Python : Aide-Mémoire

Depuis le concours 2023, le langage officiel pour la programmation d'algorithmes est **Python**

1 Calculs numériques

Les commandes qui suivent permettent d'utiliser les fonctionnalités de base en calcul numérique :

Opérations

Opération :	addition	soustraction	multiplication	division	puissances
Python	$+$	$-$	$*$	$/$	$**$

Peu de diférence donc pour l'aspect *calculatrice*. Retenir simplement l'usage de $**$ pour les puissances qui diffèrent de l'habitude majoritaire en informatique (et calculatrices).

Nombres remarquables

Les valeurs particulières mathématiques à connaître en ECT sont les constantes π et $e = \exp(1)$. Cette dernière sera introduite et étudiée en semestre 2.

constante :	$\pi \approx 3,14$	$e \approx 2,72$
SciLab	%pi	%e
Python	`pi`	`e`

Import Python

Les constantes mathématiques sont accessibles une fois la bibliothèque (*library*) mathématique de Python importée.
Pour cela, taper au préalable la commande `from math import *` (pensez à bien mettre l'étoile)

Fonctions usuelles

Certaines de ces fonctions seront vues plus tard dans l'année, mais on en donne tout de même les commandes :

fonction :	racine carrée	partie entière	valeur absolue	exponentielle	logarithme
Python	`sqrt`	`floor`	`abs`	`exp`	`log`

On constate la similarité assez facilement.

Import Python

Ces fonctions mathématiques spéciales sont accessibles une fois la bibliothèque (*library*) mathématique de Python importée.
Pour cela, taper au préalable la commande `from math import *`.

2 Variables

Affectation et gestion des variables

En Python, l'affectation se fait avec le symbole $=$
On retiendra donc que l'affection se décrit de la façon `nom = expression` ; le symbole = n'est donc pas employé dans son sens usuel (il a valeur de $:=$ utilisé dans d'autres langages).

Exemple : La commande `X = 31**5/7` affecte à `X` une valeur approchée de $\dfrac{31^5}{7}$, résultat du calcul produit par la console Python.

- Par défaut, la console Python *n'affiche PAS* le résultat d'une affectation.
  On peut, à la console, l'obtenir en tapant le nom de la variable (puis -entrée-)
- On peut effacer un nom de variable `nom` = `None` (taper vraiment `None`).
- On peut afficher une variable à tout moment en tapant `print(nom_variable)`.
- On peut *réaffecter* une variable en réutilisant son nom et la commande d'affectation.

Attention ! En Python certaines lettres sont déjà utilisées par défaut !
Vous ne pouvez donc pas, par exemple, déclarer une variable `e` ou `pi` une fois la bibliothèque (library) `math` importée.

Typage des variables : Python

En Python, les variables doivent disposer d'un *type* de données. On peut affecter à une variable son type ou le modifier.

Type :	entier	flottant	booléen	listes	matriciel	chaîne de caractères
nom commun :	integer	float	boolean	array	ndarray	string
Commande	`int`	`float`	`bool`	`array`	`np.array`	`str`

Import Python

Les matrices mentionnées ne sont accessibles qu'une fois la bibliothèque (*library*) NumPy de Python importée.
La commande recommandée par le programme officiel ECT est : `import numpy as np`

Affection d'entrée

Il est possible de déclarer qu'une variable `X` soit saisie en entrée par l'utilisateur d'un programme. Pour cela, on utilise la commande `input`.
Pour utiliser cette fonctionnalité, il faudra taper une commande du type `nom = input("texte")` (et avec les guillemets). La valeur `texte` représente le message qui s'affichera lors de l'exécution de cette commande.

A Retenir : La saisie d'une entrée par la commande `input` est considérée, par défaut, comme du texte.
A noter : L'utilisation directe dans le terminal -ou console- est peu utile. Comme nous le verrons plus tard, il est préférable d'employer un éditeur de script Python pour rendre cette commande attrayante.

