Lycée Loges

Chapitre 5
Dessins, Graphes de fonctions

Simon Dauguet
simon.dauguet@gmail.com

12 décembre 2024

Grâce à l'algorithmie, on sait calculer beaucoup de choses et de façon automatique. Ce qui produit énormément de données. Et il n'est pas toujours aisé d'analyser une grande quantité de données. Un graphique est une solution rapide et efficace pour faire une première analyse rapide de ces données. Nous allons voir dans ce cours comment faire des graphiques.

Python étant un outil de programmation, il est capable de faire un peu de calcul (mais ce n'est pas l'idéal). Il est possible de le forcer à faire des graphique, mais il n'est pas vraiment fait pour. Les commandes pour faire des graphes sont assez pénibles et pas vraiment naturelles. Il va donc falloir prendre garde aux syntaxes, encore un peu plus que d'habitude.

Table des matières

1 Les tracés **2**
1.1 Bibliothèque de dessins et principes . . . 2
1.2 Ligne brisée . . . 2
1.3 Tracé d'une fonction . . . 4
1.4 Tracé d'une famille de fonctions . . . 6
1.5 Fonction lambda . . . 7

2 Stylisme graphique **9**
2.1 La mode en graphe . . . 9
2.1.1 Agir sur les axes . . . 9
2.1.2 Style du trait et légende . . . 11

3 Histogrammes **15**

1 Les tracés

1.1 Bibliothèque de dessins et principes

Le tracé d'un graphique se fait grâce à la fonction `plot` qui se trouve dans le paquet `pyplot` qui est lui même dans le module `matplotlib`. Donc pour utiliser la fonction `plot`, il faut faire

```
1 import matplotlib.pyplot
```

Et bien sûr, pour pouvoir faire des calculs, il faut aussi importer le paquet `math`. L'idéal est donc de commencer son fichier de procédures et fonctions par

```
1 from matplotlib import pyplot as plt
2 import math as mt
```

Avec ces deux lignes, on peut (presque) tout faire. Il faudra bien penser à mettre les préfixes `mt` ou `plt`.

Remarque :
Il existe une bibliothèque un peu plus pratique pour faire des dessins. C'est la bibliothèque `numpy`. Ce package est officiellement hors programme. Néanmoins, les concours de la banque Centrale l'utilise régulièrement (jusque là) dans leurs écrits ou leurs oraux (malgré les vives protestations des enseignants de CPGE).

Nous n'utiliserons pas cette bibliothèque. On se contentera du programme. Même s'il est vrai que `numpy` facilite grandement les choses.

Python ne fait pas dessin directement. Pas au sens où nous l'entendons, nous, être humain. Python décompose le processus en étapes plus fines que ce que nous faisons naturellement.

Plus précisément, Python ne sait dessiner que des lignes brisées. Elles paraissent plus ou moins lisse en fonction du nombre de points utilisés. Plus il y a de points, moins les brisures seront visibles à l'œil nu.

De plus, Python ne dessins rien directement. Il ne fait que générer des calques les uns à la suite des autres et les affiches tous en même temps au moment où lui demande de les afficher. Autrement dit, Python fait la distinction entre le dessins, le fait de tracé une ligne ; et le fait de regardé, d'observer la ligne dessinée.

Pour que tout fonctionne bien, il faudra donc commencer par lui dire qu'on commence un nouveau dessins, puis fabriquer les calques des dessins, et ensuite afficher ces calques.

Définition 1.1 (`figure` et `show()`) :
La fonction `plt.figure()` permet de définir à Python une nouvelle fenêtre de dessin dans laquelle sera affiché tous les calques qui seront créés jusqu'à la prochaine commande d'affichage des calques.

La fonction `plt.show()` permet d'afficher tous les calques créés depuis le dernier appel de la fonction `plt.show()` dans la fenêtre de dessins actuellement active.

1.2 Ligne brisée

Définition 1.2 (`plot(X,Y)` et `show()` (`matplotlib.pyplot`)) :
Une fois une bibliothèque de dessin importée (`matplotlib.pyplot`) et deux listes (ou tableaux) `X` et `Y` ayant le même nombre d'éléments (qui doivent obligatoirement être des nombres, flottants ou entiers), on peut utiliser `plot(X:list,Y:list)` pour dessiner la ligne brisée dont les sommets sont les points de coordonnées $(X[i], Y[i])$.

