pile pour tout remettre dans l'ordre dans celle d'origine.

b) Donner le code Python de cette fonction `taille(P)` (on pourra utiliser les cinq fonctions déjà programmées).



`taille(P)` renverra l'entier 5

```
def taille(P):
    compteur = 0
    Q = creer_pile()
    while not est_vide(P):
        empiler(Q, depiler(P))
        compteur = compteur + 1
    while not est_vide(Q):
        empiler(P, depiler(Q))
    return compteur
```

Exercice 2 : Simon souhaite créer en Python le jeu de cartes "la bataille" pour deux joueurs. Les questions qui suivent demandent de reprogrammer quelques fonctions du jeu.

Règles du jeu :

Préparation :

- Distribuer toutes les cartes aux deux joueurs.
- Les joueurs ne prennent pas connaissance de leurs cartes et les laissent en tas face cachée devant eux.

Déroulement :

- À chaque tour, chaque joueur dévoile la carte du haut de son tas.
- Le joueur qui présente la carte ayant la plus haute valeur emporte les deux cartes qu'il place sous son tas.
- Les valeurs des cartes sont : dans l-ordre de la plus forte à la plus faible : As, Roi, Dame, Valet, 10, 9, 8, 7, 6, 5, 4, 3 et 2 (la plus faible)

Si deux cartes sont de même valeur, il y a « bataille ».

- Chaque joueur pose alors une carte face cachée, suivie d'une carte face visible sur la carte dévoilée précédemment.
- On recommence l'opération s'il y a de nouveau une bataille sinon, le joueur ayant la valeur la plus forte emporte tout le tas.

Lorsque l'un des joueurs possède toutes les cartes du jeu, la partie s'arrête et ce dernier gagne.

Pour cela Simon crée une classe Python `Carte`. Chaque instance de la classe a deux attributs : un pour sa `valeur` et un pour sa `couleur`. Il donne au valet la valeur 11, à la dame la valeur 12, au roi la valeur 13 et à l'as la valeur 14. La couleur est une chaine de caractères :

"trefle", "carreau", "coeur" ou "pique".

1) Simon a écrit la classe Python `Carte` suivante, ayant deux attributs `valeur` et `couleur`, et dont le constructeur prend deux arguments : `val` et `coul`.

   a) Compléter les lignes 3 et 4 ci-dessous

```
class Carte:
    def __init__(self, val, coul):
        self.valeur = val
        self.couleur = coul
```

   b) Parmi les propositions ci-dessous quelle instruction permet de créer l'objet "7 de cœur" sous le nom `c7` ?

   - `c7.__init__(self, 7, "coeur")`
   - `c7 = Carte(self, 7, "coeur")`
   - `c7 = Carte(7, "coeur")`
   - `from Carte import 7, "coeur"`

2) On souhaite créer le jeu de cartes. Pour cela, on écrit une fonction `initialiser` sans paramètre qui renvoie une liste de 52 objets de la classe `Carte` représentant les 52 cartes du jeu.
   Compléter les lignes suivantes pour que la fonction réponde à la demande :

```
def initialiser():
    jeu = []
    for coul in ["coeur", "carreau", "trefle", "pique"]:
        for val in range(2, 15):
            carte_cree = Carte(val, coul)
            jeu.append(carte_cree)
    return jeu
```

3) On rappelle que dans une partie de bataille, les deux joueurs tirent chacun une carte du dessus de leur tas, et celui qui tire la carte la plus forte remporte les deux cartes et les place en dessous de son tas.
   Parmi les structures linéaires de données suivantes : Tableau, File, Pile, quelle est celle qui modélise le mieux un tas de cartes dans ce jeu de la bataille ? Justifier votre choix.
   Solution : Puisque on prend les cartes sur le dessus du tas et qu'on les défausse en dessous, les cartes sont tirées dans l'ordre d'arrivé. La première arrivée est la première à sortir. Cela correspond donc à une file.

4) Écrire une fonction `comparer` qui prend en paramètres deux objets de la classe `Carte` : `carte_1`, `carte_2`. Cette fonction renvoie :

   - `0` si la valeur des deux cartes est identique ;
   - 1 si la carte `carte_1` a une valeur strictement plus forte que celle de `carte_2` ;
   - -1 si la carte `carte_2` a une valeur strictement plus forte que celle de `carte_1`.

```
def comparer(carte_1, carte_2):
    if carte_1.valeur > carte_2.valeur:
        return 1
    elif carte_1.valeur < carte_2.valeur:
        return -1
    else:
        return 0
```

Exercice 3 : Les interfaces de structures de données abstraites `Pile` et `File` sont proposées

ci-dessous. On utilisera uniquement les fonctions ci-dessous :

