Initiation Python/Jupyter TP2

Germain POULLOT

13 Octobre 2021

Table des matières

1 Objectifs

1. Créer des fonctions
2. Importer des bibliothèques et jouer avec
3. Quelques structures de données
4. Des plot (matplotlib ou `list_plot`)

2 Afficher et stocker ses résultats

2.1 Fonctions

Exercise 2.1.1. Finir le TP précédent.

Les fonctions sont très utiles, mais il ne faut pas en abuser : lorsqu'on peut coder sans fonction, autant le faire, et on transformera plus tard le code en une fonction.

2.2 Importer des bibliothèques

Exercise 2.2.1. Pour travailler sur les bibliothèques, oublions SageMath quelques temps.

Q : Lancez un nouveau document en Python 3.

Q : Essayez d'obtenir un nombre aléatoire dans $[0, 1[$ avec `random()` et lisez l'erreur.

Tapez le code suivant pour importer la bibliothèque (ou "module") NumPy :

```
import numpy
print(numpy.pi)
print(numpy.e)
print(numpy.euler_gamma)
```

Pour appeler la bibliothèque, on utilise le mot `numpy`, puis un `.` pour dire "dans cette bibliothèque, je voudrais", et ensuite ce qu'on veut. NumPy propose des valeurs approchées de π, e et γ, ce que Python n'a pas nativement : essayez de taper `pi`, `e` et `euler_gamma` pour voir.

Dans une bibliothèque, il peut y avoir des sous-bibliothèques, ou *packages* :

```
print(numpy.random.rand())
print(numpy.random.randint(1,10))
```

Ces packages sont directement importés avec la bibliothèque. Le package `random` de NumPy gère l'aléatoire, sa fonction `rand()` tire uniformément dans $[0, 1[$, et `randint(a,b)` tire uniformément dans $[\![a, b[\![$.

Q : Tapez `print(numpy.random.binomial(100,0.25,10))` et utilisez la commande `help` pour comprendre ce que fait la fonction `numpy.random.binomial`.

Q : Déterminez si le cosinus de e^2 est plus grand que 0.4. **Attention :** Dans Python seul, l'exponentiation se fait avec `**`, pas `^`.

Exercise 2.2.2. On peut importer des bibliothèques en ajoutant certaines options pour éviter de retaper `numpy.` à chaque fois. Notamment, on peut donner un alias et on peut tout importer sans préfixe :

```
from numpy import *
import numpy.random as npr

print(pi)
print(e)
print(random.rand())
print(npr.rand())
print(rand())
```

Q : Importez `matplotlib.pyplot` sous l'alias `plt`. Importez le module `scipy` en entier. Importez la fonction `factorial` du module `math` et appelez-la `fact`.

Lorsque vous importez tout un module sans préfixe, vous risquez d'avoir des conflits de noms : si une fonction du module que vous impotez s'appelle `numy.zorglub` et que vous avez déjà une fonction `zorglub` (dans votre code ou issue d'un autre module), alors excécuter `from numpy import *` effacera votre `zorglub` pour mettre celui de NumPy à la place.

Le module MatPlotLib permet de gérer les affichages de résultats. En particulier, `pyplot` est utile pour les courbes. Exécutez :

```
plt.plot([1,2,3],[-5,-7,-3])
plt.show()
```

Q : À quoi correspond le résultat affiché ?

Si je cherche quelquechose dans un module, mais que je n'ai pas accès à la documentation, je peux utiliser la touche tabulation : tapez `numpy.random.` puis tabulation, vous aurez la liste de (presque) tout ce que contient le module `random` de NumPy.

Q : Trouvez-y de quoi simuler une loi du χ^2 et obtenez sa documentation (`help`) pour en comprendre le fonctionnement.

2.3 Listes, ensembles, itérateurs

Pour pouvoir stocker les données plus efficacement, il faut les encapsuler des des *structures de données*. Il en existe de plusieurs type, nous verrons surtout des *itérateurs*, c'est-à-dire des structures qui stockent un nombre fini (voire dénombrable) de données, et qu'on peut parcourir.

2.3.1 Listes et tables

Exercise 2.3.1. Dans Python, les listes (de type *"last in, first out"* pour les puristes) et les tables sont (presque) la-même chose. Elles se codent avec des crochets `[...]`.

```
L = [2,3,5,7,11,13,17,19]
print(L)
print(type(L))
print(L[0])
print(L[3])
print(L[-1])
print(len(L))
print(L[:4])
print(L[4:])
print(L[1:6])
```

Q : Comprenez ce que font ces différentes lignes.