Techniquement, à cette étape, le graphe est virtuel. Il n'est pas encore affiché. C'est une sorte de calque. Il faut alors faire appel à la fonction `show()` (qui n'a pas d'argument) qui permet d'afficher l'ensemble de tous les calques générés depuis le dernier affichage.

```
>>> X=[x1,x2,...,xn]      # Listes des abscisses
>>> Y=[y1,y2,...,yn]      # Listes des ordonnées
>>> plt.plot(X,Y)         # Création du graphes des points (X[i],[Y[i])
>>> plt.show()            # Affichage du graphe
```

L'appel de la fonction `show()` ouvre une fenêtre supplémentaire avec tous les graphes. ATTENTION ! Il faut fermer la fenêtre dès qu'elle n'est plus utile. Sans quoi, l'interpréteur reste bloqué. Python ne peut en fait gérer qu'une fenêtre de graphe à la fois et pas plus. Et comme il n'aime pas vraiment manipuler des graphes, il faut être gentil avec lui et ne pas garder des graphes ouverts qui ne servent à rien.

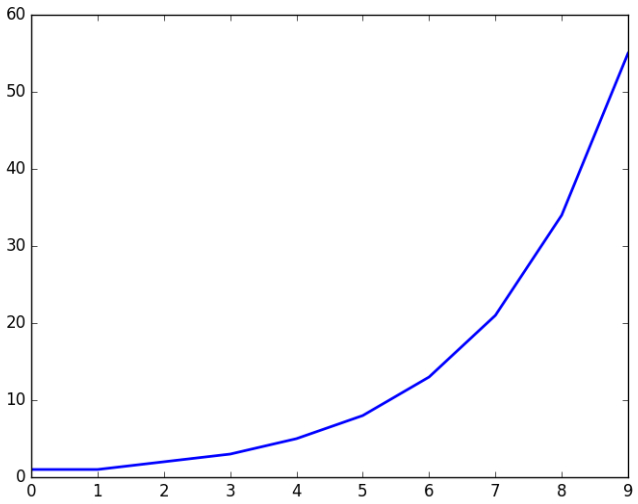
Remarque :
Certains logiciel (comme Spyder) ont une interface "user-friendly" et s'occupe de compléter le code automatiquement par un `show()` s'il est omis. Ce qui peut engendrer quelques problèmes dans certains situations. Mais surtout, c'est une très mauvaises habitudes que de se reposer sur un logiciel qui fait tout à notre place. Ne pas oublier qu'au concours, l'épreuve est à l'écrit. Et l'oubli de la fonction `show()` sera évidemment pénalisée.

On préférera donc utiliser des logiciels plus rustique que Spyder, mais plus formateurs.

Exemple 1.1 :
Graphe des n premiers termes de la suite de Fibonacci.

```
def GrapheFibo(n:int) -> None :
    plt.figure()
    X=range(0,n)
    Y=[1,1]
    for k in range(0,n-2) : # Création de la liste des ordonnées
        Y=Y+[Y[-1]+Y[-2]]
    plt.plot(X,Y)              # Création du graphe
    plt.show()                 # Affichage du graphe
```

On obtient alors le graphe suivant dans une nouvelle fenêtre pour $n = 10$:



Remarque :
D'une façon générale, Python ne gère pas très bien les graphiques. Un peu comme les fichiers. Il faut donc penser au maximum à fermer un graphe avant d'en ouvrir un second. Il n'aime pas trop faire des graphes. Mais on le force un peu. Il arrivera donc régulièrement que Python plante en faisant des graphes. Pour minimiser ces plantages, il faudra être attentionné avec Python et essayé de ne pas lui demander trop de dessins en même temps pour ne pas trop le surcharger.

Remarque :
Capytale gère les dessins dans une instance parallèle de l'éditeur. Lors de la création d'un graphe, le bouton permettant d'afficher le graphe clignote assez visiblement. En cliquant dessus, l'interpréteur sera caché et le graphe sera alors affiché. Il est possible à tout moment de basculer entre les deux en utilisant le bouton adéquat.

