TD ipt^2 n° 5: Piles et POO

Principe et applications

Piles

NB: dans tous les exercices qui suivent, seules les fonctions définies dans le type `Pile` sont utilisables.

Exercice n°1: Hauteur d'une pile

On considère une pile `p` non bornée. Ecrire une fonction `hauteur(p)` qui renvoie la hauteur de celle-ci tout en la laissant intacte.

Exercice n°2: Rotation circulaire

Entre autres fonctions, les calculatrices HP sont équipées de la commande `roll(niv)` qui permet de réaliser une rotation circulaire sur la pile d'entrée des données. Concrêtement, cela signifie que l'élément du niveau *niv* est placé au sommet de la pile.

Ecrire une fonction `roll(p,niv)` qui procède à une rotation circulaire du niveau *niv* de la pile au sommet de la pile.

Exercice n°3: Histoire de galanterie

On considère une pile de dossiers sur le bureau d'un employé de la caisse primaire d'assurance maladie. Chaque dossier a été enregistré informatiquement sous forme d'une pile (virtuelle cette fois!) comportant comme informations d'identification un tuple, constitué selon le format `(nom, sexe, date de naissance)`; `nom` est une chaine de caractère, `sexe` est un entier codé avec 1 pour homme et 2 pour femme, et `date de naissance` est codée avec par une chaîne de caractère au format `JJMMAAAA`.

La pile est organisée par ordre décroissant de date de naissance (le plus âgé au sommet, et le plus jeune en fond de pile, sans tenir compte du sexe.).

Ecrire une fonction python `galanterie(Pile)` qui permet de classer les femmes en tête de pile (sommet) et les hommes ensuite, tout en conservant le tri suivant les dates de naissance.

Exercice n°4: Rotations circulaires multiples

Ecrire une fonction `roll(p,n)` qui reçoit en argument une pile `p` et un entier `n` et effectue sur la pile `n` permutations circulaires successives et renvoie la pile ainsi modifiée.

Exercice n°5: Inversion de pile

Écrire une fonction `reverse(p)` qui inverse l'ordre des éléments d'une pile `p` et renvoie le résultat.

Exercice n°6: Copie inversée

Reprendre l'exercice précédent avec cette fois une fonction qui doit générér une pile inversée en préservant la pile initiale.

Exercice n°7: Jeu du "4 à la suite"

Le candidat à un jeu TV doit répondre à une suite de questions. Son score est le nombre maximal de bonnes réponses données à la suite.
Écrire une fonction `jeu(exp)` qui donne le score du candidat à partir d'une expression de longueur quelconque du type `"VFFVVF"` qui indique dans l'ordre chronologique la vérité des réponses du candidat aux questions (V pour vrai et F pour faux). La fonction utilise une pile et empile les bonnes réponses mais dépile toute la pile à la première mauvaise réponse. De plus, elle arrête d'examiner les réponses si le candidat a donné 4 bonnes réponses à la suite.

Exercice n°8: Usage des piles dans les tours d'Hanoï

On souhaite compléter le code récursif de résolution des tours d'Hanoï afin de permettre l'affichage du numéro du disque que l'on déplace d'une tour à l'autre et également l'état de remplissage de chaque tour à tout instant. On propose pour cela d'exploiter des piles.

❶ Proposer une fonction `afficher_pile(p)` permettant l'affichage de la pile `p` en respectant le format suivant:

Contenu de la tour i:
1
2
3
4
5

❷ On propose de matérialiser les tours A,B, et C sous forme de trois piles `pA`, `pB`, et `pC`. Les disques sont initialement empilés sur la tour *A* dans l'ordre de taille décroissante. Si *n* est le nombre de disques à déplacer, on notera *n* la taille du plus grand, $n-1\ldots 1$ celle des suivants. On veut modifier le code récursif des tours de Hanoï (rappelé ci-dessous) afin d'afficher d'une part par le numéro du disque à déplacer d'une tour à l'autre, et également l'état de remplissage des trois tours après chaque déplacement.

Listing 1:

```
def Hanoi(n,init,final,aux)
    if n>0:
        Hanoi(n-1,init,aux,final)
        print(init,"␣vers␣",final)
        Hanoi(n-1,aux,final,init)
```

Premiers pas en Programmation Orientée Objet

NB: on rappelle la synthaxe de déclaration d'une **classe** en python (commande de déclaration et constructeur) :

Listing 2:

```
class NouvelleClasse:
    def __init__(self,arg1,arg2,...):
        self._argappel1,self._argappel2,... = valeur1(arg1,arg2,...),valeur2(arg1,
    arg2,...)
```

ainsi que celle d'une méthode attachée à la classe `NouvelleClasse`:

Listing 3:

```
    def __methode__(self): #attention à l'indentation!!!
        .....
        return ....
```

Exercice n°9: **Premiers pas**

❶ Définir une classe `ClassePoly2` possédant d'une part les attributs suivants: `a` `b` et `c`, et une méthode `affiche_Poly2` contenant un attribut d'instance `x`, permettant l'affichage de la valeur ax^2+bx+c.

En programme principal, instancier un objet de la classe `ClassePoly2` et invoquer la méthode `affiche` avec une valeur d'instanciation de *x* de votre choix.

Exercice n°10: **Définition d'une classe `Vecteur` du plan**

❶ Définir une classe `Vecteur2D` avec un constructeur fournissant les coordonnées par défaut d'un vecteur du plan (par exemple $x=0$ et $y=0$).

Dans le programme principal, instanciez un objet `Vecteur2D` sans paramètre, un objet `Vecteur2D` avec ses deux paramètres et afficher les.

❷ Enrichissez la classe `Vecteur2D` précédente en lui ajoutant une méthode d'affichage et une méthode d'addition de deux vecteurs du plan.

❸ Tester cette classe en instanciant deux vecteurs `Vecteur2D`, afficher-les (sous forme de listes contenant les deux composantes), et afficher le vecteur somme (là-encore sous forme d'une liste de deux composantes).