Python pour la physique

L'outil informatique est devenu incontournable dans le domaine des sciences (simulation numérique, traitement des données, robotique, intelligence artificielle, etc). Logiquement, l'apprentissage des bases de la programmation s'est imposé dans l'enseignement scientifique, et ce dès le secondaire. En CPGE, l'accent est mis sur le langage Python car il est très largement répandu et assez facile à prendre en main ; c'est ce que vous avez commencé à faire grâce au cours d'IPT. L'enseignement d'informatique pour la physique permet d'envisager l'utilisation de Python sous un autre angle : celui de l'expérimentateur. Les objectifs de cette année sont les suivants :

- simuler une expérience à l'aide d'un calcul numérique (évolution temporelle d'un circuit électrique, filtrage d'un signal, trajectoire d'un objet soumis à la pesanteur, à des frottements, à un champ électrique) ;
- traiter des données (récupération de données venant d'un fichier texte ou d'une carte arduino, tracé de graphe, régression, évaluation d'incertitude) ;
- tester la validité d'un modèle (comparaison entre des simulations numériques et des données expérimentales).

À cette occasion, vous allez découvrir des outils différents de ceux vus en IPT, dont on présente ci-dessous les grandes lignes.

1 Les bibliothèques *numpy* et *matplotlib.pyplot*

1.1 *Numpy*

Cette bibliothèque va principalement nous permettre de définir des **tableaux de nombres** (objets de type *np.ndarray*) et réaliser des opérations mathématiques sur ces tableaux. Ceci nous sera extrêmement utile pour effectuer des simulations numériques mais également pour tracer des graphes ou encore évaluer des incertitudes. Afin de pouvoir l'utiliser, il est nécessaire d'importer cette bibliothèque avec l'instruction suivante :

```
import numpy as np
```

ou encore :

```
from numpy import *
```

Avec la première instruction, il sera nécessaire d'utiliser le préfixe `np.` devant toutes les fonctions de cette bibliothèque. Avec la deuxième instruction, on pourra utiliser ces fonctions sans préfixe. Par la suite, on supposera toujours que numpy a été importée avec la première instruction. L'utilisation du préfixe vous aidera de reconnaître du premier coup d'œil les fonctions qui appartiennent à cette bibliothèque.

1.1.1 Tableau numpy

Un objet de type *np.ndarray* ne peut contenir que des nombres. Il peut être à une dimension (il s'apparente alors à un **vecteur**), à deux dimensions (il s'apparente alors à une **matrice**), ou davantage. Voici par exemple comment déclarer un tableau à une dimension :

```
A = np.array([0, 1, 2])
print(A)
print(A[0], ";", A[1], ";", A[2])
print(type(A))
```

```
[0 1 2]
0 ; 1 ; 2
<class 'numpy.ndarray'>
```

et à deux dimensions :

```
A = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print(A)
print(A[0][1], ";", A[1][0], ";", A[2][2])
```

```
[[0 1 2] [3 4 5] [6 7 8]]
1 ; 3 ; 8
```

Attention : on rappelle que le rang commence toujours à **zéro !**

Rq : On notera que la création et l'appel d'un élément d'un tableau numpy s'effectue de la même manière que pour les listes (ou bien les listes de liste si l'on est à deux dimensions). Quelle(s) différence(s) y a-t-il alors entre ces deux objets ? C'est ce que l'on va voir dans peu de temps!

1.1.2 Fonctions très classiques permettant de créer un tableau numpy

On a vu ci-dessus une première manière de créer un tableau numpy. Toutefois, si l'on veut définir un tableau de grande taille (1000 éléments par exemple), cette méthode n'est pas du tout adaptée! Fort heureusement, il existe plusieurs fonctions dans la bibliothèque numpy qui permettent de créer rapidement des tableaux de taille arbitraire.

- `np.zeros(N)` : crée un tableau de N éléments ne contenant que des 0.

```
A = np.zeros(5)
print(A)
```

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

Rq : L'écriture 0. indique que les zéros sont ici des nombres **flottants** (type *float*).

