

Tutoriel Python

PROJ0001-1: Introducion aux méthodes numériques et projet

Bachelier en Sciences de l'Ingénieur (Bloc 1)

Année académique 2020-2021

Si vous lisez ce tutoriel sous sa version html ou pdf, il vous faudra ouvrir Spyder pour modifier et tester les cellules de code de ce notebook. Si vous êtes sur myBinder alors, vous pourrez modifier de manière interactive le notebook mais les changements effectués ne sont que temporaires.

Pour naviguer à travers le Notebook Jupyter:

1. Navigation: Lorsque vous ouvrez un Notebook, vous constaterez que vous pouvez déplacer un bloc surligné (avec une ligne bleue à gauche) avec les flèches du curseur pour se déplacer vers le haut et vers le bas. Ce bloc met en évidence une cellule. (Vous pouvez également utiliser la souris pour sélectionner une cellule). C'est ce qu'on appelle le mode Commande.
2. Exécuter du code: Si vous voulez exécuter une cellule (par exemple une cellule qui contient du code Python), vous pouvez appuyer sur Shift+ENTER. Si la cellule crée une sortie, elle sera affichée sous la cellule.
3. Edition de code: Si vous voulez changer le code dans la cellule actuellement surlignée, vous devez appuyer sur ENTER (ou double cliquer dessus). Vous avez maintenant saisi le mode d'édition, et le contenu de la cellule peut être édité. Si vous avez terminé vos modifications, et que vous souhaitez les exécuter, utiliser le raccourci Shift+ENTER.

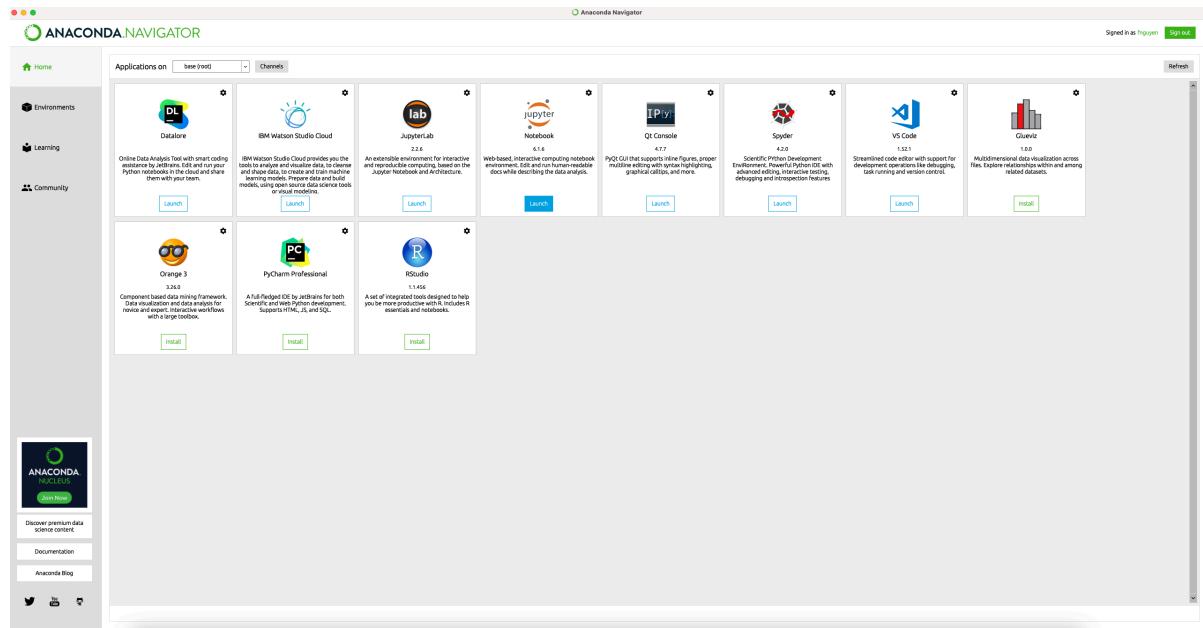
Installation d'Anaconda, de Python et de l'IDE Spyder

Python est un langage de programmation open source orienté objet de haut niveau largement utilisé dans le monde industriel et scientifique. Ce langage permet non seulement d'effectuer du calcul scientifique et de la visualisation graphique, mais également de créer des sites web. Afin de pouvoir en bénéficier, nous utiliserons la plateforme de distribution Anaconda permettant d'installer Python et l'environnement de programmation Spyder. Pour débuter, téléchargez et installez la distribution d'Anaconda de votre système d'exploitation (Windows, Mac OS ou Linux):

<https://www.anaconda.com/products/individual>
[\(https://www.anaconda.com/products/individual\)](https://www.anaconda.com/products/individual)

En installant Anaconda, vous installerez Python, Jupyter Notebook et Spyder. Une fois Anaconda installé, vous pourrez lancer l'environnement de programmation Spyder et faire tourner les Notebooks (comme ce tutoriel par exemple).

Une fois que vous avez installé Anaconda, démarrez l'application. A partir de l'écran d'accueil, vous pourrez démarrer soit Spyder soit Jupyter Notebook en double cliquant sur l'icône correspondante.



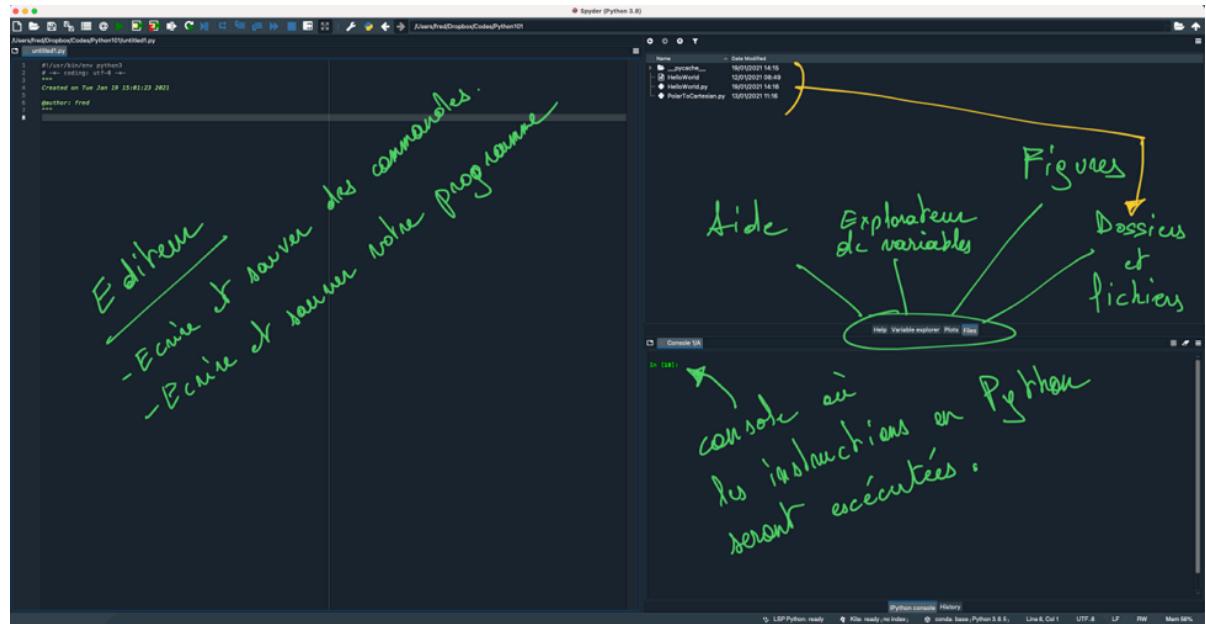
IDE Spyder

Spyder est un environnement de développement scientifique pour Python : il dispose de fonctions avancées d'édition, de tests interactifs, de débogage et d'analyse de performance (profiler).

<https://www.spyder-ide.org/> (<https://www.spyder-ide.org/>).

- Dans Spyder, la console IPython est l'interpréteur Python par défaut
- Le code dans l'éditeur peut être exécuté entièrement ou partiellement dans cette console
- L'éditeur prend en charge la vérification automatique des erreurs Python
- Le débogueur IPython peut être activé
- Un profiler est fourni pour analyser l'efficacité du code
- Un explorateur d'objets montre la documentation des fonctions, des méthodes, ...
- L'explorateur de variables affiche les noms, la taille et les valeurs des variables numériques
- L'explorateur de fichier permet de naviguer dans votre arborescence de fichiers.

Une fois Spyder lancé, vous verrez une interface organisée en trois grandes fenêtres: l'éditeur utilisé pour écrire et sauvegarder vos commandes ou programmes, l'explorateur de fichiers, de figures ainsi que l'aide, et la console qui exécute les commandes écrites en Python ainsi que l'historique des commandes entrées. Cette dernière option est très utile pour remonter dans le temps et permet de voir quelles commandes ont été exécutées, dans quel ordre et à quel moment.



La console IPython permet d'entrer des commandes après l'invite de commande "In []":

In [1]: `a=2`

L'exemple ci-dessus montre que l'on a affecté la valeur 2 à la variable `a` à l'aide de l'opérateur `=`. Une variable est un conteneur d'information sous la forme d'un scalaire, d'un tableau ou d'une suite de caractères. Nous pouvons affecter plusieurs variables à la fois. La définition est par défaut silencieuse, c'est-à-dire que la valeur de `a` ne s'affichera pas à l'exécution de la commande `a=2`. Si nous voulons afficher la valeur de `a`, nous pouvons utiliser la fonction `print()` ou utiliser l'explorateur de variables. Nous pouvons voir dans l'exemple ci-dessous que nous avons créé une variable appelée `a`, qui est un entier de dimension 1 et de valeur "2".

L'explorateur des variables constitue donc notre espace de travail. Il fournit à l'utilisateur des informations sur les variables définies à l'instant présent. Il offre des fonctionnalités pour la gestion des variables (création, édition, suppression, etc.). Ces informations sont également accessibles par le biais des commandes `who` et `whos` ou la commande `type()`.

