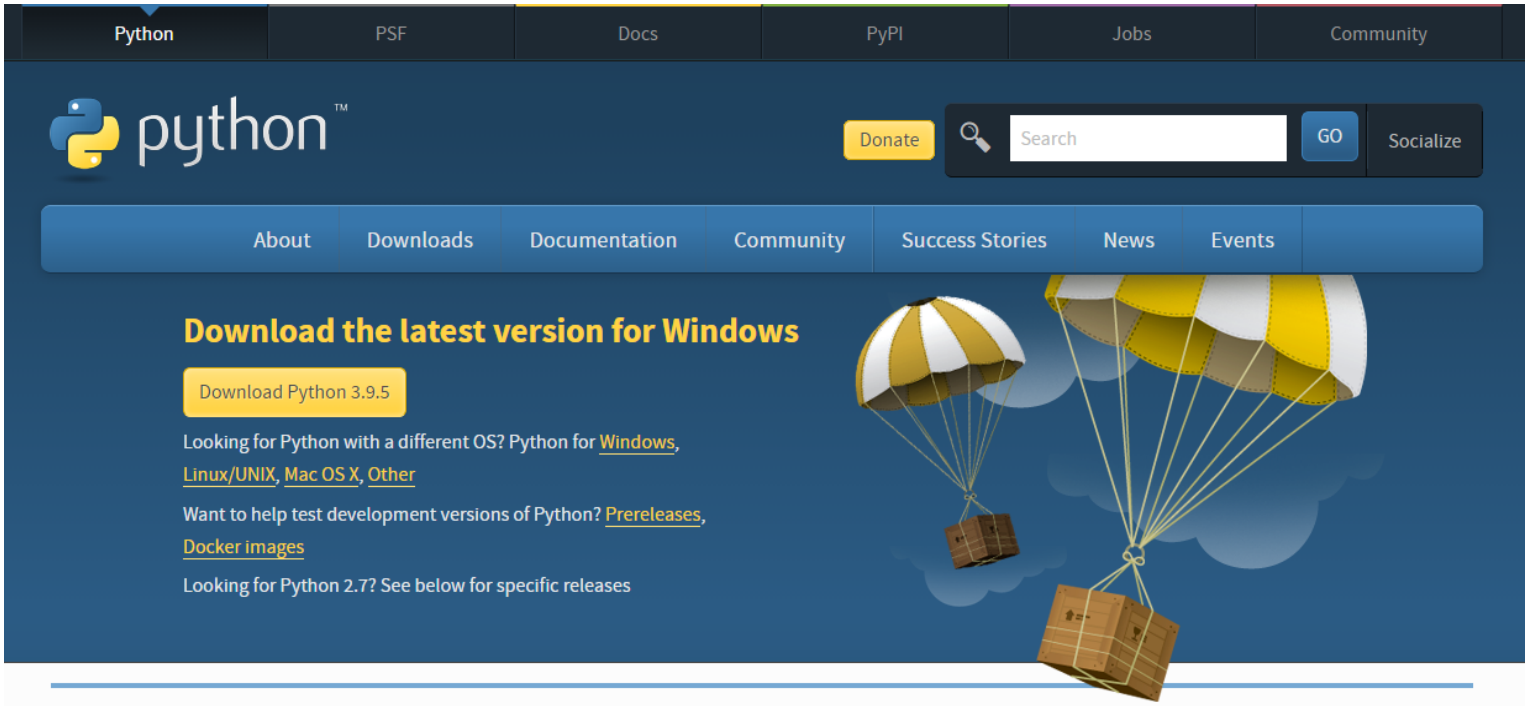
# Programmation Python

Institut Supérieur du Numérique
(SUPNUM)

13 mars 2023

Python est un langage de programmation sous licence libre promu par la Python Software Foundation. Il fonctionne sur la plupart des systèmes d'exploitation et fait partie des langages les plus utilisés pour apprendre la programmation. Il inclut l'environnement de développement IDLE, qui vise à faciliter la programmation en Python. Les programmes Python sont placés dans des fichiers d'extension « .py ».

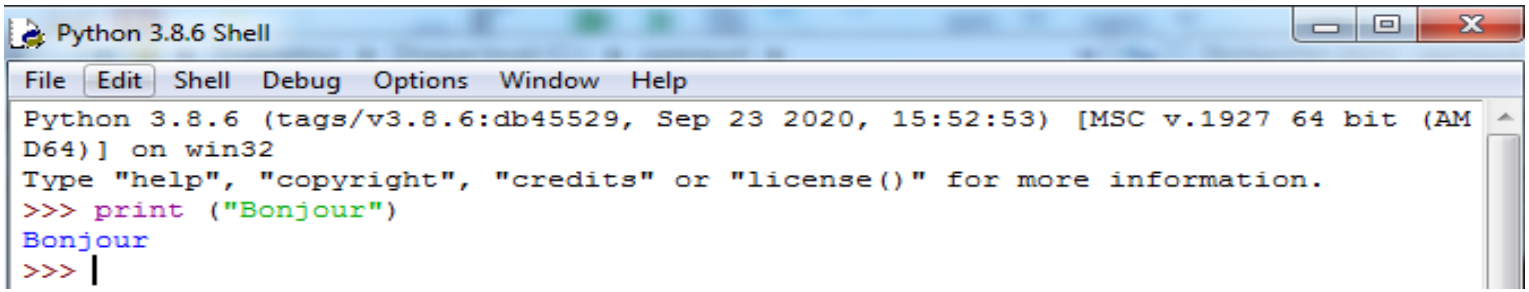
Pour commencer à programmer avec Python, il vous faut un interpréteur du code et pour le télécharger rendez-vous sur le site officiel de Python (`https://www.python.org/`) et cliquez sur Downloads.
Après avoir télécharger la version compatible avec votre système d'exploitation lancez l'installation.



Pour programmer avec Python vous avez deux modes d'exécution :

- Utiliser l'interpréteur Python (IDLE).

  **Exemple :**




- Écrire un programme dans un fichier puis l'exécuter via CMD.

  **Exemple :**

Au cours de ce chapitre nous allons travailler avec un environnement de développement intégré (IDE) appelé PyCharm.

PyCharm développé par l'entreprise tchèque JetBrains, c'est un logiciel multi-plateforme qui fonctionne sous Windows, Mac OS et Linux. Il est décliné en édition professionnelle, diffusé sous licence propriétaire, et en édition communautaire diffusé sous licence Apache.


Afin de télécharger **Pycharm** rendez-vous sur :
(`https://www.jetbrains.com/fr-fr/pycharm/`)

En **Python**, la déclaration d'une variable et son initialisation (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps.

**Exemple :**

```
1    x = 2
2    print(x)
```

Ligne 1. Dans cet exemple, nous avons déclaré, puis initialisé la variable x avec la valeur 2. Notez bien qu'en réalité, il s'est passé plusieurs choses :

- Python a « deviné » que la variable était un entier. On dit que Python est un langage au **typage dynamique** et pour savoir le type de **x** on tape **type(x)**.
- Python a alloué (réservé) l'espace en mémoire pour y accueillir un entier. Python a aussi fait en sorte qu'on puisse retrouver la variable sous son nom **x**.
- Enfin, Python a assigné la valeur 2 à la variable **x**.

