

# INFORMATIQUE AVEC PYTHON

PRÉPAS  
SCIENTIFIQUES

EXERCICES  
INCONTOURNABLES

Tout le catalogue sur  
[www.dunod.com](http://www.dunod.com)



ÉDITEUR DE SAVOIRS

JEAN-NOËL BEURY

# INFORMATIQUE AVEC PYTHON

PRÉPAS  
SCIENTIFIQUES

EXERCICES  
INCONTOURNABLES

DUNOD

*l'intégrale*

## Conception graphique de la couverture : Hokus Pokus Créations

|  |  |
|--|--|
| <p>Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.</p> <p>Le Code de la propriété intellectuelle du 1<sup>er</sup> juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements</p> | <p>d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.</p> <p>Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).</p> |
|--|--|



© Dunod, 2018

11 rue Paul Bert, 92240 Malakoff

[www.dunod.com](http://www.dunod.com)

ISBN 978-2-10-076901-8

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2<sup>o</sup> et 3<sup>o</sup> a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

# Table des matières

## Partie 1

### PRISE EN MAIN DE PYTHON ET QUELQUES PIÈGES À ÉVITER

- |                                    |           |
|------------------------------------|-----------|
| <b>1. Prise en main de Python</b>  | <b>3</b>  |
| <b>2. Quelques pièges à éviter</b> | <b>21</b> |

## Partie 2

### ALGORITHMES

- |  |           |
|--|-----------|
| <b>3. Algorithmes à connaître</b>        | <b>33</b> |
| <b>4. Extraits de sujets de concours</b> | <b>45</b> |

## Partie 3

### SIMULATION NUMÉRIQUE

- |  |            |
|--|------------|
| <b>5. Méthode d'Euler – Problème de Cauchy</b>               | <b>61</b>  |
| <b>6. Méthode des rectangles – Méthode des trapèzes</b>      | <b>97</b>  |
| <b>7. Méthode de Gauss avec recherche partielle du pivot</b> | <b>103</b> |
| <b>8. Traitement numérique du signal</b>                     | <b>113</b> |

## Partie 4

### FICHIERS

- |                                |            |
|--------------------------------|------------|
| <b>9. Gestion des fichiers</b> | <b>129</b> |
|--------------------------------|------------|

## Partie 5

### RÉCURSIVITÉ ET PILES

- |                        |            |
|------------------------|------------|
| <b>10. Récursivité</b> | <b>137</b> |
| <b>11. Piles</b>       | <b>157</b> |

|                                      |                         |            |
|--------------------------------------|-------------------------|------------|
|                                      | <b>Partie 6</b>         |            |
|                                      | <b>TRI</b>              |            |
| <b>12. Algorithmes de tri</b>        |                         | <b>173</b> |
|                                      | <b>Partie 7</b>         |            |
|                                      | <b>APPLICATIONS</b>     |            |
| <b>13. Traitement d'images</b>       |                         | <b>189</b> |
| <b>14. Algorithmique des graphes</b> |                         | <b>193</b> |
| <b>15. Codage</b>                    |                         | <b>201</b> |
|                                      | <b>Partie 8</b>         |            |
|                                      | <b>BASES DE DONNÉES</b> |            |
| <b>16. Requêtes SQL</b>              |                         | <b>217</b> |
| <b>Index</b>                         |                         | <b>231</b> |

Les exercices sont destinés aux étudiants de Sup et Spé,  
sauf ceux dont le titre porte la mention « (SPÉ) »,  
qui sont réservés aux étudiants de Spé.

Partie 1

# Prise en main de Python et quelques pièges à éviter

# Plan

|  |           |
|--|-----------|
| <b>1. Prise en main de Python</b>  | <b>3</b>  |
| 1.1 : Boucles, test, fonctions (banque PT 2015)                              | 3         |
| 1.2 : Manipulations de listes et de tableaux numpy                           | 7         |
| 1.3 : Tracé d'une fonction avec matplotlib                                   | 10        |
| 1.4 : Matrices   | 14        |
| <b>2. Quelques pièges à éviter</b>   | <b>21</b> |
| 2.1 : Variables locales, variables globales                                  | 21        |
| 2.2 : Copies de listes et de tableaux  | 22        |
| 2.3 : Passage par référence pour les listes<br>et les tableaux numpy         | 24        |
| 2.4 : Slicing, ou découpage en tranches,<br>et range, np.arange, np.linspace | 27        |



# Prise en main de Python

## Exercice 1.1 : Boucles, test, fonctions (banque PT 2015)

1. Soit l'entier  $n = 1\,234$ . Quel est le quotient, noté  $q$ , de la division euclidienne de  $n$  par 10 ? Quel est le reste ? Que se passe-t-il si on recommence la division euclidienne par 10 à partir de  $q$  ?

Écrire une fonction `calcul_base10` d'argument  $n$ , renvoyant une liste  $L$  contenant les restes des divisions euclidiennes successives.

Écrire le programme principal demandant à l'utilisateur de saisir un entier  $n$  strictement positif et renvoyant la décomposition en base 10 de l'entier  $n$ .