In [2]: `a1 = a2 = a3 = 3
print(a1 * a2 * a3)`

In [3]: `print(a)`

2

In [4]: `who`

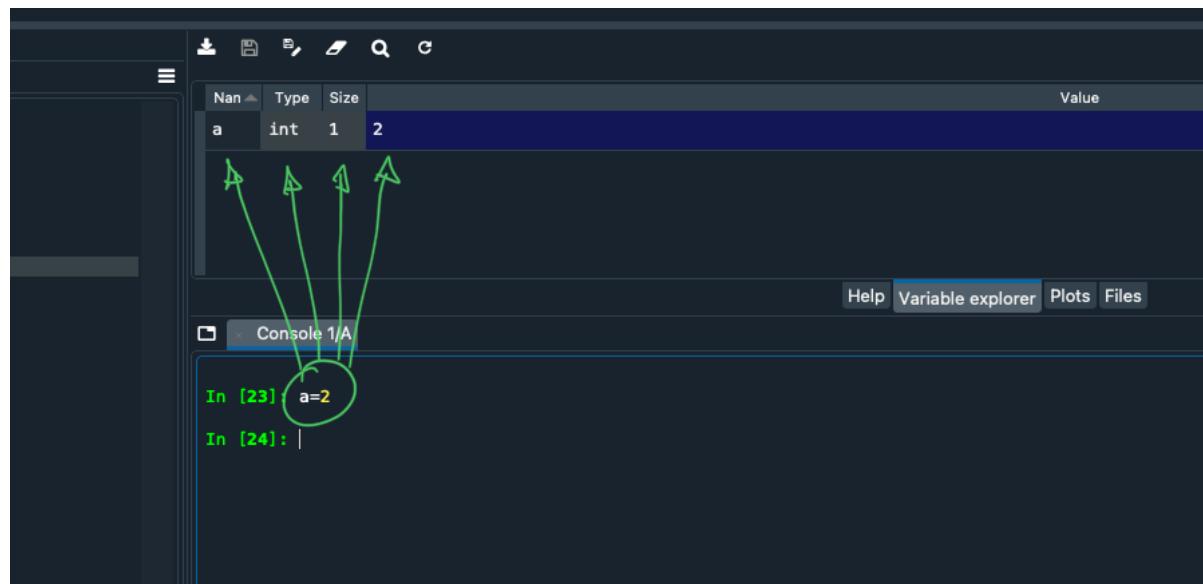
a a1 a2 a3 os sys

In [5]: `whos`

Variable	Type	Data/Info
a	int	2
a1	int	3
a2	int	3
a3	int	3
os	module	<module 'os' from 'C:\\\\So<...>\\\\Anaconda3\\\\lib\\\\os.py'>
sys	module	<module 'sys' (built-in)>

In [6]: `type(a)`

Out[6]: int



Si nous voulons effacer des variables, nous pouvons utiliser la commande `%reset` (qui demande une confirmation) pour effacer toutes les variables ou la commande `del` pour effacer une variable spécifique. La commande `clear` ne fait, elle, que nettoyer la console tout en gardant les variables en mémoire.

In [7]: `del a`

In [8]: `%reset`

On peut également utiliser les opérateurs d'addition `+`, de soustraction `-` de multiplication `*`, de division `/` et de puissance `**` ou les combiner. Par exemple `+=` incrémentera la variable affectée d'une valeur à spécifier.

In [9]: `a = 2 + 3
print(a)`

5

In [10]: `print(2**3)`

8

In [11]: `a += 1
print(a)`

6

In [12]: `a -= 1
print(a)`

5

Python définit plusieurs types d'opérations de base par défaut (affectation, arithmétique, puissance, valeur absolue, comparaisons, logique):

In [13]: `1 != 2`

Out[13]: True

In [14]: `True & False`

Out[14]: False

In [15]: `True & True`

Out[15]: True

Types de variables

Dans Python, il n'est pas nécessaire de déclarer les variables avant de pouvoir les utiliser. La valeur que l'on affecte à la variable définit son type. Les principaux types sont :

Type int (nombre entier)

```
In [16]: a = 300  
type(a)
```

```
Out[16]: int
```

Type float (nombre à virgule flottante)

```
In [17]: a = 1.25e3  
type(a)
```

```
Out[17]: float
```

Type complex (complexe)

```
In [18]: a = 1 + 3j  
type(a)
```

```
Out[18]: complex
```

Type str (chaîne de caractères)

```
In [19]: a = "bonjour"  
print(a)  
type(a)
```

```
bonjour
```

```
Out[19]: str
```

```
In [20]: a = 'aurevoir'  
print(a)  
type(a)
```

```
aurevoir
```

```
Out[20]: str
```

Type bool (booléen)

```
In [21]: a = True  
type(a)
```

```
Out[21]: bool
```

In [22]:

```
b = not(a)
print(b)
```

False

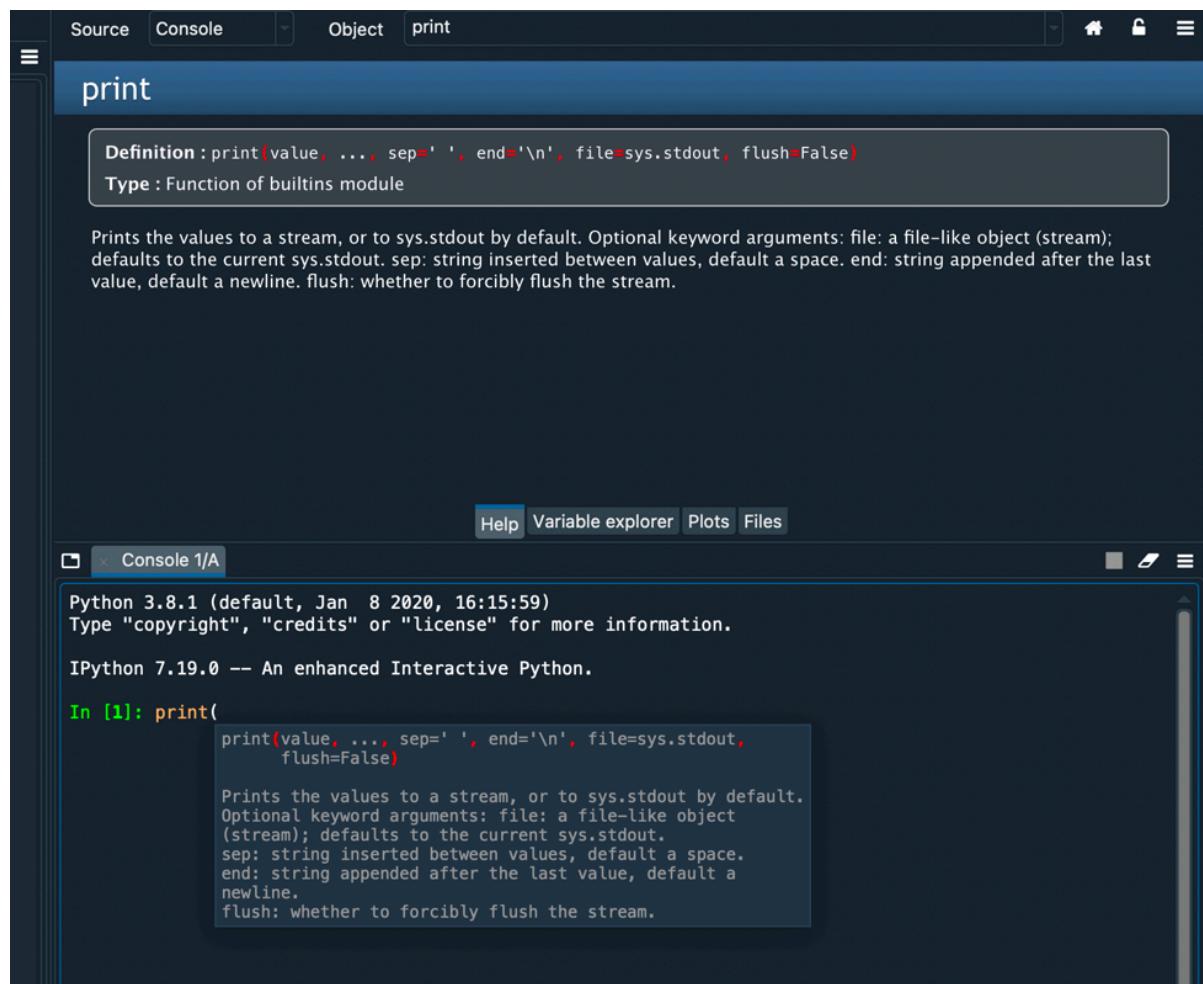
Nous avons utilisé l'opérateur `not()` pour renvoyer l'opposé de la valeur booléenne transmise.

Exercice: Créez deux variables de type `str` et additionnez les. Quel est le résultat?

In [23]:

Aide

Il existe énormément de ressources pour Python sur internet (<https://www.python.org/search/> (<https://www.python.org/search/>), <https://docs.python.org/3/contents.html> (<https://docs.python.org/3/contents.html>), <https://docs.python.org/3/tutorial/> (<https://docs.python.org/3/tutorial/>)) mais l'utilisateur peut d'abord trouver de l'aide localement en appuyant Ctrl+I (Windows) ou Cmd+I (Mac OS) devant n'importe quel objet. Vous pouvez également utiliser l'aide interactive fournie dans l'environnement Spyder dans l'aide ou à travers la console en entrant par exemple `help(print)`. Finalement, une aide se montrera automatiquement après avoir écrit une parenthèse de gauche à côté d'un objet.



In [31]: `help(print)`

```
Help on built-in function print in module builtins:
```

```
print(...)  
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline

flush: whether to forcibly flush the stream.

Plus simplement, si nous souhaitons obtenir de l'aide au niveau de l'utilisation de la console, nous pouvons utiliser ?

In []: `?`

Exercice: Recherchez de l'aide sur la fonction `abs()` et utilisez cette fonction sur un nombre de votre choix défini dans une variable `a`

In []:

Exercice: Affectez à une variable votre prénom et votre nom séparé par un . et calculez la longueur de la chaîne de caractères ainsi créée

In []:

Terminologie

Python utilise un certain nombre de concepts propres qu'il est important de définir. Vous trouverez de plus amples informations au sein du glossaire

(<https://docs.python.org/fr/3/glossary.html#glossary>

(<https://docs.python.org/fr/3/glossary.html#glossary>)).

Scripts

Un script est simplement un fichier `.py` où est enregistré un ensemble d'instructions destinées à être exécutées. L'exécution du script créera de nouvelles variables ou des graphiques dans l'espace de travail.

Fonctions/functions

Une fonction un est ensemble de lignes d'instructions organisé selon une syntaxe bien définie (utilisant le mot-clé `def`) et réutilisable, qui est employé pour effectuer une action unique correspondante à la fonction. La syntaxe correspondante est donnée dans l'exemple suivant qui effectue simplement la somme de deux variables `a` et `b` et qui affecte le résultat dans une variable `c` renommée à l'aide du mot clé `return` comme sortie de la fonction:

```
In [32]: %reset
def somme_a_b(a,b):
    #Cette fonction effectue la somme de a et de b
    c = a + b
    return c
```

Once deleted, variables cannot be recovered. Proceed (y/[n])? y

```
In [35]: somme_a_b(1,3)
```

```
Out[35]: 4
```

Une définition de fonction associe un nom de fonction à un objet fonction dans l'espace de noms actuel que l'on peut voir ci-dessous à travers la commande `whos`.