Structure de données abstraite : `Pile`
Utilise : `Elément`, `Booléen`

Opérations :
• `creer_pile_vide()` : $\emptyset \longrightarrow$ `Pile`
`creer_pile_vide()` renvoie une pile vide
• `est_vide()` : `Pile` $\longrightarrow$ `Booléen`
`est_vide(pile)` renvoie **`True`** si `pile` est vide, **`False`** sinon
• `empiler(pile, element)` : `Pile`,`Elément` $\longrightarrow \emptyset$
`empiler(pile, element)` ajoute `element` à la pile `pile`
• `depiler(pile)` : `Pile` $\longrightarrow$ `Elément`
`depiler(pile)` renvoie l'élément au sommet de la `pile` en le retirant de la `pile`

Structure de données abstraite : `File`
Utilise : `Elément`, `Booléen`
Opérations : • `creer_file_vide()` : $\emptyset \longrightarrow$ `File` `creer_file_vide()` renvoie une file vide • `est_vide()` : `File` $\longrightarrow$ `Booléen` `est_vide(file)` renvoie **`True`** si **`file`** est vide, **`False`** sinon • `enfiler(file, element)` : `File`,`Elément` $\longrightarrow \emptyset$ `enfiler(file, element)` ajoute `element` dans la file **`file`** • `defiler(file)` : `File` $\longrightarrow$ `Elément` `defiler(file)` renvoie l'élément en tête de la **`file`** en le retirant de la **`file`**

1) a) On considère la file `F` suivante :

enfilement $\longrightarrow$ `"rouge" "vert" "jaune" "rouge" "jaune"` $\longrightarrow$ défilement

Quel sera le contenu de la pile `P` et de la file `F` après l'exécution du programme Python suivant ?

```
P = creer_pile_vide()
while not(est_vide(F)):
    empiler(P, defiler(F))
```

La file est vide et la pile contient :

P = `"rouge"` / `"vert"` / `"jaune"` / `"rouge"` / `"jaune"`

b) Créer une fonction `taille_file` qui prend en paramètre une file `F` et qui renvoie le nombre d'éléments qu'elle contient. Après appel à cette fonction, la file `F` doit avoir retrouvé son état d'origine.

```
def taille_file(F):
    """File -> Int"""
    compteur = 0
    F2 = creer_file_vide()
    while not est_vide(F):
        enfiler(F2, defiler(F))
        compteur = compteur + 1
    while not est_vide(F2):
        enfiler(F, defiler(F2))
    return compteur
```

2) Écrire une fonction `former_pile` qui prend en paramètre une file `F` et qui renvoie une pile `P` contenant les mêmes éléments que la file.
Le premier élément sorti de la file devra se trouver au sommet de la pile, le deuxième élément sorti de la file devra se trouver juste en-dessous du sommet, etc. **Exemple :**
Si `F` = `"rouge" "vert" "jaune" "rouge" "jaune"`, alors l'appel `former_pile(F)` va renvoyer la pile `P` ci-dessous :

P = | "jaune" |
| "rouge" |
| "jaune" |
| "vert" |
| "rouge" |

```
def former_pile(F):
    P1 = creer_pile_vide()
    while not est_vide(F):
        enpiler(P1, defiler(F))
    P2 = creer_pile_vide()
    while not est_vide(P1):
        enpiler(P2, depiler(P1))
    return P2
```

3) Écrire une fonction `nb_elements` qui prend en paramètres une file `F` et un élément `elt` et qui renvoie le nombre de fois où `elt` est présent dans la file `F`.
Après appel de cette fonction, la file `F` doit avoir retrouvé son état d'origine.

```
def nb_elements(F, elt):
    compteur = 0
    for i in range(taille(F)):
        e = defiler(F)
        if e == elt:
            compteur = compteur + 1
        enfiler(F, e)
    return compteur
```

4) Écrire une fonction `verifier_contenu` qui prend en paramètres une file `F` et trois entiers : `nb_rouge`, `nb_vert` et `nb_jaune`. Cette fonction renvoie le booléen **`True`** si `"rouge"` apparaît au plus `nb_rouge` fois, `"vert"` apparaît au plus `nb_vert` fois et `"jaune"` apparaît au plus `nb_jaune` fois dans `F`. Elle renvoie **`False`** sinon. On pourra utiliser les fonctions précédentes.

```
def verifier_contenu(F, nb_rouge, nb_vert, nb_jaune):
    return nb_elements(F, "rouge") <= nb_rouge and\
           nb_elements(F, "vert") <= nb_vert and\
           nb_elements(F, "jaune") <= nb_jaune
```