L'opération habituelle consiste à ajouter une valeur à la fin d'une liste. Cela se fait avec la commande `append`. Cette commande est programmée en orienté-objet, il faut donc taper :

```
L.append(23)
print(L)
L.append(29)
print(L)
L.append("N'importe quoi")
print(L)
L.append([1,1,2,3,5,8,13,21])
print(L)
L[-1].append(34)
print(L)
```

On peut mélanger plusieurs types de données dans une même liste.

Si on veut ajouter un élément à un autre endroit de la liste, on utilise `insert(`*la_position*`,`*l'élément*`)`. **Attention :** L'index va de 0 à $\texttt{len}(L)-1$ (et la longueur change avec l'insertion).

```
L.insert(0,1)
print(L)
L.insert(11,31)
print(L)
L[-1].insert(0,0)
print(L)
```

On peut aussi concatener des liste avec `+`, ce qui ne modifie pas les listes qu'on concatène (contrairement à `append` qui modifie définitivement la liste).

```
M = [1,2,3]
N = [4,5,6]
print(M,N)
K = M+N
print(K)
print(M,N)
```

Pour copier une liste, on n'écrit pas `M = L`, mais `M = list(L)` ou `M = copy(L)`.

Q : Créez une liste `L` qui contient 100 valeurs aléatoires de $[0,1[$. Affichez les 20 premiers éléments de `L`.

Q : En utilisant l'aide à la saisie (`L.` puis tabulation), trier votre liste. Affichez les 20 premiers éléments de `L`.

Exercise 2.3.2. On peut en fait construire des listes en 1 ligne en mettant la boucle `for` "dans" la liste. Pour l'exercice précédent, cela donnerait :

```
L = [npr.rand() for k in range(100)]
```

On peut aller plus loin en intégrant plusieurs boucles, en ajoutant des conditions :

```
L = [(i,j) for i in range(1,21) for j in range(1,21) if i-j >= 10]
print(L)

L = [npr.rand() for k in range(100)]
M = [x for x in L if x > 1/2]
print(len(M),min(M))
```

On peut faire un exemple un peu plus puissants en SageMath (`graphs(n)` génère tous les graphes à `n` sommets) :

```
L = [(H,G) for H in graphs(3) for G in graphs(5) if G.is_connected() and
    H.is_connected() and H.is_subgraph(G)]
for H,G in L:
    H.show()
    G.show()
```

Q : Créez en une ligne la liste de tous les diviseurs de $10! = 3628800$, affichez sa longueur et le 100ème diviseurs.

Exercise 2.3.3. Vous l'avez vu, `matplotlib.pyplot.plot` (ou `plt.plot` via notre alias), prend en entrée 2 listes (de même longueur) `L1` et `L2` et trace le graphique de la fonction : $L_1[k] \mapsto L_2[k]$.

Q : Grâce aux fonctions précédemment codées (cf TP1), affichez le graphique de : $n \mapsto$ plus grand diviseurs premier(n) ; pour n de 2 à 2000. Commentez.

Pour placer des points sans les relier, vous pouvez utiliser `plt.scatter`, ou bien jouer avec les options de `plt.plot` : ajoutez un troisième argument `'g.'` par exemple, tracera la courbe en vert (`g` pour *green*) et en petits points ; ajouter `'r+'` pour avoir une courbe rouge avec des petites croix. Cf page suivante.

Exercise 2.3.4 (Facultatif)**.** Avec SageMath, on peut afficher plus facilement un nuage de points : on fabrique une liste `L` de couple `(x_i,y_i)` (les coordonnées de nos points dans le plan), et on écrit `list_plot(L)`. Outre de ne pas avoir besoin d'importer `plt`, il est souvent plus naturel d'engendrer les points ainsi.

Q : Retournez sur votre fichier SageMath et refaite la question de l'exercice précédent en SageMath.

Voici la liste des symboles (pour plus d’options de `plt.plot`, regarder le site, très moche mais bien renseigné : http://www.python-simple.com/python-matplotlib/pyplot.php) :

- Couleurs :

  `’b’` : blue

  `’g’` : green

  `’r’` : red

  `’c’` : cyan

  `’m’` : magenta

  `’y’` : yellow

  `’k’` : black

  `’w’` : white

- Symboles :

