

Polycopié de Formation Python



par

Régis FERREIRA DA SILVA
regis.ferreirads@ac-nice.fr

2021

Introduction

Depuis une trentaine d'années, la programmation s'est imposée comme une discipline essentielle à la formation des étudiants, futurs acteurs de la société de demain.

Ainsi, La loi d'orientation et de programmation pour la refondation de l'École de la République a renforcé la place du numérique au sein du système éducatif, s'inscrivant alors dans le cadre européen du DIGCOMP (Cadre européen des compétences numériques pour les citoyens [1]).

Les nouveaux programmes de lycée de Physique-Chimie prenant effet à la rentrée 2019 ont promu les activités de modélisation et de programmation. Le langage de programmation conseillé est le langage **Python** .

L'objectif de cette formation sera donc de vous familiariser avec ce langage. De nombreuses applications issues de situations pratiques seront proposées.

À la fin de la formation, vous devrez être capable :

- ☞ d'afficher un graphique simple et un histogramme,
- ☞ de savoir récupérer des données depuis un fichier importé sous format *.txt* ou *.csv*,
- ☞ de savoir faire un traitement statistique sur des données,
- ☞ de savoir faire une régression polynomiale,
- ☞ de savoir tracer des vecteurs,
- ☞ de créer des animations simples,
- ☞ de savoir résoudre numériquement ou analytiquement des équations.

1. `https://ec.europa.eu/social/main.jsp?langId=fr&catId=1315`

COMPÉTENCES NUMÉRIQUES AU LYCÉE

En classe de seconde

Erreur et Incertitude	**Évaluer qualitativement la dispersion d'une série de mesures indépendantes.** Représenter l'histogramme associé à une série de mesures à l'aide d'un tableur.
Mouvement et interactions	**Caractériser différentes trajectoires.** — Représenter les positions successives d'un système modélisé par un point lors d'une évolution unidimensionnelle ou bidimensionnelle à l'aide d'un langage de programmation. — *Réaliser et/ou exploiter une vidéo ou une chronophotographie d'un système en mouvement et représenter des vecteurs vitesse ; décrire la variation du vecteur vitesse.* Représenter des vecteurs vitesse d'un système modélisé par un point lors d'un mouvement à l'aide d'un langage de programmation.
Ondes et signaux	*Représenter et exploiter la caractéristique d'un dipôle.* Représenter un nuage de points associé à la caractéristique d'un dipôle et modéliser la caractéristique de ce dipôle à l'aide d'un langage de programmation.

En classe de Première - Spécialité

Erreur et Incertitude	**Évaluer qualitativement la dispersion d'une série de mesures indépendantes.** Représenter l'histogramme associé à une série de mesures à l'aide d'un tableur.
Mouvement et interactions	*Réaliser et/ou exploiter une vidéo ou une chronophotographie d'un système modélisé par un point matériel en mouvement pour construire les vecteurs variation de vitesse. Tester la relation approchée entre la variation du vecteur vitesse entre deux instants voisins et la somme des forces appliquées au système.* Utiliser un langage de programmation pour étudier la relation approchée entre la variation du vecteur vitesse d'un système modélisé par un point matériel entre deux instants voisins et la somme des forces appliquées sur celui-ci.
Constitution et transformations de la matière	*Déterminer la composition de l'état final d'un système et l'avancement final d'une réaction.* Déterminer la composition de l'état final d'un système siège d'une transformation chimique totale à l'aide d'un langage de programmation.
L'énergie : conversions et transferts	*Utiliser un dispositif (smartphone, logiciel de traitement d'images, etc.) pour étudier l'évolution des énergies cinétique, potentielle et mécanique d'un système dans différentes situations : chute d'un corps, rebond sur un support, oscillations d'un pendule, etc.* Utiliser un langage de programmation pour effectuer le bilan énergétique d'un système en mouvement.
Ondes et signaux	*Déterminer la période, la longueur d'onde et la célérité d'une onde progressive sinusoïdale à l'aide d'une chaîne de mesure.* Représenter un signal périodique et illustrer l'influence de ses caractéristiques (période, amplitude) sur sa représentation. Simuler à l'aide d'un langage de programmation, la propagation d'une onde périodique.

Projet expérimental et numérique

La dimension numérique repose sur l'utilisation de matériels(capteur éventuellement associé à un microcontrôleur) et de logiciels (tableur, environnement de programmation).

Terminale - Spécialité

Erreur et Incertitude	**Représenter l'histogramme associé à une série de mesures à l'aide d'un tableur ou d'un langage de programmation.**
	Simuler un processus aléatoire illustrant la détermination de la valeur d'une grandeur avec incertitudes-types composées.
Constitution et transformations de la matière	**À partir de données expérimentales, tracer l'évolution temporelle d'une concentration, d'une vitesse volumique d'apparition ou de disparition et tester une relation donnée entre la vitesse volumique de disparition et la concentration d'un réactif.**
	Déterminer le taux d'avancement final d'une transformation, modélisée par la réaction d'un acide sur l'eau.
	Tracer le diagramme de distribution des espèces d'un couple acide-base de pKA donné.
Mouvement et interactions	**Représenter des vecteurs accélération d'un point lors d'un mouvement.**
	Représenter, à partir de données expérimentales variées, l'évolution des grandeurs énergétiques d'un système en mouvement dans un champ uniforme à l'aide d'un langage de programmation ou d'un tableur.
	Exploiter des données astronomiques ou satellitaires pour tester les deuxième et troisième lois de Kepler.

Python dans les programmes de mathématiques

En seconde

— décrire les variations d'une fonction donnée par une formule

— Pour des données réelles ou issues d'une simulation, lire et comprendre une fonction écrite en Python renvoyant la moyenne m , l'écart type s , et la proportion d'éléments appartenant à $[m - 2s, m + 2s]$.

— Lire et comprendre une fonction Python renvoyant le nombre ou la fréquence de succès dans un échantillon de taille n pour une expérience aléatoire à deux issues.

— Observer la loi des grands nombres à l'aide d'une simulation sur Python ou tableur.

En première

— Lire, comprendre et écrire une fonction Python renvoyant la moyenne d'un échantillon de taille n d'une variable aléatoire.

— Simuler, avec Python ou un tableur, N échantillons de taille n d'une variable aléatoire, d'espérance μ et d'écart type σ . Si m désigne la moyenne d'un échantillon, calculer la proportion des cas où l'écart entre m et μ est inférieur ou égal à $\dfrac{2\sigma}{\sqrt{n}}$.

En Terminale

En algorithmique et programmation, le programme reprend les programmes de seconde et de première sans introduire de notion nouvelle, afin de consolider le travail des classes précédentes.

L'algorithmique trouve naturellement sa place dans toutes les parties du programme et aide à la compréhension et à la construction des notions mathématiques.

TABLE DES MATIÈRES

Rudiments 1

1 Prise en main de Python **3**

1. Présentation de **Spyder** . . . 3
2. Première utilisation . . . 3
3. Variable . . . 4
4. Intervention pendant la compilation . . . 5
5. Conversions de format . . . 5
6. Exécution dans une cellule . . . 6

2 Modules **7**

1. Qu'est-ce qu'un module ? . . . 7
2. Quel module utiliser ? . . . 8
3. Comment trouver de l'aide sur un module ou une fonction ? . . . 8

3 Les listes et les tableaux numpy **9**

1. Les listes . . . 9

   Créer une liste . . . 9

   Accès aux éléments de la liste . . . 10

Longueur d'une liste . . . 10
2. Les tableaux de numpy . . . 10
La fonction linspace() . . . 10
Action d'une fonction mathématique sur un tableau . . . 10

4 Premiers graphiques **11**
1. Pour commencer . . . 11
2. Exploitation simple de données . . . 12
3. Histogramme . . . 13

5 Comparaison de valeurs et tests **17**
1. Bloc et indentation . . . 17
2. Commande if . . . 17
3. Commande If ... Else . . . 18

6 Notion de boucle **21**
1. While . . . 21
2. For ... in 22

7 Les manipulations de listes **23**
1. Parcours séquentiel d'une liste . . . 23
2. Opérations possibles . . . 24
3. Les listes en compréhension . . . 24

8 Les fonctions **27**
1. Syntaxe de déclaration de fonction . . . 27
2. Méthode pour écrire une fonction . . . 28
Généralités . . . 28
Exemple . . . 28
3. Procédure . . . 29
Définition . . . 29
Exemples . . . 29
4. Appel de fonction . . . 29
5. Fonction avec un nombre indéterminé de paramètres . . . 30

6. Les fonctions lambda . . . 30

9 Fichier : lecture et écriture **31**
1. Ouverture et fermeture . . . 31
2. Lecture et écriture . . . 32
3. Lecture de fichiers avec pandas . . . 32
4. Lecture de fichiers avec la librairie csv . . . 33

10 Les vecteurs **35**
1. plt.arrow . . . 35
2. plt.quiver . . . 37

Éléments **39**

11 Traitement de données **41**
1. Traitement statistique avec numpy . . . 41
   Variables statistiques . . . 41
   Régression avec polyfit . . . 42
2. Régression avec scipy et la fonction linregress . . . 42
3. Modélisation par une fonction quelconque avec scipy.optimize . . . 43

12 Animations **45**
1. Un premier exemple . . . 45
2. Animations plus élaborées . . . 46
3. Construire des fichiers **gif** animés . . . 46

13 Des graphiques plus élaborés **49**
1. Anatomie d'une figure . . . 49
2. Dimensions de la fenêtre **figure** . . . 50
3. Les options proposées de **pyplot** . . . 50
4. Personnalisation des graduations . . . 51
5. Fonction paramètres du graphique . . . 52
6. Comment créer plusieurs figures ? . . . 52
   Deux courbes sur le même graphique . . . 52

Une fenêtre contenant deux figures . . . 53
7. Barres d'incertitudes . . . 53

Compléments 57

14 Résolutions d'équations **59**
1. Résolutions d'équations avec le module **scipy** . . . 59
2. Résolutions d'équations avec le module avec le module **sympy** . . . 60
Utilisation de la fonction **solve** . . . 60
Autres possibilités de **sympy** . . . 60

15 Interfaces graphiques **63**

16 Liaison série **67**
1. Le port série . . . 67
2. Récupérer les données du port série . . . 67
3. Sauvegarder les données dans un fichier csv . . . 68
4. Une communication dans les deux sens . . . 69

Annexe Utiliser du langage Python avec le logiciel Spyder **71**

Annexe Installer Pydroïd 3 sur votre tablette **73**

Annexe LaTeX **77**

Annexe Ressources numériques **83**

Rudiments

CHAPITRE 1

PRISE EN MAIN DE PYTHON

Des nombreux logiciels permettent d'écrire et de compiler du code ***Python*** *. Nous avons fait le choix du logiciel* ***Spyder*** *(**S**cientific **Py**thon **D**evelopment **Envir**onment) qui nous semble le plus conviviale et le plus simple d'utilisation. Il existe aussi des compilateurs en ligne sur internet tel que repl.it qui permettent de pallier à l'absence de logiciel installé sur les ordinateurs.*

1. Présentation de Spyder

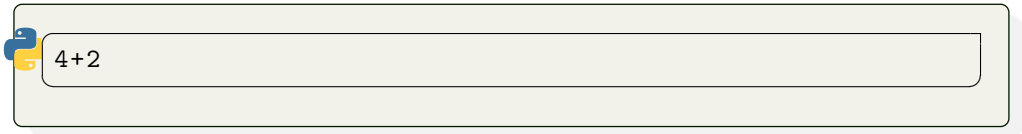
La plupart des distributions **Python** sont constituées de trois fenêtres :

1. L' éditeur qui permet de rédiger des programmes (fenêtre de gauche).
2. La console Python Interactive qui permet de tester des commandes (onglet : Console IPython) et qui affiche les résultats des programmes rédigés dans l'éditeur (autres onglets).
3. L'explorateur avec pour onglets :
   a. L'inspecteur d'objets qui donne des informations sur l'utilisation des fonctions activées ;
   b. L'explorateur de variables qui donne la liste et les valeurs de toutes les variables qui ont été créées ;
   c. L'explorateur de fichiers qui donne accès au disque dur.

2. Première utilisation

Un programme est une suite d'instructions dans un langage (**Python**) données à un interprète ici **Spyder** .

Commençons tout d'abord par un calcul très simple depuis l'éditeur de programmes. Afin que **Spyder** compile le programme, vous pouvez appuyer sur la flèche verte ou la touche F5 de votre clavier.



Aucun résultat n'apparaît dans la console. Il est donc nécessaire de demander à Spyder d'afficher la valeur de ce calcul. Cette commande s'appelle **print** :

```
print(4+2)
```

Après compilation, le résultat s'affiche dans la console.

Des commentaires peuvent être ajoutés au code, en faisant commencer le commentaire par le caractère hashtag (#).

```
#commentaire du premier code
print(4+2)
```

Si vous souhaitez mettre plusieurs lignes en commentaires, encadrez vos commentaires de ''' comme suit :

```
'''
commentaire
sur plusieurs
lignes
'''
print(4+2)
```

3. Variable

Dans le langage Python, il y a la possibilité de stocker des valeurs ou des caractères dans des variables. Ces dernières, une fois définies, peuvent être rappelés dans tout le programme.

```
A=4+2
B=2*A
print(B)
```

La valeur d'une variable peut être modifiée en lui attribuant une nouvelle valeur.

```
A=4+2
B=2*A
B=2
print(B)
```

Si l'on souhaite rajouter du texte, il est nécessaire de préciser que le texte est au format texte. Deux possibilités s'offrent à vous soit des apostrophes soit des guillemets

```
A=4+2
#première possibilité
print('valeur de A: ' ,A)
#deuxième possibilité
print("valeur de A:", A)
```

Afin de connaître le type d'une variable, la commande **type** est nécessaire.

```
A=4+2
print(type(A))
```

Application 1

Déterminer le type de :

1. 1.2 ;
2. 1 ;
3. 'mot'.

Plusieurs types de variables existent : les "chaines de caractères" ap-

pelés **string ou str** , les entiers ou **integer** ou les nombres à virgules flottantes nommés **float** .

Important



Tout calcul effectué sur des nombres entiers est parfaitement exact. La plupart des fractions décimales ne peuvent pas avoir de représentation exacte en fractions binaires. Par conséquent, en général, les nombres à virgule flottante que vous donnez sont seulement approximés en fractions binaires pour être stockés dans la machine. Les calculs utilisant des nombres flottants peuvent être entâchés d'erreurs.

4. Intervention pendant la compilation

De façon courante, un programme peut nécessiter l'intervention de l'utilisateur pour entrer un paramètre par exemple. La fonction **input** permet cela en provoquant une interruption dans l'exécution du programme pour que l'utilisateur puisse rentrer une valeur au clavier qui pourra être affectée à une variable. Une fois la valeur rentrée et après avoir validé avec la touche Enter, l'exécution du programme se poursuit.

```
prenom=input("Quel est votre prenom ?")
print("Bonjour, ",prenom,". Je suis ravi de vous
    rencontrer!")
```

Application 2

1. Faire un programme qui demande l'âge que vous avez, en créant une variable **age** .
2. Afficher le format de la variable **age** .
3. Exécuter-le en renseignant votre âge en toutes lettres puis en nombre.

La fonction **input()** renvoie toujours une chaîne de caractères. Si un entier est attendu en réponse ou autre chose, il faut alors faire une conversion de typage.

5. Conversions de format

Quatre formats vont être utilisés dans ce polycopié :

— le format **int** utilisé pour les nombres entiers relatifs, traduction du mot *integer* en anglais.

— le format **float** utilisé pour les nombres réels, aussi appelés nombres à virgule flottante approximant d'un nombre décimal.

— le format **str** utilisé pour les chaînes de caractères, traduction du mot chaîne en anglais *string*.

— le format **bool** pouvant contenir deux valeurs (True ou False).