3 Editeur de script Python

Pour réaliser des scripts/programmes où plusieurs lignes de commandes sont nécessaires, on utilise souvent un éditeur de texte capable de reconnaître le langage Python.
Le principe est le même que pour SciNotes mais l'aspect visuel et les commandes reconnues peuvent varier d'un éditeur à l'autre.
Une liste non-exhaustive d'éditeurs Python :
— PyScripter
— PyCharm
— EduPython
— Visual Studio Code (qui gère d'autres langages de programmation)

4 Fonctions

Il est possible de créer des fonctions soi-même dans Python : pour cela, on utilise la structure de commandes suivantes :

```
def {nom_fonction}({variable}):
        {commandes choisies}
        [option :] return({sortie})
```

où les éléments écrits entre {} sont ceux choisis par l'utilisateur et l'usage des deux points : est *nécessaire*.
Par exemple :

```
def f(x):
        y=2*x^2-3*x+4
        return(y)
```

permet de déclarer la fonction du second degré $f : x \mapsto y = 2 * x^2 - 3 * x + 4$.

Avec plusieurs entrées

Il est possible de déclarer des fonctions de plusieurs variables. Pour cela, on utilise la syntaxe :

```
def {nom_fonction}({var_1, var_2, ... , var_p}):
        {commandes choisies}
        [option :] return({sortie})
```

Autre exemple (deux variables) :

```
from math import *              #biblothèque de mathématiques pour la racine carrée
def h(x,y):
        s=x*sqrt{y}+ y**2-x
        return(s)
```

permet de déclarer la fonction h de deux variables $h : (x; y) \mapsto s = x\sqrt{y} + y^2 - x$.

5 Listes et tableaux numériques

Création de Listes

Pour créer une liste, on place entre crochets **[ ]** des valeurs séparées chacune par une virgule **,**
On parlera aussi bien de *listes* (type `list`) que de *vecteurs* (type `array`) dans le cas de nombres. Python ouvre aussi la possibilité de créer des listes d'objets de types autres que numériques.
Les listes se déclarent donc de la façon suivante (sans être obligés de les nommer en pratique) :

liste = $[\, v_1 \, , \, v_2 \, , \, v_3 \, , \, \ldots \, , v_n]$.

Création de Tableaux

Import Python : Cette section requiert l'appel de `import numpy as np`.

On peut créer des tableaux nommés mathématiquement *matrices* dont le type Python est alors `ndarray` en les déclarants comme liste de listes :

matrice = `np.array`$([[\, m_{11} \, , \, m_{12} \, , \, \ldots \, , \, m_{1k} \,], [\, m_{21} \, , \, m_{22} \, , \, \ldots \, , \, m_{2k} \,], \, \ldots \, , [\, m_{l1} \, , \, m_{l2} \, , \, \ldots \, , \, m_{lk}]])$

si l'on choisit l lignes et k colonnes pour notre matrice.

Attention ! Le mode de déclaration diffère sensiblement des matrices sous SciLab et l'affiche est... disons moins ergonomique et naturel.
La plupart des commandes pratiquées sur les matrices porteront alors le préfixe `np`
Par exemple :

```
import numpy as np
Tbl = np.array([[2,3,4],[16,25,36]])
Sbl = np.sqrt(Tbl)
```

Permet de déclarer la matrice $Tbl = \begin{pmatrix} 2 & 3 & 4 \\ 16 & 25 & 36 \end{pmatrix}$ et $Sbl = \begin{pmatrix} \sqrt{2} & \sqrt{3} & 2 \\ 4 & 5 & 6 \end{pmatrix}$

6 Structure Conditionnelle

6.1 Aspect informatique

On appelle *structure conditionnelle* la partie d'un algorithme qui teste une *condition* puis réalise des instructions spécifiques selon la réalisation -ou non- de cette condition.
En Python, cette structure prend la forme suivante :

```
if {condition} :
        {instruction(s) si réalisée}
else :
        {instruction(s) si non-réalisée}
```

Remarques :
— L'usage des deux points : est encore nécessaire ici et remplace le très traditionnel "`then`" de beaucoup d'autres langages.
— Dans le cas où l'on n'aurait rien à faire si la condition n'est pas valide, il suffira de retirer les lignes `else` et toute instruction qui suit.
— L'indentation (comprendre, l'espace obtenu par tabulation) est interprété comme l'intérieur du bloc "`if`. Restez très vigilent sur son usage ou non-usage.
— On peut ajouter des `elif` au même niveau d'indentation que `else` pour créer d'autres options. Le `else` devra cependant toujours rester le dernier à être utilisé dans un même bloc `if`
Par exemple (à essayer) :

```
from math import *
x=float(input("choisissez un nombre"))
x=abs(x)
if x<100 :
        print("ce nombre est assez petit")
else :
        print("ce nombre a au moins trois chiffres")
print(x)
```

