Introduction à Matplotlib

23 janvier 2024

L'objectif de ce texte est de présenter les bases de `matplotlib`, dont la syntaxe est pour le moins confuse. En effet :

— Tout programme de dessin commence par le cryptique `fig, ax = plt.subplots()` (ou une variante).

— Puis, `fig` n'apparaît presque plus jamais. C'était bien la peine d'inventer l'amorce ci-dessus !

— Si l'on utilise l'amorce `plt.subplots()`, toute figure est donc une sous-figure. Certes, on pourrait aussi utiliser l'amorce plus simple `fig = plt.figure()`, mais attention : les commandes pour les sous-figures ne sont, bien sûr, pas les mêmes que les commandes pour les figures (si l'on veut ajouter un titre à la figure, modifier les axes, etc.). Par exemple, `plt.title("Titre")` change le titre de la figure, mais il faut utiliser `ax.set_title("Titre")` si on passe par les sous-figures.

— Imaginons que l'on souhaite dessiner un cercle (de centre (a, b) et de rayon R). Un cercle, hein, pas une courbe compliquée. C'est la commande `ax.add_patch(plt.Circle((a, b), R))`. À condition qu'on ait pris la bonne habitude d'utiliser des sous-figures : sinon, il faudra utiliser l'absurde

```
fig.get_axes()[0].add_patch(plt.Circle((a, b), R))
```

À titre de comparaison, l'équivalent avec `SageMath` serait le très lisible `fig += circle((a, b), R)`.

Bref : c'est l'horreur.

1 Amorce

Tout d'abord, importons quelques modules pour pouvoir dessiner des objets intéressants.

```
import math
import numpy as np
import numpy.random as random
import scipy.stats as scs
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
```

Les quatre premiers imports concernent des bibliothèques mathématiques ; en particulier, on manipulera des tableaux de nombres de type *numpy array*, grâce à la bibliothèque `numpy`. Le cinquième import est celui qui appelle la bibliothèque `matplotlib` ; on abrégera toutes les commandes `matplotlib.pyplot.command` en `plt.command`. Le dernier import est facultatif : il permet d'avoir des graphiques un peu plus jolis, à condition d'avoir `seaborn` installé (ce qui est à une ligne de commande près : `pip install seaborn`).

Comme expliqué dans l'introduction, pour créer un nouveau dessin, on utilise l'amorce suivante :

```
fig, ax = plt.subplots()
```

La taille de la figure peut être donnée en argument optionnel : `plt.subplots(figsize=(a, b))` force la figure à occuper un rectangle de taille $a \times b$. Alternativement, on peut a posteriori redimensionner toute la figure avec la commande `fig.set_size_inches(a, b)`.

Si l'on veut une figure avec plusieurs dessins séparés, c'est très simple : on rajoute le nombre de lignes et de colonnes d'un tableau dont chaque case est une sous-figure. Ainsi,

```
fig, axes = plt.subplots(2,3)
```

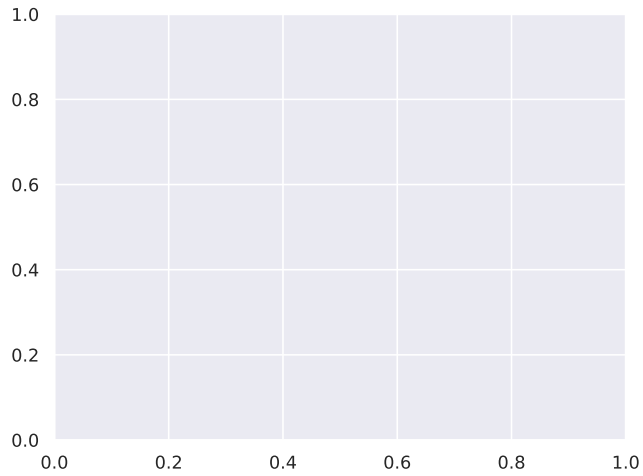
crée une figure avec 2 lignes de 3 sous-figures chacune. On opérera sur chaque sous-figure d'indices i, j avec `axes[i,j].command` et les `command` qu'on expliquera plus loin. Pour simplifier la syntaxe, on utilisera toujours dans ce qui suit l'amorce avec des sous-figures, même s'il y en a une seule. Bien sûr, on peut appeler `fig` et `axes` différemment, en particulier s'il faut manipuler plusieurs figures et sous-figures.