**Remarque :**

Certaines règles doivent être respectées pour nommer les variables et les fonctions :

- Le nom d'une variable doit obligatoirement commencer par une lettre suivie d'une suite de lettres et de chiffres, il ne doit pas contenir d'espace.
- **noms** interdits : ce qui a déjà été utilisé, ainsi que les mots réservés et les fonctions prédéfinies de Python.

**Les opérateurs arithmétiques :**

Les opérateurs arithmétiques utilisent des valeurs numériques (variables ou expressions) comme opérandes et renvoient une valeur numérique.

| Expression | Description | Opérateur |
|---|---|---|
| X + Y | Addition | + |
| X - Y | Soustraction | - |
| X * Y | Multiplication | * |
| X//Y | Division entière | // |
| X/Y | Division | / |
| X % Y | Modulo (le reste de la division ) | % |
| X ** Y | Puissance | ** |

**Remarque :**

Certains opérateurs peuvent avoir des comportements différents en fonction des types d'opérandes sur lesquels ils agissent, par exemple l'opérateur "+" additionne des nombres, mais concatène des chaines de caractères.

## Les opérateurs de comparaison :

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne “True” ou “False”. Les opérateurs de comparaisons s’appliquent sur tous les types de base.

| Expression | Description | Opérateur |
|---|---|---|
| X < Y | Inférieur | < |
| X <= Y | Inférieur ou égal | >= |
| X > Y | Supérieur | > |
| X >= Y | Supérieur ou égal | >= |
| X == Y | Égal (attention !) | == |
| X != Y | Différent | != |

Pour réunir des expressions booléennes :

- X and Y , le ET logique
- X or Y , le OU logique
- Not X , le NON logique

## Les opérateurs d'affectation :

L'affectation est l'instruction qui associe une valeur à une variable. Elle est symbolisée en python par le signe "=".

Les expressions dans le tableau ci-dessous prennent toujours en considération l'ancienne valeur de X et le résultat sera la nouvelle valeur de X.

| Expression | Équivalent | Description | Opérateur |
|---|---|---|---|
| X = 1 | X = 1 | Affecte 1 à la variable X | = |
| X += 1 | X = X + 1 | Incrémente X de 1 | += |
| X -= 1 | X = X - 1 | Décrémente X de 1 | -= |
| X *= 2 | X = X * 2 | Multiplie X par 2 | *= |
| X /= 2 | X = X / 2 | Divise X par 2 | /= |
| X //= 2 | X = X // 2 | la division entière de X par 2 | //= |
| X **= 4 | X = X ** 4 | X à la puissance 4 | **= |
| X %= 2 | X = X % 2 | le reste de la division entière de X par 2 | %= |

- **Écrire sur la sortie standard :**

  La fonction **print()** accepte un nombre arbitraire d'expressions. Elle affichera chacune d'elles dans l'ordre, séparées par un espace. La dernière valeur est suivie d'une retour à la ligne.

**Exemple :**

```
1  a = 3
2  b = 5
3  print(a, b)
4  c = a * b
5  d = c - b
6  e = 2 * c + b
7  print('Résultats :', d, e)
```

**Résultat :**

```
3 5
Résultats : 10 35
```

- **Lire au clavier :**
  Pour avoir un programme intéractif, nous allons utiliser la fonction **input()**. Elle affiche une invite (chaîne de caractères qui lui est fournie en paramètre) et récupère les caractères saisis au clavier par l'utilisateur. Cette fonction renvoie la chaîne de caractères contenant les caractères saisis par l'utilisateur. Si une information d'un type particulier est attendu (un entier par exemple), nous utiliserons les opérations de conversion de type.

**Exemple :**

```
1  entree_age = input('Entrez votre âge : ')
2  age = int(entree_age)
3  double_age = age * 2
4  print('Le double de votre âge est :', double_age)
```

**Résultat :**

```
Entrez votre âge : 29
Le double de votre âge est : 58
```

Certaines instructions sont regroupées en blocs de la façon suivante :

```
entête du bloc:
    instruction 1 du bloc
    instruction 2 du bloc
    instruction 3 du bloc
instruction hors bloc
```

l'entête du bloc peut être (un test, une boucle, une fonction....), d'ou vient l'importante notion de l'indentation en Python.
L'indentation (le décalage) se fait avec la touche tabulation ou espace, avec ce décalage on peut insérer un bloc dans un bloc, dans un bloc.....
L'indentation fait partie du langage Python, changer l'indentation change la signification du programme

## La structure if :

La forme la plus simple est de if :

**Syntaxe :**

**if expression :**

instruction 1 du if

instruction .....

suite

**Exemple :**

```
1  age = input('Entrez votre âge : ')
2  age = int(age)
3  if age >= 18:
4      print('Vous  êtes majeur')
```

- Expression est une expression qui renvoie un booléen, qui est donc évaluée à True ou False.
- Les instructions du bloc du **if** sont effectuées uniquement si l'expression est évaluée à True.
- Dans tous les cas, le programme reprend à l'instruction après **if**.

## La structure if.... else :

On utilise généralement cette structure si nous avons seulement deux cas à traiter :

**Syntaxe :**

**if expression :**
    bloc d'instructions du if
**else :**
    bloc d'instructions du else

**Exemple :**

```
1  age = input('Entrez votre âge : ')
2  age = int(age)
3  if age >= 18:
4      print('Vous êtes majeur')
5  else:
6      print('Vous êtes mineur')
```

## La structure if.... elif .... else :

Si le nombre des cas à tester dépasse deux, nous faisons appel à cette structure pour faire le nécessaire :

**Syntaxe :**
**if expression1 :**
bloc d'instructions du if
**elif expression2 :**
bloc d'instructions du elif
.....
**else :**
bloc d'instructions du else

**Exemple :**

```
1  moyenne = input('Entrez votre moyenne : ')
2  moyenne = float(moyenne)
3  if moyenne >= 10:
4      print('Admis')
5  elif moyenne >= 8:
6      print('Sessionaire')
7  else:
8      print('Ajourner')
```

En programmation, on est souvent amené à répéter plusieurs fois une instruction, les boucles vont nous aider à réaliser cette tâche de manière compacte et efficace.

En python nous avons deux types de boucle :

- **Boucle bornée**
  Quand on sait combien de fois doit avoir lieu la répétition, on utilise généralement une boucle for.
- **Boucle non bornée**
  Si on ne connait pas à l'avance le nombre de répétitions, on choisit une boucle while.

