Représentations graphiques avec python

▶ Présentation du module matplotlib.pyplot

Pour faire des représentations graphiques avec Python, il faudra éventuellement installer au préalable le module **matplotlib.pyplot**.

Par ailleurs, il est intéressant d'importer ce module en utilisant un « alias » grâce à l'instruction `import ... as ...` : cela permet d'utiliser les fonctions du module en utilisant le préfixe `graph` à la place de `matplotlib.pyplot`, beaucoup plus long et moins lisible.

Vidéo
Installer un module
hatier-clic.fr/ma1ra2a

Exemple 1

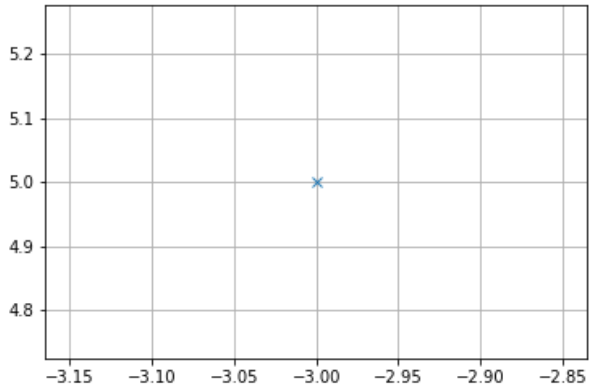
```
1  import matplotlib.pyplot as graph
2  graph.plot(-3,5,marker="x")
3  graph.grid()
4  graph.show()
```

L'instruction **plot** trace le point de coordonnées (3 ; 5) sous la forme d'une croix, dans un repère automatiquement créé.

L'instruction **grid** fait apparaître un quadrillage.

L'instruction **show** affiche le résultat final dans une fenêtre de sortie : elle sera toujours obligatoire.

>>> Graphique obtenu :



Le cadre gradué ne constitue pas les axes du repère.

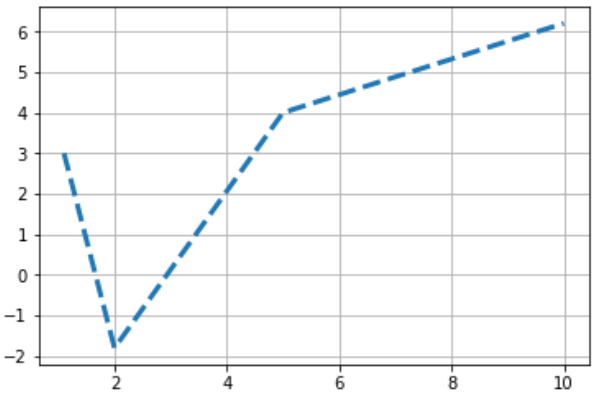
Exemple 2

```
1  import matplotlib.pyplot as graph
2  x=[1.1,2,5,10]
3  y=[3,-1.8,4,6.2]
4  graph.plot(x,y,linestyle="--",linewidth=3)
5  graph.grid()
6  graph.show()
```

Lorsque ses paramètres sont des listes de nombres, l'instruction `plot` trace une ligne polygonale dont les abscisses des sommets sont données par la première liste et leurs ordonnées respectives par la seconde. Ici, cette ligne a pour premier sommet le point de coordonnées (1,1 ; 3).

Les paramètres **linestyle** et **linewidth** indiquent respectivement le type de ligne utilisée ("--" pour pointillés, "-" pour trait plein et " " pour ne pas relier les points) et l'épaisseur de celle-ci.