Les constructeurs **int()** et **float()** transforment une chaîne ou avec un nombre de type **float** : de caractères en entier ou flottant. On peut aussi faire la conversion inverse avec le constructeur **str()** .

```
variable="128.68"
flottant=float(variable)
somme=flottant+12
print(somme)
```

Il est aussi possible d'afficher un nombre selon un format désiré (binaire 'b', hexadécimal 'x') en utilisant la fonction **format(valeur,'option')** qui crée une chaîne de caractères contenant la valeur écrite selon un format précisé dans 'option'.

Pour les nombres à virgule flottante (**float**), les options doivent indiquer la précision et/ou le format choisi pour le nombre à virgule, sous la forme '.précisionformat' :

- la précision correspond au nombre de chiffres affichés après la virgule
- le format est à choisir parmi les options ci-dessous :
  - **'f'** , affiche le nombre comme un nombre à virgule fixe.
  - **'e'** , affiche le nombre dans sa notation scientifique en utilisant la lettre "e" pour indiquer l'exposant.

La précision par défaut est 6.

Par exemple, le script suivant affichera le nombre a en écriture scientifique avec 2 chiffres après la virgule :

```
a=50026
print(format(a,'.2e'))
```

c'est-à-dire $5{,}00 \times 10^4$.

6. Exécution dans une cellule

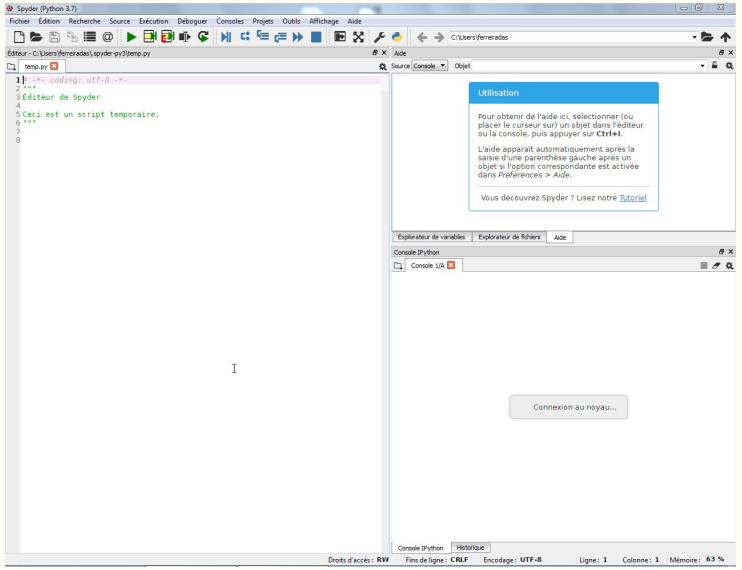
Si vous souhaitez exécuter une seule partie d'un code, il est nécessaire d'encadrer la partie de code avec les caractères #%%. Vous avez alors construit une **cellule** .

Pour exécuter la cellule crée, il vous suffit de cliquer sur l'icône  à droite de l'icône ▶.

Application 3 Identification des différentes fenêtres de **Spyder**

Sur la copie d'écran ci-dessous, identifier les différentes fenêtres ou raccourci :

- ✓ Éditeur pour écrire ses scripts **Python** ;
- ✓ Interpréteur **I**nteractif **Python** (**IPython**) ;
- ✓ Exécuter un fichier avec la touche ▶ (possibilité d'utiliser le touche F... du clavier) ;
- ✓ Arrêt d'exécution depuis la console IPython : ■.



CHAPITRE 2

MODULES

1. Qu'est-ce qu'un module ?

Essayons maintenant de faire un calcul en utilisant une fonction trigonométrique le cosinus.

```
A=cos(4)
print(A)
```

Le message d'erreur : **NameError : name 'cos' is not defined** s'affiche. Nous allons donc avoir besoin d'utiliser des fonctions prédéfinies.

Python offre de très nombreuses librairies de fonctions prédéfinies pour réaliser des tâches courantes. On a déjà vu les fonctions **input** et **print** . D'autres fonctions sont regroupées dans ce qui s'appelle des modules.

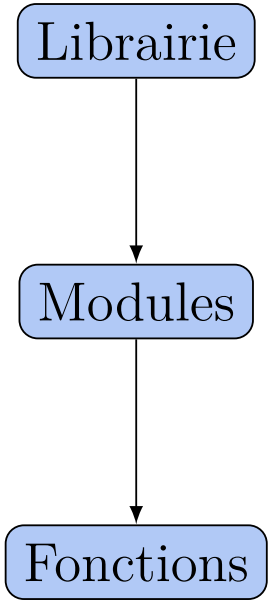


FIGURE 2.1 – Structure d'une librairie

Contrairement aux fonctions **print** et **input** qui sont accessibles

directement, pour pouvoir utiliser une fonction d'un module, il est nécessaire d'importer la fonction explicitement.

Par exemple, pour utiliser la fonction cos (cosinus) du module **numpy** :

```
from numpy import cos
A=cos(4)
print(A)
```

La ligne **from** numpy **import** cos indique que la fonction cos est importée du module **numpy** . Si on souhaite importer toutes les fonctions du module **numpy** , on peut écrire :

```
from numpy import *
```

Cette façon d'importer toutes les fonctions est pratique quand on utilise l'interpréteur interactif, mais peu recommandée dans les programmes, car elle nuit aux performances et à la lisibilité. La méthode recommandée est la suivante :

```
import numpy
A=numpy.cos(4)
print(A)
```

Si l'on souhaite importer plusieurs librairies, il est souhaitable de renommer les modules comme suit. L'appel de fonctions de **numpy** devra alors être précédé de **np.** .

```
import numpy as np
A=np.cos(4)
print(A)
```

2. Quel module utiliser ?

Voici quelques modules utiles dans un premier temps :

→ **numpy** : fonctions mathématiques, stocker et manipuler des données ;

→ **pyplot** (contenue dans **matplotlib**) : graphique ;

→ **sympy** : calcul symbolique ;

→ **stats** (contenue dans sympy) : traitement statistique ;

→ **random** : utilisation de fonctions permettant de simuler un comportement aléatoire ;

→ **csv** : manipulation de fichiers **csv** ou **txt** .

3. Comment trouver de l'aide sur un module ou une fonction ?

Depuis la console **IPython** , importer le module dont vous chercher des informations, puis taper la commande **?module** ou **?fonction** .

CHAPITRE 3

LES LISTES ET LES TABLEAUX NUMPY

Une liste est une collection ordonnée et mutable (modifiable) d'objets (type ***list*** *).*

- ☞ ***Collection ordonnée :*** *le premier élément de la liste est indexé par 0, le second par 1, et ainsi de suite.*
- ☞ ***Collection mutable :*** *il est possible, sans réaffectation complète, de changer le contenu de la liste : plus exactement, il est possible d'ajouter un élément, de supprimer un élément et d'étendre la liste par une autre liste.*

Les listes permettent de construire des collections d'objets de proche en proche : ce n'est que par la pratique que vous commencerez à imaginer l'étendue des possibilités !

1. Les listes

Créer une liste

Il existe plusieurs façons de créer une liste :

1 Directement en plaçant les éléments de la liste, nommée dans l'exemple suivant L, entre crochets [ ], séparés par des virgules ;

```
L=[1,3,7.4,5]
```

Une liste vide se crée donc ainsi :

```
L=[]
```

2 Par conversion d'un tuple (collection ordonnée et non modifiable d'objets, par exemple en utilisant la fonction **range** qui est un itérateur qui correspond à un objet ne contenant qu'un nombre en mémoire) ou d'une chaîne de caractères :

```
L=list(range(0,101,5))
```

ou

```
L=list('salut')
```

3 Par ajout élément par élément des objets dans une liste initialement vide.

4 En utilisant une liste en compréhension, vue précédemment au chapitre sur les boucles.

Accès aux éléments de la liste

liste[k] est le k+1$^{\text{ème}}$ élément de la liste, enfin tant que k reste positif. Ainsi dans la liste définie précédemment :

```
L=[1,3,7.4,5]
```

L[2] est égal à 7,4.

Longueur d'une liste

La commande **len()** (*lenght*, en anglais) permet très simplement de déterminer la longueur d'une liste.

Application 4

Déterminer la **longueur** de la liste : L=[1,3,7.4,5]

2. Les tableaux de numpy

Nous allons nous intéresser aux tableaux **numpy** qui vont nous permettre au prochain chapitre de tracer très facilement des courbes avec **Python**.

La fonction linspace()

numpy.linspace() permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments. Le code :

```
import numpy as np
np.linspace(3, 9, 10)
```

affiche un tableau de 10 éléments compris entre 3 et 9.

Action d'une fonction mathématique sur un tableau

NumPy dispose d'un grand nombre de fonctions mathématiques qui peuvent être appliquées directement à un tableau. Dans ce cas, la fonction est appliquée à chacun des éléments du tableau :

```
import numpy as np
x = np.linspace(-np.pi/2, np.pi/2, 3)
y = np.sin(x)
print(y)
```

affichera un tableau contenant le sinus de chacun des éléments du tableau x.

Application 5

Écrire un script permettant d'afficher un tableau contenant le logarithme (fonction **log** du module **numpy**) d'un tableau contenant 10 valeurs de concentrations en quantité de matière de valeurs comprises entre 0.010 et 0.15.

CHAPITRE 4

PREMIERS GRAPHIQUES

1. Pour commencer

Nous allons nous servir de **Python** pour tracer des graphiques.

Pour ce faire, il est nécessaire de demander à **Python** de charger le module **pyplot** de la librairie **matplotlib** , module dédié aux tracés de courbes.

Les fonctions que nous appellerons seront précédées par " **plt.** ".

Nous allons, par exemple, tracer la fonction f définie par $f(x) = x^2$.

```
import matplotlib.pyplot as plt # le module pyplot est
    renommé plt
import numpy as np # le module numpy est renommé np
x=np.linspace(1,10,100) # crée une liste de 100
   nombres sur l'intervalle [1; 10]
y=x**2 #fonction x au carré

#Début de la figure graphique
plt.figure(1) #nom du graphique
plt.clf() #nettoyage de la figure
plt.plot(x,y) # Trace la fonction mais ne montre pas
   le graphique.
plt.grid(True) #Affichage d'une grille
plt.show() # Affiche le graphe.
```

Le fait de nommer la figure et de la nettoyer (*clean figure*, en anglais), nous sera utile si nous faisons tracer plusieurs courbes dans le même script.

L'affichage d'un quadrillage (grille) se fait grâce à la commande **plt.grid(True**

.

Le programme se termine par la commande **plt.show()** qui affiche toutes les courbes. Si tout se passe correctement, vous devriez voir votre graphe avec un axe horizontal, un axe vertical et la courbe en bleu. S'il y a plusieurs courbes, chacune aura une couleur différente.

On peut regrouper les éléments de base d'une figure par le graphique suivant :

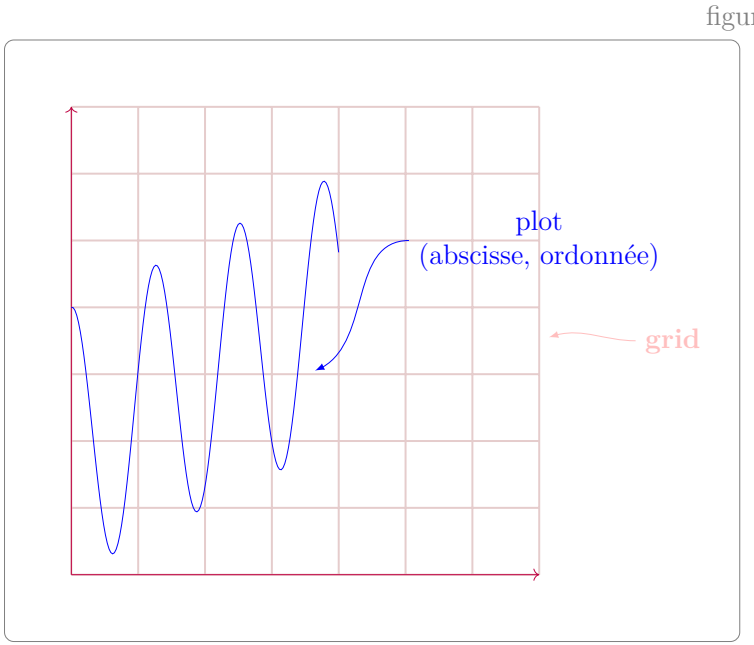


FIGURE 4.1 – Les éléments de base d'un graphique

La sauvegarde d'une figure s'effectue via la commande **plt.savefig('nom_du_fichier')** , par défaut le fichier sera sauvegarder au format **png** .

Si vous souhaitez modifier la résolution du fichier, il est nécessaire de préciser à la suite du nom du fichier la résolution souhaitée. Il est possible aussi de forcer la sauvegarde du fichier sous le format **pdf** , la résolution étant fixée. En reprenant le programme précédent :

```
import matplotlib.pyplot as plt # le module pyplot est
    renommé plt
import numpy as np # le module numpy est renommé np
x=np.linspace(1,10,100)
y=x**2
#Début de la figure graphique
plt.figure(1)
plt.clf()
plt.plot(x,y)
plt.grid(True)
plt.savefig('premier_graphique',dpi=350)
plt.show()
```

Application 6

1. Reprendre le programme précédent en modifiant la résolution.
2. Observer la variation de la taille du fichier sauvegardé.

Application 7 — Première Spé

1. Représenter le signal sinusoïdal suivant

$$5,2 \times \sin(\frac{2\pi \times t}{0,10})$$

en fonction du temps sur 4 périodes.

2. Modifier les paramètres du signal (amplitude, période et phase à l'origine) et observer les courbes obtenues.

2. Exploitation simple de données

La librairie **matplotlib** permet aussi de tracer des nuages de points assez facilement grâce à la commande **scatter** , *dispersion* en anglais (équivalent à la commande **scatter(x,y)** <=> **plot(x,y,'.')** . Deux arguments (abscisses et ordonnées) sont alors attendus :

```python
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [1, 2, 3, 4, 5]

#Début de la figure graphique
plt.figure(1)
plt.clf()
plt.scatter(x, y)
plt.grid(True)
plt.show()
```

Si l'on souhaite afficher plusieurs courbes sur le même graphique, il suffit de rajouter une deuxième commande **scatter** :

```python
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y1 = [1, 2, 3, 4, 5]
y2 = [1, 4, 9, 16, 25]


#Début de la figure graphique
plt.figure(1)
plt.clf()
plt.scatter(x, y1)
plt.scatter(x, y2)
plt.grid(True)
plt.show()
```

Une petite remarque, afin d'afficher les courbes demandées les listes créées (x et y) doivent impérativement être de même format(**type**) et de même taille (**len**).

Application 8 — Seconde

Afficher la trajectoire dont les données expérimentales obtenues sont :

Sur l'axe Ox : 0,2,4,6,8,10,12,14,16,18,20
Sur l'axe Oy : 0,1,3,6,10,15,21,28,36,45,55

Application 9 — Seconde

Tracer la caractéristique, $U = f(I)$, d'un dipôle dont on a mesuré la tension, U, à ses bornes en faisant varier l'intensité du courant, I, qui le traverse.

I(A) : 0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09
U(V) : 0.68,1.37,2.03,2.73,3.50,4.07,4.79,5.30,6.15

3. Histogramme

Pyplot propose la fonction **hist** permettant de tracer très simplement des histogrammes.

```python
import matplotlib.pyplot as plt
x = [1, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5]

#Début de la figure graphique
plt.figure(1) #nom du graphique
plt.clf() #nettoyage de la figure
plt.hist(x)
plt.grid(True)
plt.show()
```

Les options **bins** permet de choisir le nombre d'intervalles et **range** l'étendue des abscisses.

```python
import matplotlib.pyplot as plt

x = [1, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5]

#Début de la figure graphique
plt.figure(1) #nom du graphique
plt.clf() #nettoyage de la figure
plt.hist(x, range = (0, 5))
plt.grid(True)
plt.xlabel('valeurs')
plt.ylabel(r'$N_1$')
plt.title('Exemple d\' histogramme simple')
plt.show()
```

Application 10

Voici les moyennes de Physique-Chimie du premier trimestre de la classe 501.