```
In [36]: whos
```

Variable	Type	Data/Info
somme_a_b	function	<function somme_a_b at 0x7fde98401550>

Exercice: Soit la fonction suivante

$$f(x) = ax^2 + bx$$

Créez une fonction qui reçoit en argument `x` et qui retourne $f(x)$ si $a = 3$ et $b = 2.4$

```
In [ ]:
```

Modules

Un module est un fichier Python (.py) destiné à être importé dans des scripts, dans la console ou d'autres modules. Il définit des classes, des fonctions et des variables destinées à être utilisées une fois importées. On peut ainsi réutiliser des fonctions écrites pour un programme dans un autre sans avoir à les copier. La commande `import` permet d'importer le module en entier ou des parties de celui-ci. On peut ensuite accéder à celle-ci en utilisant `nom_du_module.nom_de_l_objet`. Dans l'exemple ci-dessous, le module `dummy_module.py` contient les instructions suivantes:

```
In [25]: #Contenu du fichier dummy_module.py
course_code = 'PROJ0001-1'

students_number = 300

def number_group(n):
    print(n/3)
```

```
In [37]: %reset
import dummy_module
```

Once deleted, variables cannot be recovered. Proceed (y/[n])? y

```
In [38]: whos
```

Variable	Type	Data/Info
dummy_module	module	<module 'dummy_module' from 'C:\Users\frdnguyen-tuto_python_proj0001-obs3xwsk\Notebooks\Tutoriel\Python\20notebook.ipynb'>

```
In [39]: print(dummy_module.course_code)
n = dummy_module.students_number
dummy_module.number_group(n)
```

PROJ0001-1
100.0

Notons que si nous exécutons le contenu du fichier `dummy_module.py` directement dans la console, nous créerions une fonction `number_group()`, ainsi que deux variables: la chaîne de caractères `course_code` et le nombre entier `students_number`. L'importation du fichier ne créera que le module `dummy_module` dans l'espace de travail.

Paquets/packages

C'est en fait un certain type de modules Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`. Les paquets sont des moyens de structurer les différents modules en utilisant une notation « pointée ». Par exemple, le nom de module A.B désigne le sous-module B du paquet A. Vous pouvez vous représenter les paquets comme des répertoires dans le système de fichiers et les modules comme des fichiers dans ces répertoires.

Librairies

Une librairie peut contenir des dizaines, voire des centaines de modules individuels qui peuvent fournir un large éventail de fonctionnalités. `Matplotlib` est une librairie permettant de créer des graphiques pour visualiser des données (<https://matplotlib.org> (<https://matplotlib.org>)). La librairie standard de Python (<https://docs.python.org/3/library/> (<https://docs.python.org/3/library/>)) contient des centaines de modules permettant d'effectuer des tâches courantes, comme par exemple la fonction `help()` qui permet d'invoquer le système d'aide.

Classes

Une classe peut être vue comme étant un modèle contenant des attributs (ou variables) et des méthodes (ou fonctions) permettant de créer des instances de cette classe. Les classes permettent de définir de nouveaux types de variables propres à la personne qui programme. L'instanciation d'une classe permet de créer un objet de cette classe. Les méthodes associées à une classe ont un accès privilégié aux données de la classe et les attributs se comportent comme des variables globales pour les méthodes de la classe. Une classe se définit par le mot-clé `class` suivi du nom de la classe, de `:` puis un retour à la ligne. Nous pouvons par exemple décider de créer une classe `student` où des attributs tels que l'âge ou le prénom seront définis et une méthode permettant de calculer la probabilité de réussir son année. Ensuite, nous utiliserons cette classe pour créer différentes instances d'étudiant.e.s et examiner leur chances de réussite.

```
In [40]: class student:
    def __init__(self, age, prenom, nom, ects_obtenu, bloc1, cycle):
        self.age = age
        self.prenom = prenom
        self.email = prenom + '.' + nom + '@student.uliege.be'
        self.nom = nom
        self.ects_obtenu = ects_obtenu
        self.bloc1 = bloc1
        self.cycle = cycle

    def proba_reussite(self):
        pr = (self.age / self.ects_obtenu)
        return pr

s932810 = student(20, 'John', 'Smith', 30, True, 'bachelier')
s493304 = student(22, 'Jane', 'Doe', 30, False, 'Master')
```

```
In [41]: print(s932810.email)
print(s493304.email)
```

```
John.Smith@student.uliege.be
Jane.Doe@student.uliege.be
```

```
In [42]: print(s932810.proba_reussite())
0.6666666666666666
```

```
In [43]: print(s493304.proba_reussite())
0.7333333333333333
```

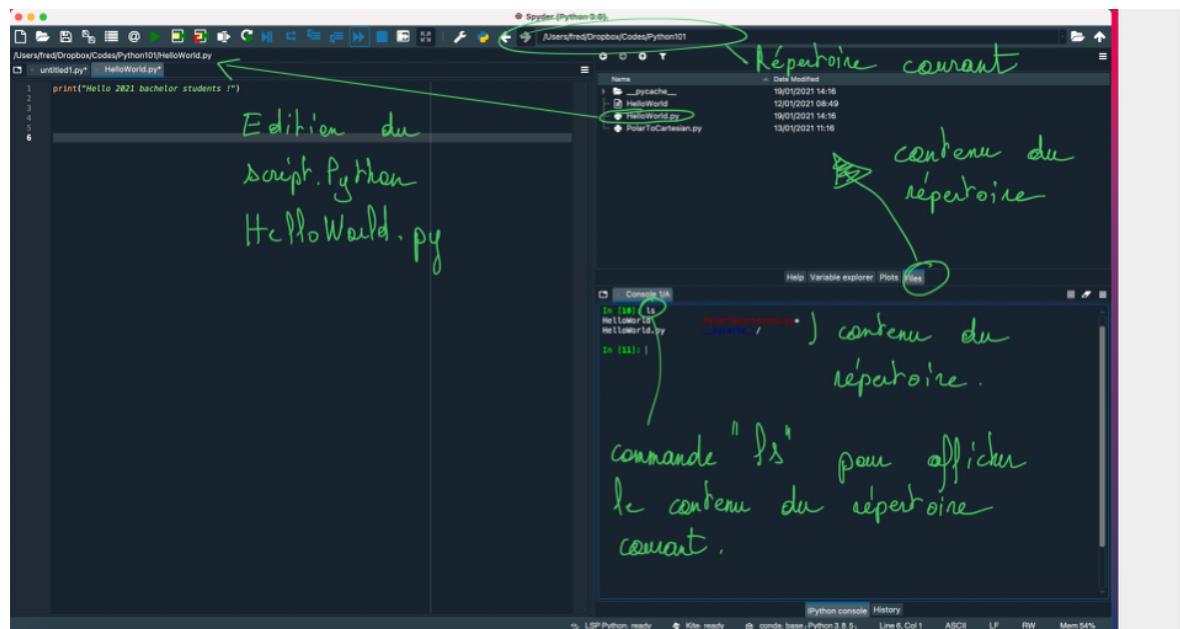
Le mot clé `self` désigne l'instance de la classe sur laquelle va s'appliquer la méthode et `__init__` est le constructeur de la classe qui permet d'initialiser les attributs de la classe.

Exercice: Créez votre propre classe de Professeur avec ses attributs et ses méthodes et trouvez lui une utilité.

```
In [ ]:
```

Edition de scripts

Le répertoire courant est le répertoire où l'utilisateur travaille. Le contenu de ce répertoire est affiché dans Spyder et des fonctionnalités pour la gestion de son contenu sont fournies. L'affichage du contenu du répertoire courant peut également être obtenu dans la fenêtre de commande à l'aide de la commande "ls".



In [44]: ls

HelloWorld.py	img16.png
OdeFun.py*	img2.png
PolarToCartesian.py*	img3.png
Tutoriel Python notebook.ipynb	img4.png
__pycache__/	img5.png
chgtcoord.py*	img6.png
data.txt	img7.png
debug.py*	img8.png
dummy_module.py*	img9.png
img1.png	normalised_data.txt
img10.png	ping.py*
img11.png	pingpong.py*
img12.png	student_class.py*
img15.png	test_profile.py*

Comme défini ci-dessus, un script est simplement un fichier où est enregistré un ensemble de commandes qui sera exécuté dans la console exactement comme si on y entrait les commandes directement. Les fichiers de script s'enregistrent dans le répertoire courant avec l'extension ".py" et peuvent être lancés à partir de l'éditeur dans Spyder en cliquant sur run ou en exécutant la commande %run suivie du nom du fichier du script. Tout ce qui suit le caractère # est considéré comme un commentaire et ne sera donc pas exécuté. On peut faire intervenir dans un script des instructions simples, des importations de modules et des appels à des fonctions pour autant que celles-ci soient définies dans l'espace de travail.

In [45]: `%run HelloWorld.py`

Hello 2021 bachelor students !
Welcome to PROJ0001

```

print("Hello 2021 bachelor students!")
print("Welcome to PROJ0001")

```

Exécute les commandes enregistrées dans le script HelloWorld.py

Un script est donc simplement un fichier où est enregistré un ensemble de commandes qui sera exécuté dans la console exactement comme si on y entrat les commandes directement comme illustré ci-dessous:

In [46]: `print("Hello 2021 bachelor students !") #Ceci est un commentaire
print("Welcome to PROJ0001")`

Hello 2021 bachelor students !
Welcome to PROJ0001

Quand on exécute un script, tout le script s'exécute, et aucune fonction n'est appelée automatiquement, à l'inverse d'autres langages comme le C où la fonction `main()` est la première fonction exécutée. Ceci peut s'avérer problématique quand on importe un script plutôt que de le lancer directement. Afin d'éviter cela, il suffit d'ajouter dans le corps du script `if __name__=='__main__':` afin de spécifier que le code qui suit la condition ne sera exécuté que si le script est appelé directement et pas importé. Nous l'illustrons en dessous avec les scripts ping.py et pingpong.py.

In [49]: `#Script de pingpong.py
def ping():
 print("pong")
ping()`

pong

In [50]: #Script de ping.py

```
def ping():
    print("pong")

if __name__ == '__main__':
    ping()

pong
```

In [1]: import pingpong

```
pong
```

In [2]: import ping

In [3]: whos

Variable	Type	Data/Info
ping	module	<module 'ping' from '/Use<...>hon101/Tutoriel/ping.py'>
pingpong	module	<module 'pingpong' from '<...>01/Tutoriel/pingpong.py'>

Exercice: Créez un script Python dans lequel est défini une fonction greet(nom) qui imprime Hello 'nom', et utilisez cette fonction pour faire apparaître Hello 'notre nom' dans la console.

In []:

Indentation

La plupart des langages informatiques ont une forme de structure début-fin telle que des accolades d'ouverture et de fermeture, ou quelque chose du genre pour délimiter clairement le morceau de code qui se trouve dans une boucle, ou dans différentes parties d'une structure. En général, les programmeurs aguéris indentent également leur code afin qu'il soit plus facile pour un lecteur de voir ce qui se trouve à l'intérieur d'une boucle, en particulier s'il y a plusieurs boucles imbriquées. Mais dans la plupart des langages, l'indentation n'est qu'une question de style et la structure début-fin de la langue détermine comment elle est réellement interprétée par l'ordinateur.

En Python, l'indentation est tout ce qu'il y a de plus important. Il n'y a pas de début et de fin, seulement de l'indentation. Tout ce qui est censé se trouver à un niveau d'une boucle doit être indenté à ce niveau. Une fois que la boucle est terminée, l'indentation doit revenir au niveau précédent.

Le nombre d'espaces à indenter à chaque niveau est une question de style, mais vous devez être cohérent dans un seul code. La norme est souvent de 4 espaces. Ceci permet de rendre le code lisible. N'hésitez pas également à ajouter des commentaire afin d'augmenter la lisibilité de votre code avec l'utilisation du caractère `#` ou des caractères `""" """` permettant d'insérer des commentaires sur plusieurs lignes.

In [4]:

```
"""
Created on Wed Jan 20 11:19:08 2021

Ce programme calcule le cosinus de i pour i allant de 0 à 3 avec un
et affiche la dernière valeur obtenue. Il ne nécessite aucun paramètre
et ne fournit aucune sortie

@author : proj0001
"""

# Import de librairies nécessaires à ce script
import numpy

# Corps principal du script
for i in [0, 1, 2, 3]:
    x = numpy.cos(i)

# Affichage de la valeur de x
print('x vaut', x)
```

x vaut -0.9899924966004454

Librairies, fonctions, variables locales et globales

Au démarrage de Python, un certain nombre de fonctions de base sont disponibles, par exemple la fonction `print()` qui permet d'afficher des variables ou des chaînes de caractères, ainsi que la syntaxe générale du langage. Cependant, la plupart des fonctions nécessaires à des fins spécifiques comme simplement calculer le cosinus d'un angle se trouvent dans des librairies qui ne se chargent pas par défaut afin de ne pas consommer trop de temps pour démarrer Python. Par exemple, si on essaye de lancer la fonction `cos()` on fait face à un message d'erreur:

In [5]: `cos(0)`

```
NameError                                 Traceback (most recent c  
all last)  
<ipython-input-5-8d215eb8f10> in <module>  
----> 1 cos(0)  
  
NameError: name 'cos' is not defined
```

La fonction cosinus est disponible dans la librairie NumPy . Les librairies sont importées dans l'espace de travail avec la commande d'importation `import` . Chaque librairie contient de nombreuses fonctions. Pour utiliser la fonction `cos()` de la librairie NumPy , il nous faut le spécifier. Pour ce faire, la syntaxe est la suivante: `librairie.fonction()` .

In [2]: `import numpy`
`numpy.cos(0)`

Out[2]: `1.0`

In [4]: `type(numpy)`

Out[4]: `module`

In [5]: `type(numpy.cos)`

Out[5]: `numpy.ufunc`

Nous pouvons également choisir d'importer uniquement la fonction `cos()` du module NumPy et de la renommer, ici `cosinus` .