1.3 Tracé d'une fonction

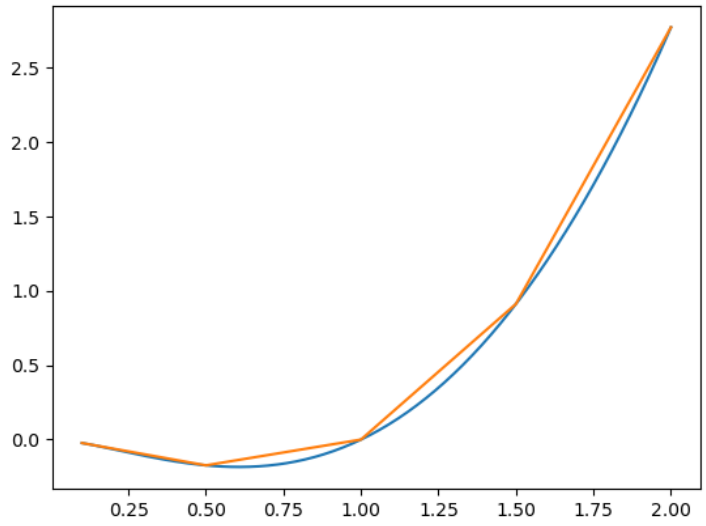
Définition 1.3 (Graphe d'une fonction) :
Le graphe d'une fonction est l'ensemble des points de coordonnées $(x, f(x))$ (voir le cours de maths pour plus de précision). Pour tracer avec Python le graphe d'une fonction, il faut donc une liste de valeurs `X` et la liste des images de ces valeurs `f(X)`. Il suffit de tracer le graphe de la ligne brisée de ces points.

Bien sûr, plus il y a de points, meilleure est l'approximation. Il est conseillé de prendre des listes avec un nombre raisonnable de points (au moins une cinquantaine).

Exemple 1.2 :

```
1 def Log() -> None :
2     plt.figure()
3     X = [0.1+k/100 for k in range(0,191)]        # Listes des abscisses sur ]0,2] pour un tracé
  "lisse"
4     Y = [0.1] + [k*0.5 for k in range(1,5)]      # Listes des abscisses sur ]0,2] pour un tracé
  "haché"
5     fX = [x**2*mt.log(x) for x in X]     # Création des images de X par la fonction x|-> x**2*ln(x)
6     fY = [y**2*mt.log(y) for y in Y]     # Création des images de Y par la même fonction
7     plt.plot(X,fX)         # Création du graphe "lisse"
8     plt.plot(Y,fY)         # Création du graphe "haché"
9     plt.show()             # Affichage du graphe "haché"
```

et on obtient les deux graphes



Exercice 1 :

On considère la fonction

$$f : x \mapsto \begin{cases} x^2 & \text{si } x < 3 \\ -3x + 18 & \text{sinon} \end{cases}$$

Tracer la courbe de f sur l'intervalle $[10, 10]$.

Remarque :
Il est évident que plus les points sont denses dans l'intervalle du tracé de la fonction, plus la courbe sera lisse. Mais il y a aussi

plus de calculs à faire. Il faut donc choisir le nombre de points du découpage de l'intervalle du tracé en conséquence (grand pour que la courbe ait une apparence lisse, mais pas trop pour que ce ne soit pas trop long à calculer ni qu'il y ait des problèmes aux bords).

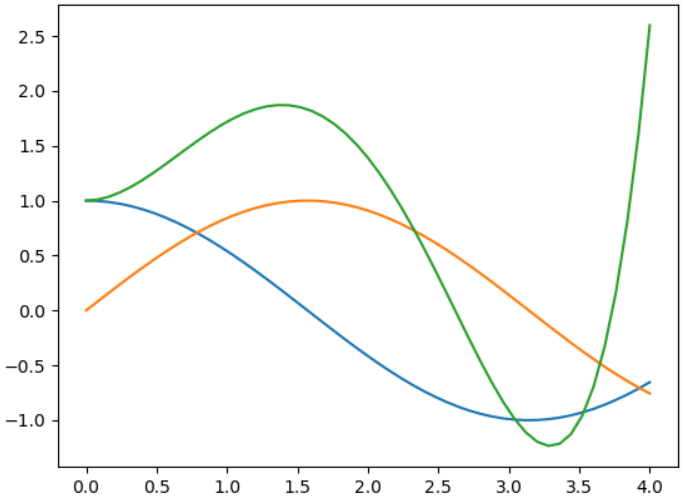
1.4 Tracé d'une famille de fonctions

Pour tracer une famille de fonctions, il suffit de charger les graphes que l'on souhaite et de faire un seul `show()` à la fin. Tous les graphes créés depuis la dernière implémentation de `show()` seront alors affichés sur le même graphe dans la même fenêtre. À chaque appel de la fonction `plot()` un calque est créé. Tous les calques actifs sont alors affichés en même temps lors de l'appel de la fonction `show()`. Et les calques sont alors effacés à la fermeture de la fenêtre avec les graphes.