0,7,2,5,1,3,5,6,3,4,1,8,9,1,4,2,6,3,8,4,5,3

Tracer un histogramme des moyennes de cette classe.

Application 11 Mettre un peu d'ordre dans un script Python

Soit une bille tombant dans un fluide visqueux. Le script **Python** permettant de tracer la solution analytique de la vitesse, v, a malheureusement été mélangé :

$$v(t) = g \times \tau(1 - e^{-\frac{t}{\tau}})$$

Remettre en ordre les morceaux de code suivants :

Partie 1

```python
# tracé des trajectoires
plt.plot(tab_t,tab_v,'g--',label="Solution exacte
    ")
plt.xlabel('Temps (en s)')
plt.ylabel('Vitesse de $M$ (en m/s)')
plt.title("Chute libre avec frottements fluides
    et sans vitesse initiale")
plt.text(0.7 * t_max,40.0,r'$h = ' + str(h) +'$ m
    ')
plt.text(0.7 * t_max,35.0,r'$t_{max} = ' + str(t_
    max) + '$ s')
plt.text(0.7 * t_max,45.0,r'$m = ' + str(m) +'$
    kg')
plt.text(0.7 * t_max,50.0,r'$\lambda = ' +str(
    lamb) + '$ kg/s')
plt.legend(loc=4)
plt.show()
```

Partie 2

```python
# paramètres physiques
g = 9.81 # accélération de la pesanteur
m = 1.0 # masse
lamb = 0.1 # coef. de frottement
tau = m / lamb
v0 = 0.0 # vitesse initiale
h = 5.0 # hauteur initiale
```

Partie 3

```python
#importation modules
import numpy as np
import matplotlib.pyplot as plt
```

Partie 4

```python
# fonction associée à la loi horaire de la
    vitesse
def v(tab_t,g,h,tau):
    return g * tau * (1.0 - np.exp(-tab_t /tau))
```

Partie 5

```python
#paramètres abscisses
t_min = 0.0 # instant initial
t_max = 5.0 * tau # instant final
n_t = 100
# listes de valeurs de t et v
tab_t = np.linspace(t_min,t_max,n_t)
tab_v = v(tab_t,g,h,tau)
```

CHAPITRE 5

COMPARAISON DE VALEURS ET TESTS

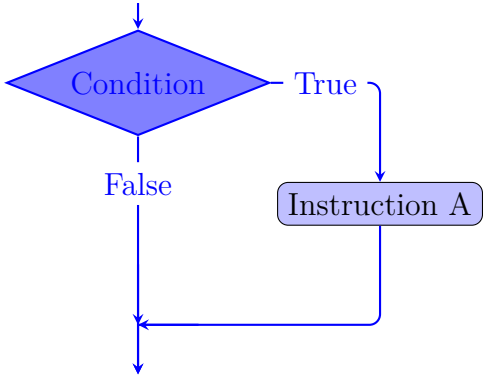
Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d' aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d' instructions capables de tester une certaine condition et de modifier le comportement du programme en conséquence. L'instruction ***if*** *(si en français) est la structure de test la plus simple. Sa syntaxe en* ***Python*** *fait intervenir la notion de bloc. Nous allons d'abord étudier cette notion plus en détail.*

1. Bloc et indentation

Un bloc est défini par une indentation obtenue en décalant le début des instructions vers la droite grâce à des espaces en début de ligne (4 espaces sous Windows). Toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces).

Un bloc peut contenir une ou plusieurs instructions, et notamment des instructions composées (tests, boucles, etc.).

2. Commande if



Ici **condition** est une expression booléenne, c'est-à-dire une expression qui prend pour valeur **True** (Vrai) ou **False** (Faux).

L'instruction **A** n'est exécutée que si la condition est vérifiée (c'est-à-dire si elle prend pour valeur **True**).

L'instruction **if** est une instruction composée. Le **:** (deux-points) à la fin de la ligne introduit le bloc d'instructions qui sera exécuté si la

condition est vérifiée.

Exemple 1 avec condition vraie :

```
x = 15
if x > 10:
    print(x, "est plus grand que 10")
print("Fin")
```

Exemple 1 avec condition fausse :

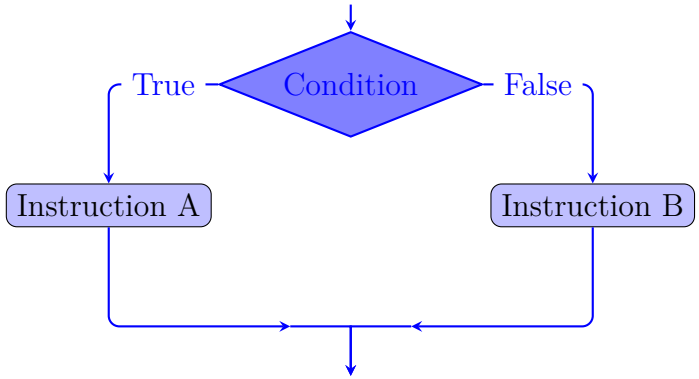
```
x = 3
if x > 10:
    print x, "est plus grand que 10"
print("Fin")
```

Exemple 2 avec condition vraie :

```
x = 5
if x > 0:
    print(x, "est plus grand que 0")
    print("il est strictement positif")
print("Fin")
```

3. Commande If ... Else

L'instruction **else** (sinon) permet de rajouter une instruction si la condition du **if** n'est pas remplie (**false**).



La syntaxe de cette commande est :

```
if condition:
    Instruction A
else:
    Instruction B
```

Exemple où la condition est vraie :

```
x = 5
if x > 0:
    print(x, "est positif")
else:
    print(x, "est négatif ou nul")
print("Fin")
```

Exemple où la condition est fausse :

```
x = -2
if x > 0:
    print(x, "est positif")
else:
```

```
    print(x, "est négatif ou nul")
print("Fin")
```

Une expression booléenne peut contenir les opérateurs de comparaison suivants :

Opérateur	**Signification**
$x == y$	x est égal à y
$x! = y$	x est différent de y
$x > y$	x est supérieur à y
$x < y$	x est inférieur à y
$x >= y$	x est supérieur ou égal à y
$x <= y$	x est inférieur ou égal à y

Attention cependant à la condition d'égalité qui risque de ne jamais être atteinte avec des nombres flottants (de type float). On préférera comparer les valeurs de deux nombres ou comparer leur différence à un nombre tout petit.

Important



Ne jamais tester l'égalité de deux flottants !!!

CHAPITRE 6

NOTION DE BOUCLE

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction ***while*** *puis nous aborderons l'instruction* ***for*** .

1. While

Il est souvent nécessaire de répéter plusieurs fois une même séquence d'instructions jusqu'à ce qu'une certaine condition soit satisfaite. Pour se faire, on utilise une boucle. Le mot-clé **while** (tant que) permet de réaliser cela. Vous trouverez, ci-dessous, un exemple complet de la structure d'une boucle.

```
instruction 1
while condition A : # se termine par :
    instruction 2 # indentation
    instruction 3 # indentation
instruction 4
```

Dans le programme ci-dessus, l'instruction 1 est exécutée puis les instructions 2 et 3 sont répétées tant que la condition A est vraie (**True**). Quand la condition A devient fausse (**False**), on passe à l'instruction 4. Si la condition A est fausse dès le départ, les instructions 2 et 3 ne sont jamais exécutées.

La ligne du bloc qui commence par **while** se termine par " :" et les lignes suivantes sont indentées.

Si on veut exécuter les instructions 2 et 3 quand la condition est fausse, on doit ajouter **not** après while dans l'algorithme ci-dessus et si la condition est vraie alors c'est l'instruction 4 qui sera seulement exécutée.

```
instruction 1
while True : # se termine par :
    instruction 2 #indentation
    instruction 3 #indentation
instruction 4
```

Application 12

À l'aide d'une boucle **while** , afficher la table de 12

2. For ... in ...

Une autre façon de faire une boucle est d'utiliser les mot-clés **for** et **in** quand on doit parcourir une séquence (chaîne de caractères ou liste). Cette structure de boucle évite de devoir définir et incrémenter un compteur, la variable créée juste après la commande **for** étant locale. La grande différence avec le mot-clé **while** réside dans le fait que le nombre d'itérations est prédéfini.

Voici des exemples pour mieux comprendre la syntaxe. Les possibilités sont nombreuses.

Exemple 1 : La table de 9

```
for i in range(11) :
    print(i," *9= ",9*i)
```

Ici, la fonction **range** crée un itérateur (ne stockant qu'un seul nombre en mémoire) dont les valeurs entière varient entre 0 et 10 inclus.

Exemple 2 : carré des 100 premiers nombres impairs

```
for i in range(100) :
    print((2*i+1)**2)
```

Application 13

À l'aide d'une boucle **for** , afficher la table de 122

CHAPITRE 7

LES MANIPULATIONS DE LISTES

1. Parcours séquentiel d'une liste

Au moins trois méthodes existent afin d'accéder à un élément d'une liste :

1 Boucle **for** avec range() et l'indexation :

```
for i in range(len(liste)):
    L[i]
```

2 Boucle **for** et le parcours sans utiliser d'index :

```
for el in L:
    el
```

3 Boucle **while** et l'indexation :

```
i=0
while i<len(L):
    L[i]
    i+=1
```

2. Opérations possibles

Initialisation d'une liste :	**L=[]**
Ajout d'un élément en fin de liste :	**liste.append(element)**
Extension d'une liste :	**liste.extend(liste2)**
Suppression du dernier élément de liste :	**liste.pop()**

Application 14

Soit la liste des résultats expérimentaux suivants :

$$L = [1.2, 3.5, 6.7]$$

1. Créer la liste L.
2. Supprimer le dernier élément de cette liste.
3. Ajouter la valeur 4,5 à la liste L de la question précédente.

Les listes peuvent assez facilement être manipulées.
Soit une liste :

```
liste = [1, 10, 100, 250, 500]
```

Ainsi la commande :

```
liste[-1]
```

renverra pour la valeur 500. La commande

```
liste[-4:]
```

ne renverra que les quatre dernières occurrences de la liste. La commande

```
liste[:2]
```

renverra les deux premières occurrences de la liste, c'est-à-dire 1 et 10. La commande

```
liste[3:]
```

renverra la liste « amputée » de ces trois premiers éléments.

3. Les listes en compréhension

C'est la méthode **la plus rapide** pour créer une liste : plus que les méthodes (par exemple avec append (ajouter en anglais)), qui étaient déjà plus rapides qu'une concaténation d'autres listes. Au final, construire une liste en compréhension permet de remplacer une ou plusieurs boucles **for** et l'évaluation d'une condition **if** par une seule ligne de code : celui-ci est rendu particulièrement compacte mais parfois peu lisible, ce qui n'est pas un avantage.

Les exemples sont parlants :

1

```
liste=[]
for i in range(10)
    liste.append(i)
```

```
liste=[i for i in range(10)]
```

En sortie, **IPython** affiche la liste des entiers de 0 (inclus) à 10 (exclu).

2

```
liste=[]
for i in range(20)
    if i%2==1:
        liste.append(i)
```

```
liste=[i for i in range(20) if i%2==1] # i modulo
   2=1
```

En sortie, **IPython** affiche la liste des entiers impairs de 0 (inclus) à 20 (exclu).

3

```
liste=[]
for i in range(20)
    liste.append((i+1)**2)
```

```
liste=[(i+1)**2 for i in range(20)
```

En sortie, **IPython** affiche la liste des carrés des entiers de 1 (inclus) à 20 (inclus).

4

```
liste=[]
for i in range(10)
    liste.append(rd.randint(1,6))
```

```
liste=[rd.randint(1,6) for i in range(10)
```

En sortie, **IPython** affiche la liste de 10 entiers aléatoires entre 1 (inclus) à 6 (inclus).

L'utilisation des **dataframe** de la librairie **pandas** peut s'avérer utile afin de d'afficher une liste :

```
Affichage_Tableau=pd.DataFrame({'liste des nombres':
   liste})
Affichage_Tableau_h=Affichage_Tableau.T  #
   Transposition du tableau à l'horizontale.
Affichage_Tableau_h
```

Application 15

Soit la liste de longueurs, L, exprimées en centimètre, suivante :

$$L = [0.20, 0.50, 0.70, 0.80, 2.05, 2.50]$$

1. Utiliser une **liste comprehension** permettant de convertir cette liste en une liste où les distances sont en mètre.
2. Afficher la liste des distances en centimètre ainsi que la liste des distances en mètre dans le même tableau **pandas** .

Application 16 — TS

Soit la liste des largeurs, a en μm, de fentes de diffraction.

$$a = [60, 25, 35, 90, 100, 250, 350]$$

1. Utiliser une **liste comprehension** permettant de convertir cette liste en une liste, nommée **inv_a** , contenant les valeurs de $\frac{1}{a}$ en m^{-1}.
2. Afficher la liste des largeurs a en μm ainsi que la liste des valeurs de $\frac{1}{a}$ en m^{-1} dans le même tableau **pandas** .

Application 17

Soit les listes des résultats expérimentaux suivantes :

$$\text{temps} = [0, 1, 2, 3, 4, 5, 6, 7, 8]$$

$$\text{tension} = [0, 0, 0, 0.5, 1, 1.2, 1.3, 1.35, 1.39]$$

1. Créer les listes temps et tension.
2. Créer un script permettant de ne garder que les valeurs des listes temps et tension pour lesquelles les valeurs de tension sont supérieures à 0.2.
3. Modifier les deux listes précédentes pour que n'y figurent que les valeurs pour lesquelles les valeurs de la tension sont supérieures à 0.2.
4. À l'aide d'une liste en comprehension, modifier la liste temps pour que le temps soit translaté du temps pour lequel la condition précédente est vérifiée.

CHAPITRE 8

LES FONCTIONS

Lorsque l' on écrit des programmes, nous sommes souvent amenés à répéter plusieurs fois une même séquence d' instructions. C' est fastidieux, surtout que des erreurs ne manquent pas de se produire. Pour cela, le langage dispose d' une syntaxe qui permet de regrouper une séquence et de la désigner par un mot unique, une ***fonction*** *. Une fonction est une portion de code réutilisable que l' on va regrouper pour lui donner un nom. On pourra exécuter de nouveau cette portion en appelant simplement le nom de la fonction et ses arguments.*

1. Syntaxe de déclaration de fonction

Voici la syntaxe de déclaration d' une fonction

```
def nom_de_Fonction (parametre1, parametre2, ...) :
    #documentation de la fonction
    instruction 1
    instruction 2
    ...
    return (parametreDeSortie1, parametreDeSortie2,
   ...)
```

La documentation est facultative mais vraiment conseillée, **return** peut aussi être absent, ce que l' on appelle parfois une procédure.

Après avoir exécuter l' instruction **return** , le programme sort du bloc de la fonction : aucune autre instruction de la fonction ne sera exécutée après **return** .

Remarque fondamentale : **return** renvoie un objet (entier, flottant, ...,ou une collection comme un tuple, une liste), mais ne l' affiche pas (c' est le rôle de **print**). En revanche, un autre programme peut utiliser le contenu renvoyé par **return** , ce qui n' est pas possible avec **print** .

Important



Ne confondez pas les deux termes et surtout leurs utilisations !

2. Méthode pour écrire une fonction

Généralités

On commence par identifier un besoin récurrent dans les taches à réaliser ou bien on souhaite structurer son programme en sous-problèmes bien identifiés.

1. On répond à la question : "Sur quelles données (variables) travaille l' algorithme de résolution ? "

   On obtient les paramètres d' entrée de la fonction.

   Ensuite, on démarre par le mot-clé **def** suivi du nom explicite de la fonction, des parenthèses contenant les paramètres d' entrée, puis un "deux-points (:)".

2. On répond à la question : "Que dois-je renvoyer pour répondre à la question ? "

   On obtient la sortie à renvoyer par la fonction.

   Ensuite, on place le mot-clé return suivie de la variable contenant la réponse au problème de départ (cette réponse = la sortie).