2 Dessin

Pour dessiner une figure créée avec l'amorce ci-dessus, on peut utiliser la commande `plt.show()`. Ainsi, un programme typique de dessin est :

```
def dessin(args):
        fig, ax = plt.subplots()
        ...
        plt.show()
```

Voici le résultat du programme `dessin()` si l'on met juste la ligne d'amorce et la ligne de dessin :

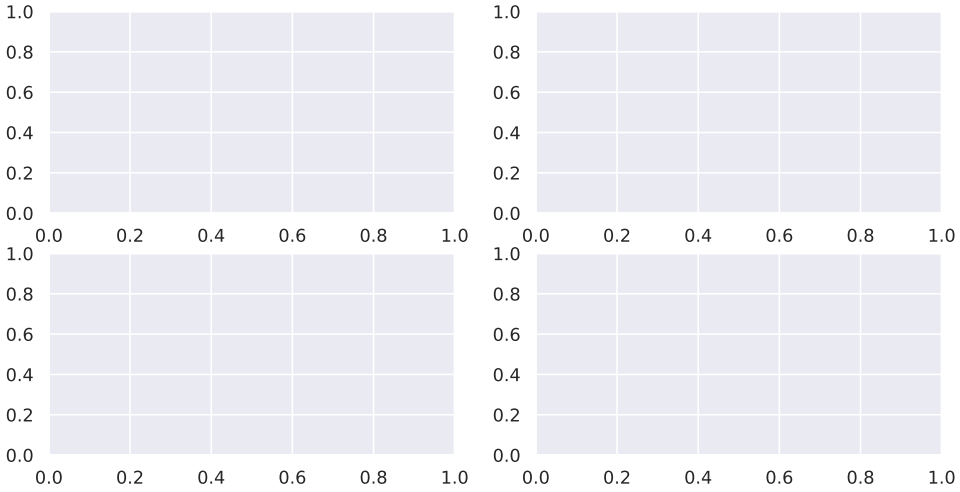


Si l'on souhaite sauvegarder le dessin plutôt que le dessiner, on peut utiliser la commande `fig.savefig("path/to/file.pdf")`, avec un chemin valide (c'est comme cela qu'on a obtenu tous les dessins du texte).

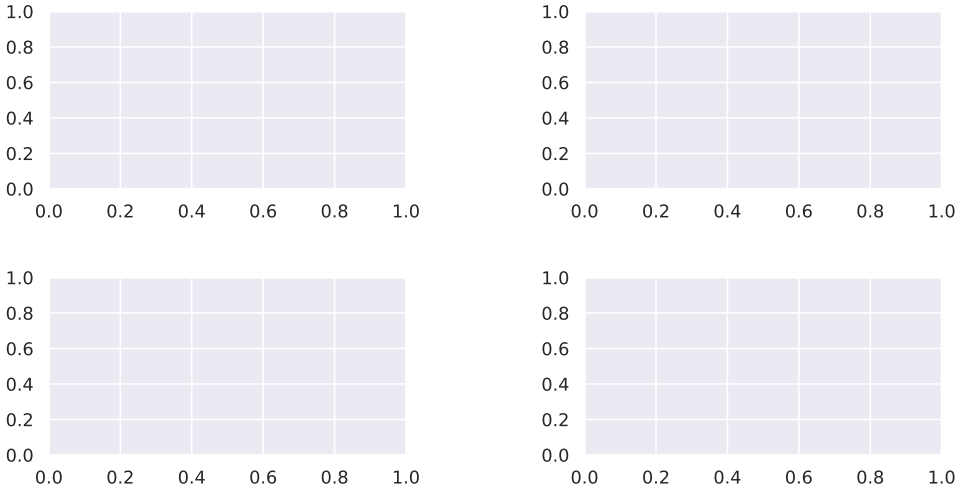
Exemple 1. Le programme suivant dessine une grille de sous-figures vides avec a lignes et b colonnes :

```
def dessin_vide(a, b):
        fig, axes = plt.subplots(a, b)
        plt.show()
```

Voici le résultat avec $a = b = 2$:



C'est un peu serré : on peut rajouter de l'espacement entre les sous-figures grâce à la commande `plt.subplots_adjust(wspace=x, hspace=y)`, avec des valeurs réelles pour x et y. Par exemple, $x = y = 0.5$ donne :