Rq : Cette fonction peut être utile lorsque l'on connaît à l'avance la taille du tableau voulu. Il s'agira ensuite de modifier la valeur des éléments, en utilisant une boucle itérative par exemple (voir en page suivante).

```
A = np.zeros(10)
for i in range(10):
    A[i] = i ** 2
print(A)
```

```
[ 0. 1. 4. 9. 16. 25. 36. 49. 64. 81.]
```

On peut également utiliser cette fonction pour remplir un tableau de dimension supérieure à 1. On remplace alors l'argument N de type *int* par un *tuple*. Voici un exemple à deux dimensions :

```
A = np.zeros((2,4))
print(A)
```

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

- `np.ones(N)` : crée un tableau de N éléments ne contenant que des 1.

```
A = np.ones(5)
print(A)
```

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

Rq : Comme pour la fonction `np.zeros`, on peut remplacer N par un *tuple* pour créer un tableau de dimension supérieure à 1.

- `np.linspace(a, b, N)`: crée un tableau de N éléments régulièrement répartis entre a (inclue) et b (**inclue**).

```
A = np.linspace(0, 10, 11)
B = np.linspace(100, 0, 11)
print(A)
print(B)
```

```
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
[100. 90. 80. 70. 60. 50. 40. 30. 20. 10. 0.]
```

Rq : les valeurs du tableau sont croissantes si $a < b$ et décroissantes si $a > b$.

- `np.arange(a, b, p)`: crée un tableau dont les valeurs sont régulièrement réparties entre a (inclue) et b (**exclue**) avec un pas fixé, égal à p.

```
A = np.arange(0, 100, 10)
B = np.arange(5, -5, -0.5)
print(A)
print(B)
```

```
[ 0 10 20 30 40 50 60 70 80 90]
[ 5. 4.5 4. 3.5 3. 2.5 2. 1.5 1. 0.5 0. -0.5 -1. -1.5 -2. -2.5 -3. -3.5 -4. -4.5]
```

Rq : si $a < b$ alors il faut que p soit positif. Si $a > b$ alors il faut que p soit négatif.

Rq : Ces deux dernières fonctions sont proches ; lorsque l'on veut obtenir des valeurs régulièrement réparties dans un intervalle, on a toujours le choix d'utiliser l'une ou l'autre. Suivant la situation (on connaît à l'avance le nombre total de valeurs ou bien le pas), on utilisera `np.linspace` ou bien `np.arange`.

1.1.3 Opérations sur les tableaux

Le grand avantage des tableaux numpy, comparés aux listes, c'est que l'on peut effectuer très facilement des opérations mathématiques sur leurs éléments en utilisant les fonctions de *numpy*. Voici quelques exemples :

```
A = np.linspace(0, 7, 8)
print(A)
print(A + 1)
print(A * 2)
print(A ** 2)
print(np.exp(A))
```

```
[0. 1. 2. 3. 4. 5. 6. 7.]
[1. 2. 3. 4. 5. 6. 7. 8.]
[ 0. 2. 4. 6. 8. 10. 12. 14.]
[ 0. 1. 4. 9. 16. 25. 36. 49.]
[1.00000000e+00 2.71828183e+00 7.38905610e+00 2.00855369e+01 5.45981500e+01
1.48413159e+02 4.03428793e+02 1.09663316e+03]
```

Rq : L'opération `L + 1` n'est pas définie pour une liste `L`. Pour rajouter la même valeur à tous les éléments d'une liste qui contiendrait uniquement des nombres, il faudrait nécessairement écrire une boucle itérative.

Comme vous venez de le voir, *numpy* possède une fonction `np.exp(x)` qui calcule une exponentielle. On donne ci-dessous une liste non-exhaustive de fonctions *numpy* que l'on utilisera régulièrement cette année.

`np.log(x)`	logarithme **néperien**
`np.log10(x)`	logarithme **décimal**
`np.sqrt(x)`	racine carrée
`np.sin(x)`, `np.arcsin(x)`, etc...	fonctions trigonométriques
`np.sinh(x)`, `np.arcsinh(x)`, etc...	fonctions hyperboliques
`np.abs(x)`	valeur absolue
`np.degrees(x)`	conversion des radians en degrés
`np.radians(x)`	conversion des degrés en radians
`np.pi`	approximation du nombre π

Pour chacune de ces fonctions (mis à part `np.pi`), l'argument peut être un nombre ou bien un tableau numpy. Dans ce dernier cas la fonction s'appliquera à l'ensemble des éléments du tableau numpy.

