Exercices de Python

par Raymond Moché

ancien élève de l'École normale d'instituteurs de Douai et de l'École normale supérieure de Saint Cloud

ancien professeur de l'Université des Sciences et Technologies de Lille
et des universités de Coïmbre et de Galatasaray

http ://gradus-ad-mathematicam.fr

Table des matières

Chapitre 1

Liste des Énoncés

Chapitre 2 : Fonctions, boucles, instructions conditionnelles

---

Énoncé n° 2.1 [*] : Pour commencer : des calculs.

Quelle est la valeur de π ? de e ? de $\pi \cdot e$? Calculer $\dfrac{\sin(3) \cdot e^2}{1 + \tan\left(\frac{\pi}{8}\right)}$.

Mots-clefs : constantes pré-définies e, π, calcul numérique, module `math`, fonctions `sin` et `tan`.

Énoncé n° 2.2 [*] : Pour commencer : graphes simples.

1 - Tracer le segment de droite d'extrémités (1,3) et (15,-2), dans un repère orthonormé.
2 - Tracer le graphe de la fonction `sinus` quand la variable varie de 1 à 15 radians.

Mots-clefs : Instructions simples de tracer des graphes, commandes `linspace(a,b,n)` et `array([a,…,x])` du module `numpy`, fonctions vectorisées.

Énoncé n° 2.3 [*] : Représentations graphiques avec Python.

Pour tracer un graphique, Python en calcule des points consécutifs et les relie par un segment. Habituellement, on obtient ainsi une bonne approximation de la courbe à tracer quand il y a suffisamment de points car alors, nos yeux ne font plus la différence.
On se propose d'étudier des approximations du graphique de la fonction `sin()` lorsque la variable `x` varie sur l'intervalle `[0,`π`]`.
1 - Tracer l'approximation du graphique obtenue en calculant ses points d'abscisse 0, $\pi/2$ et π.
2 - Pour `n` valant successivement 3, 6, 10, 20, tracer l'approximation du graphique obtenue en calculant les points dont les abscisses sont données par la commande `linspace(0,pi,n+1)` (cette représentation graphique est une ligne polygonale à `n` côtés) .

Mots-clefs : Représentation graphique d'une fonction, repère orthonormé, commande `axis('equal')`, d'autres commandes graphiques.

Énoncé n° 2.4 [*] : Programmer des fonctions : valeur absolue, volume du cône.

1 - Programmer une fonction qui retourne la valeur absolue de tout nombre.
2 - Programmer une fonction qui retourne le volume de tout cône droit à base circulaire, de hauteur `h` cm et de rayon `r` cm.

Mots-clefs : programmer une fonction, instruction conditionnelle `if … else …`, `return`, importer une constante ou une fonction d'un module de Python.

Énoncé n° 2.5 [*] : Trinôme du second degré.

Résoudre l'équation du second degré (E) : $ax^2 + bx + c = 0$ $(a \neq 0)$

Mots-clefs : Fonction `sqrt`, à importer du module `math`, sortie d'une fonction par `return` ou `print`, instruction conditionnelle `if: … elif: … else: ….`

Énoncé n° 2.6 [**] : Minimum de 2, 3, 4 nombres (instructions conditionnelles).

1.a - Programmer une fonction qui retourne le minimum de 2 nombres.
1.b - Programmer une fonction qui retourne le minimum et le maximum de 2 nombres.
2 - Programmer une fonction qui retourne le minimum de 3 nombres.
3 - Programmer une fonction qui retourne le minimum de 4 nombres.

Mots-clefs : Programmation de fonctions, instruction conditionnelle `if … else …`, `return`, négation d'une proposition logique, importation de constantes et de fonctions pré-définies.

Énoncé n° 2.7 [***] : Alignement de 3 points.

On se donne, à l'aide de leurs coordonnées, 3 points distincts A, B et C d'un plan rapporté à un repère orthonormé. On appelle d_1, d_2 et d_3 les distances de B à C, de A à C et de A à B. On peut supposer que $d_1 \leqslant d_2 \leqslant d_3$.
1 - Démontrer que les points sont alignés si et seulement si $d_3 = d_1 + d_2$.
2 - Sans utiliser la fonction racine carrée, programmer un algorithme qui retourne, à partir de la liste des coordonnées de A, B et C, "Les points sont alignés" ou "Les points ne sont pas alignés", suivant le cas.

Mots-clefs : Calculer avec seulement des additions et des multiplications, transformer un problème en un problème équivalent. Commentaires : tester l'égalité de 2 nombres.

Énoncé n° 2.8 [*] : Affectation de données

Soit f une fonction définie sur $\mathbb{R}$ par $f(x) = x^2 - 5x + 3$ et k un nombre réel. On souhaite déterminer tous les entiers n compris entre deux entiers donnés a et b $(a < b)$ tels que $f(n) < k$.

1 - -10 est-il solution de $f(n) < 5$? Même question pour 0.

2 - Écrire un script Python appelant un nombre et affichant "oui" s'il est solution de $f(n) < 5$, "non" sinon.

3 - Modifier le script pour appeler k et afficher successivement tous les entiers solutions de $f(n) < k$ sur l'intervalle $[-10, 10]$.

4 - Modifier le script pour demander aussi a et b.

Mots-clefs : Affecter des données à un script (commandes **input** et **eval**), définir une fonction, instruction conditionnelle `if …then …else …`, boucle `pour`, conditions de type booléen, variables booléennes, masques.

Énoncé n° 2.9 [*] : Algorithme d'Euclide

1 - Programmer une fonction qui, étant donnés deux entiers positifs `a` et `b`, retourne leur PGCD calculé selon l'algorithme d'Euclide.
2 - En déduire le calcul de leur PPCM.

Mots-clefs : reste et quotient de la division euclidienne (`%` et `//`), algorithme d'Euclide, boucle `tant que`, importation de fonctions, module `math`.

Chapitre 3 : Listes

Énoncé n° 3.1 [*] : Simuler `n` lancers d'un dé équilibré, sous forme de liste

En utilisant la fonction `randint` du module `random`, simuler, sous forme de liste, `n` lancers d'un dé équilibré.

Mots-clefs : liste, module `random`, commandes `randint` et `append`, boucle `pour`.

Énoncé n° 3.2 [*] : Maximum d'une liste de nombres

1 - Calculer le maximum `M` d'une liste `L` de nombres, de longueur `l` comme suit :
(a) On pose `M=L[0]`.
(b) Ensuite, si `l` $\geqslant 2$, `i` prenant successivement les valeurs 1, …, `l-1`, on pose `M=L[i]` si `L[i] > M`.
On présentera la solution sous la forme d'une fonction.
2 - Application : calculer le maximum d'une liste de 10, puis de 10000 nombres réels engendrée par la fonction `reels()`, cf.(3).
3 - Calculer le minimum `m` de `L` en utilisant le calcul du maximum programmé ci-dessus.

Mots-clefs : boucle `pour`, instruction conditionnelle `if`, appeler une fonction nouvellement programmée, méthode `list.append()` des listes, primitives `min()` et `max()`.

Énoncé n° 3.3 [*] : Ranger une liste de nombres

Une liste de nombres `L` est donnée.
1 - à l'aide de la primitive `min()`, ranger cette liste dans l'ordre croissant (le premier terme `m` de la liste à calculer sera le minimum de `L`, le suivant sera le minimum de la liste `L` dont on aura retiré `m`, *etc*).
2 - Ranger `L` dans l'ordre décroissant.

Mots-clefs : primitives `min()` et `max()`, boucle `pour`, méthodes `list.sort()` et `list.reverse()`, fonction `len()` (pour les listes), primitive `sorted()`.

Énoncé n° 3.4 [*] : Permutation aléatoire d'une liste.

Programmer une fonction qui retourne sous forme de liste une permutation aléatoire de la liste `L = [1, 2, …,n]`.
Méthode imposée : on commencera par tirer au hasard un élément de `L` qui deviendra le premier élément de la permutation aléatoire recherchée et on l'effacera de `L`. Puis on répètera cette opération jusqu'à ce que `L` soit vide.

Mots-clefs : boucle `pour`, fonction `randint()` du module `random`, primitive `len()`, L.append(x), fonction `del()`, `list(range())`.

Chapitre 4 : Matrices (m,)

Énoncé n° 4.1 [**] : Le lièvre et la tortue, jeu équitable ?

On lance un dé équilibré. Si le 6 sort, le lièvre gagne. Sinon la tortue avance d'une case et on rejoue. La tortue gagne si elle parvient à avancer de `n` cases (`n` $\geqslant 1$ donné).

1 - Modéliser et simuler ce jeu à l'aide de matrices-lignes.
2 - Calculer la probabilité pour que le lièvre gagne, que la tortue gagne.
3 - Existe-t-il une valeur de `n` pour laquelle le jeu est équitable ?

Mots-clefs : matrice-ligne, modélisation, mathématiques, probabilités, simulation.

Énoncé n° 4.2 [**] : Maximum et minimum d'une matrice-ligne

1 - Calculer le maximum d'une matrice-ligne de nombres `M` en procédant comme suit :
(a) - on choisit arbitrairement l'un de ces nombres que l'on note `ma` et on efface les nombres $\leqslant$ `ma`.
(b) - s'il n'en reste plus, `ma` est le maximum recherché.
(c) - sinon, on recommence (a) et (b).
2 - Calculer le minimum `mi` de `M`.

Mots-clefs : matrices, masque, maximum et minimum, fonctions primitives `max` et `min`, boucle `tant que`, définition d'une fonction, appel d'une nouvelle fonction.

Énoncé n° 4.3 [***] : Permutations aléatoires d'une matrice à une dimension.

1 - Programmer une fonction qui retourne une permutation aléatoire de la matrice à une dimension `V = array([1, 2, …,n])` sous forme de matrice à une dimension.
Méthode imposée : On commencera par tirer au hasard un élément de `V` qui deviendra le premier élément de la permutation aléatoire `W` recherchée et on l'effacera de `V`. Puis on répètera cette opération tant que `V` n'est pas vide.
2 - Utiliser la fonction précédente pour fabriquer des permutations aléatoires de n'importe quelle matrice à une dimension `M`.

Mots-clefs : fonctions `array()`, `arange()`, `hstack()`, `delete()`, `len()`, `shuffle()` du module `numpy`, `randint()`, `rand()` du module `numpy.random()`, échange d'éléments et extraction d'une sous-matrice d'une matrice à une dimension, `size`, méthode des matrices à une dimension, conversion d'une matrice à une dimension en une liste.

Chapitre 5 : Matrices ou matrices (n,m)

Énoncé n° 5.1 [*] : Matrices croix (fantaisie).

Écrire la matrice carré e M de taille 2n+1 comportant des 1 sur la (n+1)iè me ligne et la (n+1)iè me colonne et des 0 ailleurs.

Mots-clefs : module `numpy` : fonctions `np.zeros(( , ), int)`, `np.ones(( , ), int)`, concaténation hori-

zontale et verticale de matrices (respectivement `np.concatenate(( , , ),axis = 1)` et `np.concatenate(( , , ),axis = 0)`).

Énoncé n° 5.2 [*] : Matrice Zorro (fantaisie)

Écrire la matrice `M` de taille n, définie comme la matrice carrée de taille n comprenant des 1 sur la première ligne, sur la deuxième diagonale et sur la dernière ligne et des 0 partout ailleurs.

Mots-clefs : module `numpy` : fonctions `np.zeros((n, m),int)`, `np.ones((m, ),int)`, `np.arange(p,q,r)`, extraction d'un bloc d'éléments d'une matrice : `M[m1,m2]`.

Énoncé n° 5.3 [*] : Transformations aléatoires de matrices.

1 - Programmer une fonction qui, pour tout entier `n` positif, retourne la matrice carrée dont la première ligne est `1, 2, …, n`, la suivante `n+1, n+2, …, 2n`, et ainsi de suite jusqu'à la dernière ligne [a].
2 - à l'aide de la fonction `shuffle()` de `numpy.random`, effectuer une permutation aléatoire équiprobable des éléments d'une matrice quelconque donnée `M` et afficher la matrice obtenue.

a. qui est donc `n(n-1)+1, n(n-1)+2, …, n*n`

Mots-clefs : fonctions `arange()`, `reshape()` du module `numpy`, fonction `shuffle()` du module `numpy.random`, commande `B = A.flatten()`.

Énoncé n° 5.4 [*] : Calculer la trace d'une matrice.

Soit `M` une matrice à `n` lignes et `m` colonnes.
1 - Calculer la somme `t1` de ses éléments dont l'indice de ligne est égal à l'indice de colonne.
2 - Calculer la somme `t2` de ses éléments dont l'indice de ligne `i` et l'indice de colonne `j` vérifient `i+j=min(n,m)-1`, les lignes et les colonnes étant numérotées à partir de 0).

Mots-clefs : boucle `pour` ; module `numpy` : extraire un bloc d'éléments d'une matrice, changer l'ordre des lignes ou des colonnes d'une matrice, fonctions `shape()`, `diag()`, `sum()`, `trace()` ; module `numpy.random` : fonction `rand()`, module `copy` : fonction `deepcopy`.

Énoncé n° 5.5 [**] : Ce carré est-il un carré magique ?

Convenons qu'une matrice carrée `M` d'ordre `n` est un carré magique d'ordre `n` si ses éléments sont les entiers de 1 à n^2 disposés de sorte que leurs sommes sur chaque ligne, sur chaque colonne et sur les deux diagonales soient égales. La valeur commune de ces sommes est appelée constante magique de `M`.
1 - S'il existe un carré magique d'ordre `n`, combien vaut sa constante magique ?
2 - `LuoShu = array([[4,9,2],[3,5,7],[8,1,6]])` est-il un carré magique ? Si oui, quelle est sa constante magique ?
3 - Même question pour `Cazalas = array([[1,8,53,52,45,44,25,32],`
`[64,57,12,13,20,21,40,33], [2,7,54,51,46,43,26,31],`
`[63,58,11,14,19,22,39,34], [3,6,55,50,47,42,27,30],`
`[62,59,10,15,18,23,38,35], [4,5,56,49,48,41,28,29],`
`[61,60,9,16,17,24,37,36]])`.
4 - Même question pour `Franklin = array([[52,61,4,13,20,29,36,45],`
`[14,3,62,51,46,35,30,19], [53,60,5,12,21,28,37,44],`
`[11,6,59,54,43,38,27,22], [55,58,7,10,23,26,39,42],`
`[9,8,57,56,41,40,25,24], [50,63,2,15,18,31,34,47],`
`[16,1,64,49,48,33,32,17]])`.

Mots-clefs : `V.size`, `M.sum(axis=1)`, `M.sum(axis=0)`, `trace(M)`, `M[arange(n-1,-1,-1), arange(0,n)]` (extraction d'une matrice 1-dimensionnelle de `M`), `hstack((,))`, `min()`, `max()`.

Énoncé n° 5.6 [***] : Transposer une matrice

Programmer une fonction qui, étant donné une matrice numérique `A`, retourne la matrice `B` dont la première colonne est la première ligne de `A`, la seconde la deuxième ligne de `A`, etc. `B` s'appelle la transposée de `A`.

Mots-clefs : Boucle `pour` imbriquée dans une autre, commande `range`, module `numpy`, fonctions `shape()` (dimension d'une matrice), `zeros()`, `vstack()` et `hstack()` (empilements de matrices), `transpose()`, méthode (de l'objet matrice) `.T`.

Énoncé n° 5.7 [****] : Écrire toutes les permutations de `(1, ..., n)`.

1 - `M` désignant une matrice quelconque d'entiers à `li` lignes et `co` colonnes, `n` un entier quelconque, programmer une fonction qui retourne la matrice `Sortie` qui empile verticalement les `co+1` matrices obtenues en adjoignant à `M` une colonne `A` à `li` lignes dont tous les éléments sont égaux à `n`, `A` étant placée d'abord devant `M`, puis entre la première et deuxième colonne de `M`[a], *etc*, jusqu'à ce que `A` devienne sa dernière colonne. La matrice obtenue aura `co+1` colonnes et `(n+1).li` lignes.
2 - Écrire toutes les permutations de `(1, ..., n)`.

a. ce qui revient à décaler `A` d'un cran vers la droite ou d'échanger les deux premières colonnes

---

Mots-clefs : appel de modules et de fonctions, fonctions `ones()`, `zeros()`, `hstack()` du module `numpy`, méthode `shape()` des matrices, fonction `clock()` du module `time`, échange de colonnes d'une matrice, extraction d'une sous-matrice d'une matrice.

Énoncé n° 5.8 [****] : Carrés magiques d'ordre 3.

Combien y a-t-il de carrés magiques d'ordre 3 ?

Mots-clefs : permutations, appeler une fonction, commandes `factorial()`, `clock()`, matrices à une (commandes `min` et `max`, `reshape()`) et deux dimensions (extraction d'une ligne, somme sur les lignes et les colonnes, `trace()`), conversion d'une matrice à une dimension en une matrice à deux dimensions, listes (commande `append()`).

Chapitre 6 : Arithmétique, compter en nombres entiers

Énoncé n° 6.1 [***] : PGCD de `p` entiers positifs

1 - Programmer une fonction qui, étant donnés `p` entiers positifs $a_1, \ldots, a_p$ dont le minimum est noté m, retourne la matrice à une dimension de leurs diviseurs communs en procédant de la manière suivante : retirer de la suite $1, \ldots, m$ les nombres qui ne sont pas des diviseurs de a_1, puis ceux qui ne sont pas des diviseurs de a_2, *etc.*
2 - En déduire leur PGCD.

Mots-clefs : module `numpy`, fonctions `arange()`, `size()`, `sort()`, masque de matrice, `where()`, reste de la division euclidienne, extraction des éléments d'une matrice (m,) dont les indices sont donnés, fonction pré-définie `min()`, module `math`, fonction `gcd` de ce module.

Énoncé n° 6.2 [****] : Crible d'Ératosthène

1 - Programmer une fonction qui, pour tout entier $n \geqslant 2$ donné, retourne les nombres premiers compris entre 2 et `n`, rangés dans l'ordre croissant.
2 - Programmer une fonction qui, étant donné un nombre entier $n \geqslant 2$, retourne suivant le cas "`n` est premier" ou "`n` n'est pas premier".

Mots-clefs : effacer des éléments d'une matrice à une dimension à l'aide d'un masque.

Énoncé n° 6.3 [***] : Factoriser un entier en produit de nombres premiers.

Programmer une fonction qui, étant donné un entier $n \geqslant 2$, retourne sa factorisation en produit de nombres premiers [a].

a. On peut utiliser la fonction `erato()` de l'exercice 6.2.

Mots-clefs : appel d'une fonction, boucles `pour` et `tant que`.

Chapitre 2

Fonctions, boucles, instructions conditionnelles

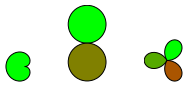
Liste des exercices :

Cet ouvrage est écrit à l'aide de l'environnement `Spyder` de Python. La plupart des graphes y sont tracés à l'aide du module `matplotlib`, l'exception étant l'usage de la tortue (`turtle`)[1]. Au début d'un script contenant un tracé, il faut charger les modules cités ci-dessous. Dans notre environnement, c'est inutile. Aussi, pour qu'un éventuel lecteur ne les oublie pas, nous les avons laissés en commentaire :

```
# from numpy import *
# from matplotlib.pyplot import *
# from pylab import *
```

Avec un réglage différent de `Spyder` ou avec un autre environnement de programmation comme **idle**, il faudra peut-être activer ces commandes.

Représenter une fonction graphiquement est compliqué car les tracés peuvent être l'objet de nombreuses options. Les quelques exercices sur les graphes qui suivent sont des modèles de référence pour débuter, qui se compliquent au fur et à mesure. Il existe une excellente documentation sur `matplotlib` en français, avec beaucoup d'exemples, à savoir [16]. Ce texte en est en grande partie inspiré.



1. pour les tracés qui s'y prêtent

2.1 [*] Pour commencer : des calculs.

Quelle est la valeur de π ? de e ? de $\pi \cdot e$? Calculer $\dfrac{\sin(3) \cdot e^2}{1 + \tan\left(\frac{\pi}{8}\right)}$.

Mots-clefs : constantes pré-définies e, π, calcul numérique, module `math`, fonctions `sin` et `tan`.

Dans Python, il y a des constantes pré-définies, comme e et π, des fonctions standard (utilisables immédiatement) [2] et une foule d'autres fonctions qui sont regroupées par thème dans des bibliothèques de fonctions ou `modules`. Par exemple, `sin`, `tan` sont des fonctions du module `math`. Si dans un même script, on veut se servir de la constante e et de la fonction `sin`, il faudra les importer, en important tout le module `math` comme ceci :

```
from math import *
```

ou mieux [3], en important seulement e et `sin`, comme cela :

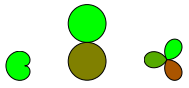
```
from math import e, sin
```

Dans l'exemple qui suit, on a oublié d'importer π, ce qui provoque un message d'erreur :

```
from math import e,sin
e
Out[2]: 2.718281828459045
sin(pi/6)
Traceback (most recent call last):
  File "<ipython-input-3-dbaab9b2f48b>", line 1, in <module>
    sin(pi/6)
NameError: name 'pi' is not defined
```

Nous ne soucions pas du format des nombres dans cette prise de contact [4]. Les règles de calcul sont les règles usuelles. Le problème posé est trivial. Comme il y a peu à écrire, on peut travailler directement dans la console [5]. Ouvrons-en une nouvelle :

```
from math import e, pi, sin, tan
pi
Out[2]: 3.141592653589793
e
Out[3]: 2.718281828459045
pi*e
Out[4]: 8.539734222673566
sin(3)*e**2/(1+tan(pi/8))
Out[5]: 0.7373311103636621
```



2. En anglais, les "built-in functions", voir [10].
3. pour économiser du temps, de l'espace-mémoire, *etc.*
4. Nous utilisons donc les réglages de Python par défaut.
5. Normalement, on écrit dans l'interpréteur, qui est un traitement de texte adapté à Python. On écrit plus vite, les retraits - très importants en programmation Python - se font automatiquement, on peut corriger facilement, *etc.*

2.2 [*] Pour commencer : graphes simples.