## Boucle for :

**Syntaxe :**
**for i in ensemble :**
bloc d'instructions

Où i est un élément de l'ensemble ensemble. Le bloc instructions est exécuté pour chaque élément i de l'ensemble ensemble. Cet ensemble peut être une chaîne de caractères, un tuple, une liste, un dictionnaire....
La boucle la plus répandue est celle qui parcourt des indices entiers compris entre **0** et **n-1**. On utilise pour cela la boucle for et la fonction **range** comme dans l'exemple qui suit où nous voulons afficher "Bonjour" dix fois :

**Exemple :**

```
1  for i in range(0, 10):
2      print("Bonjour")
```

**Fonction range() dans la boucle for :**

Il y a trois versions d'utilisation de cette fonctions :

**Type n°1 : pas par défaut =1 :**

**Exemple :**

```
1    for n in range(3, 6):
2        print(n)
```

**Résultat :**

```
3
4
5
```

**Fonction range() dans la boucle for :**

**Type n°2 : pas par défaut =1 :**

**Exemple :**

```
1    for n in range(4):
2        print(n)
```

**Résultat :**

```
0
1
2
3
```

**Fonction range() dans la boucle for :**

**Type n°3 : pas est donné :**

**Exemple :**

```
1    for n in range(3, 8, 2):
2        print(n)
```

**Résultat :**

```
3
5
7
```

## Boucle while :

**Syntaxe :**
**while expression :**
    bloc d'instructions

Où expression est une condition qui détermine la poursuite de la répétition des instructions incluses dans la boucle. Tant que celle-ci est vraie, le bloc d'instructions est exécuté.
Tout comme les tests, l'indentation joue un rôle important. Le décalage des lignes d'un cran vers la droite par rapport à l'instruction while permet de les inclure dans la boucle.

**Exemple :** nous voulons afficher "Bonjour" dix fois

```
1    n = 10
2    while n > 0:
3        print("Bonjour")
4        n -= 1
```

Une liste est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.

Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets.

**Exemples :**

```
1  animaux = ['girafe', 'tigre', 'singe', 'souris']
2  tailles = [5, 2.5, 1.75, 0.15]
3  mixte = ['girafe', 5, 'souris', 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie, en tapant print(animaux) par exemple on aura :

```
['girafe', 'tigre', 'singe', 5]
```

**Manipulation de listes :**

On peut accéder aux éléments d'une liste en utilisant leurs position (indices), sachant que les indices en python commencent de 0 jusqu'au n-1 pour une liste de n éléments.

```
Liste: ['girafe', 'tigre', 'singe', 'souris']
Indice:    0         1         2         3
```

**Exemple :**

```
1  animaux = ['girafe', 'tigre', 'singe', 'souris']
2  animaux[1] = 1
3  print(animaux)
```

**Résultat :**

```
['girafe', 1, 'singe', 'souris']
```

**Manipulation de listes :**

On peut aussi accéder aux éléments par indexation négative. Le dernier élément de la liste est alors numéroté -1. Avec la liste de l'exemple précédent, on obtient par exemple :

**Exemple :**

```
1  L = [4, 0, 10, 3]
2  print(L[-1])
```

**Résultat :**

```
3
```

**Exemple :**

```
4  L = [4, 0, 10, 3]
5  print(L[-3])
```

**Résultat :**

```
0
```

**Opération sur les listes :**

Tout comme les chaînes de caractères, les listes supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication.

**Exemple :**

```
1  ani_1 = ['girafe', 'tigre']
2  ani_2 = ['singe', 'souris']
3  print(ani_1+ani_2)
4  print(ani_1*2)
```

**Résultat :**

```
['girafe', 'tigre', 'singe', 'souris']
['girafe', 'tigre', 'girafe', 'tigre']
```

**Opération sur les listes :**

Les listes possèdent de nombreuses fonctions qui leur sont propres et qui peuvent se révéler très pratiques comme (append, insert, len....).

**Fonction append() :**

La fonction append() ajoute un élément à la fin d'une liste :

**Exemple :**

```
a = [1, 2, 3]
a.append(5)
```

**Résultat :**

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

**Opération sur les listes :**

**Fonction insert() :**

La fonction insert() insère un objet dans une liste avec un indice déterminé :

**Exemple :**

```
1    a = [1, 2, 3]
2    a.insert(2, -15)
```

**Résultat :**

```
[1, 2, -15, 3]
```

## Opération sur les listes :

### Fonction len() :

L'instruction len() vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste.

### Exemple :

```
1    longueur = len([1, 2, 3, 4, 5, 6, 7, 8])
```

### Résultat :

```
8
```

**Remarque :**

Les chaînes de caractères peuvent être considérées comme des listes (de caractères) un peu particulières :
Exemple :
animaux = "girafe tigre"
print(animaux) ===> 'girafe tigre', pas de crochets.
animaux[3] ===> 'a'

## Opération sur les listes : Slicing

Il s'agit simplement de l'extraction d'une tranche de la liste, en précisant un indice initial et un indice final. On peut définir un pas p, ce qui permet de n'extraire qu'un terme sur p (par exemple les termes d'indice pair ou impair en prenant p = 2)

```
Technique de slicing:
L[i:j]        # Extraction de la tranche [L[i], ... , L[j-1]]
L[i:j:p]      # De même de p en p à partir de L[i], tant que i+k*p < j
```

- Si le premier indice est omis, il est pris égal à 0 par défaut.
- Si le deuxième indice est omis, il est pris égal à la longueur de la liste par défaut (on extrait la tranche finale)
- Si le troisième indice est omis, il est pris égal à 1 par défaut (cas de la première instruction ci-dessus)
- Un pas négatif permet d'inverser l'ordre des termes
- Le slicing est possible aussi avec des indexations négatives.

**Exemple :**

```
>>> M = [0,1,2,3,4,5,6,7,8,9,10]
>>> M[3:6]
[3, 4, 5]
>>> M[2:8:2]
[2, 4, 6]
>>> M[:3]
[0, 1, 2]
>>> M[3::3]
[3, 6, 9]
>>> M[::5]
[0, 5, 10]
```

Dans l'exemple précédent, En premier on définit un dictionnaire vide avec les accolades (tout comme on peut le faire pour les listes avec []). Ensuite, on remplit le dictionnaire avec différentes clés ("Numerique","Stat", "energie") auxquelles on affecte des valeurs ("SUPNUM","ISMS", "ISME"). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste).

Pour récupérer la valeur associée à une clé donnée, il suffit d'utiliser la syntaxe suivante dictionnaire["cle"].

**Exemple :**

```
res = ISM["Numerique"]
print(res)
```

**Résultat :**

```
SUPNUM
```

Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets, c'est-à-dire qu'il n'y a pas de notion d'ordre (i.e. pas d'indice). On accède aux **valeurs** d'un dictionnaire par des **clés**. Ceci semble un peu confus ? Regardez l'exemple suivant.

**Exemple :**

```
ISM = {}
ISM['Numerique'] = "SUPNUM"
ISM['Stat'] = "ISMS"
ISM['Energie'] = "ISME"
```

**Résultat :**

```
{'Numerique': 'SUPNUM', 'Stat': 'ISMS', 'Energie': 'ISME'}
```

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques.

**Exemple :** si vous appelez la fonction len() de la manière suivante.

