Grille en art ascii

Codez une fonction nommée `grille_art_ascii` qui accepte en entrée trois arguments:

1. Le nombre de lignes de la grille.
2. Le nombre de colonnes de la grille.
3. l'ensemble des tuples (i, j) des cases actives de la grille.

Pour les coordonnées des cases de la grille, nous adopterons la convention habituelle du Python, à savoir que les indices débutent **toujours** à zéro, et nous placerons l'origine de la grille en haut à gauche.

Votre fonction doit **retourner** une chaîne de caractères qui représente la grille en «*art ascii*» selon le format suivant. Les cases actives y sont représentées par des plus (`+`) et les autres cases par des points (`.`). Par exemple, l'expression suivante:

```
grille_art_ascii(3, 3, {(0, 1), (1, 0), (1, 2), (2, 1)})
```

doit produire la chaîne suivante:

```
' -----\n|. + .|\n|+ . +|\n|. + .|\n -----'
```

qui lorsqu'affichée avec `print` produira:

```
 -----
|. + .|
|+ . +|
|. + .|
 -----
```

Notez qu'un **espace** est inséré entre les cases d'une **même** ligne afin d'équilibrer les espacements horizontal et vertical. Testez **bien** votre solution avec les tuples d'arguments définis dans le **contexte** de cet exercice. Par exemple, en exécutant:

```
configurations = [tob, blinker, toad, beacon]
for config in configurations:
    print(grille_art_ascii(*config))
```

Indice: pour tester l'appartenance d'un objet à un ensemble, utilisez simplement l'opérateur `in` dans une expression booléenne. Par exemple, l'expression `objet in ensemble` vous dira si oui ou non l'objet `objet` appartient à l'ensemble `ensemble`.

Contexte de l'exercice

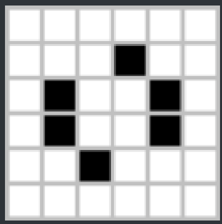
```
# différentes configurations pour faire vos tests
tob = (3, 3, {(0, 1), (1, 0), (1, 2), (2, 1)})
blinker = (3, 3, {(0,1), (1, 1), (2, 1)})
toad = (4, 4, {(1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2)})
beacon = (4, 4, {(0, 0), (0, 1), (1, 0), (1, 1), (2, 2), (2, 3), (3, 2), (3, 3)})
```

Solution du professeur

Notez que cette solution n'est généralement **pas** unique.

```python
def grille_art_ascii(m, n, grille):
    en_tête = f" {'-'*(2*n-1)}"

    # insérer l'en-tête devant la première ligne
    lignes = [en_tête]

    # pour chaque ligne
    for i in range(m):
        cases = []
        # et chaque colonne
        for j in range(n):
            if (i, j) in grille:
                # la case est vivante
                cases.append('+')

            else:
                # la case est morte
                cases.append('.')

        # joindre les colonnes et ajouter une nouvelle ligne
        lignes.append(f"|{' '.join(cases)}|")

    # insérer l'en-tête à la suite de la dernière ligne
    lignes.append(en_tête)

    # retourner la jointure des différentes lignes
    return '\n'.join(lignes)
```

Jeu de la vie (fonction)



Le jeu de la vie est un jeu de simulation mathématique **sans** joueur explicite, où des cellules vivantes évoluent de génération en génération. Il s'agit d'un automate qui joue de lui-même. Le jeu est habituellement représenté sur une grille 2D dont les dimensions sont a priori infinies. Chaque case de la grille peut héberger une cellule, auquel cas on dit que la case correspondante est **vivante**. Sinon, la case est **morte**. L'état d'une case à la **génération** suivante dépend de son état actuel ainsi que de l'état de ses huit cases adjacentes.

Les règles du jeu sont les suivantes:

1. Toute case **vivante** entourée d'exactement deux (2) ou trois (3) cases **vivantes** va survivre à la génération suivante.
2. Toute case **morte** entourée d'exactement trois (3) cases **vivantes** va renaître à la génération suivante.
3. Dans **tous** les autres cas, la case sera **morte** à la génération suivante.