1.2 *Matplotlib.pyplot*

La bibliothèque *matplotlib.pyplot* contient des fonctions destinées au tracé de graphes 2D. Nous présentons ici, sans trop rentrer dans les détails, quelques unes des fonctions que vous utiliserez cette année. Tout d'abord il faut importer cette bibliothèque, par exemple avec l'une des deux instructions suivantes (mêmes remarques que pour *numpy*) :

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

ou bien :

```
from matplotlib.pyplot import *
```

Par la suite, on supposera que l'on a utilisé la première instruction et toutes les fonctions seront précédées de `plt.`.

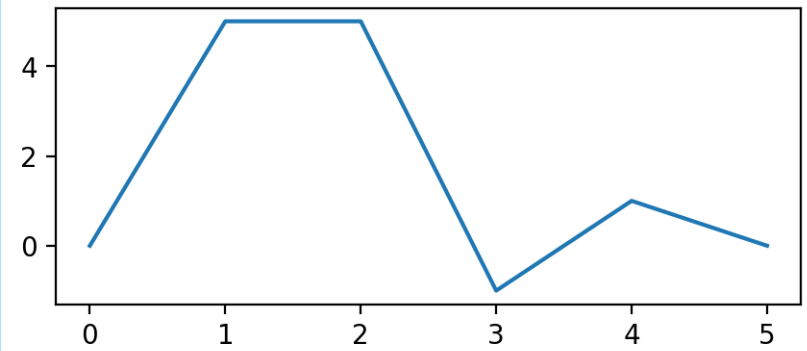
1.2.1 Calculer et afficher un nuage de points

Le tracé d'un graphe s'effectue à l'aide d'une succession d'instructions, la dernière consistant toujours à afficher le graphe à l'écran. Voyons d'abord comment demander à Python d'enregistrer un graphe en mémoire.

- `plt.plot(abs, ord)` : prend en arguments deux tableaux numpy **de même taille** (les listes sont également acceptées si elles ne contiennent que des valeurs numériques) , le premier contenant l'abscisse des différents points et le second leur ordonnée. Elle calcule le graphe et le garde en mémoire, **mais ne l'affiche pas**. On peut rajouter des arguments supplémentaires qui sont facultatifs ; nous en reparlerons très bientôt.

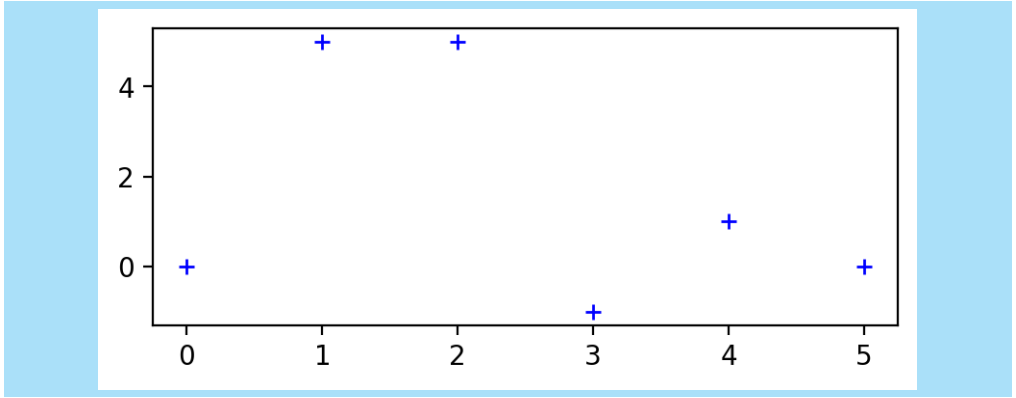
Une fois le graphe calculé, on l'affiche à l'écran avec l'instruction `plt.show()`, qui ne prend pas d'argument. Voyons un exemple ci-dessous :

```
import numpy as np
import matplotlib.pyplot as plt
A = np.array([0, 1, 2, 3, 4, 5])
B = np.array([0, 5, 5, -1, 1, 0])
plt.plot(A, B)
plt.show()
```



Comme vous pouvez le constater, la fonction `plt.plot` trace par défaut une ligne entre chaque point. Si l'on veut obtenir uniquement un nuage de points, on doit rajouter un argument supplémentaire. En voici un exemple :

```
plt.plot(A, B, 'b+')
plt.show()
```



La lettre `'b'` désigne la couleur que l'on donne aux points, ici le bleu. Voici d'autres possibilités :

Code	`'b'`	`'r'`	`'g'`	`'y'`	`'c'`	`'m'`	`'k'`	`'w'`
Couleur	bleu	rouge	vert	jaune	cyan	magenta	noir	blanc

Le symbole `'+'` désigne le style des points, ici un "plus". On donne d'autres exemples ci-dessous ; vous pouvez consulter en ligne la documentation de `plt.plot` pour une liste complète.