3 Décorations

Chaque (sous-)figure occupe une certaine boîte $(x_{\min}, x_{\max}) \times (y_{\min}, y_{\max})$, et a éventuellement une échelle sur les deux axes, des labels pour ces axes, et un titre. Pour une sous-figure, on peut spécifier tout cela avec les commandes :

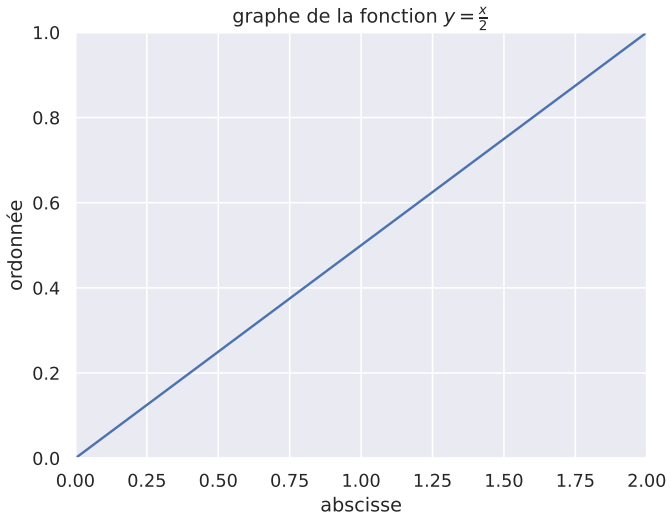
```
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
ax.set_xlabel("label des abscisses")
ax.set_ylabel("label des ordonnées")
ax.set_title("un titre")
```

avec des valeurs réelles pour les bornes $x_{\min}, x_{\max}, y_{\min}, y_{\max}$.

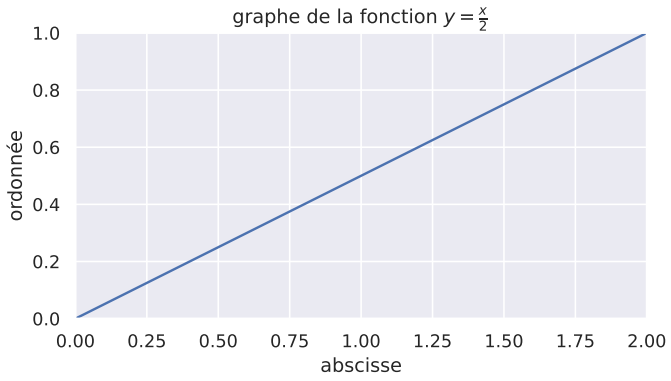
Exemple 2. La suite de commandes

```
fig, ax = plt.subplots()
ax.plot([0,2], [0,1])
ax.set_xlim(0, 2)
ax.set_ylim(0, 1)
ax.set_xlabel("abscisse")
ax.set_ylabel("ordonnée")
ax.set_title("graphe de la fonction $y=\frac{x}{2}$")
plt.show()
```

dessine ceci :



Comme on le verra plus loin, `ax.plot([0,2], [0,1])` ajoute un segment entre les points de coordonnées $(0, 0)$ et $(2, 1)$. Le graphe ci-dessus ne respecte pas vraiment les échelles (l'ordonnée croît plus vite que l'abscisse). Pour remédier à cela, on peut utiliser la commande `ax.set_aspect(1)` :



Il y a encore plusieurs ajustements possibles :

— Si l'on veut donner un titre global à la figure (qui peut contenir plusieurs sous-figures contenues dans le tableau `axes`), on peut le faire avec `fig.suptitle("un titre global")`.

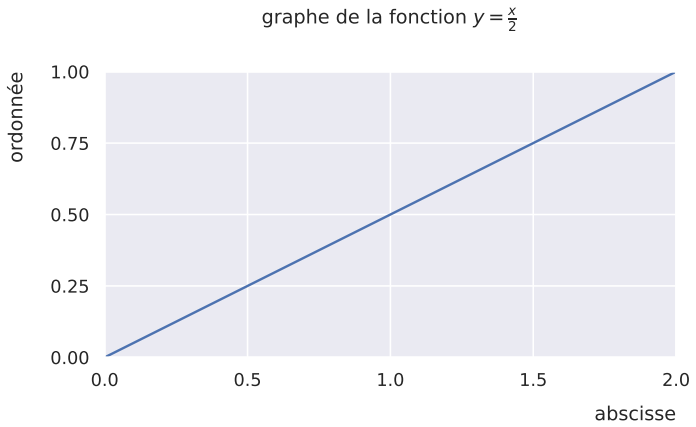
— Dans certains cas, on peut souhaiter retirer les axes : c'est la commande `ax.set_axis_off()`.