3. On écrit l' algorithme, que l' on traduit en un script python. Le script python du programme est indenté (tabulation), ce qui assure la reconnaissance du début et de la fin du bloc d' instructions appartenant à la fonction.

Exemple

On souhaite calculer la valeur moyenne de la liste de nombres suivante :

$$[17, 25, 54, 75, 88, 21, 147, 19]$$

Identification du besoin : "moyenne d' une série de nombres"

1. Les données de départ sont une série de valeurs : il n' y a aucun autre paramètre d' entrée. On peut décider de les placer dans un tuple (si on ne prévoit aucune modification) ou une liste (en cas de modification à venir). On code en Python :

```
def moyenne(liste) :
```

2. La fonction répondra correctement à la question si elle renvoie un nombre qui est la moyenne de la série de valeurs. Ainsi chaque utilisateur de la fonction fournira sa liste de valeurs, la fonction renverra la moyenne de ces valeurs stockée dans la variable m. On code en Python :

```
    return m
```

3. Algorithme :

somme ← 0

pour i **allant de** 0 à nombre_ valeurs - 1 faire :

somme ← somme + valeurs[i]

m ← $\frac{\text{somme}}{\text{nombre_valeurs}}$

Traduction en script Python de cet algorithme :

```
somme=0
for i in range(len(liste)):
    somme = somme + liste[i]
    m=somme/len(liste)
```

3. Procédure

Définition

Une procédure est une fonction qui ne renvoie rien en retour.
Il n' y aura donc pas de **return** dans la définition de ces fonctions.

Exemples

Imaginons que l' on souhaite formater un ensemble de noms de famille entrés par différents utilisateurs. On souhaite obtenir des noms commençant par une majuscule, le reste en minuscule.

En entrée, nous avons besoin d' une chaine de caractère. En sortie nous voulons renvoyer l' affichage de cette chaine mais pas la chaine elle-même en tant qu' objet **Python** de type chaine.

En ce sens là, la fonction ne renvoie pas d' objet typé mais seulement un affichage : il n' y a pas lieu d' utiliser "return".

Sans utiliser de fonction :

```
nom="mArtin"

if str(nom[0]).isupper(): #on vérifie si la première
    lettre est une majuscule
    print(nom)
else:
    nom=nom.capitalize() #majuscule sur la première
    lettre du mot
    print(nom)
```

En utilisant une fonction :

```
nom="mArtin"

def majuscule_premiere_lettre(nom)
    if str(nom[0]).isupper(): #on vérifie si la premiè
    re lettre est une majuscule
        print(nom)
    else:
        nom=nom.capitalize() #majuscule sur la premiè
    re lettre du mot
        print(nom)
```

Voici maintenant un exemple de procédure permettant d' afficher une table de multiplication, avec comme arguments un nombre de départ, un de fin et la base.

```
def table(base,debut,fin):
    # Affiche la table de multiplication de <base> de
    <début> à <fin>
    n=debut
    while n<=fin:
        print(n,' *' ,base,' =' ,n*base)
        n+=1
```

Si vous cherchez de la documentation sur **IPython** concernant la procédure ou la fonction que vous venez de créer, passez par la commande **?table** ou **table ?** ou encore **help(table)** .

4. Appel de fonction

Définir et *exécuter* une fonction sont deux choses différentes. On ne peut exécuter une fonction que si elle a été définie auparavant. Pour ce faire, il faudra appeler la fonction, pour faire simple, donner son nom et lui fournir entre parenthèses les paramètres nécessaires à son fonctionnement.

Voici un exemple extrêmement simple, le calcul du volume d' une sphère $\frac{4\pi}{3}R^3$:

```python
def volume_sphere(R):
    return 4 / 3 * np.pi * R**3

volume_sphere(7.8)
```

5. Fonction avec un nombre indéterminé de paramètres

Il suffit d' utiliser pour un nombre quelconque d' arguments le terme ***args** , placé derrière tous les autres arguments de la fonction.

Cherchons à calculer la somme d' une série de nombres. Nous ne savons pas à l' avance combien de nombres seront à sommer.

```python
def somme(*arg):
    somme=0  #initialisation
    for i in args:  # parcours des arguments
        somme=somme+i  #construction de la spmme
    return somme

print(somme(2,3,5))

print(somme(1,2,3,5,7,8,9))
```

6. Les fonctions lambda

Python permet une syntaxe intéressante qui vous laisse définir des mini-fonctions d'une ligne à la volée. Ces fonctions dites **lambda** peuvent être employées partout où une fonction est nécessaire. Voici un exemple de fonction ainsi définie :

```python
g = lambda x: 2*x
```

Cette fonction est équivalente à la fonction h :

```python
h = lambda x: 2*x
```

Plus généralement, une fonction **lambda** est une fonction qui prend un nombre quelconque d'arguments (y compris des arguments optionnels) et retourne la valeur d'une expression unique. Les fonctions **lambda** ne peuvent pas contenir de commandes et elles ne peuvent contenir plus d'une expression.

Les fonctions **lambda** sont une question de style. Les utiliser n'est jamais une nécessité, partout où vous pouvez les utiliser, vous pouvez utiliser une fonction ordinaire. Elles seront utilisées là où on souhaite incorporer du code spécifique et non réutilisable sans encombrer mon code de multiples fonctions d'une seule ligne.

Application 18

Définir une fonction qui calcule la surface d'une sphère $4\pi R^2$, de rayon R donné, dans un premier temps avec le mot-clé **def** puis avec le mot-clé **lambda** .

Application 19 — Enseignement scientifique

L'objectif est de simuler un lancé de dé grâce à la fonction **randint** du module **random** .
Pour cela, vous définirez une fonction **lancer** qui prendra comme argument le nombre de lancés et qui retournera une liste des lancés.

CHAPITRE 9

FICHIER : LECTURE ET ÉCRITURE

De nombreuses applications sur ordinateur, tablette ou smartphone permettent l'acquisition d'un grand nombre de grandeurs physiques (température, accélération, champ magnétique, ...). Les fichiers d'export sont la plupart du temps au format ***csv*** *(**c**omma-**s**eparated **v**alues, données tabulaires (tableau) sous forme de valeurs numériques séparées par des virgules ou des points virgules) ou* ***txt*** *. Nous allons donc nous intéresser au traitement de ces fichiers par* ***Python*** *.*

1. Ouverture et fermeture

L'ouverture et la fermeture de fichier s'effectue avec la commande suivante :

```
variable=open("chemin_complet","mode en ouverture")
```

```
variable.close()
```

mode	effet
'r'	mode lecture seule
'w'	mode écriture, fichier créé sinon écrasé
'a'	fichier créé sinon données ajoutées en fin de fichier

La première chose que l'on va faire sera de créer un fichier qui n'existe pas encore, comme cela il sera placé dans le même dossier que le programme **Python** .

```
objet=open('objet.txt','w')
objet.close()
```

Quand vous exécutez ces deux lignes, un fichier texte apparaît. Si on l'ouvre avec un éditeur de texte, on constate qu'il est vide. Maintenant on va le remplir en l'ouvrant en mode "a", on veut ajouter des données et non écraser ce qui existe même si il n'y a rien pour l'instant.

```
objet=open('objet.txt','a')
objet.write('Lampe')
objet.close()
```

Quand on va voir ce que contient notre fichier avec un éditeur de texte, on remarque que le mot "Lampe" est bien écrit. Si on veut s'éviter la peine d'écrire cette dernière ligne avec le close , on peut adopter une autre syntaxe :

```
with open('objet.txt','a') as objet:
objet.write('Table')
```

Et là, avec l'éditeur de texte, on voit que le nouveau mot est bien ajouté mais qu'il est collé au premier. On va ajouter avec l'éditeur une deuxième ligne dans le fichier "objet.txt" qui contient un seul mot : **Chaise** . On enregistre et on referme le fichier. On passe à la lecture.

2. Lecture et écriture

On ouvre et on lit le fichier avec Python très facilement :

```
with open('objet.txt','r') as text: # 'r' pour lecture
print(text.readlines())
```

La méthode **readlines** appliquée à l'objet nommé text, permet de lire tout le fichier. Dans la console, on peut lire quelque chose comme :

```
['LampeTable\n', 'Chaise']
```

On remarque dans l'élément zéro, un **\n** . C'est un caractère invisible pour un éditeur de texte qui représente un saut de ligne. Donc en **Python** , si on veut écrire une nouvelle ligne dans un fichier, nous devons taper :

```
with open('objet.txt','a') as objet:
objet.write('Table')
```

Il existe également une autre méthode de lecture qui s'appelle **readline** . C'est une méthode pas à pas, une seule ligne est lue à chaque fois.

```
with open('objet.txt','r') as text: # 'r' pour lecture
print(text.readline())
```

Si on veut lire les trois lignes :

```
with open('objet.txt','r') as text: # 'r' pour lecture
for i in range(3):
print('Sur la ligne {} on peut lire {}'.format(i,text.
    readline()))
```

On remarque que chaque ligne est considérée comme une chaîne de caractère, le **\n** n'est plus affiché mais il est exécuté car dans la console il y a un espace d'une ligne entre chaque ligne.

3. Lecture de fichiers avec pandas

La librairie **pandas** fait partie des librairies de base pour la data science en Python. Elle permet de manipuler facilement des données à analyser.

L'objet qui permet de représenter des « tableaux » est l'objet **DataFrame** .

Lorsque que vous souhaitez analyser des données expérimentales, il est primordiale de visualiser l'allure du fichier. Par exemple :

```
"T(s)";"X(m)";"Y(m)"
```

```
"0,000";"2,79E-1";"1,16E+0"
"0,040";"3,72E-1";"1,28E+0"
```

Attention au format des valeurs qui doit être en décimale (la virgule doit être un point). Le séparateur entre deux variables ne doit donc pas être une virgule. Le fichier précédent peut alors être modifié pour devenir :

```
T(s);X(m);Y(m)
0.000;2.79E-1;1.16E+0
0.040;3.72E-1;1.28E+0
```

Dans le script suivant, le dataframe data stocke les valeurs de la fonction **pd.read_csv** prenant comme argument l'adresse du fichier de données et le séparateur utilisé.

```
import pandas as pd
# Charge les données du fichier CSV dans un dataframe
# avec comme séparateur le ";" ou la ","
# data est le nom de l'objet de type data frame créé
# data = pd.read_csv('nomfichier.csv',sep = ',')
url = 'ParaboleAvimeca.csv'
data = pd.read_csv(url,sep = ';') #Lecture du fichier
    csv
```

Le script suivant permet de convertir les listes de caractères valeurs numériques en sans avoir besoin de les supprimer manuellement dans le fichier initial.

```
xL=data["X(m)"].values.tolist()
yL=data["Y(m)"].values.tolist()
```

Quant au au script suivant, il permet de convertir via une liste comprehension les valeurs en nombre réels :

```
x = [float(item) for item in xL]
y = [float(item) for item in yL]
```

Application 20 — Seconde

À l'aide du fichier **acquisition_ mt_ initial.csv** réaliser un script permettant d'afficher la trajectoire du mobile.

4. Lecture de fichiers avec la librairie csv

La librairie **csv** peut aussi être utilisée assez facilement mais son usage est moins répandu que la librairie **pandas** . Le principe de l'importation consiste à créer une fonction (**LectureFichier**) qui va permettre de stocker les valeurs numériques dans des listes (ici **colonne**).

Le fichier **txt** ou **csv** est parcouru, et à chaque nouvelle ligne du fichier, une nouvelle valeur est rajoutée dans la liste créée via la commande **append** . Une conversion en nombre flottant est alors nécessaire.

```
import csv #importation du module csv


'''Définition de la fonction LectureFichier permettant
    d'extraire les données issues d'Avimeca'''

def LectureFichier(fichier,separateur,n):
    file = open(fichier,"r") #ouverture du fichier en
   mode lecture
    reader = csv.reader(file,delimiter=separateur)
    colonne = [] #création d'une liste colonne qui
   contiendra les données d'une colonne du fichier CSV
    for row in reader:
        try: #assertion qui permet de mettre les
```

```
    valeurs numériques au bon format
            notation_point = row[n].replace(",",".") #
    remplace la virgule par un point
            colonne.append(float(notation_point)) #
    ajoute la donnée lue dans la liste colonne en tant
    que nombre et non plus comme chaîne de caractère
         except: # Rien ne doit se passer en cas d'
    erreur
            pass
     file.close()
     return colonne

''' Extraction des dates, abscisses et ordonnées du
    fichier généré par aviméca'''

temps = LectureFichier("ParaboleAvimeca.txt",";",0) #
    0 correspond à la première colonne
abscisses = LectureFichier("ParaboleAvimeca.txt",";"
    ,1) # 1 correspond à la colonne 2
ordonnees = LectureFichier("ParaboleAvimeca.txt",";"
    ,2) # 1 correspond à la colonne 3
```

CHAPITRE 10

LES VECTEURS

Il existe plusieurs modules qui permettent de tracer des vecteurs. Dans un premier temps, nous utiliserons la module ***matplotlib.pyplot*** *qui possède la fonction* ***arrow*** *. Puis, nous utiliserons* ***quiver*** *.*

1. plt.arrow

Pour représenter un vecteur, il suffit d'appeler la fonction **plt.arrow** qui a comme arguments :

- l'abscisse du point d'application,
- l'ordonnée du point d'application,
- l'abscisse du vecteur,
- l'ordonnée du vecteur,
- options éventuelles.

```
import matplotlib.pyplot as plt

#nom de la figure
plt.figure(1)
plt.clf()

#définition des axes
ax = plt.axes()
#tracé du vecteur
plt.arrow(2.0, 4.0, 6.0, 4.0, head_width=0.5, head_
   length=0.7, fc='lightblue', ec='black')
plt.grid()
plt.xlim(0,10)
plt.ylim(0,10)

plt.title('Comment tracer un vecteur avec Matplotlib?'
   ,fontsize=10)

plt.show()
```

La représentation des vecteurs va surtout nous servir en Physique-Chimie pour tracer des vecteurs vitesse et accélération.

Le script suivant exploite une chronophotographie fournie dans le fichier **ParaboleAvimeca.csv** , utilisée au chapitre précédent :

Le vecteur vitesse $\overrightarrow{v}$ sera approché à chaque instant par le vecteur

vitesse moyenne :

$$\overrightarrow{v}_i = \frac{\overrightarrow{OM_{i+1}} - \overrightarrow{OM_i}}{\Delta t} = \frac{\overrightarrow{M_i M_{i+1}}}{\Delta t}$$

et l'accélération $\overrightarrow{a}$ par :

$$\overrightarrow{a}_i = \frac{\overrightarrow{v_{i+1}} - \overrightarrow{v_i}}{\Delta t}$$

```python
#importation fichier dans un dataframe
import pandas as pd
url = 'ParaboleAvimeca.csv'
data = pd.read_csv(url,sep = ';') #Lecture du fichier
    csv
#conversion des listes en valeurs numériques
tL=data["T(s)"].values.tolist()
xL=data["X(m)"].values.tolist()
yL=data["Y(m)"].values.tolist()
#Conversion en nombres réels
temps = [float(item) for item in tL]
abscisses = [float(item) for item in xL]
ordonnees = [float(item) for item in yL]

#Nombre de points contenus dans l'acquisition AVIMECA
N_points=len(temps)
#Intervalle de temps entre chaque image
delta_t = temps[1]-temps[0]

#Calcul des coordonnées de la vitesse entre le premier
    et l'avant dernier point
vx = []
vy = []
for i in range(0,N_points-1):
    vx.append((abscisses[i+1]-abscisses[i])/delta_t)
    vy.append((ordonnees[i+1]-ordonnees[i])/delta_t)

#Calcul de l'accélération entre le deuxième et l'avant
    avant dernier point
ax = []
ay = []
for i in range(1,N_points-2):
     ax.append((vx[i+1]-vx[i])/delta_t)
     ay.append((vy[i+1]-vy[i])/delta_t)

#Echelles vitesse, variation de vitesse et accélé
   ration
Echelle_Vitesse = 0.15
Echelle_Acceleration = 0.01

#Tracé de la trajectoire
import matplotlib.pyplot as plt
#Création d'une fenêtre graphique
plt.figure(1)
#Nettoyage de la fenêtre
plt.clf()
#Paramétrages du graphique - optimisation des échelles
plt.grid()
plt.xlim(0,1.5 * max(abscisses))
plt.ylim(-1, 1.5 * max(ordonnees))
#Création du graphique
plt.plot(abscisses,ordonnees,'r+')
#Vecteur vitesse
for i in range(1,N_points,2): #Tracé du vecteur
   vitesse tous les deux points
    plt.arrow(abscisses[i],ordonnees[i],Echelle_
   Vitesse * vx[i],Echelle_Vitesse * vy[i],color='r',
   head_width=0.05)
#Vecteur accélération
for i in range(2,N_points-1,2): #Tracé du vecteur accé
   lération tous les deux points
      plt.arrow(abscisses[i],ordonnees[i],Echelle_
   Acceleration*ax[i-2],Echelle_Acceleration*ay[i-2],
   color='b',head_width=0.05)
#Affichage du graphique
plt.show()
```

Application 21 — Seconde

À l'aide du script précédent :

1. Afficher le vecteur accélération $\overrightarrow{a}$ tous les 4 points.
2. Afficher le vecteur variation du vecteur vitesse $\Delta\overrightarrow{v}$ en rouge tous les trois points.