Code	`'+'`	`'x'`	`'o'`	`'s'`	`'*'`	`'d'`	`'p'`
Couleur	plus	croix	cercle	carré	étoile	losange	pentagone

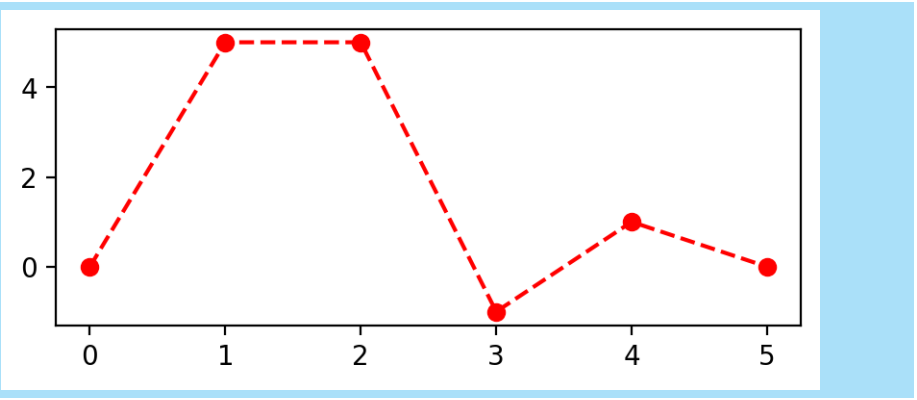
Généralement, on rassemble ces deux instructions en une seule : `'kx'`, `'r*'`, etc. L'ordre n'a pas d'importance : `'r*'` et `'*r'` donnent le même résultat.

Revenons au premier graphe ; imaginons que l'on veuille tracer une ligne **tiretée** rouge entre chaque points, chacun d'entre eux étant indiqué par un cercle. Il faut encore rajouter un argument pour définir le style de la ligne ; voici quelques exemples :

Code	`'-'`	`'--'`	`':'`	`'-.'`
Couleur	ligne continue	tirets	pointillés	point-tirets

Le code à écrire est donc le suivant :

```
plt.plot(A, B, 'r--o')
plt.show()
```

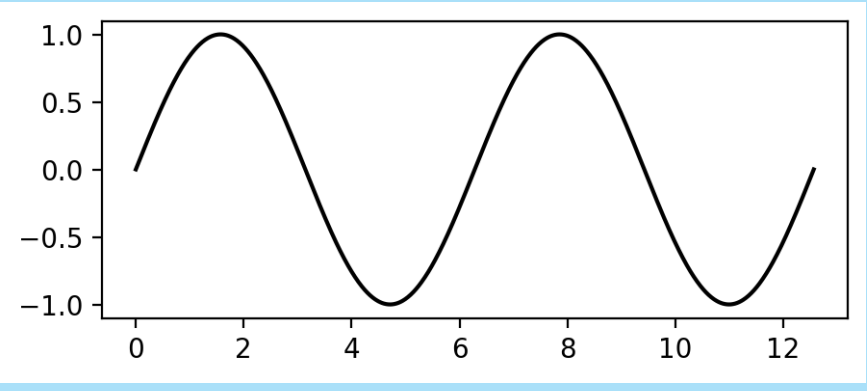


Vous pouvez désormais personnaliser votre graphe en choisissant votre propre style !