>>> Graphique obtenu :



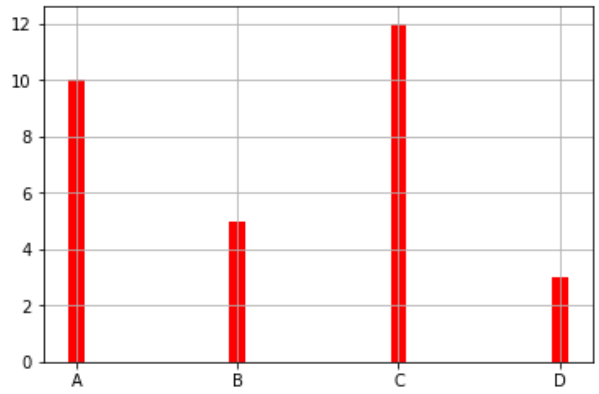
Exemple 3

```
1  import matplotlib.pyplot as graph
2  effec=[10,5,12,3]
3  classes=["A","B","C","D"]
4  graph.bar(classes,effec,color="red",width=0.1)
5  graph.grid()
6  graph.show()
```

L'instruction **bar** trace un diagramme en bâtons à partir d'une série statistique dont les classes sont données par la première liste et les effectifs par la seconde.

Le paramètre **color** indique la couleur des barres et le paramètre **width** indique leur largeur.

>>> Graphique obtenu :



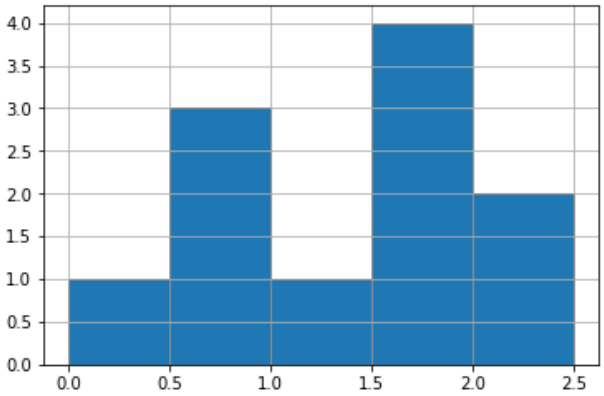
Exemple 4

```
1  import matplotlib.pyplot as graph
2  data=[0.9,0.5,2,0.8,1.2,2.2,0.2,1.5,1.8,1.5,1.9]
3  classes=[0,0.5,1,1.5,2,2.5]
4  graph.hist(data,classes,edgecolor="black")
5  graph.grid()
6  graph.show()
```

L'instruction **hist** trace un histogramme dont les données « brutes » fournies par la première liste sont rangées selon les intervalles de même amplitude fournis par la seconde. Ici, la première classe est l'intervalle [0 ; 0,5[, la deuxième [0,5 ; 1[, etc.

Le paramètre **edgecolor** détermine la couleur des côtés des rectangles.

>>> Graphique obtenu :



Une fois classées, les données de la liste *data* correspondent au tableau ci-dessous.

Intervalles	[0 ; 0,5[	[0,5 ; 1[	[1 ; 1,5[	[1,5 ; 2[	[2 ; 2,5[
Effectifs	1	3	1	4	2

Le programme affiche alors l'histogramme correspondant.

▶ Quelques programmes commentés

A. Le triangle de Sierpinsky

Nous allons construire une approximation du triangle de Sierpinsky, point par point, en suivant l'algorithme suivant :

A, B, C et M sont quatre points de coordonnées respectives A(–10 ; 0), B(10 ; 0), C(0 ; 10) et M(0 ; 0).
Répéter 5 000 fois
  Marquer la position du point M
  Choisir aléatoirement l'un des trois points A, B ou C, que l'on nommera N
  Redéfinir M comme le milieu de [MN]
Fin Répéter

```
1  import matplotlib.pyplot as graph
2  import random
3
4  A=[-10,0]
5  B=[10,0]
6  C=[0,10]
7  M=[0,0]
8  ABC=[A,B,C]
9  for point in range(0,5000):
10   graph.plot(M[0],M[1],marker="o",markersize=1)
11   N=ABC[random.randint(0,2)]
12   M[0]=(N[0]+M[0])/2
13   M[1]=(N[1]+M[1])/2
14 graph.show()
```

Les coordonnées des points A, B, C et M sont stockées sous la forme de listes.