1 - Tracer le segment de droite d'extrémités (1,3) et (15,-2), dans un repère orthonormé.
2 - Tracer le graphe de la fonction `sinus` quand la variable varie de 1 à 15 radians.

Mots-clefs : Instructions simples de tracer des graphes, commandes `linspace(a,b,n)` et `array([a,…,x])` du module `numpy`, fonctions vectorisées.

Solution

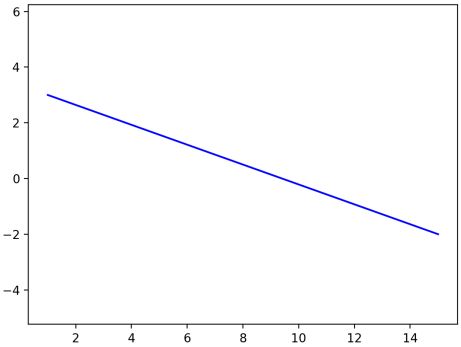
Cet exercice[6] peut paraître débile. En fait, il est important, tout graphe se réduisant à une succession de segments de droite. En effet, pour tracer un graphe, `Python` calcule les coordonnées de points du graphe et relie les points consécutifs par un segment (voir 2.3). Si ces points sont assez nombreux, les imperfections de nos yeux font que l'on voit une courbe lisse et non une ligne polygonale. De plus, il y a mieux : `Python`[7] ne trace pas vraiment des segments de droite mais des ensembles de pixels plus ou moins bien disposés le long du segment à tracer, voir [17]. Heureusement, grâce à nos mauvais yeux, tout s'arrange.

1 - Voici un script simple solution du problème :

```
# from numpy import *
# from matplotlib.pyplot import *
# from pylab import *
figure()# Création d'une figure.
x = array([1,15])# x de type 'vecteur' (= matrice à une dimension).
y = array([3,-2])# y de type 'vecteur' (= matrice à une dimension).
plot(x,y,color='blue')# Le tracé sera bleu.
axis('equal')# Le repère sera orthonormé.
savefig("at34gdroite")# Sauvegarde de la figure (fichier at34gdroite.png).
show()# Sinon, la figure n'apparaît pas.
```

Commentaires :

1 - Un vecteur est une matrice ('`array`') pour `Python`, qui n'a qu'une dimension (le nombre de ses éléments)[8]. `x` ne comprend ici que les abscisses de l'origine et de l'extrémité du segment à tracer, de même `y` avec les ordonnées.
2 - Les commandes `color='blue'`, `axis('equal')` et `savefig("at34gdroite")` ne sont pas indispensables. Voici la figure obtenue grâce à la commande `savefig("at34gdroite"` :



6. qui a pour but de fournir des modèles de tracés de graphes
7. ainsi que tous les autres logiciels de calcul
8. Une matrice à une dimension est très perturbant pour un mathématicien et est source d'erreurs de programmation.

Vers le graphe d'une fonction 'quelconque' :

Dans le script ci-dessous, après avoir remarqué que la droite qui porte le segment à tracer a pour équation

$$y = -5/14 * x + 47/14$$

nous traçons le graphe de la fonction

$$f(x) = -5/14 * x + 47/14$$

entre ses points d'abscisses 1 et 15 :

```
# from numpy import *
# from matplotlib.pyplot import *
# from pylab import *
figure()# Création d'une figure.
def f(x):
    return -5/14*x+47/14
x = linspace(1,15,2)# Commande du module numpy qui produit array([1,15]).
y = f(x)# Commande du module numpy qui produit array([3,-2]).
plot(x,y,color='blue')# Le tracé sera bleu.
axis('equal')# Le repère sera orthonormé.
savefig("at34gdroite")# Sauvegarde de la figure (fichier at34gdroite.png).
show()# Sinon, la figure n'apparaît pas.
```

Ce script est un modèle à retenir. On obtient la même figure que précédemment au moyen d'instructions qui sont certes plus compliquées mais qui conviennent à la représentation graphique d'une fonction 'quelconque'.

Précisions supplémentaires sur le script ci-dessus :

• On constate que f est définie par les instructions

```
def f(x):
    return -5/14*x+47/14
```

• La commande `linspace(a,b,n)` du module `numpy` produit `n` nombres dont `a` et `b`, qui partagent l'intervalle `[a,b]` en `n-1` intervalles de même longueur. Par exemple `linspace(1,15,2)` produit `array([1,15])`.

• Si `x=linspace(a,b,n)`, `f(x)` produit les valeurs de `f` aux `n` points de `x`. On dit que `f` est une fonction vectorisée.

Vers une figure de meilleure qualité

La figure obtenue ci-dessus est très sommaire. Son plus grave défaut est que les axes de coordonnées n'apparaissent pas. On peut repérer les abscisses sur la graduation du bas, les ordonnées sur la graduation de gauche. Pour corriger cela, on peut effacer les bords du cadre à droite et en haut et faire glisser verticalement l'axe gradué du bas et horizontalement l'axe gradué de gauche pour les faire passer par le point (0,0). On obtient ainsi les axes de coordonnées. C'est l'objet du groupe de commandes suivant :

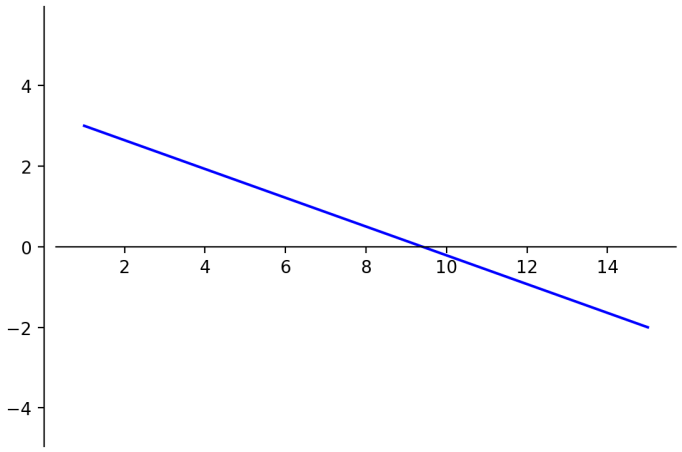
```
# Positionnement des axes de coordonnées
ax = gca()
ax.spines['right'].set_color('none')# Fait disparaître le
# bord droit du cadre.
ax.spines['top'].set_color('none')# Fait disparaître le
# bord supérieur du cadre.
ax.xaxis.set_ticks_position('bottom')# Les traits de graduation
# sur l'axe des abscisses seront en-dessous.
ax.spines['bottom'].set_position(('data',0))# Bouge verticalement le
# bord inférieur du cadre jusqu'à l'ordonnée 0.
ax.yaxis.set_ticks_position('left')# Les traits de graduation
# sur l'axe des ordonnées seront à gauche.
```

```
ax.spines['left'].set_position(('data',0))# Bouge horizontalement le bord
# gauche du cadre jusqu'à l'abscisse 0.
```

Testons donc le script obtenu après ces divers perfectionnements :

```
# from numpy import *
# from matplotlib.pyplot import *
# from pylab import *
figure()
def f(x):
    return -5/14*x+47/14
x = linspace(1,15,2)
y = f(x)
plot(x,y,color='blue')
axis('equal')
ax = gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
savefig("at34gdroite2")
show()
```

En exécutant ce sript, on obtient la figure :



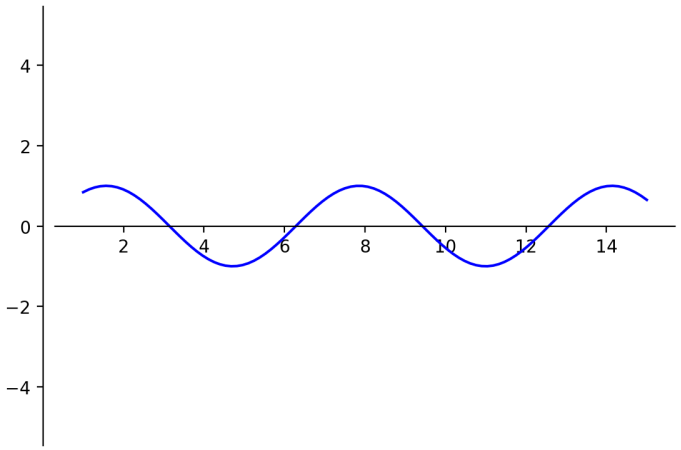
qui est nettement plus satisfaisante que la précédente.

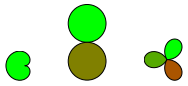
2 - Il suffit de remplacer, dans le script précédent, la fonction `f` par `sin`[9]. Voici successivement le script utilisé et le graphe obtenu :

```
# from numpy import *
# from matplotlib.pyplot import *
# from pylab import *
from math import sin
figure()
```

9. `sinus` est une fonction du module `math`.

```
x = linspace(1,15,100)# On calcule 100 valeurs du sinus équiréparties
# entre 1 et 15.
y = sin(x)
plot(x,y,color='blue')
axis('equal')
ax = gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
savefig("at34gsin")
show()
```





2.3 [*] Représentations graphiques avec Python

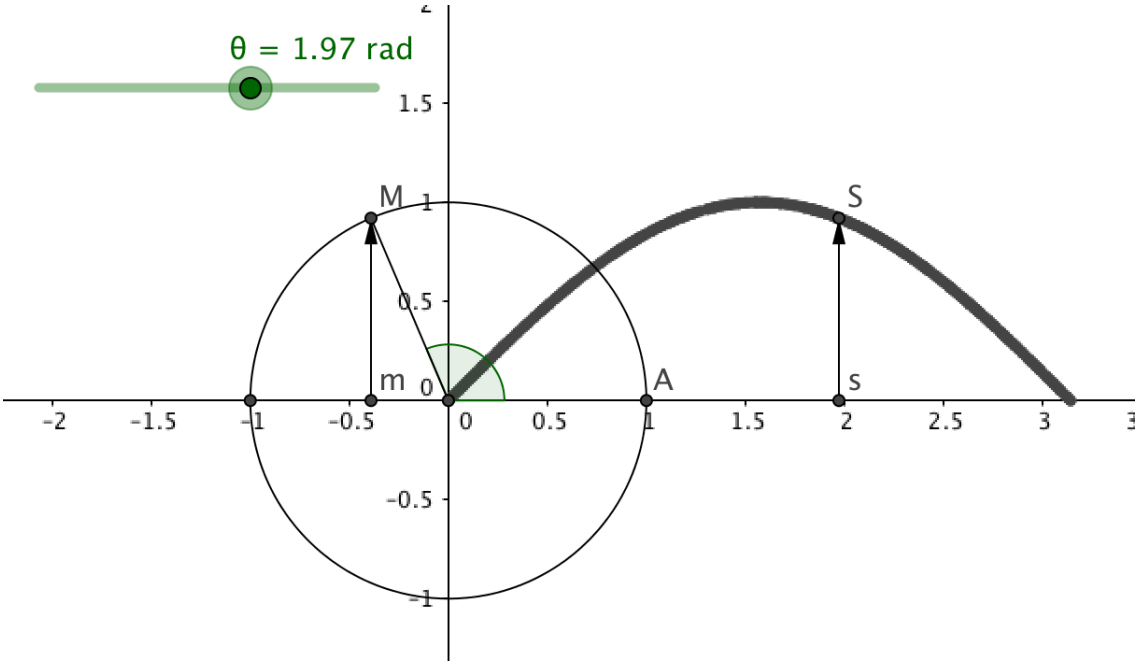
Pour tracer un graphique, Python en calcule des points consécutifs et les relie par un segment. Habituellement, on obtient ainsi une bonne approximation de la courbe à tracer quand il y a suffisamment de points car alors, nos yeux ne font plus la différence.
On se propose d'étudier des approximations du graphique de la fonction `sin()` lorsque la variable `x` varie sur l'intervalle `[0,`π`]`.
1 - Tracer l'approximation du graphique obtenue en calculant ses points d'abscisse 0, $\pi/2$ et π.
2 - Pour `n` valant successivement 3, 6, 10, 20, tracer l'approximation du graphique obtenue en calculant les points dont les abscisses sont données par la commande `linspace(0,pi,n+1)` (cette représentation graphique est une ligne polygonale à `n` côtés) .

Mots-clefs : Représentation graphique d'une fonction, repère orthonormé, commande `axis('equal')`, d'autres commandes graphiques.

Bien sûr, il est souvent impossible de représenter graphiquement une fonction parce que dans le cas courant, une représentation graphique a une infinité de points [10]. C'est pourquoi les logiciels de calcul se contentent de représenter ces graphiques par des lignes polygonales [11]. Cet exercice peut être considéré comme une suite de l'exercice 2.2.

Avant de commencer

On peut représenter simplement une sinusoïde à l'aide d'un logiciel de géométrie dynamique [12]. Le point M est repéré sur le cercle trigonométrique par l'angle de vecteurs $\theta = (\overrightarrow{OA}, \overrightarrow{OM})$. Les coordonnées de M sont $(\cos\theta, \sin\theta)$. Le point S de coordonnées $(\theta, \sin\theta)$ engendre la sinusoïde quand θ parcourt le segment $[0, \pi]$.



Solution

1 - On peut utiliser le script suivant :

```
from math import sin
from numpy import *
```

10. Un ordinateur ne peut exécuter une infinité de commandes.
11. dont les côtés sont eux-mêmes des successions de pixels, ce qui n'apparaîtra pas dans cet exercice
12. L'image ci-dessous a été créée à l'aide de GeoGebra.

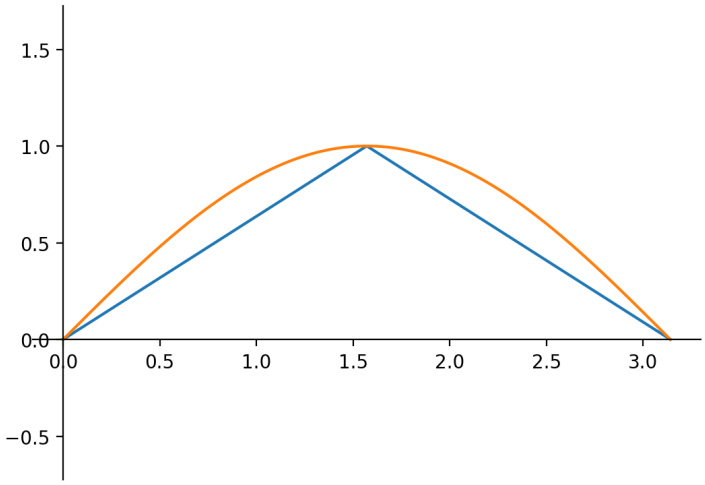
```
from matplotlib.pyplot import *

figure()
x = array([0,pi/2,pi])# ou x = linspace(0,pi,3).
y = sin(x)
plot(x, y)
xx=linspace(0,pi,100)
yy=sin(xx)
plot(xx,yy)

axis('equal')
ax = gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_position(('data',0))
ax.spines['left'].set_position(('data',0))
# ax.xaxis.set_ticks_position('bottom')
# ax.yaxis.set_ticks_position('left')

savefig('at1s1f1')
show()
```

Ce script [13] répond à la question et produit en plus une représentation graphique acceptable [14] de `sin` entre 0 et π. C'est l'approximation la plus simple possible. On obtient :



La ligne polygonale bleue (deux côtés) ne fait pas vraiment penser à la sinusoïde. Ce n'est pas une bonne approximation de la sinusoïde, mais ce n'est pas franchement mauvais.

2 - Le script suivant définit une fonction appelée `sinpolygone(n)` qui retourne comme ci-dessus deux tracés sur la même fenêtre graphique : la représentation graphique de `sin` ainsi que la ligne polygonale

`(0,0) --> (pi/n,sin(pi/n)) --> (2pi/n,sin(2pi/n)) --> ... --> (pi,sin(pi))`.

qui est censée en être une approximation.

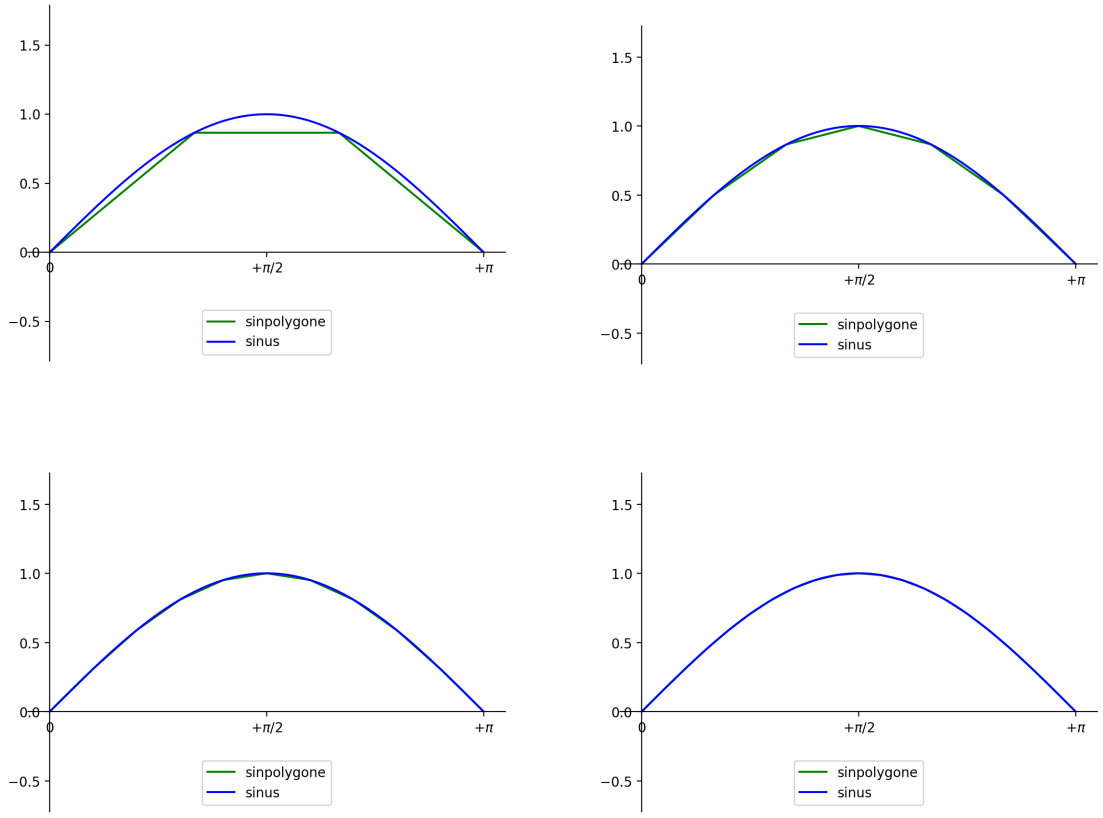
```
from math import sin
from numpy import *
```

13. Pour tout ce qui concerne le module `matplotlib`, voir [16]. Python dispose d'autres modules graphiques.
14. On a déjà expliqué pourquoi il n'est pas possible de représenter graphiquement `sin`.

```
from matplotlib.pyplot import *

def sinpolygone(n):# n est le nombre de côtés de la ligne polygonale.
    figure()
    x=linspace(0,pi,n+1)
    y=sin(x)
    xx=linspace(0,pi,100)
    yy=sin(xx)
    plot(x,y,color='green', label="sinpolygone"), plot(xx,yy,color='blue',
    label="sinus")
    axis('equal')
    xticks([0, pi/2, pi],[ r'$0$', r'$+\pi/2$', r'$+\pi$'])
    ax = gca()
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.spines['bottom'].set_position(('data',0))
    ax.spines['left'].set_position(('data',0))
    legend(loc='lower center')
    savefig('at1s4f1')
    show()
```

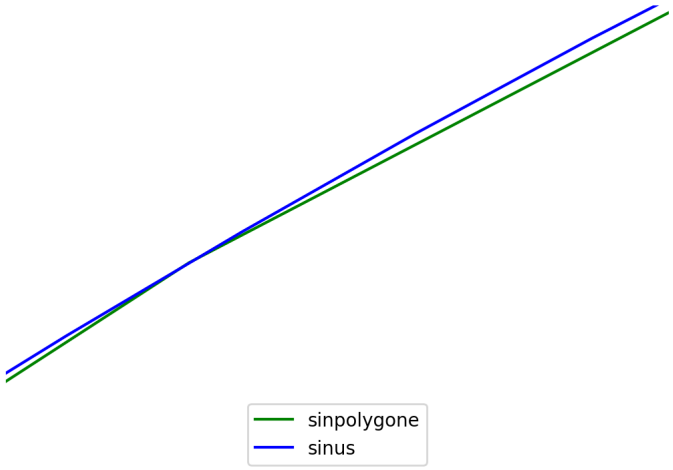
Nous avons regroupé ci-dessous en un tableau à 2 lignes et 2 colonnes l'approximation de la sinusoïde obtenue pour n = 3 (3 côtés), puis n = 6, 10, 20 (de gauche à droite et de haut en bas), en tapant `sinpolygone(3)`, *etc*, dans la console.[15]



On constate que pour `n=20` (et même pour des valeurs plus petites de `n`), on ne distingue plus la sinusoïde

15. Les tracés ont été ordonnés à l'aide de `Latex`.

de la ligne polygonale. C'est évidemment une impression fausse : voici un agrandissement d'une partie de ce dernier tracé (capture d'écran) :



Complémments :

