MOOC Python 3

Session 2018

Corrigés de la semaine 7

checkers - Semaine 7 Séquence 05

```
def checkers(size, corner_0_0=True):
    """
    Un damier
    le coin (0, 0) vaut 1 ou 0 selon corner_0_0
    se souvenir que False == 0 et True == 1

    credits: JeF29 pour avoir suggéré une simple
    addition plutôt qu'un xor
    """
    # on peut voir le damier comme une fonction sur
    # les coordonnées, du genre (i + j) % 2
    # pour choisir le coin, on ajoute avant de faire le % 2
    I, J = np.indices((size, size))
    return (I + J + corner_0_0) % 2
```

checkers_2 - Semaine 7 Séquence 05

```
def checkers_2(size, corner_0_0=True):
    """
    sur une ligne, avec
    * sum() pour l'addition I + J

    et, pour les illustrer un petit, les opérateurs bit-wise:
    * et logique (&) pour le modulo 2
    * et xor (^) pour inverser

    credits: j4l4y
    """
    # avec sum() sur indices()
    # on peut tout faire en une ligne:
    return sum(np.indices((size, size))) & 1 ^ corner_0_0
```

checkers_3 - Semaine 7 Séquence 05

```
def checkers_3(size, corner_0_0=True):
    """
    Une autre approche complètement
    """
    # on part de zéro
    result = np.zeros(shape=(size, size), dtype=int_)
    # on remplit les cases à 1 en deux fois
    # avec un slicing astucieux; c'est le ::2 qui fait le travail
    result[1::2, 0::2] = 1
    result[0::2, 1::2] = 1
    # encore une autre façon de renverser,
    # plutôt que le xor, puisque False == 0 et True == 1
    if corner_0_0:
        result = 1 - result
    return result
```

checkers_4 - Semaine 7 Séquence 05

```
def checkers_4(size, corner_0_0=True):
    """
    Et encore une autre, sans doute pas très lisible
    mais très astucieuse

    credits: j4l4y
    """
    # une utilisation très astucieuse de resize,
    # broadcasting, décalage, bravo !
    return (np.resize((corner_0_0, 1-corner_0_0),
                      (1, size))
            ^ np.arange(size)[:, np.newaxis] & 1)
```

hundreds - Semaine 7 Séquence 05

```
def hundreds(lines, columns, offset):
    """
    Fabrique un tableau lines x columns où:

    tab[i, j] = 100 * i + 10 * j + offset
    """
    # avec indices(), on a directement
    # deux tableaux prêts à être broadcastés
    indx, indy = np.indices((lines, columns))
    return 100*indx + 10*indy + offset
```

hundreds_bis - Semaine 7 Séquence 05

```
def hundreds_bis(lines, columns, offset):
    """
    Pareil, toujours à base de broadcasting
    """
    # cette fois on se fabrique soi-même la souche
    # des lignes et des colonnes pour montrer
    # comment on peut se faire indices() à la main
    # dans du vrai code, utilisez indices()
    #
    # une colonne 0, 1, .. lines-1
    column = np.arange(lines)[:, np.newaxis]
    # une ligne 0, 1, ... columns-1
    line = np.arange(columns)
    # il n'y a plus qu'à broadcaster les deux
    # attention toutefois que c'est column qui contient
    # les indices en i
    return 100*column + 10*line + offset
```

hundreds_ter - Semaine 7 Séquence 05

```
def hundreds_ter(lines, columns, offset):
    """
    Une approche discutable
    """
    # à la Fortran; ça n'est pas forcément
    # la bonne approche ici bien sûr
    # mais si un élève a des envies de benchmarking...
    result = np.zeros(shape=(lines, columns), dtype=np.int_)
    for i in range(lines):
        for j in range(columns):
            result[i, j] = 100 * i + 10 * j + offset
    return result
```

stairs - Semaine 7 Séquence 05

```
def stairs(taille):
    """
    la pyramide en escaliers telle que décrite dans l'énoncé
    """
    # on calcule n
    total = 2 * taille + 1
    # on calcule les deux tableaux d'indices
    # tous les deux de dimension total
    I, J = np.indices((total, total))
    # on décale et déforme avec valeur absolue, pour obtenir
    # deux formes déjà plus propices
    I2, J2 = np.abs(I-taille), np.abs(J-taille)
    # si ajoute on obtient un négatif,
    # avec 0 au centre et taille aux 4 coins
    negatif = I2 + J2
    # ne retse plus qu'à renverser
    return 2 * taille - negatif
```