```
In [6]: from numpy import cos as cosinus  
cosinus(0)
```

```
Out[6]: 1.0
```

Pour définir une fonction, il suffit d'utiliser la syntaxe `def`. Les commandes ci-dessous entrées dans la console définissent une fonction "PolarToCartesian" qui prend en paramètres d'entrée le rayon "rho" et l'angle "theta" défini en degrés. Ces paramètres suivent le nom de la fonction et sont définis au sein des parenthèses. Cette fonction produit en sortie les coordonnées cartésiennes "x" et "y" correspondantes grâce à la commande `return` :

```
In [9]: # Importation de librairies ou de fonctions externes  
import numpy  
  
# Définition de la fonction  
def PolarToCartesian(rho,theta):  
    """  
        Parameters  
        -----  
        rho : rayon  
  
        theta : angle en degrés  
  
    Returns  
    -----  
    Coordonnées cartésiennes d'un point défini à partir de ses co  
    .....  
  
    theta = theta * numpy.pi/180  
    x = rho * numpy.cos(theta)  
    y = rho * numpy.sin(theta)  
    return x,y
```

```
In [8]: %reset
```

Once deleted, variables cannot be recovered. Proceed (y/[n])? y

```
In [10]: whos
```

Variable	Type	Data/Info
PolarToCartesian	function	<function PolarToCartesian at 0x7f9 930c85ca0>
numpy	module	<module 'numpy' from '/Us<...>kages /numpy/__init__.py'>

Pour appeler une fonction dans la console une fois qu'elle y est définie, il faut taper son nom et lui faire suivre des parenthèses en y entrant les arguments de la fonction (si celle-ci en nécessite):

In [15]: `PolarToCartesian(1,0)`

Out [15]: `(1.0, 0.0)`

On peut voir ci-dessus que la fonction `PolarToCartesian()` nécessite l'utilisation de la librairie NumPy pour calculer le cosinus et le sinus de l'angle donné. Il est important de noter qu'une fonction est un ensemble de commandes exécutées dans un sous-environnement qui n'existe que pendant l'exécution de la fonction. Si nous essayons d'appeler la variable "x" en dehors de la fonction `PolarToCartesian()`, la console nous retournera un message d'erreur. Un appel à `who` permet de vérifier que la fonction existe sans que ce soit le cas des variables définies au sein de cette fonction. Les variables définies au sein d'une fonction ont donc une **portée locale**.

In [16]: `x`

```
NameError: name 'x' is not defined
```

NameError
all last)
<ipython-input-16-6fcf9dfbd479> in <module>
----> 1 x

Traceback (most recent c

In [11]: `who`

PolarToCartesian numpy

Une variable peut avoir une **portée globale** si elle est déclarée en dehors d'une fonction.

In [18]: `global_x = 'variable globale'`

```
def dummy_function():
    print('Je suis une ' + global_x)

dummy_function()
```

Je suis une variable globale

Si l'on souhaite pouvoir modifier une variable globale au sein d'une fonction, il faut la déclarer comme `global`

```
In [19]: global_x = 'variable globale'

def dummy_function():
    global global_x
    global_x = 'Je suis une ' + global_x

dummy_function()
print(global_x)
```

Je suis une variable globale

Finalement, Python offre la possibilité de déclarer une variable comme étant **non-locale** avec le mot clé `nonlocal`. Ceci est utilisé lorsque des fonctions imbriquées sont implémentées dans un même script. Ainsi, la fonction imbriquée a accès aux variables des fonctions englobantes. Supposons que l'on veuille calculer l'aire d'un disque et la circonférence d'un trou centré sur ce disque et dont le rayon est deux fois plus petit que le rayon du disque.

```
In [20]: import numpy

def cercle(rayon):
    pi = numpy.pi
    aire_disque = pi * (rayon) ** 2
    def circonference():
        nonlocal rayon
        rayon = rayon/2
        circonference = 2 * pi * rayon
        return circonference
    print(circonference())
    return aire_disque
print(cercle(3))
```

9.42477796076938
28.274333882308138

La portée d'une variable en Python est donc la partie du code où elle est visible. Elle peut être locale, non-locale, globale et "built-in".

Sauf pour des fonctions très simples, vous ne voulez pas écrire les fonctions directement dans la console. Normalement, vous devrez créer un fichier `.py` contenant votre fonction et importer le module résultant dans votre session interactive ou dans votre script. Dans l'exemple suivant, nous avons copié l'ensemble des commandes de la fonction `PolarToCartesian` ci-dessus dans un fichier "chgtcoord.py". Nous importons le module "chgtcoord" que nous renommons "coord" et nous utilisons la fonction "PolarToCartesian" définie au sein du module.

```
In [1]: import chgtcoord as coord
```

In [2]: whos

Variable	Type	Data/Info
coord	module	<module 'chgtcoord' from <...>1/Tutoriel/chgtcoord.py'>

In [3]: coord.PolarToCartesian(1, 0)

Out[3]: (1.0, 0.0)

Liste de librairies utiles

Nous listons ci-dessous une série de librairies et de fonctions utiles au déroulement du cours. N'hésitez pas à lire l'aide correspondante aux différentes fonctions proposées et à les tester par vous-même.

matplotlib

Exemples de fonctions utiles

pyplot

plot() legend() xlabel() ylabel() show() xlim() grid()

numpy

Exemples de fonctions utiles

abs() sum() zeros() asarray() arange() min() max() where() linspace() shape() loadtxt() floor()
ceil() log() cos() sqrt()

scipy

Exemples de fonctions utiles

integrate.solve_ivp() interpolate.splrep() interpolate.splev()

Exercice: Soit l'équation quadratique

$$ax^2 + bx + c = 0$$

Créez une fonction qui reçoit en argument les coefficients a, b, c et qui fournit en sortie les deux racines.

In []:

D'autres types d'objets: list, tuple et array

Nous utiliserons la librairie `numpy` pour effectuer des opérations mathématiques sur des tableaux (appelés arrays). A côté de ces tableaux, Python possède d'autres types de données pouvant enregistrer des séquences d'éléments : les listes et les tuples. Ces derniers ne sont cependant pas utilisés pour effectuer des opérations mathématiques.

Liste et tuple

Une liste se définit par des `[]` tandis qu'un tuple se définit par des `()`. On peut modifier les valeurs entrées dans une liste mais pas dans un tuple.

```
In [4]: une_liste=[2, 4, 6, 8, 10]
print(une_liste)
type(une_liste)
```

```
[2, 4, 6, 8, 10]
```

```
Out[4]: list
```

```
In [5]: un_tuple=(2, 4, 6, 8, 10)
print(un_tuple)
type(un_tuple)
```

```
(2, 4, 6, 8, 10)
```

```
Out[5]: tuple
```

Si nous essayons d'effectuer une opération sur ces deux objets, nous voyons que ce qui pourrait être compris comme une opération mathématique, crée en réalité une nouvelle liste qui contient `une_liste` deux fois tandis qu'il n'est pas possible de modifier `un_tuple`

```
In [6]: une_liste * 2
```

```
Out[6]: [2, 4, 6, 8, 10, 2, 4, 6, 8, 10]
```

```
In [7]: un_tuple + 2
```

```
-----  
TypeError  
all last)  
<ipython-input-7-da236c663ea6> in <module>  
----> 1 un_tuple + 2  
  
TypeError: can only concatenate tuple (not "int") to tuple
```

Traceback (most recent c

Les listes et les tuples peuvent contenir des séquences de plusieurs types, et pas uniquement des nombres. Dans l'exemple ci-dessous, nous créons une liste contenant une chaîne de caractères, des nombres, ainsi qu'une fonction.

```
In [8]: une_liste_variee = ['liste', 1, 20.45, print]  
print(une_liste_variee)
```

```
['liste', 1, 20.45, <built-in function print>]
```

Tableau à 1 dimension

Pour effectuer des opérations mathématiques sur des objets contenant des séquences de valeurs, nous utiliserons des tableaux. Pour ceci, nous aurons besoin des librairies NumPy et matplotlib

```
In [9]: %matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt
```

Ensuite, nous pouvons utiliser différentes fonctions pour créer un tableau à 1 dimension allant par exemple de 1 à 5:

```
In [10]: x = np.array([1, 2, 3, 4, 5]) # Nous avons insérer manuellement les  
# du tableau en utilisant une liste  
print('x:', x, type(x))  
  
y = np.arange(1, 6, 1) #Démarre de 1, s'arrête avant 6 avec un pas de  
print('y:', y, type(y))  
  
z = np.linspace(1, 5, 5) #Démarre de 1 jusque 5 avec 5 points espacés  
print('z:', z, type(z))  
  
x: [1 2 3 4 5] <class 'numpy.ndarray'>  
y: [1 2 3 4 5] <class 'numpy.ndarray'>  
z: [1. 2. 3. 4. 5.] <class 'numpy.ndarray'>
```

En utilisant les fonctions `len` (built-in), `shape` (utile pour les tableaux à plus de 1 dimensions) et `ndim` (méthodes de `ndarray`), nous pouvons calculer les formes, les dimensions des tableaux et leurs longueurs:

In [21]: `print(len(x))`

5

In [17]: `x.ndim`

Out[17]: 1

In [16]: `x.shape`

Out[16]: (5,)

Pour accéder aux éléments des tableaux, nous utilisons leurs indices linéaires en commençant par le 0. Le premier élément d'un tableau à 1 dimension a donc l'indice `0`. Le dernier élément d'un tableau à 1 dimension peut être accédé à l'aide de l'indice `-1`. Les indices sont spécifiés en utilisant des crochets `[indice]`. Les valeurs des éléments peuvent ainsi être modifiées.

In [22]: `print('La valeur du premier élément de x est', x[0])`

`x[0]=10`

`print('Le premier élément de x vaut maintenant', x[0])`

La valeur du premier élément de x est 1

Le premier élément de x vaut maintenant 10

In [23]: `x[-1]`

Out[23]: 5

On peut également accéder à plusieurs éléments en utilisant `:` et en spécifiant le début, la fin et le pas.

In [24]: `x[0:4:2] #x[début:avant_fin:pas]`

Out[24]: `array([10, 3])`

In [25]: `x[:]`

Out[25]: `array([10, 2, 3, 4, 5])`

Il est également possible de parcourir les tableaux à l'envers