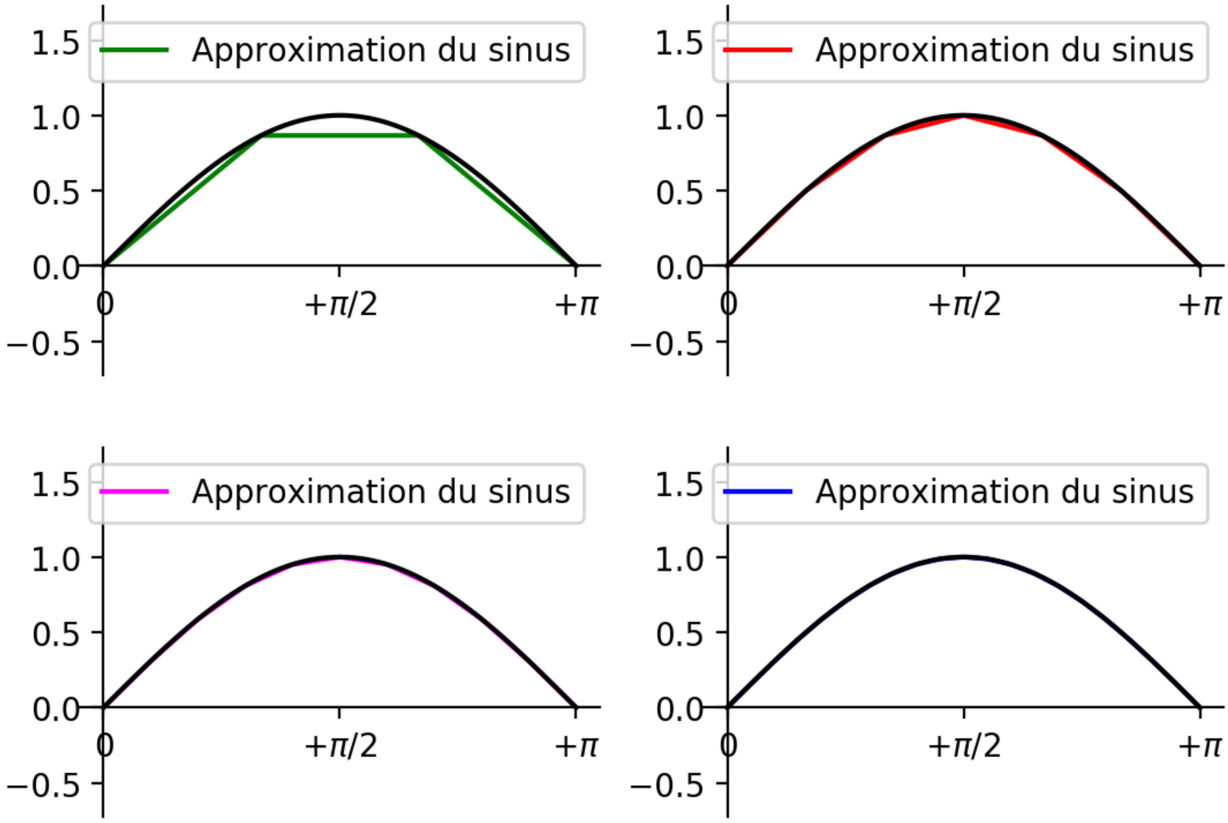
Nous aurions pu faire le tableau des 4 graphes à l'aide de la commande `subplot` de `matplotlib`, par exemple avec le script suivant :

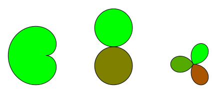
```
from math import *
from numpy import *
from matplotlib.pyplot import *

figure()
A=array([3,6,10,20])
l=['green', 'red','magenta', 'blue']
for i in range(1,5):
    xx=linspace(0,pi,100)
    yy=sin(xx)
    subplot(2,2,i)
    x=linspace(0,pi,A[i-1]+1)
    y=sin(x)
    axis('equal')
    plot(x,y,color=l[i-1], label="Approximation du sinus")
    plot(xx,yy,color='black')
    xticks([0, pi/2, pi],[ r'$0$', r'$+\pi/2$', r'$+\pi$'])
    ax = gca()
    # ax.xaxis.set_ticks_position('bottom')
    # ax.yaxis.set_ticks_position('left')
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.spines['bottom'].set_position(('data',0))
    ax.spines['left'].set_position(('data',0))
    legend(loc='upper right')

  show()
savefig('at1s2f1')
```

Voici le tableau de tracés obtenu en exécutant le script précédent :


Approximation du sinus
1.5
1.0
0.5
0.0
−0.5
0
+π/2
+π
Approximation du sinus
1.5
1.0
0.5
0.0
−0.5
0
+π/2
+π
Approximation du sinus
1.5
1.0
0.5
0.0
−0.5
0
+π/2
+π
Approximation du sinus
1.5
1.0
0.5
0.0
−0.5
0
+π/2
+π



2.4 [*] Programmer des fonctions : valeur absolue, volume du cône

1 - Programmer une fonction qui retourne la valeur absolue de tout nombre.
2 - Programmer une fonction qui retourne le volume de tout cône droit à base circulaire, de hauteur `h` cm et de rayon `r` cm.

Mots-clefs : programmer une fonction, instruction conditionnelle `if … else …`, `return`, importer une constante ou une fonction d'un module de Python.

Solution

1 - Il est difficile de trouver un exercice plus simple ! Appelons `vabsolue` cette fonction. Nous la définissons par le script :

```
# La fonction vabsolue retourne la valeur absolue de tout nombre x.
def vabsolue(x):
    if x >= 0:
        return x
    else :
        return -x
```

Exemple : calculons la valeur absolue de -2.17 (nombre réel) et de -125 (nombre entier), après avoir chargé `vabsolue` dans la console [16] :

```
runfile('Chemin de la fonction vabsolue')
# Le chemin détaillé s'inscrit automatiquement quand on exécute le script.
vabsolue(-2.17)
Out[2]: 2.17
vabsolue(-125)
Out[3]: 125
```

On obtient un nombre réel dans le premier cas, un entier dans le second. Bien sûr, pour calculer une valeur absolue, on utilise habituellement la fonction pré-définie `abs(x)` (voir [10]).

2 - La formule qui donne le volume v du cône, ici en cm^3, est $v = \dfrac{\pi r^2 h}{3}$. Programmer une fonction de deux variables ne pose pas de problème :

```
# La fonction volcone retourne le volume d'un cône droit
# à base circulaire de rayon r et de hauteur h.
# Charger pi du module math.
def  volcone(r,h):
    return (pi*r**2*h)/3
```

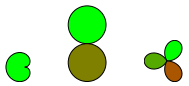
Avant d'exécuter `volcone`, il faut charger la valeur de la constante π du module `math`.
Exemple :

```
from math import pi # Importation de pi.
runfile('Chemin de la fonction volcone')
volcone(1.88,3.17)
Out[3]: 11.732851629089138
pi*1.88*1.88*3.17/3 # Vérification (inutile) par calcul direct.
```

16. Nous supprimons systématiquement les lignes blanches pour gagner de la place.

Out[6] : 11.732851629089138

On aurait pu importer le module `math` tout entier par `from math import *`.



2.5 [*] Trinôme du second degré

Résoudre l'équation du second degré (E) : $ax^2 + bx + c = 0$ $(a \neq 0)$

Mots-clefs : Fonction `sqrt`, à importer du module `math`, sortie d'une fonction par `return` ou `print`, instruction conditionnelle `if: …elif:…else:….`

Solution

La programmation de la solution doit envisager les 3 cas : $b^2 - 4ac > 0$, $b^2 - 4ac = 0$ et $b^2 - 4ac < 0$, d'où l'utilisation de l'instruction conditionnelle `if: …elif:…else:….` Les scripts ci-dessous permettront aux professeurs de proposer des trinômes moins banals que les trinômes habituels à coefficients entiers dont la résolution est plus ou moins évidente. Les deux scripts se distinguent par leurs sorties : la fonction `trinome` (instruction `return`) retourne une `liste`, éventuellement vide[17], que l'on peut utiliser dans des calculs ultérieurs ; la seconde `trinomebis` (instruction `print`) donne les solutions de (E), s'il y en a, au moyen d'une phrase en français ; mais il ne semble pas qu'on puisse récupérer ces solutions pour des calculs ultérieurs. La sortie de `trinomebis` est de type `nonetype`.

```
# La fonction trinome ci-dessous retourne la liste des solutions de
# l'équation (E) : ax^2+bx+c=0 (où a est un réel non nul).
# L'utilisation de la fonction racine carrée impose qu'elle soit
# importée du module math.
from math import sqrt
def trinome(a,b,c):
    delta=b**2-4*a*c
    if delta > 0:
        x1 = (-b+sqrt(delta))/(2*a)
        x2 = (-b-sqrt(delta))/(2*a)
        return [x1,x2]
    elif delta == 0:
        x12 = -b/(2*a)
        return [x12]
    else:
        return []
```

Voici quelques exemples d'application. On a ajouté le type de la solution et le calcul de la somme des carrés des racines. Les commentaires n'ont pas été retournés par Python. Ils ont été ajoutés lors de la rédaction.

```
runfile('Chemin de la fonction trinome')
L=trinome(1,2*(2+sqrt(3)),7+4*sqrt(3))# Exemple n°1.
L
Out[3]: [-3.732050807568877]# La solution unique de (E).
type(L)
Out[4]: list
L[0]**2# Calcul de la somme des carrés des solutions.
Out[5]: 13.928203230275509# La somme des carrés des solutions de (E).
L = trinome(-10,2*(2+sqrt(3)),7+4*sqrt(3))# Exemple n°2.
L
Out[7]: [-0.8645761419679951, 1.6109863034817706]# Les 2 solutions de (E).
type(L)
Out[8]: list
```

17. Il s'agit, suivant le cas, de 0 (liste vide), 1 ou 2 nombres séparés par une virgule et placés entre crochets

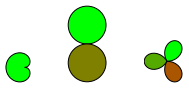
```
L[0]**2+L[1]**2# Calcul de la somme des carrés des solutions.
Out[9]: 3.3427687752661224# La somme des carrés des solutions de (E).
L = trinome(10,2*(2+sqrt(3)),7+4*sqrt(3))# Exemple n°3.
L
Out[11]: []# (E) n'a pas de solution.
type(L)
Out[12]: list
```

Remplaçons `return` par `print` :

```
# La fonction trinome ci-dessous retourne les solutions de
# l'équation (E) : ax^2+bx+c=0 (où a est un réel non nul).
# L'utilisation de la fonction racine carrée impose qu'elle soit
# importée du module math.
from math import sqrt
def trinomebis(a,b,c):
    delta=b**2-4*a*c
    if delta > 0:
        x1 = (-b+sqrt(delta))/(2*a)
        x2 = (-b-sqrt(delta))/(2*a)
        print('(E) a deux solutions ',x1,' et ',x2,'.')
    elif delta == 0:
        x12 = -b/(2*a)
        print('(E) a une solution : ',x12,'.')
    else:
        print("(E) n'a  pas de solution.")
```

Voici un exemple de chaque cas :

```
runfile('Chemin de la fonction trinomebis')
L = trinomebis(1,0,-2)
(E) a deux solutions  1.4142135623730951  et  -1.4142135623730951 .
type(L)
Out[3]: NoneType
L = trinomebis(1,2*(2+sqrt(3)),7+4*sqrt(3))
(E) a une solution :  -3.732050807568877 .
type(L)
Out[5]: NoneType
L = trinomebis(1,2,2)
(E) n'a  pas de solution.
type(L)
Out[7]: NoneType
```



2.6 [**] Minimum de 2, 3, 4 nombres (instructions conditionnelles)

1.a - Programmer une fonction qui retourne le minimum de 2 nombres. 1.b - Programmer une fonction qui retourne le minimum et le maximum de 2 nombres. 2 - Programmer une fonction qui retourne le minimum de 3 nombres. 3 - Programmer une fonction qui retourne le minimum de 4 nombres.

Mots-clefs : Programmation de fonctions, instruction conditionnelle `if … else …`, `return`, négation d'une proposition logique, importation de constantes et de fonctions pré-définies.

Solution

Quand on veut calculer le minimum d'une liste de nombres, on utilise bien évidemment la fonction pré-définie `min` de Python, voir [10]. Ici, notre but est de calculer le minimum de 2, 3 ou 4 nombres sans utiliser cette fonction.

1.a - Cette question est élémentaire :

```
# La fonction min2nom retourne le minimum de 2 nombres a et b.
def min2nom(a,b):
    if a <= b:
        return a
    else:
        return b
```

Application :

```
runfile('Chemin de la fonction min2nom')
min2nom(-7.17,-7.28)
Out[5]: -7.28
```

2.b - Ce n'est guère plus compliqué :

```
# minmax2nom retourne le minimum et le maximum de 2 nombres a et b.
def minmax2nom(a,b):
    if a <= b:
        return [a,b]
    else:
        return [b,a]
```

Remarques :
a - On a choisi ici de faire apparaître le résultat sous forme d'une `liste` de 2 nombres.
b - Si `L` est une liste de nombres, `max(L) = -min(-L)`. Savoir calculer un minimum suffit.
Application :

```
runfile('Chemin de la fonction minmax2nom')
minmax2nom(1.7,-3.84)
Out[7]: [-3.84, 1.7]
```

Si on remplace [a,b] et [b,a] après la commande `return` du script par a,b et b,a, le résultat apparaîtra sous forme de tuple. Dans l'exemple ci-dessus, on obtiendra (-3.84, 1.7) [18].

18. On peut sans dommage escamoter les notions de `liste` et de `tuple`.

3 - Ça se complique un peu car nous utilisons ci-dessous des instructions conditionnelles emboîtées. C'est aussi plus intéressant. Par exemple, on utilise

$$(a \geqslant b) \text{ ou } (a \geqslant c) \quad \Longleftrightarrow \quad \neg\,(a < b \text{ et } a < c)$$

(la négation de "l'un ou l'autre", c'est "ni l'un, ni l'autre").

```
# La fonction min3nom retourne le minimum de 3 nombres a, b et c.
def min3nom(a,b,c):
    if (a > b) or (a > c):
        if b <= c:
            return b
        else:
            return c
    else:
            return a
```

On fait très attention, bien sûr, aux indentations.
Application :

```
runfile('Chemin de la fonction min3nom')
from math import pi
min3nom(-2.1,3.2,-pi)
Out[10]: -3.141592653589793
```

Remarques :
On a dû importer la valeur de `pi` du module `math`.

4 - Le problème paraît plus difficile car il y aura plus d'instructions conditionnelles emboîtées ainsi que des difficultés logiques. Voici une solution commentée :

```
# min4nombis retourne le minimum de 4 nombres a, b, c et d.
def min4nombis(a,b,c,d):
    if (a >= b) or (a >= c) or (a >= d):
        # Le minimum est égal à b ou c ou d.
        if (b >= c) or (b >= d):
            # Le minimum est égal à c ou d.
            if c >= d :
                return d
            else:
                return c
        else :
            return b
    else:
        return a
```

Cela repose sur l'équivalence logique :

$$(a \geqslant b) \text{ ou } (a \geqslant c) \text{ ou } (a \geqslant d) \quad \Longleftrightarrow \quad \neg\,(a < b \text{ et } a < c \text{ et } a < d)$$

Application : dans l'exemple ci-dessous, nous devons importer la fonction `cos` et la constante `pi` du module `math`.

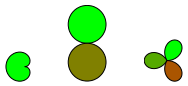
```
runfile('Chemin de la fonction min4nombis')
from math import pi, cos# Importation de la constante pi et de la fonction cos.
min4nombis(3.14,-2.71,1.4142,-cos(pi/6))
Out[3]: -2.71
```

C'est plus simple si l'on se permet d'utiliser la fonction `min3nom` [19], qu'il faudra alors importer [20] :

```
# min4nom retourne le minimum de 4 nombres a, b, c et d.
# Elle utilise la fonction min3nom.
def min4nom(a,b,c,d):
    if a >= b:
        return min3nom(b,c,d)
    else:
        return min3nom(a,c,d)
```

Application : pour reprendre l'exemple ci-dessus, importons donc, outre la fonction `min3nom`, `cos` et `pi` du module `math`.

```
runfile('Chemin de la fonction min4nom')
from min3nom import *
from math import cos,pi
min4nom(3.14,-2.71,1.4142,-cos(pi/6))
Out[4]: -2.71
```



19. De même, on aurait simplifié le script définissant `min3nom` en utilisant la fonction `min2nom`.
20. si on ne l'a pas définie ou utilisée au cours de la même session

2.7 [***] Alignement de 3 points

On se donne, à l'aide de leurs coordonnées, 3 points distincts A, B et C d'un plan rapporté à un repère orthonormé. On appelle d_1, d_2 et d_3 les distances de B à C, de A à C et de A à B. On peut supposer que $d_1 \leqslant d_2 \leqslant d_3$.
1 - Démontrer que les points sont alignés si et seulement si $d_3 = d_1 + d_2$.
2 - Sans utiliser la fonction racine carrée, programmer un algorithme qui retourne, à partir de la liste des coordonnées de A, B et C, "Les points sont alignés" ou "Les points ne sont pas alignés", suivant le cas.

Mots-clefs : saisie de données, affectation de variables, additions et multiplications de nombres, problèmes équivalents. Dans les commentaires sur le test de l'égalité de 2 nombres : équation d'une droite.

On peut supposer que $d_1 \leqslant d_2 \leqslant d_3$ parce que l'on peut échanger, si nécessaire, les lettres A, B et C. L'échange éventuel de ces lettres est effectué ci-dessous par la commande `sorted([d1ca,d2ca,d3ca])`.

Solution

Dans cet exercice, les calculs sont faciles. Les difficultés sont ailleurs.

1 - On peut s'appuyer sur un peu de géométrie : 2 cercles dont les centres sont distants de d et de rayons r_1 et r_2 plus petits que d sont tangents si et seulement si $d = r_1 + r_2$.

2 - On calcule facilement le carré d'une distance à l'aide d'additions et de multiplications tandis que le calcul d'une distance n'est pas élémentaire, voir les commentaires, plus loin.

On sait bien que $a = b$ n'est pas équivalent à $a^2 = b^2$. Pourtant, au moyen de 2 élévations au carré, dans le cas particulier de l'exercice présent, $d_1 + d_2 = d_3$ équivaut à $4d_1^2d_2^2 = (d_3^2 - d_1^2 - d_2^2)^2$. Démontrer cette équivalence est en soi un exercice intéressant. On en déduit une solution de notre problème sous forme de script :

```
ent=eval(input('Liste des coordonnées de A, B et C :'))
d3ca = (ent[2]-ent[0])**2+(ent[3]-ent[1])**2
# d3ca est le carré de la distance de A à B.
d2ca = (ent[0]-ent[4])**2+(ent[1]-ent[5])**2
# d2ca est le carré de la distance de C à A.
d1ca = (ent[4]-ent[2])**2+(ent[5]-ent[3])**2
# d1ca est le carré de la distance de B à C.
L=sorted([d1ca,d2ca,d3ca])
if 4*L[0]*L[1]==(L[2]-L[0]-L[1])**2 :
    print('les points A, B et C sont alignés')
else :
    print('les points A, B et C ne sont pas alignés')
```

Par exemple, exécutons ce script lorsque la liste des entrées est égale à `[-1,3,2,0,4,-2]`, puis à `[-4.74, 34.46, 0.17, 6.58, 5, -5.32]`. On obtient :

```
Liste des coordonnées de A, B et C :[-1,3,2,0,4,-2]
les points A, B et C sont alignés
```

```
Liste des coordonnées de A, B et C :[-4.74, 34.46, 0.17, 6.58, 5, -5.32]
les points A, B et C ne sont pas alignés
```

Le premier essai est évident car B et C se trouvent manifestement sur la droite de pente `-1` qui passe par `A`. Pour le deuxième cas, il pourrait être utile de faire un graphe.

Fin de la solution

Commentaires

1 - Calculer une distance impose d'utiliser la fonction racine carrée qui appartient au module `math`. Il faudra l'importer. Voici ce qui arrive quand on oublie l'importation :

```
sqrt(2)
Traceback (most recent call last):
  File "<ipython-input-9-40e415486bd6>", line 1, in <module>
    sqrt(2)
NameError: name 'sqrt' is not defined
```

La fonction `sqrt` est inconnue ! Quand on importe le module en question, ça marche :

```
from math import *# ou from math import sqrt
sqrt(2)
Out[2]: 1.4142135623730951
```

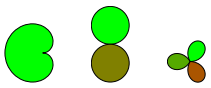
2 - Faisons un autre essai du script ci-dessus :

```
Liste des coordonnées de A, B et C :[2.4, 0.197, -1.16, -0.2658, 7.7, 0.886]
les points A, B et C ne sont pas alignés
```

Ce résultat est faux. Il est en effet facile de vérifier à la main que les points A, B et C se trouvent sur la droite d'équation `y=0.13*x-0.115`. Vérifions avec Python :

```
0.13*2.4-0.115==0.197
Out[1]: True
0.13*(-1.16)-0.115==-0.2658
Out[4]: True
0.13*7.7-0.115==0.886
Out[2]: False
0.13*7.7-0.115
Out[3]: 0.8860000000000001
```

On constate que pour Python, `C` n'est pas sur la droite `(AB)`. Cela provient du fait que Python calcule mal la quantité `0.13*7.7-0.115`. Il ne trouve pas 0.886 mais 0.8860000000000001, ce qui est faux. Ce problème est imparable[21]. Habituellement, on se contente d'une quasi-égalité : on considère que deux nombres x et y sont égaux si $|x - y| \leqslant \varepsilon = 2.220446049250313 * 10^{-16}$.



21. Voir [1], chapitre 4, 1 : « Les nombres en notation scientifique ».

2.8 [**] Affectation de données

Soit f une fonction définie sur $\mathbb{R}$ par $f(x) = x^2 - 5x + 3$ et k un nombre réel. On souhaite déterminer tous les entiers n compris entre deux entiers donnés a et b ($a < b$) tels que $f(n) < k$.

