Table des matières

Listes, tableaux et représentations graphiques

I Listes

Une liste, de type `list`, est une structure de données non homogènes : ses éléments peuvent être de types différents, et modifiable : on peut changer ou même supprimer des éléments de la liste.

I. 1 Création d'une liste

On peut créer une liste manuellement en tapant l'instruction suivante :

```
l=[x1,x2, ..., xn]
```

On obtient alors une liste de n éléments, les `xi`, qui peuvent être de types différents. Par exemple, si on définit `l=[1,2+3j,"texte",[2,1.1]]`, on a une liste de 4 éléments, dont les types sont .....
On voit sur cet exemple que l'on peut même avoir des listes de listes.
La liste vide est définie par `l=` .....
On rappelle que l'instruction `range` permet de créer des listes d'entiers de la façon suivante :

- `range(n)` : .....
- `range(d, n)` : .....
- `range(d, n, p)` : .....

  .....

On peut créer une liste à l'aide d'opérations sur une autre liste . La syntaxe est la suivante :

```
l=[f(k) for k in liste]
```

On parle parfois de « liste en compréhension »
Par exemple, si on veut créer une liste dont tous les éléments sont le triple des éléments d'une liste donnée `liste`, on tape : .....

Exemple. Créer une liste `L` contenant les 5 premiers nombres pairs en partant de 0 avec chacune des méthodes précédentes.

— Manuellement .....

— Avec range() .....

— En compréhension .....

I. 2 Accès aux éléments d'une liste

La commande `len` appliqué à une liste donne le nombre d'éléments de la liste.

```
L=[0, 3.1 ,'coucou', 1+2*1j,3**2, [0,1]]
print(len(L))
>>> 6
```

Les éléments d'une liste `L` sont indexées de `0` à `len(L)-1`. Par exemple, dans la liste `L=[0, 3.1 ,'coucou', 1+2*1j, [0,1]]` les éléments sont indexés de 0 à 5 :

0	3.1	'coucou'	1+2*1j	3**2	[0,1]
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

On voit aussi qu'on peut les indexer par l'ordre inverse en partant de -1 jusqu'à $-$`len(L)`. Méthode à utiliser avec parcimonie et précautionneusement, de la même façon que `range` avec un pas négatif.

Pour appeler un élément d'une liste `L`, on utilise les crochets : `L[i]`. Si on veut extraire une 'sous-liste' on utilise les " :", `L[i:j]` est la sous-liste de `L` allant de i à .....

Exemple Si `l=[1,2,3,4]`, alors `l[0]` = ..., `l[1:3]` = ..., `l[2:2]` = ...
Si `l=[[1,2],[3,4]]`, alors `l[0][-1]` = ...

Attention au fait que `l[k]` renvoie l'élément numéroté `k` de la liste, tandis que `l[i:j]` renvoie une liste constituée des éléments numérotés `i` à `j-1`. En particulier, `l[k]` et `l[k:k+1]` ne renvoient pas la même chose !

Les listes étant des types modifiables, on peut également changer ou supprimer des éléments (on reprend l'exemple `l=[1,2,3,4]`) :

Instruction Python	Opération effectuée	Exemples
`l[k]=x`	Remplacer l'élément numéroté `k` par `x`	`l[0]=4` →
`l[i:j]=liste`	Remplacer les éléments `i` à `j-1` par `liste`	`l[1:3]=[0,1,2]` →
`l[i:j]=[]`	Supprimer les éléments `i` à `j-1`	`l[1:2]=[]` →
`del(l[i])`	Supprimer l'élément numéroté `i`	`del(l[0])` →
`l[i:i]=liste`	Insérer `liste` à la place `i`	`l[1:1]=[5,2]` →

Mise en garde ! ⚠ Les types modifiables sont plus pratiques, en particulier lorsque l'on veut stocker des valeurs qui évoluent au cours du temps. Malheureusement, ils sont plus coûteux en terme de stockage. Ainsi, pour économiser de la mémoire, `python` ne crée pas de nouvelle liste lors d'une affectation du type `l2=l`, mais donne juste un deuxième nom à la liste `l`, qui est `l2`. Le gros inconvénient est que si l'on modifie `l2`, `l` sera modifié aussi...

Regardons l'exemple suivant :
`l=[1,2,3]`

```
l2=l
l2[0]=4
print l2  → ..............
print l  → ..............
```

Pour créer une nouvelle liste identique à `L`, il y a deux solutions :

— .....