stairs_2 - Semaine 7 Séquence 05

```
def stairs_2(taille):
    """
    même idée, modalités légèrement différentes
    Aussi on peut inverser plus tôt
    """
    total = 2 * taille + 1
    # on peut préciser le type, mais ce n'est pas
    # réellement nécessaire ici
    I, J = np.indices((total, total), dtype=np.int8)
    # on peut inverser avant d'ajouter si c'est plus naturel
    return  (taille - np.abs(I-taille)) + (taille - np.abs(J-taille))
```

stairs_3 - Semaine 7 Séquence 05

```
def stairs_3(taille):
    """
    en fait on n'a pas vraiment besoin d'indices
    """
    # la première ligne
    line = taille - np.abs(np.arange(-taille, taille+1))
    # la première colonne est la transposée
    # comme je n'aime pas utiliser .T
    # je préfère un reshape
    # et il n'y a qu'à ajouter
    return line + line.reshape((2*taille+1, 1))
```

stairs_4 - Semaine 7 Séquence 05

```
def stairs_4(taille):
    """
    une approche par mosaique
    on construit un quart, et on le duplique avec
    * np.hstack (une fonction d'empilement)
    * np.flip (une fonction de miroir)

    credits: JeF29
    """
    a = np.arange(taille)
    b = np.hstack((a, taille, np.flip(a)))
    return b + b.reshape(-1, 1) # ou b + b[:, np.newaxis]
```

stairs_ter - Semaine 7 Séquence 05

```
def stairs_ter(taille):
    """
    Version proposée par j4l4y
    Dans la rubrique 'oneliner challenge'

    credits: j4l4y
    """
    # la forme np.abs(np.range(-n, n+1)) correspond à la forme
    # en V, par exemple pour n=3 : -3, -2, -1, 0, 1, 2, 3
    # dans cette version, on l'agrandit artificiellement en 2D
    # pour pouvoir prendre sa transposée
    return (lambda x: x + x.T)(
        taille - np.abs(range(-taille, taille+1))[:, np.newaxis]
    )
```

dice - Semaine 7 Séquence 05

```
def dice(target, nb_dice=2, nb_sides=6):
    """
    Pour un jeu où on lance `nb_dice` dés qui ont chacun `sides` faces,
    quel est le nombre de tirages dont la somme des dés fasse `target`

    Version force brute, il y a bien sûr des outils mathématiques
    pour obtenir une réponse beaucoup plus rapidement

    Toutes les solutions procèdent en deux étapes

    * calcul de l'hypercube qui énumère les tirages,
      et calcule la somme des dés pour chacun de ces tirages
    * trouver le nombre de points dans le cube où la somme des dés
      correspond à ce qu'on cherche

    les deux étapes sont indépendantes, et peuvent donc être mélangées
    entre les solutions
    """

    # pour élaborer le cube, on procède par broadcating
    # on commence avec un simple vecteur de shape (nb_sides,) - e.g. de 1 à 6
    # on lui ajoute lui-même mais avec une forme (nb_sides, 1) - en colonne donc
    # et ainsi de suite avec
    # shape=(nb_sides, 1, 1) pour la dimension 3,
    # shape=(nb_sides, 1, 1, 1) pour la dimension 4
    sides = np.arange(1, nb_sides+1)
    cube = sides
    # une liste plutôt qu'un tuple pour décrire la shape,
    # car on va y ajouter '1' à chaque tour
    shape = [nb_sides]
    # on a déjà un dé
    for _dimension in range(nb_dice - 1):
        shape.append(1)
        cube = cube + sides.reshape(shape)

    # le cube est prêt,
    # pour chercher combien de cases ont la valeur target,
    # on peut faire par exemple
    return np.sum(cube == target)

```

dice_2 - Semaine 7 Séquence 05

```
def dice_2(target, nb_dice=2, nb_sides=6):
    """
    une variante de la première forme, qui utilise
    astucieusement une matrice diagonale pour énumérer
    les 'shapes' qui entrent en jeu

    credits: aurelien
    """
    sides = np.arange(1, nb_sides+1)
    shapes = np.diag([nb_sides-1]*nb_dice) + 1
    # attention ici c'est le sum Python
    # et non pas np.sum qui ferait complètement autre chose
    cube = sum(sides.reshape(s) for s in shapes)

    # une autre façon de faire le décompte
    return np.count_nonzero(cube == target)
```

dice_3 - Semaine 7 Séquence 05

```
def dice_3(target, nb_dice=2, nb_sides=6):
    """
    même logique globalement, mais en utilisant
    np.newaxis pour changer de dimension
    """
    sides = np.arange(1, nb_sides+1)
    cube = sides
    # on a déjà un dé
    for _dimension in range(nb_dice - 1):
        sides = sides[:, np.newaxis]
        cube = cube + sides

    # une autre façon de faire le décompte
    return np.count_nonzero(cube == target)
```