1 - -10 est-il solution de $f(n) < 5$? Même question pour 0.

2 - Écrire un script Python appelant un nombre et affichant "oui" s'il est solution de $f(n) < 5$, "non" sinon.

3 - Modifier le script pour appeler k et afficher successivement tous les entiers solutions de $f(n) < k$ sur l'intervalle $[-10, 10]$.

4 - Modifier le script pour demander aussi a et b.

Mots-clefs : Affectation de données (commandes **input** et **eval**), instruction conditionnelle `if … then … else …`, boucle `pour`, instruction `print`, itérateur `range`. On admet que `for n in range(a,b+1) :` où `a` et `b` sont des entiers tels que `a < b` donne successivement à `n` les valeurs `a`, `a+1`, …, `b`.

Solution

1 - Directement dans la console :

```
(-10)**2-5*(-10)+3  <  5
Out[1]: False
```

Ceci permet d'introduire les conditions de type booléen et les variables booléennes[22]. Ici, la variable booléenne prend la valeur `False`, ce qui signifie que l'inégalité `f(-10) < 5` est fausse.

2 - Le script ci-dessous convient :

```
n=eval(input('la valeur de n est :'))
if n**2 - 5*n + 3 < 5:
    print('oui')
else :
    print('non')
```

Appliquons ce script[23] pour `n = -4` après l'avoir chargé dans la console de Python :

```
la valeur de n est : -4
non
```

Remarque : La commande `n=input('la valeur de n est :')` introduirait `n` comme une chaîne de caractères et non un nombre entier. Il faut l'«évaluer» à l'aide de la commande `eval`.

3 - Cette question est une répétition fastidieuse de la précédente. Les répétitions seront assurées par une boucle `pour`.

```
k=eval(input('la valeur de k est :'))
for n in range(-10,11):
    if n**2 - 5*n + 3 < k:
        print(n)
```

Faisons l'essai pour `k = 17`. Ce qu'on obtient n'est pas très joli :

22. Voir [1], chapitre 1, 7 La structure conditionnelle.
23. qui a été écrit dans l'interpréteur

```
la valeur de k est :17
-1
0
1
2
3
4
5
6
```

4 - On a maintenant un problème à 3 paramètres : k (réel), a, b (entiers tels que $a < b$).

```
k=eval(input('la valeur de k est :'))
a=eval(input('la valeur de a est :'))# par hypothèse, a est un entier.
b=eval(input('la valeur de b est :'))# par hypothèse, b est un entier.
for n in range(a,b+1):# On cherche les solutions dans [a,b].
    if n**2 - 5*n + 3 < k:
        print(n)
```

L'essai ci-dessous est fait pour `k = -2.2`, `a = -7` et `b = 22`.

```
la valeur de k est :-2.2
la valeur de a est :-7
la valeur de b est :22
2
3
```

Fin de la solution

Remarque et compléments :

Ce qui suit ne convient pas à des débutants en Python.

1 - On pourrait aussi traiter cet exercice à l'aide du signe du trinôme $x^2 - 5x + 3 - k$.

2 - On aurait pu rédiger cette solution à l'aide d'une fonction-Python, voir la définition dans le script ci-dessous aux lignes 6 et 7.

3 - On peut illustrer l'exercice précédent à l'aide d'une représentation graphique :

```
from numpy import *
from matplotlib.pyplot import*
k=eval(input('La valeur de k est :'))#k est un nombre réel.
a=eval(input('La valeur de a est :'))# a est un entier.
b=eval(input('La valeur de b est :'))# b est un entier > a.
def f(x):
    return x**2 - 5*x + 3
abs=linspace(a,b,100)# [a,b] est partagé par 99 points équidistants
# en 100 intervalles égaux.
ord=f(abs)# f est vectorisée.
plot(abs,ord)# Graphe de f entre a et b.
plot([a,b],[k,k])# Graphe de y=k entre a et b.
plot([a,b],[0,0])# Graphe de y=0 entre a et b.
c=arange(a,b+1)# c est la matrice [a, a+1, ..., b].
d=f(c)# f est vectorisée.
masque=(d<k)
y=d[masque]
```

```
x=c[masque]
for i in x:
    plot([i,i],[0,f(i)])# Graphe de x=i entre 0 et f(i).

grid(True)
savefig('sec2fig5.png')# Production d'une image au format png.
show()# On affiche tous les graphes simultanément.
```

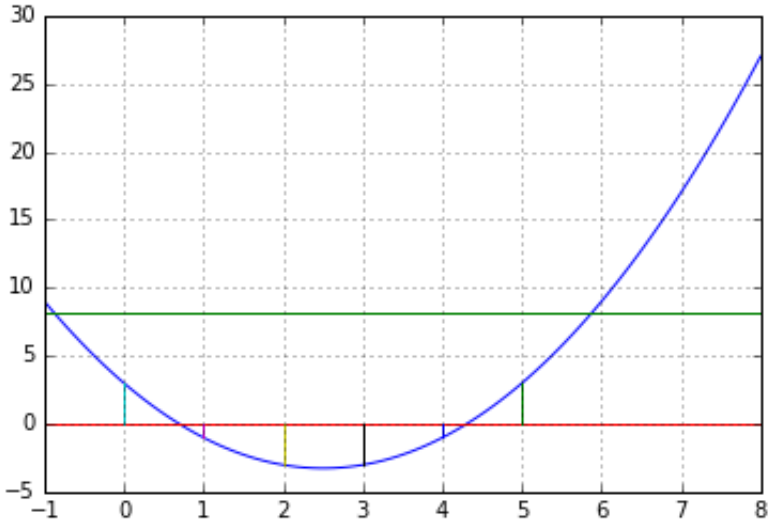
Application :

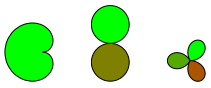
```
la valeur de k est  :8.21

la valeur de a est  :-1

la valeur de b est  :8
```

Voici le graphe obtenu (image png produite par Python) :





2.9 [*] Algorithme d'Euclide.

1 - Programmer une fonction qui, étant donnés deux entiers positifs `a` et `b`, retourne leur PGCD [a] calculé selon l'algorithme d'Euclide.
2 - En déduire le calcul de leur PPCM [b].

a. plus grand diviseur commun
b. plus petit multiple commun

Mots-clefs : reste et quotient de la division euclidienne (`%` et `//`), algorithme d'Euclide, boucle `tant que`, importation de fonctions, module `math`.

Solution [24]

1 - La justification de l'algorithme d'Euclide est bien connue : si `r` désigne le reste de la division euclidienne de `a` par `b` [25], l'ensemble des diviseurs communs de `a` et `b` est le même que l'ensemble des diviseurs communs de `b` et `r`. Comme `r < b`, on tombe sur un problème plus simple si, au lieu de chercher l'ensemble des diviseurs communs de `a, b`, on cherche l'ensemble des diviseurs communs de `b, r`. Il est même possible que `r = 0`. Sinon, on recommence. On est sûr d'arriver à 0. Or 0 étant divisible par tout entier positif, l'ensemble des diviseurs communs de `n)`, entier positif quelconque et 0 est l'ensemble des diviseurs de `n`, le plus grand étant évidemment `n` lui-même. L'algorithme ci-dessous se trouve ainsi justifié.

```
def euclide(a,b):
    while b > 0:
        a,b = b,a%b
    return a
```

Applications [26] :

```
from euclide import *
euclide(177,353)
Out[2]: 1
euclide(123456789,987654321)
Out[3]: 9
```

Les calculs sont pratiquement instantanés : l'algorithme d'Euclide est très efficace. Bien sûr, Python fournit une fonction primitive qui fait la même chose : la fonction `gcd()` [27] :

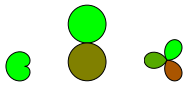
```
from math import gcd
math.gcd(177,353)
Out[5]: 1
math.gcd(123456789,987654321)
Out[6]: 9
```

2 - Le calcul du PPCM de deux entiers positifs se déduit du calcul de leur PGCD parce que le produit du PPCM par le PGCD est égal au produit des deux nombres. En continuant le calcul ci-dessus, on obtient :

24. Le calcul du PGCD de `p` entiers positifs est repris à l'exercice 6.1.
25. Plus généralement, dans l'algorithme d'Euclide, `a` peut être un entier de signe quelconque, `b` un entier de signe quelconque non nul.
26. On remarquera sur le listing qui suit que la fonction `euclide()` a été chargée dans la console en l'appelant exactement comme un module (créé par nous). Autre possibilité : si on vient de la rédiger dans l'interpréteur, on peut aussi la charger en l'exécutant.
27. Le code source de la fonction `gcd()` du module `math`, voir [11] est exactement celui de la fonction `euclide()`.

```
print(177*353//1)
62481
print(123456789*987654321//9)
13548070123626141
```

Les PPCM recherchés sont respectivement égaux à 62481 et 13548070123626141.



Chapitre 3

Listes

Liste des exercices :

Énoncé n° 3.1 [*] : Simuler `n` lancers d'un dé équilibré, sous forme de liste

Énoncé n° 3.2 [*] : Maximum d'une liste de nombres

Énoncé n° 3.3 [*] : Ranger une liste de nombres

Énoncé n° 3.4 [*] : Permutation aléatoire d'une liste.

Ce chapitre contient des listes et des boucles `pour`.

Une liste est une séquence d'éléments rangés dans un certain ordre[1]. Ces éléments peuvent être de types différents[2]. Nous ne considèrerons dans ce chapitre que des listes de nombres. On dispose, pour manipuler des listes, d'un certain nombre d'instruments appelés `méthodes`[3].

Si l'on veut tester un algorithme qui calcule, par exemple, le maximum d'une liste de nombres, on a besoin de longues listes de nombres. On les engendre habituellement avec des générateurs de nombres aléatoires[4]. Dans ce chapitre, elles seront systématiquement engendrées par les deux fonctions `reels()` et `entiers()` suivantes :

Fonction `reels()`

```
from random import gauss
from copy import deepcopy
def reels(n):
    L=[]# Liste vide.
    for i in range(n):
        L.append(gauss(7,10))# gauss() est une fonction du module random.
    Lo=deepcopy(L)# deepcopy()  est une fonction du module copy.
    return[L,Lo]
# reels() retourne une liste composée de 2 listes.
# Lo est une copie indépendante de L. Si on modifie L, Lo ne change pas.
```

Fonction `entiers()`

```
from random import randint
from copy import deepcopy
```

1. Voir [1], pp. 20-27.
2. On en déduit que les listes ne sont pas faites pour les calculs.
3. Voir [1], p. 27 et les exercices qui suivent.
4. On n'a pas besoin de savoir comment ces listes sont fabriquées.

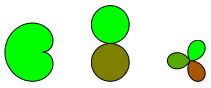
```
def entiers(n):
    L = []# Liste vide.
    for i in range(n):
        L.append(randint(-50,100))
    Lo = deepcopy(L)
    return [L,Lo]
# entiers() retourne une liste composée de 2 listes.
# Lo est une copie indépendante de L. Si on modifie L, Lo ne change pas.
```

qui retournent une liste [L,L0] formée d'une liste `L` de `n` nombres réels ou entiers suivant le cas ainsi qu'une copie `Lo` de `L` qui est souvent utile parce que si `L` est modifiée au cours de calculs ultérieurs, `Lo` conservera l'état initial de `L`.

Exemple : Engendrons une liste de 10 entiers et une copie indépendante de cette liste.

```
import entiers as en
A = en.entiers(10)# ou from entiers import * auquel cas, A = entiers(10).
A
Out[3]:
[[88, 53, 88, -38, 23, 16, -11, 96, 9, 87],
 [88, 53, 88, -38, 23, 16, -11, 96, 9, 87]]
type(A)
Out[4]: list
type(A[0])
Out[5]: list
```

Engendrer une liste de 1 000 000 de réels avec la fonction `reels()` prend à peu près 1.5 secondes.



3.1 [*] Simuler n lancers d'un dé équilibré, sous forme de liste

En utilisant la fonction `randint()` du module `random`, simuler, sous forme de liste, `n` lancers d'un dé équilibré.

Mots-clefs : liste, module `random`, fonction `randint()`, méthode `append()` de l'objet liste, boucle `pour`.

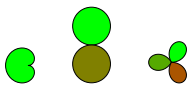
Solution

`randint(a,b)` où `a` et `b`, $a \leqslant b$ sont des entiers, retourne un nombre entier tiré au hasard dans l'intervalle `[a,b]`, extrémités comprises. Par exemple, `randint(1,6)` retourne un nombre tiré au hasard dans l'ensemble $\{1, \ldots, 6\}$ et par conséquent `randint(1,6)` simule le lancer d'un dé. Cela fournit le premier terme de la liste[5] recherchée. À l'aide d'une boucle `pour`, on ajoutera à cette liste les tirages suivants comme dans le script ci-dessous, à l'aide de la commande `append()`[6],[7]. La fonction `lancersDe` est une solution du problème posé.

```
from random import randint
def lancersDe(n):
    L=[]# Liste vide.
    for i in range(n):# Pour engendrer une liste de n termes.
        L.append(randint(1,6))# randint est une fonction du module random.
    return(L)
# lancerDe(n) retourne une simulation d'une suite de n lancers d'un dé.
```

Exemple : simuler 15 lancers successifs d'un dé.

```
runfile('Chemin de la fonction lancersDe')
lancersDe(15)
Out[2]: [4, 4, 6, 4, 2, 6, 6, 3, 1, 5, 4, 4, 2, 1, 5]
```



5. Sur la notion de `liste` dans Python, voir [1], chapitre 1, section 9.
6. `append()` est une méthode de l'objet liste. Les méthodes usuelles des listes sont décrites dans [1], déjà cité.
7. Par exemple, directement dans la console :

```
L = [1,2,3,4]
L.append(5)
L
Out[11] : [1, 2, 3, 4, 5]
```

3.2 [*] Maximum d'une liste de nombres

1 - Calculer le maximum `M` d'une liste `L` de nombres, de longueur `l` comme suit :

(a) On pose `M=L[0]`.
(b) Ensuite, si `l` $\geqslant 2$, `i` prenant successivement les valeurs 1, ..., `l-1`, on pose `M=L[i]` si `L[i] > M`.
On présentera la solution sous la forme d'une fonction.

2 - Application : calculer le maximum d'une liste de 10, puis de 10 000 nombres réels engendrée par la fonction `reels()`, cf.(3).

3 - Calculer le minimum `m` de `L` en utilisant le calcul du maximum programmé ci-dessus.

Mots-clefs : boucle `pour`, instruction conditionnelle `if`, appeler une fonction nouvellement programmée, méthode `list.append()` des listes, commande `len()` [8], primitives `min()` et `max()`.

Solution

1 - Nous proposons la fonction `atlas()` suivante, conforme à l'énoncé :

```
def atlas(L):# L est une liste de nombres (au moins 1).
    for i in range(len(L)):# len(L) : longueur de L (notée l dans l'énoncé).
        if L[i]>L[0]:
            L[0] = L[i]
    return L[0]
```

Remarquons que lors d'une exécution d'`atlas()`, seul le premier terme de `L` est changé et ceci seulement dans le cas où il n'en est pas le maximum. La fonction `atlas()` retourne un nombre de type `float` [9], qui est le type des nombres produits par la fonction `reels()`.

2 - On aura besoin de charger dans la console les fonctions `reels()` et `atlas()`, ce qui se fait via les commandes suivantes :

```
from atlas import *
from reels import *
```

Appliquons-les pour calculer le maximum d'une liste de 10 réels fournie par la fonction `reels()` :

```
from reels import *
from atlas import *
[L,L0] = reels(10)
L
Out[4]:
[10.531352495090177,
 12.825675783103737,
 10.077230201543921,
 6.782970458445565,
 1.9166906476453551,
 15.849889158355168,
 -0.7293027684673463,
 11.407060614760322,
 -3.7720041187274074,
```

8. `len(L)` retourne la longueur de la liste `L`.
9. nombre réel à virgule flottante

```
 6.436637330311071]
L0
Out[5] :
[10.531352495090177,
 12.825675783103737,
 10.077230201543921,
 6.782970458445565,
 1.9166906476453551,
 15.849889158355168,
 -0.7293027684673463,
 11.407060614760322,
 -3.7720041187274074,
 6.436637330311071]
atlas(L)
Out[6] : 15.849889158355168
L
Out[7] :
[15.849889158355168,
 12.825675783103737,
 10.077230201543921,
 6.782970458445565,
 1.9166906476453551,
 15.849889158355168,
 -0.7293027684673463,
 11.407060614760322,
 -3.7720041187274074,
 6.436637330311071]
```

On constate bien qu'au cours de l'exécution de la commande `atlas(L)`, seule la première valeur de la liste `L` a changé. Calculons de même le maximum de 10 000 nombres :

```
[L,L0] = reels(10000)
atlas(L)
Out[9] : 47.553319964137636
```

Les calculs sont à peu près instantanés. Il est facile de vérifier ces résultats [10] qui sont donnés directement par la primitive `max()` [11] :

```
max(L0)
Out[9] : 47.553319964137636
```

3.1 - Normalement, on calcule le minimum de `L` à l'aide de la primitive `min()` et de l'instruction `min(L)`.
3.2 - Une autre solution serait de programmer une fonction analogue de `atlas()` qui donnerait le minimum au lieu de donner le maximum (il suffit de remplacer `>` par `<` dans le script définissant `atlas()`).
3.3 - On peut déduire le calcul du minimum d'une liste de nombres du calcul du maximum. En effet, le minimum de `n` nombres est l'opposé du maximum des opposés de ces nombres. C'est compliqué parce que `L` étant une liste de nombres, `- L` n'a pas de sens pour Python. Il faut changer les signes des éléments de `L` un par un à l'aide d'une boucle `pour`. Dans le script ci-dessous, la liste obtenue est notée `Lm` :

```
def hera(L) :
    Lm = []
```

10. qui n'ont pas besoin d'être vérifiés
11. Elle est disponible sans qu'il soit besoin de charger un module.

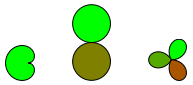
```
    for i in range(len(L)):
        Lm.append(-L[i])
    return Lm
```

Remarquons que dans le script de la fonction `hera`, on peut remplacer la commande `Lm.append(-L[i])` par la commande `Lm = Lm + [-L[i]]` [12] ou par la commande `Lm += [-L[i]]`.

Application : calculons le minimum d'une liste `L` produite par la commande `[L,L0] = reels(100000)` [13] :

```
from hera import *
[L,L0] = reels(100 000)
Lm = hera(L)
-atlas(Lm)
Out[13]: -35.7044541020379
min(L)
Out[14]: -35.7044541020379
```

Cette solution est. évidemment mauvaise.



12. Concaténation de listes.
13. Les fonctions `atlas()` et `reels()` ont déjà été importées.

3.3 [*] Ranger une liste de nombres

Une liste de nombres `L` est donnée.
1 - À l'aide de la primitive `min()`, ranger cette liste dans l'ordre croissant (le premier terme `m` de la liste à calculer sera le minimum de `L`, le suivant sera le minimum de la liste `L` dont on aura retiré `m`, *etc*).
2 - Ranger `L` dans l'ordre décroissant.

Mots-clefs : primitives [14] `min()` et `max()`, boucle `pour`, méthodes `list.sort()` et `list.reverse()`, commandes `L[: p]` et `L[p :]` [15], fonction `len()` (pour les listes), primitive `sorted()`.

Solution [16]

Pour ranger `L` dans l'ordre croissant puis dans l'ordre décroissant, on utilise normalement les méthodes `list.sort()` [17], [18] et `list.reverse()` [19].

1 - L'énoncé suggère le procédé suivant :

- le premier terme de la future liste dans l'ordre croissant sera le minimum `m` de `L`,
- le second sera le minimum de la liste obtenue en enlevant `m` de `L` (une seule fois si `m` y figure plusieurs fois)
- et ainsi de suite.

Cela se traduit par la fonction `aphrodite()` suivante qui retourne toute liste de nombres donnée après l'avoir rangée dans l'ordre croissant :

```
def aphrodite(L):
    Lcroissante = []# [] : liste vide
    for i in range(len(L)):# i vaut 0 puis 1 ... et enfin len(L)-1
        m = min(L)
        Lcroissante.append(m)# Ajoute m à la fin de la liste Lcroissante.
        L.remove(m)# Retire la première occurence de m dans L.
    return(Lcroissante)
```

`L` perd un élément à chaque passage de la boucle. À la fin, `L` est vide. La fonction `aphrodite()` a été écrite dans l'éditeur de texte de l'environnement de programmation utilisé (`Spyder`). Pour ranger dans l'ordre croissant la liste `L= [-2,4.17,8.79,-0.57,0.19,-4.23]`, exécutons `aphrodite()`, qui est ainsi chargée dans la console, ce qui se traduit par :

```
runfile('Chemin de la fonction aphrodite')
```

puis tapons :

```
L = [-2,4.17,8.79,-0.57,0.19,-4.23]
aphrodite(L)
Out[3]: [-4.23, -2, -0.57, 0.19, 4.17, 8.79]
L
Out[4]: []
```

14. cf. [1], section 4, p. 37 : une primitive désigne une fonction de base fournie par Python.
15. listes des `p` premiers termes et des `p` derniers termes de la liste `L`
16. Ceci est un exercice d'apprentissage. Il existe de nombreuses méthodes de tri.
17. remplace la liste par la même, ordonnée dans l'ordre croissant
18. ou la primitive `sorted()`
19. remplace la liste par la même, ordonnée dans l'ordre décroissant

On a vérifié que la valeur finale de `L` est `[]` et on a trouvé `Lcroissante=[-4.23, -2, -0.57, 0.19, 4.17, 8.79]`.
On a déjà dit qu'en temps normal, on aurait utilisé la méthode `list.sort()` ou la primitive `sorted()` [20].
La valeur originale de `L` ayant été récupérée par un copier-coller, cela donne :

```
L = [-2,4.17,8.79,-0.57,0.19,-4.23]
L.sort()
L
Out[9]: [-4.23, -2, -0.57, 0.19, 4.17, 8.79]
L = [-2,4.17,8.79,-0.57,0.19,-4.23]
sorted(L)
Out[11]: [-4.23, -2, -0.57, 0.19, 4.17, 8.79]
```

2 - Il suffit de modifier le script qui définit la fonction `aphrodite()` en remplaçant `min` par `max`.