— .....

I. 3 Opérations sur les listes

Voici les opérations sur les listes que vous pouvez être amenés à utiliser :

Instruction Python	Opération effectuée	Exemples
`+`		`[2,4] + [3,2]` →
`*`		`3*[0,1]` →
`==`		`[0,2]==[0,2,2]` →
`len`		`len([0,4])` →
`in`		`[0,1] in [0,1,2]` →

Exemple. Soit `l=[5,"bonjour",3.1]`. Que renvoient les opérations suivantes ?

`l+1j` → ..... `len(l)` → .....

`2*l` → ..... `5 in l` → .....

Remarque. Il faut faire attention que les opérations `+` et `*` ne désignent pas des sommes et multiplications des éléments de la liste. Par exemple, `[1,2]+[3,4]` ne renvoie pas `[4,6]`, mais ....., et `3*[1,2]` ne renvoie pas `[3,6]`, mais .....

On peut également ajouter des éléments de la façon suivante (en prenant pour exemple `l=[3,2,1]`) :

Instruction Python	Opération effectuée	Exemples
`l.append(x)`	Insérer `x` à la fin de la liste	`l.append(5)` →
`l.insert(i,x)`	Insérer `x` à la place numérotée `i`	`l.insert(1,5)` →
`l.remove(x)`	Enlève de la liste le premier élément dont la valeur est égale à x.	`l.remove(1)` →
`l.pop(i)`	Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour.	`print(l.pop(1))` → `print(l)` →
`l.index(x)`	Renvoie la position du premier élément de la liste dont la valeur égale x	`[4,2,3,2].index(2)` →
`l.count(x)`	Renvoie le nombre d'éléments ayant la valeur x dans la liste.	`[4,2,3,2].count(2)` →

Exemple. Pour ajouter 3 à la fin de la liste `[3,4,2]` il y a au moins 3 façons :

1. Avec `+` .....
2. Avec `append()` .....
3. Avec `insert` .....

I. 4 Boucles sur les listes

On peut faire des boucles `FOR` sur les listes, comme avec la fonction range(a,b). Il faut penser à range(a, b) comme une liste particulière de la forme $[a, a+1, a+2, \cdots, b-1]$

La syntaxe est la suivante :

```
L=[1,2,'a'] #creation de liste
for element in L: #boucle
  print(element) #instructions dans la boucle
```

L'exemple précédent affichera 1, puis 2, puis 'a'.

Exemple. Pour connaitre la position de tous les 3 dans la liste `[3,4,3,2,3]` on peut faire par exemple :

```
def ou_est_le_trois(L):
  position=[]
  compteur=0
  for element in L:
    if element==3:
      position=position+[compteur]
    compteur+=1
  return(position)
```

```
def ou_est_le_trois_bis(L):
  position=[]
  for i in range(len(L)):
    if L[i]==3:
      position.append(i)
  return(position)
```

II Tableaux

II. 1 Creations de tableaux

Les tableaux sont des structures de données qui permettent d'effectuer facilement des opérations sur les données qu'ils contiennent. Ils permettent par exemple de représenter des matrices, des grilles de jeux, ou des images.
C'est un type de structure de données homogènes (toutes les données doivent être du même type), et modifiable : on peut modifier un ou une partie des éléments du tableau.
Les tableaux, de type `array`, existent déjà dans la bibliothèque standard de `python`. Cependant, on va privilégier la bibliothèque appelée `numpy`, qui possède de nombreuses fonctions prédéfinies pour les opérations sur les tableaux. Pour charger cette bibliothèque, on tapera :

```
import numpy as np
```

Si vous souhaitez l'utiliser chez vous il faudra peut-etre installer le module, pour cela vous taperez la commande : `pip install numpy`
Pour utiliser une fonction `fonct` de numpy, on utilisera alors toujours la syntaxe `np.fonct`, qui indique que la fonction utilisée fait partie du module `numpy`.

Remarque. On pourrait utiliser la syntaxe "`from numpy import *`" pour importer `numpy`.
Problème : .....
On aurait également pu utiliser "`import numpy`".
Problème : .....
Il est ainsi préférable d'utiliser la syntaxe ci-dessus. Cependant, si le module est importé de façon différente, il suffit d'utiliser les fonctions avec le préfixe adapté.

Manuellement On peut créer un tableau à une dimension en tapant l'instruction suivante :

```
t=np.array([x1,x2, ..., xn])
```

On rentre ainsi les éléments les uns à la suite des autres comme pour une liste.