Exemple 1.3 :

```
1  def f(x:float) -> float :
2      return(mt.exp(x)-x**3+x**2-x)
3
4  def Famille() -> None :
5      plt.figure()
6      X=[2*k/25 for k in range(0,51)]      # Création d'une liste de points dans [0,4]
7      C=[mt.cos(x) for x in X]
8      S=[mt.sin(x) for x in X]
9      F=[f(x) for x in X]
10     plt.plot(X,C)          # Création graphe du cos
11     plt.plot(X,S)          # Création graphe du sin
12     plt.plot(X,F)          # Création graphe de f
13     plt.show()             # Affichage de tous les graphes
```

et on obtient le graphe :



Le problème ici est que l'on ne sait pas quel courbe représente quelle fonction. On a un problème de légende. On a un problème de choix stylistique.

1.5 Fonction lambda

Il existe une version raccourcie pour définir une fonction simple. Ce sont les fonctions lambda. Elles sont tout à fait indiquées pour définir une fonction mathématique.

Définition 1.4 (Fonction lambda) :
Une fonction lambda se créé simplement avec la syntaxe :

```
1 >>> f = lambda var_1,...,var_n : expression(var_1,...,var_n)
```

Cette syntaxe peut être utilisé dans l'interpréteur ou dans l'éditeur. Elle peut également être incluse dans le corps d'une fonction plus compliquée.

L'intérêt est de pouvoir définir des fonctions simples (souvent mathématiques) sans avoir besoin de réutiliser la syntaxe (un peu lourde) d'une fonction algorithmique. Cette syntaxe permet de pouvoir alléger le script. Mais elle ne fonctionnera pas pour des fonctions trop compliqué. Elle est surtout utile pour donner un nom à un calcul effectué régulièrement dans un algorithme.
Exemple 1.4 :

```
>>> import math as mt
                      #Importation du paquet math qui contient la fonction racine
>>> f = lambda x : x**2+3*x-25   # f(x) = x^2 + 3x - 25
>>> g = lambda x : mt.sqrt(x)-abs(x) # g(x) = √x - |x|
>>> f(0),g(0)
(-25, 0.0)
>>> f(5),g(5)
(15, -2.7639320225002102)
>>> f(-5),g(-5)
(-15, nan)
>>> h = lambda x,y : x*y     # h(x,y) = xy
>>> h(0,1), h(1,0), h(-1,-1), h(5,-5)
(0, 0, 1, -25)
```

Remarque :
`nan` est la contraction de *not a number* est signifie que le résultat du calcul n'est pas défini, qu'il n'existe pas. Typiquement, c'est ce qu'on obtient lorsque l'on essaie de calculer une fonction mathématique en un réel où elle n'est pas définie. En général, cela indique un problème de définition.

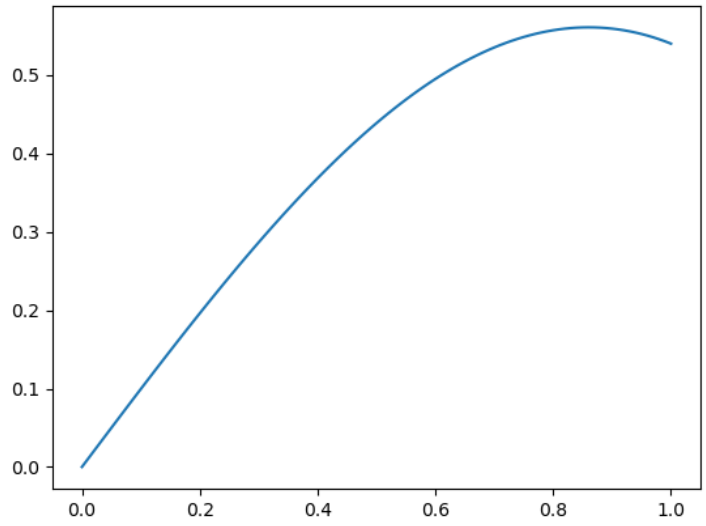
Remarque :
L'avantage des fonctions `lambda` est de permettre de pouvoir définir une fonction dans une fonction. Par exemple, si dans un algorithme on a besoin d'une fonction, jusque là, il fallait créés une nouvelle fonction à côté qui permettait de calculer les images d'un nombre par cette fonction polynomiale. Mais si cette fonction n'est utile que dans cet algorithme, ça alourdit le fichier pour pas grand chose.

