Feuille de travaux pratiques - Python #1

Emeline LUIRARD

Préambule :

- Python est un logiciel libre qu'on peut télécharger sur https://www.python.org/. Nous travaillerons avec Python 3 et son environnement de travail Pyzo, qui est téléchargeable sur https://pyzo.org/. C'est ce que vous trouverez le jour de l'épreuve. Si ce n'est pas installé sur votre ordinateur en salle de TP, voici les commandes à saisir dans un terminal :

```
$ sudo apt-get update
$ sudo apt-get install python3.6
$ sudo apt-get install python3-pip python3-pyqt5
$ sudo python3 -m pip install pyzo --upgrade
```

  Ensuite, il suffit de lancer Pyzo.
- Python possède une aide consultable en tapant `help()` dans la console. Plus généralement pour obtenir une aide concernant une fonction `bidule` du module `machin`, tapez `help(machin.bidule)`. Vous pouvez aussi ouvrir la fenêtre de l'aide interactive, dans le menu Outils.
- Pour vous familiariser avec les fonctions de base de Python, vous trouverez de nombreux tutoriels sur internet, par exemple ici.

Vous n'aurez pas accès à internet le jour de l'épreuve. Apprenez donc, dès maintenant, à travailler avec les livres. Par exemple [Vig18], et [CBCC16]. Ce dernier est plus axé informatique et programmation.

1 Prise en main

1.1 Fenêtre de commande, éditeur, aide en ligne

Lorsqu'on lance le logiciel Pyzo, la fenêtre est découpée en divers cadres avec notamment :

- une fenêtre de commande (interpréteur) : où l'on peut exécuter des commandes : >>> indique que le logiciel attend vos instructions. L'interpréteur peut être utilisé comme une calculatrice : vous entrez un calcul, vous tapez sur la touche "entrée" et Python vous donne le résultat.
- une fenêtre d'éditeur de texte : où l'on peut regrouper les commandes et les sauvegarder dans un `fichier.py`.

On peut, dans la fenêtre de Python copier et coller des lignes à l'aide de la souris (y compris les exemples donnés par l'aide). On peut aussi naviguer dans l'historique des commandes au moyen des flèches haut et bas, cela évite de retaper les commandes si l'on exécute une fonction plusieurs fois de suite par exemple.

Python est un logiciel de calcul numérique et non de calcul formel : il effectue des calculs numériques approchés et non des calculs exacts. Par défaut, les réels sont affichés avec 15 chiffres significatifs.

```
-->pi
pi  =
3.141592653589793
```

N.B. *A la différence de Scilab, Python ne code pas tout en matrices. Il a ce qu'on appelle une programmation orientée objet.*

1.2 Programmation

Dès que l'on souhaite exécuter plus de deux ou trois tâches consécutives, i.e. dès que l'on sort du mode d'utilisation "supercalculatrice" de Python, on utilise l'éditeur de texte. On enregistre le fichier obtenu avec le suffixe `.py` puis on exécute le fichier dans Pyzo, au moyen du menu `Exécuter...` ou par un raccourci clavier. Pour exécuter seulement un bout de code, vous pouvez le séléctionner et cliquer sur `Exécuter la sélection`.

- Lors de l'oral vous **devez** utiliser de tels fichiers.
- Pour faciliter la mise au point et la relecture du programme par vous-même et sa compréhension par le jury, vous **devez également le commenter**.

Une ligne de commentaires commence par `#`. De manière générale, Python ignore tout ce qui, sur une ligne, suit la commande `#`.

```
Debut programme
[...]
# on definit maintenant les abscisses et ordonnées
x=np.arange(10) ; # le vecteur des abscisses  entier de 0 à 9
y=np.sin(x) ; # le vecteur des ordonnées
[...]
Fin du programme
```

Si vous voulez afficher une variable, vous devrez utiliser `print(variable)`.