```
In [26]: x[len(x):-1]
```

```
Out[26]: array([ 5,  4,  3,  2, 10])
```

On peut également rechercher certains éléments spécifiques d'un array avec la fonction where comme illustré ci-dessous:

```
In [27]: a = np.array([-1, 0, 1.3, 4, -10, 0.4, 3, 9, 1])
ind_a_neg = np.where(a<0)
print('les indices correspondants à des valeurs négatives sont:', i)
print('les valeurs négatives de a sont:', a[ind_a_neg])
```

```
les indices correspondants à des valeurs négatives sont: (array([0
, 4]),)
les valeurs négatives de a sont: [-1. -10.]
```

Les slices en Python permettent de découper des objets contenant des séquences de valeurs d'un objet a (nparray, list, tuple, chaîne de caractères) en ne sélectionnant qu'une partie de ceux-ci. La syntaxe des slices est a[i:j] ou s[i:j:k], les indices i (début), j (fin) et k (incrément) peuvent être omis:

```
In [28]: a = 'abcdefghijklmnopqrstuvwxyz'
type(a)
```

```
Out[28]: str
```

```
In [29]: a[0:2]
```

```
Out[29]: 'ab'
```

```
In [30]: a[0:]
```

```
Out[30]: 'abcdefghijklmnopqrstuvwxyz'
```

```
In [31]: a[0:26:2]
```

```
Out[31]: 'acegikmoqsuwy'
```

```
In [32]: a[:12]
```

```
Out[32]: 'abcdefghijklmnopqrstuvwxyz'
```

```
In [33]: a[:]
```

```
Out[33]: 'abcdefghijklmnopqrstuvwxyz'
```

In [34]: `a[-1]`

Out[34]: 'z'

In [35]: `a_tuple=(1, 2, 3, 'quatre')`

In [36]: `a_tuple[2:4]`

Out[36]: (3, 'quatre')

In [37]: `a_liste=[1, 3, 4, 4]
a_liste[0:3]`

Out[37]: [1, 3, 4]

In [38]: `a_array=np.array(a_liste)
a_array[0:3]`

Out[38]: array([1, 3, 4])

Opérations sur un tableau à 1 dimension

Lorsque l'on effectue des opérations sur des arrays NumPy , la vitesse d'exécution peut être grandement améliorée en évitant l'utilisation de boucles pour parcourir les tableaux et en utilisant les différentes fonctions de NumPy qui peuvent s'appliquer directement aux tableaux.

In [39]: `x = np.arange(1,6,1)
y = np.zeros(x.shape) #initialisation de y de même taille que x
for i in range(len(x)):
 y[i] = np.sin(x[i])
print(y)`

[0.84147098 0.90929743 0.14112001 -0.7568025 -0.95892427]

Ceci peut être simplement écrit:

In [40]: `y = np.sin(x)
print(y)`

[0.84147098 0.90929743 0.14112001 -0.7568025 -0.95892427]

Les opérations dites vectorielles peuvent s'appliquer aux tableaux à 1 dimension de type array en utilisant les fonctions appropriées de NumPy tandis les opérations de base s'appliquent élément par élément. Par exemple la multiplication de x par y à l'aide de l'opérateur * donne :

```
In [41]: x = np.arange(1,6,1)
y = np.arange(0,5,1)
print('x:', x)
print('y:', y)
x * y
```

```
x: [1 2 3 4 5]
y: [0 1 2 3 4]
```

```
Out[41]: array([ 0,  2,  6, 12, 20])
```

Pour effectuer le produit scalaire des vecteurs x et y, on peut utiliser la fonction `dot` de NumPy

```
In [42]: np.dot(x,y)
```

```
Out[42]: 40
```

Exercice: Créez un tableau à une dimension (vecteur ligne) `x` de 5 nombres successifs entre 2 et 3 et séparés par des intervalles égaux.

```
In [ ]:
```

Exercice: Ajoutez 1 au deuxième élément de `x`

```
In [ ]:
```

Exercice: Créez un deuxième tableau ligne `y` de la même dimension que `x` mais dont les éléments sont les nombres pairs successifs en commençant à 4.

```
In [ ]:
```

Tableaux à 2 dimensions

Les tableaux peuvent être de dimensions arbitraires, mais nous utiliserons dans ce cours fréquemment les tableaux à 2 dimensions pour lesquels des opérations matricielles peuvent être définies. Pour créer ces tableaux, nous pouvons réutiliser la fonction `array` de NumPy

```
In [44]: A = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]])
print(A)
print('A has', np.ndim(A), 'dimensions')
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
A has 2 dimensions
```

```
In [46]: print('La taille de A est: ', A.shape)
```

```
La taille de A est: (3, 5)
```

L'indice d'un tableau à 2 dimensions est spécifié par deux valeurs, la première correspond aux lignes, la seconde aux colonnes du tableau et sont données à l'aide d'un tuple (donc défini entre parenthèse). Dans l'exemple ci-dessous, nous créons un tableau à 2 dimensions rempli de 0 et nous le modifions en changeant 1 élément, puis une partie de ligne et enfin le dernier élément.

```
In [47]: B = np.zeros((3,5))
print('Initialisation de B:')
print(B)
print('Modification de B:')
B[0,0] = 5
B[1,2:] = 10
B[-1,-1] = 500
print(B)
```

```
Initialisation de B:
[[0.  0.  0.  0.  0.]
 [0.  0.  0.  0.  0.]
 [0.  0.  0.  0.  0.]]
Modification de B:
[[ 5.   0.   0.   0.   0.]
 [ 0.   0.  10.  10.  10.]
 [ 0.   0.   0.   0.  500.]]
```

Comme précédemment, les opérations de base s'effectuent élément par élément:

In [48]: `print(A * B)`

```
[[5.0e+00 0.0e+00 0.0e+00 0.0e+00 0.0e+00]
 [0.0e+00 0.0e+00 8.0e+01 9.0e+01 1.0e+02]
 [0.0e+00 0.0e+00 0.0e+00 0.0e+00 7.5e+03]]
```

L'objet `array` de NumPy ,ici A, possède certaines méthodes définies automatiquement comme la transposée de A qui se note `A.T`. Les opérations matricielles peuvent être effectuées en utilisant les fonction de NumPy :

In [49]: `print(np.dot(A.T,B))`

```
[[5.00e+00 0.00e+00 6.00e+01 6.00e+01 5.56e+03]
 [1.00e+01 0.00e+00 7.00e+01 7.00e+01 6.07e+03]
 [1.50e+01 0.00e+00 8.00e+01 8.00e+01 6.58e+03]
 [2.00e+01 0.00e+00 9.00e+01 9.00e+01 7.09e+03]
 [2.50e+01 0.00e+00 1.00e+02 1.00e+02 7.60e+03]]
```

Exercice: Créer un tableau de type `ndarray` à 2 dimensions A dont la première ligne est égale à `x` , la deuxième ligne est remplie de 1 et la troisième ligne est égale à `y` . `x` et `y` sont définis dans l'exercice précédent.

In []:

Exercice: Soit deux matrices

$$A = \begin{bmatrix} 4 & -5 \\ \sqrt{3} & \pi/4 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 3+i \\ -72/3 & 0.2 \end{bmatrix}$$

où i est le nombre imaginaire. Calculer les éléments suivants :

$$A + B, \quad AB, \quad A^2, \quad A^T, \quad B^{-1}, \quad B^T A^T, \quad A^2 + B^2 - AB$$

Résolution de systèmes d'équations linéaires

Pour résoudre un système linéaire $Ax = b$, par exemple

$$\begin{pmatrix} 5 & 6 & 10 \\ -3 & 0 & 14 \\ 0 & -7 & 21 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 10 \\ 0 \end{pmatrix}$$

nous pouvons utiliser la méthode `solve`, qui fait partie de la sous-librairie `linalg` de NumPy . La méthode de résolution prend en entrée un tableau bidimensionnel (la matrice A) et un tableau unidimensionnel (le côté droit) et renvoie la solution.

In [50]:

```
A = np.array([[5, 6, 10], [-3, 0, 14], [0, -7, 21]])
b = np.array([4, 10, 0])
solution = np.linalg.solve(A, b)
print(solution)
```

```
[ -1.45454545  1.20779221  0.4025974 ]
```

On peut simplement vérifier que la solution fournie est la bonne en multipliant la matrice A par la solution

In [51]:

```
np.dot(A,solution)
```

Out[51]:

```
array([ 4., 10.,  0.])
```

Exercice: Résoudre le système d'équations linéaires suivant

$$\begin{aligned} 5.4x + 2y &= 0 \\ -x + 4y - 25z &= 3 \\ 3x + 7z &= 2 \end{aligned}$$

In []:

Structures de contrôle: `for`, `if/elif/else`, `while`, `break`

Python intègre de base des structures de contrôle. Elles permettent d'effectuer des boucles, de vérifier si certaines conditions sont remplies avant d'exécuter certaines lignes de codes. Nous aurons ici besoin de `numpy` et de `matplotlib`, nous commençons donc par les importer.

La boucle `for`

La syntaxe est la suivante et permet d'exécuter un bloc de commandes de manière répétée :

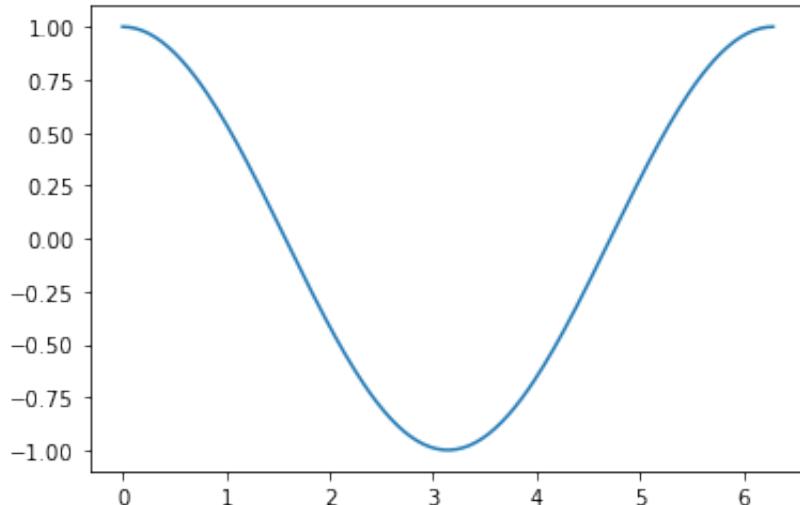
```
In [52]: for i in [0, 1, 2, 3, 4]: # La ligne se termine par :
    print('La valeur de i est :', i) # Le bloc de commandes est indenté
print('Nous sommes sortis de la boucle')
```

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
Nous sommes sortis de la boucle
```

Il faut remarquer ici qu'à la fin du `for` il est nécessaire d'introduire `:` et d'indenter le bloc de commandes à exécuter au sein de la boucle. La fin de l'indentation met fin à la boucle.