Il est donc plus pratique de définir la fonction rapidement au sein de l'algorithme où on en a besoin.

Exemple 1.5 :

```
def graphe(f : "function") -> None :
    plt.figure()
    X = [k/100 for k in range(0,101)]    # Abscisses dans [0,1]
    FX = [f(x) for x in X]
    plt.plot(X,FX)
    plt.show()
```

On peut alors faire l'appele `>>> graphe(lambda x : x*mt.cos(x))` pour tracer le graphe de $x \mapsto x\cos(x)$ sur $[0,1]$. Ce qui donne :



2 Stylisme graphique

2.1 La mode en graphe : Titre, Légendes, Étiquettes, Couleurs, Styles

Il est possible de modifier l'apparence d'un graphe. Pour cela, il faut jouer entre des paramètres optionnels de la fonction `plot` et certaines fonctions supplémentaires à intercaler avant la fonction `show()`. En d'autres termes, il faut définir l'aspect du graphe avant d'afficher le graphe. Sinon, c'est trop tard.

2.1.1 Agir sur les axes

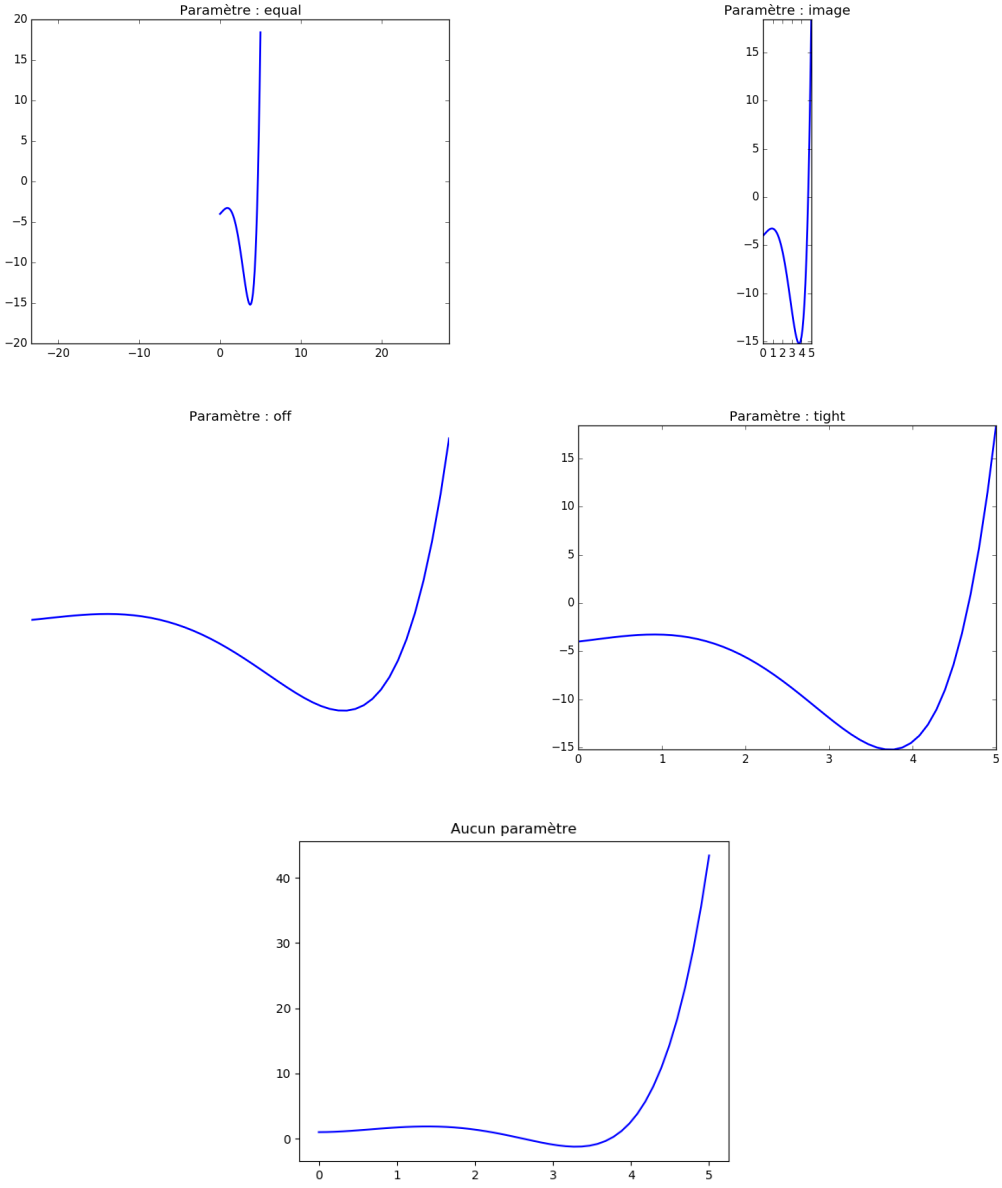
La fonction `axis` dans le paquet `matplotlib.pyplot` permet d'agir sur les axes du graphe. Beaucoup de choses sont possibles. Voir l'aide contextuelle pour plus de détails. Elle est a insérer avant la fonction `show()`. On retiendra surtout :