2. Écrire une fonction `somcube`, d'argument  $n$ , renvoyant la somme des cubes des chiffres du nombre entier  $n$ . On pourra utiliser la fonction `calcul_base10`.

3. Écrire une fonction permettant de trouver tous les nombres entiers strictement inférieurs à 1 000 égaux à la somme des cubes de leurs chiffres.

4. Écrire une fonction `somcube2` qui convertit l'entier  $n$  en une chaîne de caractères permettant ainsi la récupération de ses chiffres sous forme de caractères. Cette nouvelle fonction renvoie la chaîne de caractères ainsi que la somme des cubes des chiffres de l'entier  $n$ . On pourra utiliser la fonction `str` et manipuler les chaînes de caractères.

### Analyse du problème

Cet exercice est extrait du sujet de concours 0 de la banque PT 2015. Il permet de s'entraîner à manipuler les fonctions, les boucles, les tests et les différents types rencontrés dans Python 3.

#### Cours :

L'installation de Python 3 peut se faire très facilement avec la suite Anaconda 3 (recherche Google : « anaconda 3 python 3 »). On pourra utiliser Spyder comme éditeur qui s'installe automatiquement avec Anaconda.



Dans Python 3,  $4/3$  renvoie 1.333 alors que, dans Python 2,  $4/3$  renvoie 1, c'est-à-dire le quotient de la division euclidienne de 4 par 3 ! Les programmes seront réalisés exclusivement avec Python 3 dans cet ouvrage.

Une affectation dans Python se fait avec la syntaxe  $n = 3$  :  $n$  prend la valeur 3.

Il est préférable de ne pas utiliser d'accent pour les noms de variables. On peut utiliser « `_` » dans le nom des variables mais pas « `-` ».

On peut ajouter des commentaires dans les programmes Python avec le symbole dièse :  
`# commentaire sur le programme.`

Le type d'une variable s'obtient avec la syntaxe `type(n)`.

Les types de base des variables dans Python sont :

- Entiers : `int`  
`n // 10` : quotient de la division euclidienne de  $n$  par 10  
`n % 10` : reste de la division euclidienne de  $n$  par 10  
`n ** 3` :  $n$  puissance 3
- Nombres à virgule flottante : `float`
- Opérations de base : `+`, `-`, `*`, `/`
- Booléens : `bool`. Les variables booléennes sont égales à `True` (vrai) ou `False` (faux).
- Opérateurs :  
`a == b` : cet opérateur compare  $a$  et  $b$ . Si  $a=b$ , Python retourne `True` sinon `False`  
`a != b` :  $a$  différent de  $b$   
`a > b`, `a < b`, `a >= b`, `a <= b` : strictement supérieur, strictement inférieur, supérieur ou égal, inférieur ou égal  
`or` : ou  
`and` : et  
`not` : non

Les types de base des conteneurs dans Python sont :

- Chaînes de caractères : `str`

### Exemple

```
s = "abcdef" # on peut écrire également s = 'abcdef'
s[0] # affiche le premier caractère, ici 'a'
s[-1] # affiche le dernier caractère, ici 'f'
```

Pour extraire des caractères, voir exercice 2.4 « Slicing, ou découpage en tranches, et `range`, `np.arange`, `np.linspace` ».

- Listes : `list`

### Exemple

```
L = [] # création d'une liste vide
L.append(3) # ajoute 3 dans la liste L
L.append(2) # ajoute 2 dans la liste L
L.pop() # efface le dernier élément (ici 2) de la liste L et retourne
# sa valeur
L1 = ['r', 3, 'te'] # création d'une liste contenant des caractères et des
# entiers
len(L) # affiche la longueur de la liste L
```

Pour extraire des éléments d'une liste, voir exercice 2.4 « Slicing, ou découpage en tranches, et `range`, `np.arange`, `np.linspace` ».

- Tuples : `tuple`. Une fois le tuple créé, il ne peut pas être modifié.  
`M = (2, 3, 8)` : création du tuple. On met des parenthèses. Ne pas confondre avec les listes, dans lesquelles on met des crochets.

- Tableaux numpy : pour leur utilisation, voir exercice 1.2 « Manipulations de listes et de tableaux numpy ».

Quelques fonctions intrinsèques :

abs(x) : renvoie la valeur absolue  
 int(x) : convertit x en entier  
 float(x) : convertit x en flottant  
 str(x) : convertit x en chaîne de caractères  
 bool(x) : convertit x en booléen

### Première utilisation de la boucle for :

```
for i in range(n): # boucle faisant varier i de 0 à n-1
    # ne pas oublier ':' à la fin de la ligne
    # voir exercice 2.4 "Slicing, ou découpage en tranches,
    # et range, np.arange, np.linspace"
    x = x + i # ce code ajoute i à x à chaque passage dans la boucle
    # attention à l'indentation
```

### Deuxième utilisation de la boucle for :

```
for elt in chaine: # elt prend successivement les éléments de chaine
```

### Pour l'utilisation de la boucle while, ne pas oublier :

```
i = 0
while i < 3: # boucle exécutée tant que i < 3
    x = x + i # attention à l'indentation
    i = i + 1
```