1.3 Importation de données

> "Le jury rappelle que de nombreux textes sont assortis d'un jeu de données numériques sur lesquelles les candidats sont invités à mettre en œuvre des illustrations. Le jury se réjouit de l'augmentation du nombre de candidats traitant effectivement ces données. Cela constitue une réelle plus-value pour l'exposé. Signalons que ces textes sont accompagnés d'une notice expliquant comment ouvrir les fichiers de données avec les logiciels Python, Scilab, Octave et R, logiciels que le jury estime les mieux adaptés à l'option A."

Les commandes de base sont les suivantes :


Sur Windows, le chemin vers le fichier s'écrit avec des \.

```
import csv
with open('/home/dossier/blabla.csv', newline='') as f:
        lignes=[ligne for ligne in csv.reader(f)]
```

`lignes` contient une liste avec les lignes du fichier, sous forme de chaînes de caractères.

2 Syntaxe de base

2.1 Modules

```
from nom_du_module1 import *
#importe tous les élémens du module1
```

ou

```
import module2 as mod2


from module3 import sous_module as smod
```

Pour la deuxième méthode, les fonctions du module2 devront être appelées par `mod2.fonction()`. On l'utilise pour ne pas qu'il y ait d'écrasement d'autres fonctions, pour ne pas encombrer l'espace des noms. Pour les sous modules, on fera `smod.fonction()`.
Voici les principaux modules que nous utiliserons :

- `math` : pour importer les fonctions mathématiques usuelles et certaines constantes usuelles comme π (`pi`).
- `cmath` : pour les complexes : (`1j` pour i).
- `numpy` : pour utiliser le type `array` (tableau dont les éléments sont tous du même type, pratique pour les vecteurs, les matrices). On conseille `import numpy as np`.
- `scipy` : outils nécessaires au calcul matriciel. Il contient un sous-module qui nous servira pour la partie aléatoire.
- `matplotlib` : pour générer des graphiques.

`numpy, scipy` et `matplotlib` sont (peut être) à installer (ils seront installés le jour de l'agreg) à l'aide de la commande `python3 -m pip install -user scipy` dans le terminal linux.

Souvenez vous du nom de ces modules, vous les importez tous au début de votre code, et vous serez tranquilles. Seuls ces modules seront disponibles le jour de l'agreg.

2.2 Opérations usuelles

`x+=1`	remplace x par $x+1$ (valable aussi pour -,*,/)
`x**a`	x^a
`x,y=17,42`	pour affecter deux variables en même temps, toutes les expressions sont évaluées *avant* la première affectation, ex : `x,y=x+2,x`
`x,y=y,x`	du même type que le précédent, pour échanger deux variables

2.3 Fonctions

Elles seront très utiles pour itérer un bout de code en faisant varier des paramètres. La syntaxe est la suivante

```
def nom_de_la_fonction(parametres):
        """ Details de ce que fait la fonction"""
        calculs
        return valeur_sortie
```

Dès que Python trouve un `return truc`, il renvoie `truc` et abandonne ensuite l'exécution de la fonction.

Les variables définies à l'intérieur d'une fonction sont locales : une fois le code de la fonction effectuée, elles n'existent plus. De manière générale, on évitera de donner des noms de variables identiques aux variables locales et aux variables globales.

On donnera des noms longs et explicites aux variables globales.

N.B. *Vous pouvez mettre des paramètres pas défaut, un nombre de paramètres arbitraire, etc. On s'en sortira sans mais pour ceux que ça intéresse, voir [CBCC16, p. 42] et/ou [Vig18, p. 14].*

2.4 Opérateurs logiques et booléens

Les opérations logiques de bases sont le "et" logique (`and`), le "ou" logique (`or`) et la négation (`not`). Les valeurs logiques sont `True` et `False`. Voici divers opérateurs de comparaison :

`x==y`	x est égal à y
`x!=y`	x est différent de y
`x<=y`	$x \leq y$
`x in y`	x appartient à y (pour les listes)

Pour demander des conditions plus complexes, on peut combiner ces opérateurs de comparaison avec les booléens, par exemple, `(x==y) or (x>5)`.

