MÉTHODOLOGIE II

(RÉ)-INTRODUCTION À PYTHON POUR LES ÉTUDIANTS PRESSÉS.

Le but de ce texte est de réviser le fonctionnement, la syntaxe et les commandes de base du langage Python. Le travail est à faire avec un ordinateur capable d'interpréter les commandes Python. Pour cela, on peut par exemple télécharger et installer (gratuitement) Spyder (https://www.spyder-ide.org/)

Boucles for. Une boucle sert à donner une série d'instructions en une seule commande, c'est l'outil de base de l'algorithmique. Il en existe de deux types : les boucles for et les boucles while. Les boucles for effectuent un nombre d'opérations donné à l'avance par l'utilisateur, tandis que les boucles while effectuent un nombre d'opération variables jusqu'à temps qu'une certaine condition renseignée par l'utilisateur soit vérifiée.

La boucle for se construit de la manière suivante

```
for i in range(...) :
    # Instruction 1.
    # Instruction 2.
    ...
```

Plusieurs remarques s'imposent

1. Python est un langage *à indentation* : l'espace entre le début de la ligne et le début de l'instruction est interprété par Python. Pour indiquer que les instructions font partie de la même boucle for, on les écrit sous la déclaration de la boucle et avec une indentation.
2. Ne pas oublier les : à la fin de la la ligne for i in range (...) :
3. La fonction range sert à indiquer le nombre de fois que Python doit effectuer les instructions données dans la boucle. Ainsi

   ```
   for i in range (4)
   ```

   indique à Python que l'on veut effectuer les instructions 4 fois. Attention, la commande for i in range(4) donne à i les valeurs successives $i = 0, i = 1, i = 2$ et $i = 3$. La commande i in range(4) est donc interprétée par
   $$i \in [\![0, 4[\![.$$
4. Les instructions dans la boucle peuvent dépendre ou ne pas dépendre de i : si elles n'en dépendent pas, toutes les opérations effectuées dans chaque tour de boucle seront identiques et il y en a autant que le nombres d'entiers dans le range
5. Il existe des options dans la fonction range pour changer l'intervalle dans lequel se trouvent les indices i, faire avancer i de 2 en 2, à reculons, ...
6. La commande # sert à faire des commentaires : tout ce qui figure après le symbole # est ignoré par Python. Les accents rendent mal dans les lignes de commentaires.

Exercice 1.-
On considère le code suivant

```
S=0
for i in range(10):
  S=S+i
print(S)
```

La commande print(S) sert à afficher S.

1. Analyser le programme pour comprendre ce qu'il retourne.
2. Faire tourner ce programme dans un ordinateur, le résultat obtenu est-il conforme à ce qu'on attendait.
3. Que se passe-t-il si on change le 10 en un autre entier ? Tester sur le programme.

Exercice 2.-
Écrire un programme qui calcule la factorielle d'un nombre entier.

Boucles while. Contrairement à la boucle for, la boucle while ne s'arrête pas au bout d'un nombre d'instructions connu à l'avance, mais plutôt lorsqu'une certaine condition se produit. La syntaxe générale est la suivante

```
while  condition(s) :
  instruction(s)
```

Par exemple, considérons la suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$u_n = \frac{n}{\ln(3)} + \sqrt{2}.$$

C'est bien sûr une suite croissante qui diverge vers $+\infty$. Cherchons alors le terme de la suite à partir duquel $u_n \geqslant 100$.

```
import numpy as np

n=0
while (n/(np.log(3))+2**(1/2) < 100 ) :
  n=n+1
print(n)
```

Quelques remarques :

1. La toute première ligne sera commentée plus tard : elle ne sert qu'à dire à Python ce qu'est la fonction ln.
2. Les symboles pour les opérations élémentaires de l'algèbre sont + pour l'addition, - pour la soustraction, * pour la multiplication, / pour la division, ** pour la puissance.
3. En langage courant ce programme se lit de la manière suivante : "tant que u_n est plus petit que 100, on augmente n de 1. On s'arrête dès que u_n est supérieur ou égal à 100. Et on affiche le terme de la suite (n)".

Exercice 3.-
Utiliser un ordinateur pour savoir quel est l'entier n à partir duquel n est plus grand que 100.