Commande	Description
`axis("equal")`	Permet d'avoir un repère orthonormé. Un cercle est alors vraiment un cercle.
`axis([xmin,xmax,ymin,ymax])`	Modifie la fenêtre du graphe pour correspondre au rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Le repère n'est donc a priori plus normé (mais toujours orthogonal).
`axis("image")`	Permet de choisir le cadre automatiquement pour contenir juste les données utiles dans un cadre orthonormé. Autrement dit, le cadre est ajusté autour du graphe de la fonction à afficher dans un repère orthonormé.
`axis("off")`	Supprime les axes. On obtient alors juste la courbe de la fonction sur un fond gris.
`axis("tight")`	Ajuste le cadre pour être carré et qu'il corresponde juste aux valeurs utiles du graphe a afficher. Autrement dit, on obtient une fenêtre carré avec des axes orthogonaux mais non normé qui colle parfaitement à la fonction (elle touche les quatre bords).
`xlabel("nom_abscisse")`	Rajouter un nom sur l'axe des abscisses.
`ylabel("nom_ordonnée")`	Rajouter un nom sur l'axe des ordonnées.

Exemple 2.1 :

```
1  def TestAxe() -> None :
2      plt.figure()
3      commande=input("Paramètre de axis (equal, image, off, tight, ou rien.)\n\n")
4      X = [k/20 for k in range(0,101)]          # Abscisses dans [0,5]
5      F = [f(x) for x in X]                     # la fonction f définie au-dessus
6      plt.plot(X,F)                           # Création du graphe de f
7      if commande!="" :
8          plt.axis(commande)                    # Modification des axes
9          plt.title("Paramètre : "+commande)   # Titre du graphe
10     else :
11         plt.title("Aucun paramètre")
12     plt.show()                                 # Affichage du graphe
```

permet de tester les différents paramètres d'axes possibles.



2.1.2 Style du trait et légende

Remarque :
Les différents styles de tracés présentés ici ne forment pas une liste exhaustive de tout ce que peut propose `pyplot`. Il est fortement recommandé (et même conseillé) de se référer à la documentation de la fonction `plot` pour avoir toutes les possibilités. De même, on peut customiser les graphes beaucoup plus que ce

qui est proposé dans ce cours. Là encore, il est recommandé de faire appel à la documentation du package `pyplot` pour plus de détails.

Définition 2.1 (`legend()` (`matplotlib.pyplot`)) :
La fonction **`legend()`** (sans argument) permet d'afficher l'éventuelle légende créée lors de la création des graphes à partir de l'argument **`label`** de la fonction **`plot`**. Elle doit se placer *avant* la fonction **`show()`**.