2. plt.quiver

La fonction **quiver** peut s'avérer plus intéressante que **arrow** utilisée précédemment lorsque le repère n'est pas orthonormé , les vecteurs représentés se trouvent déformés.

Pour tracer des vecteurs, on peut dessiner des flèches en tapant :

```
plt.quiver(x0, y0, deltax, deltay, angles='xy')
```

La commande précédente dessinera une flèche avec les caractéristiques suivantes :

— x0 et y0 sont les coordonnées du point de départ

— deltax et deltay sont les composantes du vecteur selon l'abscisse et l'ordonnée.

— angles='xy' permet d'obtenir une flèche qui a une direction cohérente avec l'échelle des axes. Donc c'est un paramètre à utiliser systématiquement en l'écrivant toujours tel quel.

Si à l'affichage, la taille de la flèche est trop faible ou trop importante, il faudra rajouter deux paramètres :

```
plt.quiver(x0, y0, deltax, deltay, angles='xy', scale
    =1, scale_units='xy')
```

Si vous voulez agrandir la flèche choisissez une valeur de scale inférieure à 1. Si vous souhaitez rétrécir la flèche, choisissez une valeur de scale supérieure à 1.

Voici un exemple de champ de vecteurs, en chaque point de l'espace, un vecteur (U, V) de coordonnées proportionnelles aux coordonnées du point d'application (X, Y) est créé.

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

#création de deux variables X et Y
X = np.linspace(-10, 10, 21)
Y = np.linspace(-10, 10, 21)

#Attribution pour chaque point de l'espace d'une
# abscisse U et d une ordonnée V pour le champ de
    vecteur
U, V = np.meshgrid(X, Y)

#nom de la figure
plt.figure(1)
plt.clf()
fig, ax = plt.subplots()
q = ax.quiver(X, Y, U, V)
ax.quiverkey(q, X=0.3, Y=1.1, U=10,
             label='Quiver key, length = 10', labelpos
    ='E')

plt.show()
```

Voici un autre exemple, cette fois-ci une valeur Z est attribuée à chaque point de l'espace Y, Y.

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

```
x = np.linspace(-3, 3, 51)
y = np.linspace(-2, 2, 41)
X, Y = np.meshgrid(x, y)

# calcul du tableau des valeurs de Z
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

#nom de la figure
plt.figure(1)
plt.clf()
plt.pcolor(X, Y, Z)
plt.colorbar()

plt.show()
```

Visualisation de la fonction sinus cardinal $\frac{\sin(x)}{x}$. Afin pour lever l'indétermination de la division $\frac{0}{0}$, nous allons utiliser la constante prédéfinie **finfo(float).eps** .

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

epsilon = np.finfo(float).eps
print("epsilon =", epsilon)

xmin = -8
xmax = 8
nbx = 41
ymin = -8
ymax = 8
nby = 41

x = np.linspace(xmin, xmax, nbx)
y = np.linspace(ymin, ymax, nby)
X, Y = np.meshgrid(x, y)

R = np.sqrt(X**2 + Y**2) + epsilon
Z = np.sin(R) / R

plt.imshow(Z, interpolation="bicubic",
       origin="lower", extent=[xmin,xmax,ymin,ymax])
plt.colorbar()

plt.show()
```

Application 22 — Seconde

Voici les positions d'un mobile obtenues lors d'un TP, l'intervalle de temps entre chaque image vaut $\Delta t = 40\,\text{ms}$:

Sur l'axe Ox : 0,2,4,6,8,10,12,14,16,18,20
Sur l'axe Oy : 0,1,3,6,10,15,21,28,36,45,55

Visualiser les vecteurs-vitesses.

Éléments

CHAPITRE 11

TRAITEMENT DE DONNÉES

Le traitement statistiques de données expérimentales fait partie intégrante du travail de l'expérimentateur. Dans un premier temps, nous utiliserons le module ***nympy*** *qui propose de nombreuses fonctions pour le traitement de données. La fonction* ***polyfit*** *de ce module permettra d'effectuer une régression polynomiale. Ensuite, nous utiliserons la fonction* ***linregress*** *du module* ***scipy.stats*** *qui permet d'obtenir tous les paramètres statistiques lors de régressions linéaires. Enfin, la modélisation d'une fonction par une exponentielle sera abordée.*

1. Traitement statistique avec numpy

Variables statistiques

Imaginons une liste de valeurs numériques :

$$L = [20.5, 19.6, 20.4, 19.25, 19, 20, 20.75, 20.5]$$

Afin de connaître la moyenne, m, il suffit d'appeler la fonction **average** du module **numpy** . L'écart-type, σ, quant à lui s'obtient grâce à la fonction **nanstd** . Il suffit alors d'imprimer ses valeurs pour les afficher.

La fonction **around** permet d'arrondir un nombre, **decimals** précisant le nombre de décimales de ce nombre.

```python
import numpy as np
L=[20.5,19.6,20.4,19.25,19,20,20.75,20.5]

#exploitation statistique valeurs expérimentales
m=np.around(np.average(L),decimals=2)
ecarttype=np.around(np.std(L),decimals=2)

#impression des valeurs
print(m)
print(ecarttype)
```

moyenne	**average**
écart-type	**nanstd**
arrondi	**around**
minimum et maximum	**min** et **max**

Application 23 — Terminale Spé

Un émetteur et un récepteur d'ultrasons sont placés à $d = 30\,\text{cm}$ l'un de l'autre. Le retard temporel $\Delta t(\mu s)$ est mesuré plusieurs fois. Les valeurs suivantes sont trouvées :

905,870,910,880,890,915,900,885

1. Afficher la moyenne, m, et l'écart-type, σ, de cette liste de valeurs.
2. Afficher l'incertitude élargie, n étant le nombre de mesures.

$$U(\Delta t) = 2 \times \frac{\sigma}{\sqrt{n}}$$

3. En déduire une valeur expérimentale de la célérité des ultrasons dans l'air :

$$c_{\text{us, air}} = \ldots \pm \ldots$$

Régression avec polyfit

Les régressions linéaires peuvent être utilisées dès la classe de seconde : *« Représenter un nuage de points associé à la caractéristique d'un dipôle et modéliser la caractéristique de ce dipôle à l'aide d'un langage de programmation »*. En choisissant la valeur 1 (modélisation par un pôlynome d'ordre 1) comme troisième argument de la fonction **polyfit** , on obtient une régression linéaire qui permet, par exemple, de modéliser la caractéristique d'un conducteur ohmique.

Imaginons que I soit la liste des intensités qui traversent un conducteur ohmique et U la liste des tensions entre les bornes de ce même dipôle.

```
import numpy as np
coeff = np.polyfit(I, U, 1) # régression linéaire
```

$coeff$ est à présent une liste. La droite modélisée a pour équation : $coeff[0] \times x + coeff[1]$.

Attention, $coeff[0]$ est bien le coefficient pour la puissance la plus élevée du polynôme.

Donc pour tracer cette droite modélisée, il suffit de taper :

```
plt.plot([0, 5], [coeff[1], 5*coeff[0]+coeff[1] ])
```

Application 24 — Seconde

Dans une application précédente, nous avions tracé la caractéristique, $U = f(I)$, d'un dipôle dont on a mesuré la tension, U, à ses bornes en faisant varier l'intensité du courant, I, qui le traverse.
I(A) : 0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09
U(V) : 0.68,1.37,2.03,2.73,3.50,4.07,4.79,5.30,6.15

1. **Faire la régression linéaire de la caractéristique.**
2. À l'aide de la fonction **annotate** , afficher sur le graphique les paramètres obtenus après modélisation.

2. Régression avec scipy et la fonction linregress

Pour effectuer des régressions, la fonction **linregress** du module **scipy** (comme scientific Python) peut-être aussi utiliser. Elle permet de stocker dans une liste cinq valeurs :

— **slope** ou pente de la régression,
— **intercept** correspondant à l'ordonnée à l'origine,
— **rvalue**, cœfficient de détermination (qualifiant la qualité de la détermination). Plus ce cœfficient est proche de l'unité plus la détermination est proche de la réalité,
— **pvalue** correspondant à la probabilité pour une valeur expérimentale d'être en dehors de la plage de mesures,
— **stderr**, la déviation standard ou écart-type, σ.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import linregress as lr

#Les données

x=[0.4,0.7,0.9,1.2,1.5]
y=[1.2,2.1,3.1,4.2,4.9]
 #Conversion entableau numpy
x=np.array(x)
#Graphique des données
plt.figure()
plt.clf()
plt.plot(x,y, 'bx',markersize=5)
plt.xlabel('x')
plt.ylabel('y')

#Régression linéaire
resultat=lr(x,y)
#print(resultat)
a,b,coeffCorr,pValue,stdDev=resultat
plt.plot(x,a*x+b,'g-',markersize=1)

plt.show()

#Analyse des résultats
print("le coefficient de corrélation vaut %f" %
    coeffCorr)
print("Le z-score vaut %f" %pValue)
```

Cette fonction nous donnera les données statistiques essentielles de la régression réalisée.

Application 25

Reprendre l'application précédente en utilisant la fonction **linregress** du module **scipy.stats**.

3. Modélisation par une fonction quelconque avec scipy.optimize

Le module **curve**$_{f}$*itdumodule***optimize***delalibrairie***scipy***permetdemodélise*

```
import numpy as np
def exponential_func(x, a, b, c):
    return a*np.exp(-b*x)+c
```

Ensuite il s'agît de récupérer les résultats de la modélisation dans les variables popt (paramètre(s) optimisé(s)) et pcov (paramètre de covariance) :

```
from scipy.optimize import curve_fit

x = [30,40,50,60]
y = [0.027679854,0.055639098,0.114814815,0.240740741]

popt, pcov = curve_fit(exponential_func, x, y, p0=(1,
    1e-6, 1))
```

p0 indique les paramètres initiaux de modélisation. Ses valeurs vont guider l'optimisation. Par défaut elles sont fixées à une liste de 1.

On peut ensuite représenter la courbe de y en fonction de x et sa modélisation :

```
import matplotlib.pyplot as plt

xx = np.linspace(0.8*min(x),1.2*max(x),1000)
yy = exponential_func(xx, *popt)

plt.figure(1,figsize=(10,7))
plt.plot(xx,yy)
plt.plot(x,y,'o')
plt.show()
```

CHAPITRE 12

ANIMATIONS

Une des activités de la classe de Première consiste à « Simuler à l'aide d'un langage de programmation, la propagation d'une onde périodique ». Il faut donc faire une animation à l'écran. Pour cela, il existe des fonctions de matplotlib destinées aux animations mais elles ne sont pas toujours simples à utiliser.

Heureusement, matplotlib permet une autre technique plus rudimentaire. Elle fonctionne bien dans des situations simples. Elle consiste à dessiner la figure, faire une pause puis effacer l'image. Il faut ensuite dessiner la figure suivante, faire une pause et l'effacer à nouveau. Et ainsi de suite... Pour cela, on utilise les fonctions :

```
plt.clf() # cette fonction efface l'image

#dessin de l'image

plt.pause(0.01) #pause de 0,01 seconde
```

1. Un premier exemple

Voici un script permettant de dessiner une flèche qui tourne en recopiant le code suivant :

```
import matplotlib.pyplot as plt
import numpy as np
for i in range(100):
    plt.clf() #efface l'image
    plt.axis([-5, 5, -5, 5]) #dimensions de la fenêtre
    graphique
    plt.quiver(0, 0, 4*np.cos(i/10), 4*np.sin(-i/10),
angles='xy', scale=1, scale_units='xy')
    plt.pause(0.01) #valeur de la pause
plt.show()
```

Application 26 — **Première Spé**

Dans le script précédent, mettre la ligne plt.clf() en commentaire pour la désactiver et observer la différence.

2. Animations plus élaborées

Voici maintenant un exemple d'animation attendue pour le niveau Première : *« Représenter un signal périodique et illustrer l'influence de ses caractéristiques (période, amplitude) sur sa représentation. Simuler à l'aide d'un langage de programmation, la propagation d'une onde périodique »*.

Il y a de nombreuses méthodes pour faire cet exercice mais nous allons faire ici une des plus courtes à taper. Commençons par faire les importations et les déclarations :

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

A = 1
longueurOnde = 300
#Attention, le mot  lambda est déjà utilisé par Python donc on ne peut pas
# appeler une variable lambda mais on peut utiliser le symbole
Phi = 3.14
v = 40
T = longueurOnde/v
```

Nous allons faire avancer le temps grâce à une boucle `for` et à chaque temps, nous allons calculer les ordonnées de la courbe pour toutes les abscisses allant de 0 à $x_{\max} = v \times t$. Autrement dit pour chaque temps, on doit calculer en priorité jusqu'où l'onde s'est propagée.

```
for t in range(200): #le temps va évoluer de 0 à 199
    plt.clf()
    x_max = v*t #on calcule l'abscisse de la propagation
    y = [] #on initialise la liste des ordonnées
    x = [] #pareil pour les abscisses
    for xi in range(0, x_max):
        x.append(xi) # on ajoute xi à la liste des abscisses
        y.append(A*np.sin((2*3.14/longueurOnde)*xi-(2*3.14/T)*t+Phi)) # dans la ligne précédente, on calcule l'ordonnée en xi
    x.append(1500) # on rajoute le point 1500,0 pour afficher un trait où l'onde ne s'est pas encore propagée
    y.append(0)
    plt.axis([-1, 1500, -2, 2])
    plt.plot(x, y, 'r-')
    plt.pause(0.01)
plt.show()
```

Application 27 — Première Spé

1. À l'aide du programme précédent, faire varier l'amplitude et observer l'onde obtenue.
2. Faire de même en modifiant la célérité de l'onde.

Application 28 — Seconde

Tracer la trajectoire d'un mobile ayant un mouvement rectiligne uniforme puis accéléré.

3. Construire des fichiers gif animés

Une autre possibilité est de sauvegarder une image à chaque tour de boucle puis de construire un fichier gif . Pour cette dernière étape, il existe de nombreuses possibilités. En voici une

utilisant les modules os et imageio .

```python
import os
import imageio

png_dir = 'chemin_d_acces'
images = []
for file_name in os.listdir(png_dir):
    if file_name.endswith('.png'):
        file_path = os.path.join(png_dir, file_name)
        images.append(imageio.imread(file_path))
imageio.mimsave('nom_du_fichier.gif', images, fps=1)
```

Prenons un exemple :

Soit un ballon lancé avec une vitesse initiale, v_0 faisant un angle α avec l'horizontale et d'une hauteur h, dans le champ de pesanteur de la Terre.