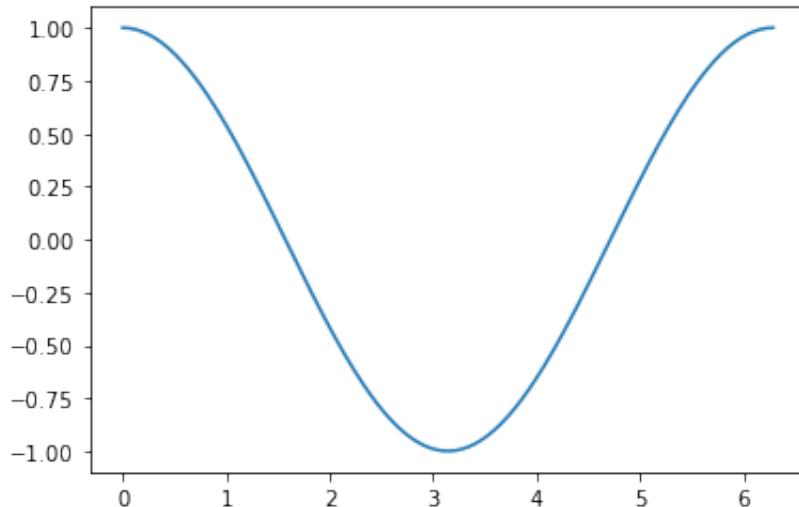
Nous pouvons utiliser les boucles `for` pour affecter des valeurs à des éléments d'un tableau

```
In [53]: x = np.linspace(0, 2*np.pi, 200)
y = np.zeros_like(x)
for i in range(len(x)):
    y[i] = np.cos(x[i])
plt.plot(x,y);
```



Notons que nous aurions dû tirer parti de la vectorisation offerte par NumPy et éviter d'écrire une boucle `for` :

In [54]: `plt.plot(x,np.cos(x));`



if/elif/else

La condition `if` permet d'exécuter un bloc de commandes pour autant que le résultat de l'évaluation de `if` soit `True`

In [55]: `a = 2
if a == 2:
 print('a est bien égale à', a)`

a est bien égale à 2

La condition `if` peut être suivie d'un `else` qui sera évalué si l'évaluation du `if` est `False`

In [56]: `a = 3
if a == 2:
 print('a est bien égale à', a)
else:
 print('a est différent de 2')`

a est différent de 2

Finalement, nous pouvons utiliser la commande `elif` pour rajouter des conditions avant l'utilisation du `else`

```
In [57]: a = 3
if a == 2:
    print('a est bien égale à', a)
elif a==3:
    print('a est bien égale à', a)
else:
    print('a est différent de 2 et de 3')
```

a est bien égale à 3

La boucle while/break

La boucle `while` permet d'exécuter un bloc de commandes jusqu'à ce qu'une condition soit rencontrée ou lorsqu'une commande `break` est rencontrée. La syntaxe est la suivante:

```
In [58]: a = 0
while a < 6:
    a += 1
    print(a)
```

1
2
3
4
5
6

Il est important d'être vigilant à ne pas créer de boucle infinie et d'utiliser la commande `break` après un certain nombre de passage dans la boucle.

```
In [59]: a = 0
while a < 60000:
    a += 1
    print(a)
    if a > 15:
        print('Sortie anticipée de boucle')
        break
print('Nous sommes sortis de la boucle')
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Sortie anticipée de boucle
Nous sommes sortis de la boucle
```

Exercice: La fonction exponentielle admet le développement en série

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Ecrire une fonction Python qui prend en entrée les variables `n` et `x`, et qui calcule cette série limitée aux `n+1` premiers termes.

```
In [ ]:
```

Les polynômes

Les polynômes peuvent être conçus et manipulés à travers la classe `numpy.polynomial` qui contient différentes possibilités (puissance, Chebyshev, Legendre...). Par exemple le polynôme

$$x^4 - 12x^3 + 5x$$

peut-être représenté par les coefficients repris dans la liste `coef` ci-dessous

In [60]: `coef = [0, 5, 0, -12, 1]`

In [61]: `from numpy.polynomial import Polynomial as poly`
`p = poly(coef)`
`p`

Out [61]: $x \mapsto 0.0 + 5.0x + 0.0x^2 - 12.0x^3 + 1.0x^4$

In [62]: `type(p)`

Out [62]: `numpy.polynomial.polynomial.Polynomial`

In [63]: `print(p)`

`poly([0. 5. 0. -12. 1.])`

Nous pouvons ensuite effectuer un nombre important d'opérations comme calculer la dérivée du polynôme, l'évaluer en un point, par exemple en `x = 5`, ou encore déterminer ses racines:

In [64]: `p.deriv(1)`

Out [64]: $x \mapsto 5.0 + 0.0x - 36.0x^2 + 4.0x^3$

In [65]: `p(5)`

Out [65]: -850.0

In [66]: `p.roots()`

Out [66]: `array([-0.62921183, 0. , 0.66413705, 11.96507478])`

Si un ensemble de points nous est donné mais que nous ne connaissons pas le polynôme associé, nous pouvons ajuster au sens des moindres carrés un polynôme d'un certain degré à ces points en utiliser la fonction `polyfit`

In [67]: `from numpy.polynomial.polynomial import polyfit`
`coef2 = polyfit([0, 1, 4, 5], [-1, 2, 1, 4], 3)`

```
In [68]: p2 = poly(coef2)
print(p2)

poly([-1.           5.16666667 -2.5           0.33333333])
```

Graphiques

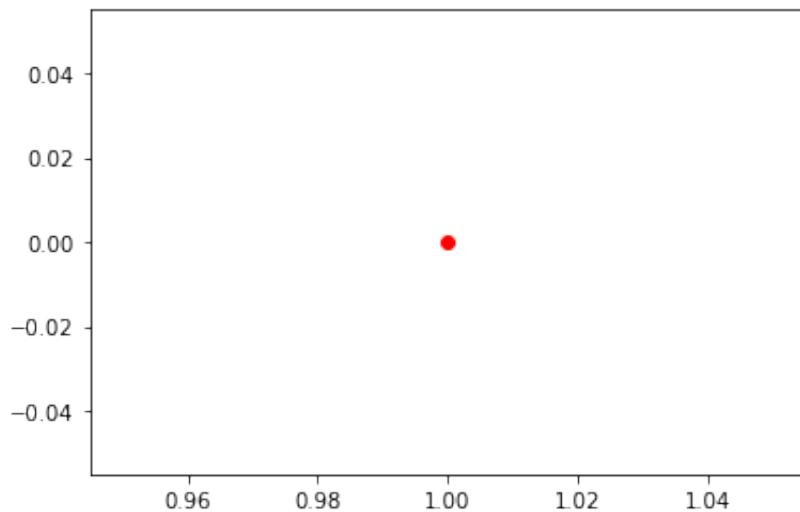
Les bases

Pour effectuer des graphiques dans Python, nous utiliserons la librairie `matplotlib` et la sous-librairie `pyplot`. La commande `%matplotlib inline` est ici utilisée pour ne pas faire apparaître les graphiques sur une page à part. L'ensemble des graphiques disponibles est décrit ici: [\(https://matplotlib.org/gallery.html\)](https://matplotlib.org/gallery.html)

```
In [69]: %matplotlib inline
import matplotlib.pyplot as plt
```

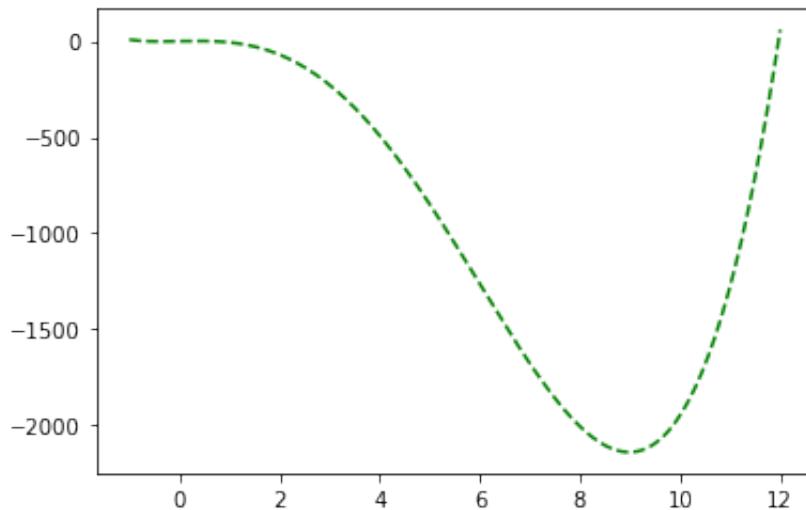
Nous pouvons maintenant faire appel aux fonctions graphiques en utilisant la commande `plt.fonction`, par exemple la fonction `plot(x, y, chaîne de caractères)` où la chaîne de caractères définit le type de marqueur ou de trait utilisé.

```
In [70]: plt.plot(1,0,'or');
```



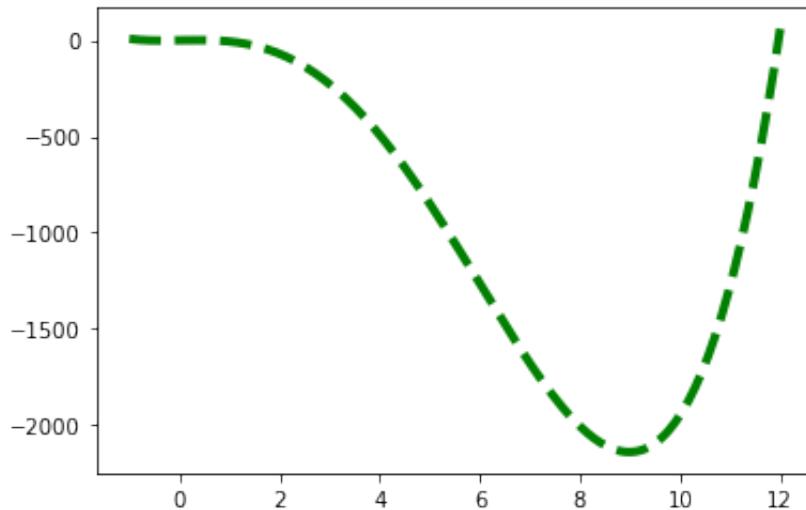
Nous pouvons ainsi visualiser le polynôme généré ci-dessus à l'aide des tableaux `x` et `y`

```
In [71]: x = np.linspace(-1, 12, 200)
y = np.zeros_like(x)
plt.plot(x,p(x),'g--');
```



La fonction `plot` peut prendre un grand nombre d'arguments optionnels à travers des mots-clés. La syntaxe est `fonction(mot-clé1=valeur1, mot-clé2=valeur2)`. Par exemple, pour tracer le polynôme en vert avec une épaisseur de trait de 4 on écrit

```
In [72]: plt.plot(x,p(x),'g--', linewidth = 4);
```