ABC est une liste contenant les coordonnées des points A, B et C : elle sera utile lorsqu'il faudra choisir aléatoirement l'un de ces trois points.

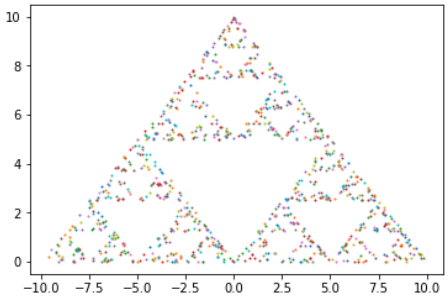
On trace le point M dont l'abscisse est le premier élément de la liste M et l'ordonnée le second.
Le paramètre **`markersize`** indique la taille de la trace (qui est ici un disque).

N est l'un des trois éléments de la liste ABC, dont l'indice est choisi aléatoirement parmi 0, 1 et 2.

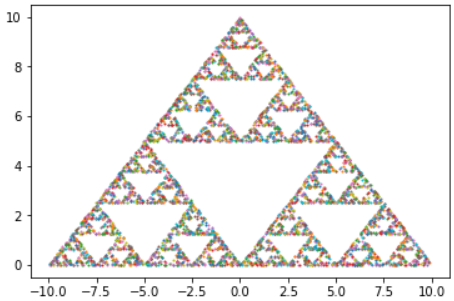
Les coordonnées de M sont modifiées pour correspondre au milieu de [MN].

>>> Graphiques obtenus :

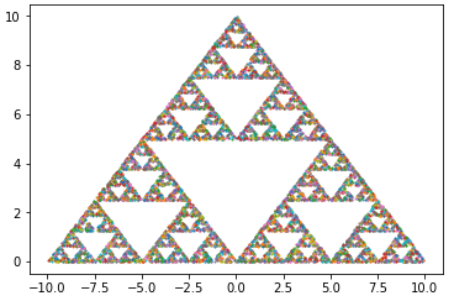
pour 1 000 points



pour 5 000 points
(programme ci-dessus)



pour 10 000 points



B. Le vol parabolique d'un avion

Il s'agit ici de tracer la représentation graphique de la fonction $f : t \mapsto \frac{-900}{121}t^2 + \frac{1800}{11}t + 7600$ qui modélise l'altitude d'un avion lors d'un vol parabolique en fonction du temps t sur l'intervalle [0 ; 22] (▶ Exercice 73 p. 91).

```
1  import matplotlib.pyplot as graph
2  import math
3
4  def f(t):
5      return -900*t**2/121+1800*t/11+7600
6
7  temps=[]
8  altitudes=[]
9  t=0
10 while t<22.1:
11     temps.append(t)
12     altitudes.append(f(t))
13     t=t+0.1
14
15 graph.plot(temps,altitudes,color="red",linewidth="2")
16 graph.title("Altitude d'un avion en fonction du temps
   lors d'un vol parabolique")
17 graph.xlabel("Temps (en s)")
18 graph.ylabel("Altitude (en m)")
19 graph.text(2,8450,"Alt. max: 8500 m",fontsize=8)
20 graph.grid()
21 graph.show()
```

La fonction *f* renvoie l'altitude en fonction du temps *t* en paramètre.