### Définition d'une fonction :

```
def f(x): # définition de la fonction f ayant pour argument d'entrée x.
    # ne pas oublier ':' à la fin de la ligne
    y = x + 3 # y est une variable locale : elle est créée à l'appel
    # de la fonction et est détruite à la fin de la fonction
    # voir exercice 2.1 "Variables locales, variables globales"
    return y # fin de la fonction et retourne la valeur y
    # attention à l'indentation
```

### Structure conditionnelle :

```
if x == 3: # teste si x = 3
    y = 3 * x
elif x > 3 and x <= 4: # si le test précédent n'est pas vérifié,
    # alors teste si x > 3 et si x <= 4
    y = x + 2
elif x > 4 and x < 5: # si le test précédent n'est pas vérifié,
    # alors teste si x > 4 et si x < 5
    y = x - 2
else: # sinon (les tests précédents ne sont pas vérifiés)
    y = 0
```



1.  $n = 1\ 234 = 123 \times 10 + 4$ . Le reste vaut 4.

Si on recommence la division euclidienne de 123 par 10 :  $123 = 12 \times 10 + 3$ .  
Le reste vaut 3.

Si on recommence la division euclidienne de 12 par 10 :  $12 = 1 \times 10 + 2$ .  
Le reste vaut 2.

Si on recommence la division euclidienne de 1 par 10 :  $1 = 0 \times 10 + 1$ .  
Le reste vaut 1.

On obtient la décomposition en base 10 de  $n$  : 1, 2, 3, 4.

```
def calcul_base10(n):
    L=[] # création d'une liste vide
    while n > 0: # boucle tant que n > 0
        q=n//10 # quotient de la division euclidienne de n par 10
        r=n%10 # reste de la division euclidienne de n par 10
        L.append(r) # on ajoute le reste dans la liste L
        n=q
    L.reverse() # renverse la liste L
    return L # on retourne la liste L

# programme principal
n=int(input('Taper un entier strictement positif : '))
# conversion en entier du résultat de la saisie
print('Décomposition en base 10 : ',calcul_base10(n))
```

2.

```
def somcube(n):
    somme=0 # initialisation de la variable somme
    L=calcul_base10(n) # on récupère la liste donnant
                        # la décomposition en base 10 de n
    for i in range(len(L)): # i varie de 0 à len(L)
        somme=somme+L[i]**3 # on ajoute L[i] à la puissance 3
                            # on peut écrire somme+=L[i]**3
    return somme

# programme principal
n=1234
print(somcube(n)) # affiche 100 pour n = 1234
```

3.

```
def affiche_liste_entier_cube(): # pas d'argument d'entrée
    L=[] # création d'une liste vide
    for i in range(1000): # i varie entre 0 et 999
        if i==somcube(i): # teste si i est égal à la somme des
                            # cubes de ses chiffres
            L.append(i) # ajoute i dans la liste L
    return L # fin de la fonction et renvoi de la liste L

# programme principal
print(affiche_liste_entier_cube()) # affiche
                                    # [0, 1, 153, 370, 371, 407]
```

## 4.

```
def somcube2(n):
    somme=0
    chaine=str(n) # convertit n en une chaîne de caractères
    L=[ ]
    for elt in chaine: # elt prend successivement les éléments
                        # de chaine
        L.append(elt) # elt est un caractère que l'on ajoute
                    # dans L
        somme=somme+(int(elt)**3) # il faut convertir elt
                                # en entier

    return L,somme

# programme principal
n=int(input('Taper un entier strictement positif : '))
L1,res=somcube2(n) # L1 contient la liste des chiffres de n
                  # res = somme des cubes des chiffres de n
```

### Exercice 1.2 : Manipulations de listes et de tableaux numpy

On considère la fonction  $f$  définie par :  $f(x) = \sin(x) - x$ .

1. Définir la fonction  $f$  dans Python en utilisant la bibliothèque `numpy`.
2. Définir une liste `X1` contenant 20 valeurs régulièrement espacées entre 10 et 30. Définir une liste `Y1` contenant 20 valeurs définies par  $Y1[i] = f(X1[i])$ . On utilisera la syntaxe `range`.
3. Définir un tableau `numpy` `X2` contenant 20 valeurs régulièrement espacés entre 10 et 30. Définir un tableau `Y2` contenant 20 valeurs définies par  $Y[i] = f(X[i])$ . On utilisera la syntaxe `range`.
4. Définir un tableau `numpy` `X3` contenant 20 valeurs régulièrement espacés entre 10 et 30 en utilisant la syntaxe `np.linspace`. Définir un tableau `Y3` contenant 11 valeurs définies par  $Y[i] = f(X[i])$  sans utiliser la syntaxe `range`.

#### Analyse du problème

On va étudier dans cet exercice la différence entre les listes et les tableaux `numpy`. Il faut savoir manipuler aussi bien les listes que les tableaux `numpy`.

Voir exercice 13.1 « Traitement d'images et filtrage passe-bas » pour l'utilisation des matrices.