2.5 Boucles

Même si on cherche en général à l'éviter pour la rapidité d'exécution des programmes, l'usage des boucles est parfois inévitable. Les syntaxes des boucles `for`, `while` et des instructions conditionnelles sont les suivantes :

```
i, aux = 0,2
while (i<5)and(aux<10):
        aux=aux**2
        i+=1
```

```
n=5
for i in range(10):
        a[i]=1/(i+1)
        a[i]=1/(i+1)
```

```
if i == j:
        a[i,j] = 2
else:
        a[i,j] = 0
```

Vérifiez impérativement que vos boucles `while` vont s'arrêter !

On veillera à bien indenter les codes dans l'éditeur : en Python, il n'y a pas de `end`, c'est l'indentation qui indique lorsque la boucle est fermée.

N.B. *Lorsqu'il y a plus de deux cas, on utilisera `elif` :*

```
if i == j:
        a[i,j] = 2
elif i == j+1:
        a[i,j] = 0
else:
        a[i,j] = 1
```

2.6 Listes

Une liste est une séquence d'éléments rangés dans un certain ordre, elle peut contenir des objets de types différents, par exemple, `liste=[42,'reponse']`, et sa taille peut varier au cours du temps.


En Python, on commence à compter à 0.

Pour créer une liste, connaître sa longueur et accéder à ses éléments :

`liste=[]`	crée une liste vide
`len(liste)`	renvoie la longueur de la liste
`liste[i]`	extrait l'élément à l'indice i de la liste (en comptant à partir de 0)
`liste[-1]`	extrait le dernier élément
`liste[2:4]`	extrait une sous liste

"Il faut penser les indices comme repérant non pas les tranches de saucisson, mais les coups d'opinel qui ont permis de couper les tranches."

Les "intervalles" dans Python sont fermés à gauche et ouverts à droite : `liste[2:4]` extrait les éléments de l'indice 2 à 3 de la liste.
Pour créer des listes, Python est sympa, grâce aux listes en compréhension :
`[x**2 for x in range(10)]` retourne la liste `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`.
Enfin, vous pouvez transformer une liste en ensemble (i.e. pas de doublons) par `set(liste)`.

Modification de listes :

`ma_liste[1]=4`	remplace l'élèment à l'indice 1 par 4
`liste1+liste2`	concatène deux listes
`ma_liste.append(5)`	ajoute un élément à la fin de la liste
`ma_liste.extend(L)`	ajoute en fin de liste les éléments de L
`ma_liste.insert(i,x)`	insère un élément x en position i
`ma_liste.remove(x)`	supprime la première occurence de x
`ma_liste.sort()`	trie la liste
...	

Taper `help(list)` pour obtenir toutes les "méthodes" associées. (Une méthode est une fonction qui ne s'applique qu'à un type d'objet, ici les `list`.)


On utilise assez souvent la fonction `range(start, stop[, step])` qui renvoie les entiers de start à stop. Attention, cependant, ce n'est pas une liste, c'est ce qu'on appelle un "itérable" : un curseur qui se déplace, tous ses entiers ne sont pas stockés. Si on veut une liste, on pourra utiliser `list(range(...))` ou `[range(...)]`.


Attention quand vous copiez des listes :

```
>>> liste=[42, 'reponse']
>>> copie=liste
```

```
>>> liste[1]=17
>>> liste, copie
[42,17], [42,17]
>>> copie[0]=0
>>> liste, copie
[0,17], [0,17]
```

Les variables `liste` et `copie` pointent vers le même objet. On dit qu'elles font "référence" au même objet. Pour faire une copie, on peut utiliser `copie=liste[:]`, mais attention si la liste contient des sous-listes !

2.7 Array : vecteurs, matrices

Un objet de type `array` est un tableau dont tous les éléments sont du même type. La différence avec les `list` : sa taille n'est pas modifiable une fois créé. Il servira à représenter les vecteurs et les matrices. Il est dans le module `numpy`. Dans la suite, je noterai M pour une matrice, V pour un vecteur, A pour un array quelconque.

```
M=np.array([[0,1],[2,3]]) # matrice
V=np.array([0,1,4,9])# vecteur ligne
```

Dans le module `scipy`, vous trouverez le sous-module `linalg` permettant de faire du calcul matriciel. Voici quelques commandes utiles pour manipuler les `array` :