Exemple. Créer un tableau contenant les entiers 3, 7, 5 et 2 : .....
Que renvoie l'instruction suivante : `np.array([1,1.5,7])` → .....
Que renvoie l'instruction suivante : `np.array([1,"blabla",7])` → .....

On peut créer un tableau à deux dimensions en tapant l'instruction suivante :

`t=np.array([[`$x_{1,1}, x_{1,2}, \ldots, x_{1,p}$`], ..., [`$x_{n,1}, x_{n,2}, \ldots, x_{n,p}$`]])`

Exemple. Créer un tableau représentant la matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$: .....

Créer un tableau représentant la matrice $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$: .....

Tableaux en compréhension Comme avec les listes, on peut créer un tableau à l'aide d'opérations sur une liste ou un tableau. La syntaxe est la suivante pour un tableau à une dimension :

`T=np.array([f(k) for k in `*`liste`*`])`

Et pour un tableau à deux dimensions, on utilise :

`T=np.array([[f(i,j) for j in `*`colonnes`*`] for i in `*`lignes`*`])`

Pour ces deux syntaxes, *`liste`*, *`lignes`* et *`colonnes`* désignent des listes ou des tableaux déjà prédéfinis.

Exemple. Créer un tableau contenant les 10 premiers entiers positifs au cube.
Créer un tableau représentant la matrice $A \in \mathcal{M}_{np}$ de coefficients $a_{i,j} = \dfrac{1}{i+j}$.

II. 2 Fonctions `numpy`

La bibliothèque `numpy` contient des fonctions permettant de créer des matrices particulières.

Instruction Python	Matrice créée	Exemples
`np.zeros(n)`	Matrice ligne nulle de taille n	`np.zeros(3)` → `array([ 0., 0., 0.])`
`np.zeros((n,p))`	Matrice nulle de taille $n \times p$	`np.zeros((2,3))` → `array([[ 0., 0., 0.]` `[ 0., 0., 0.]])`
`np.ones(n)`	Matrice ligne de 1 de taille n	`np.ones(2)` → `array([ 1., 1.])`
`np.ones((n,p))`	Matrice nulle de 1 de taille $n \times p$	`np.ones((2,2))` → `array([[ 1., 1..],` `[ 1., 1.]])`
`np.eye(n)`	Matrice identité de taille n	`np.eye(2)` → `array([[ 1., 0..],` `[ 0., 1.]])`
`np.eye(n,p)`	Matrice pseudo identité $n \times p$	`np.eye(2,3)` → `array([[ 1., 0., 0.],` `[ 0., 1., 0.]])`
`np.diag([x1, ..., xn])`	Matrice diagonale	`np.diag([1,-5])` → `array([[ 1., 0..],` `[ 0., -5.]])`
`np.arange(xmin,xmax,pas)`	Intervalle avec pas	`np.arange(0,1,0.1)`
`np.linspace(xmin,xmax,nb)`	Intervalle avec nombre de points	`np.linspace(0,1,11)`
`np.random.rand(n)`	Matrice ligne aléatoire	`np.random.rand(3)`
`np.random.rand(n,p)`	Matrice $n \times p$ aleatoire	`np.random.rand(2,2)`

Exemple. Créer une matrice identité de taille 4, puis la matrice diagonale $A = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 2.5 & 0 \\ 0 & 0 & 3 \end{pmatrix}$.

II. 3 Accéder aux éléments d'un tableau et les modifier

Pour accéder aux éléments d'un tableau à une dimension, on utilise la même syntaxe que pour les listes.

Exemple. Si `T = np.array([-3,1,0,2])`, alors `T[0]` = ..., `T[1:3]` = ..., `T[2:2]` = ...

Pour les tableaux à deux dimensions, les lignes ainsi que les colonnes sont indexées à partir de `0`. Par exemple, pour la matrice $M = \begin{pmatrix} -1 & 2 & 0 & 3 \\ 4 & -2 & 1 & -4 \end{pmatrix}$, on obtient :

	0	1	2	3
0	-1	2	0	3
1	4	-2	1	-4

Si l'on veut accéder à certains éléments, on utilise les syntaxes suivantes :