dice_4 - Semaine 7 Séquence 05

```
def dice_4(target, nb_dice=2, nb_sides=6):
    """
    on peut aussi tirer profit de indices()
    qui fait déjà presque le travail
    puisqu'il construit plusieurs cubes de la bonne dimension
    qu'il ne reste plus qu'à additionner
    """
    # il faut quand même faire attention
    # car indices() commence à 0
    all_indices = np.indices(nb_dice * (nb_sides,)) + 1
    cube = sum(all_indices)

    return np.count_nonzero(cube == target)
```

dice_5 - Semaine 7 Séquence 05

```
def dice_5(target, nb_dice=2, nb_sides=6):
    """
    une très légère variante
    """
    all_indices = np.indices(nb_dice * (nb_sides,))
    # une façon plus pédante mais plus propre de faire la somme
    # si on n'a pas rectifié avant, il faut maintenant ajouter nb_dice
    cube = np.add.reduce(all_indices) + nb_dice

    return np.count_nonzero(cube == target) # ou return len(res[res == target])
```

dice_6 - Semaine 7 Séquence 05

```
# on peut aussi utiliser itertools.product qui permet
# d'itérer sans aucune mémoire sur le même hypercube
#
# de manière un peu paradoxale, cette version en Python pur,
# bien que nécessitant en théorie beaucoup moins de mémoire,
# est beaucoup moins efficace que la version numpy
# je vous renvoie à la discussion sur le forum intitulée
# "Exercice dice"
from itertools import product

def dice_6(target, nb_dice=2, nb_sides=6):
    """
    Une autre méthode complètement, qui n'alloue aucun tableau
    du coup on n'a pas besoin de numpy
    """
    # en version facile, on peut utiliser le paramètre `repeat`
    # de product qui fait exactement ce qu'on veut, puisque
    # tous les dés ont le même nombre de faces
    #
    # par exemple le cas standard (2 dés, 6 faces) se ferait avec
    # quelque chose comme
    # (for (i, j) in itertools.product(range(1, 7), repeat=2))
    #
    # le premier sum compte les occurences de True dans l'itération
    return sum(
        # ici sum(x) fait la somme des tirages des dés
        sum(x) == target
        for x in product(range(1, sides+1), repeat=nb_dice))
```

matdiag - Semaine 7 Séquence 05

```
import numpy as np

def matdiag(liste):
    """
    si les arguments sont x1, x2, .. xn
    retourne une matrice carrée n x n
    dont les éléments valent
    m[i, j] = xi si i == j
    m[i, j] = 0 sinon

    credit: JeF29
    """
    # on crée une matrice diagonale unité avec np.eye
    # (car I se prononce comme eye en anglais)
    # et on la multiplie par broadcasting avec un vecteur
    # composé de nos arguments
    # on la crée de type `int64` de façon à obtenir
    # pour le résultat final un type entier, flottant
    # ou complexe, selon les valeurs dans liste
    return np.eye(len(liste), dtype=np.int64) * liste
```

matdiag_2 - Semaine 7 Séquence 05

```
def matdiag_2(liste):
    """
    même propos mais cette fois avec du slicing
    """
    #
    # on initialise un tableau de la bonne taille n x n
    # mais tout à plat, avec des zéros
    # ici si on veut que ça marche avec des complexes,
    # il faut alors créer tout de suite le tableau de type
    # complexe, sinon on n'a pas la place
    n = len(liste)
    plat = np.zeros((n * n,), dtype=np.complex)
    #
    # dans cette représentation là, la diagonale correspond
    # à un slice qui commence à 1 avec un pas de n+1
    plat[0 : : n+1] = liste
    #
    # maintenant on peut remettre
    # dans une forme n x n avec reshape
    #
    return plat.reshape((n, n))
```

matdiag_3 - Semaine 7 Séquence 05

```
def matdiag_3(liste):
    """
    bon maintenant qu'on s'est bien creusé les méninges
    pour le faire à la main, il se trouve qu'il y a
    - bien sûr - une fonction pour ça dans numpy
    """
    return np.diag(liste)
```

xixj - Semaine 7 Séquence 05

```
import numpy as np

def xixj(*args):
    """
    si les arguments sont x1, x2, .. xn
    retourne une matrice carrée n x n
    dont les éléments valent
    m[i, j] = xi * xj

    première solution à base de produit usuel
    entre un vecteur et une colonne, en utilisant
    le broadcasting

    credits: JeF29
    """

    # une ligne qui contient x1, .. xn
    line = np.array(args)
    # habile façon de reshaper automatiquement
    column = line.reshape(-1, 1)
    # on aurait pu faire aussi
    #column = line[:, np.newaxis]
    return line * column
```