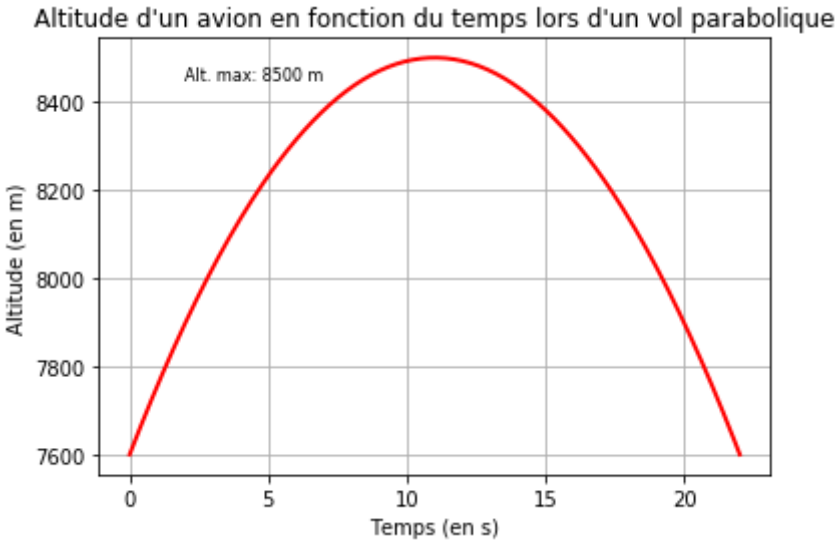
temps est une liste remplie par des nombres décimaux compris entre 0 et 22, le premier valant 0 et les suivants s'obtenant en ajoutant 0,1 au précédent.

altitudes contient les images des nombres de la liste *temps* par la fonction *f*.

Les instructions **`title`**, **`xlabel`** et **`ylabel`** permettent respectivement de donner un titre au graphique et de légender les graduations horizontales et verticales.

L'instruction **`text`** ajoute un texte, ici aux coordonnées (2 ; 8 450), avec une taille de caractères donnée par le paramètre **`fontsize`**.

>>> Graphique obtenu :



En ligne 10, pourquoi effectuer les instructions de la boucle « tant que t < 22,1 » plutôt que « tant que t ≤ 22 » ?

Les ordinateurs ne travaillent pas avec les nombres décimaux mais avec les nombres flottants, qui sont des approximations des décimaux.
Pour s'en convaincre, il suffit d'ajouter l'instruction `print(t)` dans le corps de la boucle : on constate ainsi qu'en lui ajoutant successivement 0,1, la variable *t* n'atteint pas exactement 22, mais qu'elle dépasse cette valeur très légèrement (22.000000000000043).
Il faut donc stopper la boucle au dixième suivant.

C. Lancé de dés

On considère l'expérience aléatoire qui consiste à lancer simultanément dix dés équilibrés à six faces (numérotés de 1 à 6) et à relever la somme obtenue.
On souhaite simuler 100 000 répétitions de cette expérience et construire un diagramme en bâtons représentant les sommes obtenues (entre 10 et 60).

```
1  import matplotlib.pyplot as graph
2  import random
3
4  def lancer():
5      resultat=0
6      for de in range(0,10):
7          resultat=resultat+random.randint(1,6)
8      return resultat
9
10 donnees_brutes=[]
11 for experience in range(100000):
12     donnees_brutes.append(lancer())
13
14 sommes=[]
15 effectifs=[]
16 for s in range(10,61):
17     sommes.append(s)
18     effectifs.append(donnees_brutes.count(s))
19
20 graph.bar(sommes,effectifs,width=0.25)
21 graph.xlabel("Sommes de 10 dés")
22 graph.ylabel("Effectifs")
23 graph.grid()
24 graph.show()
```

La fonction *lancer* simule le lancer de dix dés et renvoie leur somme.

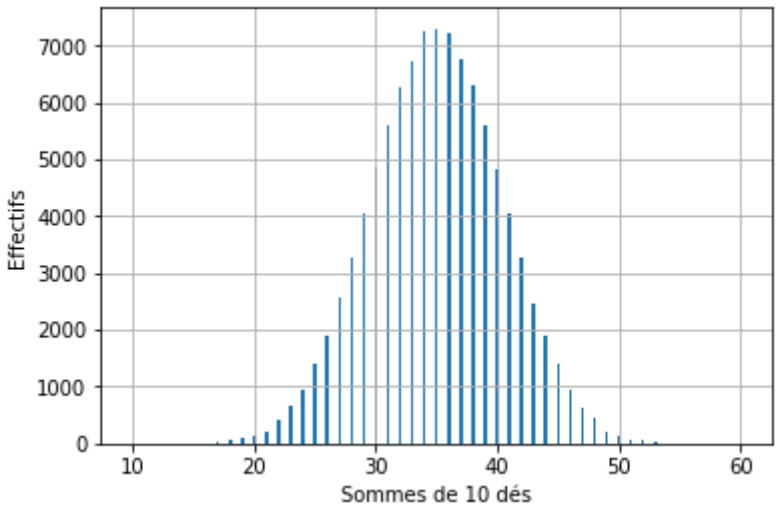
On choisit aléatoirement dix nombres compris entre 1 et 6 inclus, que l'on additionne au fur et à mesure à la variable *resultat*.