Instruction Python	Opération effectuée	Exemples
`M[i,j]`	Élément numéroté (i, j)	`M[0,2]` →
`M[i,:]`	Ligne numérotée i	`M[1,:]` →
`M[:,j]`	Colonne numérotée j	`M[:,2]` →
`M[i1:i2,j1:j2]`	Lignes i_1 à $i_2 - 1$ et colonnes j_1 à $j_2 - 1$	`M[0:2,1:3]` →

Les tableaux étant des types modifiables, on peut également changer ou supprimer des éléments :

Instruction Python	Opération effectuée	Exemples
`M[i,j]=x`	Remplacer l'élément (i,j) par x	`M[0,2]=4` →
`M[i,:]=ligne`	Remplacer la ligne numérotée i	`M[1,:]=[0,1,2,3]` →
`M[:,j]=colonne`	Remplacer la colonne numérotée j	`M[:,2]=[1,2]` →
`M[i1:i2,j1:j2]=T`	Remplacer la sous-matrice	`M[0:2,1:3] = np.zeros((2,2))` →
`np.delete(M,i,0)`	Supprimer la ligne numérotée i	`np.delete(M,1,0)` →
`np.delete(M,j,1)`	Supprimer la colonne numérotée j	`np.delete(M,2,1)` →

Remarque. Comme pour les listes, l'opération `T2=T` ne permet pas de créer un nouveau tableau. Malheureusement, cette fois, écrire `T2=T[:,:]` ne suffit pas non plus ! Ainsi, pour créer un nouveau tableau, il faut utiliser la syntaxe suivante :

```
T2=np.copy(T)
```

II. 4 Opérations sur les tableaux

Dans les exemples suivants, on prendra `M=np.array([[1,2,3],[4,5,6]])`, `T=np.array([[0,1,2],[-1,0,1]])`, `u=np.array([1,2])` et `v=np.array([[1],[2],[-1]])` :

Instruction Python	Opération effectuée	Exemples
`np.size(M)`	Nombre total d'éléments	`np.size(M)` →
`np.size(M,0)`	Nombre de lignes	`np.size(M,0)` →
`np.size(M,1)`	Nombre de colonnes	`np.size(M,1)` →
`in`	Test d'appartenance	`2 in M` →
`np.sum`	Somme de tous les éléments	`np.sum(M)` →
`np.prod`	Produit de tous les éléments	`np.prod(M)` →
`np.max`	Maximum des éléments	`np.max(M)` →
`np.min`	Minimum des éléments	`np.min(M)` →

Contrairement aux listes, on peut également effectuer des opérations arithmétiques sur les tableaux de nombres.

Instruction Python	Opération effectuée	Exemples
`+`	Somme de deux matrices	`M+T` →
`-`	Différence	`M-T` →
`x*M`	Multiplication de chaque élément par x	`2*M` →
`M*T`	Produit **terme à terme**	`M*T` →
`M**n`	Puissance **terme à terme**	`M**2` →
`/`	Division **terme à terme**	`T/M` →
`np.fonction_math(M)`	Appliquer une fonction aux éléments	`np.log(M)` →

Il est également possible d'effectuer des calculs matriciels.

Instruction Python	Opération effectuée	Exemples
`np.dot(u,v)`	Produit scalaire entre deux vecteurs	`np.dot(u,u)` →
`np.dot(M,u)`	Produit matriciel	`np.dot(M,v)` →
`np.transpose(M)`	Transposée	`np.transpose(M)` →
`np.linalg.inv(M)`	Inverse	`np.linalg.inv(np.diag([2,3,0.5]))` →

Exercice 1. Écrire une fonction qui prend en argument une matrice carrée M et un entier positif n, et qui renvoie la puissance n-ième de M.

III Représentation graphique

Comme tous les modules, il faut le charger et plus précisément c'est un sous-module qui va nous intéresser : pyplot. Pour cela, nous n'allons pas charger toutes les fonctions comme d'habitude mais l'importer sous un nom plus court à utiliser. On utilisera donc

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

Ce qui signifie que pour utiliser une fonction de ce module comme `show()` par exemple, on devra écrire `plt.show()` (puisqu'on a importé le module sous le nom plt). De plus, le module matplotlib est très lié à un autre module qui sert à faire du calcul numérique qui s'appelle `numpy` et qu'on import souvent sous le nom np.