xixj_2 - Semaine 7 Séquence 05

```
def xixj_2(*args):
    """
    pareil mais on construit la colonne avec .T
    qui est la transposée - méfiance quand même
    """
    # sauf que pour pouvoir utiliser .T il faut
    # une shape qui est explicitement [1, n]
    #
    # c'est pourquoi moi j'ai tendance à éviter .T
    # voyez plutôt np.transpose() si vous avez besoin
    # de transposer une matrice
    line = np.array(args).reshape((1, -1))
    return line * line.T
```

xixj_3 - Semaine 7 Séquence 05

```
def xixj_3(*args):
    """
    on peut aussi penser à faire un produit matriciel
    """
    # on doit lui donner une dimension 2 même si c'est une ligne
    line = np.array(args).reshape((1, -1))
    column = line.reshape((-1, 1))
    return column @ line
```

xixj_4 - Semaine 7 Séquence 05

```
def xixj_4(*args):
    """
    pareil mais en utilisant .dot()
    """
    column = np.array(args).reshape((-1, 1))
    # dans cette version on fait le produit de matrice
    # en utilisant la méthode dot sur les tableaux
    return column.dot(column.T)
    # remarquez qu'on aurait pu faire aussi bien
    # return np.dot(column, column.T)
```

npsearch - Semaine 7 Séquence 05

```
import numpy as np

def npsearch(world, needle):
    """
    world est la "grande" matrice dans laquelle
    on cherche les occurrences de needle
    qui peut être une matrice 2d ou une simple ligne

    npsearch est une fonction génératrice qui énumère
    les tuples (i, j) correspondant à une occurrence de
    needle dans world
    """
    if len(needle.shape) == 1:
        needle = needle[np.newaxis, :]
    n, m = needle.shape
    # pas la peine de faire une grande boucle sur tout le tableau
    # s'il y a égalité c'est nécessairement que
    # le world[i, j] == needle[0, 0]
    for i, j in np.argwhere(world == needle[0][0]):
        # c'est ici le point délicat
        # si vous comparez les deux tableaux à base de ==
        # (même en utiisant np.all)
        # vous allez potentiellement mettre en oeuvre
        # un broadcasting non souhaitable
        if np.array_equal(world[i:i+n, j:j+m], needle):
            yield i, j
```

taylor - Semaine 7 Séquence 10

```
class Taylor:
    """
    provides an animated view of Taylor approximation
    where one can change the degree interactively

    Taylor is applied on X=0, translate as needed
    """

    def __init__(self, function, domain):
        self.function = function
        self.domain = domain

    def display(self, y_range):
        """
        create initial drawing with degree=0

        Parameters:
          y_range: a (ymin, ymax) tuple
            for the animation to run smoothly, we need to display
            all Taylor degrees with a fixed y-axis range
        """
        # create figure
        x_range = (self.domain[0], self.domain[-1])
        self.figure = figure(title=self.function.__name__,
                             x_range=x_range, y_range=y_range)

        # each of the 2 curves is a bokeh line object
        self.figure.line(self.domain, self.function(self.domain), color='green')
        # store this in an attribute so _update can do its job
        self.line_approx = self.figure.line(
            self.domain, self._approximated(0), color='red', line_width=2)

        # needed so that push_notebook can do its job down the road
        self.handle = show(self.figure, notebook_handle=True)
```

taylor (continued) - Semaine 7 Séquence 10

```
    def _approximated(self, degree):
        """
        Computes and returns the Y array, the images of the domain
        through Taylor approximation

        Parameters:
          degree: the degree for Taylor approximation
        """
        # initialize with a constant f(0)
        # 0 * self.domain allows to create an array
        # with the right length
        result = 0 * self.domain + self.function(0.)
        # f'
        derivative = autograd.grad(self.function)
        for n in range(1, degree+1):
            # the term in f(n)(x)/n!
            result += derivative(0.)/factorial(n) * self.domain**n
            # next-order derivative
            derivative = autograd.grad(derivative)
        return result

    def _update(self, degree):
        # update the second curve only, of course
        # the 2 magic lines for bokeh updates
        self.line_approx.data_source.data['y'] = self._approximated(degree)
        push_notebook(handle=self.handle)

    def interact(self, degree_widget):
        """
        Parameters:
          degree_widget: a ipywidget, typically an IntSlider
            styled at your convenience
        """
        interact(lambda degree: self._update(degree), degree=degree_widget)
```