TP n° 2 : Boucles, fonctions et éléments de `numpy`

Objectifs du TP : utiliser des tests (if...then...else), des boucles (for et while) rédiger une fonction avec de l'aide. Utliser des mesures d'efficacité temporelle.

Repartir du TP1 et renommé votre fichier en suivant la même procédure, et en utilisant `filename` pour votre nom de TP :

```
# Changer ici par votre Prenom Nom:
prenom = "Joseph" # à remplacer
nom = "Salmon" # à remplacer
extension = ".ipynb"
tp = "TP2_HMLA310"
filename = "_".join([tp, prenom, nom]) + extension
filename = filename.lower()
```

Exercice 1. Petit théorème de Fermat

Le petit théorème de Fermat est un résultat connu de la théorie des nombres. Illustrons le dans cet exercice numériquement. Son énoncé est le suivant :

Soit p un nombre premier et a un entier non divisible par p alors $a^{p-1} - 1 \equiv 0[p]$.

(a) Écrire une fonction nommée 'fermat' qui prend en entrée a et p, et qui renvoie a^{p-1} mod p

(b) Montrez que ce résultat est vrai pour $p = 13$ et $a = 16$. On testera l'égalité avec le symbole '==' et la fonction `fermat`.

(c) Créer une liste nommée 'prems', contenant les 7 premiers nombre premiers.

(d) Tester que le théorème est vrai pour les 7 premiers nombres premiers et pour tous les nombres a plus petit que 100 satisfaisant la contrainte a non divisibles par p. On pourra pour cela utiliser

- la liste `prems`
- deux boucles `for` imbriquées
- un test `if`

Exercice 2. Séries

Calculez la série $\sum_{j=0}^{n} r^j$ pour $r = .75$ et $n = 10, 20, 30, 40$. On utilisera deux méthodes :

(a) une solution avec une boucle sur une liste

(b) une solution avec la fonction sum sur un numpy array.

Enfin comparez votre résultat à celui de la formule $(1 - r^{n+1})/(1 - r)$

Exercice 3. Gymnastique de création de matrices/vecteurs en `numpy`

Créer les vecteurs et matrices suivants sans renter les valeurs une à une :

(a) array([1., 1., 1.]) (dont la taille est (3,))

(b) array([[1., 1., 1.]]) (dont la taille est (1,3))

(c) array([1, 2, 3])

(d) array([2, 1, 0, -1, -2])

(e) array([1, 2, 3, 1, 2, 3])

(f) array([[1, 2, 3],[4, 5, 6]])

(g) array([[0., 0., 0., 0., 0., 0.],
[1., 0., 0., 0., 0., 0.],
[0., 1., 0., 0., 0., 0.],
[0., 0., 1., 0., 0., 0.],
[0., 0., 0., 1., 0., 0.],
[0., 0., 0., 0., 1., 0.]])

Exercice 4. Suite de Fibonacci

On se propose de faire l'étude de cette suite et de comparer l'efficacité en mémoire et en temps, en passant par des boucles et des listes, ou bien par un modèle matriciel. On pourra consulter la page : https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html pour plus de détails. Pour rappel la suite est définie par la récurrence d'ordre deux :

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_n = F_{n-1} + F_{n-2}, \text{ si } n \geqslant 2. \end{cases}$$

et on peut donc calculer les premiers termes de la suite : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

(a) Utiliser la commande magique `%timeit` (et `%memit` si `%load_ext memory_profiler` est disponible sur votre machine) pour mesurer la performance du calcul de `fibonacci_naive(35)`, où la fonction `fibonacci_naive` est définie par :

```
def fibonacci_naive(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return fibonacci_naive(n-1)+fibonacci_naive(n-2)
```

À votre avis qu'est ce qui fait que cette fonction est si lente ?

(b) Proposez une implémentation avec deux variables et une boucle `for` qui soit plus efficace.

(c) Proposez une implémentation avec deux variables, une liste et une boucle `for` qui permet de ressortir une liste contenant toutes les valeurs de la suite jusqu'à un entier donné de manière efficace.

(d) Comparez avec la version suivante qui utilise un cache

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fibonacci_cache(n):
    if n < 2:
        return n
    return fibonacci_cache(n-1) + fibonacci_cache(n-2)
```

(e) Proposez une implémentation avec `numpy` qui utilise la relation suivante :

$$\text{pour } n \geqslant 0, \quad \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} .$$

ainsi qu'une puissance de matrice (*i.e.,* la fonction `np.linalg.matrix_power` de `numpy`).