  `’-’` : solid line style

  `’-’` : dashed line style

  `’-.’` : dash-dot line style

  `’:’` : dotted line style

  `’.’` : point marker

  `’,’` : pixel marker

  `’o’` : circle marker

  `’v’` : triangle down marker

  `’^’` : triangle up marker

  `’<’` : triangle left marker

  `’>’` : triangle right marker

  `’1’` : tri down marker

  `’2’` : tri up marker

  `’3’` : tri left marker

  `’4’` : tri right marker

  `’s’` : square marker

  `’p’` : pentagon marker

  `’*’` : star marker

  `’h’` : hexagon1 marker

  `’H’` : hexagon2 marker

  `’+’` : plus marker

  `’x’` : x marker

  `’D’` : diamond marker

  `’d’` : thin diamond marker

  `’|’` : vline marker

  `’_’` : hline marker

2.3.2 Array et matrices

Exercise 2.3.5. Les modules NumPy et MatPlotLib sont faits pour fonctionner de concert. Par exemple, NumPy dispose d'une structure `array` qui est similaire aux listes mais avec des propriétés supplémentaires.

Pour transformer une `list` en un `array`, il suffit d'écrire `array(ma_liste)`, et inversement avec `list(mon_array)`.

On ne peut pas utiliser `append`, `insert` ou la concaténation sur les `array`, ils ne sont pas fait pour ça. Par contre, ils sont optimisés pour le calcul numérique et le calcul matriciel :

```
A = numpy.array([1,2,3,4,5,6,7,8,9,10])
print(A)
B = numpy.cos(A)
print(B)
plt.plot(A,B)
```

Q : Renseignez-vous sur `numpy.linspace` (via `help` ou via Internet), et créez un `array` contenant 1000 réels de 0 à 4π régulièrement espacés. Affichez le graphique de la fonction sinus.

Q : Dans la même cellule de code, ajouter des lignes pour calculer et afficher le graphique du cosinus sur $[0, 4\pi]$ (sur le même graphique que le sinus).

Q : Faites un tracé de courbes pour identifier graphiquement la solution de $\arccos x = \arcsin x$.

Exercise 2.3.6. Les matrices sont gérée nativement en SageMath grâce à `matrix`. En Python seul, les `array` du module NumPy sont là pour ça.

```
M = numpy.array([[1,2],[4,5],[7,8]]) #M = matrix([[1,2],[4,5],[7,8]])
print(M)
```

Q : Calculez la matrice des carrés et le carré de la matrice $\begin{pmatrix} 2 & 1 & 3 & 1 \\ 6 & 3 & 9 & 3 \\ 2 & 1 & 3 & 1 \\ 6 & 3 & 9 & 3 \end{pmatrix}$ (`A.dot(B)` est la multiplication matricielle de `A` par `B`). Étrange n'est-ce pas ?

Q : Rassurons nous sur la capacité de NumPy a faire un produit matriciel : créez une matrice 3×3 contenant des entiers aléatoires entre 0 et 9 (inclus), calculez son carré et vérifiez certaines cases à la main.

On peut accéder à une cellule d'une matrice en précisant ses deux coordonnées `M[i,j]` (qui commencent à 0), et on peut créer des matrices usuelles grâce à : `numpy.ones(n,m)` matrice $n \times m$ qui ne contient que des 1, `numpy.eye(n)` matrice identité $n \times n$, `numpy.zeros(n,m)` matrice $n \times m$ qui ne contient que des 0.

Q : **Exercise 2.3.7** (Difficile mais complet)**.** Écrire une fonction qui décide si une matrice est vampire : une matrice A est vampire si ses entrées sont dans $[\![0, 9]\!]$ et si la cellule (i, j) de A^2 vaut 11 fois la cellule (i, j) de A (`numpy.shape(A)` pour le nombre de lignes et de colonnes de la matrice `M`).

Q : Pour `n` et `m` deux entiers, `S` une liste, écrivez une fonction qui fait la liste de toutes les matrices de $\mathcal{M}_{n,m}(S)$. Testez pour $n = 2$, $m = 3$ et $S = [0, 1, 2, 3, 4]$: est-ce que vous obtenez une liste de la bonne longueur ? Est-ce que votre liste comporte deux fois la même matrice ?

Q : Écrivez une fonction qui prend en entrée n et qui retourne la liste de toutes les matrices vampires $n \times n$. Vérifiez que cela fonctionne avec $n = 2$.

Q : Obtenez la liste des traces et la liste des déterminants des matrices vampires de taille 2×2.

Q : Modifiez votre code pour qu'il prenne en entrée un entier $k \in [\![0, 9]\!]$ et qu'il retourne les matrices vampires $n \times n$ dont les entrées sont dans $[\![0, k]\!]$. Déterminez le nombre de matrices vampires de taille 3×3 et dont les entrées sont inférieures ou égales à 4.