Pour effectuer des tests, on dispose des commandes suivantes :

relation :	égal	différent	inférieur strict	supérieur strict	inférieur large	supérieur large
commande	$==$	$!=$	$<$	$>$	$<=$	$>=$

Pour combiner des tests, on utilise les commandes suivantes :

connecteur :	et	ou	négation	constante vraie	constante fausse
commande	`and`	`ou`	*not*	`True`	`False`

Par exemple, on peut tester si $x \in [-3; 2]$ ainsi :

```
x=float(input("choisissez un nombre"))
if -3<=x & x<=2 :
        disp("x est bien dans l'intervalle")
end
```

6.2 Aspect Logique : les booléens

Les objets manipulés par Python pour les test sont dits *booléens* et correspondent au type `bool`. La présentation suivante est faite en français : le lecteur les traduira en langage Python.

On peut étudier une condition à l'aide de tableaux appelés *table de vérité*. Ils prennent cette forme :

ET	vrai	faux
vrai	vrai	faux
faux	faux	faux

OU	vrai	faux
vrai	vrai	vrai
faux	vrai	faux

$\Rightarrow$	vrai	faux
vrai	vrai	faux
faux	vrai	vrai

$\Leftrightarrow$	vrai	faux
vrai	vrai	faux
faux	faux	vrai

On peut ensuite les combiner pour tester si une condition correspond, telle qu'on l'a programmée, aux attentes.
Notations : En logique, on note $\wedge$ le connecteur ET ainsi que $\vee$ le connecteur OU

7 Structures répétitives

Les structures répétitives sont les parties d'algorithme consacrée à la réalisation répétée d'instructions identiques. Ces répétitions peuvent se faire en nombre fixé ou encore sous condition. On parle aussi de *boucles*.

7.1 Boucle For

Lorsque l'on connaît précisemment le nombre de tours à effectuer où les éléments à traiter de façon similaire, on a plutôt recourt à une boucle **for** dont la syntaxe est :

```
for {compteur} in {liste} :
        {instructions à executer}
```

On utilise souvent une lettre pour compteur et la commande `range(entier)` est très fréquente pour générer la liste d'énumération.
Exemple :

```
S = 0
for k in range(10):              # range(10) = [0,1,2,3,4,5,6,7,8,9]
        S=S+2**k                 # aspect récurrent S_(k+1) = S_k + 2^k
print(S)                         # on affiche le résultat
```

Remarque la commande `range(n)` génère la liste des entiers de 0 à $n-1$: c'est donc une liste à n éléments.

7.2 Boucle While

Cette boucle sera utilisée lorsque l'on connait une condition sous laquelle les instructions continuent d'être exécutées. Elle est adaptée aux situations où l'on ne connaît pas précisemment le nombre de tours. Sa syntaxe (assez simple) est :

```
while {condition} :
        {instructions à executer}
```

Les conditions employées suivent les mêmes schémas que pour la structure conditionnelle.
On donne un programme résolvant l'équation $1,025^n \geq 100$ en exemple (d'inconnue $n \in \mathbb{N}$) :

```
n = 0
q = 1.025
while q**n < 100 :
        n=n+1
print(n)
```

8 Compléments sur listes et tableaux

8.1 Listes et Matrices Particulières

Listes -sans import-

On peut retenir les commandes de générations automatiques de listes suivantes (sans importation) :

— `range(n)` qui génère la liste $[0, 1, 2 \ldots, n-1]$.
La lettre n doit alors désigner un `int` (entier)

— `range(n,m)` qui génère la liste $[n, n+1, n+2 \ldots, m-1]$.
Les lettres n et m doivent alors désigner des `int` (entiers) et $n < m$

— `range(n,m,k)` qui génère la liste $[n, n+k, n+2k \ldots, m-1]$.
Les lettres n, m et k doivent alors désigner des `int` (entiers) avec $n < m$ et k qui représente le *saut* d'incrémentation.

Listes -avec import de numpy-

On peut aussi retenir les commandes de générations automatiques de listes suivantes -avec usage de `import numpy as np`-

— `np.zeros(n)` qui génère une liste (vecteur) consituté de n (`int`) occurrences de "0" (zéro).

— `np.ones(n)` qui génère une liste (vecteur) consituté de n (`int`) occurrences de "1" (un).

— `np.arange` est une commande qui a exactement les mêmes modalités et fonctionnement que `range` mais qui génère un type `ndarray` compatible avec les matrices.