Instructions conditionnelles. Dans la boucle while précédente, il a fallu indiquer à Python comment s'arrêter, en lui demandant de comparer les deux nombres $\frac{n}{\ln(3)} + \sqrt{2}$ et 100 pour savoir lequel des deux était plus grand. On peut aussi formuler une condition "inférieur ou égal", "strictement plus grand", "différent de". On peut aussi donner comme critère d'arrêt à la boucle while une condition qui dit que **deux** contraintes sont réalisées simultanément, ou bien encore soit une des deux contraintes est réalisée, soit l'autre.

Il faut donc savoir formuler les opérations logiques ET, OU et NON.

Gardons par exemple la même suite $(u_n)_{n\in\mathbb{N}}$ et demandons à Python d'afficher tous les indices n pour lequel $u_n \in [100, 200]$:

```
import numpy as np

n=0
while (n/(np.log(3))+2**(1/2) < 100 ) :
  n=n+1
while (n/(np.log(3))+2**(1/2) <= 200 ) and  (n/(np.log(3))+2**(1/2) => 100 ):
  n=n+1
  print(n)
```

Dans la première boucle, on cherche comme précédemment le premier entier à partir duquel $u_n \geqslant 100$. On garde ensuite cette valeur de n pour démarrer la deuxième boucle. Dans cette seconde boucle, on a formulé une condition d'arrêt équivalente à

$$u_n \in [100, 200]$$

et on demande d'afficher tous les indices de la suite pour lesquels cette condition est réalisée. Remarquer qu'ici la commande print(n) est dans la boucle de sorte que chaque tour de boucle comporte une commande d'affichage, c'est ce qui permet d'afficher *tous* les indices satisfaisants et pas seulement le dernier.

Il faut donc retenir la syntaxe suivante

- and sert à fabriquer un **et** logique (il faut que les **deux** conditions soient réalisées).
- or sert à fabriquer un **ou** logique (il faut que **l'une ou l'autre** des conditions soit réalisée).
- not sert à fabriquer un **non** logique (il faut que la condition **ne soit pas** réalisée).

Définir une fonction. On veut maintenant définir des fonctions, qui pourront ensuite être appelées pour réaliser une certaine action. C'est la deuxième activité fondamentale de l'algorithmique. Regardons par exemple le code

```
def fonction(x) :
  return x**2+x+1
```

Quelques remarques

1. Ce code Python n'affiche rien, et c'est normal. Pour le moment, on ne demande à Python que de se souvenir de la définition de la fonction, et on ne lui demande donc aucun calcul.
2. La même syntaxe que dans les boucles est utilisée : les : à la fin de la déclaration de la fonction et l'indentation dans la ligne qui suit pour indiquer à Python qu'on est en train de définir la fonction. La définition dure tant que le début de la ligne est décalé.
3. La commande return sert à faire un calcul sans l'afficher. Ainsi, si plus tard dans le programme, on donne une valeur et x et on demande à Python de faire fonction(x) alors le programme va faire le calcul de la valeur de fonction(x) mais ne va pas l'afficher. Sur un ordinateur, vérifier que les deux programmes

```
def fonction(x) :
  return x**2+x+1

x=4
fonction(4)
```

et

```
def fonction(x) :
  return x**2+x+1

x=4
print(fonction(4))
```

renvoient bien ce qu'on en attend (le premier n'affiche rien, tandis que le second affiche la valeur de fonction(4)).

Exercice 4.-
À l'aide de l'exercice **2.**, écrire maintenant *une fonction* factorielle qui calcule et affiche la valeur de $n!$ où n est en entier qui doit être rentré par l'utilisateur.

Bibliothèques. Revenons maintenant sur la ligne

```
import numpy as np
```

du programme précédent. Ici, on a demandé à Python d'utiliser une bibliothèque de fonction prédéfinies. Python est un langage très riche qui offre énormément de possibilités. Aucun des utilisateurs de Python n'a besoin de toutes les fonctionnalités disponibles et les charger dans la machine serait trop coûteux en ressources matérielles. C'est pourquoi, on n'y fait appel que si besoin et on ne les charge dans la RAM de l'ordinateur que si on est sûr de les utiliser.