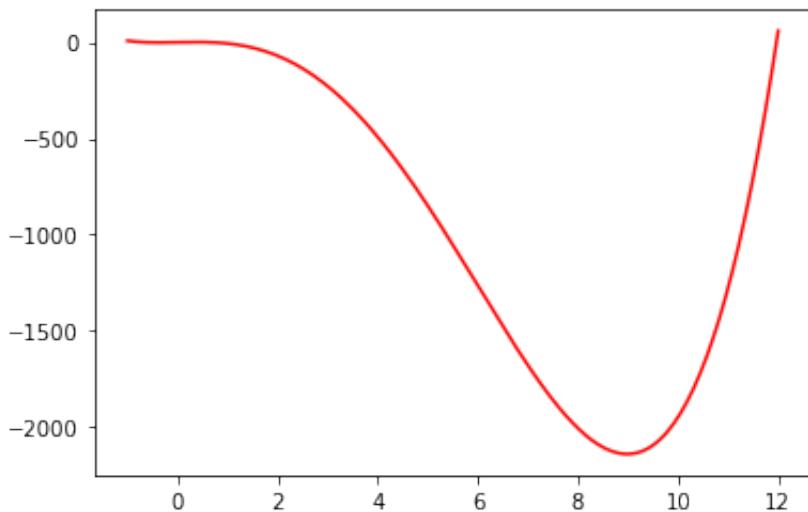
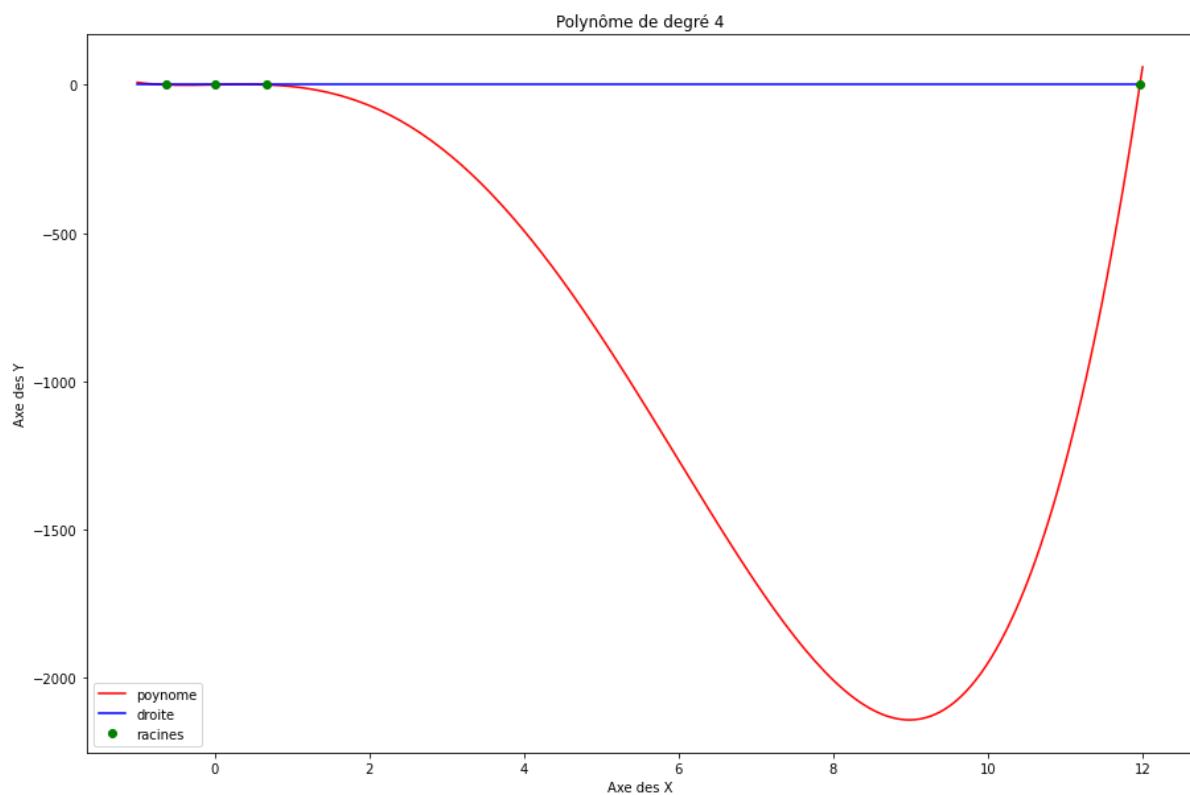
Afficher plusieurs courbes dans un graphique, créer une nouvelle figure

Pour créer une nouvelle figure, il faut utiliser la fonction `figure` de `matplotlib.pyplot`. On peut alors spécifier la taille de la figure. Les commandes suivantes utilisant la fonction `plot` utilisera alors cette figure, afin, par exemple, de superposer des courbes. On peut y ajouter une légende, un titre ainsi que labéliser les axes.

```
In [73]: plt.figure(figsize=(15, 10)) # Crée une nouvelle figure
plt.plot(x,p(x),'r');
plt.plot(x,np.zeros_like(x),'b');
plt.plot(p.roots(),np.zeros_like(p.roots()),'og')
plt.legend(['poynome','droite', 'racines'],loc='best') #Ajout d'un titre

plt.title('Polynôme de degré 4') #Ajout d'un titre
plt.ylabel('Axe des Y') # Labélation de l'axe des ordonnées
plt.xlabel('Axe des X') # Labélation de l'axe des abscisses

plt.figure() # Crée une nouvelle figure
plt.plot(x,p(x),'r');
```



Exercice: Tracez le graphique de la fonction

$$y(x) = e^{-0.8x} \sin \omega x$$

pour $\omega = 10$ rad/s et $x \in [0 \ 10]$ s.

In []:

Exercice: Soit la fonction

$$y(x) = 10 + 5e^{-x} \cos(\omega x + 0.5)$$

Ecrivez un script qui trace le graphique pour $\omega = 1, 3, 10$ rad/s et $x \in [0 \ 5]$ s. Les trois courbes doivent apparaître en vert, avec une ligne continue pour $\omega = 1$ rad/s, une ligne en traits discontinus pour $\omega = 3$ rad/s et une ligne en pointillés pour $\omega = 10$ rad/s.

In []:

Interpolation par splines

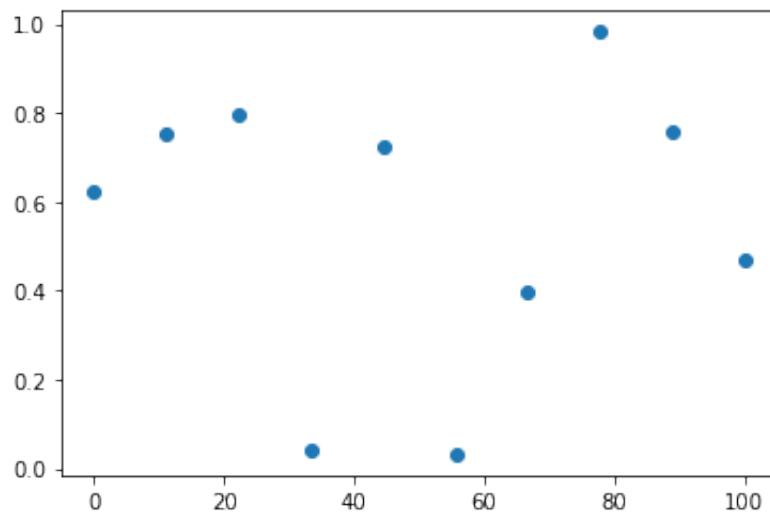
Python offre la possibilité de réaliser des interpolations par splines à travers la librairie SciPy et la sous-librairie `interpolate`. Nous utiliserons par exemple la fonction `CubicSpline`.

In [74]:

```
import numpy as np
import matplotlib
from scipy.interpolate import CubicSpline
%matplotlib inline
import matplotlib.pyplot as plt
```

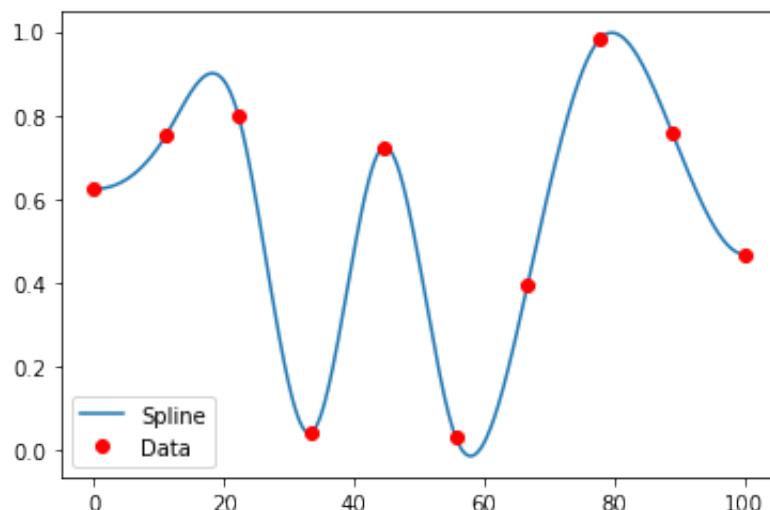
Considérons tout d'abord un ensemble de points x et y entre lesquels nous souhaitons interpoler.

```
In [75]: x = np.linspace(0,100,10)
y = np.random.rand(10)
plt.plot(x,y,'o');
```



```
In [76]: x_cs = np.linspace(0,100,500)
cs = CubicSpline(x,y,bc_type='clamped') #le paramètre bc_type permet
plt.plot(x_cs,cs(x_cs),label='Spline')
plt.plot(x,y,'ro', label='Data')
plt.legend(loc='best')
```

Out[76]: <matplotlib.legend.Legend at 0x7fc598546850>



Entrées et sorties

Plutôt que de créer des points aléatoirement comme dans l'exemple de la spline cubique, nous pourrions avoir besoin d'en importer de l'extérieur. Par exemple ces points pourraient être contenus dans une fichier txt `data.txt`. La fonction `loadtxt` de NumPy nous permet d'ouvrir ce fichier, de lire son contenu et de le mettre dans une nouvelle variable.

```
In [77]: import numpy as np  
a = np.loadtxt('data.txt')  
print(a)  
type(a)
```

```
[[ 0. -1.  
  1.  2.  
  3.  0.1  
  4. -0.1  
  5. -2.  
  6. -8.  
  7. -9. ]]
```

```
Out[77]: numpy.ndarray
```

Nous pouvons maintenant effectuer des opérations mathématiques sur le tableau créé (ici nous normalisons la seconde colonne) et ensuite le sauver sous format `.txt` à l'aide de la fonction `savetxt` de NumPy

```
In [78]: a[:,1]=a[:,1]/np.max(a[:,1])  
np.savetxt('normalised_data.txt',a)  
print(a)
```

```
[[ 0. -0.5  
  1.  1.  
  3.  0.05  
  4. -0.05  
  5. -1.  
  6. -4.  
  7. -4.5 ]]
```

D'autres possibilités existent à travers des fonctions telles que les fonctions de base `open` utilisée pour lire, écrire et modifier un fichier ou `input()` qui permet une saisie au clavier. De nombreux modules existent également pour lire différents types de fichier comme `xlrd` utilisé pour lire les fichiers du logiciel Microsoft Excel (`.xlsx`, `.xls`).

```
In [79]: prenom = input('Quel est votre prénom ? :')
```

```
Quel est votre prénom ? :Frederic
```

```
In [80]: print(prenom)
```

Frederic

```
In [3]: with open('data.txt') as f_data:  
    print(f_data.read())
```

0 -1
1 2
3 0.1
4 -0.1
5 -2
6 -8
7 -9

Exercice: Tracez le graphique de l'interpolation par spline cubique et de l'interpolation linéaire des points de données du fichier 'normalised_data.txt' ci-dessus.

```
In [ ]:
```

Résolution d'équations différentielles

Considérons l'équation différentielle de l'oscillateur de Van der Pol

$$\frac{dx_1}{dt} = x_2$$

$$\frac{dx_2}{dt} = \epsilon\omega(1 - x_1^2)x_2 - \omega^2 x_1$$

Avec les constantes ϵ et ω égales à 0.1 et 1 respectivement. Dans Python, cette équation différentielle peut être représentée par une fonction définie comme suit :

```
In [157]: def odefunction(t,y,const):
    # Système d'équations différentielles ordinaires défini par :
    #   t: le temps
    #   y: les variables du système
    # La fonction retourne dy, un array contenant les dérivées

    #Import nécessaire
    import numpy as np

    #Partie principale
    #Definitions de constantes
    epsilon=const[0]
    omega=const[1]

    dy = np.zeros(2)
    dy[0] = y[1]
    dy[1] = epsilon * omega * (1-y[0]**2)*y[1] - (omega**2)*y[0]

    return dy
```

