Grille en art ASCII (3)

Codez une fonction nommée `grille_ASCII` qui accepte en entrée un ensemble de tuples (i, j) et qui **retourne** en sortie une chaîne de caractères représentant une grille correspondante. Dans cette grille, les cases **vides** sont représentées par des `.` et les cases **pleines** par des `+`. L'ensemble des tuples reçus en argument spécifie les coordonnées des cases pleines; toutes les autres cases étant vides.

La dimension de la grille est déterminée par les valeurs minimum et maximum des indices i et j de l'ensemble des tuples, où le couple $(i_{\min}, j_{\min})$ correspondent au coin **inférieur gauche** de la grille, alors que le couple $(i_{\max}, j_{\max})$ correspondent au coin **supérieur droit**.

Votre fonction doit **retourner** une chaîne de caractères qui représente la grille en *art ASCII*. Par exemple :

```
grille_ASCII({(-1, 1), (1, 0), (1, -2), (2, 3)})
```

doit produire la chaîne suivante:

```
'.....+\n+.+...\n......\n...+..\n'
```

qui lorsqu'affichée avec `print` produira:

```
.....+
+.+...
......
...+..
```

Dans cet exemple, nous avons $(i_{\min}, j_{\min}) = (-1, -2)$ et $(i_{\max}, j_{\max}) = (2, 3)$.

INDICE: commencez par déterminer $(i_{\min}, j_{\min})$ et $(i_{\max}, j_{\max})$, puis initialisez une variable d'accumulation et bouclez sur les $i = i_{\max}, \ldots, i_{\min}$, et les $j = j_{\min}, \ldots, j_{\max}$, en ajoutant un `+` à la variable d'accumulation lorsque (i, j) est une case pleine et `.` autrement.

Solution de référence

Notez qu'une solution n'est généralement **pas** unique.

In [ ]:

```python
def grille_ASCII(cases):
    # déterminer les indices minimum et maximum
    i_min = min([i for i, _ in cases])
    j_min = min([j for _, j in cases])
    i_max = max([i for i, _ in cases])
    j_max = max([j for _, j in cases])

    # initialiser une variable d'accumulation
    result = ''

    # pour chaque ligne
    for i in range(i_max, i_min-1, -1):
        # et chaque colonne
        for j in range(j_min, j_max+1):
            if (i, j) in cases:
                # la case est pleine
                result += '+'

            else:
                # la case est vide
                result += '.'

        # ajouter un retour à la ligne
        result += '\n'

    return result
```

Classe Item

Écrivez une classe nommée `Item` qui encapsule différentes propriétés d'un item. Votre classe doit définir les trois méthodes suivantes :

1. Un constructeur qui accepte dans l'ordre les **trois** (3) arguments suivants : le **nom** de l'item, son **volume** et son **poids**.
2. Une méthode de conversion en chaîne de caractères qui produit le format suivant :
   ```
   item nom: (volume, poids)
   ```
   où `nom` désigne le nom de l'item, `volume` son volume et `poids` son poids.
3. Une méthode `propriété` qui accepte en argument l'une des valeurs suivantes : `'nom'`, `'volume'` ou `'poids'`; et qui retourne la valeur de la propriété correspondante. Dans le cas où cette méthode reçoit un argument inconnu, elle doit soulever une exception de type `ValueError`.

Assurez-vous dans votre constructeur de valider vos trois arguments en soulevant une exception de type `TypeError` pour les cas suivants :

- le nom de l'item n'est pas une chaîne de caractères ;
- les propriétés de volume et de poids ne sont pas des entiers ou des nombres à virgule flottante.

Soulevez également une exception de type `ValueError` si le volume ou le poids n'est pas strictement positif (> 0).

Solution de référence

Notez qu'une solution n'est généralement **pas** unique.