— On peut aussi décider de la graduation sur chaque axe. Par exemple, pour forcer N graduations entre $x_{\min}$ et $x_{\max}$, on peut utiliser `ax.set_xticks(np.linspace(xmin, xmax, N))`

— Les labels des axes et le titre peuvent être positionnés différemment, grâce aux options des commandes `set_xlabel`, `set_ylabel` et `set_title`.

Exemple 3. Un aspect optimal pour la figure précédente est obtenu avec les commandes suivantes :

```
fig, ax = plt.subplots()
ax.plot([0,2], [0,1])
ax.set_xlim(0, 2)
ax.set_ylim(0, 1)
ax.set_xlabel("abscisse", labelpad=10, loc="right")
ax.set_ylabel("ordonnée", labelpad=15, loc="top")
ax.set_title("graphe de la fonction $y=\\frac{x}{2}$", pad=30)
ax.set_xticks(np.linspace(0, 2, 5))
ax.set_yticks(np.linspace(0, 1, 5))
ax.set_aspect(1)
plt.show()
```



4 Objets

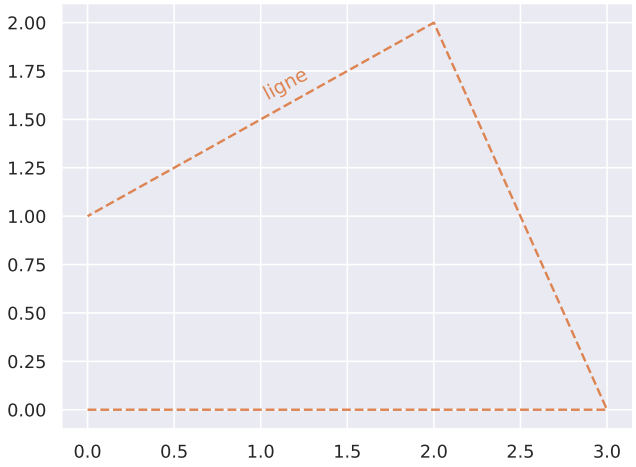
Expliquons maintenant comment dessiner divers objets sur une sous-figure. Toutes les commandes seront du type `ax.command(args)`, où `ax` est la sous-figure sur laquelle on veut dessiner, `command` est la commande de dessin, et `args` est la liste des arguments de cette commande. Si l'on veut tout effacer sur `ax`, on peut utiliser `ax.clear()`.

Lignes. On peut relier des points $(x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)$ en utilisant la commande `ax.plot(x, y)`, où x est la liste des abscisses $(x_1, x_2, \ldots, x_l)$ donnée sous forme de liste ou de *numpy array*, et y est la liste des ordonnées $(y_1, y_2, \ldots, y_l)$. Divers arguments optionnels permettent d'ajuster l'épaisseur du trait (`linewidth=a`), la forme (trait plein, pointillé, etc.), la couleur, etc.

Exemple 4. Les commandes

```
fig, ax = plt.subplots()
ax.plot([0, 3, 2, 0], [0, 0, 2, 1], color=sns.color_palette()[1], linestyle="--")
ax.text(1, 1.6, "ligne", rotation=28, color=sns.color_palette()[1])
plt.show()
```

dessinent une ligne pointillé orange :