Pour créer un `array`, connaître sa longueur et accéder à ses éléments :

`np.eye(n)`	matrice identité de taille n
`np.ones((n,m))`	matrice 1 de taille $n \times m$
`np.zeros((n,m))`	matrice nulle de taille $n \times m$
`np.empty((n,m))`	matrice vide de taille $n \times m$
`np.arange(debut, fin(non inclus), pas)`	le tableau des entiers de debut à fin avec un pas
`np.linspace(min, max, nb de points)`	points réguliers $[min, ..., max]$
`np.diag(V)`	matrice avec en diagonale V
`V[-1]`	dernier élément du vecteur V
`V[start:end:step]`	pour extraire une sous-liste
`M[i,j]`	élément à l'indice $(i_{ligne}, j_{colonne})$
`A.shape`	renvoie (nb de lignes, nb de colonnes)
`len(V)`	renvoie la longueur du vecteur
`A.reshape([n,m])`	redimensionne le tableau A en n lignes et m colonnes. Si l'une des valeurs vaut -1, `numpy` la trouve tout seul.
`V.reshape([nbLign,1])` ou `V[:,np.newaxis]`	transforme un vecteur ligne en colonne

Modification et opérations sur les listes :

Faire des opérations sur des `array` de tailles différentes, s'appelle du broadcasting. Cela fait les opérations terme à terme :

`A+10`	ajoute terme à terme
`M+V`	ajoute v à chaque ligne de M
`A*A`	multiplication terme à terme
`V*M`	répète v selon l'autre dimension pour avoir la même taille que M avant de multiplier terme à terme
`1/A`	inverse terme à terme

N.B. *De manière générale, quand vous n'êtes pas sûr du résultat que donne un code (ex : opération terme à terme ou non), testez sur un exemple.*

Les opérations usuelles sont les suivantes :

`A.dot(B)`	multiplication matricielle : matrice/matrice, matrice/vecteur, vecteur/vecteur (produit scalaire)
`np.linalg.inv(M)`	inverse matriciel
`M.T`	transposée
`np.linalg.det(M)`	déterminant
`V[0:3]=5`	remplace les 3 premiers éléments de V par des 5.
`A>0`	renvoie un `array` de même taille que A avec des booléens.
`A[A>0]`	renvoie un `array` avec les éléments de A qui vérifient la condition Ici ça fait une copie et non une référence. Pour les conditions, on utilise `&` pour "et" et `|` pour "ou"

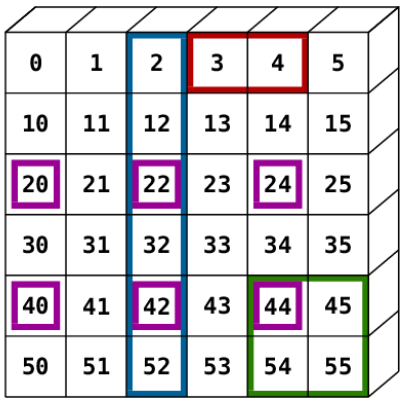
`np.sum(A)`
`np.sum(A,axis=0)`
`np.sum(A,axis=1)`
	somme des éléments de A
	pour sommer selon les colonnes,
	selon les lignes

`np.cumsum(A)`	somme cumulative des éléments de A, on peut encore utiliser `axis`
`np.mean(A)`	moyenne des éléments de A, on peut encore utiliser `axis`
`np.std(A)`	écart-type des éléments de A, on peut encore utiliser `axis`

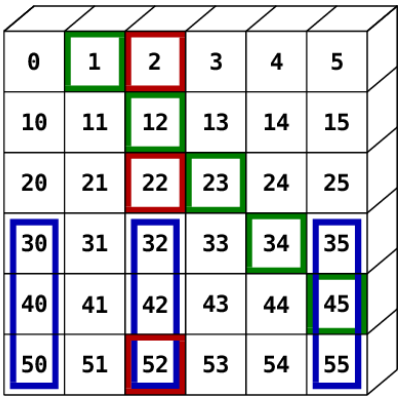
`eigVal,eigVec=np.linalg.eig(M)`
 	
`eigVal,eigVec=np.linalg.eigh(M)`
	valeurs propres (avec multiplicité) et vecteurs propres à droite, en colonne, normalisés et dans le même ordre.
	pour les matrices symétriques, hermitiennes

Si vous cherchez quelque chose en particulier vous pouvez aussi utiliser `np.lookfor('create array')`.
Deux images assez explicites, pour l'extraction de listes, tirées de https://scipy-lectures.org :

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



Pour un gain de temps, il faut, chaque fois qu'on le peut, utiliser les opérations matricielles prédéfinies.