```
1  long = len([0, 1, 2])
2  print(long)
```

**Résultat :**

```
3
```

Voici ce qui se passe :

- vous appelez "len()" en lui passant une liste en argument (ici la liste [0, 1, 2]) ;
- la fonction calcule la longueur de cette liste ;
- elle vous renvoie un entier égal à cette longueur (stocké dans la variable long).

## Définition

Pour définir une fonction, Python utilise le mot-clé **def**. Si on souhaite que la fonction renvoie quelque chose, il faut utiliser le mot-clé return.

**Exemple :**

```
def carre(x):
    return x ** 2
print(carre(2))
```

Notez que la syntaxe de **def** utilise les deux-points comme les boucles **for** et **while** ainsi que les tests **if**, un bloc d'instructions est donc attendu. De même que pour les boucles et les tests, l'**indentation** de ce bloc d'instructions (qu'on appelle le corps de la fonction) est **obligatoire**.

Dans l'exemple précédent, nous avons passé un argument à la fonction **carre()** qui nous a renvoyé (ou retourné) une valeur que nous avons immédiatement affichée à l'écran avec l'instruction **print()**. Que veut dire valeur **renvoyée** ? Et bien cela signifie que cette dernière est récupérable dans une **variable**.

Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur.

**Exemple :**

```
1  def hello():
2      print("Bonjour SUPNUM")
```

Dans ce cas la fonction, **hello()** affiche la chaîne de caractères "Bonjour SUPNUM". Elle ne prend aucun argument et ne renvoie rien.

### Passage d'arguments

Le nombre d'arguments que l'on peut passer à une fonction est variable. Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui développe une nouvelle fonction.

**Exemple :** Vu que le typage est dynamique c'est pas la peine de mettre les types des arguments.

```
1  def fois(x, y):
2      return x*y
```

## Fonction récursive :

Une fonction récursive est une fonction qui s'appelle elle-même.

Exemple :

```
1    def factorielle(n):
2        if n == 0:
3            return 1
4        else:
5            return n * factorielle(n - 1)
6    print("3! = ",factorielle(3))
```

Résultat :

```
3! =  6
```

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou libraries). Ce sont des « boîtes à outils » qui vont vous être très utiles.

Les développeurs de Python ont mis au point de nombreux modules qui effectuent une quantité phénoménale de tâches. Pour cette raison, prenez toujours le réflexe de vérifier si une partie du code que vous souhaitez écrire n'existe pas déjà sous forme de module.

La plupart de ces modules sont déjà installés dans les versions standards de Python. Vous pouvez accéder à une documentation exhaustive sur le site de Python (`https://docs.python.org/fr/3/py-modindex.html`). N'hésitez pas à explorer un peu ce site, la quantité de modules disponibles est impressionnante (plus de 300).

Par exemple le module **math** est un module standard en Python, toujours disponible (c'est pas la peine de le télécharger). Pour utiliser des fonctions mathématiques sous ce module, vous devez importer le module en utilisant import math.

**Exemple :**

```
1  import math
2
3  racine = math.sqrt(4)
4  print(racine)
```

**Résultat :**

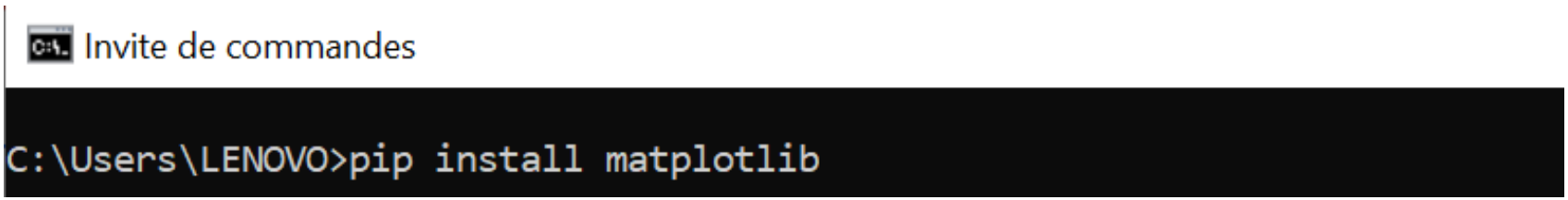
```
2.0
```

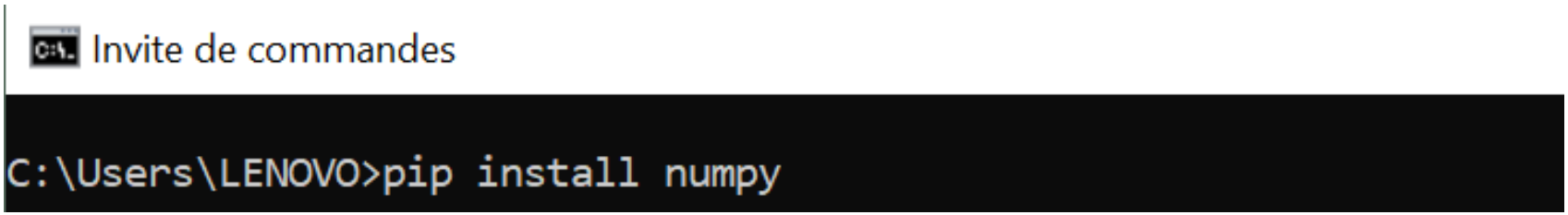
Voici quelques modules inclus dans la version standard de Python :

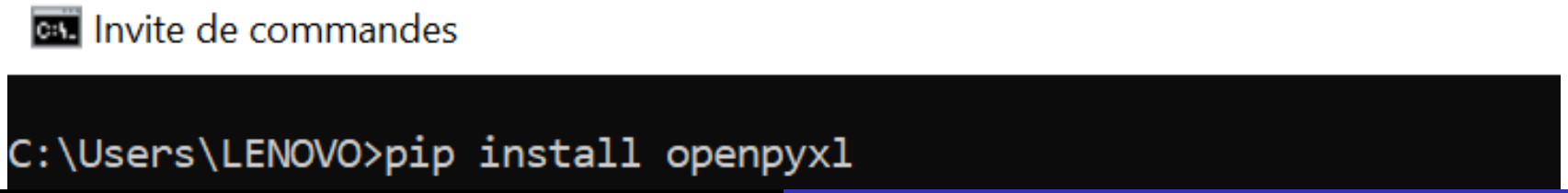
- random : fonctions permettant de travailler avec des valeurs aléatoires
- sys : fonctions systèmes
- os : fonctions permettant d'interagir avec le système d'exploitation
- time : fonctions permettant de travailler avec le temps

Quand on a un grand nombre de modules, il peut être intéressant de les organiser dans des dossiers. Un dossier qui rassemble des modules est appelé un package (paquetage en français). Le nom du package est le même que celui du dossier. Par exemple, on crée un dossier package1 dans lequel on place le fichier module1.py.

Pour importer les modules qui ne sont pas inclus dans la version standard de Python on utilise la commande ( **pip install nom_du_module**) afin de les télécharger puis on utlise import...
Exemples sur le téléchargement est l'installation des packages dont on aura besoin dans ce cours :



Lorsqu'on doit traiter un grand flot de données il est parfois utile de stocker ces données dans des fichiers (ou des bases de données) puis de lire ces fichiers à l'aide de programme.

f = open('nomFichier', 'modeLecture')

La fonction open va créer un objet-fichier (f ici) auquel on peut appliquer des méthodes. Il est ouvert dans le mode de lecture spécifié. Les cas possibles sont :

- 'r' : ouverture en lecture (read).
- 'w' : ouverture en écriture (write). Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé.
- 'a' : ouverture en écriture en mode ajout (append). On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé

1. **Lecture dans un fichier** :

   Méthode readlines() :

   Avant de passer à un exemple concret, créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire courant avec le nom test.txt et le contenu suivant :