— `np.linspace(a,b,n)` qui génère une liste d'exactement n (`int`) valeurs régulièrement espacées entre a (`float`) et b (`float`) .

Remarque : Les commandes `np.zeros(n)` et `np.ones(n)` s'étendent aux matrices en tapant respectivement `np.zeros((l,k))` et `np.ones((l,k))` où l (`int`) désigne le nombre de lignes souhaitées et k (`int`) le nombre de colonnes.

Matrices -avec import de numpy-

On peut donner des commandes de générations automatiques de certaines matrices -sous couvert d'importer la bibliothèque `numpy` :

— `np.eyes(n)` la matrice identité I_n de taille n (`int`)

— `np.zeros((l,k))` la matrice de dimensions l (`int`) par k (`int`) remplie de "0" (zéro)

— `np.ones((l,k))` la matrice de dimensions l (`int`) par k (`int`) remplie de "1" (un)

Remarque : il y a bien un double parenthèsage en Python pour la création des matrices remplies de 0 ou remplies de 1.

8.2 Manipulation d'indices

- Pour une liste L donnée, on peut *appeler* le $k^{\text{ième}}$ élément en tapant simplement `L[k]`.
Attention ! En Python les indices de décomptes de lignes et colonnes commencent à 0 (zéro) et non à 1 (un).

  Par exemple (à taper pour vérifier) :

```
import numpy as np
L = [3,5,7]
x = L[1]                #c'est en fait 5
print(L)
print(x)
```

- Pour une matrice M, on peut appeler le coefficient de ligne l et colonne k en tapant `M`$[l,k]$.
  Attention ! En Python les indices de décomptes de lignes et colonnes commencent à 0 (zéro) et non à 1 (un).

  Par exemple (à taper pour vérifier) :

```
import numpy as np
M=np.array([[1,2,3],[3,5,6]])
a=M[1,2]                #c'est en fait 6
print(M)
print(a)
```

8.3 Transferts d'opérations sur listes

Les opérations usuelles d'addition (+), soustraction (-), multiplication ($*$), division décimale ($/$) et puissance ($**$) peuvent se transférer aux matrices (`ndarray`) par *usage habituel*. L'opération est alors effectuée coefficient par coefficent.

Attention ! La multiplication matricielle (mathématique) n'est pas la multiplication coefficient par coefficent !
Par exemple (à essayer) :

```
import numpy as np
L = np.array([[1,2],[3,5],[4,7]])
M = np.array([[1,2],[0,1],[-2,2]])
print(L+M, "add coeff à coeff")
print(L**M, "multi coeff à coeff")
```

On peut aussi utiliser le préfixe `np.` devant les constantes et fonctions usuelles présentées en section 1 pour les transférer, coefficient à coefficient, sur les patrices (`ndarray`)
Par exemple (à essayer) :

```
import numpy as np
E=np.e          #nombre e
T=np.array([[-2,6],[-E,E]])
T=np.abs(T)
print(T)
```

8.4 Opérations spécifiques sur matrices

Ces commandes Python nécessitent l'import de la bibliothèque `numpy`. Elles ne peuvent être employées que sur des objets de type `ndarray`

- La commande `np.shape` permet d'obtenir les dimensions d'un tableau.
  Par exemple :

```
import numpy as np
T=np.array([[-2,6],[-1,1],[4,0]])
a,b = np.shape(T)
print(a,b)
```

affiche les dimensions de la matrice T définie.

- La commande `np.objet.reshape` permet de reconstituer une matrice de type `ndarray` en une autre à condition que le nombre de coefficients total soit le même.
  Par exemple :

```
import numpy as np
T=np.array([[-2,6],[-1,1],[4,0]])
print(T)
M=T.reshape(2,3)
print(M)
```

  montre comment s'opère la transformation de T ayant trois lignes et deux colonnes en M ayant deux lignes et trois colonnes.
- La multiplication matricielle usuelle (au sens mathématique) s'effectue au moyen de la commande `np.dot`.
  Par exemple :

```
A=np.array([[1,0],[-1,2]])
B=np.array([[0,-1],[0,1]])
H=np.dot(A,B)
print(H)
H2=np.dot(B,A)
print(H2)
```

  permettra de déclarer les matrices $A = \begin{pmatrix} 1 & 0 \\ -1 & 2 \end{pmatrix}$ ainsi que $B = \begin{pmatrix} 0 & -1 \\ 0 & 1 \end{pmatrix}$ puis de calculer les produits $H = AB$ et $H_2 = BA$.
  Tester cet exemple permettra de s'assurer que le produit de deux matrices ne commute pas.
- Les commandes suivantes seront également étudiées :

  `np.sum` `np.min` `np.max` `np.mean` `np.cumsum` `np.median` `np.var` `np.std`

  Voir les TP correspondant pour plus d'informations sur ces commandes (leur usage s'accompagne d'une documentation *fournie* selon les termes du bulletin officiel.