Q : Depuis le module `time`, importez la fonction `time`. Tapez `time()` vous donne la date exacte, c'est-à-dire le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit. On peut s'en servir pour mesurer le temps que prend un programme à s'exécuter. Mesurez le temps d'exécution de la question précédente.

Q : Tracez deux graphiques : le nombre de matrices vampires dans $\mathcal{M}_n([\![0, k]\!])$ et le temps de calcul que cela vous prend, en fonction de $k \in [\![0, 9]\!]$. **Attention :** Vous pouvez conjecturer que cela va être très très long pour $k \geq 6$, donc arrangez vous pour pouvoir arrêter votre programme sans perdre vos données !

2.4 Autres structures de données

Plein d'autres structures de données sont disponibles :

- Les ensembles `set` :
  - ⋆ L'ensemble vide $\emptyset$: `set()`
  - ⋆ Un ensemble quelconque : `S = {1,2,6,8,100}` (un seul type de données dedans)
  - ⋆ Ajouter, retirer, union, intersection, différence symétrique, etc : https://docs.python.org/2/library/sets.html
- Dictionnaire `dict` :
  - ⋆ Ils généralisent les listes : au lieu de stocker des valeurs dans l'ordre, ils les stockent par clefs. Exemple : je voudrais une structure de donnée telle que `D[p] = nombre_de_diviseurs_de_p-1`, mais seulement pour les nombres premiers. Je pourrais travailler avec en parallèle la liste des nombres premiers dans l'ordre, mais c'est casse-pieds. Autre exemple : je voudrais une structure telle que `D['abc'] = nombre_de_mots_qui_commencent_par_'abc'`, et idem pour toutes les chaînes de 3 caractères. Les dictionnaires permettent de le faire.
  - ⋆ Dictionnaire vide : `{}`
  - ⋆ Dictionnaire quelconque : `{'a' : 3 , 'b' : 4 , 123 : 7 , 124 : 'e'}`, c'est une "liste" de "clef : valeur".
  - ⋆ Pour accéder à une valeur : `D[clef]`.

- ⋆ Pour ajouter ou mettre à jour une valeur : `D[clef] = machin` (la clef sera créee automatiquement si elle n'existe pas déjà dans le dictionnaire).
- ⋆ `D.keys()` donne la liste des clefs, et `D.values()` la liste des valeurs.
- ⋆ Tout n'est pas permis comme clef (les sets ne peuvent pas servir de clef par exemple), mais on peut mettre n'importe quoi comme valeur.

Q : **Exercise 2.4.1.** Je lance deux dés à six faces : construisez un dictionnaire dont les clefs sont `0`, `1`, `2`, `3`, `4`, `5` et dont les valeurs vérifient `D[k] =` la liste des couples de dés dont la différence (absolue) fait `k`. Écrivez une fonctions qui calcule l'espérance, le mode et la variance de la différence (absolue) de deux dés à `n` faces. Tracez l'évolution de ces trois valeurs en fonction de $n \in [\![2, 100]\!]$. Tracez sur une même fenêtre les histogrammes des loi de la différence entre deux dés pour $n \in \{5, 10, 25, 40, 60, 75, 100\}$.

Exercise 2.4.2 (Bonus)**.** On se donne la formule de Héron : pour un triangle de côtés a, b et c, on note $s = \frac{1}{2}(a + b + c)$ le demi-périmètre, alors l'aire du triangle est $A = \sqrt{s(s-a)(s-b)(s-c)}$. On s'intéresse aux triangles dont les longueurs des côtés côtés sont des entiers. Fixons $N = 35$. Affichez les points (Périmètre, Aire) pour $a, b, c \in [\![1, N]\!]$ (faites en sorte que votre graphique soit lisible). Tracez sur le même graphique $x \mapsto \frac{\sqrt{3}}{36}x^2$ et $x \mapsto \frac{1}{4}\sqrt{(x-1)^2 - 1}$ pour $x \in [3, 3N]$. Commentez, interprétez, réfléchissez (et jouez avec la valeur de N pour voir si le dessin change).

3 Informations annexes

Regardez cette page pour trop d'informations sur les itérateurs : https://docs.python.org/fr/3/tutorial/datastructures.html.

Regardez la page du Project Euler : https://projecteuler.net/.

Pour ceux qui utilisent SageMath, la majorité des modules sont déjà importés ou recodés. Un excellent résumé de ce qu'on peut faire avec des matrices est ici : https://wiki.sagemath.org/quickref?action=AttachFile&do=get&target=quickref-linalg.pdf.