```
girafe
tigre
singe
souris
```

La fonction **readlines** de Python lit tout le contenu du fichier texte et les place dans une liste de lignes.

Exemple complet de lecture d'un fichier avec Python :

```
fichier = open("test.txt", "r")
lignes = fichier.readlines()
for ligne in lignes:
    print(ligne)
fichier.close()
```

Résultat :

```
girafe
tigre
singe
souris
```

## Méthode readline() :

La méthode **readline()** (sans **s** à la fin) lit une ligne d'un fichier et la renvoie sous forme de chaîne de caractères. À chaque nouvel appel de **readline()**, la ligne suivante est renvoyée. Associée à la boucle **while**, cette méthode permet de lire un fichier ligne par ligne.

```
with open("test.txt", "r") as fichier:
    ligne = fichier.readline()
    while ligne != "":
        print(ligne)
        ligne = fichier.readline()
```

## Résultat :

```
girafe

tigre

singe

souris
```

2. **Écriture dans un fichier** :

Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

```
animaux = ["poisson", "abeille", "chat"]
with open("test.txt", "w") as fichier:
    for animal in animaux:
        fichier.write(f"{animal}\n")
```

Résultat :

```
poisson
abeille
chat
```

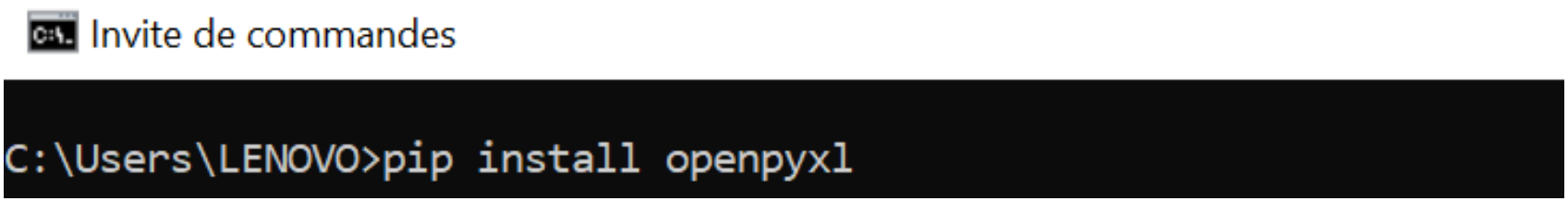
## Les fonctions de lecture / écriture :

| *Méthode* | *Description* |
|---|---|
| `read()` | le fichier jusqu'à EOF et renvoie une chaîne de caractères |
| `read(n)` | Lit `n` caractères dans le fichier à partir de la position courante |
| `readline()` | Lit une ligne du fichier jusqu'à \n et renvoie la chaîne |
| `readlines()` | Lit l'ensemble des lignes d'un fichier et les met dans une liste |
| `write(string)` | Ecrit la chaîne dans le fichier |

Python fournit le module **openpyxl**, qui est utilisé pour traiter les fichiers **Excel** sans impliquer de logiciel d'application Microsoft tiers. En utilisant ce module, nous pouvons contrôler Excel sans ouvrir l'application. Il est utilisé pour effectuer des tâches Excel telles que lire des données à partir d'un fichier Excel ou écrire des données dans le fichier Excel...

## Installation d'openpyxl :

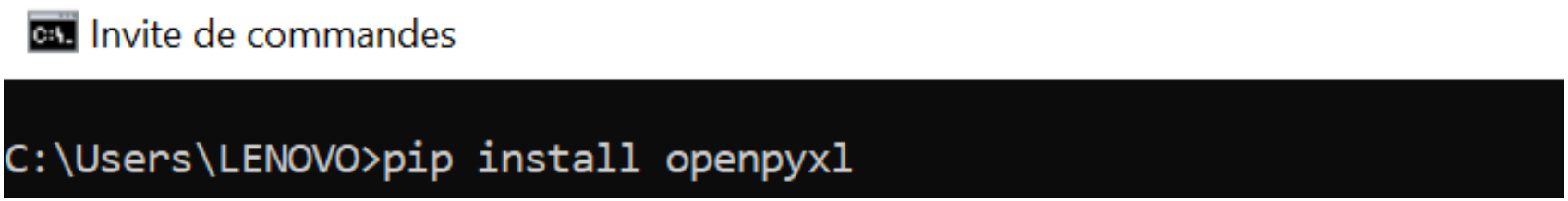
Openpyxl est disponible sur PyPI, vous pouvez donc l'installer en utilisant pip :



Python fournit le module **openpyxl**, qui est utilisé pour traiter les fichiers **Excel** sans impliquer de logiciel d'application Microsoft tiers. En utilisant ce module, nous pouvons contrôler Excel sans ouvrir l'application. Il est utilisé pour effectuer des tâches Excel telles que lire des données à partir d'un fichier Excel ou écrire des données dans le fichier Excel...

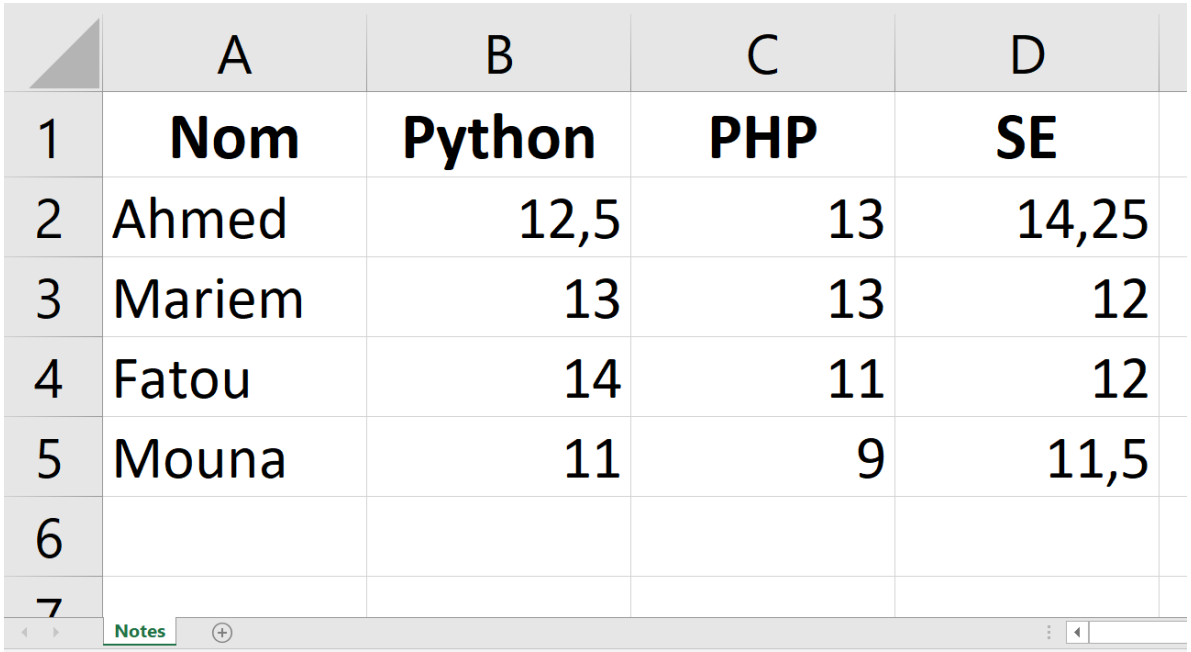
## Installation d'openpyxl :

Openpyxl est disponible sur PyPI, vous pouvez donc l'installer en utilisant pip :



## 1. Lire des données à partir d'un fichier Excel :

Supposons que nous avons un classeur Excel appelé "Etudiants" qui contient les notes des étudiants dans la feuille "Notes".



| | A | B | C | D |
|---|---|---|---|---|
| 1 | **Nom** | **Python** | **PHP** | **SE** |
| 2 | Ahmed | 12,5 | 13 | 14,25 |
| 3 | Mariem | 13 | 13 | 12 |
| 4 | Fatou | 14 | 11 | 12 |
| 5 | Mouna | 11 | 9 | 11,5 |
| 6 | | | | |

Notes

## 1. Lire des données à partir d'un fichier Excel :