2.8 Tracer des graphes

Pour tracer des graphes, nous avons besoin du sous module `pyplot` de `matplotlib`. Ensuite, on définit la figure, qu'on appelle en général `fig`. Puis dans la figure, on définit les sous-figures, selon les axes. On va utiliser la fonction `plot` qui prend en arguments des `array`.

```
fig, axs=plt.subplots(n,m) # on aura des sous-figures sur n lignes et m colonnes
#sans argument quand on n'en veut qu'un.
axs[i,j].plot(x,y,"r", label="nom fonction") # trace dans le graphique (i,j),
#en couleur rouge, et associe une legende
fig.suptitle("titre principal")
axs[0,0].set_title("titre graphe 1")
axs[0,1].set_title(r"$math latex$") # le 'r' dit à Python de ne pas interpreter
#les caracteres speciaux
#Ne pas oublier:
axs.legend() # affiche les légendes
plt.show() # pour ouvrir la fenetre graphique
plt.clf() #efface la fenêtre graphique
```

N.B. *Le troisième argument de* `plot` *est le style du trait. On mettra "o" pour avoir des points non reliés.*

D'autres commandes :

`axs[..].set_ylabel("titre ordonnees")`	donne un titre aux ordonnées
`axs[..].set_ylim([a,b])`	limite le graphe aux ordonnées $[a, b]$
`ax[..].set_xticks(rray)`	graduation axe des abscisses
`ax[..].set_sticklabels(["0", "1",...])`	noms des graduations de l'axe des abscisses

Diagramme en bâtons : histogrammes de lois discrètes

`ax.bar(x,y,width=...)` à la place de `plot`.

Histogramme

`plt.hist(X,bins=n)` répartit les données de X dans n sous intervalles de $[min(X), max(X)]$. `hist` prend la place de `plot`, on garde la même syntaxe que précédemment pour faire différents graphes par exemple. Mais pour mettre plusieurs histogrammes sur le même on fait `plt.hist([X1,X2,X3], bins=n, label=...)`.

N.B. *Avec `a=plt.hist(X,bins=n)`, on récupère en premier la hauteur des bâtons et en deuxième le découpage des sous-intervalles fait par Python.*

N.B. *On peut donner à `bins` ses propres sous-intervalles dans un `array`.*

Pour superposer avec une densité, il faut demander à Python de normaliser les histogrammes, ça se fait avec l'option de `hist` : `density=True`. Puis on plote la densité.

2.9 Aléatoire

`Scipy` contient un sous-module `stats`.
On conseille de l'importer sous le format `import scipy.stats as stats`
À chaque loi de probabilité sont associées des commandes :

- `rvs` : simulation
- `pdf(x, parametre)` : probability density function i.e. densité en x
- `pmf(k, parametre)` : probability mass function i.e. densité discrête en k
- `cdf(x,parametre)` : cumulative density function i.e. fonction de répartition en x
- `ppf(q, parametre)` : percent point function i.e. q-ième quantile.