Remarque importante :

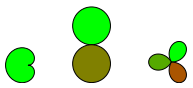
La liste `L` ci-dessus étant très courte et sans intérêt, remplaçons-la par une liste de 10 000 réels engendrée par la commande `reels(10000)` et ordonnons-la à l'aide de la fonction `aphrodite()` puis de la fonction `sorted()` :

```
from reels import *
[L,Lo] = reels(10000)# Lo est une copie indépendante de L.
from aphrodite import *
Lcroissante = aphrodite(L)# Suite L rangée dans l'ordre croissant
# à l'aide de la fonction aphrodite.
Lcroissante[:4]# 4 plus petits termes de L.
Out[6]:
[-29.41317007426497,
 -28.747176136743192,
 -26.928746199250646,
 -25.74374951009743]
Lcroissante[9996:]# 4 plus grands termes de L.
Out[7]:
[40.68458999805904,
 41.64978402811625,
 41.821776781909804,
 43.184709110328164]

Lcroissante2 = sorted(Lo)# Suite L rangée dans l'ordre croissant
# à l'aide de la fonction sorted().
Lcroissante2[:4]# Vérification : 4 plus petits termes de L.
Out[9]:
[-29.41317007426497,
 -28.747176136743192,
 -26.928746199250646,
 -25.74374951009743]
Lcroissante2[9996:]# Vérification : 4 plus grands termes de L.
Out[10]:
[40.68458999805904,
 41.64978402811625,
 41.821776781909804,
 43.184709110328164]
```

20. Voir [1], chapitre 1, 9. Les listes, p.27.

La primitive `sorted()` est beaucoup plus rapide que la fonction `aphrodite()` à tel point que si on veut ranger une liste d'un million de nombres, par exemple, on sera très tenté d'interrompre le premier calcul tellement c'est long alors que le second dure à peu près une seconde.



3.4 [*] Permutation aléatoire d'une liste.

Programmer une fonction qui retourne sous forme de liste une permutation aléatoire de la liste `L = [1, 2, …,n]`.
Méthode imposée : on commencera par tirer au hasard un élément de `L` qui deviendra le premier élément de la permutation aléatoire recherchée et on l'effacera de `L`. Puis on répètera cette opération jusqu'à ce que `L` soit vide.

Mots-clefs : boucle `pour`, fonction `randint()` du module `random`[21], primitive `len()`, L.append(x)[22], fonction `del()`, `list(range())`.

Solution : Le script est imposé par l'énoncé. Il suffit de connaître les commandes suivantes :

- `randint(a,b)`, `a` et `b` entiers tels que $a \leqslant b$, retourne un entier tiré au hasard dans l'intervalle $[a, b]$.
- `len(L)` retourne la longueur de la liste `L`,
- L.append(x) modifie la liste `L` en lui ajoutant (à la fin) l'élément `x`.
- `list(range(n))` retourne `[0, 1, ..., n-1]`.

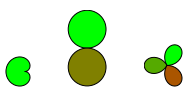
Cela qui permet de définir une fonction `alealiste` solution du problème comme suit :

```
from random import randint
def alealiste(L):# L est une liste.
    alea = []
    n = len(L)
    for i in range(n):
        j = randint(0,n-1-i)
        alea.append(L[j])
        del(L[j])
    return alea
```

Exemple : Produire une permutation aléatoire de la liste `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` à l'aide de la fonction `alealiste()`.

```
L = list(range(11))
L
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
from alealiste import *
alealiste(L)
Out[4]: [4, 2, 1, 7, 0, 9, 8, 5, 3, 10, 6]
```

Remarque importante : L'énoncé manque de précision puisque l'expression *permutation aléatoire* est très ambigü. Le lecteur un peu au fait du Calcul des probabilités comprendra aisément qu'en modélisant comme il faut ce problème, on pourrait démontrer que toutes les permutations possibles de la liste `list(range(len(L))` sont équiprobables.



21. Le module `random` doit être distingué du module `numpy.random`.
22. `append()` est une méthode de l'objet liste.

Chapitre 4

Matrices (m,)

Liste des exercices :

Énoncé n° 4.1 [**] : Le lièvre et la tortue, jeu équitable ?

Énoncé n° 4.2 [**] : Maximum et minimum d'une matrice-ligne.

Énoncé n° 4.3 [***] : Permutations aléatoires d'une matrice à une dimension.

Les matrices relèvent du module `numpy` de Python qu'il faut donc charger dans la console lorsqu'on en utilise. Malheureusement, `numpy` distingue deux sortes de matrices, ce que l'on ne fait pas en mathématiques [1] et qui perturbera, au début, les mathématiciens :

• les matrices `(m, )` ou matrices à une dimension [2] ou vecteurs [3], par exemple

```
import numpy as np
A = np.array([1,2,3])
A
Out[3]: array([1, 2, 3])
type(A)
Out[4]: numpy.ndarray
A.dtype
Out[5]: dtype('int64')# Les éléments de A sont des entiers codés en 64 bits.
A.shape# ou shape(A)
Out[6]: (3, )# A est bien une matrice (m, ).
```

• les matrices multidimensionnelles (on se limitera aux matrices 2-dimensionnelles) [4]. Ainsi dans la console, à la suite de l'exemple précédent :

```
B = np.array([[1.1,2.2,3.3],[4.4,5,6]])
B
Out[8]:
array([[ 1.1,  2.2,  3.3],
       [ 4.4,  5. ,  6. ]])
type(B)
Out[9]: numpy.ndarray
```

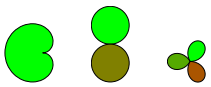
1. En mathématiques, au niveau élémentaire, les matrices sont de simples tableaux rectangulaires de nombres qui ont donc «deux dimensions» : le nombre des lignes `n` et le nombre des colonnes `m` résumées en `(n,m)`.
2. sic
3. terme ambigü
4. Ce sont les vraies matrices, les matrices des mathématiciens. Elles servent principalement à représenter des applications linéaires, ce qui explique la curieuse définition du produit matriciel, qui correspond à la composition des applications.

```
B.dtype
Out[10]: dtype('float64')
B.shape
Out[11]: (2, 3)
```

On peut convertir une matrice `(m, )` en une matrice `(n,m)` comme suit :

```
C = A[np.newaxis]
C
Out[13]: array([[1, 2, 3]])# À comparer avec la sortie [3].
type(C)
Out[14]: numpy.ndarray
C.shape
Out[15]: (1, 3)# À comparer avec la sortie [6].
```

Ce chapitre comprend seulement des exercices sur les matrices (m,).



4.1 [**] Le lièvre et la tortue, jeu équitable ?

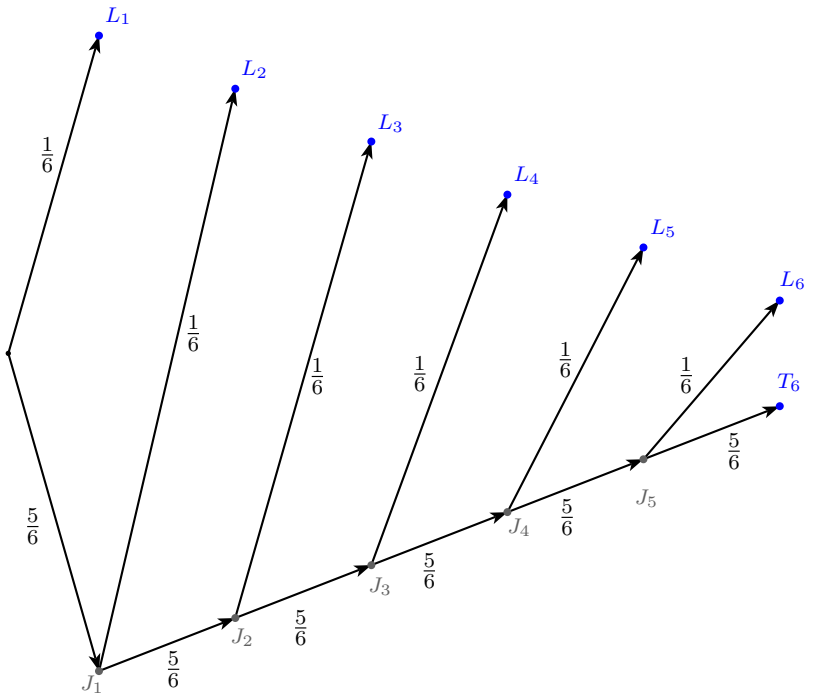
On lance un dé équilibré. Si le 6 sort, le lièvre gagne. Sinon la tortue avance d'une case et on rejoue. La tortue gagne si elle parvient à avancer de `n` cases (`n` $\geqslant 1$ donné).

1 - Modéliser et simuler ce jeu à l'aide de listes Python.
2 - Calculer la probabilité pour que le lièvre gagne, que la tortue gagne.
3 - Existe-t-il une valeur de `n` pour laquelle le jeu est équitable ?

Mots-clefs : liste, modélisation, mathématiques, probabilités, simulation.

Cet exercice très connu a peu d'intérêt calculatoire. C'est avant tout un exercice de modélisation[5], très simple s'il est bien modélisé. Les bonnes questions pour commencer sont : y a-t-il toujours un vainqueur ? combien faut-il de lancers ?
Il est évident que le jeu sera fini au $\texttt{n}^{\text{ième}}$ lancer parce qu'un 6 sera sorti, auquel cas le lièvre aura gagné ; sinon, la tortue gagnera au $\texttt{n}^{\text{ième}}$ lancer car elle aura avancé de `n` cases.

1 - Solution n°1 : Il n'est pas nécessaire dans tous les cas de lancer `n` fois le dé. Par exemple, si 6 sort au premier lancer et si `n>1`, les lancers suivants ne servent à rien car le lièvre a gagné au premier coup. Le nombre de lancers nécessaires pour que l'une des deux bestioles gagne est aléatoire. Cela conduit à un graphe[6] avec des branches de longueurs différentes. Par exemple[7], si `n=6`,
- J_i, $i = 1, \ldots, 5$ désigne l'événement : « Après le $i^{\text{ème}}$ lancer, le jeu continue »,
- L_i, $i = 1, \ldots, 6$ l'événement « Le lièvre gagne au $i^{\text{ème}}$ lancer »,
- T_6 l'événement « La tortue gagne (nécessairement au $6^{\text{ème}}$ lancer) ».



Cette modélisation sous forme de graphe est assez jolie et pratique. Dans ce cadre, nous simulerons le jeu par la fonction `lafontaine()` définie ci-dessous. On observera qu'elle traduit exactement l'énoncé : si un 6

5. On ne peut chercher de solution que dans le cadre d'une théorie mathématique ! Sinon, comment pourrait-on justifier nos calculs ?
6. qu'il est inutile de tracer
7. pour le plaisir

apparaît, elle retourne « Le lièvre a gagné », les calculs s'arrêtent. Sinon, les calculs continuent et s'ils vont jusqu'au bout, elle retourne « La tortue a gagné » [8].

```
from random import *
def lafontaine(n):
    for i in range(n):#Il y aura au plus n lancers du dé.
        if randint(1,6) == 6:
            return 'Le lièvre a gagné'
    return 'La tortue a gagné'
```

Exemple :

```
lafontaine(6)
Out[10]: 'Le lièvre a gagné'
lafontaine(3)
Out[11]: 'La tortue a gagné'
```

1 - Solution n°2 : Dans cette solution et dans la suivante, on imagine que l'on effectue les `n` lancers, même si un `6` sort avant le dernier lancer [9], [10]. On peut représenter ces `n` lancers à l'aide de `n` variables aléatoires indépendantes X_1, ..., X_n ne prenant chacune que les valeurs 1, 2, 3, 4, 5, 6 avec la probabilité $\frac{1}{6}$.

Simuler le jeu, c'est donc d'abord simuler `n` lancers successifs d'un dé, ce que l'on peut faire avec la fonction `lancersDe` de l'exercice 3.1, qui retournera la liste des chiffres sortis ; c'est ensuite interpréter cette liste : l'événement «L'une de ces variables a pris la valeur 6» modélise la victoire du lièvre, son complémentaire la victoire de la tortue. On obtient ainsi la fonction notée `esopeliste()`, très simple :

```
from lancersDe import *
def esopeliste(n):
    if max(lancersDe(6)) == 6:
        return 'Le lièvre a gagné'
    else:
        return 'La tortue a gagné'
```

Exemple : Importons la fonction `esopeliste()` dans la console puis exécutons-la 5 fois, dans le cas `n=6` :

```
from esopeliste import *
esopeliste(6)
Out[2]: 'Le lièvre a gagné'
esopeliste(6)
Out[3]: 'La tortue a gagné'
esopeliste(6)
Out[4]: 'La tortue a gagné'
esopeliste(6)
Out[5]: 'Le lièvre a gagné'
esopeliste(6)
Out[6]: 'Le lièvre a gagné'
```

1 - Solution n°3 : C'est la même que la solution n°2 sauf que l'on simule les `n` lancers du dé à l'aide d'une matrice-ligne retournée par la fonction `randint()` [11] du module `numpy.random`. au lieu d'une liste fournie

8. Dans ce script, on voit aussi comment utiliser la commande `return` pour arrêter une boucle.
9. C'est une modélisation différente de la précédente.
10. Si on veut représenter ce jeu par un graphe, chaque branche listant les résultats d'une suite de `n` lancers du dé, ce graphe aura 6^n branches de longueur `n`. Par exemple, si `n=6`, il aura $6^6 = 46\,656$ branches, ce qui est beaucoup trop. Heureusement, tracer un tel graphe est inutile.
11. commande `random.randint(1,7,n)` ou simplement `randint(1,7,n)`

par la fonction `entiersDe()`. C'est donc encore plus simple. La fonction simulatrice correspondante a été appelée `esopeligne()`.

```
from numpy.random import *
def esopeligne(n):
    if max(randint(1,7,n)) == 6:
        return 'Le lièvre a gagné'
    else:
        return 'La tortue a gagné'
```

Par exemple, simulons 4 fois le jeu du lièvre et de la tortue dans le cas `n=6`, directement dans la console :

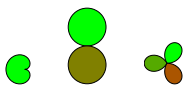
```
from esopeligne import *
esopeligne(6)
Out[2]: 'Le lièvre a gagné'
esopeligne(6)
Out[3]: 'Le lièvre a gagné'
esopeligne(6)
Out[4]: 'La tortue a gagné'
esopeligne(6)
Out[5]: 'Le lièvre a gagné'
```

2 - Solution pour la première modélisation : On peut imaginer le graphe pour `n` quelconque. Il y a une seule branche à `n` rameaux, chacun de probabilité $\frac{5}{6}$, qui mène à la victoire de la tortue. La probabilité pour que la tortue gagne est donc $\left(\frac{5}{6}\right)^n$. Comme l'événement "Le lièvre gagne" et l'événement "La tortue gagne" sont complémentaires, le lièvre gagne donc avec la probabilité $1-\left(\frac{5}{6}\right)^n$.

2 - Solution pour la deuxième modélisation : La probabilité pour que la tortue gagne est $\left(\frac{5}{6}\right)^n$ puisque la probabilité pour que chacune des variables aléatoires ne prennent pas la valeur 6 est $\frac{5}{6}$ et que ces variables aléatoires sont indépendantes. On obtient donc les mêmes résultats dans les deux modélisations. On n'avait d'avance aucun doute à ce sujet [12].

3 - La question posée signifie : "Y a-t-il une valeur de `n` telle que $\left(\frac{5}{6}\right)^n = \frac{1}{2}$ ou $5^n = 2^{n-1}\times 3^n$. Si une telle valeur de `n` existait, on en déduirait que 3 divise 5, ce qui est faux [13]. On peut aussi résoudre l'équation $\left(\frac{5}{6}\right)^x = \frac{1}{2}$, $x\in[1,+\infty[$ dont l'unique solution, qui n'est pas un nombre entier, est $x=\dfrac{\log 2}{\log 6-\log 5}$, que l'on peut calculer avec Python :

```
from math import *
x=(log(2))/(log(6)-log(5))
x
Out[3]: 3.801784016923929
```



12. Dans des problèmes compliqués, on peut imaginer que des modélisations différentes conduisent à des résultats différents, auquel cas se pose le problème de la validation des modèles. Dans les cas simples, ce problème ne se pose jamais.

13. ou qu'un même entier $\geqslant 1$ peut avoir deux décompositions comme produit de nombres premiers, ce qui est faux aussi.

4.2 [**] Maximum et minimum d'une matrice-ligne

1 - Calculer le maximum d'une matrice-ligne de nombres `M` en procédant comme suit :
(a) - on choisit arbitrairement l'un de ces nombres, noté `ma` [a] et on efface les nombres $\leqslant$ `ma`.
(b) - s'il n'en reste plus, `ma` est le maximum recherché.
(c) - sinon, on recommence (a) et (b).
2 - Calculer le minimum `mi` de `M`.

a. Ci-dessous, `ma` est le premier terme de `M`.

Mots-clefs : matrices-lignes, commande `len()`, masque, maximum et minimum, fonctions primitives `max()` et `min()`, boucle `tant que`, importation de fonctions, module `time`, fonction `clock()`, module `copy`, fonction `deepcopy()`.

Solution

S'agissant de matrices, on aura besoin d'importer le module `numpy` ; on utilisera un masque de matrice [14], [15], ce qui fait l'intérêt de cet exercice au demeurant très court.

1 - On peut utiliser le script commenté suivant qui définit une fonction notée `maxime()` qui, comme la primitive `max()` [16], retourne le maximum d'une matrice-ligne de nombres donnée.

```
# La fonction maxime utilise le masque M > ma.
from numpy import *

def maxime(M) : # M est une matrice-ligne de nombres.
    while len(M)>=1 :# Tant que M n'est pas vide.
        ma = M[0]# Le choix de ma dans M est arbitraire.
        M = M[M>ma]# On ne conserve que les éléments de M > ma.
        # La longueur de M diminue à chaque passage de la boucle.
    return(ma)
```

La commande `M = M[M>ma]` ne retient de `M` que les nombres `> ma`, autrement dit efface les autres.

Exemple : Importons la fonction `maxime()` dans la console [17], [18], puis cherchons le maximum de la matrice-ligne `M` retourné par la commande `M = randn(1000000)` [19] :

```
from maxime import *
from numpy.random import *
M = randn(1000000)
maxime(M)
Out[7] : 4.8898710392903109
```

Le calcul a été pratiquement instantané.

2 - Pour calculer le minimum d'une matrice `Mat`, on voudrait utiliser l'égalité
`min(Mat) = - max(-Mat)`, autrement dit, `min(Mat) = - maxime(-Mat)`.

14. cf. [7], p. 17-18.
15. On verra que les masques de matrices sont très utiles.
16. `max()` fait mieux. En effet, `max()s` retourne le maximum d'une matrice-ligne ou d'une liste de nombres alors que `maxime()` ne s'applique pas aux listes.
17. On la charge comme si c'était un module.
18. ce qui provoque aussi le chargement de `numpy`
19. `randn` est une fonction du module `numpy.random` qu'il faut aussi charger dans la console. Il est inutile de savoir que `M` est un échantillon de la loi normale centrée réduite de taille 1 000 000.

vraie pour n'importe quelle matrice `Mat`. Malheureusement, la valeur initiale de `M` a été perdue dans le calcul de `maxime(M)`. On aurait dû conserver cette valeur initiale la ré-utiliser dans le calcul du minimum. Cela se fait avec la fonction `deepcopy()` du module `copy`[20].

Exemple : Engendrons une matrice-ligne de 1 000 000 de nombres flottants comme ci-dessus, puis nous calculons son maximum et son minimum ; enfin, faisons apparaître ces valeurs à l'aide de `print()`.

```
from maxime import *
from copy import deepcopy
from numpy.random import randn

M = randn(1000000)
Mbis = deepcopy(M)
ma = maxime(M)
mi = -maxime(Mbis)
print(ma,mi)
4.59811940548  -4.59811940548
```

Fin de l'exercice

Compléments : comparaison des vitesses de `maxime()` et de `max()`.

Pour comparer les vitesses de `maxime()` et de la primitive `max()`, considérons le script suivant, qui donne les durées du calcul du maximum de la matrice-ligne `M` ci-dessous par `maxime()`, puis par `max()`. Pour fixer les idées, nous lançons une nouvelle session de calcul en tapant dans l'interpréteur :

```
from time import clock # Pour compter les durées de calcul.
from copy import deepcopy # Pour fabriquer une copie indépendante de M.
from maxime import * # Importation de la fonction maxime.
# Cette importation comprend le module numpy.

M=random.randn(1000000)
Mbis=deepcopy(M)
d=clock()
print('Le maximum de M par maxime est',maxime(M))
f=clock()
print('Le temps de calcul par maxime est',f-d)
d=clock()
print('Le maximum de Mbis par max est',max(Mbis))# Autre calcul du maximum.
f=clock()
print('Le temps de calcul par max est',f-d)
```

puis nous exécutons ce script :

```
runfile('Chemin du script')
Le maximum de M par maxime est 4.69931038606
Le temps de calcul par maxime est 0.025789999999999758
Le maximum de Mbis par max est 4.69931038606
Le temps de calcul par max est 0.07035400000000003
```

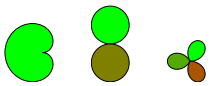
La comparaison des durées de calcul est surprenante car la fonction `maxime()` paraît plus rapide que la fonction primitive de Python `max()`.