Par exemple, voici une grille 6×6 qui selon ces règles produira une oscillation de génération en génération:

```
. . . . . .     . . . . . .     . . . . . .
. . . . . .     . . . + . .     . . . . . .
. . + + + . --> . + . . + . --> . . + + + . --> etc.
. + + + . .     . + . . + .     . + + + . .
. . . . . .     . . + . . .     . . . . . .
. . . . . .     . . . . . .     . . . . . .
```

où les symboles `+` et `.` représentent respectivement les cases vivantes et mortes.

On vous demande de coder une fonction nommée `jeu_de_vie` qui accepte en entrée un [ensemble]() de tuples (i, j) représentant les coordonnées des cases **vivantes** actuelles, et qui **retourne** en sortie l'ensemble des cases vivantes pour la **génération** suivante dans la séquence du jeu.

Testez **bien** votre solution avec les ensembles définies dans le **contexte** de cet exercice. Par exemple, la boucle:

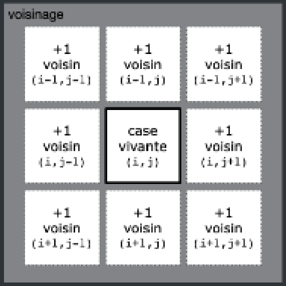
```
for config in [tob, blinker, toad, beacon]:
    print(grille_suivante(config))
```

devrait produire les ensembles suivants:

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

Notez que l'ordre des éléments d'un ensemble est **imprévisible** et sans importance.

Indices pour résoudre ce problème:



1. Initialiser un **dictionnaire** vide afin de pouvoir compter les nombres de voisins de chaque case potentielle.
2. Pour chaque case (i, j) de l'**ensemble** actuel des cases vivantes (voir figure ci-contre):
   A. Créez la liste des huit (8) cases voisines.
   B. Pour **chacune** de ces cases voisines:
      a. Ajoutez un (+1) à la case correspondante dans le **dictionnaire** de voisinage.
3. Parcourir les éléments du **dictionnaire** de voisinage en ne retenant que les cases qui respectent les règles du jeu quant au nombre de voisins.
4. Retourner l'ensemble de ces cases vivantes pour la nouvelle génération.

Notez que dans le contexte d'un énoncé `if`, l'opérateur `in` peut servir à tester si une clé **appartient** à un dictionnaire ou à un ensemble. Dans le contexte d'un énoncé `for`, ce même opérateur permet **d'itérer** sur les clés d'un dictionnaire, ou sur les éléments d'un ensemble. Faites-en bon usage!

Contexte de l'exercice

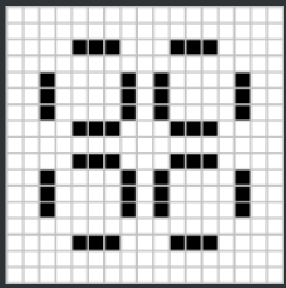
```
tob = {(0, 1), (1, 0), (1, 2), (2, 1)}
blinker = {(0,1), (1, 1), (2, 1)}
toad = {(1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2)}
beacon = {(0, 0), (0, 1), (1, 0), (1, 1), (2, 2), (2, 3), (3, 2), (3, 3)}
```

Solution du professeur

Notez que cette solution n'est généralement **pas** unique.

```
def jeu_de_vie(cases):
    # initialiser le dictionnaire de voisinage
    voisinage = {}

    # pour chaque case vivante
    for case in cases:
        i, j = case
        # créer la liste des huit cases voisines
        voisins = [
            (i-1, j-1),
            (i, j-1),
            (i+1, j-1),
            (i-1, j),
            (i+1, j),
            (i-1, j+1),
            (i, j+1),
            (i+1, j+1)
        ]

        # ajouter un (1) à chaque voisin
        for voisin in voisins:
            voisinage[voisin] = voisinage.get(voisin, 0) + 1

    # retourner les cases du voisinage qui respectent les règles du jeu
    return {
        case for case in voisinage
        if case in cases and 2 <= voisinage[case] <= 3 or voisinage[case] == 3
    }
```

Jeu de la vie (classe)



Le jeu de la vie est un jeu de simulation mathématique **sans** joueur explicite, où des cellules vivantes évoluent de génération en génération. Il s'agit d'un automate qui joue de lui-même. Le jeu est représenté sur une grille 2D dont les dimensions sont a priori infinies. Chaque case de la grille peut héberger une cellule, auquel cas on dit que la case correspondante est **vivante**. Sinon, la case est **morte**. L'état d'une case à la **génération** suivante dépend de son état actuel ainsi que de l'état de ses huit (8) cases adjacentes.