1.2.2 Tracer le graphe d'une fonction

Nous venons de voir un exemple de tracé de graphe avec deux tableaux remplis "à la main" (cela peut correspondre à une situation où les grandeurs en abscisse et en ordonnées sont des données expérimentales par exemple). Comment faire si l'on veut tracer le graphe d'une fonction mathématique ? Nous allons illustrer la méthode sur l'exemple de la fonction $x \mapsto \sin(x)$.

```
import numpy as np
import matplotlib.pyplot as plt
#on definit les valeurs en abscisse
x = np.linspace(0, 4 * np.pi, 1000)
#on calcule les valeurs en ordonnee
y = np.sin(x)
plt.plot(x, y, 'k-')
plt.show()
```



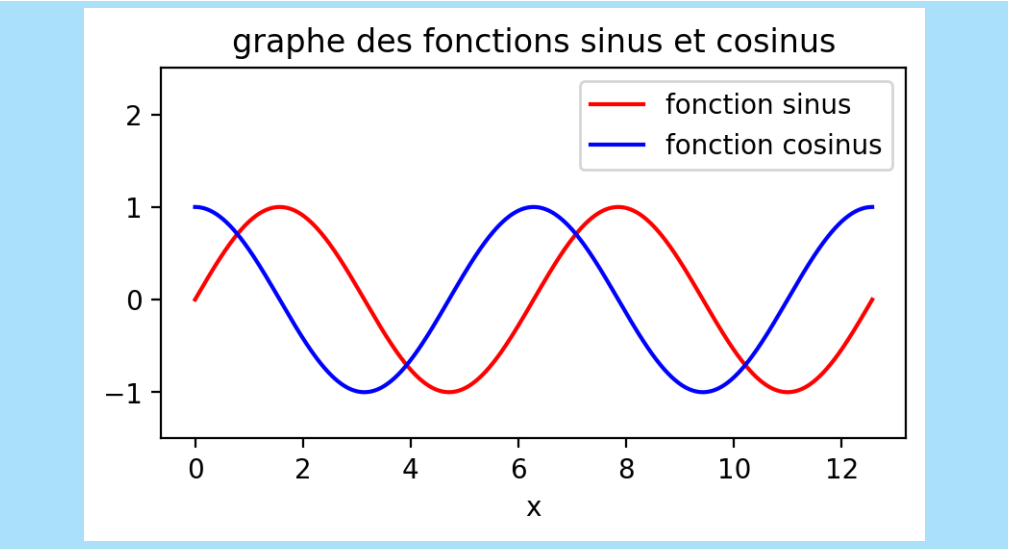
Nous n'avons pas réellement tracé une fonction "continue". Le graphe a été calculé avec $N = 1000$ points répartis régulièrement entre 0 et 4π et reliés les uns aux autres par des lignes continues. Plus N est élevé et plus le graphe est précis, mais plus le temps de calcul est important. On note que le calcul des ordonnées est rendu très simple par l'utilisation de tableaux numpy.

1.2.3 Quelques fonctions supplémentaires pour agrémenter votre graphe

- `plt.xlabel()` : ajoute une étiquette pour l'axe des abscisses.
- `plt.ylabel()` : ajoute une étiquette pour l'axe des ordonnées.
- `plt.legend()` : ajoute une légende.
- `plt.title()` : ajoute un titre au graphe. L'argument doit être une chaîne de caractère (par exemple `plt.title("Voici un titre")`)
- `plt.xlim()` : impose l'intervalle d'affichage sur l'axe des abscisses.
- `plt.ylim()` : impose l'intervalle d'affichage sur l'axe des ordonnées.
- `plt.figure(figsize = (a, b))` : ouvre une nouvelle fenêtre graphique de dimensions $a \times b$ (les valeurs sont en pouces (*inches*)).