*donnees_brute*s est une liste qui contient les 100 000 sommes obtenues.

La liste *sommes* contient toutes les sommes potentielles (les entiers de 10 à 60.
La liste *effectifs* contient le nombre de fois que chacune de ces sommes a été obtenue, grâce à l'instruction **`count`**.

On trace enfin le diagramme en bâtons représentant les effectifs de chaque somme obtenue.

>>> Graphique obtenu :



D. Jeu de fléchettes

On considère l'expérience aléatoire qui consiste à lancer une fléchette dans une cible carrée de 20 cm de côté et à relever la distance entre le centre du carré et le point d'impact de la fléchette.
On souhaite simuler n répétitions de cette expérience et :
– construire un histogramme classant les distances obtenues dans des intervalles d'amplitude 1 ;
– visualiser les impacts dans une seconde représentation graphique.

```
1  import matplotlib.pyplot as graph
2  import random
3  import math
4
5  def lancer(n):
6      liste_x=[]
7      liste_y=[]
8      distances=[]
9      classes=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
10
11     for experience in range(n):
12         x=random.uniform(-10,10)
13         liste_x.append(x)
14         y=random.uniform(-10,10)
15         liste_y.append(y)
16         distances.append(math.sqrt(x**2+y**2))
17
18     graph.figure(1)
19     graph.hist(distances,classes,edgecolor="black")
20     graph.title("Distances au centre du carré")
21     graph.xlabel("Distances (en cm)")
22     graph.ylabel("Effectifs")
23
24     graph.figure(2)
25     graph.plot(liste_x,liste_y,"o",markersize=2)
26     graph.axis("equal")
27     graph.title("Impacts de la fléchette dans
   la cible carrée")
28     graph.grid()
29     graph.show()
30
31 lancer(1000)
```

liste_x, *liste_y* et *distances* sont des listes qui contiendront respectivement les n abscisses de la fléchette, ses n ordonnées et les n distances la séparant du centre du carré.

classes est la liste décrivant les intervalles nécessaires à la construction de l'histogramme. La distance maximale entre le centre du carré et le point d'impact de la fléchette est la demi-diagonale du carré ; cette distance vaut $\frac{\sqrt{20^2+20^2}}{2} \approx 14{,}1$ cm (d'après le théorème de Pythagore). Le dernier intervalle est donc [14 ; 15[.

Une abscisse x et une ordonnée y décimales sont choisies aléatoirement entre –10 et 10 à l'aide de la fonction `random.uniform`.

On ajoute à la liste *distances* la distance entre ce point de coordonnées $(x\,;y)$ et le centre du carré, de coordonnées (0 ; 0).

`graph.figure(1)` crée la première fenêtre de sortie. Les instructions qui suivent s'exécuteront dans cette fenêtre.