```python
import openpyxl

# Ouvrir le fichier Excel
wb = openpyxl.load_workbook('Etudiants.xlsx')

# Sélectionner la feuille de calcul
feuille = wb['Notes']

# Lire les valeurs de chaque ligne
for row in feuille.iter_rows(min_row=2, max_row=5):
    # Créer une liste de valeurs de chaque cellule dans la ligne
    valeurs = [cell.value for cell in row]
    # Afficher les valeurs de la ligne
    print(valeurs)
```

Ce code ouvre le fichier Excel Etudiants.xlsx, sélectionne la feuille de calcul "Notes" et lit les valeurs de chaque ligne en bouclant sur chaque ligne avec la méthode iter_rows(). Pour chaque ligne, une liste de ses valeurs est créée en utilisant une liste de compréhension, puis cette liste de valeurs est affichée à l'écran.

## 2. Écrire des données dans un fichier Excel :

```python
import openpyxl

# Ouvrir le fichier Excel
wb = openpyxl.load_workbook('Etudiants.xlsx')

# Sélectionner la feuille de calcul souhaitée
Notes = wb['Notes']

# Modifier les notes de l'étudiant Ahmed en utilisant le nom de la cellule
Notes['B2'].value = 18
Notes['C2'].value = 20

# Modifier les notes de l'étudiant Ahmed en utilisant le numéro de la ligne et de la colonne
Notes.cell(2,4).value = 5

# Enregistrer les modifications dans le fichier Excel
wb.save('Etudiants.xlsx')
```

## 3. Créer un nouveau fichier Excel :

```
1  import openpyxl
2  workbook = openpyxl.Workbook()
3  worksheet = workbook.active
4  worksheet.title = 'Ma feuille de calcul'
5  workbook.create_sheet('supnum 2023', 0)
6  workbook.save('nouveau-fichier.xlsx')
```

**La deuxième** ligne crée un nouveau classeur Excel en utilisant la méthode Workbook() de la bibliothèque openpyxl. Cela crée un nouveau fichier Excel vide.
**La troisième** ligne crée une nouvelle feuille de calcul dans le classeur en cours d'utilisation en utilisant la méthode **active**. Cette méthode renvoie la première feuille de calcul du classeur (qui est créée automatiquement lorsqu'un nouveau classeur est créé).
**La quatrième** ligne renomme la feuille de calcul en utilisant la propriété **title**. Dans cet exemple, la feuille de calcul est renommée "Ma feuille de calcul".
**La cinquième** ligne crée une nouvelle feuille qui s'appele 'supnum 2023' en première position.
**La dernière** ligne enregistre le classeur en cours d'utilisation sous le nom "nouveau-fichier.xlsx" en utilisant la méthode save(). Ce fichier sera créé dans le même répertoire que le fichier Python en cours d'exécution. Si un fichier avec le même nom existe déjà dans le même répertoire, il sera remplacé par le nouveau fichier Excel.

## 4. Mise en forme :

```
from openpyxl.styles import Font
from openpyxl import load_workbook
wb = load_workbook('Etudiants.xlsx')
Notes = wb['Notes']
Notes['A1'].font = Font(name='Calibri', size=12, bold=True, italic=False, color='000000FF')
Notes['B1'].font = Font(name='Calibri', size=12, bold=True, italic=False, color='000000FF')
Notes['C1'].font = Font(name='Calibri', size=12, bold=True, italic=False, color='000000FF')
Notes['D1'].font = Font(name='Calibri', size=12, bold=True, italic=False, color='000000FF')
wb.save("Etudiants.xlsx")
```

**Résultat**

| | A | B | C | D |
|---|---|---|---|---|
| 1 | **Nom** | **Python** | **PHP** | **SE** |
| 2 | Ahmed | 18 | 20 | 5 |
| 3 | Mariem | 13 | 13 | 12 |
| 4 | Fatou | 14 | 11 | 12 |
| 5 | Mouna | 11 | 9 | 11,5 |

Notes

## Quelques fonctions utiles :

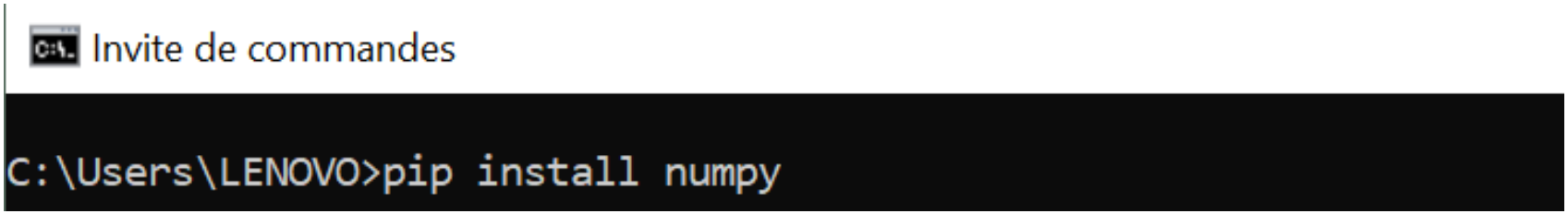
- get_sheet_names() : retourne une liste de chaînes de caractères contenant les noms des feuilles.
- iter_cols() :permet d'itérer sur toutes les colonnes d'une feuille de calcul Excel à partir d'une colonne donnée.
- insert_rows() : permet d'insérer des lignes dans une feuille de calcul Excel à partir d'un emplacement donné.
- delete_rows() : Supprime des lignes d'une feuille de calcul Excel à partir d'un emplacement donné.
- insert_cols() : permet d'insérer des colonnes dans une feuille de calcul Excel à partir d'un emplacement donné.
- delete_cols() : Cette fonction permet de supprimer des colonnes d'une feuille de calcul Excel à partir d'un emplacement donné.
- merge_cells() : Cette fonction permet de fusionner des cellules adjacentes en une seule cellule. Les données de la première cellule seront conservées et les autres données seront supprimées.
- unmerge_cells() : Cette fonction permet de scinder une cellule fusionnée en plusieurs cellules distinctes.
- append() : Cette fonction permet d'ajouter une ligne de données à la fin d'une feuille de calcul Excel.
- number_format() : Cette fonction permet de définir le format numérique d'une cellule, tel que le nombre de décimales affichées, le symbole de devise, etc.

Avant de rentrer dans le vif du sujet "les graphiques en Python", nous allons présenter la bibliothèque NumPy.

## Bibliothèque NumPy :

La bibliothèque NumPy (`http://www.numpy.org/`) permet d'effectuer des calculs numériques avec Python. Elle introduit une gestion facilitée des tableaux de nombres (matrices).

## Installation :



1. **Importation du module NumPy**