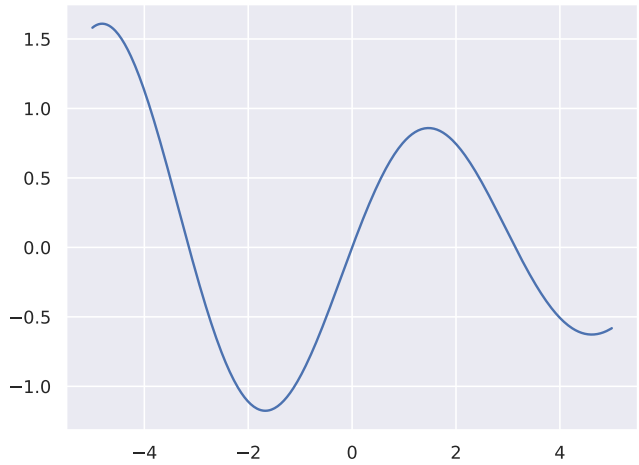
La commande `sns.color_palette()` appelle une palette de couleur ; on peut lui passer des arguments optionnels pour sélectionner la palette précise que l'on souhaite employer. Par défaut, on obtient une liste de 10 couleurs, qu'on peut sélectionner comme ci-dessus. Il existe aussi des palettes de couleurs continues indexées par un réel dans $[0, 1]$; par exemple, `sns.color_palette("flare", as_cmap=True)(0.3)` sélectionne la couleur d'indice 0.3 dans une palette continue allant du orange au violet. On renvoie à https://seaborn.pydata.org/tutorial/color_palettes.html pour plus de détail sur ce mode de sélection des couleurs. On peut aussi se passer des palettes et rentrer des couleurs prédéfinies (`"blue"`, `"red"`, etc.), ou donner un triplet RGB $(\rho, \gamma, \beta) \in [0, 1]^3$. On a aussi employé ci-dessus la commande `ax.text(a, b, "texte")` pour placer un texte sur le graphe. Cette commande admet de nombreux arguments optionnels (police de caractères, rotation, boîte entourant le texte, couleur, etc.) pour spécifier comment le texte doit apparaître.

Fonctions. Si l'on veut dessiner le graphe d'une fonction $y = f(x)$, on peut utiliser une ligne reliant de nombreux points (x_i, y_i) avec $y_i = f(x_i)$. Plus précisément, la fonction

```
def draw_function(f, a, b, samples=500):
        x = np.linspace(a, b, samples)
        y = f(x)
        fig, ax = plt.subplots()
        ax.plot(x, y, color=sns.color_palette()[0])
        plt.show()
```

prend en arguments une fonction f et deux bornes a et b, et dessine le graphe de la fonction f.

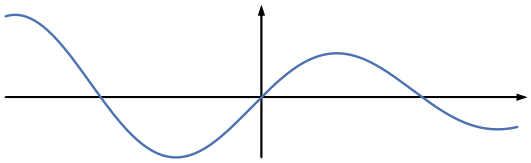
Exemple 5. La commande `draw_function(lambda x : np.sin(x) * np.exp(-x/10), -5, 5)` dessine ceci :



C'est tout à fait lisible, mais si l'on souhaite un graphe plus conventionnel avec un repère orthonormé, on peut utiliser les commandes

```
...
ax.set_axis_off()
ax.arrow(a, 0, b-a, 0, color="black", shape="full", head_width=0.1)
ax.arrow(0, min(y), 0, max(y)-min(y), color="black", shape="full", head_width=0.1)
ax.set_aspect(1)
...
```

juste après l'amorce du dessin dans le programme `draw_function` pour obtenir :



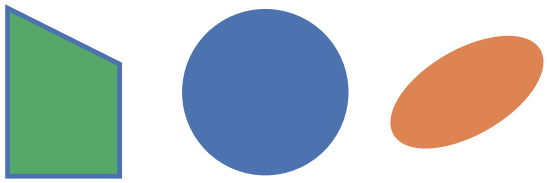
Au passage : on a utilisé la commande `ax.arrow(x, y, dx, dy)` pour dessiner une flèche allant de (x, y) à $(x + dx, y + dy)$.

Notons que sur le même principe, on peut dessiner n'importe quelle courbe paramétrée $\gamma(t) = (x(t), y(t))$ avec $t \in [0, T]$. Il suffit de créer un échantillonnage `E = np.linspace(0, T, samples)` et de relier les points avec `ax.plot(x(E),y(E))`. Il peut y avoir des petits problèmes de lissage (la courbe dessinée a des irrégularités non souhaitées), qu'on peut généralement résoudre en augmentant le nombre `samples`.

Figures géométriques. Les figures usuelles de la géométrie sont dans `matplotlib` des *Patches*, munis en particulier d'une couleur intérieur et d'une couleur de bord. On les ajoute à la sous-figure `ax` avec la commande `ax.add_patch()`.

Exemple 6. On peut dessiner en particulier des polygones, des cercles, des ellipses.