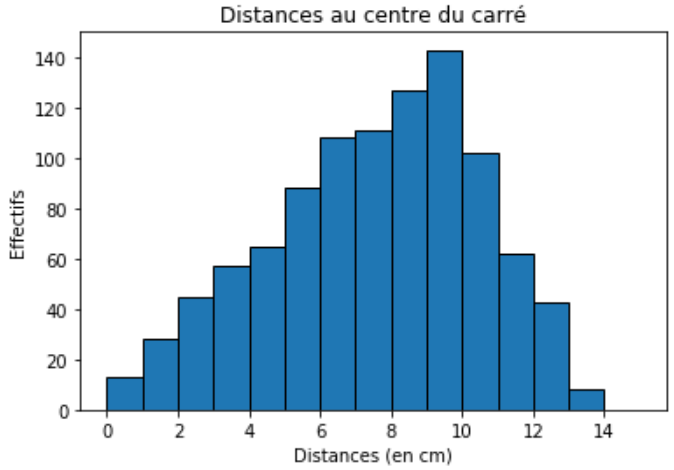
`graph.figure(2)` crée une deuxième fenêtre. Les instructions qui suivent s'exécuteront dans cette fenêtre.

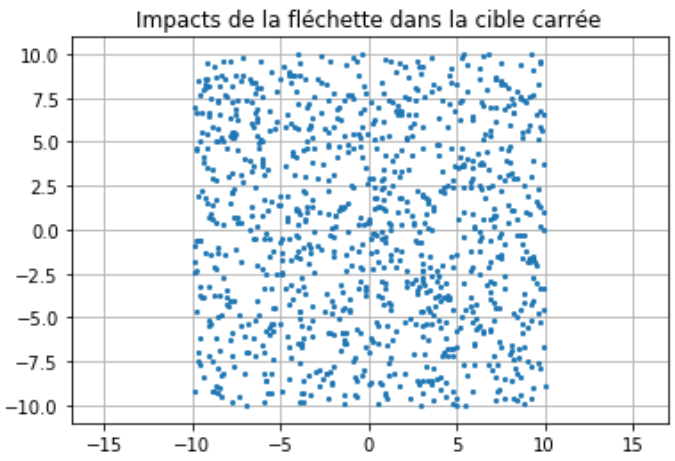
L'instruction `axis("equal")` contraint le repère à être orthonormé, afin que la représentation du carré soit exacte.

`graph.show` concerne toutes les fenêtres simultanément.

On appelle la fonction pour 1 000 lancers (dans le programme ou la console).

>>> Graphiques obtenus :





Pour aller plus loin…

Il est possible d'animer ces deux représentations graphiques pour visualiser le déroulé de l'expérience.

Pour cela, il faut reprendre le programme en profondeur :

```
1  import matplotlib.pyplot as graph
2  import random
3  import math
4
5  def lancer(n):
6      distances=[]
7      classes=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
8
9      graph.figure(1)
10     graph.title("Distances au centre du carré")
11     graph.xlabel("Distances (en cm)")
12     graph.ylabel("Effectifs")
13     graph.grid()
14
15     graph.figure(2)
16     graph.title("Impacts de la fléchette dans
   la cible carrée")
17     graph.axis("equal")
18     graph.grid()
19
20     for experience in range(n):
21         x=random.uniform(-10,10)
22         y=random.uniform(-10,10)
23         d=math.sqrt(x**2+y**2)
24         distances.append(d)
25
26         graph.figure(1)
27         graph.cla()
28         graph.hist(distances,classes,edgecolor="black")
29
30         graph.figure(2)
31         graph.plot(x,y,"o",color="blue",markersize=2)
32
33         graph.pause(0.1)

lancer(1000)
```

On crée deux fenêtres de sortie et on les prépare comme précédemment (légende, quadrillage, repère orthonormé).

La fléchette est positionnée aléatoirement et sa distance au centre du carré est ajoutée à la liste *distances*.

`graph.figure(1)` : les instructions de constructions s'exécuteront dorénavant dans la première fenêtre.

`graph.cla` efface l'histogramme tracé lors du précédent lancer pour qu'on puisse y redessiner l'histogramme à partir de la version actualisée de la liste *distances*.

`graph.figure(2)` : les instructions de constructions s'exécuteront dorénavant dans la seconde fenêtre ; on y ajoute un nouveau point.

`graph.pause(0.1)` actualise l'affichage de toutes les fenêtres (équivalent de `show`) et marque une pause de 0,1 seconde.