#### Cours :

#### Manipulation de listes avec Python

##### Exemple

```
L1 = [ ] # création d'une liste vide
L2 = [1,2,'a','b'] # création d'une liste avec 4 éléments
L2.append(3) # on ajoute 3 dans la liste L2. On a alors
            # L2 = [1,2,'a','b',3]
len(L2) # renvoie la longueur de la liste L2, ici 5
x=L2.pop() # efface le dernier élément de la liste et retourne sa valeur
          # on obtient L2 = [1,2,'a','b']
```

```
del L[i] # supprime de la liste L son élément d'indice i
L3=[2,5,3,1] # création de la liste L3
L3.sort() # cette fonction ne retourne rien, elle trie sur place
# la liste L3
# on obtient alors L3=[1,2,3,5]
2 in L3 # détermine si 2 est dans la liste L3 donc renvoie true ici
8 not in L3 # déterminer si 8 n'est pas dans la liste L3 donc renvoie
# true ici
L=list(range(4)) # renvoie une liste contenant les 4 premiers entiers
# dans l'ordre croissant
# donc L=[0,1,2,3]
```

Une liste peut contenir des éléments avec des types différents.



Dans Python, les indices des listes commencent à 0. Par contre, dans le logiciel Scilab, les indices des listes commencent à 1.

### Cours :

#### Manipulation de tableaux numpy avec Python

La bibliothèque `numpy` permet d'effectuer du calcul matriciel avec une structure de tableaux, de vecteurs et de matrices. Les calculs sont beaucoup plus rapides qu'avec des listes car on peut avoir des types différents dans une liste. Tous les éléments d'un tableau `numpy` doivent avoir le même type.

La bibliothèque `numpy` est importée dans Python avec la syntaxe :

```
| import numpy as np
```

On peut alors utiliser les fonctions de la bibliothèque `numpy` :

- `np.array(liste)`

*Argument d'entrée* : une liste définissant un tableau à 1 dimension (vecteur) ou à 2 dimensions (matrice).

*Argument de sortie* : un tableau (matrice).

*Description* : cette fonction permet de créer une matrice (de type tableau) à partir d'une liste.

#### Exemple :

```
L1=np.array([1,2,3]) # retourne [1,2,3]. Cette fonction transforme la
# liste en tableau numpy
# L1 est un tableau contenant 3 valeurs
L1[0] # retourne l'élément 0 du tableau, c'est-à-dire la valeur 1
np.size(L1) # la fonction retourne la taille du tableau : 3

L2=np.array([[1,2,3,4],[5,6,7,8]])
# L2 est une matrice contenant 2 lignes et 4 colonnes
np.shape(L2) # la fonction retourne (2,4) soit 2 lignes et 4 colonnes
L2[i,j] # la fonction qui retourne l'élément de la ligne i et
# de la colonne j de la matrice L2
L2[0,0] # la fonction retourne 1
L2[0,1] # la fonction retourne 2
```

```
L2[1,3] # la fonction retourne 8
L2[0,:] # la fonction retourne la ligne : [1,2,3,4]
L2[:,1] # la fonction retourne la colonne : [2,6]
```

- `np.zeros((n,m))`

*Argument d'entrée* : un tuple de deux entiers correspondant aux dimensions de la matrice à créer.

*Argument de sortie* : un tableau (matrice) d'éléments nuls.

*Description* : fonction créant une matrice (tableau) de dimensions  $n \times m$  dont tous les éléments sont nuls.

**Exemple :**

```
np.zeros(3) # crée le tableau [0,0,0]
np.zeros((2,4)) # crée la matrice :
#           array([[0., 0., 0., 0.],
#                  [0., 0., 0., 0.]])
```

Voir cours sur les matrices dans l'exercice 1.4 « Matrices ».

- `np.linspace(Min,Max,nbElements)`

*Argument d'entrée* : un tuple de 3 entiers.

*Argument de sortie* : un tableau (vecteur).

*Description* : fonction créant un vecteur (tableau) de `nbElements` nombres espacés régulièrement entre `Min` et `Max`. Le premier élément est égal à `Min`, le dernier est égal à `Max` et les éléments sont espacés de  $\frac{Max-Min}{nbElements-1}$ .

**Exemple :**

```
np.linspace(1,3,5) # la fonction retourne : [1, 1.5, 2, 2.5, 3]
np.arange(0,10) # la fonction retourne : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Voir exercice 2.4 « Slicing, ou découpage en tranches, et range, np.arange, np.linspace » pour avoir plus d'explications sur `np.arange`.



Lorsqu'on manipule des tableaux numpy, il faut connaître à l'avance le nombre d'éléments. Ce n'est pas nécessairement le cas avec les listes.



**1.**

```
import numpy as np
def f(x):
    return np.sin(x)-x
```

**2.**

```
N=20 # nombre de points
X1=[] # création de la liste vide X1
Y1=[] # création de la liste vide Y1
xmin=10
xmax=30
pas=(xmax-xmin)/(N-1) # intervalle entre deux valeurs de x
for i in range(N): # i varie entre 0 et N-1
    X1.append(xmin+i*pas)
    Y1.append(f(X1[i]))
```

## 3.