20. Voir [1], p. 23-25

Faisons un deuxième essai. `M` est maintenant un échantillon de taille 80000 de la loi uniforme sur l'intervalle d'entiers [-100000, 250000][21], [22]. On obtient :

```
runfile('Chemin du script modifié')
Le maximum de M par maxime est 249999
La durée du calcul par maxime est 0.001866999999999841
Le maximum de M par max est 249999
La durée du calcul par max est 0.00849499999999992
```

Cela semble confirmer la remarque précédente sur la vitesse de `maxime()` et de `max()`.



21. ce que l'on peut ignorer ; sur la fonction `randint` du module `numpy.random`, voir [1], p. 163.
22. Il suffit, dans le script précédent, de remplacer la commande `M=random.randn(1000000)` par la commande `M=randint(-100000,250001,size = 80 000)`.

4.3 [***] Permutations aléatoires d'une matrice à une dimension.

1 - Programmer une fonction qui retourne une permutation aléatoire de la matrice à une dimension `V = array([1, 2, …,n])` sous forme de matrice à une dimension.
Méthode imposée : On commencera par tirer au hasard un élément de `V` qui deviendra le premier élément de la permutation aléatoire `W` recherchée et on l'effacera de `V`. Puis on répètera cette opération tant que `V` n'est pas vide.
2 - Utiliser la fonction précédente pour fabriquer des permutations aléatoires de n'importe quelle matrice à une dimension `M`.

Mots-clefs : fonctions `array()`, `arange()`, `hstack()`, `delete()`, `len()`, `shuffle()` du module `numpy`, `randint()`, `rand()` du module `numpy.random()`, échange d'éléments et extraction d'une sous-matrice d'une matrice à une dimension, `size`, méthode des matrices à une dimension, conversion d'une matrice à une dimension en une liste.

Solution

1 - Les deux fonctions `orphise()` et `orphise1()` ci-dessous, solutions de la première question, ne se distinguent que par la façon d'effacer de `V` les éléments tirés au hasard successifs.

Solution n°1 :

```
import numpy as np
def orphise(n):# n est un entier >=1.
    W=np.array([],int)# W est une matrice d'entiers (m, ) vide.
    V=np.arange(n)# V est une matrice (m, ).
    for i in range(n):
        j=np.random.randint(0,len(V))# j est choisi au hasard,
        # équiprobablement, dans l'ensemble 0, ..., n-i-1.
        W=np.hstack((W,np.array([V[j]])))# On ajoute V[j] à la fin de W.
        V=np.delete(V,[j])# On efface V[j] de V.
    return W # W est une matrice (m, ) obtenue par une permutation aléatoire
    # de 0, 1, ..., n-1.
```

Exemple :

```
from orphise import *
orphise(15)
Out[2]: array([ 7, 12,  8,  4,  5, 15,  1, 13, 11,  3,  2,  0, 14, 16,  6])
```

Solution n°2 :

```
import numpy as np
def orphise1(n):# n entier >=1.
    W=np.array([],int)# W est une matrice d'entiers (m, ) vide.
    V=np.arange(n)# V est une matrice (m, ).
    for i in range(n):
        j=np.random.randint(0,len(V))# j est choisi au hasard,
        # équiprobablement, dans l'ensemble 0, ..., n-i-1.
        W=np.hstack((W,np.array([V[j]])))# On ajoute V[j] à la fin de W.
        V=np.hstack((V[0:j],V[j+1:len(V)]))# On efface V[j] de V.
    return W # W est une matrice (m, ) obtenue par une permutation aléatoire
    # de 0, 1, ..., n-1.
```

Exemple :

```
from orphise1 import *
orphise1(15)
Out[4]: array([ 0,  3,  6,  7, 15, 10,  1,  5, 16, 12, 13, 11,  2,  9,  8])
```

2 - On permute aléatoirement les éléments d'une matrice en permutant aléatoirement les indices de ces éléments. Cela donne la fonction `orphise2()` suivante qui, étant donné une matrice `M` à une dimension retourne une matrice à une dimension qui est une permutation de `M`.

```
from orphise import *
def orphise2(M):# M est une matrice (m, ).
    m = M.size
    W = orphise(m)# W est une matrice (m, ).
    L = list(W)# Conversion de type, de "matrice à une dimension" vers "liste".
    Ma = M[L]
    return Ma
```

Exemple :

```
from orphise2 import *
M = numpy.random.rand(25, )# M est une matrice à une dimension à 25 éléments.
M# Pour voir.
Out[3]:
array([ 3.59103529e-03,  8.36588940e-01,  9.31836143e-01,
        5.97425051e-01,  3.98459887e-01,  2.46173892e-01,
        9.09341745e-01,  5.15681264e-01,  3.44637311e-01,
        1.21643146e-04,  4.67647036e-01,  4.90701074e-01,
        9.47681157e-01,  1.87804186e-01,  6.02391582e-01,
        1.06736664e-01,  8.00597715e-01,  2.76038296e-01,
        3.31099944e-02,  8.22724748e-01,  9.89534567e-01,
        6.37337017e-01,  4.26496185e-01,  4.95840654e-01,
        2.31432883e-01])
orphise2(M)
Out[4]:
array([ 3.59103529e-03,  5.97425051e-01,  5.15681264e-01,
        2.46173892e-01,  4.90701074e-01,  4.26496185e-01,
        9.47681157e-01,  1.06736664e-01,  3.44637311e-01,
        6.02391582e-01,  9.89534567e-01,  2.76038296e-01,
        8.36588940e-01,  8.22724748e-01,  2.31432883e-01,
        9.09341745e-01,  4.67647036e-01,  8.00597715e-01,
        1.87804186e-01,  3.98459887e-01,  6.37337017e-01,
        9.31836143e-01,  4.95840654e-01,  3.31099944e-02,
        1.21643146e-04])
```

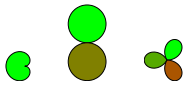
3 - Solution Python des questions 1 et 2 :

• La fonction `shuffle()` rebat les cartes : la commande `shuffle(M)` remplace toute matrice à une dimension `M` par la matrice qui en est issue quand on permute aléatoirement équiprobablement les éléments de `M`. La valeur initiale de `M` est perdue.

Exemple :

```
M = np.arange(7)
M
Out[4]: array([0, 1, 2, 3, 4, 5, 6])# Matrice à une dimension.
```

```
np.random.shuffle(M)
M
Out[6]: array([6, 0, 2, 3, 5, 1, 4])
```



Chapitre 5

Matrices (matrices (n,m))

Liste des exercices :

Énoncé n° 5.1 [*] : Matrices croix (fantaisie).

Énoncé n° 5.2 [*] : Matrice Zorro (fantaisie)

Énoncé n° 5.3 [*] : Transformations aléatoires de matrices.

Énoncé n° 5.4 [*] : Calculer la trace d'une matrice.

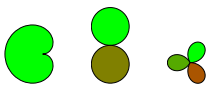
Énoncé n° 5.5 [**] : Ce carré est-il magique ?

Énoncé n° 5.6 [***] : Transposer une matrice

Énoncé n° 5.7 [****] : Écrire toutes les permutations de `(1, ..., n)`.

Énoncé n° 5.8 [****] : Carrés magiques d'ordre 3.

Les matrices `(n,m)` ou matrices à deux dimensions : le nombre de lignes `n` et le nombre de colonnes `m` sont les matrices des mathématiciens ! Le module `numpy` leur est consacré, voir [12].



5.1 [*] Matrice croix (fantaisie).

Écrire la matrice carrée M de taille 2n+1 comportant des 1 sur la (n+1)ième ligne et la (n+1)ième colonne et des 0 ailleurs.

Mots-clefs : module `numpy` : fonctions `np.zeros(( , ), int)`, `np.ones(( , ), int)`, concaténation horizontale et verticale de matrices [1].

Ceci est un exercice d'apprentissage des fonctions citées dans les mots-clefs ci-dessus.

Solution

C'est très facile : `M` est un assemblage de 7 matrices constituées soit de 0, soit de 1, objet de la fonction appelée `croix()` ci-dessous [2].

```
import numpy as np
def croix(n):
    A = np.zeros((n,n),int)
    B = np.ones ((n,1),int)
    C = np.ones ((1,2*n+1),int)
    D = np.concatenate((A,B,A),axis = 1)
    E = np.concatenate((D,C,D),axis = 0)
    return E
```

<u>Exemple :</u> pour `n = 8`,

```
runfile('Chemin␣de␣la␣fonction␣croix')
croix(8)
Out[15]:
array([[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]])
```

1. respectivement `np.concatenate(( , , ),axis = 1)` et `np.concatenate(( , , ),axis = 0)`
2. Comme l'énoncé ne le précise pas, on a supposé que ces 0 et 1 sont des nombres entiers ; sinon, remplacer `int` par `float`

5.2 [*] Matrice Zorro (fantaisie)

Écrire la matrice `M` de taille n, définie comme la matrice carrée de taille n comprenant des 1 sur la première ligne, sur la deuxième diagonale et sur la dernière ligne et des 0 partout ailleurs.

Mots-clefs : module `numpy` : fonctions `np.zeros((n, m),int)`, `np.ones((m, ),int)`, `np.arange(p,q,r)`, extraction d'un bloc d'éléments d'une matrice : `M[m1,m2]`.

Étant donné une matrice `M`, `M[m1,m2]`, où `m1` et `m2` sont des matrices (m,) de même longueur, retourne la matrice `(m, )` dont les éléments sont ceux de `M` dont les indices de ligne forment `m1` et les indices de colonne forment `m2`. Par exemple, `M[arange(n-1,-1,-1), arange(0,n)]` est la deuxième diagonale de `M`. Les matrices `m1` et `m2` peuvent être remplacées par des listes.

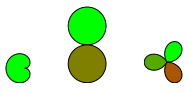
Solution

Pour ce faire, on part d'une matrice carrée d'ordre `n` ne comprenant que des 0 entiers (on aurait pu les choisir flottants), puis on remplace ces 0 par des 1 sur la première et la dernière lignes ainsi que sur la deuxième diagonale. Cette fonction a été appelée `zorro()`.

```
import numpy as np
def zorro(n):
    M = np.zeros((n,n),int)
    M[0,:] = np.ones((n, ),int)
    M[n-1,:] = np.ones((n, ),int)
    M[np.arange(n-2,0,-1),np.arange(1,n-1,1)] = np.ones((1,n-2),int)
    return M
```

Exemple : `n=13`

```
runfile('Chemin de la fonction zorro')
zorro(13)
Out[2]:
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
```



5.3 [*] Transformations aléatoires de matrices

1 - Programmer une fonction qui, pour tout entier `n` positif, retourne la matrice carrée dont la première ligne est `1, 2, …, n`, la suivante `n+1, n+2, …, 2n`, et ainsi de suite jusqu'à la dernière ligne [a].
2 - À l'aide de la fonction `shuffle()` de `numpy.random`, effectuer une permutation aléatoire équiprobable des éléments d'une matrice quelconque donnée `M` et afficher la matrice obtenue.

a. qui est donc `n(n-1)+1, n(n-1)+2, …, n*n`

Mots-clefs : fonctions `arange()`, `reshape()` du module `numpy`, fonction `shuffle()` du module `numpy.random`, commande `B = A.flatten()`, voir ci-dessous.

Rappels :

• Étant donné une matrice à une dimension `M`, la commande `shuffle(M)` retourne une matrice à une dimension qui est une permutation de `M` choisie équiprobablement parmi toutes ses permutations possibles. La valeur initiale de `M` est perdue.
• La commande `B = A.flatten()` [3] retourne la matrice à une dimension `B` qui est en fait la matrice `A` écrite en ligne, les lignes successives de `A` étant écrites à la queue leu-leu.
• Si `A` est une matrice à une ou deux dimensions dont `n` est le nombre d'éléments, `A.reshape(p,q)` [4] où p et q sont des entiers tels que `p*q = n`, retourne une matrice à p lignes et q colonnes écrites dans l'ordre attendu [5].

Solution

1 - La fonction `papageno()` ci-dessous est une solution quasi évidente de la première question :

```
import numpy as np
def papageno(n):# n est un entier positif.
    A = np.arange(1,n*n+1)
    return A.reshape(n,n)
```

Exemple :

```
from papageno import *
papageno(125)
Out[16]:
array([[    1,     2,     3, ...,   123,   124,   125],
       [  126,   127,   128, ...,   248,   249,   250],
       [  251,   252,   253, ...,   373,   374,   375],
       ...,
       [15251, 15252, 15253, ..., 15373, 15374, 15375],
       [15376, 15377, 15378, ..., 15498, 15499, 15500],
       [15501, 15502, 15503, ..., 15623, 15624, 15625]])
```

2 - La fonction `papagena()` ci-dessous est une solution plutôt facile de la deuxième question :

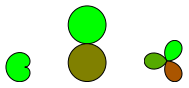
```
import numpy as np
from numpy.random import shuffle
```

3. `flatten()` est une méthode de l'objet `matrice`.
4. `reshape()` est une méthode de l'objet `matrice`.
5. `A` ayant été écrite en ligne, les `q` premiers termes de cette ligne forment la première ligne de `A.reshape(p,q)`, *etc.*

```
def papagena(M):
    N = M.flatten()
    shuffle(N)
    P = N.reshape(np.shape(M))
    return M,P
```

Exemple :

```
from papagena import *
M = (np.arange(1,26)).reshape(5,5)
papagena(M)
Out[3]:
(array([[ 1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10],
        [11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20],
        [21, 22, 23, 24, 25]]), array([[24, 20, 12,  8, 19],
        [18,  5,  4,  9, 23],
        [11, 16,  7, 22,  3],
        [ 2, 17, 25,  6,  1],
        [14, 10, 13, 21, 15]]))
```



5.4 [*] Calculer la trace d'une matrice

Soit `M` une matrice à `n` lignes et `m` colonnes.
1 - Calculer la somme `t1` de ses éléments dont l'indice de ligne est égal à l'indice de colonne.
2 - Calculer la somme `t2` de ses éléments dont l'indice de ligne `i` et l'indice de colonne `j` vérifient `i+j=min(n,m)-1`, les lignes et les colonnes étant numérotées à partir de 0).

Mots-clefs : boucle `pour` ; module `numpy` : extraire un bloc d'éléments d'une matrice, changer l'ordre des lignes ou des colonnes d'une matrice, fonctions `shape()`, `diag()`, `sum()`, `trace()` ; module `numpy.random` : fonction `rand()`, module `copy` : fonction `deepcopy`.

Si `M` est carrée (`n=m`), `t1` est la somme de ses éléments diagonaux, `t2` la somme des éléments de sa deuxième diagonale. Si `M` n'est pas une matrice carrée, on se ramène immédiatement à ce cas. En effet, il est clair que `t1` et `t2` ne changent pas quand on remplace `M` par la sous-matrice de `M` formée par ses `min(n,m)` premières lignes et ses `min(n,m)` premières colonnes[6].

Solution

1 - Le script ci-dessous donne en fait 3 solutions de la première question, appliquées à un exemple.

• Afin de comparer ces solutions, nous avons ajouté le calcul des durées d'exécution.

• Au cours de la première solution, la matrice `M` est modifiée. Comme nous avons besoin de sa valeur initiale pour calculer les solutions suivantes, nous avons conservé cette valeur à l'aide de l'instruction `MM = deepcopy(M)`.

• Comme il est commode d'utiliser des matrices aléatoires[7] quand on a besoin d'une matrice pour tester un script, on a choisi ci-dessous une matrice `M` de taille (10000,120000) dont les éléments sont tirés au hasard suivant la loi uniforme sur l'intervalle [0,1], indépendamment les uns des autres.

```
import numpy as np
from time import clock
M = np.random.rand(10000,120000)# Matrice à 1 200 000 000 éléments !
from copy import deepcopy
MM = deepcopy(M)# MM est une copie indépendante de M.
a = clock()
p = min(np.shape(M))# Minimum du 2-uplet np.shape(M).
M = M[0:p,0:p]# La valeur initiale de M est perdue. M est carrée.
t11 = M[0,0]# Premier terme de la diagonale de M.
for i in range(1,p):# i prend successivement les valeurs 1, ..., p-1
    t11 = t11 + M[i,i]
# t11 est la trace de M.
b = clock()
d11 = b-a
# Deuxième solution.
a = clock()
diag2 = MM[list(np.arange(0,10000,1)),list(np.arange(0,10000,1))]# Matrice
# à une dimension des éléments diagonaux de M
t12 = sum(diag2)# t12 est la trace de MM et de M.
b = clock()
d12 = b-a
# Troisième solution
a = clock()
t13 = np.trace(MM)# trace() est une fonction du module numpy.
```

6. commande `M=M[0:min(n,m),0:min(n,m)]`
7. parce qu'elles sont très faciles à engendrer

```
b = clock()
d13 = b-a
```

Commentaires : La solution n°1 est très simple car elle ne comporte qu'une boucle `pour`. La solution n°2 est plus sophistiquée car on y extrait les éléments de MM situés sur une ligne oblique[8]. La solution n°3 est fournie par `Python` : c'est la fonction `trace()` de `numpy`.

Suite des calculs, sur la console :

```
runfile('Chemin du script')
t11,t12,t13
Out[2]: (4993.9678710011149, 4993.9678710011158, 4993.9678710011158)
d11,d12,d13
Out[3]: (1.9125719999999973, 2.0644849999999977, 0.005915999999999144)
```

On voit que `trace()` est beaucoup plus rapide que les solutions 1 et 2. C'est cette fonction que l'on utilise normalement.

2 - À étant donné une matrice `M`, on peut la remplacer par la matrice carrée `MM` définie comme précédemment. La commande `N = MM[range(shape(MM)[0]-1,-1,-1),:]` retourne la matrice carrée `N` de même dimension que `MM` dont la première ligne est la dernière ligne de `MM`, etc[9]. Dans cette manipulation, les diagonales se sont échangées. En particulier, la première diagonale de `N` est la deuxième diagonale de `MM` dont la somme est donc `t2`. La matrice `M` étant donnée, le calcul de `t2` se résume donc maintenant à :

```
p = min(np.shape(M))
M = M[0:p,0:p]
t2 = trace(M[range(p-1,-1,-1),:])
```

Exemple : dans une nouvelle console,

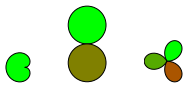
```
import numpy as np
M = np.random.rand(4,6)
M
Out[3]:
array([[ 0.09844368,  0.36667626,  0.48009385,  0.87182384,  0.1740172 ,
         0.37642479],
       [ 0.38778403,  0.98906187,  0.61910549,  0.40341863,  0.59408361,
         0.18363526],
       [ 0.99841847,  0.1614101 ,  0.43666381,  0.59053185,  0.45606471,
         0.23099434],
       [ 0.33903413,  0.31377958,  0.12998992,  0.90110014,  0.19849328,
         0.84980818]])
p = min(np.shape(M))
M = M[0:p,0:p]
M
Out[6]:
array([[ 0.09844368,  0.36667626,  0.48009385,  0.87182384],
       [ 0.38778403,  0.98906187,  0.61910549,  0.40341863],
       [ 0.99841847,  0.1614101 ,  0.43666381,  0.59053185],
       [ 0.33903413,  0.31377958,  0.12998992,  0.90110014]])
M = M[range(p-1,-1,-1),:]
M
```

8. commande `diag2 = MM[list(np.arange(0,10000,1)),list(np.arange(0,10000,1))]`

9. de même pour les lignes : la commande `P = MM[:,range(shape(MM)[1]-1,-1,-1)]` retourne la matrice `P` dont la première colonne est la dernière colonne de `MM`, etc

```
Out[8] :
array([[ 0.33903413,  0.31377958,  0.12998992,  0.90110014],
       [ 0.99841847,  0.1614101 ,  0.43666381,  0.59053185],
       [ 0.38778403,  0.98906187,  0.61910549,  0.40341863],
       [ 0.09844368,  0.36667626,  0.48009385,  0.87182384]])
t2 = np.trace(M)
t2
Out[10] : 1.9913735509512127
0.33903413+0.1614101+0.61910549+0.87182384
Out[11] : 1.9913735600000002
```

Le calcul à la main de la trace de `M` se fait avec les nombres affichés, qui sont arrondis, ce qui explique la légère différence avec `t2`.



5.5 [**] Ce carré est-il magique ?

On appelle carré magique d'ordre `n` toute matrice carrée `M` d'ordre `n` dont les éléments sont les entiers de 1 à n^2 disposés de sorte que leurs sommes sur chaque ligne, sur chaque colonne et sur les deux diagonales soient égales. La valeur commune de ces sommes est appelée constante magique de `M`.
1 - S'il existe un carré magique d'ordre `n`, combien vaut sa constante magique ?
2 - `LuoShu = array([[4,9,2],[3,5,7],[8,1,6]])` est-il un carré magique ? Si oui, quelle est sa constante magique ?
3 - Même question pour `Cazalas = array([[1,8,53,52,45,44,25,32],`
`[64,57,12,13,20,21,40,33], [2,7,54,51,46,43,26,31],`
`[63,58,11,14,19,22,39,34], [3,6,55,50,47,42,27,30],`
`[62,59,10,15,18,23,38,35], [4,5,56,49,48,41,28,29],`
`[61,60,9,16,17,24,37,36]])`.
4 - Même question pour `Franklin = array([[52,61,4,13,20,29,36,45],`
`[14,3,62,51,46,35,30,19], [53,60,5,12,21,28,37,44],`
`[11,6,59,54,43,38,27,22], [55,58,7,10,23,26,39,42],`
`[9,8,57,56,41,40,25,24], [50,63,2,15,18,31,34,47],`
`[16,1,64,49,48,33,32,17]])`.

