Université de Montpellier **HLMA310**
2019/2020 **Joseph Salmon**

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

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

ATTENTION : si vous n'avez pas accès à `jupyter notebook` sur votre machine, vous pouvez utiliser le site : https://mybinder.org/. Renseignez alors dans la boite "GitHub repository name or URL" l'url suivant : https://gitlab.com/josephsalmon/hlma310/, en mettant l'option `Gitlab` puis cliquez sur "Launch". Cela permet de lancer un notebook de manière distante, et d'effectuer ce TP sans installation sur votre machine (le temps de lancement peut être plus ou moins long en fonction du trafic).

Comme au TP1 : renommez votre fichier notebook en `filename`, où `filename` est obtenu comme suit :

```
# Changer ici par votre Nom / Prénom:
nom = "Salmon" # à remplacer
prenom = "Joseph" # 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. Nous allons l'illustrer numériquement dans cet exercice. Son énoncé est le suivant :

Petit théorème de Fermat. *Soit p un nombre premier et soit 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) Vérifiez que ce résultat est vrai pour $p = 13$ et $a = 16$. On testera l'égalité avec le symbole `==` et la fonction `fermat` précédente.

(c) Créer la liste `prems` qui contient les 7 premiers nombre premiers.

(d) Vérifiez la justesse du théorème pour les 7 premiers nombres premiers et pour tous les nombres $a \in [\![1, 100]\!]$ satisfaisant la contrainte a non divisibles par p. On utilisera pour cela une boucle `while` et une variable booléenne qui arrêterais la boucle si le théorème était faux :

- la liste `prems`
- un booléen
- une boucle extérieur `while` et `for` imbriquées
- un test `if` (ou deux si vous le souhaitez).

Exercice 2. (Séries)

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

(a) une solution avec une boucle `for` sur une liste

(b) une solution avec un `numpy` array et la fonction `sum` du même module.

Enfin comparez le résultat pour ces quatre valeurs de n à celui donné par la formule $\dfrac{1 - r^{n+1}}{1 - r}$.

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

Créez les vecteurs et matrices qui suivent **sans rentrer les valeurs à la main** une à une. On pourra utiliser notamment : `np.ones`, `np.arange`, `np.diag`, `np.reshape` :

(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])` (avec `np.hstack` ou `np.tile`)
(f) `array([[1, 2, 3],[4, 5, 6]])`
(g) l'`array` qui représente la matrice $\begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 0. \\ 1. & 0. & 0. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 1. & 0. & 0. & 0. \\ 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 0. & 0. & 0. & 1. & 0. \end{bmatrix}$

Exercice 4. (Suite de Fibonacci)

On se propose de faire l'étude de cette fameuse suite et de comparer l'efficacité en mémoire et en temps de diverse implémentations. On utilisera par exemple une version avec des boucles et des listes, et une version avec `numpy` et un modèle matriciel. Pour rappel la suite de Fibonacci est définie par la récurrence d'ordre deux :

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

Les premiers termes de cette suite sont donc : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

(a) Utiliser la commande magique `%timeit` (si `%load_ext memory_profiler` est disponible sur votre machine) [1] pour mesurer la performance du calcul de `fibonacci_naive(28)`, 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 `fibonacci_list` plus efficace, utilisant deux variables, une liste et une boucle `for` et qui prend en entrée n et qui ressort une liste contenant toutes les valeurs de la suite $F_0, F_1, \ldots, F_n$. Par exemple `fibonacci_list(5)` donnera en sortie : `[0, 1, 1, 2, 3, 5]`.

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

(d) Comparez avec la version suivante qui utilise un cache (*i.e.,* un stockage temporaire) :

```
from functools import lru_cache
@lru_cache()
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`). Comparez l'efficacité de cette méthode avec les précédentes implémentations.

1. on peut obtenir des informations sur l'utilisation de la mémoire de manière similaire, en utilisant la commande `%memit`

Pour aller plus loin

On pourra consulter la page :
https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html
pour plus de détails sur les implémentations de la suite de Fibonaci.