Les règles du jeu sont les suivantes:

1. Toute case **vivante** entourée d'exactement deux (2) ou trois (3) cases **vivantes** va survivre à la génération suivante;
2. Toute case **morte** entourée d'exactement trois (3) cases vivantes va renaître à la génération suivante;
3. Dans **tous** les autres cas, la case sera **morte** à la génération suivante.

On vous demande de coder une classe nommée `JeuDeVie` pour encapsuler ce jeu. Votre classe doit être **composée** (ne **pas** utiliser l'héritage) d'un ensemble de couples (i, j) représentant les **coordonnées** des cellules actuellement vivantes. Son interface doit posséder les méthodes suivantes:

1. Un constructeur qui accepte trois (3) arguments:
   A. le nombre de lignes de la grille;
   B. le nombre de colonnes de la grille;
   C. l'ensemble des couples (i, j) spécifiant les coordonnées des cases initialement vivantes de la grille.
2. Une méthode (`__str__`) pour convertir l'état interne du jeu en chaîne de caractères (grille).
3. Les méthodes nécessaires pour rendre **itérable** les instances de la classe (méthodes `__iter__` et `__next__`; voir la leçon 6.1).
4. Une méthode nommée `cases_vivantes` qui retourne l'ensemble **actuel** des cases vivantes.

Pour les coordonnées des cases, nous adopterons la convention habituelle du Python, à savoir que les indices débutent **toujours** à zéro, et nous placerons l'origine de la grille en haut à gauche.

Pour la conversion de l'état interne du jeu en chaîne de caractères, nous utiliserons le caractère `+` pour désigner les cases vivantes et le `.` pour les cases mortes. Sur chaque ligne, on insérera un espace entre chaque symbole afin d'égaliser autant que possible l'espacement entre les lignes et les colonnes.

Par exemple, le code suivant:

```python
config = (5, 5, {(1, 2), (3, 2), (0, 0), (1, 3), (2, 3), (2, 2), (4, 2), (1, 0), (4, 1), (1, 1)})

# créer une instance de jeu à partir de la config
jeu = JeuDeVie(*config)

# afficher la grille initiale
print(jeu)

# afficher les grilles de la séquence de jeu
for i, grille in enumerate(jeu):
    print(grille)
    if i >= 5:
        # sortir de la boucle après un maximum de 5 générations
        break
```

devra produire la sortie suivante:

```
 ---------
|+ . . . .|
|+ + + + .|
|. . + + .|
|. . + . .|
|. + + . .|
 ---------
 ---------
|+ . + . .|
|+ . . + .|
|. . . . .|
|. . . . .|
|. + + . .|
 ---------
 ---------
|. + . . .|
|. + . . .|
|. . . . .|
|. . . . .|
|. . . . .|
 ---------
 ---------
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
|. . . . .|
 ---------
```

Indices pour résoudre ce problème:

1. Dans votre constructeur, faites une **copie** de l'ensemble reçu en argument et affectez cette copie à une variable d'instance. Pour ce faire, utilisez simplement le constructeur d'ensemble: `copie = set(arg)` où `arg` est l'ensemble à copier.
2. Dans votre méthode `__iter__` retournez simplement l'intance de jeu (`self`).
3. Dans votre méthode `__next__`:
   A. Soulevez une exception de type `StopIteration` si aucune case vivante n'existe.
   B. Appliquez les règles du jeu afin de mettre à jour l'ensemble des cases vivantes de l'instance.
   C. Retourner l'instance de jeu (`self`).