Mots-clefs : `M.size`, `M.sum(axis=1)`, `M.sum(axis=0)`, `trace(M)`, `M[arange(n-1,-1,-1), arange(0,n)]` (voir ci-dessous), `concatenate(( , ),axis =)`, `min()`, `max()`, `shape()`, `reshape()`.

Solution [10]

1 - La somme des éléments de ce carré est $1+2+\cdots+n^2 = \frac{n^2(n^2+1)}{2}$. Ce nombre est aussi n fois la constante magique [11]. La constante magique est donc $\dfrac{n(n^2+1)}{2}$.

Solution n°1 des questions 2, 3 et 4 [12] **:** Ces questions sont faciles puisque, une matrice d'ordre `n` dont les éléments sont les entiers de 1 à n^2 étant donnée, il suffit de calculer les sommes de ses éléments sur chaque ligne, chaque colonne et sur les deux diagonales et de les comparer. On pourra utiliser les commandes suivantes :
• `M.size`, `M` étant une matrice à une ou deux dimensions, retourne le nombre de ses éléments,
• `M.sum(axis=1)` retourne la matrice à une dimension des sommes des lignes de `M`,
• `M.sum(axis=0)` retourne la matrice à une dimension des sommes des colonnes de `M`,
• `trace(M)` retourne la somme des éléments de la diagonale principale de `M` [13] ,
• `M[m1,m2]` où `m1` et `m2` sont des matrices (m,) de même longueur retourne la matrice (m,) dont les éléments sont ceux de `M` dont les indices de ligne forment `m1` et les indices de colonne forment `m2`. Les matrices `m1` et `m2` peuvent être remplacées par des listes. Par exemple, `M[arange(n-1,-1,-1), arange(0,n)]` est la deuxième diagonale de `M` [14], [15]
• `np.concatenate((A,B),axis=0)` retourne la matrice obtenue en empilant verticalement les matrices A et B qui doivent avoir le même nombre de dimensions (soit matrices à une dimension, soit matrices à 2 dimensions) et le même nombre de colonnes. La commande `np.diag(M)[np.newaxis,:]` ajoute une dimension à `diag(M)`.C'est une matrice (1,n).

10. Voir l'intéressant article [18] de Wikipedia sur les carrés magiques.
11. Il y a n lignes, la somme de chacune d'elles étant la constante magique.
12. Il est sans doute judicieux de se reporter directement à la solution n°2 des questions 2, 3 et 4 ci-dessous. Cette deuxième solution, quoique comportant des tests, se révèle en effet beaucoup plus rapide, car ces tests évitent de nombreux calculs.
13. `trace()` est une fonction du module `numpy`, voir l'exercice 5.4.
14. analogue de `diag(M)` qui retourne la première diagonale de `M`
15. On peut aussi calculer la somme des éléments de la deuxième diagonale de `M` avec la commande : `trace(A[:,range(shape(A)[1]-1,-1,-1)])`. Voir l'exercice 5.4.

Cela donne la fonction `mascarille` suivante :

```
# M étant une matrice carrée d'ordre n dont les éléments
# sont les entiers de 1 à n*n, la fonction mascarille()
# retourne suivant le cas 'M est ou n'est pas un carré magique'
# ainsi, optionnellement (sans #), que le temps de calcul.
# Une autre option (##) est de retourner 1 si M est un carré magique, 0 sinon.
import numpy as np
#from time import clock
def mascarille(M):
    #a = clock()
    n = M[:,0].size# Nombre de colonnes de M.
    MM = np.concatenate((M,M.T,np.diag(M)[np.newaxis,:],
                         np.diag(M[:,range(n-1,-1,-1)])[np.newaxis,:]),axis=0)
    c = n*(n**2+1)//2# Constante magique.
    MMM = MM.sum(axis=1)
    #b = clock()
    if np.array_equal(MMM,c*np.ones((2*n+2,),int)):
        return "M est un carré magique"#,b-a##,1
    else:
        return "M n'est pas un carré magique"#,b-a##,0
```

Commentaires : Dans ce script, on empile donc verticalement les quatre matrices à deux dimensions `M`, `M.T`, `np.diag(M)[np.newaxis,:]` et `np.diag(M[:,range(n-1,-1,-1)])[np.newaxis,:]`. On calcule les sommes de toutes les lignes de la matrice obtenue, qui produit une matrice à une dimension. On la compare avec la matrice à une dimension et `n` colonnes ne comportant que des `c`. On teste ensuite l'égalité de ces matrices.

Application aux matrices `LuoShu`, `Cazalas` et `Franklin` :

```
from mascarille import *
LuoShu = np.array([[4,9,2],[3,5,7],[8,1,6]])
mascarille(LuoShu)
Out[3]: 'M est un carré magique'
Cazalas = array([[1,8,53,52,45,44,25,32],
[64,57,12,13,20,21,40,33],
[2,7,54,51,46,43,26,31],
[63,58,11,14,19,22,39,34],
[3,6,55,50,47,42,27,30],
[62,59,10,15,18,23,38,35],
[4,5,56,49,48,41,28,29],
[61,60,9,16,17,24,37,36]])
mascarille(Cazalas)
Out[5]: 'M est un carré magique'
Franklin = array([[52,61,4,13,20,29,36,45],
[14,3,62,51,46,35,30,19],
[53,60,5,12,21,28,37,44],
[11,6,59,54,43,38,27,22],
[55,58,7,10,23,26,39,42],
[9,8,57,56,41,40,25,24],
[50,63,2,15,18,31,34,47],
[16,1,64,49,48,33,32,17]])
mascarille(Franklin)
Out[7]: "M n'est pas un carré magique"
```

On constate que les deux premiers carrés proposés sont magiques tandis que le troisième ne l'est pas, contrairement à ce qui est dit dans [18]. Regardons le détail des calculs :

```
Franklin.sum(axis=1)
Out[9]: array([260, 260, 260, 260, 260, 260, 260, 260])
Franklin.sum(axis=0)
Out[10]: array([260, 260, 260, 260, 260, 260, 260, 260])
n = Franklin[:,0].size
array([trace(Franklin),sum(Franklin[arange(n-1,-1,-1), arange(0,n)])])
Out[12]: array([228, 292])
```

On constate que les sommes des lignes et des colonnes de la matrice de Franklin sont égales à 260 alors que sa trace vaut 228 et la somme des éléments de la deuxième diagonale 292 !

Solution n°2 des questions 2, 3 et 4 :
La définition de la fonction `marotte()` qui suit contient des tests qui permettent d'arrêter des calculs inutiles [16]. Ces calculs sont faits quand on utilise la fonction `mascarille()` ci-dessus.

```
# Si M est une matrice carrée dont les éléments sont les entiers de 1 à n*n,
# marotte(M) retourne 1 si M est un carré magique, 0 sinon.
# marotte() semble beaucoup plus rapide que mascarille().
import numpy as np
def marotte(M):
    n = np.shape(M)[0]
    c = n*(n*n+1)//2
    if (sum(np.diag(M))!= c) or (sum(np.diag(M[:,range(n-1,-1,-1)]))!= c):
        return 0
    for i in range(n-1):
        if (sum(M[i,:])!= c) or (sum(M[:,i])!= c):
            return 0
    return 1
```

Exemple :
Appliquons `marotte()` à une matrice `M` d'un million de termes [17] en faisant apparaître la durée du calcul :

```
import numpy as np
import marotte as mar
from time import clock
M = np.arange(1,1000001)# M = np.array([1, ...,1000000])
np.random.shuffle(M)#Brouillage aléatoire des éléments de M.
M = M.reshape(1000,1000)# M est transformée en matrice (1000,1000)
a = clock()# Déclenchement du chronomètre.
x = mar.marotte(M)
b = clock()# Arrêt du chrnomètre.
print(x,b-a)
```

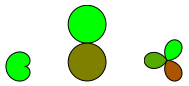
Exécutons-le :

```
runfile('Chemin du script')
0 [[330450 195238 123356 ..., 805557 635024 471971]
 [ 43855 957950 912146 ..., 430652 269016 302098]
 [ 25366 835825 331316 ..., 972094 101682 346065]
 ...,
```

16. grâce à la commande `return`
17. n = 1000

```
 [473576  666289  964539 ...,  219784  721260  588693]
 [396011  694839  760462 ...,  643871  620982  497566]
 [345506   86068  334800 ...,  811284  731600  125029]] 0.0001930000000003318
```

L'exécution de `marotte()` a duré moins de deux dix-millièmes de seconde. `M` n'est pas un carré magique.



5.6 [***] Transposer une matrice

Programmer une fonction qui, étant donné une matrice à deux dimensions `A`, retourne la matrice `B` dont la première colonne est la première ligne de `A`, la seconde la deuxième ligne de `A`, etc. `B` s'appelle la transposée de `A`.

Mots-clefs : Boucle `pour` imbriquée dans une autre ; commande `range` ; module `numpy` : fonctions `np.shape()` (dimension d'une matrice), `np.zeros()`, `np.concatenate(( , ),axis = )` (empilements horizontaux et verticaux de matrices), `np.transpose()` ; méthode (de l'objet matrice) `.T`.

Solution n°1 - Il y a évidemment des commandes de Python qui retournent la transposée d'une matrice à deux dimensions [18], [19] :

```
import numpy as np
A = np.array([[1.5,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
#A est une matrice (3,5) de nombres réels.
print(A)
Bt = np.transpose(A)
print(Bt)#Matrice (5,3).
BT = A.T
print(BT)# Matrice (5,3). Bt et BT sont égales à la transposée de A.
```

Exécutons ce script dans la console :

```
runfile('Chemin du script précédent')
[[  1.5   2.   3.   4.   5. ]
 [  6.    7.   8.   9.  10. ]
 [ 11.   12.  13.  14.  15. ]]
[[  1.5   6.  11. ]
 [  2.    7.  12. ]
 [  3.    8.  13. ]
 [  4.    9.  14. ]
 [  5.   10.  15. ]]
[[  1.5   6.  11. ]
 [  2.    7.  12. ]
 [  3.    8.  13. ]
 [  4.    9.  14. ]
 [  5.   10.  15. ]]
```

Une matrice à deux dimensions qui n'a qu'une ligne est transformée en une matrice à deux dimensions qui n'a qu'une colonne et inversement. `transpose()` et `.T` n'agissent pas sur les matrices à une dimension, comme le montre l'exemple suivant :

```
import numpy as np
A = np.array([1,2,3,4,5])# A est une matrice (m, ) d'entiers.
print(A)
Bt = np.transpose(A)
print(Bt)# Bt est une matrice (5,1).
BT = A.T
print(BT)# Bt est une matrice (5,1).
```

18. `transpose()` est une fonction du module `numpy` qui retourne la transposée de toute matrice à deux dimensions.
19. `.T` est une méthode de l'objet matrice - à deux dimensions - qui transpose cette matrice.

Exécutons ce script dans la console :

```
runfile('Chemin du script précédent')
[1 2 3 4 5]
[1 2 3 4 5]
[1 2 3 4 5]
```

Solution n°2 - Programmons maintenant nous-mêmes une fonction transposant toute matrice `A` à deux dimensions. Pour cela, il suffit de remarquer que l'élément qui est sur la i$^{\text{ème}}$ ligne et la j$^{\text{ème}}$ colonne de la matrice transposée `B` est `A(j,i)` [20]. Si `l` et `c` désignent respectivement le nombre de lignes et le nombre de colonnes de `A` [21], `i` courra de 1 à `c` et `j` courra de 1 à `l`. Il y aura donc une boucle `pour` imbriquée dans une boucle `pour` dans l'algorithme définissant la fonction appelée `transpoze()` ci-dessous :

```
import numpy as np
def transpoze(A):# A est une matrice à 2 dimensions données.
    a = type(A[0,0])# a est le type des éléments de A.
    (l,c) = np.shape(A)
    B = np.zeros((c,l),a)
    # B est une matrice de 0 de même type que les éléments de A.
    # La dimension de B est (c,l).
    for i in range(c):
        for j in range(l):
            B[i,j] = A[j,i]
    return B
```

Exemple :

```
import numpy as np
A = np.random.rand(3,5)
A
Out[3]:
array([[ 0.49253415,  0.8270516 ,  0.30327453,  0.11759094,  0.19154943],
       [ 0.2482007 ,  0.58941235,  0.55208151,  0.21272643,  0.23828777],
       [ 0.08465715,  0.0335026 ,  0.93107188,  0.96414524,  0.96205538]])
runfile('Chemin de la fonction transpoze')
A = transpose(A)
A
Out[6]:
array([[ 0.49253415,  0.2482007 ,  0.08465715],
       [ 0.8270516 ,  0.58941235,  0.0335026 ],
       [ 0.30327453,  0.55208151,  0.93107188],
       [ 0.11759094,  0.21272643,  0.96414524],
       [ 0.19154943,  0.23828777,  0.96205538]])
transpose(A)
Out[7]:
array([[ 0.49253415,  0.8270516 ,  0.30327453,  0.11759094,  0.19154943],
       [ 0.2482007 ,  0.58941235,  0.55208151,  0.21272643,  0.23828777],
       [ 0.08465715,  0.0335026 ,  0.93107188,  0.96414524,  0.96205538]])
```

On constate, ce qui était évident, que la transposée de la transposée de `A` est `A`.

Solution n°3 - Dans l'espoir d'obtenir un algorithme plus rapide, essayons de manipuler directement les

20. commande `B[i,j] = A[j,i]`
21. donnés par la commande `shape(A)`, qui retourne le 2-uplet `(l,c)` ; `l = shape(A)[0]` ; `c = shape(A)[1]`

lignes ou les colonnes de `A`. On sait que si `A` est une matrice à deux dimensions et si nous en extrayons la première colonne (commande `A[:,0]`), on obtient une matrice `(m, )` qui peut être transformée en une matrice à deux dimensions (1,c) (commande `A[:,0][np.newaxis,:]`). Il reste à faire de même pour les autres colonnes de `A` et à empiler les `l` matrices (1,c) obtenues à l'aide de la fonction `concatenate((,),axis = 0)` de `numpy`[22].

```
import numpy as np
def transpozze(A):# A est une matrice donnée à 2 dimensions.
    (l,c) = np.shape(A)
    B = A[:,0][np.newaxis,:]# Matrice à 2 dimensions réduite à une ligne
    # de l colonnes.
    for i in range(1,c):
        B = np.concatenate((B,A[:,i][np.newaxis,:]),axis = 0)
    return B# Matrice à c lignes et l colonnes : transposée de A.
```

Exemple :

```
runfile('Chemin de la fonction transpozze')
A = np.random.rand(4,6)
A
Out[3]:
array([[ 0.14375364,  0.41827656,  0.10152442,  0.05841431,  0.1026309 ,
         0.03954335],
       [ 0.79758356,  0.50179013,  0.95236493,  0.94198123,  0.54791014,
         0.83251176],
       [ 0.75258107,  0.26999299,  0.93020474,  0.46672265,  0.31675741,
         0.87933763],
       [ 0.0363524 ,  0.8232996 ,  0.13521136,  0.53129757,  0.38076294,
         0.17876873]])
transpozze(A)
Out[4]:
array([[ 0.14375364,  0.79758356,  0.75258107,  0.0363524 ],
       [ 0.41827656,  0.50179013,  0.26999299,  0.8232996 ],
       [ 0.10152442,  0.95236493,  0.93020474,  0.13521136],
       [ 0.05841431,  0.94198123,  0.46672265,  0.53129757],
       [ 0.1026309 ,  0.54791014,  0.31675741,  0.38076294],
       [ 0.03954335,  0.83251176,  0.87933763,  0.17876873]])
```

Supplément : Comparons sur un exemple les fonctions `transpose()`, `transpoze()` et `transpozze()`. On s'attend à ce que la première soit la plus rapide, suivie par la troisième. Pour cela, exécutons le script suivant dans une nouvelle console.

```
import numpy as np
import transpoze as z
import transpozze as zz
from time import clock
A = np.random.rand(1000,2500)
a = clock()
B = np.transpose(A)
b = clock()
d = b - a
```

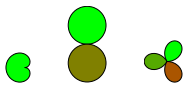
22. np.concatenate((A ,B),axis = 0) empile verticalement les matrices à deux dimensions `A` et `B` pourvu qu'elles aient le même nombre de colonnes.

```
a = clock()
Bz = z.transpoze(A)
b = clock()
dz = b - a
a = clock()
Bzz = zz.transpozze(A)
b = clock()
dzz = b - a
print(d, dz, dzz)
```

On obtient :

```
runfile('Chemin du script')
Reloaded modules: transpoze, transpozze
0.0016400000000018622 0.6863809999999972 22.59821
```

`transpozze()` est la plus lente !



5.7 [****] Écrire toutes les permutations de `(1, ..., n)`.

1 - `M` désignant une matrice quelconque d'entiers à `li` lignes et `co` colonnes, `n` un entier quelconque, programmer une fonction qui retourne la matrice `Sortie` qui empile verticalement les `co+1` matrices obtenues en adjoignant à `M` une colonne `A` à `li` lignes dont tous les éléments sont égaux à `n`, `A` étant placée d'abord devant `M`, puis entre la première et deuxième colonne de `M`[a], *etc*, jusqu'à ce que `A` devienne sa dernière colonne. La matrice obtenue aura `co+1` colonnes et `(n+1).li` lignes.
2 - Écrire toutes les permutations de `(1, ..., n)`.

a. ce qui revient à décaler `A` d'un cran vers la droite ou d'échanger les deux premières colonnes

Mots-clefs : appel de modules et de fonctions ; module `numpy` : fonctions `ones(( , ), int)`, `zeros(( , ), float)`, `concatenate(( , ), axis = )`, échange de colonnes d'une matrice, extraction d'une sous-matrice ; méthode `shape()` des matrices ; fonction `clock()` du module `time`.

Solution

1 - La fonction `nabucco()` définie ci-dessous [23] suit fidèlement l'énoncé de la question 1 [24] :

```
# Fonction auxiliaire sans intérêt en soi.
# M est par hypothèse une matrice d'entiers à deux dimensions, n un entier.
def nabucco(M,n):
    import numpy as np
    (li,co) = M.shape
    A = n*np.ones((li,1),int)# Colonne de n à li lignes.
    Sortie = np.zeros(((co+1)*li,co+1),int)# Initialisation de la future
    # sortie de nabucco().
    B = np.concatenate((A,M),axis=1)
    # B est une matrice à li lignes et co+1 colonnes obtenue
    # en mettant A devant M.
    Sortie[0:li,:] = B# Début du calcul de la sortie de nabucco().
    # Le bloc de zéros de la valeur initiale de Sortie, porté par
    # les lignes 0, ..., li-1, est remplacé par B.
    for i in range(co):
        B[:,[i,i+1]] = B[:,[i+1,i]]# Echange des colonnes i et i+1 de B. En
        # clair, dans B, la colonne de n se déplace d'un cran vers la droite.
        Sortie[(i+1)*li:(i+2)*li,0:co+1] = B# Le bloc de zéros de Sortie
        # porté par les lignes d'indices (i+1)*li à (i+2)*li-1 est remplacé
        #par B. A la fin, Sortie a (co+1).li lignes et co+1 colonnes.
    return Sortie
```

Voici une autre version de la fonction `nabucco` [25] :

```
# nabucco est une fonction auxiliaire sans intérêt en soi.
# M est par hypothèse une matrice d'entiers à deux dimensions, n un entier.
import numpy as np
def nabucco(M,n):
    (l,c) = M.shape
```

23. script commenté
24. Elle sera utilisée pour définir la fonction `permut()` qui suit.
25. moins rapide que la première version de `nabucco()`

```
    S = n*np.ones((l,c+1),int)
    for i in range(c+1):
        A = n*np.ones((l,c+1),int)
        A[:,0:i] = M[:,0:i]
        A[:,i+1:c+1] = M[:,i:c]
        S = np.concatenate((S,A),axis = 0)
    return S[l:l*(c+2),:]
```

Commentaires : La première valeur de `S` est une matrice de 1 à `l` lignes et `c+1` colonnes qu'il faudra retirer à la fin et qui permet d'empiler les matrices d'intérêt à l'aide de la fonction `np.concatenate()`.