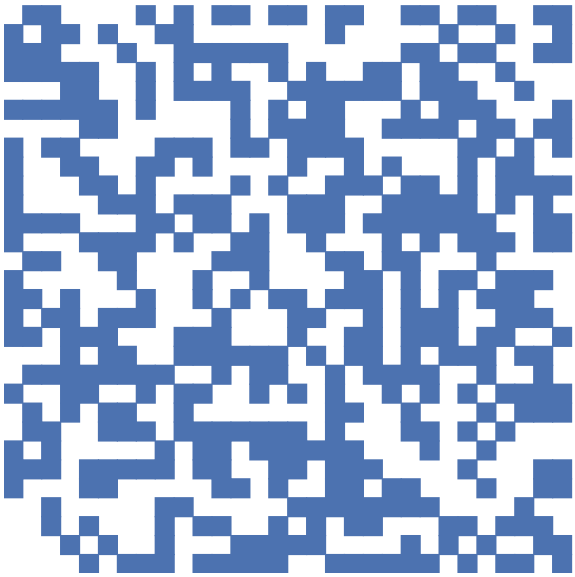
```
fig, axes = plt.subplots(1, 3)
for ax in axes:
        ax.set_axis_off()
        ax.set_aspect(1)
axes[0].add_patch(plt.Polygon(np.array([[0,0], [2,0], [2,2], [0,3]]), linewidth=3,
                        edgecolor=sns.color_palette()[0], facecolor=sns.color_palette()[2]))
axes[0].set_xlim(-0.5, 2.5)
axes[0].set_ylim(-0.5, 3.5)
axes[1].add_patch(plt.Circle((0, 0), 1))
axes[1].set_xlim(-1, 1)
axes[1].set_ylim(-1, 1)
from matplotlib.patches import Ellipse
axes[2].add_patch(Ellipse((0, 0), 2, 1, angle=30, color=sns.color_palette()[1]))
axes[2].set_ylim(-0.7, 0.7)
axes[2].set_xlim(-1, 1)
plt.show()
```



Exemple 7. Le programme suivant dessine la grille de taille $N \times N$ percolée avec paramètre p : chaque carré (i, j) est conservé avec probabilité p, et retiré avec probabilité $1 - p$, indépendamment pour chacun des N^2 petits carrés de la grille.

```
def percolation(N, p):
        fig, ax = plt.subplots()
        ax.set_axis_off()
        ax.set_aspect(1)
        for i in range(N):
            for j in range(N):
                if random.random() < p:
                    ax.add_patch(plt.Rectangle((i,j), 0.95, 0.95, color=sns.color_palette()[0]))
        ax.set_xlim(0, N)
        ax.set_ylim(0, N)
        plt.show()
```

Voici le résultat du programme pour $N = 30$ et $p = 0.5$:



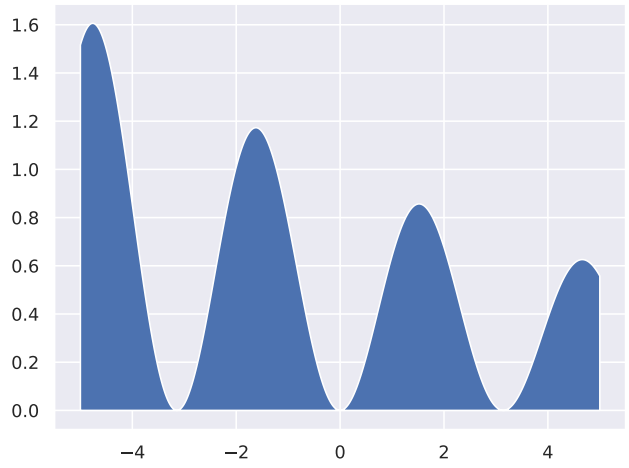
Remplissage. Une alternative aux *Patches* de type `Polygon` est la fonction `fill`. Par exemple, pour remplir le polygone (triangle) dont les sommets ont coordonnées $(0, 0)$, $(2, 0)$ et $(1, 1)$, on écrit :

```
fig, ax = plt.subplots()
ax.fill([0,2,1], [0,0,1])
ax.set_axis_off()
ax.set_aspect(1)
plt.show()
```