Pour résumer, pour représenter graphiquement des fonctions ou autres, il faudra mettre en en-tête :

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

(L'ordre entre numpy et matplolib n'a pas d'importance)

Si vous souhaitez l'utiliser chez vous il faudra peut-etre installer le module, pour cela vous taperez la commande : `pip install matplotlib`

• `plt.show()` : Pour afficher le résultat. Toute les fonctions qui suivent servent à préparer le graphique mais si on ne demande pas de l'afficher, rien ne se passera (exactement comme la fonction print : aucun calcul ne s'affiche si on ne demande pas de l'afficher avec print).

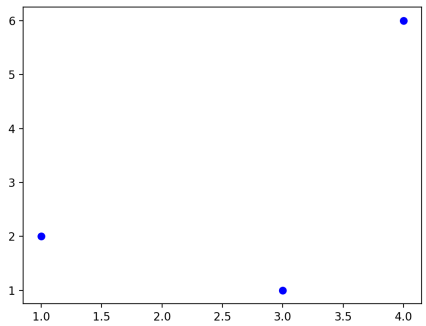
• `plt.plot(x,y)` où x et y sont deux nombres réels, marque un point de coordonnées (x,y). L'exemple suivant marquera trois points de coordonnées (1,2), (3,1), (4,6).

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

plt.plot(1,2,'bo')
plt.plot(3,1,'bo')
plt.plot(4,6,'bo')

plt.show()
```

(Ne vous occupez pas pour l'instant du 'bo',on verra ca dans la section suivante)

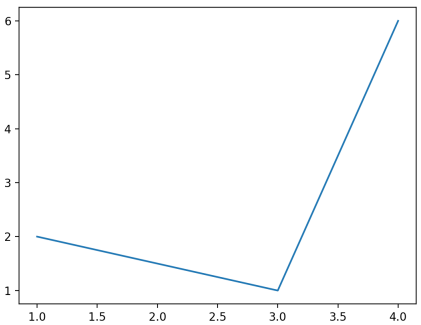


• `plt.plot(liste_x,liste_y)` : Où liste_x est une liste de nombres [x_1, x_2, ..., x_n] et liste_y une liste de nombres [y_1, y_2, ..., y_n] avec le même nombre d'éléments. Alors plt.plot(liste_x,liste_y) placera les points de coordonnées (x_1,y_1), (x_2,y_2), ..., (x_n, y_n) et les reliera de proche en proche par un segment. Voici un exemple où on relie les points (1;2), (3;1) et (4;6) :

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

plt.plot([1,3,4],[2,1,6])

plt.show()
```




```
plt.plot(liste abscisses, liste ordonnées)
plt.show()
```

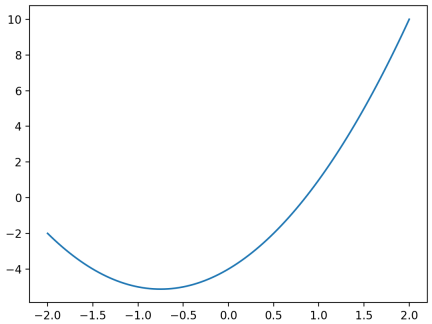
L'idée pour tracer une fonction va donc être de placer beaucoup de points de la courbe qu'on veut représenter assez proches pour qu'on ne voit pas qu'ils sont reliés par une droite.

• `np.linspace(debut, fin, N)` : C'est ici que le module numpy intervient. Pour tracer correctement une fonction, il va nous falloir beaucoup de points qu'il est hors de question de rentrer à la main comme dans l'exemple précédent. La fonction np.linspace(debut, fin, nombre) permet de créer une liste de N nombres qui commencent à la valeur debut et s'arrête à la valeur fin et uniformément répartis. De plus, si on fait une opération sur cette liste comme par exemple multiplier par 2, alors cette opération sera automatiquement appliquée à chaque terme de la liste (ce qui n'est pas vrai si on utilise une liste classique). Par exemple, traçons la fonction définie par $y = x^2 + 3x - 4$ entre -2 et 2 en utilisant 100 points :

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

x = np.linspace(-2, 2, 100)
y = 2*x**2+3*x-4

plt.plot(x,y)
plt.show()
```




```
x=np.linspace(debut, fin, taille)
y=f(x) #fonction f a definir avec numpy
plt.plot(x,y)
plt.show()
```

Amusez vous à modifier la fonction, les bornes et le nombre de points (par exemple 10) utilisés pour bien comprendre le fonctionnement.

• Supposons qu'on veuille tracer des fonctions faisant intervenir autre chose que les opérations +, -, *, / et ** comme par exemple des cosinus, sinus, exponentielle, logarithme... Dans ce cas on ne peut pas faire exactement comme dans l'exemple précédent.

Une première façon de faire est de créer "à la main" la liste des y correspondants aux x c'est à dire créer une liste composée des f(x) pour x dans la liste des abscisses. Par exemple si on veut tracer la fonction $y = \cos(x) + 3\sin(2x)$ entre -4 et 4, on pourra faire ainsi :