On les utilisera comme ceci : `stats.norm.rvs(parametres)`. Les paramètres sont (paramètres de forme, loc=μ, scale=σ, size=taille échantillon). Pour créer un vecteur aléatoire (`array`) de taille $n \times p$, on donnera l'argument `size=(n,p)`. Les paramètres μ et σ sont tels que si l'appel de `stats.xxx.rvs()` renvoie une v.a X alors `stats.xxx.rvs(loc=mu,scale=sigma)` renvoie une v.a ayant la même loi que $\sigma X + \mu$. En tapant `stats.expon?` ou `help(stats.expon)`, vous aurez la normalisation choisie pour la loi exponentielle par exemple. Seuls les paramètres de forme sont obligatoires.

loi(paramètres de forme)	Lois
`norm()`	Loi normale $\mathcal{N}(0,1)$.
`gamma(a=)`	Loi gamma $\gamma(a,1)$.
`beta(a=, b=)`	Loi béta $\beta(a,b)$.
`t(df=n)`	Loi Student de n degrés de liberté.
`cauchy()`	Loi de Cauchy.
`expon(scale=1/lambda)`	Loi exponentielle $\mathcal{E}(\lambda)$.
`uniform(loc=, scale=)`	Loi uniforme $\mathcal{U}(]loc, loc+scale[)$.
`randint(low, high)`	Loi uniforme $\mathcal{U}(\{low, low+1, \cdots, high-1\})$.
`binom(n=, p=)`	Loi binomiale $\mathcal{B}(n,p)$.
`geom(p=)`	Loi géométrique $\mathcal{G}(p)$.
`poisson(mu=)`	Loi de Poisson $\mathcal{P}(\mu)$.

N.B. *Toutes ces lois, ainsi que les relations entre elles sont rappelées dans [Vig18][p. 70].*

3 Exercices

À chaque fois, on cherchera à coder de façon "économique".

Votre devise doit être : coder compact et simple ! C'est comme cela que vos codes seront les plus lisibles et que vous ferez le moins d'erreurs.

Exercice 1. [A faire une fois que vous avez compris les `array` et les tracés de graphes.]
Télécharger le fichier `donnees.csv` dans votre espace de travail. Importer le tableau dans Python, et pour chaque feuille, tracer la seconde ligne en fonction de la première.
Aide : On pourra utiliser la fonction `a=a.astype(np.float64)` qui change le type des éléments de a en flottants.

Exercice 2. Créer une liste avec plein d'entiers différents. Trouver le nombre de 1. Trouver la première fois que 2 apparaît (si c'est le cas). Insérer `'ici'` juste avant le fameux 2.
Aide : Utiliser `help(list)` pour avoir les méthodes s'appliquant aux listes. Si x est un `array`, `list(x)` renvoie une liste.

Exercice 3. Ecrire la matrice $(i-j)_{i\in[\![1,3]\!],j\in[\![1,4]\!]}$. Afficher sa première ligne, les deux premières, sa première colonne, la dernière colonne, l'élément en bas à droite, la sous-matrice contenant les deux premières lignes et deux dernières colonnes .

Exercice 4. Simuler 1000 nombres aléatoires entre 0 et 50. Créer un tableau contenant tous ceux qui sont divisibles par 2 ou 5.
Aide : `a%b` retourne le reste de la division euclidienne de a par b. On peut passer d'une liste à un array par `np.array(ma_liste)`.

Un peu de manipulation des matrices :

Exercice 5. Écrire un vecteur aléatoire de 10 réels. Réécrire ce vecteur dans l'ordre inverse.

Exercice 6. Écrire un vecteur de taille 10 dont les éléments sont successivement $+1$ et -1.

Exercice 7. Construire les matrices suivantes en utilisant le moins d'opérations possibles.

$$A=\begin{pmatrix}2&0&1\\0&2&0\\1&0&2\end{pmatrix},\quad B=\begin{pmatrix}1&1&1\\2&1&2\\1&1&1\end{pmatrix},\quad C=\begin{pmatrix}1&1&1\\1&2&1\\1&1&3\end{pmatrix}.$$

Exercice 8. Écrire (le plus compact et simple possible) les matrices carrées d'ordre 6 suivantes :

1. Matrice diagonale, dont la diagonale contient les entiers de 1 à 6.
2. Matrice contenant les entiers de 1 à 36, rangés par lignes.
3. Matrice dont toutes les lignes sont égales au vecteur des entiers de 1 à 6.
4. Matrice diagonale par blocs, contenant un bloc d'ordre 2 et un d'ordre 4. Les 4 coefficients du premier bloc sont égaux à 2. Le deuxième bloc contient les entiers de 1 à 16 rangés sur 4 colonnes.
   Aide : `scipy.linalg` contient une fonction pour faire des matrices diagonales par blocs.
5. Matrice $A=(a_{i,j})$ dont les coefficients sont $a_{i,j}=(-1)^{i+j}$.
6. Matrice contenant des 1 sur la diagonale, des 2 au-dessus et au-dessous, puis des 3, etc., jusqu'aux coefficients d'indices (1,6) et (6,1) qui valent 6.