Pour faire une légende, il faut donc créer une étiquette (*label* en anglais) lors de la création d'une courbe. Puis, la fonction **`legend()`** permet alors l'affichage de ces étiquettes. Et pour créer une étiquette digne de son nom, il faut choisir un nom et un style particulier pour une courbe donnée. Les commandes suivantes sont donc à mettre dans la fonction **`plot`** correspondante.

Argument de **`plot`**	Description
`linewidth=nb` (ou **`lw=nb`**)	Épaisseur du trait égale **`nb`**.
`label="legende"`	Ajoute une légende pour le graphe. C'est le nom de la courbe. Dans la légende, sera affiché le style du trait choisit pour la courbe et le nom de la courbe.
`"style"`	Une chaîne de caractère qui détermine le style du trait. Cette chaîne doit contenir au maximum 2 caractère. L'un pour la couleur, l'autre pour le style du tracé (pointillés etc). L'ordre n'a pas d'importance puisqu'il n'y a pas d'intersections entre les différentes options possibles. Elles sont présentées juste après.

Pour le style du trait, on peut choisir une couleur et, un style de trait et un style de point maximum. Toutes les combinaisons sont possibles tant qu'un seul style de trait n'est imposé et qu'un seul style de point et qu'une seule couleur. Autrement dit, on ne peut choisir au maximum qu'un seul style dans chacune des catégories.

Styles de traits

Commandes	Signification
`"-"`	Trait continu
`"--"`	Pointillés avec des traits (pointillés classiques)
`"-."`	Pointillés trait et point en alternance
`":"`	Pointillés avec des points

Commandes	Signification
`"b"`	Bleu
`"g"`	Vert
`"r"`	Rouge
`"c"`	Turquoise (cyan)
`"m"`	Magenta (violet)
`"y"`	Jaune
`"k"`	Noir (black)
`"w"`	Blanc

Styles de points

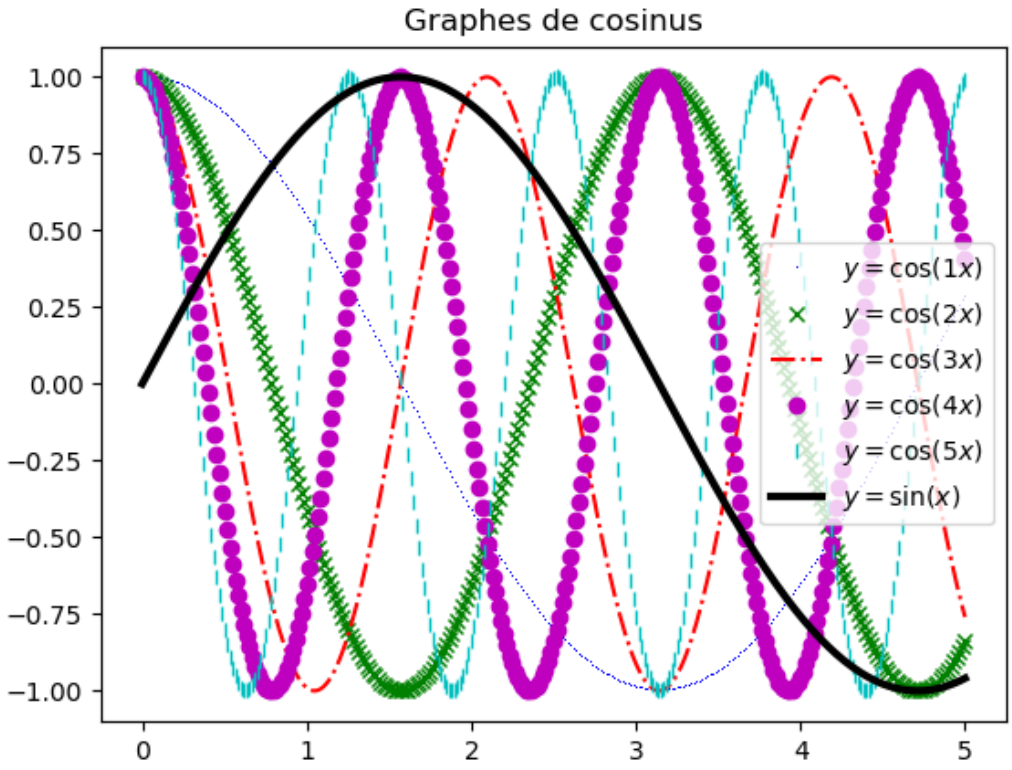
Commandes		Signification
`"v"`	▼	Triangles pleins vers le bas
`"^"`	▲	Triangles pleins vers le haut
`"<"`	◀	Triangles pleins vers la gauche
`">"`	▶	Triangles pleins vers la droite
`"1"`	⟙	Picots vers le bas
`"2"`	⟘	Picots vers le haut
`"3"`	≺	Picots vers la gauche
`"4"`	≻	Picots vers la droite
`"s"`	■	Carrés pleins
`"p"`	⬠	Pentagones pleins
`"*"`	★	Étoiles
`"h"`	⬡	Hexagones pleins
`"H"`	⬣	Hexagones pleins (d'autres)
`"+"`	+	Croix en forme de +
`"x"`	×	Croix en forme de ×
`"D"`	◆	Diamants épais pleins (losanges)
`"d"`	⧫	Diamants fins
`"|"`	|	Barres verticales
`"_"`	_	Barres horizontales
`","`	.	Pixels (on ne voit presque rien)
`"."`	•	Points
`"o"`	●	Gros points ronds pleins