```
X2=np.zeros(N) # création du tableau X2
Y2=np.zeros(N) # création du tableau Y2
                # comportant N zéros
for i in range(N): # i varie entre 0 et N-1
    X2[i]=xmin+i*pas
    Y2[i]=f(X2[i])
```

## 4.

```
X3=np.linspace(xmin,xmax,N)
Y3=f(X3)
```

La fonction  $f$  peut être utilisée avec des tableaux numpy puisqu'on utilise des fonctions mathématiques de la bibliothèque `numpy`. On verra dans l'exercice 1.3 « Tracé d'une fonction avec matplotlib » comment vectoriser une fonction, c'est-à-dire créer une fonction pouvant être utilisée avec des tableaux numpy. Le temps de calcul est dans ce cas beaucoup plus faible qu'avec des boucles `for`.

## Exercice 1.3 : Tracé d'une fonction avec matplotlib

On considère les fonctions  $f_1$  et  $f_2$  définies sur  $[0, 2]$  par :

$$f_1(x) = \begin{cases} x & \text{pour } 0 \leq x < 1 \\ 1 & \text{pour } 1 \leq x \leq 2 \end{cases} \quad \text{et } f_2(x) = \sin(x) + 0,1.$$

On rappelle la syntaxe Python pour le tracé de fonctions :

```
import matplotlib.pyplot as plt
plt.figure() # nouvelle fenêtre graphique
plt.plot(x,y,color='r', linewidth =3,marker='o')
# color : choix de la couleur ('r' : red, 'g' : green, 'b' : blue,
# 'black' : black)
# linewidth : épaisseur du trait
# marker : différents symboles '+', '.', 'o', 'v'
# linestyle : style de la ligne ('-' ligne continue,
# '--' ligne discontinue, ':' ligne pointillée)
plt.plot(x,y,'*') # points non reliés représentés par '*'
plt.grid() # affichage de la grille
plt.title('Titre') # ajout d'un titre
plt.xlabel('axe x') # affiche 'axe x' en abscisse d'un graphique
plt.ylabel('axe y') # affiche 'axe y' en ordonnée d'un graphique
plt.axis ([xmin,xmax,ymin,ymax]) # précise les bornes pour les
# abscisses et les ordonnées
plt.legend(['courbe 1','courbe 2']) # permet de légender les courbes
plt.show() # affiche la figure à l'écran
```



```
import numpy as np
f_vect=np.vectorize(f) # vectorisation de la fonction f
                        # transforme la fonction scalaire en une fonction pouvant être
                        # utilisée avec des tableaux numpy
```

1. Définir les deux fonctions  $f_1$  et  $f_2$  dans Python en utilisant la bibliothèque `math`. Tracer les courbes représentatives des deux fonctions sur l'intervalle  $[0, 2]$  avec un pas de 0,05. Le graphique doit avoir les caractéristiques suivantes :

- Utilisation de listes
- Courbe représentative de  $f_1$  : épaisseur du trait égale à 3, couleur bleue
- Courbe représentative de  $f_2$  : points non reliés représentés par \*, couleur rouge
- Légender les deux courbes
- Afficher « x » pour l'axe des abscisses et « y » pour l'axe des ordonnées
- Afficher le titre : « Tracé de fonctions »
- Axe des  $x$  compris entre 0 et 2, axe des  $y$  compris entre 0 et 1,5

2. Définir les deux fonctions  $F_1$  et  $F_2$  dans Python en utilisant la bibliothèque `numpy`. On pourra vectoriser la fonction  $F_1$ . Tracer les courbes représentatives des deux fonctions sur l'intervalle  $[0, 2]$ . Le graphique doit avoir les caractéristiques suivantes :

- Utilisation de `np.linspace` avec 21 points et de tableaux `numpy`
- Courbe représentative de  $f_1$  : épaisseur du trait égale à 3, couleur noire, ligne discontinue
- Courbe représentative de  $f_2$  : couleur bleue, symbole « o » et ligne continue
- Légender les deux courbes
- Afficher « x » pour l'axe des abscisses et « y » pour l'axe des ordonnées
- Afficher la grille
- Afficher le titre : « Tracé de fonctions »

## Analyse du problème

Il faut bien connaître la syntaxe de base pour tracer une fonction :

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(x,y) # représentation graphique de y en fonction de x
plt.show()
```

### Cours :

La bibliothèque `matplotlib` de Python permet de tracer des graphiques. On l'importe à l'aide de la commande :

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

Il faut connaître quelques fonctions de la bibliothèque `matplotlib` :

- `plt.plot(x, y)`

*Arguments d'entrée* : un vecteur d'abscisses  $x$  (tableau de dimension  $n$ ) et un vecteur d'ordonnées  $y$  (tableau de dimension  $n$ ). On peut utiliser également une liste d'abscisses  $x$  de longueur  $n$  et une liste d'ordonnées  $y$  de longueur  $n$ .

*Description* : fonction permettant de tracer un graphique de  $n$  points dont les abscisses sont contenues dans le vecteur  $x$  et les ordonnées dans le vecteur  $y$ . Cette fonction doit être suivie de la fonction `plt.show()` pour que le graphique soit affiché.