```
import matplotlib.pyplot as plt
import numpy as np
from math import *

abscisses = np.linspace(-4,4,100)
ordonnees = [cos(x)+3*sin(2*x) for x in abscisses]
plt.plot(abscisses,ordonnees)
plt.show()
```

— Une seconde méthode consiste à utiliser les fonctions classiques modifiées contenues dans numpy (en écrivant np.cos pour le cosinus par exemple). Pourquoi modifiées ? Car elles s'appliquent directement à toute la liste. Si on garde le même exemple de fonction $y = cos(x) + 3sin(2x)$ entre -4 et 4, cela donnera :

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

x = np.linspace(-4,4,100)
y = np.cos(x)+3*np.sin(2*x)
plt.plot(x,y)
plt.show()
```

Pour tracer plusieurs fonctions dans un même repère, il suffit de tracer plusieurs fois une fonction... Elles seront automatiquement dans le même repère. Par exemple, si je veux vérifier graphiquement que l'équation de droite$y = -2x + 3$ que j'ai obtenu correspond bien à l'équation de la tangente en 1 de la fonction $y = x^2 - 4x + 4$, il suffira d'écrire :

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

x = np.linspace(-1,3,100)
y = -2*x+3

plt.plot(x,y)

y = x**2 - 4*x + 4
plt.plot(x,y)

plt.show()
```

On peut aussi tracer des diagrammes en batons avec la fonction `plt.bar`

```
import matplotlib.pyplot as plt
import numpy as np
x = [3, 5, 6, 7]
y = [4, 1, 3, 4]
plt.bar(x,y)
plt.show()
```

III. 2 Fonctions de paramétrisation

Les fonctions présentées dans cette partie ne sont pas a connaitre par coeur, mais à savoir utiliser si on vous les donne. Dans la section précédente, nous avons vu la fonction `plt.plot(X,Y)` qui permet de tracer les segments dont les extrémités sont les points de coordonnées (x,y) avec x dans X et y dans Y. Cette fonction peut prendre beaucoup de paramètres qui permettent de configurer l'affichage de notre courbe comme on le souhaite. Une remarque au passage : je prendrais l'exemple de la fonction plt.plot mais ce qui suit peut fonctionner dans beaucoup d'autres fonctions du module matplotlib.

• Une première façon de configurer les courbes est d'utiliser le formatage rapide sous forme de chaine de 1 à 4 caractères qui résume les propriétés principales de la courbe : un caractère pour la couleur, un pour le style de point et un (ou deux) pour le style de la ligne. Ce paramètre se place après les X et Y dans la fonction plot.

Par exemple si je veux tracer une courbe en rouge (caractère "r") avec de gros points (caractère "o") et en ligne pointillée (caractères "–"), il faudra que j'écrive plt.plot(X,Y,"ro–") Ce qui donnera :

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

```

```
x = np.linspace(-2, 2, 10)
y = 2*x**2+3*x-4

plt.plot(x,y,"ro--")
plt.show()
```

```
plt.plot(X,Y,"ro-")
```

Voici la liste des couleurs les plus souvent utilisées :

• `plt.axis(x_min, x_max, y_min, y_max)` : Cette fonction permet de modifier les axes du repère qui sera affiché. Si on ne l'utilise pas, le choix des axes sera fait automatiquement mais des fois ce choix n'est pas pertinent et il faudra donc le modifier avec cette fonction. Les deux premières valeurs qu'on donne sont les valeurs minimale et maximale pour l'axe des abscisses et les deux suivantes sont celles pour l'axe des ordonnées.

• On peut choisir l'épaisseur de la courbe à l'aide de l'option `linewidth`

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

x = np.linspace(-2, 2, 10)
y = 2*x**2+3*x-4

plt.plot(x,y, linewidth=3)

plt.show()
```

• On peut enfin ajouter une légende aux axes à l'aide des options `plt.xlabel` et `plt.ylabel`

```
plt.xlabel('axe des abscisses' )
plt.ylabel('axe des ordonnées' )
```

et ajouter un titre à l'aide de `plt.title`

```
plt.title('Mon graphique' )
```