On illustre tout cela sur un exemple ; au passage vous noterez que l'on peut superposer plusieurs graphes avec des `plt.plot()` successifs.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 4 * np.pi, 1000)
y1 = np.sin(x)
y2 = np.cos(x)
plt.figure(figsize = (5,2.5))
#label = '' permet d'attribuer une etiquette a une courbe
plt.plot(x, y1, 'r-', label = 'fonction sinus')
plt.plot(x, y2, 'b-', label = 'fonction cosinus')
plt.xlabel('x')
plt.title("graphe des fonctions sinus et cosinus")
plt.ylim(-1.5, 2.5)
#on affiche la legende avec les etiquettes definies plus haut.
#loc = 'best' permet de placer automatiquement la legende a l'endroit
#ou elle se superpose le moins au graphe
plt.legend(loc = 'best')
plt.show()
```



2 Jupyter Notebook

Cette année, vous avez l'habitude d'utiliser EduPython pour écrire et exécuter vos scripts Python. En physique nous utiliserons un environnement différent appelé *Jupyter Notebook*.

2.1 Projet Jupyter et Jupyter Notebook

Jupyter est une application web utilisée pour programmer dans plus de 40 langages de programmation, dont Python, Julia, Ruby, R, ou encore Scala2. C'est un projet communautaire dont l'objectif est de développer des logiciels libres, des formats ouverts et des services pour l'informatique interactive. Jupyter est une évolution du projet IPython. Jupyter permet de réaliser des calepins ou notebooks, c'est-à-dire des programmes contenant à la fois du texte en markdown et du code. Ces calepins sont utilisés en science des données pour explorer et analyser des données (source : *Wikipedia*).

Un notebook offre la possibilité d'inclure dans un même document :

- du texte,
- des images,
- des formules mathématiques en langage LaTeX (très utilisé dans la littérature scientifique),
- du code informatique (en Python pour ce qui nous concerne),
- la sortie obtenue après exécution des scripts.

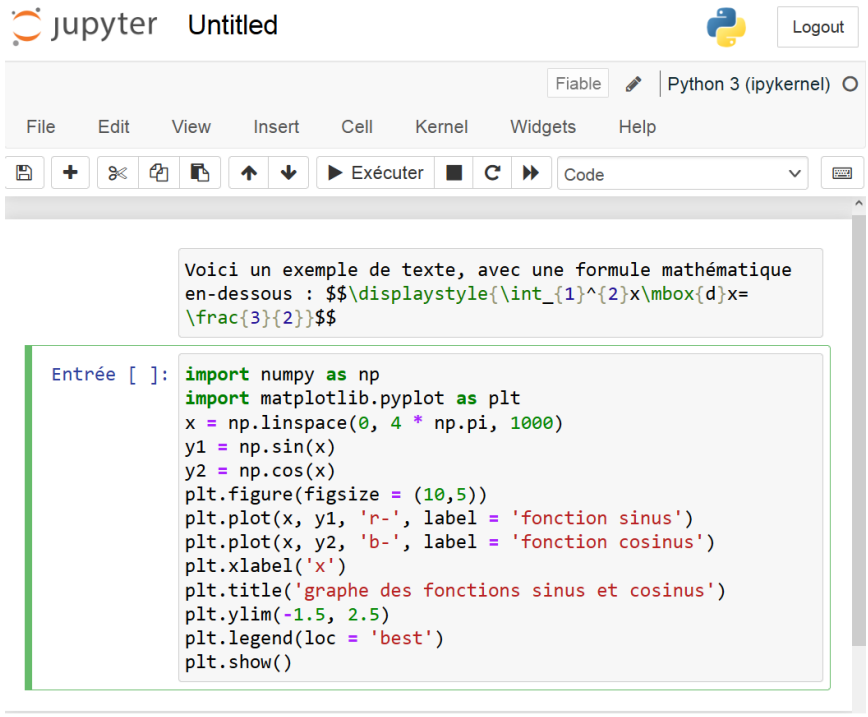
Cet environnement présente donc une certaine agilité et est très adapté à l'enseignement de Python. En effet, un seul document contiendra le sujet du TD, les scripts que vous aurez rédigés et le résultat de ces scripts.