Dans notre programme d'ECG, nous utiliserons certaines de ces bibliothèques, en particulier, celles dans lesquelles sont définies les fonctions mathématiques (ln, exp, partie entière, ...), celles qui permettent de simuler des expériences aléatoires, celles qui permettent de dessiner des graphes de fonctions ou des histogrammes de séries statistiques et celles qui permettent de faire du calcul matriciel.

Regardons dans chacun des cas ce qu'il faut déclarer.

1. La bibliothèque math contient toutes les fonctions mathématiques utilisées dans notre cours. La bibliothèque numpy aussi mais elle est beaucoup plus grosse et peut mettre du temps à se charger sur les ordinateurs les moins performants.
2. Pour simuler des variables aléatoires, on utilise la bibliothèque numpy.random (c'est en fait une sous-bibliothèque de la bibliothèque numpy et elle sera chargée automatiquement si on charge numpy).
3. Pour les différents dessins, on utilisera la bibliothèque matplotlib.pyplot.
4. Pour l'algèbre, on utilisera la bibliothèque numpy.linalg (c'est aussi une sous-bibliothèque de numpy).
5. Enfin, pour les statistiques, nous utiliserons la bibliothèque pandas.

Les deux bibliothèques de probabilités et statistiques ne seront pas présentées dans ce texte et on renvoie au deux TP sur le sujet (TP1 pour les probas, TP2 pour les stats).

Les fonctions préenregistrées dans les bibliothèques doivent ensuite être appelées en indiquant dans quelle bibliothèque elles se trouvent. Par exemple, pour calculer le logarithme népérien de 3, il faut utiliser la fonction log de la bibliothèque math

```
import math

math.log(3)
```

C'est en général un peu lourd à taper et on préfère souvent renommer une bibliothèque d'usage fréquent pour en appeler plus facilement les fonctions. On dit qu'on utilise alors un alias.

Ainsi dans la ligne

```
import numpy as np
```

on a déclaré à Python que l'on voulait utiliser la bibliothèque numpy. Mais, dans la suite du programme, pour simplifier, cette bibliothèque sera appelée np. Ainsi

```
numpy.log(3)
```

s'écrira (légèrement) plus simplement

```
np.log(3)
```

Enfin, si on veut être très économe, on peut ne choisir d'importer d'une bibliothèque que les seules fonctions qui nous intéressent. Si on n'a aucun usage des autres fonctions de la bibliothèque math, on peut ne charger que la fonction ln. On demande alors d'utiliser cette fonction en écrivant

```
from math import log
```

Dans ce cas-là, il n'est plus nécessaire d'indiquer la bibliothèque dont provient la fonction et la commande

```
from math import log

print(log(3))
```

suffit à afficher ln(3), au lieu de print(math.log(3)).

Représentations graphiques. À chaque fois qu'un programme se propose de dessiner un graphe de fonctions ou un histogramme de valeurs, il faut importer la bonne bibliothèque

```
import matplotlib.pyplot as plt
```

(renommée en plt pour simplifier les commandes).

On utilisera alors la méthode de représentation graphique suivante :

```
import matplotlib.pyplot as plt

plt.plot(ce qu il faut dessiner)

options de mises en forme

plt.show() # pour afficher le dessin
```

La fonction plot s'attend à recevoir une série de valeurs pour x et une série de valeurs pour y (sous forme de listes, voir le paragraphe suivant), plus éventuellement des options (couleur, forme du graphe). Par défaut, cette fonction produit un graphe de fonction continue, c'est-à-dire qu'elle va relier les différents points de coordonnées (x, y).

Les autres options qui ne rentrent pas dans la fonction plot servent par exemple à donner un titre au graphe, à nommer les axes, à insérer une légende,...

Exercice 5.- **1.** Recopier et faire tourner le code suivant

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4],[1, 4, 9, 16])

plt.show()
```

Quelles sont les échelles des deux axes, quel commentaire cela vous inspire-t-il ?

2. Comparer le résultat précédent avec le résultat issu de ce programme :

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4],[1, 4, 9, 16],'ro')

plt.show()
```

Ici on a ajouté une option de couleur pour le graphe et une option de forme. Le rouge est donné par le r et les points ronds isolés par la commande o. Essayer de changer ces options pour voir ce qu'il se produit.