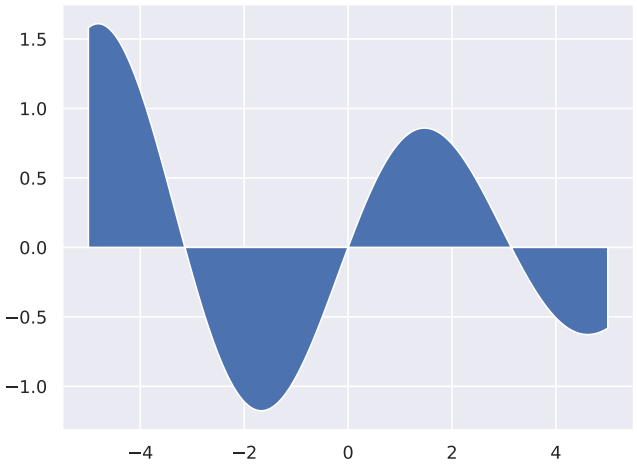
Un autre remplissage utile est celui de l'aire entre deux courbes, grâce à la fonction `fill_between`. Par exemple,

```
fig, ax = plt.subplots()
x = np.linspace(-5,5,500)
ax.fill_between(x, (lambda x : (np.sin(x)**2)*np.exp(-x/10))(x), 0)
plt.show()
```

remplit l'aire comprise entre l'axe des abscisses et la courbe d'équation $y = (\sin x)^2 \exp(-\frac{x}{10})$.



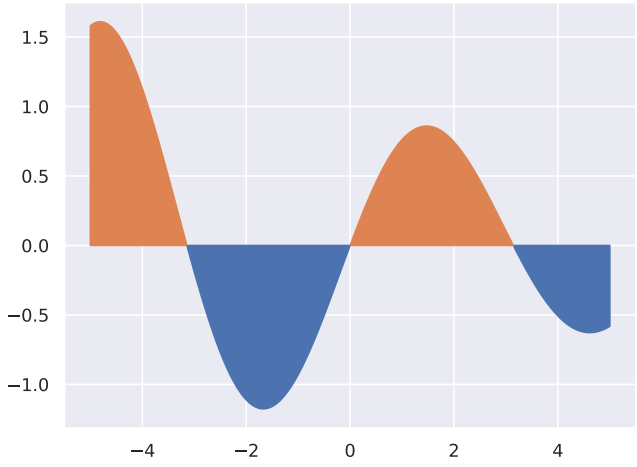
Si l'on fait la même chose avec la courbe $y = \sin x \exp(-\frac{x}{10})$, l'aire remplie est :



On peut remplir différemment les parties positives et négatives en utilisant l'argument optionnel `where`. Ainsi,

```
fig, ax = plt.subplots()
x = np.linspace(-5,5,500)
y1 = (lambda x : (np.sin(x))*np.exp(-x/10))(x)
y2 = (lambda x : 0)(x)
ax.fill_between(x, y1, y2, where=(y1>y2), color=sns.color_palette()[1])
ax.fill_between(x, y1, y2, where=(y1<y2), color=sns.color_palette()[0])
plt.show()
```

colorie en orange les parties positives, et en bleu les parties négatives :



5 Données

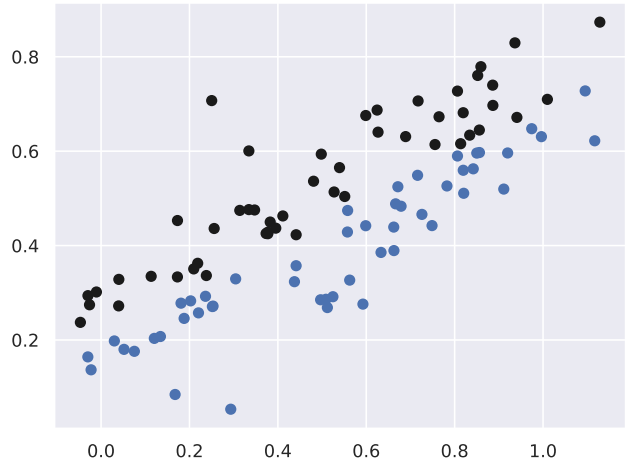
Pour conclure, voyons comment représenter des données (par exemple obtenu à partir d'une expérience aléatoire).

Nuage de points. Si l'on a un nuage de points du plan $((x_0, y_0), (x_1, y_1), \ldots, (x_l, y_l))$, on peut le dessiner avec `ax.scatter(x, y)`.