9 Simulation et étude de l'aléatoire

Pour cette section, on devra utiliser la bibliothèque Python `numpy.random`. Le programme officiel ECT propose la ligne de commande suivante pour l'importation :

```
import numpy.random as rd
```

Aussi, toutes les commandes utilisées dans ce cadre auront `rd` comme préfixe. On peut voir qu'il s'agit d'une sous-bibliothèque de `numpy`.

9.1 Commande génératrice universelle

Toute simulation d'aléatoire peut être produite à partir de la simple commande `rd.random()` qui génère un nombre (pseudo-)aléatoire de l'intervalle $[0;1[$ renvoyé sous la forme d'un `float` :

```
>>> import numpy.random as rd
>>> rd.random()
        0.3326736648613903
>>> rd.random()
        0.4303227281283244
```

Il est impératif de comprendre que vos réponses ne seront pas nécessairement celles qui précèdent : l'essentiel est de vérifier que vos réponses sont le plus souvent d'horribles séquences de chiffres précédées de 0. [zéro - point]

Nous désignerons souvent par U une valeur aléatoire générée par cette commande et on dira que U suit une loi uniforme sur $[0;1[$ ce qui se résume sous la forme de la notation suivante :

$$\text{simul} : U \hookrightarrow \mathcal{U}[0;1[$$

lu :*simulation de U suivant une loi uniforme sur l'intervalle* $[0;1[$.

On peut insérer des paramètres dans la commande pour générer d'un coup plusieurs valeurs (pseudo) aléatoires :
— La commande `rd.random(n)` génère n variables de type simul : $U \hookrightarrow \mathcal{U}[0;1[$ sur une ligne

9.2 Les pandas et la statistique

Pour étudier les résultats des simulations, on chargera la bibliothèque d'importation `pandas` et on utilisera la commande d'appel suivante proposée par la programme officiel :

```
import pandas as pd                    #si si ...
```

Les commandes essentielles à retenir sont utilisables avec l'une ou l'autre des bibliothèques `numpy` (préfixée `np`.) ou `pandas` (préfixée `pd`.) et sont celles correspondant aux indicateurs statistiques usuels :

- `.mean` pour le calcul de moyenne
- `.var` pour le calcul de la variance.
- `.std` pour le calcul de l'écart-type.
- `.median` pour le calcul de la médiane.
- `.cumsum` pour l'obtention des efffectifs cumulés croissants.

Par exemple (à taper pour vérifier) :

```
import pandas as pd
import numpy as np
import numpy.random as rd
L=np.zeros(100)
for k in range(100):
    L[k]=rd.random()
M=L.mean()
s=L.std()
print("moyenne=",M)
print("standard sigma=",s)
```

Vous pouvez, en exercice, réduire la taille du code ci-dessus à l'aide de la commande `rd.random(100)` (à essayer)
Les variables `M` et `s` calculent les valeurs de moyenne et écart-type du vecteur `L`.

9.3 Représentations graphiques

On utilisera la bibliothèque `matplotlib.pyplot` déjà rencontrée pour générer les graphiques inhérents aux statistiques produites ou étudiées. La commande d'importation retenue sera :

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

et les graphiques exploités seront :

- Pour les histogrammes : `plt.hist`
- Pour les diagrammes en bâtons (ou barres) `plt.bar`
- Pour les diagrammes en bâtons (ou barres) `plt.boxplot`

La production d'un graphique en mémoire se fait avec la commande `plt.plot` et `plt.show` force l'ouverture d'une fenêtre graphique de visualisation.
Par exemple (à taper pour vérifier) :

```
from math import *
import numpy as np
import matplotlib.pyplot as plt
X=np.linspace(0,10,101)
Y=np.zeros(101)
for k in range(101):
    Y[k]=sqrt(X[k])
G=plt.plot(X,Y)
plt.show()
```