```
import numpy as np
```

Le module numpy est importé avec l'alias np (par exemple) qui est plus rapide à écrire à chaque fois !

2. **Création de tableaux**
   - **A partir d'une liste :**

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a)
```

Résultat :

```
[1 2 3 4]
```

1. **Importation du module NumPy**

```
import numpy as np
```

Le module numpy est importé avec l'alias np (par exemple) qui est plus rapide à écrire à chaque fois !

2. **Création de tableaux**
   - **A partir d'une liste :**

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a)
```

Résultat :

```
[1 2 3 4]
```

- **A partir d'un intervalle et du nombre de d'éléments :**

La méthode **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.

```
import numpy as np
a = np.linspace(1, 7, 3)
print(a)
```

Résultat :

```
[1. 4. 7.]
```

La fonction **linspace(start,end, nb)** génère n valeurs entre start et end.

- **A partir d'un intervalle et d'un pas :**

```
import numpy as np
a = np.arange(1, 2, 0.2)
print(a)
```

La fonction **arange(a,b,p)** construit un tableau Numpy de a à b (non compris) avec un pas de p.

Résultat :

```
[1.  1.2 1.4 1.6 1.8]
```

- **Créer un tableau vide :** Il est parfois intéressant de créer un tableau vide (rempli de zéros) dont les valeurs pourront être modifiées par la suite.

```
import numpy as np
a = np.zeros(5)
print(a)
```

Résultat :

```
[0. 0. 0. 0. 0.]
```

3. **Manipulation de tableaux** :

```
import numpy as np
a = np.array([1, 2, 3, 4])
a = a*3
print(a)
```

Résultat :

```
[ 3  6  9 12]
```

Les opérations mathématiques se font itérativement sur les tableaux de type Numpy. Ce qui n'est pas le cas avec les listes !
La plupart des opérateurs sont disponibles avec les tableaux Numpy !. Par contre, il n'est pas possible d'appliquer les fonctions mathématiques du module math.
Le module Numpy intégre ses propres fonctions mathématiques comme **np.sqrt()**.

Pour les tableaux de plusieurs dimensions voici un exemple :

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[1, 2, 3], [4, 5, 6], [1, 0, -1]])
print(f"A = {A}")
print(f"B = {B}")
```

Résultat :

```
A = [[1 2 3]
 [4 5 6]]
B = [[ 1  2  3]
 [ 4  5  6]
 [ 1  0 -1]]
```

La fonction shape()

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
print(" la taille du tableau est :", np.shape(A))
```

Résultat :

```
la taille du tableau est : (2, 3)
```

La fonction eye()

```
import numpy as np
C = np.eye(3)
print(C)
```

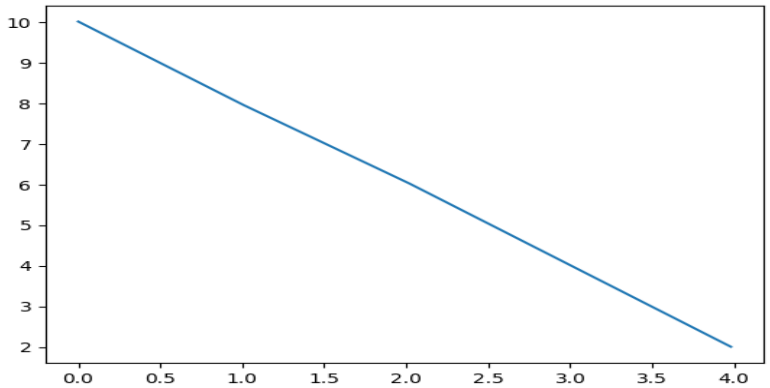
Résultat :

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

**Bibliothèque Matplotlib :** Matplotlib est une librairie Python pour la visualisation de courbes.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([0, 1.01, 2.02, 2.99, 3.98])  # Données en abscisse
y = np.array([10.02, 7.96, 6.03, 4.04, 2.01])  # Données en ordonnée
plt.plot(x, y)  # Tracé de la courbe
plt.show()  # Affichage de la courbe
```

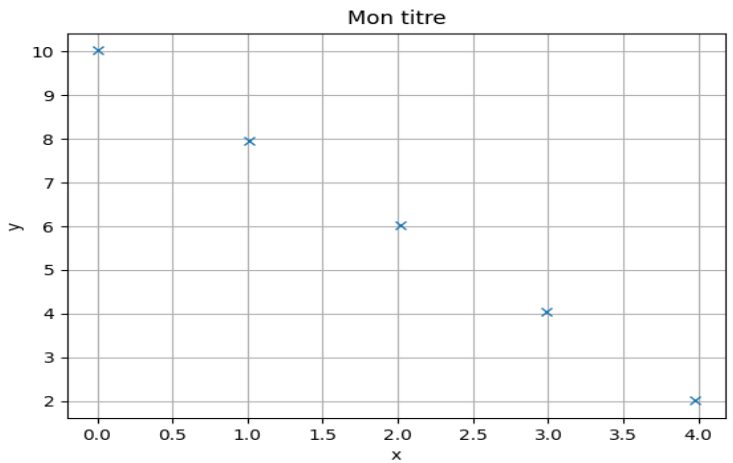
Résultat :




- La collection **pyplot** du module matplotlib est importée avec l'alias **plt**.
- La fonction **plot()** trace la courbe **y=f(x)** à partir des tableaux **x** et **y**.
- La fonction **show()** appelée en dernier affiche la fenêtre graphique.

```python
import numpy as np
import matplotlib.pyplot as plt
x = np.array([0, 1.01, 2.02, 2.99, 3.98])        # Données en abscisse
y = np.array([10.02, 7.96, 6.03, 4.04, 2.01])    # Données en ordonnée
plt.plot(x, y, 'x')       # Tracé de la courbe
plt.title('Mon titre')    # Ajout d'un titre
plt.xlabel('x')           # Nom de la grandeur en abscisse
plt.ylabel('y')           # Nom de la grandeur en ordonnée
plt.grid()                # Ajout d'une grille
plt.show()                # Affichage
```

