

TP d'Algorithmique et structures de données

Python : notions de base et itérateurs

Équipe pédagogique Algo SD

Informations utiles :

1. Écrire des itérateurs : `yield`
2. Itérateurs utiles : `itertools`
3. Fonctions avec des arguments arbitraires

1 Itérateurs

1. Écrire un itérateur `repeter` qui, étant donné un itérable et un nombre `k`, renvoie `k` répétitions de chaque élément de l'itérable.

```
def repeter(iterable, k):  
    """ renvoie chaque element de l'iterable repete k fois """
```

Sorties attendues.

```
repeter([1, 2, 3], 3) -> 1, 1, 1, 2, 2, 2, 3, 3, 3  
repeter(range(1, 7, 2), 2) -> 1, 1, 3, 3, 5, 5
```

2. Maintenant, utilisez `itertools.chain` et `itertools.repeat` (voir information utile 2) pour écrire le même itérateur qu'en question 1. Puisqu'on ne connaît pas le nombre d'éléments de l'itérateur donné, on passe à `chain` un itérable de morceaux de `k` répétitions d'éléments de l'itérable (voir listes d'arguments).

```
from itertools import chain, repeat  
  
def repeter2(iterable, k):  
    """ renvoie un itérateur qui repete chaque element de l'iterable donne k fois """
```

Sorties attendues.

Les mêmes qu'en question 1.

3. Écrire un itérateur `paires` qui, étant donné un itérable, renvoie des paires d'éléments successifs de l'itérable. Pour obtenir un itérateur à partir d'un itérable on utilise `iterateur = iter(iterable)`. Pour avancer, on utilise `next(iterateur)`. À la fin, l'itérateur lève l'exception `StopIteration` (voir le CM2).

```
def paires(iterable):  
    """ renvoie des paires d'elements successifs de l'iterable """
```

Sorties attendues.

```
paires([1, 2, 3, 4, 5]) -> (1, 2), (2, 3), (3, 4), (4, 5)
```

4. Écrire un itérateur, qui vous permet de jeter un coup d'œil ("peek") à la valeur suivante d'un itérable sans avancer l'itération. À cette fin, créez une classe `IterateurPeekable`, qui aura la structure suivante.

```

class IterateurPeekable:
    def __init__(self, iterable):
        """ initialisation """

    def __iter__(self):
        return self

    def __next__(self):
        """ avance l'itérateur et renvoie l'element suivant """

    def peek(self):
        """ renvoie l'element suivant de l'iterateur s'il existe et None sinon """
        return None if self.fini else self.suivant

    def avance(self):
        """ stocker le element suivant de l'iterateur dans self.suivant. """

```

Précisions. Il faut stocker le prochain élément de l'itérateur dans la classe et le renvoyer lors d'un appel de `peek`. Pour anticiper la fin de l'itération dans `avance`, utilisez `try/except StopIteration` (voir le CM2).

Sorties attendues.

```

I = IterateurPeekable(range(4))
for i in I:
    print('valeur courante :', i, ', valeur suivante :', I.peek())

```

```

valeur courante : 0 , valeur suivante : 1
valeur courante : 1 , valeur suivante : 2
valeur courante : 2 , valeur suivante : 3
valeur courante : 3 , valeur suivante : None

```

- Utilisez la classe `IterateurPeekable` pour écrire un itérateur `fusion`, qui fusionne deux séquences triées données par deux itérables en une séquence triée.

```

def fusion(iterable1, iterable2):
    """fusionne iterable1 et iterable2 en une sequence triee"""

```

Sorties attendues.

```

fusion(range(0, 6, 2), range(1, 6, 2)) -> 0, 1, 2, 3, 4, 5
fusion(range(0, 6, 2), range(0, 6, 2)) -> 0, 0, 2, 2, 4, 4

```

- Reprendre le TP 1 et intégrer les itérateurs dans les méthodes `retourne` et `verifications_invariants`.

2 Dictionnaires

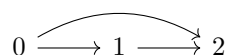
Un *graphe orienté* est une paire (V, A) d'un ensemble V de sommets et un ensemble A d'arcs. Chaque arc $v \longrightarrow w \in A$ est une paire ordonnée (v, w) . On va utiliser des dictionnaires pour représenter des graphes orientés en python de la manière suivante. Pour chaque sommet $v \in V$, on stocke dans une liste ses voisins sortants. Par exemple, le dictionnaire

```

>>> G = { 0 : [1, 2], 1 : [2], 2 : [] }

```

représente le graphe



Vocabulaire Soit $G = (V, A)$ un graphe orienté.

- Pour un arc $v \longrightarrow w \in A$, le sommet v est son *extrémité initiale* et w est son *extrémité terminale*.
- Le *degré entrant* (resp., *degré sortant*) d'un sommet $v \in V$ est le nombre d'arcs de G ayant v comme extrémité terminale (resp., initiale).
- Le graphe G est *symétrique* si $(v, w) \in A$ implique $(w, v) \in A$.
- La *matrice d'adjacence* de G est la matrice $A(G) = (a_{v,w})_{v,w \in V}$, où

$$a_{v,w} = \begin{cases} 1 & \text{si } (v,w) \in A \\ 0 & \text{sinon .} \end{cases}$$

1. Écrire une fonction `out_degree(G, v)`, qui renvoie le degré sortant du sommet v de G .

Sorties attendues.

```
out_degree({ 0 : [1, 2], 1 : [2], 2 : [] }, 0) -> 2
out_degree({ 0 : [1, 2], 1 : [2], 2 : [] }, 1) -> 1
out_degree({ 0 : [1, 2], 1 : [2], 2 : [] }, 2) -> 0
```

2. Écrire une fonction `in_degree(G, v)`, qui renvoie le degré entrant du sommet v de G . **Sorties attendues.**

```
out_degree({ 0 : [1, 2], 1 : [2], 2 : [] }, 0) -> 0
out_degree({ 0 : [1, 2], 1 : [2], 2 : [] }, 1) -> 1
out_degree({ 0 : [1, 2], 1 : [2], 2 : [] }, 2) -> 2
```

3. Écrire une fonction `arcs(G)`, qui renvoie un itérateur sur les arcs de G . **Sortie attendue.**

```
arcs({ 0 : [1, 2], 1 : [2], 2 : [] }) -> (0, 1), (0, 2), (1, 2)
```

4. Écrire une fonction `is_symmetric(G)`, qui renvoie `True` si le graphe G est symétrique et `False` sinon.

5. Écrire une fonction `make_symmetric(G)`, qui renvoie le graphe H obtenu à partir de G en ajoutant pour tout arc (u, v) de G l'arc (v, u) s'il n'est pas présent.

Sortie attendue.

```
make_symmetric({ 0 : [1, 2], 1 : [2], 2 : [] }) -> { 0 : [1, 2], 1 : [0, 2], 2 : [0, 1] }
```