Exemple 8. Les commandes

```
fig, ax = plt.subplots()
x = np.linspace(0, 1, 100) + random.normal(0, 0.1, 100)
y = 0.5*x+0.2 + random.normal(0, 0.1, 100)
ax.scatter(x, y, color=np.where(y>0.5*x+0.2, "k", "b"))
plt.show()
```

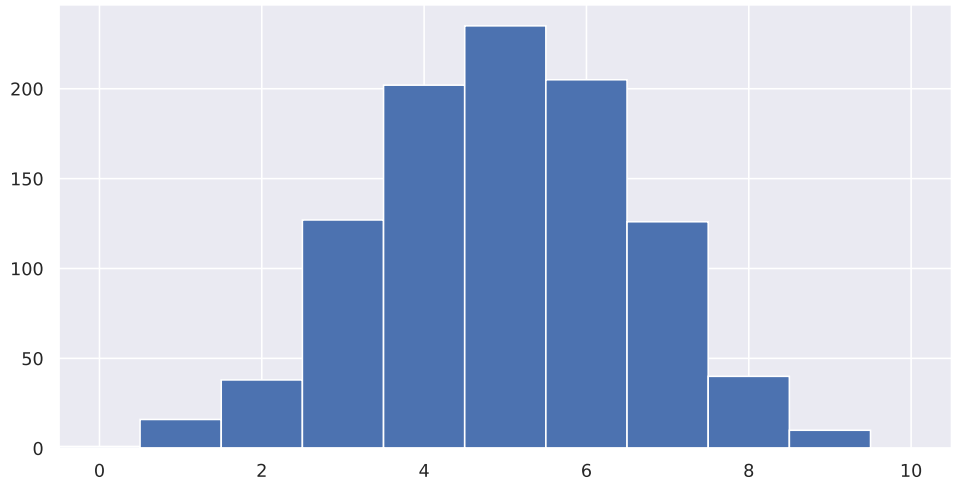
dessinent un nuage de points autour de la droite d'équation $y = \frac{x}{2} + \frac{1}{5}$, avec des erreurs d'observation gaussiennes. On a mis comme paramètre de couleur une liste, qui donne la couleur noire si $y_i > \frac{x_i}{2} + \frac{1}{5}$ et la couleur bleue sinon. On pourrait aussi modifier la forme des points (avec l'argument `marker`), et même mettre des formes différentes en fonction des positions des points.



Histogramme de données discrètes. Supposons donnée une liste $x = [x_1, x_2, \ldots, x_l]$ avec des x_i à valeurs dans un ensemble discret $\{0, 1, \ldots, K-1\}$. On peut représenter l'histogramme comptant chaque occurrence avec la commande `ax.hist(x, bins)`, où `bins` est la liste des valeurs possibles, c'est-à-dire dans notre cas `np.arange(K)`.

Exemple 9. Supposons donnée un échantillon de 100 variables binomiales de paramètres $(n = 10, p = 0.5)$. On représente l'histogramme de ces variables comme suit :

```
fig, ax = plt.subplots(figsize=(10, 5))
x = random.binomial(10, 0.5, 1000)
ax.hist(x, np.arange(11), align="left")
ax.set_xlim(-0.5, 10.5)
plt.show()
```



On a ici centré les barres de l'histogramme en chaque valeur.

Il y a une variante de cet algorithme si l'on a déjà calculé les fréquences y_i de chaque valeur possible x_i : on peut utiliser la commande `ax.bar(x, y)`, qui dessine au-dessus de chaque x_i une barre de hauteur y_i.

Fonction de répartition empirique. Pour des variables discrètes ou continues, une autre façon de représenter une distribution empirique repose sur la fonction de répartition empirique. Étant donnée une liste $X = (x_1, \dots, x_N)$, on a

$$F_X(t) = \frac{\text{card}(\{i \in [\![1, N]\!] \mid x_i \leq t\})}{N}.$$

La commande `ax.ecdf(X)` ajoute à la sous-figure `ax` la fonction de répartition empirique d'un échantillon `X`.

Exemple 10. Pour dessiner la fonction de répartition empirique d'un échantillon de variables normales standard indépendantes, on utilise :

```
fig, ax = plt.subplots(figsize=(10, 5))
X = random.normal(0, 1, 100)
ax.ecdf(X)
ax.set_xlim(-3, 3)
plt.show()
```