- `plt.xlabel(nom)`

*Argument d'entrée* : une chaîne de caractères.

*Description* : fonction permettant d'afficher le contenu de `nom` en abscisse d'un graphique.

- `plt.ylabel(nom)`

*Argument d'entrée* : une chaîne de caractères.

*Description* : fonction permettant d'afficher le contenu de `nom` en ordonnée d'un graphique.

- `plt.title(nom)`

*Argument d'entrée* : une chaîne de caractères.

*Description* : fonction permettant d'afficher le contenu de `nom` en titre d'un graphique.

- `plt.show()`

*Description* : fonction réalisant l'affichage d'un graphe préalablement créé par la commande `plt.plot(x, y)`. Elle doit être appelée après la fonction `plt.plot` et après les fonctions `plt.xlabel`, `plt.ylabel` et `plt.title`.

- `plt.axis([xmin, xmax, ymin, ymax])`

*Description* : fonction permettant de définir les valeurs limites pour l'axe des abscisses et celui des ordonnées.

- `plt.xlim([xmin, xmax])`

*Description* : fonction permettant de définir les valeurs limites pour l'axe des abscisses.

- `plt.ylim([ymin, ymax])`

*Description* : fonction permettant de définir les valeurs limites pour l'axe des ordonnées.

La bibliothèque `math` permet d'accéder à de nombreuses fonctions mathématiques. On peut l'importer à l'aide de la commande :

```
| import math
```

On pourra utiliser les fonctions trigonométriques `math.cos`, `math.sin`, `math.tan`, `math.asin`, `math.acos`, `math.atan`; les constantes `math.pi`, `math.e`; les fonctions `math.exp(x)`, `math.log(x)` logarithme népérien, `math.log10(x)` logarithme en base 10...

On peut importer la bibliothèque `math` avec la commande :

```
| from math import *
```

Dans ce cas, on peut utiliser les fonctions précédentes sans ajouter `math`.

La valeur  $\pi$  s'obtient en tapant directement `pi` dans Python.

On peut importer une seule fonction avec la commande : `from math import cos`.



### 1.

```
import matplotlib.pyplot as plt
import math

def f1(x): # définition de la fonction f1
    if x >= 0 and x < 1:
        return x
    elif x >= 1 and x <= 2:
        return 1
    else:
        return 0

def f2(x): # définition de la fonction f2
    return math.sin(x)+0.1

xmin=0 # valeur minimale de x
xmax=2 # valeur maximale de x
pas=0.05
N=int((xmax-xmin)/pas)+1 # pas=(xmax-xmin)/(N-1)
# le nombre de points doit être un entier
# N=nombre de points et N-1=nombre d'intervalles

x=[ ] # initialisation de la liste x
y1=[ ] # initialisation de la liste y1
y2=[ ] # initialisation de la liste y2

for i in range(N):
    x.append(i*pas) # ajout de l'élément x[i]
    y1.append(f1(x[i])) # ajout de l'élément f1(x[i])
    y2.append(f2(x[i])) # ajout de l'élément f2(x[i])

plt.figure() # nouvelle fenêtre graphique
plt.plot(x,y1,linewidth=3,color='b') # création de la première
# courbe
plt.plot(x,y2,'*',color='r') # création de la deuxième courbe
plt.legend(['f1(x)', 'f2(x)']) # légende des deux courbes
plt.xlabel('x') # affichage de 'x' en abscisse du graphique
plt.ylabel('y') # affichage de 'y' en ordonnée du graphique
plt.title('Tracé de fonctions') # affichage du titre du
# graphique

plt.axis([0,2,0,1.5]) # [xmin,xmax,ymin,ymax]
plt.show() # affiche la figure à l'écran
```



## 2.

```
import numpy as np # bibliothèque numpy renommée np
import matplotlib.pyplot as plt

def F1(x): # définition de la fonction F1
    if x >= 0 and x < 1:
        return x
    elif x >= 1 and x <= 2:
        return 1
    else:
        return 0

def F2(x): # définition de la fonction F2
    return np.sin(x)+0.1

N=21 # nombre de points
X=np.linspace(xmin,xmax,N) # valeurs entre xmin et xmax compris

F1_vect=np.vectorize(F1)
# transforme la fonction scalaire en une fonction
# pouvant être utilisée avec des tableaux numpy

Y1=F1_vect(X) # applique F1 à chaque élément de X
# temps de calcul moins long qu'avec des boucles
Y2=F2(X) # fonction déjà vectorisée avec np.sin

plt.figure() # nouvelle fenêtre graphique
plt.plot(X,Y1,linewidth=3,color='black',linestyle='--')
plt.plot(X,Y2,color='b',marker='o',linestyle='-')
plt.legend(['F1(x)', 'F2(x)'])
plt.xlabel('x')
plt.ylabel('y')
plt.grid() # affichage de la grille
plt.title('Tracé de fonctions')
plt.show() # affiche la figure à l'écran
```

## Exercice 1.4 : Matrices

On pourra utiliser la fonction `array` de la bibliothèque `numpy`, ainsi que les fonctions `eig` et `det` de la sous-bibliothèque `numpy.linalg` :

```
| from numpy.linalg import eig
| from numpy.linalg import det
```

La fonction `eig` renvoie les valeurs propres (liste des valeurs propres) et les vecteurs propres (colonnes de la matrice retournée). La fonction `det` renvoie le déterminant.