Il est possible également de rajouter un titre pour la fenêtre et même une grille. Cela se fait avec les commandes `title("MonTitre")` et `grid()` qui doivent figurer avant le `show()`.

Exemple 2.2 :

```
def Style() -> None :
    plt.figure()
    X=[k/60 for k in range(0,301)]   # Abscisses dans [0,5]
    couleurs=['b','g','r','m','c']       # Choix de couleurs bleu, gris, rouge, magenta et cyan
    style=[',','x','-.','o','|']         # Choix de style de pointillés et de points
    for k in range(1,6) :                # Création des différents graphes avec les styles et
légendes
        C = [mt.cos(k*x) for x in X]
        plt.plot(X,C, couleurs[k-1]+style[k-1],label=f"$y=\cos({k}x)$")
    plt.title("Graphes de fonctions trigonométriques")
    S = [mt.sin(x) for x in X]
    plt.plot(X,S, "k", linewidth=3, label="$y=\sin(x)$")
    plt.legend()                              # Création de la légende
    plt.show()                                # Affichage du graphe
```

et on obtient le graphe



Bien sûr, vous pourrez retrouver toutes ces indications dans l'aide de la commande `plot`. Il est fortement conseillé de la consulter régulièrement plutôt que d'apprendre par cœur tous les styles différents.

3 Histogrammes

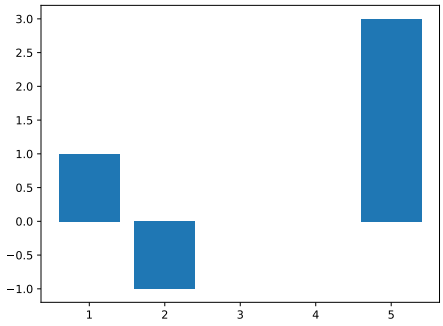
Il est possible de faire des histogrammes (diagrammes en barres) également avec la commande suivante :

Définition 3.1 (`bar()`) :
Dans le package `matplotlib.pyplot`, la fonction `bar(X:list, H:list)` permet de faire des diagrammes en barres. Elle prend en argument deux listes `X` et `H` de même taille. La liste `X` représente la position des barres sur l'axe des abscisses ; et la liste `H` correspond aux hauteurs des barres.

Il faut utiliser la fonction `show()` pour faire apparaître le diagramme.

Exemple 3.1 :

```
>>> import matplotlib.pyplot as plt
>>> plt.bar([1,2,5],[1,-1,3])
<BarContainer object of 3 artists>
>>> plt.show()
```



Évidemment, on peut choisir la couleur des barres comme pour les fonctions avec les mêmes options. On peut également superposées les graphes.

Exemple 3.2 :

```
>>> import matplotlib.pyplot as plt
>>> Age = [16,17,18,19,20,21]
>>> Nb =  [1,0,5,14,5,2]
>>> plt.bar(Age,Nb)
<BarContainer object of 6 artists>
>>> plt.plot(Age,Nb,marker="o", color="red")
[<matplotlib.lines.Line2D object at 0x7370d77c2c20>]
>>> plt.show()
```