À partir du script suivant :

```python
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np

#déclaration des variables et de leur valeur initiale
g=9.81 #m.s-2
v=11 #m/s
alpha=62 #degre
x0=0 #m
z0=2 #m
tau=0.1

distance_panier=10 #m
hauteur_panier=3.05 #m
x=x0
z=z0 #m ainsi z pourra être traité comme une variable
   (au sens mathématique) et
#z0 reste constant, ce qui permet de l'utiliser pour
   le tracé de la courbe de référence

t=0 #s
v0x=v*np.cos(alpha*np.pi/180)
v0z=v*np.sin(alpha*np.pi/180)

#x et z sont des variables de type float, qu'il faut
   stocker dans
#des listes si on veut effectuer des traces
xlist=[x]
zlist=[z]


i=0 #compteur
vz=v0z

#Calcul pas à pas des valeurs prises par t, x, z, vz (
   vx est constante)...
while z>=0:
    x = x+v0x*tau
    z = z+vz*tau
    t = t+tau
    vz = vz-g*tau

#... et stockage des valeurs calculées dans les listes
    créées
    xlist.append(x)
    zlist.append(z)
    i=i+1

#graphique
plt.figure(1)
plt.clf()
plt.xlim(0,12)
plt.ylim(0,8)
plt.plot(xlist, zlist, 'r.')
plt.xlabel(r'$x$(m)')
plt.ylabel(r'$z$(m)')
```

```
plt.grid()

plt.gca().add_patch(Rectangle((distance_panier,
    hauteur_panier), 0.5, 0.1,fc='r')) #trace du panier
plt.gca().add_patch(Rectangle((distance_panier+0.5, 0)
    , 0.2, 3.5,fc='r'))
plt.annotate("panier", xy=(distance_panier+0.25,
    hauteur_panier+0.1), xycoords='data', #trace de la
    flèche du panier
             xytext=(distance_panier-2, hauteur_panier
    +3), textcoords='data',
             arrowprops=dict(arrowstyle="->",),)
#legende
plt.text(11,7,r'$\alpha=$'+str(alpha)+"degres")
plt.text(11,6,r'$v_0=$'+str(v)+"$m/s$")
plt.text(11,5,r'$h=$'+str(z0)+"m")


plt.show()
```

Application 29 — Seconde

À l'aide du script précédent :

1. **À l'aide de la commande plt.pause , afficher une animation de la trajectoire du ballon.**
2. **Créer un gif animé de cette trajectoire.**

La commande suivante permet de supprimer les fichiers .png créés.

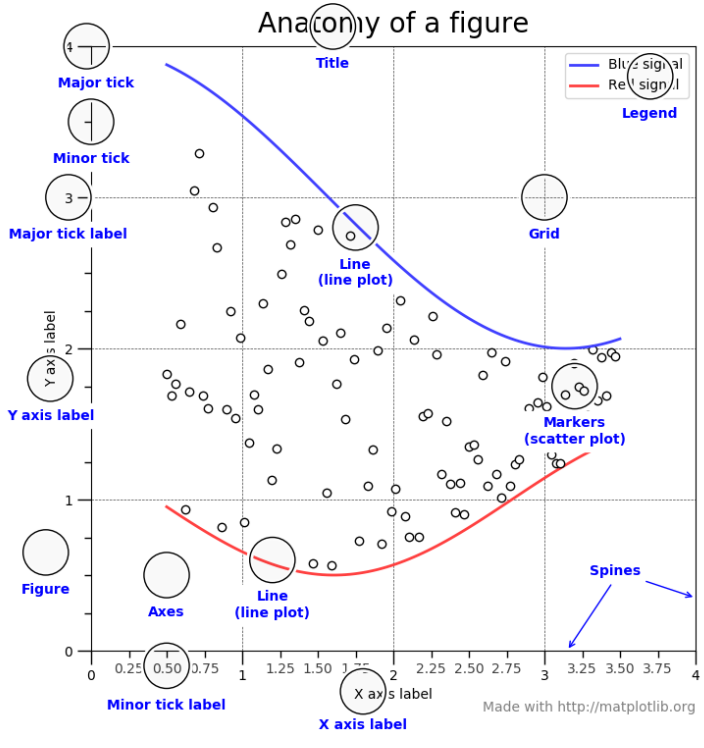
```
for file_name in os.listdir(png_dir):
    if file_name.endswith('.png'):
        os.remove(file_name)
```

CHAPITRE 13

DES GRAPHIQUES PLUS ÉLABORÉS

La première approche des représentations graphiques développée précédemment (en page ??) mérite nombreux compléments compte-tenue de la grande multiplicité des possibilités de matplotlib .

1. Anatomie d'une figure



2. Dimensions de la fenêtre figure

Lors de la création de la figure, il peut être utile de modifier la taille de celle-ci en rajoutant à la commande permettant de créer la figure en rajoutant l'instruction :

```
plt.figure(figsize=(abscisse, ordonnée))
```

3. Les options proposées de pyplot

Chaque fonction du module pyplot appelée dans le script va apporter des changements à la figure.

Lors de la première utilisation de la commande plt.plot , nous avons seulement indiqué les deux arguments essentiels entre parenthèses, x et y. Or, pyplot offre de nombreuses options au plot . Nous allons voir que tout est paramétrable. L'écriture est la suivante : plt.plot(abscisse, ordonnée, "couleur $\text{style}_{\text{point}}$ $\text{style}_{\text{trait}}$", paramètres) .

- couleur : première lettre anglaise de la couleur(b,g,y,c,r,m,w ou k pour le noir) par exemple c='r' pour que la courbe soit tracée en rouge.
- $\text{style}_{\text{point}}$: liste ci-dessous.
- $\text{style}_{\text{trait}}$: ' :' pointillé, '-' ligne continue, '-.'point tiret, '- -' ligne de tirets, ',' pixel, 'v' triangle.
- paramètres : cela peut être l'épaisseur du trait (line- width=flottant) et/ou une légende (label="ce qui doit être écrit").

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(1,100,30)
```

```
#Debut de la figure graphique
plt.figure(1) #nom du graphique
plt.clf() #nettoyage de la figure
plt.grid(True)
plt.plot(x,np.sqrt(x),"b+:")
plt.show()
```

Paramètres	Marqueurs
x	une croix
X	une croix pleine
+	un plus
.	un pixel
o	un rond
1	tri_bas
2	tri_haut
3	tri_gauche
4	tri_droite
^	triangle pointe en haut
>	triangle pointe droite
<	triangle pointe gauche
p	pentagone
h	hexagone
H	autre hexagone
*	étoile
d	carreau
D	carreau plus gros
|	barre verticale
_	barre horizontale

TABLE 13.1 – Les marqueurs de **pyplot**

Les fonctions xlabel et ylabel permettent d'attribuer aux axes des légendes. Si vous souhaiter utiliser la syntaxe LaTeX, il vous suffit de mettre un r devant la chaîne de caractères.

```python
import matplotlib.pyplot as plt

x = [1, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5]

#Début de la figure graphique
plt.figure(1)
plt.clf()
plt.hist(x, range = (0, 5))
plt.xlabel(r'$c_2$ en mol.$L^{-1}$')
plt.ylabel(r'$N_1$')
plt.grid(True)
plt.title('Exemple d\' histogramme simple')
plt.show()
```

Application 30

Reprendre les exemples du chapitre ?? en y ajoutant des labels sur les axes, un titre au graphique, une couleur que vous aurez choisi pour les courbes ou les points. Modifier aussi la taille de la figure.

4. Personnalisation des graduations

Il est possible de modifier le nombre de sous-graduations entre deux graduations de chacun des axes grâce aux commandes set_minor_locator dans la fonction axis :

```python
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator

figure=plt.figure(1)
figure.clf()
axes = figure.add_subplot(111)
```

```
axes.plot(range(5), range(5))
axes.xaxis.set_minor_locator(AutoMinorLocator(9))
axes.yaxis.set_minor_locator(AutoMinorLocator(4))
axes.grid()
plt.show()
```

Application 31

1. Tracer la fonction $y = x^4$ sur l'intervalle $[0, 5]$.
2. Modifier le nombre de sous-graduations et observer.

5. Fonction paramètres du graphique

Afin de gagner du temps, une fonction peut permettre d'automatiser les paramètres du graphique :

```
def param_graph():
    ax = plt.gca()                                          #
    définir les axes comme un objet modifiable 'ax'
    ax.spines['right'].set_color('none')                    #
    effacer le bord droit
    ax.spines['top'].set_color('none')                      #
    effacer le bord haut
    ax.spines['bottom'].set_position(('data',0))            #
    place l'axe des abscisses
    ax.spines['left'].set_position(('data',0))
    ax.xaxis.set_minor_locator(AutoMinorLocator(4))
   # sous-graduations de l'axe des abscisses
ax.yaxis.set_minor_locator(AutoMinorLocator(4))    #
   sous-graduations de l'axe des ordonnées
    gradx = np.arange(-5,10,1) #liste des graduations
   de l'axe des abscisses
    grady = np.arange(0,70,5) #liste des graduations
   de l'axe des ordonnées
    plt.xticks(gradx) #étiquettes des graduations
    plt.yticks(grady) #étiquettes des graduations
    plt.grid() # grille
    plt.title("Courbe de la fonction f") #titre
    plt.xlabel("titre axe des abscisses") #titre axe
   abscisses
    plt.ylabel("titre axe des ordonnées")#titre axe
   ordonnées
```

Il suffira uniquement de copier cette fonction param_ graph() puis de l'appeler avant l'affichage de la fenêtre graphique.

Application 32

Soit g la fonction définie sur sur l'intervalle $[-4, 8]$ par

$$\begin{cases} 1 & \text{si } x \in [-4; -1[\\ x^2 & \text{si } x \in [-1; 2[\\ -x + 6 & \text{si } x \in [2; 8] \end{cases}$$

1. Définir dans l'éditeur la fonction g.
2. En adaptant la fonction param_graph() , représenter cette fonction en vert dans un repère tel que l'axe des abscisses soit gradué de 1 en 1, et de 2 en 2 sur l'axes des ordonnées.

6. Comment créer plusieurs figures ?

Deux courbes sur le même graphique

Il est souvent utile de pouvoir afficher plusieurs graphes pour les comparer.

```python
from matplotlib import pyplot as plt # le module
    pyplot est renommé plt
import numpy as np # le module numpy est renommé np

x=np.linspace(1,10,100)
plt.figure('Exemple 1') # On crée une première fenêtre
     appelée exemple 1
plt.plot(x,x,'r')
plt.figure("Exemple 2") # une deuxième
plt.plot(x,x*x,'g+--')
plt.show()
```

Une fenêtre contenant deux figures

La méthode .add _ subplot(xyz) subdivise la fenêtre en x ligne(s) et y colonne(s) soit (x × y) cases. Chaque case est numérotée, z est le numéro de la case où je veux mettre un graphique. La numérotation se fait de gauche à droite puis de haut en bas, en commençant par 1 .

```python
from matplotlib import pyplot as plt
import numpy as np

x=np.linspace(1,10,100)
fig=plt.figure("decoupage de fenetre",figsize=(12,6),
    facecolor='w')#taille en pouce et couleur du fond
graphe1=fig.add_subplot(121) #graphe 1
graphe1.plot(x,x,'r')
graphe2=fig.add_subplot(122) #graphe 2
graphe2.plot(x,x*x,'g')
plt.show()
```

7. Barres d'incertitudes

On peut très facilement faire des barres d'erreurs dans les 2 dimensions, par exemple, pour un scatter plot :

pyplot.errorbar(xValues, yValues, xerr = xErrorValues, yerr = yErrorValues, fmt = 'none')

xValues, yValues, xErrorValues, yErrorValues étant des listes préalablement créées.

L'argument fmt permet de choisir, de façon courte, la couleur et la forme des marqueurs du graphe. Le choix none permet de ne représenter que les barres d'incertitudes.

```python
from matplotlib import pyplot as plt

xValues = [5, 3, 7, 9]
yValues = [6, 2, 9, 12]
xErrorValues = [1, 0.5, 2.5, 3]
yErrorValues = [0.5, 0.5, 2, 1.5]
plt.scatter(xValues, yValues, zorder = 2)
plt.errorbar(xValues, yValues, xerr = xErrorValues,
    yerr = yErrorValues,
  fmt = 'none', capsize = 10, ecolor = 'red', zorder =
    1)
plt.show()
```

Application 33 — Terminale Spé

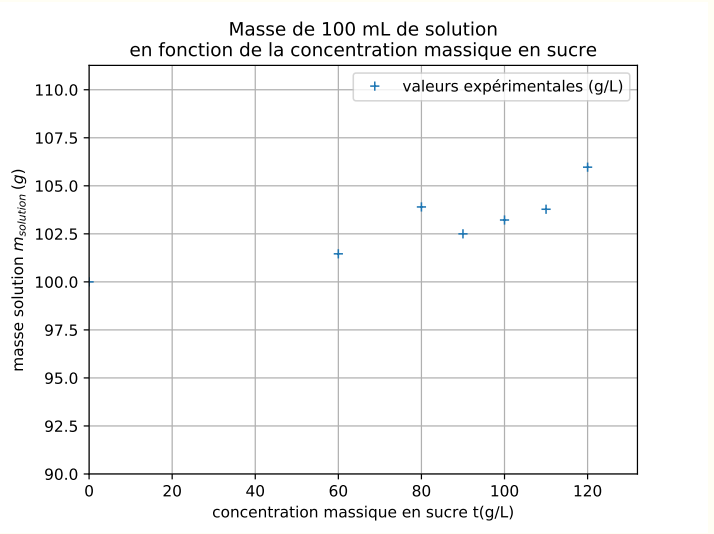
Retour sur un exemple pris précédemment :
Un émetteur et un récepteur d'ultrasons étaient placés à $d = 30\,\text{cm}$ l'un de l'autre. Le retard temporel $\Delta t(\mu\text{s})$ était mesuré plusieurs fois. Les valeurs suivantes étaient trouvées :

905,870,910,880,890,915,900,885

1. **Remplir une liste ayant pour valeur l'incertitude élargie, n étant le nombre de mesures.**

$$U(\Delta t) = 2 \times \frac{\sigma}{\sqrt{n}}$$

2. **Représenter la série de mesures avec une barre d'incertitude élargie pour les ordonnées.**



Application 34 — Écrire un script Python

Lors d'une séance de TP de seconde, des élèves ont mesuré la masse de $100\,\mathrm{mL}$ de solutions d'eau sucrée dont ils ont fait varier la concentration en masse en sucre. Voici leurs valeurs expérimentales :

$$m_{\textbf{solution}} = [100, 101.46, 103.9, 102.5, 103.22, 103.78, 105.97]$$
$$t\textbf{soluté} = [0, 60, 80, 90, 100, 110, 120]$$

Proposer un script Python permettant d'obtenir le graphique suivant :

Application 35 — Commenter un script

Le script Python suivant permet de suivre la température lors de la solidification du cyclohexane.
Commenter ce script.

```
import matplotlib.pyplot as plt


x=[]
y=[]

print(len(x))
print(len(y))


plt.figure(1)
plt.clf()
plt.grid()

plt.plot(x,y,marker='x',color='blue')
plt.xlim(min(x),1.2*max(x))
plt.ylim(min(y),1.2*max(y))

plt.xlabel('temps(s)')
plt.ylabel('Temperature (degres C)')


plt.savefig('temperature_temps_flacon.pdf')

plt.show()
```

Compléments

CHAPITRE 14

RÉSOLUTIONS D'ÉQUATIONS

Python offre deux possibilités afin de résoudre une équation :

- *numériquement, via le module scipy.optimize*
- *analytiquement, via le module sympy en faisant du calcul symbolique (résolution d'équations, intégration, dérivation, ...).*

1. Résolutions d'équations avec le module scipy

Le module scipy , comme scientifique Python , va permettre assez facilement de résoudre des équations via la fonction fsolve .

Le code suivant permet de trouver les solutions de l'équation mathématique suivante : $f(x) = x^2 - 3x - 4 = 0$.