Exemples :

Exemple n°1 :

```
runfile('Chemin de la fonction nabucco')
M = np.array([[1]])
nabucco(M,17)
Out[3]:
array([[17,  1],
       [ 1, 17]])
```

Exemple n°2 :

```
from nabucco import *
M = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13,14,15],[16,17,18]])
nabucco(M,-111)
Out[3]:
array([[-111,    1,    2,    3],
       [-111,    4,    5,    6],
       [-111,    7,    8,    9],
       [-111,   10,   11,   12],
       [-111,   13,   14,   15],
       [-111,   16,   17,   18],
       [   1, -111,    2,    3],
       [   4, -111,    5,    6],
       [   7, -111,    8,    9],
       [  10, -111,   11,   12],
       [  13, -111,   14,   15],
       [  16, -111,   17,   18],
       [   1,    2, -111,    3],
       [   4,    5, -111,    6],
       [   7,    8, -111,    9],
       [  10,   11, -111,   12],
       [  13,   14, -111,   15],
       [  16,   17, -111,   18],
       [   1,    2,    3, -111],
       [   4,    5,    6, -111],
       [   7,    8,    9, -111],
       [  10,   11,   12, -111],
       [  13,   14,   15, -111],
       [  16,   17,   18, -111]])
```

2 - Dans le script ci-dessous [26], pour calculer toutes les permutations des entiers de 1 à `n` écrites en colonne [27], on écrit toutes les permutations de 1, puis de (1, 2), puis de (1, 2, 3), *etc* [28], ce que l'on peut représenter par :

$$1 \quad \longrightarrow \quad \begin{matrix} 2 & 1 \\ 1 & 2 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 3 & 2 & 1 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \end{matrix} \quad \longrightarrow$$

et on s'arrête quand on a obtenu les permutations de $(1, 2, \ldots, n)$.

```
# La fonction permut() associe, à tout entier positif n,
# la matrice à n! lignes et n colonnes des permutations de 1, ..., n,
# chaque ligne étant une permutation. Optionnellement, elle retourne aussi
# la durée du calcul.
import numpy as np
import nabucco as nab
from time import clock
def permut(n):
    a = clock()
    M = np.array([[1]])# Initialisation de M.
    for i in range(1,n):
        M = nab.nabucco(M, i+1)
    b = clock()
    return M, b-a
```

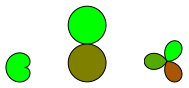
Exemples :

```
runfile('Chemin de la fonction permut')
permut(10)
Out[2]:
(array([[10,  9,  8, ...,  3,  2,  1],
        [10,  9,  8, ...,  3,  1,  2],
        [10,  9,  8, ...,  2,  3,  1],
        ...,
        [ 1,  3,  2, ...,  8,  9, 10],
        [ 2,  1,  3, ...,  8,  9, 10],
        [ 1,  2,  3, ...,  8,  9, 10]]), 0.36113399999999984)
permut(11)
Out[3]:
(array([[11, 10,  9, ...,  3,  2,  1],
        [11, 10,  9, ...,  3,  1,  2],
        [11, 10,  9, ...,  2,  3,  1],
        ...,
        [ 1,  3,  2, ...,  9, 10, 11],
        [ 2,  1,  3, ...,  9, 10, 11],
        [ 1,  2,  3, ...,  9, 10, 11]]), 5.130067)
permut(12)
Out[4]:
(array([[12, 11, 10, ...,  3,  2,  1],
        [12, 11, 10, ...,  3,  1,  2],
        [12, 11, 10, ...,  2,  3,  1],
        ...,
```

26. script commenté ; la fonction définie a été appelée `permut()`
27. ce qui donnera une matrice à $n!$ lignes et à n colonnes
28. à l'aide de la fonction `nabucco()`

```
        [ 1,  3,  2, ..., 10, 11, 12],
        [ 2,  1,  3, ..., 10, 11, 12],
        [ 1,  2,  3, ..., 10, 11, 12]]), 127.04567899999999)Populating
       the interactive namespace from numpy and matplotlib
permut(13)
Kernel est mort, redémarré
```

On constate que les temps de calcul augmentent très vite avec `n` parce que la matrice `permut(n)` grandit très vite. Par exemple, `permut(12)` a `factorial(12)` = 479 001 600 lignes et 12*12 ! = 5 748 019 200 éléments. C'est beaucoup. À partir de `n=13`, il y a un dépassement de capacité de mémoire [29].



29. `factorial(13)` = 6 227 020 800 et `13*factorial(13)`= 80 951 270 400.

5.8 [****] Combien y a-t-il de carrés magiques d'ordre 3 ?

Combien y a-t-il des carrés magiques d'ordre 3 ?

Mots-clefs : permutations, appeler une fonction ; module `math` : fonction `factorial()` ; module `time` : fonction `clock()` ; module `numpy` : matrices à une dimension (fonctions `min()`, `max()`, méthode `reshape()`), matrices à deux dimensions (extraction d'une ligne, somme sur les lignes et les colonnes, fonction `trace()`), conversion d'une matrice à une dimension en une matrice à deux dimensions ; listes : commande `append()`.

Cet exercice fait suite aux exercices 5.5 et 5.7.

Solution :

• Nous allons engendrer toutes les matrices à `n` lignes et à `n` colonnes dont les éléments sont tous les entiers de `1` à `n*n`, à l'aide de la fonction `permut()` [30] de l'exercice 5.7 [31].

• puis nous les prendrons une par une. Quand nous tomberons sur un carré magique [32], nous l'ajouterons à la liste des carrés magiques déjà trouvés (liste initialisée à `[ ]`) et nous ajouterons 1 à `c`, un compteur initialisé à 0.

L'algorithme suivant résout le problème posé. Cette fonction, appelée `cmagique()`, retourne aussi la durée du calcul et le nombre de carrés magiques trouvés. C'est donc un 3-uplet qui sera retourné.

```
# La fonction cmagique() retourne, pour tout entier positif n,
# le 3-uplet formé de la durée du calcul, de la liste A
# des carrés magiques d'ordre n et du nombre de ces carrés magiques.
import permut as per# Version de permut qui retourne
# la matrice des permutations.
import marotte as mar
from time import clock
from math import factorial
def cmagique(n):
    a = clock()
    A = []# Liste vide. Initialisation de la liste des carrés magiques
    # d'ordre n.
    M = per.permut(n*n)# M est une matrice a (n*n)! lignes et
    # n*n colonnes.
    m = factorial(n*n)
    for i in range(m):
        N = M[i]# M[i] est une matrice a une dimension à n*n colonnes
        # (ligne de M d'indice i).
        P = N.reshape(n, n)
        if mar.marotte(P) == 1:
            A.append(P)
    b = clock()
    return b-a, A, len(A)# len(A) est la longueur de A.
```

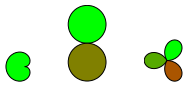
`len(A)` est évidemment le nombre de carrés magiques d'ordre `n` trouvés.

Exemples : Calculons les carrés magiques d'ordre 1, 2, 3, ... [33]

30. commande `permut(n*n)`
31. Nous confondons bien entendu chacune de ces matrices avec la permutation correspondante de `1, ..., n*n`.
32. identifié à l'aide de la fonction `marotte()` de l'exercice 5.5
33. Les nombres de carrés magiques obtenus pourront être différents de ceux que donne [18] parce que cet article de Wikipedia compte comme identiques des carrés magiques qui se déduisent les uns des autres moyennant une transformation simple.

```
from cmagique import *
cmagique(1)
Out[2] : (0.0007120000000000459, [array([[1]])], 1)
cmagique(2)
Out[3] : (0.005373000000000072, [], 0)
cmagique(3)
Out[4] :
(14.244849000000002, [array([[4, 9, 2],
         [3, 5, 7],
         [8, 1, 6]]), array([[2, 9, 4],
         [7, 5, 3],
         [6, 1, 8]]), array([[4, 3, 8],
         [9, 5, 1],
         [2, 7, 6]]), array([[2, 7, 6],
         [9, 5, 1],
         [4, 3, 8]]), array([[8, 3, 4],
         [1, 5, 9],
         [6, 7, 2]]), array([[6, 7, 2],
         [1, 5, 9],
         [8, 3, 4]]), array([[8, 1, 6],
         [3, 5, 7],
         [4, 9, 2]]), array([[6, 1, 8],
         [7, 5, 3],
         [2, 9, 4]])], 8)Populating the interactive namespace from numpy
        and matplotlib
cmagique(4)
Kernel est mort, redémarré
```

Il y a donc 8 carrés magiques d'ordre 3 dont la liste est donnée à l'issue d'un calcul qui est long. À partir de `n = 4`, ces calculs deviennent impossibles (dans ce cas, la matrice `M` a 20 922 789 888 000 lignes et 16 colonnes, ce qui est beaucoup). De plus, la fonction `cmagique()` provoque de nombreux calculs inutiles.



Chapitre 6

Arithmétique, compter en nombres entiers

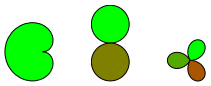
Le calcul en nombres entiers concerne notamment l'arithmétique et les problèmes de dénombrement, donc le calcul des probabilités[1], [2]. Les calculs en nombres entiers donnent souvent lieu à des dépassements de capacité de la mémoire.

Liste des exercices :

Énoncé n° 6.1 [***] : PGCD de `p` entiers positifs

Énoncé n° 6.2 [****] : Crible d'Ératosthène

Énoncé n° 6.3 [***] : Factoriser un entier en produit de nombres premiers.



1. dans le cas où il n'y a qu'un nombre fini d'issues équiprobables
2. Le module du Calcul des probabilités est le module `numpy.random`, voir [1], p.163.

6.1 [***] PGCD de p entiers positifs.

1 - Programmer une fonction qui, étant donnés `p` entiers positifs $a_1, \ldots, a_p$ dont le minimum est noté m, retourne la matrice à une dimension de leurs diviseurs communs en procédant de la manière suivante : retirer de la suite $1, \ldots, m$ les nombres qui ne sont pas des diviseurs de a_1, puis ceux qui ne sont pas des diviseurs de a_2, *etc.*
2 - En déduire leur PGCD.

Mots-clefs : module `numpy`, fonctions `arange()`, `size()`, `sort()`, masque de matrice, `where()`, reste de la division euclidienne, extraction des éléments d'une matrice (m,) dont les indices sont donnés, fonction pré-définie `min()`, module `math`, fonction `gcd` de ce module.

Solution

Notons M la matrice à une dimension[3] dont les éléments sont $a_1, \ldots, a_p$.

1 - Le script de la fonction `miris()` définie ci-dessous reproduit exactement l'énoncé. Les nombres qui restent, formant le dernier état de la matrice à une dimension L, sont les diviseurs communs de $a_1, \ldots, a_p$.

```
# La fonction miris suivante a pour données une matrice à une
# dimension M de  p nombres entiers >0. Elle retourne la matrice
# à une dimension de leurs diviseurs communs.
import numpy as np
def miris(M):
    M = np.sort(M)# M est rangé dans l'ordre croissant, ce qui ne modifie pas
    # le problème. La matrice M n'a pas besoin d'être ordonnée.
    m = min(M)
    p = np.size(M)# n est le nombre d'éléments de M.
    L = np.arange(1,m+1)# Les diviseurs communs sont à rechercher
    # dans L, une matrice à une dimension.
    for i in range(p-1,-1,-1):
        Mat = M[i]%L# Mat est la matrice à une dimension des restes dans
        # les divisions euclidiennes de M[i] par les éléments de L.
        masque = (Mat == 0)# Repérage de ces diviseurs de M[i].
        indices = np.where(masque)# Indices de ces diviseurs.
        L = L[indices]# On ne conserve que ces diviseurs.
    return L
```

Exemple n°1 :

```
from miris import *
M = np.array([114,747,1266])
miris(M)
Out[3]: array([1, 3])
type(miris(M))
Out[4]:
numpy.ndarray
```

114, 747 et 1266 ont donc 1 et 3 comme diviseurs communs. Le maximum de `L` est évidemment le PGCD des `p` entiers positifs considérés.

Exemple n°2 :

3. voir 4

```
M =np.array([2171,318215,41023,16179])
max(miris(M))
Out[5]: 1
```

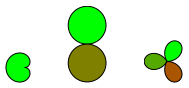
2171, 318215, 41023 et 16179 sont donc premiers dans leur ensemble.

2 - Le PGCD est le plus grand diviseur commun, autrement dit `max(L)`.

Exemple n°3 : Calculons le PGCD de 123 456 789 et 987 654 321 d'abord en utilisant la fonction `miris()` puis la fonction `gcd()` [4], [5] :

```
from miris import *
M = array([123456789,987654321])
max(miris(M))
Out[4]: 9
from math import *
gcd(123456789,987654321)
Out[6]: 9
```

Pendant l'exécution de ce script, on peut constater que la fonction `gcd()` est beaucoup plus rapide que `max(miris( ))`.



4. `gcd()`, fonction du module `math`, retourne le PGCD de deux entiers positifs donnés.
5. `gcd()` repose, sauf erreur, sur l'algorithme d'Euclide, cf.2.9

6.2 [****] Crible d'Ératosthène

1 - Programmer une fonction qui, pour tout entier $n \geqslant 2$ donné, retourne les nombres premiers compris entre 2 et `n`, rangés dans l'ordre croissant.
2 - Programmer une fonction qui, étant donné un nombre entier $n \geqslant 2$, retourne suivant le cas "`n` est premier" ou "`n` n'est pas premier".

Mots-clefs : effacer des éléments d'une matrice à une dimension à l'aide d'un masque.

1 - Rappels : [6] Tout entier $n \geqslant 2$ est appelé nombre premier s'il n'est divisible que par 1 et par lui-même [7].
Algorithme : Pour trouver les nombres premiers $\leqslant n$, partons de la suite $S = 2, 3, \ldots, n$.
• Conservons le minimum de `S`, soit 2 et supprimons ceux de ses multiples stricts 4, 6, *etc* qui se trouvent dans S.
• Recommençons cette manipulation avec le nombre suivant dans `S` jusqu'à ce que l'on ne supprime plus rien. On ne retire plus rien de S quand le premier multiple de `p` que l'on pourrait supprimer, c'est à dire p^2 [8], n'est pas dans S, ce qui équivaut à $p^2 > n$.
• Les nombres restants dans S sont les nombres premiers recherchés parce qu'ils ne sont pas des multiples de nombres plus petits. Ils sont rangés dans l'ordre croissant.

```
# erato(n) retourne les nombres premiers <= n sous forme de matrice (n, )
# ainsi, optionnellement, que la durée b - a du calcul et
# le nombre de nombres premiers trouvés S.size.
import numpy as np
#from time import clock
def erato(n):# n est un entier >=2.
    #a = clock()
    S = np.arange(2,n+1)# S est une matrice (n, ).
    p,i = 2,0# 2 est le premier nombre premier, i est un compteur.
    while p**2 <= n:# La boucle commence à 2, premier nombre premier.
        T = S%p# Les restes des multiples de p dans la division par p
        # sont nuls.
        T[i] = p# En remplaçant 0 par p, on va récupérer p ensuite.
        masque = T > 0# Pour effacer les 0 (dans S, les multiples de p > p).
        S = S[np.where(masque)]# Effacement.
        i = i + 1
        p = S[i]# Nombre premier suivant.
    #b = clock()
    return S#, S.size#, b-a
```

Application à quelques valeurs de `n` : Ci-dessous, on a demandé à `erato(n)` de retourner les nombres premiers inférieurs ou égaux à `n` ainsi que la durée des calculs :

```
runfile('Chemin de la fonction erato()')
erato(2)
Out[2]: (array([2]), 1.4999999999432134e-05)
erato(3)
Out[3]: (array([2, 3]), 2.800000000036107e-05)
```

6. Le crible d'Érathostène est supposé connu.
7. Par exemple, 2 est un nombre premier. On considère parfois que 1 est aussi un nombre premier.
8. qui ne peut pas avoir été effacé précédemment

```
erato(5)
Out[4]: (array([2, 3, 5]), 5.8000000000113516e-05)
erato(10)
Out[5]: (array([2, 3, 5, 7]), 7.599999999996498e-05)
erato(100)
Out[6]:
(array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
        61, 67, 71, 73, 79, 83, 89, 97]), 0.00010900000000013677)
erato(1000)
Out[7]:
(array([  2,   3,   5,   7,  11,  13,  17,  19,  23,  29,  31,  37,  41,
         43,  47,  53,  59,  61,  67,  71,  73,  79,  83,  89,  97, 101,
        103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
        173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239,
        241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313,
        317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397,
        401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
        479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
        571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643,
        647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
        739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823,
        827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
        919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]),
 0.0004569999999999297)
erato(10000000)
Out[8]: (array([      2,       3,       5, ..., 9999971, 9999973, 9999991]),
7.436985)
erato(100000000)
Out[9]:
(array([       2,        3,        5, ..., 99999959, 99999971, 99999989]),
 153.03122199999999)
erato(500000000)
Out[10]:
(array([        2,         3,         5, ..., 499999909, 499999931,
        499999993]), 2068.7021259999997)
```

Ce dernier essai dure près de 35 minutes. En demandant en plus le nombre de nombres premiers inférieurs ou égaux à 500 000 000, on découvre qu'il y en a 26 355 867.

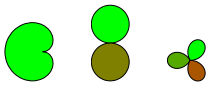
2 - Il suffit d'utiliser la fonction `erato()` ci-dessus : un entier `n` est premier si et seulement s'il figure dans la liste des entiers premiers inférieurs ou égaux à `n` ou si et seulement s'il est le maximum de cette liste.

```
from eratosthene import erato# Attention : on utilise erato(n)
# dans sa version qui retourne seulement la matrice à une dimension
# des nombres premiers <= n.
def japhet(n):
    return n == max(erato(n))
```

Application à quelques valeurs de `n` :

```
runfile('Chemin de la fonction japhet()')
Reloaded modules: eratosthene
japhet(5)
Out[2]: True
```

```
japhet(9999971)
Out[3]: True
japhet(123456789)
Out[4]: False
```



6.3 [***] Factoriser un entier en produit de nombres premiers

Programmer une fonction qui, étant donné un entier $n \geqslant 2$, retourne sa factorisation en produit de nombres premiers [a].

a. On peut utiliser la fonction `erato()` de l'exercice 6.2.

Mots-clefs : appel d'une fonction, boucles `pour` et `tant que`.

Solution

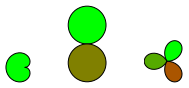
Étant donné un entier $n \geqslant 2$, `erato(n)` nous donne la matrice `(n, )` des nombres premiers $\leqslant n$. La fonction `beatris()` ci-dessous passe en revue ces nombres premiers et fait la liste de ceux qui divisent `n`. Un tel nombre premier apparaît dans la liste autant de fois que sa puissance la plus grande qui divise `n`.

```
from eratosthene import erato
def beatris(n):
    S = erato(n)
    L = []
    for i in range(S.size):
        while n%S[i] == 0:
            L.append(S[i])
            n = n//S[i]
    return L
```

Exemples :

```
runfile('Chemin de la fonction beatris')
beatris(24)
Out[6]: [2, 2, 2, 3]
beatris(79)
Out[7]: [79]
beatris(2018)
Out[8]: [2, 1009]
beatris(9999971)
Out[5]: [9999971]
beatris(123456789)
Out[3]: [3, 3, 3607, 3803]
3*3*3607*3803
Out[10]: 123456789
```

On voit donc, à la lecture de ces résultats, que $24 = 2^3 \cdot 3$, que 79 et 9 999 971 sont des nombres premiers et que $123\,456\,789 = 3^2 \cdot 3607 \cdot 3803$.



Bibliographie

[1] Alexandre Casamayou-Boucau, Pascal Chauvin, Guillaume Connan *Programmation en Python pour les mathématiques*, 2ème édition, Dunod Éditeur (2016).

Ce manuel est en fait indispensable pour un professeur de mathématiques qui débute en Python.

[2] Wikibooks *Mathématiques avec Python et Ruby.*
http://fr.wikibooks.org/wiki/Math%C3%A9matiques_avec_Python_et_Ruby OK

[3] Gérard Swinnen *Apprendre à programmer avec Python 3*, Eyrolles Éditeur, 3ème édition (2016).

Python est un langage de programmation général. Ce manuel ne vise pas particulièrement les professeurs de mathématiques.

Programmes de mathématiques au lycée

[4] *Seconde, programme de mathématiques*
http://cache.media.education.gouv.fr/file/30/52/3/programme_mathematiques_seconde_65523.pdf

[5] *Seconde, mathématiques, Ressources pour la classe de Seconde*
http://cache.media.eduscol.education.fr/file/Programmes/17/8/Doc_ress_algo_v25_109178.pdf

Documentation sur des modules de Python

Cours de Télécom ParisTech

Le cours de Télécom ParisTech, de Alexandre Gramfort & Slim Essid, est disponible aussi en version pdf. Il est intéressant :
http://www.prepas.org/2013/Info/Liesse/Telecom/

En détail :

[6] *Introduction à Python*
http://www.prepas.org/2013/Info/Liesse/Telecom/1-Intro-Python.html
http://www.prepas.org/2013/Info/Liesse/Telecom/1-Intro-Python.pdf

[7] *Numpy et matplotlib*
http://www.prepas.org/2013/Info/Liesse/Telecom/2-Numpy.html
http://www.prepas.org/2013/Info/Liesse/Telecom/2-Numpy.pdf

[8] *Librairie d'algorithmes pour le calcul scientifique en Python*
http://www.prepas.org/2013/Info/Liesse/Telecom/3-Scipy.html
http://www.prepas.org/2013/Info/Liesse/Telecom/3-Scipy.pdf

Autres sources de documentation sur les modules

[9] Python : The Standard Python Library
https://docs.python.org/py3k/library/index.html

Liste complète des modules courants de Python. Celle liste permet d'accéder à la documentation sur les fonctions pré-définies de Python via ses modules.

[10] *Built-in functions*
https://docs.python.org/3/library/functions.html
Documentation officielle en anglais sur les fonctions toujours disponibles de «Python».

[11] *math - Mathematical functions*
https://docs.python.org/3/library/math.html
Documentation officielle en anglais sur le module «math».

[12] *Numpy reference*
https://docs.scipy.org/doc/numpy/reference/index.html
Documentation officielle en anglais décrivant les fonctions, modules et objets de «Numpy».

[13] *Numpy.random reference*
https://docs.scipy.org/doc/numpy/reference/routines.random.html
Documentation officielle sur le module «numpy.random» (Random sampling) : génération de nombres aléatoires, tirages aléatoires, différents types courants de lois de probabilité.

[14] *random - Generate pseudo-random numbers*
https://docs.python.org/3/library/random.html
Documentation officielle en anglais sur le module «random».

[15] *Matplotlib User's Guide*
http://matplotlib.org/users/
Documentation en anglais sur le module «Mathplotlib». On préfèrera [16].

[16] *Tutoriel Matplotlib* sur le site «Developpez.com» :
http://python.developpez.com/tutoriels/graphique-2d/matplotlib/
Très bonne documentation en français sur le module «Mathplotlb».

Articles divers

[17] Jean-Marc Duquesnoy, Pierre Lapôtre, Raymond Moché *Algorithme de Bresenham 1 et 2* :
http://gradus-ad-mathematicam.fr/Premiere_Algorithmique2.htm

[18] Wikipedia *Carré magique (mathématiques)*
https://fr.wikipedia.org/wiki/Carré_magique_(mathématiques)

[19] Wikipedia *Crible d'Ératosthène*
https://fr.wikipedia.org/wiki/Crible_d%27%C3%89ratosth%C3%A8ne