Exercice 9. Écrire la matrice $A = (a_{i,j})$ d'ordre 12 contenant les entiers de 1 à 144, rangés par lignes. Extraire de cette matrice les matrices suivantes :

1. Coefficients $a_{i,j}$ pour i de 1 à 6 et j de 7 à 12,
2. Coefficients $a_{i,j}$ pour $i + j$ pair,
3. Coefficients $a_{i,j}$ pour $i, j = 1, 2, 5, 6, 9, 10$.

Exercice 10. Écrire une fonction `dif` réciproque de la fonction `cumsum` agissant sur les vecteurs.

Exercice 11. Écrire la matrice Zorro de taille n, i.e. la matrice carrée de taille n dont tous les coefficients sont nuls sauf la première ligne, la dernière ligne et l'anti-diagonale qui valent 1.

Exercice 12. Étant donne un triangle $A_1A_2A_3$ et un point M dans le plan, repérés par leurs coordonnées cartésiennes, écrire une fonction qui indique si M est à l'interieur du triangle ou non.

Exercice 13. Écrire une fonction renvoyant les $n + 1$ premières lignes du triangle de Pascal (complétées par des 0 pour apparaître sous forme de matrice carrée).

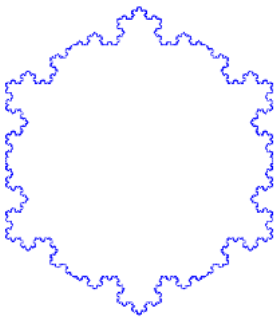
Exercice 14. Écrire une fonction renvoyant la liste de tous les nombres premiers inférieurs ou égaux à son argument en utilisant la méthode du crible d'Eratosthène. Représenter graphiquement la fonction de comptage des nombres premiers, i.e. la fonction qui à $x \in \mathbb{R}^+$ associe le nombre de nombres premiers inférieurs ou égaux à x.

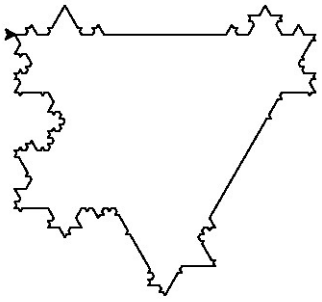
Exercice 15. Définir une fonction qui prend un vecteur $x = (x_1, \ldots, x_n)$ en entrée et retourne la matrice de Vandermonde ci-dessous ainsi que son déterminant.

$$\begin{pmatrix} 1 & 1 & \ldots & 1 \\ x_1 & x_2 & \ldots & x_n \\ x_1^2 & x_2^2 & \ldots & x_n^2 \\ \vdots & \vdots & & \vdots \\ x_1^n & x_2^n & \ldots & x_n^n \end{pmatrix}$$

Pour aller plus loin...

Exercice 16. Écrire une fonction qui, étant donné un entier n, trace le n-ième itéré du flocon de Van Koch. Voici par exemple ce que l'on obtient pour $n = 7$, lorsque l'on part d'un hexagone régulier.
Aide : On pourra utiliser le module `turtle`.





Randoniser ensuite votre algorithme pour qu'à chaque itération, seul un nombre aléatoire (à vous de choisir la loi) de triangles "ne fleurissent" sur les bords du flocon.

Références

[CBCC16] Alexandre Casamayou-Boucau, Pascal Chauvin, and Guillaume Connan. *Programmation en Python pour les mathématiques - 2e éd.* Dunod, Paris, 2e édition edition, January 2016.

[Vig18] Vincent Vigon. *python proba stat.* Independently published, October 2018.