2.2 Structure d'un fichier Jupyter Notebook

Un notebook est un fichier dont l'extension est ".ipynb" ; il y a différentes manières d'en ouvrir un, en voici quelques exemples :

- en allant directement sur le site internet du projet Jupyter : https://jupyter.org/,
- en installant la distribution *Anaconda* sur un ordinateur (*Anaconda* est une distribution libre et open source des langages de programmation Python et R appliqué au développement d'applications dédiées à la science des données et à l'apprentissage automatique, elle contient notamment l'application *Jupyter Notebook*),
- avec d'autres support permettant d'écrire et d'exécuter du code Python dans un navigateur (comme *Google Colab* par exemple).

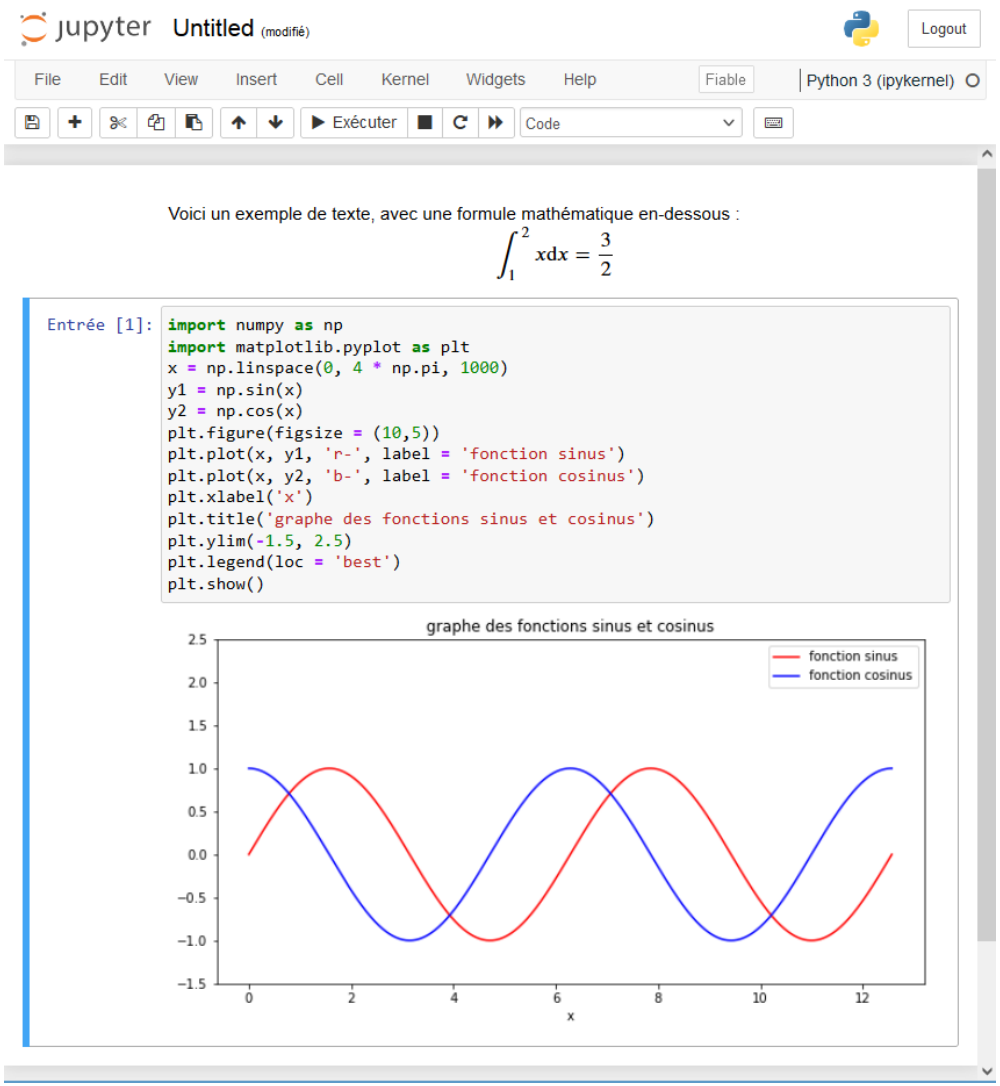
Au lycée, nous ouvrirons les notebooks par l'intermédiaire d'*Anaconda*. Voici en image à quoi peut ressembler un notebook :



On distingue deux cellules ; la première est une cellule dite "Markdown", c'est-à-dire une cellule de texte écrit avec le langage Markdown. On y distingue une formule mathématique (écrite entre les $$). Pour l'instant ce n'est pas très lisible car il s'agit d'instructions en LaTeX mais cette cellule n'est pas censée rester en l'état. Elle va être éxécutée (avec un Ctrl + Entrée ou bien en cliquant sur "Exécuter" dans le menu) pour donner un résultat plus acceptable.

La cellule suivante est dite "Code" (on le voit plus haut dans le menu), autrement dit elle contient un code Python qui peut être exécuté (là encore avec Ctrl + Entrée par exemple). Le résultat est alors affiché en-dessous de la cellule correspondante.

Voici le résultat obtenu lorsque l'on compile les deux cellules :



C'est déjà plus joli, non ? On peut ainsi ajouter autant de cellules "Markdown" et de cellules "Code" que l'on souhaite dans le fichier. Généralement, vous ouvrirez un notebook que j'aurai rédigé à l'avance et qui contiendra des consignes ainsi que des cellules de code à compléter (j'en aurai souvent déjà écrit une partie pour vous guider, ne me remerciez pas c'est mon côté généreux ☺).