1. Créer deux matrices  $\mathbf{R} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$  et  $\mathbf{S} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ . Les afficher avec Python.
2. Créer une fonction `test`, d'argument `M`, renvoyant la valeur `n` si `M` est une matrice carrée d'ordre `n` (entier naturel non nul), et zéro dans tous les autres cas. Vérifier la fonction `test` sur `R` et sur `S`.
3. Déterminer les valeurs propres et les vecteurs propres de la matrice  $\mathbf{A} = \begin{pmatrix} 5 & -3 \\ 6 & -4 \end{pmatrix}$  en utilisant la fonction `eig`. Vérifier les propriétés des valeurs propres et des vecteurs propres de la matrice `A`.

### Analyse du problème

Dans cet exercice, on manipule les matrices avec des fonctions particulières de la bibliothèque `linalg`.

Voir exercice 7.1 « Résolution d'un système linéaire par la méthode de Gauss » pour l'utilisation du calcul matriciel.

#### Cours :

La bibliothèque `numpy` permet de créer des matrices.

La fonction `np.array` permet de convertir une liste en un tableau `numpy`.

```
import numpy as np
A=np.array([[3,2],[5,4]]) # transformation de la liste [[3,2],[5,4]]
                          # en tableau numpy
print(A) # on obtient une matrice à deux lignes et deux colonnes :
          #   [[3, 2]
          #    [5, 4]]
```

La fonction `np.zeros` permet de créer une matrice de zéros en précisant le nombre de lignes et de colonnes :

```
B=np.zeros((3,2)) # matrice de zéros avec 3 lignes et 2 colonnes :
                  #   [[0., 0.],
                  #    [0., 0.],
                  #    [0., 0.]]
```

La fonction `np.ones` permet de créer une matrice de 1 en précisant le nombre de lignes et de colonnes :

```
C=np.ones((2,3)) # matrice de 1 avec 2 lignes et 3 colonnes :
                 #   [[1., 1., 1.],
                 #    [1., 1., 1.]]
```

La fonction `np.identity` permet de créer une matrice identité d'ordre `n` :

```
D=np.identity(2) # matrice (2,2) avec des 1 sur la diagonale et des 0
                 # partout ailleurs :
                 #   [[1., 0.],
                 #    [0., 1.]]
```

`B[i, j]` désigne le terme de la matrice qui est à la ligne  $i$  et à la colonne  $j$  :

```
B[1,1]=2 # on modifie la ligne 1 et la colonne 1
B[2,0]=3 # on modifie la ligne 2 et la colonne 0
print(B)
# [[0., 0.]
#  [0., 2.]
#  [3., 0.]
```

La syntaxe `B.shape` permet d'obtenir la dimension de la matrice :

```
B.shape # retourne un tuple : (nblignes,nbcolonnes)
# (3, 2)
```

On peut récupérer le nombre de lignes et de colonnes :

```
B.shape[0] # nombre de lignes de la matrice B :
# 3
B.shape[1] # nombre de colonnes de la matrice B :
# 2
```

On peut effectuer du slicing sur les matrices (voir exercice 2.4 « Slicing, ou découpage en tranches, et `range`, `np.arange`, `np.linspace` ») :

```
B[start1 : stop1 : step1 , start2 : stop2 : step2]
```

Indices des lignes :

`start1` : indice de départ, inclus

`stop1` : indice final, exclu

`step1` : cette variable désigne le pas pour les indices des lignes

Indices des colonnes :

`start2` : indice de départ, inclus

`stop2` : indice final, exclu

`step2` : cette variable désigne le pas pour les indices des colonnes

Dans `B[start1 : stop1 : step1 , start2 : stop2 : step2]`, on peut omettre n'importe lequel des arguments :

```
# slicing sur les matrices
print(B[:,0]) # on récupère toute la colonne 0
# [0., 0., 3.]
print(B[1:3,1])
# on sélectionne les lignes d'indice i tel que 1 <= i < 3 et la
# colonne d'indice 1
# le pas vaut 1 par défaut
# [2., 0.]
```

La fonction `np.dot` permet d'effectuer le produit de matrices :

```
E=np.array([[1,0],[3,2]]) # création de la matrice E :
# [[1, 0],
#  [3, 2]]
```

```
F=np.dot(A,E) # produit de la matrice A par E :
              # [[9, 4],
              # [17, 8]]
```



L'opération  $F2=A * E$  n'effectue pas le produit de la matrice **A** par la matrice **E** mais effectue une multiplication terme à terme.

```
[[3, 0],
 [15, 8]]
```

De même,  $E/A$  effectue une division terme à terme.