Notez que dans votre solution, vous pouvez **faire appel** aux fonctions des deux exercices précédents, à savoir:

1. `grille_art_ascii`
2. et `jeu_de_vie`

Lors de la correction, ces deux fonctions seront **automatiquement** ajoutées à votre solution à condition que vous ne les définissiez **pas** vous-même dans votre cellule de solution. Pour tester votre solution avant de la soumettre, collez plutôt vos fonctions dans votre cellule de test.

Le **contexte** de cet exercice contient diverses configurations de jeu. Utilisez-les pour **bien** tester votre solution.

Contexte de l'exercice

```
# configurations de test
tob = (3, 3, {(0, 1), (1, 0), (1, 2), (2, 1)})
blinker = (3, 3, {(0,1), (1, 1), (2, 1)})
toad = (4, 4, {(1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2)})
beacon = (4, 4, {(0, 0), (0, 1), (1, 0), (1, 1), (2, 2), (2, 3), (3, 2), (3, 3)})
```

Solution du professeur ¶

Notez que cette solution n'est généralement **pas** unique.

```
class JeuDeVie:
    def __init__(self, m, n, cases):
        # mémoriser les dimensions de la grille
        self.m = m
        self.n = n

        # mémoriser l'ensemble initial des cases vivantes
        self.cases = cases

    def __str__(self):
        en_tête = f" {'-'*(2*self.n-1)}"

        # insérer l'en-tête devant la première ligne
        lignes = [en_tête]

        # pour chaque ligne
        for i in range(self.m):
            cases = []
            # et chaque colonne
            for j in range(self.n):
                if (i, j) in self.cases:
                    # la case est vivante
                    cases.append('+')

                else:
                    # la case est morte
                    cases.append('.')

            # joindre les colonnes et ajouter une nouvelle ligne
            lignes.append(f"|{' '.join(cases)}|")

        # insérer l'en-tête à la suite de la dernière ligne
        lignes.append(en_tête)

        # retourner la grille complète
        return '\n'.join(lignes)

    def __iter__(self):
        # retourner self, car celui-ci implante lui-même la méthode next
        return self
```

```python

    def __next__(self):
        # s'il n'y a plus de cellule vivante, stopper l'itération
        if len(self.cases) == 0:
            raise StopIteration()

        # initialiser le dictionnaire des nombres de voisins
        voisinage = {}

        # pour toutes les cases actuellement vivantes
        for case in self.cases:
            i, j = case
            # créer la liste des huit voisins adjacents
            voisins = [
                (i-1, j-1),
                (i, j-1),
                (i+1, j-1),
                (i-1, j),
                (i+1, j),
                (i-1, j+1),
                (i, j+1),
                (i+1, j+1)
            ]

            # ajouter un (1) à chaque voisin
            for voisin in voisins:
                voisinage[voisin] = voisinage.get(voisin, 0) + 1

        # mettre à jour l'ensemble des cases vivantes en appliquant les règles du jeu
        self.cases = {
            case for case in voisinage
            if case in self.cases and 2 <= voisinage[case] <= 3 or voisinage[case] == 3
        }

        # retourner un référence sur le jeu dans son nouvel état
        return self

    def cases_vivantes(self):
        # retourner l'ensemble actuel des cases vivantes
        return self.cases
```

Convolution d'une image

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.

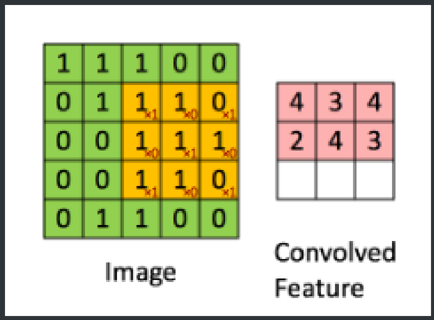
Dans le contexte d'une image I possédant m lignes et n colonnes, donc une matrice $m \times n$, et d'un noyau K de dimensions $o \times p$, la convolution $I^* = K * I$ du noyau K par l'image I est donnée par la formule suivante pour chacun des pixels (i, j) de l'image résultante:

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

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:

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



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` qui accepte en argument:

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

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 de l'exercice

```python
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 du professeur

Notez que cette solution n'est généralement **pas** unique.

```python
def convoluer(noyau, image):
    # 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
```