```
import scipy.optimize as opt
from numpy import around

f=lambda x: x**2-3*x-4

#résolution
sol=opt.fsolve(f,[-6,6]) #f(0)=-4, il y a donc une
    solution positive, une négative. Les solutions sont
     recherchées sur l'intervalle [-6,6]
print(sol)
print("la solution positive est : ",sol[1])
print("la solution négative est : ",sol[0])
```

Autre possibilité : la fonction newton ne renvoie qu'une racine

```
sol1=opt.newton(f,6)
sol2=opt.newton(f,-6)
print(sol1,sol2)
```

Application 36

Supposons que l'on cherche à déterminer le volume V occupé par 1 mole d'un gaz modélisé par l'équation de Van Der Waals de température T et de pression p.
L'équation d'état (c'est-à-dire l'équation liant les variables d'état que sont p, T et V) d'un tel gaz s'écrit :

$$(p + \frac{a}{V^2}) \cdot (V - b) = R \cdot T$$

où les coefficients a (le terme de cohésion) et b (le covolume) dépendent de la nature du gaz considéré et R désigne la constante des gaz parfaits. On est donc amené à résoudre une équation non linéaire d'inconnue V.

Gaz	a ($\mathrm{Pa \cdot m^3 \cdot mol^{-2}}$)	b ($\mathrm{m^3 \cdot mol^{-1}}$)
He	$3{,}46 \times 10^{-3}$	$23{,}71 \times 10^{-6}$
Ne	$1{,}41 \times 10^{-1}$	$39{,}13 \times 10^{-6}$
CO_2	$3{,}64 \times 10^{-1}$	$42{,}67 \times 10^{-6}$

Trouver la valeur numérique du volume de $1,0$ mole de dioxyde de carbone de température $25\,^\circ\mathrm{C}$ à la pression de $17\,\mathrm{bar}$.

2. Résolutions d'équations avec le module avec le module sympy

Utilisation de la fonction solve

La fonction solve() du module sympy permet de résoudre les équations définies à l'aide de la fonction Eq() .

Dans l'exemple ci-dessous, l'équation en x est $x^2 = 2$. Avec Eq() , la valeur des deux termes de l'équation précédente sont précisés. Ensuite, nous indiquons à solve() la variable par rapport à laquelle nous souhaitons résoudre l'équation.

```
from sympy import Symbol,solve

x = Symbol('x')#définition de la variable
print(solve(Eq(x**2 - 1,0), x)) # résolution d'une é
    quation, affichage des solutions
```

Application 37 Terminale Spé

Soit un acide faible de $\mathrm{pK}_A = 2,3$ de concentration apportée $c = 2{,}0 \times 10^{-2}\,\mathrm{mol \cdot L^{-1}}$. En négligeant l'autoprotolyse de l'eau, la concentration en acide faible finale et en ion oxonium , h, est donnée par la résolution de cette équation :

$$h^2 + \mathrm{K}_A \cdot h \cdot \mathrm{c}^0 - K_A \cdot c \cdot \mathrm{c}^0 = 0$$

1. Chercher les solutions de cette équation.
2. Sélectionner la solution positive.
3. En déduire la valeur du $pH = -\log(h)$ correspondante.

Autres possibilités de sympy

simplify	simplification
diff	dérivation
integrate	intégration
FourierTransform	transformée de Fourier

Application 38 — Terminale Spé

Un athlète lance un poids de masse m.
À la date $t = 0$, correspondant à l'instant du lancer, le poids se trouve au point $H \begin{pmatrix} 0 \\ h = 2,5m \end{pmatrix}$ et part avec une vitesse initiale $\overrightarrow{v_0}$, de valeur égale à $14\,\mathrm{m \cdot s^{-1}}$, faisant un angle α de $35°$ avec l'horizontale.
Le poids est assimilé à un point matériel. Le champ de pesanteur est considéré comme uniforme ($g = 9{,}81\,\mathrm{m \cdot s^{-2}}$).
L'objectif de cet exercice est de déterminer, par intégrations successives, l'expression de la trajectoire, $y = f(x)$, de ce poids puis de la tracer.

1. **Définir la variable temps,**
2. **Définir toutes les conditions initiales,**
3. **Définir a_x et a_y, les accélérations du poids.**
4. **Intégrer deux fois les accélérations par rapport au temps pour afficher $x = g(t)$ ainsi que $y = h(t)$,**
5. **À l'aide du module sympy.plotting et de la fonction plot , tracer $x = g(t)$ ainsi que $y = h(t)$.**
6. **Faire les changements de variables suivants :**
   - **Créer une nouvelle variable X ;**
   - **Puis créer la variable Y, obtenue en remplaçant (commande subs()) le temps t par $\frac{X}{v_{0\,x}}$.**
7. **Puis tracer, à l'aide du module sympy.plotting et de la fonction plot , $y = f(x)$. Voici une autre façon de faire :**
8. **À l'aide du module sympy.plotting et de la fonction plot_parametric , tracer $y = f(x)$.**

Application 39

Supposons que l'on cherche à déterminer le volume V occupé par 1 mole d'un gaz modélisé par l'équation de Van Der Waals de température T et de pression p.
L'équation d'état (c'est-à-dire l'équation liant les variables d'état que sont p, T et V) d'un tel gaz s'écrit :

$$(p + \frac{a}{V^2}) \cdot (V - b) = R \cdot T$$

où les coefficients a (le terme de cohésion) et b (le covolume) dépendent de la nature du gaz considéré et R désigne la constante des gaz parfaits. On est donc amené à résoudre une équation non linéaire d'inconnue V.

Gaz	a ($\mathrm{Pa \cdot m^3 \cdot mol^{-2}}$)	b ($\mathrm{m^3 \cdot mol^{-1}}$)
He	$3{,}46 \times 10^{-3}$	$23{,}71 \times 10^{-6}$
Ne	$1{,}41 \times 10^{-1}$	$39{,}13 \times 10^{-6}$
CO_2	$3{,}64 \times 10^{-1}$	$42{,}67 \times 10^{-6}$

Trouver la valeur numérique du volume de $1,0$ mole de dioxyde de carbone de température $25\,°\mathrm{C}$ à la pression de $17\,\mathrm{bar}$.

CHAPITRE 15

INTERFACES GRAPHIQUES

Dans des situations d'analyses de résultats expérimentaux, il est souvent utile d'effectuer un traitement graphique permettant ensuite la déduction d'une grandeur physique. Cette construction peut être habillement menée via Python . Plusieurs solutions existent, nous présenterons ici celles proposées pas la très riche librairie matplotlib .

Pour interagir avec les tracés, Matplotlib propose des widgets neutres pour l'interface graphique.

Tout widget doit être « attaché » à une fenêtre : lors de sa construction, cela correspond à l'argument

```
Nom_du_widget(parent, [autres arguments])
```

Pour les widgets proposées par Matplotlib , le parent est un matplotlib.axes.Axes . Matplotlib n'essaye d'être intelligent avec la disposition des objets dans la figure. Il va donc être nécessaire de préciser la façon dont vous voulez que Axes accueille le widget :

```
plt.axes([gauche, bas, largeur, taille])
```

Un Slider(curseur en anglais) représente une plage de flottants

Le curseur peut être positionner verticalement ou horizontalement à l'aide de la fonction matplotlib.widgets.Slider

Dans la documentation officielle [1], on peut trouver les arguments de la fonction Slider :

- ax,
- label,
- valmin,
- valmax,
- valinit=0.5,
- valfmt='%1.2f' (format du curseur comme la couleur),
- closedmin=True,

1. https://matplotlib.org/contents.html page 2131

- closedmax=True,
- slidermin=None,
- slidermax=None,
- dragging=True,
- valstep=None,
- orientation='horizontal',

matplotlib.widgets.Slider( )

La fonction on_changed(self, func) permet d'actualiser la valeur du curseur à sa nouvelle valeur et afficher le curseur avec sa nouvelle valeur.

Voici un script montrant l'intérêt d'un widget[2] de curseur qui modifie l'amplitude d'une courbe sinusoïdale. La fonction de mise à jour est déclenchée par l'événement on_changed() du curseur ne récupère que la première variable :

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider

TWOPI = 2*np.pi

fig, ax = plt.subplots() # Création d'une figure et d'
    une aire vide

t = np.arange(0.0, TWOPI, 0.001)
initial_amp = .5    # position initiale du curseur
    amplitude
s = initial_amp*np.sin(t)
l, = plt.plot(t, s, lw=2) # courbe initiale ajoutée
    dans la figure

ax = plt.axis([0,TWOPI,-1,1]) # coordonnées extrêmes
    des axes

axamp = plt.axes([0.25, .03, 0.50, 0.02]) #Remplissage
     de l'aire avec les valeurs extrêmales
# Slider
samp = Slider(axamp, 'Amp', 0, 1, valinit=initial_amp)

def update(val):
    # amp est la valeur actualisée par le curseur
    amp = samp.val
    # actualisation de la courbe
    l.set_ydata(amp*np.sin(t)) #Modifie les valeurs de
    l
    # Retraçage de la courbe
    fig.draw() # Retrace la courbe

# Appel de la fonction update pour récupérer les
    nouvelles valeurs du curseur et retracer la coupe
samp.on_changed(update)

plt.show()
```

2. https://riptutorial.com/fr/matplotlib/example/23577/controles-interactifs-avec-matplotlib-widgets

Application 40 — Seconde

En utilisant la fonction Slider , proposer un script permettant d'afficher une fonction sinusoïdale et avec un curseur de modifier la période de cette fonction.

Application 41 — Seconde

En utilisant la fonction Slider , proposer un script permettant d'afficher une fonction sinusoïdale et avec deux curseurs : un pour la période et un pour l'amplitude.

Application 42 — Term Spé

En utilisant la fonction Slider , proposer un script permettant d'afficher une fonction représentant une décroissance exponentielle et sa tangente à l'origine et avec deux curseurs : un pour la pente de la tangente et un pour l'asymptote horizontale.

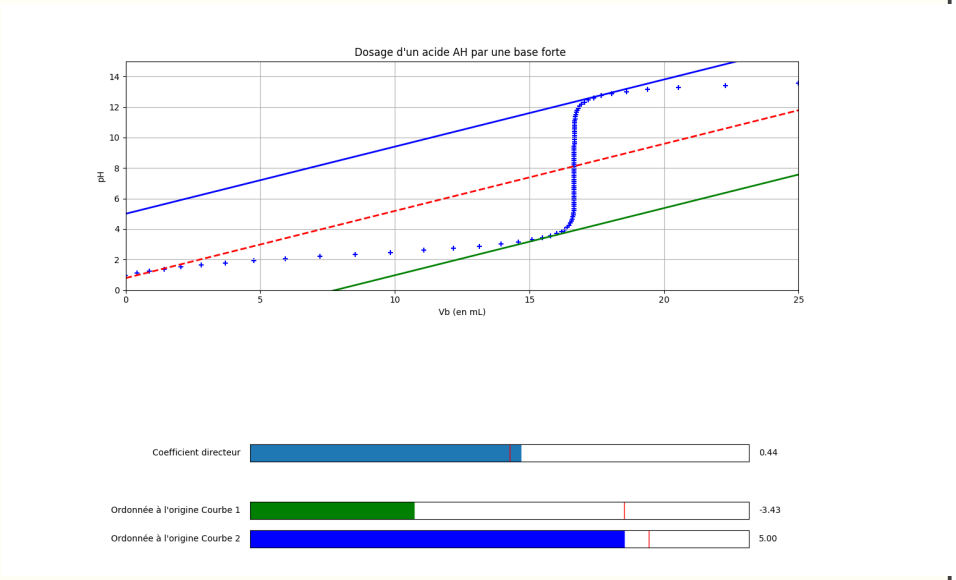


FIGURE 15.1 – Méthode des tangentes pour un titrage

Application 43 — Term Spé

En utilisant la fonction Slider et à partir du script suivant affichant une fonction représentant une courbe de titrage proposer des curseurs permettant d'utiliser la méthode des tangentes permettant de déterminer l'équivalence du titrage.

```python
#Titrage d'un acide AH par une base forte
import math as math

# Definition des fonctions
def Volb(pH): #Définition de la fonction Volb de
    solution titrante (base forte) ajoutée
    Ke = 1.0e-14
    h=10**(-pH)
    D=1+(h/(10**(-pKa)))
    return Va*(Ca/D-h+Ke/h)/(h+Cb-Ke/h)

#La fonction suivante retourne 2 listes pH et  volume
    versé Vb
```

```
def liste_pH():
    npts = 100
    pH = [] #initialisaion de la variable pH avec une
   liste vide
    Vb = [] #initialisaion de la variable Vb avec une
   liste vide
    ph_max = 14+math.log10((Vbmax*Cb-Ca*Va)/(Vbmax+Va)
   )#Définition de la variable pHmax
    dph = ph_max/(npts-1)#Définition du pas
    for i in range(npts):
        ph = dph*i #Variation du pH
        Vb.append(Volb(ph))#Création de la liste[Vb]
   avec chaque nouvel élèment Volb
        pH.append(ph)#Création de la liste[pH] avec
   chaque nouvel élèment ph
    return Vb,pH

# Saisie des valeurs par l'utilisateur
pKa=float(input("pKa="))
Va=float(input("En mL, Va="))
Ca=float(input("En mol/L, Ca="))
Cb=float(input("En mol/L, Cb="))
Vbmax=float(input("en mL, Vbmax="))

# Définition des listes
Vb,pH = liste_pH()#Affectation multiple : la fonction
   liste_pH retourne un tuple de 2 listes


# Affichage du graphe

plt.figure('Modelisation',figsize=(15,8))
plt.clf()
plt.scatter(Vb, pH,color='b',marker='+')#Nuage de
   points à partir de 2 listes
plt.xlabel("Vb (en mL)")
plt.ylabel("pH")
```

```
# fin graphique titrage
```

L'utilisation d'un réticule de données simple peut se faire grâce au module widgets de matplotlib grâce à la fonction Cursor :

```
from matplotlib.widgets import Cursor
cursor = Cursor(ax, useblit=True, color='blue',
   linewidth=1.1)
```

Cette fonction admet notamment comme argument ax (aire dans une figure pyplot) qui doit être préalablement défini, par exemple :

```
fig = plt.figure('titre',figsize=(10,7))# Création d'
   une figure
plt.clf() #nettoyage de la figure
ax = fig.subplots() #Ajout d'une aire dans la figure
```

CHAPITRE 16

LIAISON SÉRIE

La communication avec le port série va jouer un rôle majeur dans l'acquisition de données. Lorsque nous travaillerons avec les micro-contrôleurs Arduino qui ne sont pas capables de stocker un nombre important de données, il nous sera utile de pouvoir les récupérer pour pouvoir les enregistrer, par exemple dans un fichier .csv .

1. Le port série

En communication série, l'information à transmettre est découpée en petits blocs de taille fixe avant de la transmettre. La taille des blocs correspond au nombre des lignes disponibles pour la transmission des données.

Ce type de communication s'oppose à la communication parallèle. En communication parallèle, il y a une ligne par bits à transmettre. Tous les bits sont donc transmis en même temps.

Pour une même fréquence de communication, la communication parallèle est donc plus rapide.

L'avantage de la communication série sur la communication parallèle est qu'elle nécessite moins de lignes, donc moins de broches, donc moins de composants. Son coût est donc plus faible.

Avec Python 3, les communications doivent se faire avec des objets de type `bytes` (chaînes d'octets). Ceci est une différence avec Python 2 qui communique avec des objets de type chaîne de caractères (type `str`).

2. Récupérer les données du port série

Le module `serial` est nécessaire pour récupérer les données du port série.