In [ ]:

```python
class Item:
    def __init__(self, nom, volume, poids):
        if not isinstance(nom, str):
            raise TypeError("Le nom de l'item doit être de type str")

        elif not isinstance(volume, (int, float)):
            raise TypeError("Le volume de l'item doit être de type int ou float")

        elif not isinstance(poids, (int, float)):
            raise TypeError("Le poids de l'item doit être de type int ou float")

        elif volume <= 0:
            raise ValueError("Le volume de l'item doit être strictement supérieur à zéro.")

        elif poids <= 0:
            raise ValueError("Le poids de l'item doit être strictement supérieur à zéro.")

        self._nom = nom
        self._volume = volume
        self._poids = poids

    def __str__(self):
        return f'item {self._nom}: ({self._volume}, {self._poids})'

    def propriété(self, x):
        if x == 'nom':
            valeur = self._nom

        elif x == 'volume':
            valeur = self._volume

        elif x == 'poids':
            valeur = self._poids

        else:
            raise ValueError("Vous devez spécifier une propriété de 'nom', 'volume' ou 'poids'")

        return valeur
```

Classe Valise

Écrivez une classe nommée `Valise` qui encapsule un valise pour transporter des items.

Votre classe doit supporter les méthodes suivantes :

1. Un constructeur qui accepte en argument un **itérable** d'items qui possèdent une méthode nommée `propriété`, comme celle de la classe `Item` de l'exercice précédent. Cette méthode accepte notamment en argument des valeurs `'volume'` et `'poids'` et retourne le volume ou le poids de l'item en question.
2. Une méthode de conversion en chaîne de caractères qui énumère les items contenus dans la valise à raison d'un item par ligne. Vous pouvez supposer que les items savent comment se convertir en chaîne de caractères. Assurez-vous d'énumérer les items dans l'**ordre** d'insertion.
3. Une méthode nommée `ajouter` qui permet d'ajouter un nouvel item à la valise.
4. Une méthode nommée `volume` qui permet de retourner le volume total des items de la valise.
5. Une méthode nommée `poids` qui permet de retourner le poids total des items de la valise.

Ajoutez à votre constructeur un deuxième argument permettant de spécifier la capacité **maximale** de la valise en terme de volume et de poids. Lorsque spécifié, cet argument doit prendre la forme d'un couple (volume, poids). Lorsqu'omis, il doit avoir la valeur par défaut `None`. Une valeur `None` signifie une capacité infinie, autant en volume qu'en poids.

Lors de la construction de votre valise, assurez-vous de ne **jamais** dépasser sa capacité et, le cas échéant, soulevez une exception de type `RuntimeError` pour signaler le problème. Idem, lors de l'ajout d'un nouvel item dans la valise.

Solution de référence

Notez qu'une solution n'est généralement **pas** unique.

In [ ]:

```python
class Valise:
    def __init__(self, items, capacité=None):
        # initialiser une liste d'items
        self._items = []

        # mémoriser la capacité de la valise
        self._capacité = capacité

        # ajouter tous les item de l'itérable
        for item in items:
            self.ajouter(item)

    def __str__(self):
        return '\n'.join(str(item) for item in self._items) + '\n'

    def ajouter(self, item):
        # ajouter le nouvel item
        self._items.append(item)

        if self._capacité is not None:
            if self.volume() > self._capacité[0] or self.poids() > self._capacité[1]:
                # retirer le nouvel item
                del self._items[-1]

                # soulever une exception
                raise RuntimeError('La valise est pleine!')

    def volume(self):
        return sum(item.propriété('volume') for item in self._items)

    def poids(self):
        return sum(item.propriété('poids') for item in self._items)
```

Convolution d'une image (3)

Une image est simplement une grande **matrice** de pixels. Pour filtrer une image (avec par exemple les filtres contenus dans photoshop), il s'agit essentiellement de **convoluer** l'image avec un noyau (parfois avec plusieurs), c'est-à-dire avec de petites matrices de coefficients qui viennent pondérer les valeurs de chaque pixel en effectuant une combinaison linéaire de ceux qui sont adjacents. De cette manière, par exemple, on peut rendre une image plus floue ou, au contraire, la rendre plus contrastée. Tout dépend des coefficients du **noyau** utilisé par la convolution.

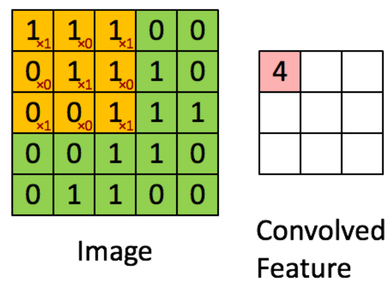
Dans le contexte d'une image I possédant m lignes et n colonnes, donc une matrice $m \times n$, et d'un noyau K de dimensions $o \times p$, la convolution $I^* = K * I$ du noyau K par l'image I est donnée par la formule suivante pour chacun des pixels (i, j) de l'image résultante:

$$I^*(i,j) = \sum_{k=-a}^{a} \sum_{l=-b}^{b} K(a+k, b+l) \times I(i+k, j+l), \quad \text{pour} \quad a \leq i \leq m-a-1 \quad \text{et} \quad -b \leq j \leq n-b-1$$

où $(a, b) = (\lfloor o/2 \rfloor, \lfloor p/2 \rfloor)$ correspond à la coordonnée du centre du noyau, avec la notation $\lfloor x \rfloor$ désignant le plus grand entier plus petit ou égal à x (la division entière qui tronque la partie fractionnaire). On suppose ici que les dimensions d'un noyau sont toujours impaires.

La figure ci-contre anime la convolution du noyau en jaune avec l'image en vert pour:

$$I = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}, \quad \text{et} \quad K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix},$$



et montre le résultat en rose, pixel par pixel. La convolution itère donc sur les pixels de l'image en plaçant au-dessus de chacun d'eux le noyau et en multpliant puis sommant les éléments correspondants. Notez que pour un noyau 3×3, l'image convoluée **perdra** un (1) pixel sur son pourtour par rapport à l'image de départ. Dans le cas général, elle perdre a pixels le long de l'axe horizontal et b pixels le long de l'axe vertical.

On vous demande de coder une fonction nommée `convoluer_image` qui accepte dans l'ordre les arguments suivants :

1. Une image à convoluer I ;
2. Un noyau de convolution K ;

tous les deux sous la forme d'un tableau numpy à deux dimensions (voir leçon 12.1), et de **retourner** en sortie l'image résultante sous la forme d'un autre tableau `numpy` de la même dimension que l'image d'entrée, mais en comblant le pourtour avec des zéros. Ainsi, le résultat de l'exemple précédent serait :

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 3 & 4 & 0 \\ 0 & 2 & 4 & 3 & 0 \\ 0 & 2 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

En utilisant des énoncés `assert`, assurez-vous dans votre fonction que les arguments reçus sont **bien** des instances de tableaux numpy (classe `ndarray` ; utiliser la fonction `isinstance`) et que les dimensions du noyau sont **bien** impaires.

INDICES - Pour résoudre ce problème:

1. Déterminer les valeurs de a et b à partir des dimensions du noyau (attribut `shape`).
2. Initialiser une matrice résultat remplie de zéros (fonction `zeros`), de la même taille que l'image d'entrée.
3. Pour chaque ligne $i \in [a, m-a[$:
   A. Et pour chaque colonne $j \in [b, n-b[$:
      a. Découper dans l'image d'entrée l'empreinte du noyau en utilisant l'opérateur `[]` de `ndarray`.
      b. Multiplier le noyau par son empreinte dans l'image en utilisant l'opérateur `*` de `ndarray`.

c. Calculer la somme des résultats de la multiplication en utilisant la méthode `ndarray.sum`.

4. Retourner la matrice résultat.

Notez que ces étapes sont très simples et courtes si vous profitez **bien** des fonctionnalités de numpy.

Contexte du problème

In [ ]:

```
import numpy as np

# image de test
image = np.array(
    [1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]
).reshape(5, 5)

# noyau de test
noyau = np.array([1, 0, 1, 0, 1, 0, 1, 0, 1]).reshape(3, 3)
```

Solution de référence

Notez qu'une solution n'est généralement **pas** unique.

In [ ]:

```
def convoluer_image(image, noyau):
    # vérifier que les arguments sont bien des tableaux numpy
    assert isinstance(image, np.ndarray) and isinstance(noyau, np.ndarray)

    # déterminer les dimensions de l'image et du noyau
    m, n = image.shape
    o, p = noyau.shape

    # vérifier que les dimensions du noyau sont bien impaires
    assert o % 2 and p % 2, "noyau invalide"

    # déterminer les paramètres de découpage
    a, b = o//2, p//2

    # initialiser la matrice résultat avec des zéros
    result = np.zeros(image.shape)

    # boucler sur les lignes de la matrice résultat
    for i in range(a, m-a):
        # boucler sur les colonnes de la matrice résultat
        for j in range(b, n-b):
            # calculer la valeur du pixel $(i,j)$
            result[i, j] = (image[i-a:i+a+1, j-b:j+b+1] * noyau).sum()

    # retourner la matrice résultat
    return result
```