Résultat :

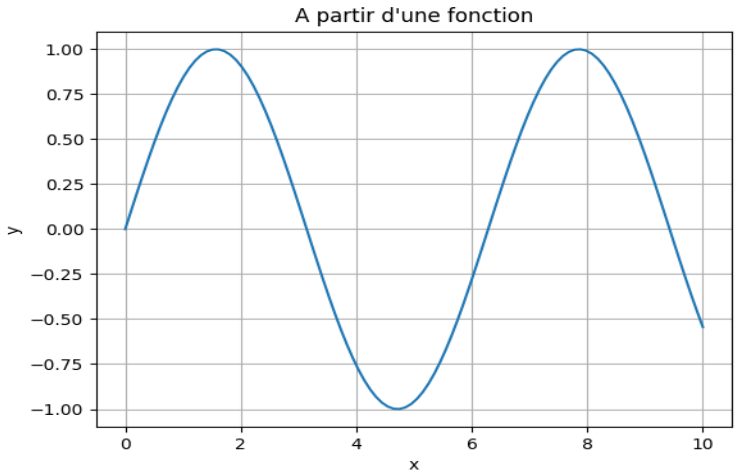



- Le paramètre x dans plot() met en évidence les points avec des croix sans les relier par des segments de droite.
- Les fonctions title(), xlabel et ylabel() ajoutent un titre et les légendes.
- La fonction grid() ajoute une grille.

Tracer une courbe à partir d'une fonction : Cas d'une sinusoïde

```python
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 100)   # Création d'un tableau de valeurs pour x
y = np.sin(x)                 # Calcul de y à partir de la fonction mathématique
plt.plot(x, y)                # Tracé de la courbe
plt.title("A partir d'une fonction")    # Titre
plt.xlabel('x')                         # Légende abscisse
plt.ylabel('y')                         # Légende ordonnée
plt.grid()                              # Ajout d'une grille
plt.show()                              # Affichage
```

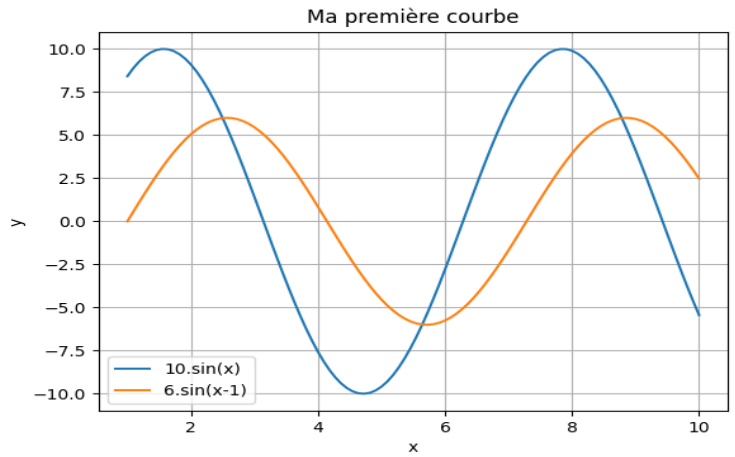
Résultat :



Cas de deux sinusoïdes avec légende

```python
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(1, 10, 100)     # Création d'un tableau de valeurs pour x
y1 = 10*np.sin(x)               # Calcul de y1
y2 = 6*np.sin(x-1)              # Calcul de y2
plt.plot(x, y1, label='10.sin(x)')   # Tracé de la courbe y1 avec texte légende
plt.plot(x, y2, label='6.sin(x-1)')  # Tracé de la courbe y1 avec texte légende
plt.title('Ma première courbe')      # Titre
plt.xlabel('x')                      # Légende abscisse
plt.ylabel('y')                      # Légende ordonnée
plt.legend()                         # Ajout de la légende
plt.grid()                           # Ajout d'une grille
plt.show()                           # Affichage
```

Résultat :



### Les marqueurs

| Marker | Result |
|---|---|
| 'o' | Circle |
| '*' | Star |
| '.' | Point |
| ',' | Pixel |
| 'x' | X |
| 'X' | X (filled) |
| '+' | Plus |
| 'P' | Plus (filled) |
| 's' | Square |
| 'D' | Diamond |

### Style de ligne

| Syntax | Line |
|---|---|
| '-' | Solid |
| ' :' | Dotted |
| '–' | Dashed |
| '-.' | Dash/dott |

### Couleurs

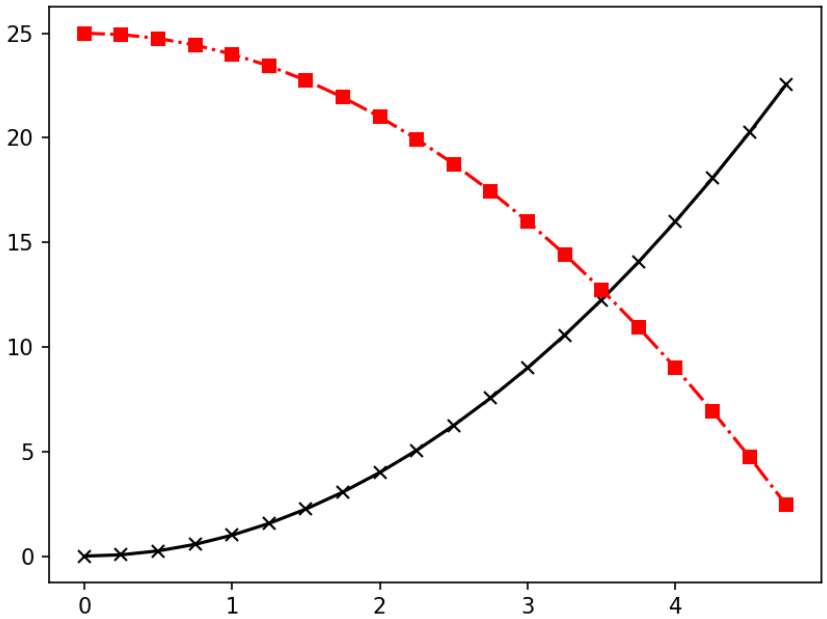
| Syntax | Color |
|---|---|
| 'r' | Red |
| 'g' | Green |
| 'b' | Blue |
| 'c' | Cyan |
| 'm' | Magenta |
| 'y' | Yellow |
| 'k' | Black |
| 'w' | White |

NB : la liste n'est pas exhaustive.

## Exemple :

```
import matplotlib.pyplot as plt
x = [n/4 for n in range(20)]
y1 = [n*n for n in x]
y2 = [25 - n*n for n in x]
plt.plot(x, y1, linestyle="-", marker="x", color="k")
plt.plot(x, y2, linestyle="-.", marker="s", color="r")
plt.show()
```

## Résultat :



En utlisant la fonction subplot() vous pouvez dessiner plusieurs courbes dans une figure :

**Exemple :**

```
import matplotlib.pyplot as plt
import numpy as np
# plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x, y)
# plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x, y)
plt.show()
```

**Résultat :**