Remarque : L'ouverture du port série peut ne pas réussir pour diverses raisons (mauvais port, etc..) et si l'on ne veut pas que le programme s'arrête, on peut utiliser une structure spéciale, la structure `try : .. except : .. .` Ce qui permet de gérer l'erreur sans interruption du programme ce qui donne :

```python
import serial
import time

#Connexion Linux au port série
serial_port=serial.Serial(port="/dev/ttyACM1",baudrate
    =9600)
serial_port.setDTR(False)
time.sleep(0.1)
serial_port.setDTR(True)
serial_port.flushInput( )
# les mesures
mesure = [ ]
temps = [ ]
serial_port.flushInput( )
for i in range(100) :
    val = serial_port.readline( ).split( ) #permet de
    couper
    try :
        t = float(val[0])/1000
        m = float(val[1])
        temps.append(t)
        mesure.append(m)
        #print(val)
        print(i,'/100')
    except :
        pass
# fermeture du port série
serial_port.close()
```

Attention le port sélectionné change de dénomination suivant votre système d'exploitation :

- Sous Linux : /dev/ttyACM suivi d'un numéro (0 ou 1 en général) ou /dev/ttyUSB0 ,
- Sous Mac : /dev/cu.usbmodem1432301 ,
- Sous Windows : COM suivi d'un numéro (1, 2, 3,...).

Remarque : Depuis le terminal, la commande ls /dev peut vous permettre de connaître la liste des ports utilisés par l'ordinateur.

3. Sauvegarder les données dans un fichier csv

```python
import csv

def writeCSV(fichier,sep,col1,nomcol1,col2,nomcol2):
#fichier <str> : Le nom du fichier CSV à créér -> "mon
    _fichier.csv"
#sep <str> : Le séparateur des colonnes
#col1 <list> : La première colonne -> [1, 1.5, 2, ...]
#nomcol1 : nom de la première liste
#col2 <list> : La deuxième colonne -> [100, 50, 100,
    ...]
#nomcol2 : nom de la deuxième liste
    file = open(fichier,"w")
    writer = csv.writer(file,delimiter =sep )
    writer.writerow((nomcol1,nomcol2))
    fin1 , fin2 = len(col1), len(col2)
    if fin1 == fin2 :
        for i in range (fin1) :
            writer.writerow((col1[i],col2[i]))
        else :
            print ( "Les deux listes n ont pas la même
    taille" )
        file.close( )

writeCSV('mesures.csv',',',temps,'temps(ms)',mesure,'
    tension(V)')
```

4. Une communication dans les deux sens

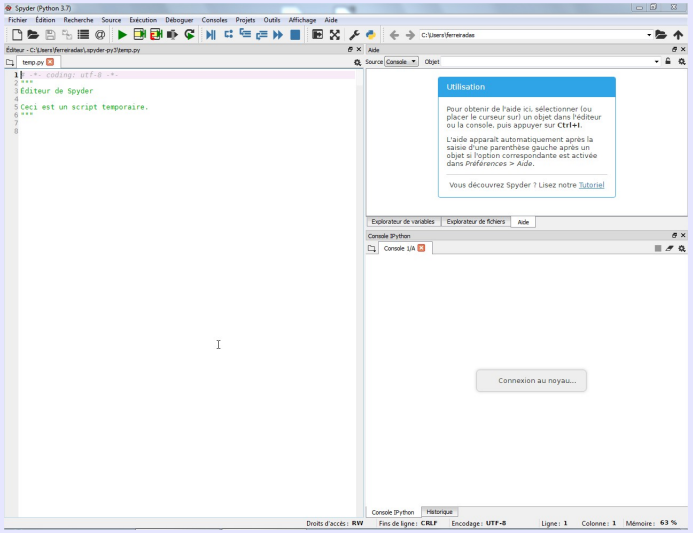
La communication entre Python et le port série ne se limite pas à la réception. Il est possible d'envoyer des données au port série :

```
from serial import *
nombre = input("Entrez un nombre : ")
port_serie.write(nombre.encode('ascii'))
```

Ouverture du logiciel Spyder

- ✓ **Pour les utilisateurs Windows, depuis le menu Démarrer du Bureau chercher Spyder puis l'ouvrir ;**
- ✓ **Une fois le logiciel ouvert, une fenêtre indiquant qu'une nouvelle version est disponibles. Cliquer sur OK.**

Les différentes fenêtres de Spyder



- ✓ ▶ **: raccourci pour exécuter un fichier ;**
- ✓ **Partie gauche : éditeur pour écrire ses scripts (code) Python ;**
- ✓ **Partie en bas à droite : interpréteur Interactif Python (IPython) : réponse du logiciel.**

Premier réglage : fenêtre graphique surgissante et interactive

- ✓ Depuis le menu Outils sélectionner Préférences puis Console Ipython puis l'onglet Graphiques puis en descendant sélectionner dans Sortie Graphique le choix automatique ;
- ✓ Fermer le logiciel Spyder puis le réouvrir.
- ✓ À partir de maintenant, les graphiques s'ouvriront dans une fenêtre surgissante.

Éxécution d'un code Python

- ✓ Icône lecture : ▶ ou la touche F5 du clavier ;
- ✓ Arrêt d'exécution depuis la console IPython : ■.

Mise en commentaire d'un code Python

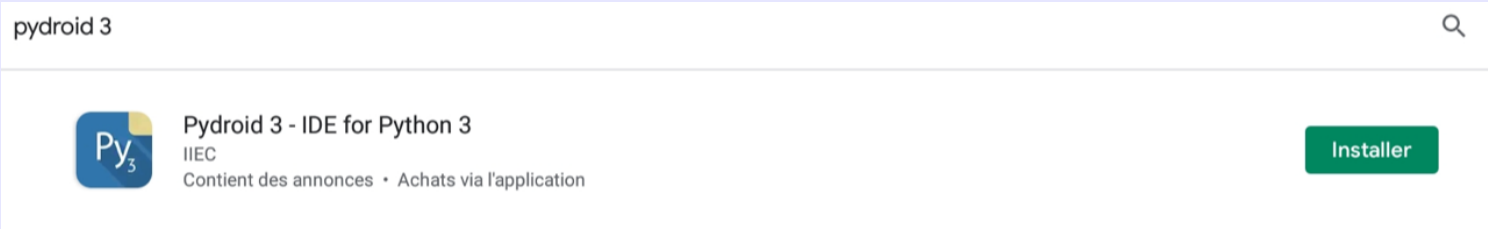
- ✓ touche # du clavier précédent le commentaire ;
- ✓ ou bien le raccourci clavier Ctrl 1.

INSTALLER PYDROÏD 3 SUR VOTRE TABLETTE

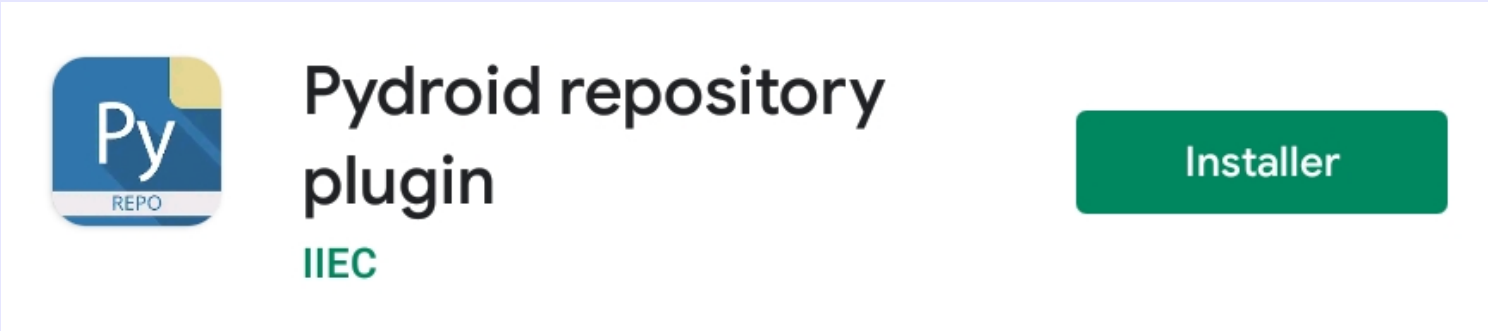
Téléchargements depuis le Play Store

✓ **Ouvrir l'application Play Store** ;

✓ **Dans la barre de recherche, chercher Pydroid 3 puis l'installer :**



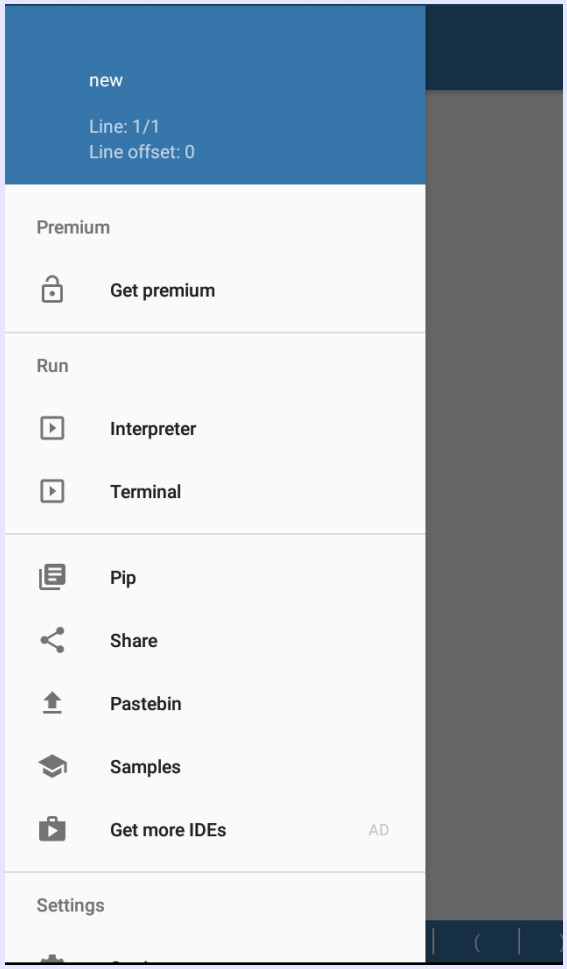
✓ **Depuis le Play Store, installer ensuite Pydroid repository plugin :**



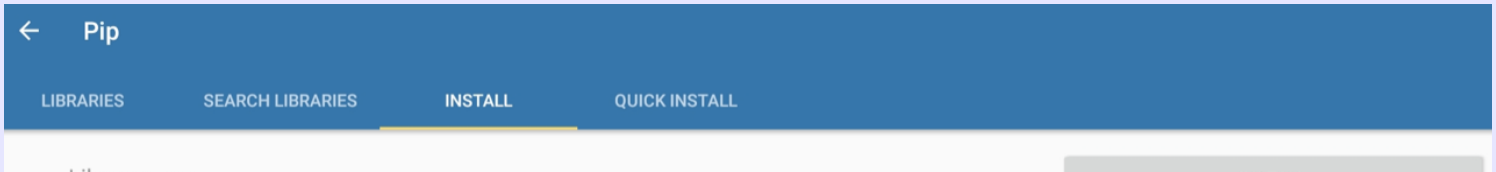
Installation de Modules nécessaires

✓ **Ouvrir Pydroid 3 ;**

✓ **Depuis l'onglet de gauche, Sélectionner l'onglet Pip :**



✓ **Puis l'onglet Install ;**



Afficher du code

Pour rédiger ce polycopié, le package *listings* a été utilisé. Il permet d'insérer du code dans le texte assez facilement et de le colorer grâce au package *xcolor*.

Dans les préambules du document, nous avons indiqué les réglages du style suivants.

```
\usepackage[table]{xcolor}
\usepackage[procnames]{listings}

%%%%%%%%%%%%%%%%%% Code Python %%%%%%%%%%%%%%%%%%
\tcbuselibrary{listings}


\definecolor{codegreen}{rgb}{0,0.6,0}
\definecolor{codegray}{rgb}{0.5,0.5,0.5}
\definecolor{codepurple}{rgb}{0.58,0,0.82}
\definecolor{backcolour}{rgb}{0.95,0.95,0.92}

\lstdefinestyle{Python_style}{
    backgroundcolor=\color{backcolour},
    commentstyle=\color{codegreen},
```

```
    keywordstyle=\color{blue},
    numberstyle=\tiny\color{red},
    %stringstyle=\color{codepurple},
    basicstyle=\ttfamily\footnotesize,
    emph={import,as,def,if,for,while,in,lambda,import,type,range},emphstyle=\color{blue},
emph={[2]list,len,print},emphstyle={[2]\color{magenta}},
    breakatwhitespace=false,
    breaklines=true,
    captionpos=b,
    keepspaces=true,
    numbers=none,
    numbersep=5pt,
    showspaces=false,
    showstringspaces=false,
    showtabs=false,
    tabsize=2,
    frameround=fttt,
    %encodage accents
    extendedchars=true,
    literate={à}{{\`a}}1 {é}{{\'e}}1 {ê}{{\^e}}1 {è}{{\`e}}1 {î}{{\^i}}1 {ç}{{\c c}}1 {ù}{{\`u}}1,
    language=Python,
}

\lstset{style=Python_style}



%https://tex.stackexchange.com/questions/495385/problem-with-lstlistings-inside-tcolorbox/495386

\tcbuselibrary{listings,breakable,skins}

% tcolorbox settings
\tcbset{
```

```
reglagestcb/.style={breakable,skin=bicolor,%
       colback=backcolour,%
       colframe=backcolour,%
       colbacklower=white,%
       boxrule=0.5pt,
       title style={draw=none,fill=none},%
       boxed title style={
arc=2mm,
        auto outer arc,
        frameround=tttt,
    },
    enhanced,
    shadow={2mm}{-1.5mm}{0mm}{fill=black!10!white,
                      opacity=0.5},
       }%
}%



%%%%%%%%%%%%%%%%%% Fin Code Python %%%%%%%%%%%%%%%%%%
```

Un code Python peut s'afficher grâce à l'environnement suivant :

```
\newtcblisting{CodePython}{reglagestcb,
bicolor,colbacklower=white,
colframe=green!10!black,breakable,
listing only,
listing options={
keywordstyle=\color{blue},
commentstyle=\color{green!90!black},
 numberstyle=\tiny\color{red},
}}
```

Il suffit alors d'appeler cet environnement :

```
\begin{CodePython}
liste=[]  #initialisation de la liste
for i in range(10)
liste.append(i)  #remplissage de la liste
\end{CodePython}
```

Pour obtenir l'affichage suivant :



```python
liste=[]  #initialisation de la liste
for i in range(10)
    liste.append(i)  #remplissage de la liste
```

Encadré application

Afin de créer des box pour les applications, j'ai utilisé le package tcolorbox puis la commande :

```
%% commande application \app{niveau}{texte}

\tcbset{
  additional title/.style={after title={\hfill #1}}
}

\newcounter{num_app}
\setcounter{num_app}{0}
\newcommand{\app}[2]{\refstepcounter{num_app}
```

```
\begin{tcolorbox}[colback=yellow!5,check odd page,
title=Application \arabic{num_app},
additional title={#1}
]
#2
\end{tcolorbox}
}
```

- **Django Dev ;**
- **Python appliqué à la physique ;**
- **Tracé de courbes avec Python ;**
- **Polycopié de formation de l'académie de Bordeaux ;**
- **Polycopié de formation de l'académie de Versailles ;**
- **Polycopié de formation de l'académie de Poitiers ;**
- **Site de classe préparatoire du lycée Kleber.**

animation, 45
arrow, 35

B.O., v
boucle, 21

calcul symbolique, 57
cellule, 3
code LaTeX, 75
courbe paramétrique, 57
Cursor, 64

dataframe, 25

Écriture scientifique, 5

fichier csv, 31
fichier txt, 31
fonction, 27
fonction paramètres plot, 49
for, 21

histogramme, 11

if, 17
if ... then, 17
input, 3
integrate, 57

lambda, 27
liste, 9, 23
liste comprehension, 9, 23

module, 7

options plot, 49
ouvrir un fichier, 31

pandas, 25, 32
plot, 11
procedure, 27
Pydroïd 3, 72

quiver, 35

range, 9
régression linéaire, 41

scatter, 11
scipy, 57
sitographie, 81
solve, 57
Spyder, 3, 69
statistisque, 41
subplot, 49
substitution, 57
sympy, 57

type, 3

variable, 3
vecteur, 35

while, 21
widget, 61