### Cours :

On peut multiplier tous les termes de la matrice par un réel :

```
G = 3*A # multiplie tous les termes de la matrice A par 3 :
        # [[9, 6],
        # [15, 12]]
```

On peut effectuer des sommes terme à terme ainsi qu'une différence terme à terme :

```
H=A+E
    # [[4, 2],
    # [8, 6]]
I=G-2*D
    # [[7., 6.],
    # [15., 10.]]
```

Voir exercice 2.2 « Copies de listes et de tableaux » pour les problèmes de copies de matrices.



#### 1.

```
import numpy as np #bibliothèque numpy pour les tableaux numpy
R=np.array([[1,2,3],[4,5,6]]) # création de la matrice R
print(R)
S=np.array([[1,2,3],[4,5,6],[7,8,9]]) # création de la
                                     # matrice S
print(S)
```

Python affiche :

```
[[1, 2, 3]
 [4, 5, 6]]
```

et

```
[[1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]]
```

## 2.

```
def test(M):
    nbligne,nbcolonne=M.shape # on récupère nbligne et
                               # nbcolonne de la matrice M
    if (nbligne==nbcolonne):
        return nbligne
    else:
        return 0
print('Fonction test pour R : ',test(R)) # renvoie 0
print('Fonction test pour S : ',test(S)) # renvoie 3
```

3. On cherche les valeurs propres et les vecteurs propres de la matrice

$$\mathbf{A} = \begin{pmatrix} 5 & -3 \\ 6 & -4 \end{pmatrix}.$$

```
from numpy.linalg import eig # import de la fonction eig
from numpy.linalg import det # import de la fonction det
A=np.array([[5,-3],[6,-4]])
lambda0,vect0=eig(A)
print("valeurs propres",lambda0)
```

Python affiche :

```
[2., -1.]
```

Les valeurs propres de  $\mathbf{A}$  sont les scalaires  $\lambda$  vérifiant  $\det(\mathbf{A} - \lambda\mathbf{I}) = \begin{vmatrix} 5 - \lambda & -3 \\ 6 & -4 - \lambda \end{vmatrix} = 0$  avec  $\mathbf{I}$  la matrice identité. On peut résoudre l'équation du second degré  $(5 - \lambda)(-4 - \lambda) + 18 = 0$ , soit  $-20 - 5\lambda + 4\lambda + \lambda^2 + 18 = 0$ . En simplifiant, on a :  $\lambda^2 - \lambda - 2 = 0$ . On trouve les deux solutions affichées dans Python : 2 et  $-1$ .

Le programme Python renvoie les valeurs propres dans le tableau `lambda0`, donc `lambda0 = [2., -1.]`.

`lambda0[0]` permet de récupérer la valeur propre 2. De même, `lambda0[1]` permet de récupérer  $-1$ .

On vérifie que 2 et  $-1$  sont bien les valeurs propres de la matrice  $\mathbf{A}$  :

```
I2=np.identity(2) # création de la matrice identité d'ordre 2
print(det(A-lambda0[0]*I2)) # retourne 0 pour la valeur
                             # propre 2
print(det(A-lambda0[1]*I2)) # retourne 0 pour la valeur
                             # propre -1

print("vecteurs propres",vect0)
```

Les vecteurs propres sont les colonnes de la matrice `vect0` :

```
[[0.70710678, 0.4472136]
 [0.70710678, 0.89442719]]
```



Pour avoir le vecteur propre  $\mathbf{V}_0$  associé à  $\text{lambda0}[0]$ , on extrait la colonne 0 de la matrice  $\text{vect}_0$  :

```
V0=vect0[:,0] # vecteur propre associé à lambda0[0]=2
print(np.dot(A,V0)-lambda0[0]*V0) # on obtient [0,0]
print(V0) # [0.70710678, 0.70710678]
```

On vérifie que  $(\mathbf{A} - \lambda_0 \mathbf{I}_2) \mathbf{V}_0 = \mathbf{0}$  avec  $\lambda_0 = 2$  et  $\mathbf{V}_0 = \begin{pmatrix} 0,707 \\ 0,707 \end{pmatrix}$

Pour avoir le vecteur propre  $\mathbf{V}_1$  associé à  $\text{lambda0}[1]$ , on extrait la colonne 1 de la matrice  $\text{vect}_0$  :

```
V1=vect0[:,1] # vecteur propre associé à lambda0[1]=-1
print(np.dot(A,V1)-lambda0[1]*V1) # on obtient [0,0]
print(V1) # [0.4472136, 0.89442719]
```

On vérifie que  $(\mathbf{A} - \lambda_1 \mathbf{I}_2) \mathbf{V}_1 = \mathbf{0}$  avec  $\lambda_1 = -2$  et  $\mathbf{V}_1 = \begin{pmatrix} 0,447 \\ 0,894 \end{pmatrix}$



$\text{np.dot}(A, V1)$  : cette fonction renvoie le produit de la matrice  $\mathbf{A}$  par  $\mathbf{V}_1$ .

$\text{lambda0}[0] * \mathbf{I}_2$  : cette opération multiplie tous les termes de la matrice identité  $\mathbf{I}_2$  par le réel  $\text{lambda0}[0]$ .