3. On peut en fait donner à la fonction plot plusieurs ensembles de nombres à représenter simultanément. Essayer

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0., 5., 0.2)

```

```
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

Commenter. À votre avis, quel est le rôle de la fonction arange de numpy ? La fonction np.linspace est d'usage identique.

4. Proposer une méthode pour dessiner le graphe d'une fonction.

Listes et matrices. Le langage Python est aussi utilisé pour faire du calcul matriciel. Toutes les fonctions nécessaires se trouvent dans les bibliothèques numpy et numpy.linalg et on commencera donc les programmes d'algèbre linéaire par les commandes

```
import numpy as np
import numpy.linalg as alg
```

ou en utilisant l'alias que vous voulez.

Dans un programme Python, on a tendance à ranger ses données sous forme de listes. Par exemple, les fonctions de plot attendent des listes de données. En Python, une liste est donnée entre crochet et les éléments sont séparés par des virgules :

```
u = [1, 2, 2]
```

représente le vecteur à 1 ligne et 3 colonnes $(1, 2, 2)$. Une liste n'est pas figée et on peut très bien lui ajouter ou lui enlever des cases. Ainsi par exemple, pour générer et garder en mémoire les termes d'une suite récurrente, on se sert d'une liste et d'une boucle for pour créer dans chaque tour de boucle une case que l'on remplie avec la valeur du terme de la suite qu'on vient de créer. Par exemple avec la suite

$$\begin{cases} u_0 & = & 1 \\ u_{n+1} & = & \sqrt{u_n} + 1 \end{cases}$$

on utilise le programme suivant pour en construire les termes

```
def suite(n) :
  L = [1] # on commence avec la liste qui ne contient que le premier terme
  u = 1
  for in in range (n) :
    u = u**(1/2)+1
    L.append(u)
  return L
```

C'est la commande append qui sert à créer une nouvelle entrée et à la remplir avec la valeur du terme dernièrement crée de la suite. La liste retournée par le programme est en suite directement exploitable par la fonction matplotlib.pyplot.plot.

Une matrice est une généralisation d'une liste, il s'agit en fait d'une liste de listes. Il faut penser que l'on définit une matrice ligne par ligne et que chaque ligne est en fait une liste. Pour que Python interprète cette donnée non pas comme une liste de listes brutes mais plutôt comme une matrice (pour pouvoir ensuite calculer des produits de matrices, leurs rangs, leurs valeurs propres,...), on utilise la fonction numpy.array. Par exemple

```
import numpy as np

A = np.array([[1,2,3],[4,5,6]])
```

crée la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

On peut ensuite faire toutes les opérations de l'algèbre linéaire avec Python :

1. Faire un produit de matrices avec np.dot

```
A = np.array([[1,2], [3,4]])
B = np.array([[1,1,1], [2,2,2]])
np.dot(A, B)
```

2. Élever une matrice à une certaine puissance avec alg.matrix_power

```
A = np.array([[1,2], [3,4]])

alg.matrix_power(A,3)
```

renvoie la matrice A à la puissance 3.

3. Résoudre un système linéaire avec alg.solve

```
A = np.array([[1,2], [3,4]])
b = np.array([1,5])
alg.solve(A, b)
```

retourne la solution du système $AX = b$ (il n'existe qu'une seule solution car la fonction alg.solve ne peut prendre comme argument qu'une matrice A inversible).

4. Calculer le rang d'une matrice [1] avec alg.matrix_rank

```
A = np.array([[1,2], [3,4]])
alg.matrix_rank(A)
```

Simulations de variables aléatoires. On renvoie au TP1 pour la manipulation de la bibliothèque numpy.random dédiée aux probas. Dans ce TP, nous verrons comment simuler les variables aléatoires discrètes de références.

Manipulations de données. On renvoie au TP2 pour la manipulation de la bibliothèque pandas et son utilisation dans les études statistiques.

1. L'auteur de ce texte conjecture que cette fonction sera très couramment employée par les concepteurs de concours (voir la feuille méthodo d'algèbre linéaire)