Ces lignes de codes sont enregistrés dans un fichier portant le nom de `OdeFun.py`. La résolution numérique nécessite la librairie SciPy et la fonction `solve_ivp` de la sous librairie `scipy.integrate`. Ici la sortie d'`odefunction` fournit une solution de type `numpy.ndarray` mais nous pourrions également sortir un type `list`.

```
In [81]: %reset
from scipy.integrate import solve_ivp as ode45 # Defining the solve
from matplotlib import pyplot
import numpy
import OdeFun
```

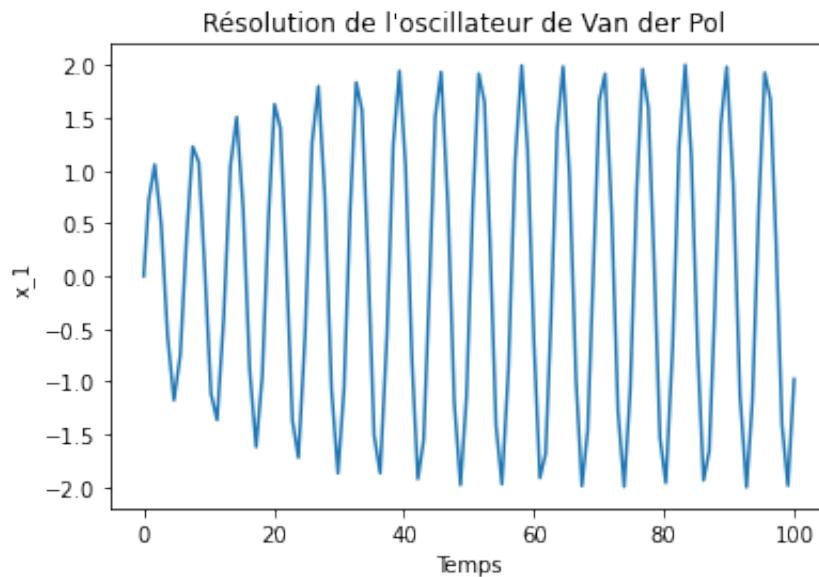
Once deleted, variables cannot be recovered. Proceed (y/[n])? y

```
In [82]: whos
```

Variable	Type	Data/Info
OdeFun	module	<module 'OdeFun' from '/U<...>n101/Tutoriel/0deFun.py'>
numpy	module	<module 'numpy' from '/Us<...>kages(numpy/_init__.py')>
ode45	function	<function solve_ivp at 0x7fc5c160dc10>
pyplot	module	<module 'matplotlib.pyplot'>

La fonction `scipy.integrate.ode_ivp` et ici renommée `ode45` utilise par défaut une méthode de Runge-Kutta explicite pour la résolution de l'équation et les paramètres d'intégration peuvent être ajustés avec des options. Le code suivant résout l'équation différentielle entre 0 et 12 et trace le résultat à l'écran :

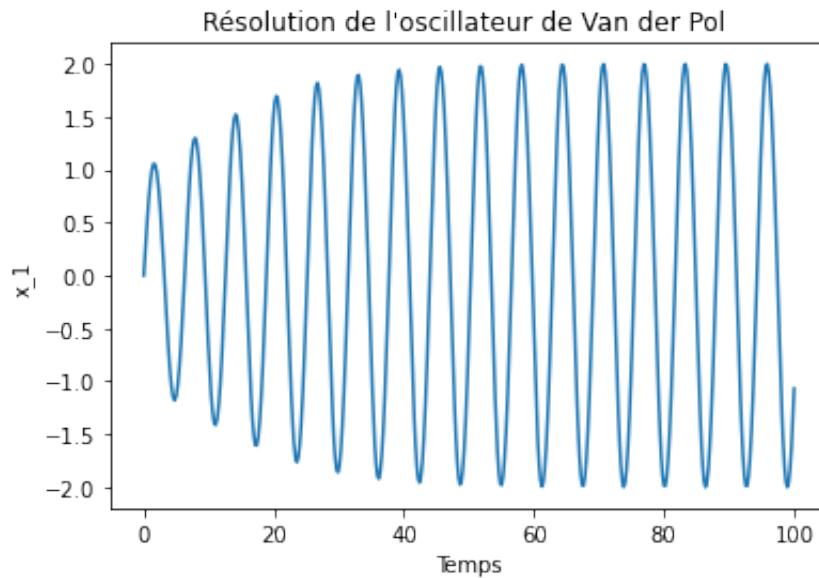
```
In [83]: c=[0.1, 1]
solution = ode45(lambda t, y: OdeFun.odefunction(t,y,c),[0,100],[0,
pyplot.plot(solution.t,solution.y[0,:]);
pyplot.title('Résolution de l\'oscillateur de Van der Pol'); #Ajou
pyplot.ylabel('x_1'); # Labélisation de l'axe des ordonnées
pyplot.xlabel('Temps'); # Labélisation de l'axe des abscisses
```



La fonction `lambda` permet de remplacer l'appel à `OdeFun.odefunction` qui admet normalement trois arguments (`t, y, c`) en entrée par un appel avec une fonction `lambda` à deux arguments (`t, y`) accepté par le solveur `scipy.integrate.ode_ivp`.

Le solveur `scipy.integrate.ode_ivp` offre un nombre important d'options. Ces dernières permettent par exemple de définir les valeurs de tolérance relative et absolue.

```
In [85]: solution = ode45(lambda t, y: OdeFun.odefunction(t,y,c),[0,100],[0,0]);
pyplot.plot(solution.t,solution.y[0,:]);
pyplot.title('Résolution de l\'oscillateur de Van der Pol'); #Ajout
pyplot.ylabel('x_1'); # Labélisation de l'axe des ordonnées
pyplot.xlabel('Temps'); # Labélisation de l'axe des abscisses
```

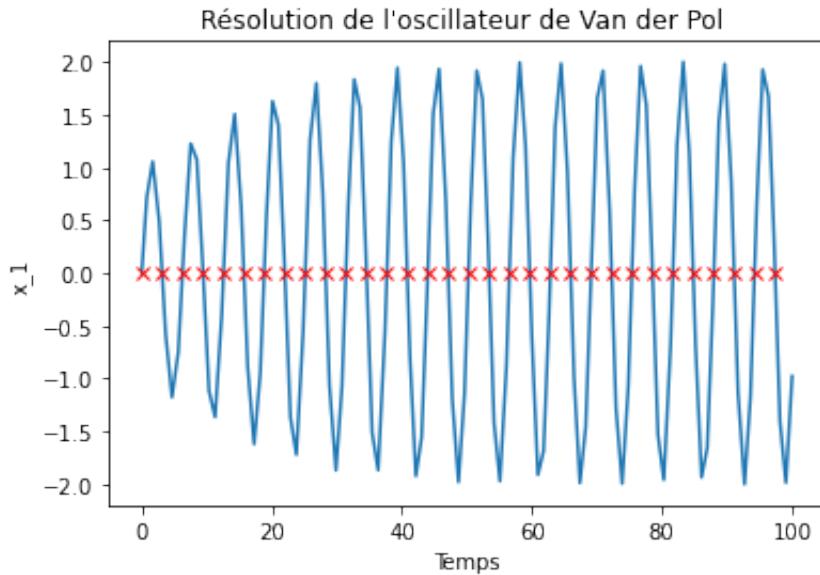


Il est souvent utile de déterminer quand la solution du système d'équations différentielles atteint une valeur particulière (typiquement 0). Ceci peut se faire grâce à la définition de l'option `events`. Ceci demande de créer une fonction définissant la variable à comparer puis de recalculer la solution. Les événements sont enregistrés dans `solution.t_events` et `solution.y_events`.

Pour plus d'informations, n'hésitez pas à consulter l'aide de la fonction `scipy.integrate.solve_ivp`.

```
In [86]: def event(t,y): return y[0]
```

```
In [87]: solution = ode45(lambda t, y: OdeFun.odefunction(t,y,c),[0,100],[0,
pyplot.plot(solution.t,solution.y[0,:]);
pyplot.plot(solution.t_events[0],solution.y_events[0][0:,0],'rx');
pyplot.title('Résolution de l\'oscillateur de Van der Pol'); #Ajou
pyplot.ylabel('x_1'); # Labélisation de l'axe des ordonnées
pyplot.xlabel('Temps'); # Labélisation de l'axe des abscisses
```



Exercice: Pour l'oscillateur de Van der Pol ci-dessus, trouvez tous les points pour lesquels la valeur de x_1 vaut 1 et tracer un graphique les illustrant.

```
In [ ]:
```

Exercice: Résolvez l'équation différentielle suivante

$$\frac{dy_1}{dt} = \cos(y_2) * y_3$$

$$\frac{dy_2}{dt} = -y_1/y_3$$

$$\frac{dy_3}{dt} = -0.8 y_1 y_2$$

avec les conditions initiales $y_1(0) = 0$, $y_2(0) = 1$ et $y_3(0) = 1$ pour un temps allant de 0 à 100 et tracer l'évolution des trois variables dans une même graphique. Trouvez les zéros de la variable y_2 si il y en a.

```
In [ ]:
```

Débogage

Spyder comprend un debugger intégré à l'éditeur: <https://docs.spyder-ide.org/current/debugging.html> (<https://docs.spyder-ide.org/current/debugging.html>)

Tout d'abord, lorsque l'on édite un fichier .py, l'éditeur détectera automatique des erreurs de syntaxe par exemple et affichera un cercle rouge avec une croix dedans pour signaler l'erreur. Ce type d'erreur critique ne permettra pas au script de s'exécuter. L'éditeur peut également afficher une icône avertissement pour indiquer par exemple la non-utilisation de la librairie NumPy. Il ne s'agit pas d'une erreur critique et le programme pourra s'exécuter.

The screenshot shows the Spyder IDE interface with a code editor window open. The file path is /Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py. The code editor contains the following Python code:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Wed Jan 20 11:19:08 2021
5
6  @author: fred
7  """
8  import numpy as np
9  import matplotlib.pyplot as plt
10 size = 20
11 z = np.zeros(size)
12 for i in range(size)
13     Code analysis
14
15
16
```

A tooltip box is displayed over the line "for i in range(size)". The box contains:

- Code analysis
- Invalid syntax (pyflakes E)

Handwritten annotations in green and red are present:

- A green arrow points from the word "oubli" to the opening parenthesis of the "range" function call.
- A green arrow points from the text "après le for." to the closing parenthesis of the "range" function call.
- A red arrow points from the text "Erreur critique" to the tooltip box.
- The text "oubli de la syntaxe : après le for." is written in green near the top right of the code editor.
- The text "Erreur critique" is written in red at the bottom left of the code editor.

The screenshot shows the Spyder IDE interface with the file `debug.py*` open. The code editor displays the following Python script:

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4     Created on Wed Jan 20 11:19:08 2021
5
6     @author: fred
7 """
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import math
11
12
13 # 'math' imported but unused (pyflakes E)
14 for i in range(10):
15     z[i] = i**2 + 4
16
17 plt.plot(range(size),z)
```

A yellow bracket on the left side of the code editor highlights the first two lines of code. A yellow callout box labeled "Avertissement." covers the line `# 'math' imported but unused (pyflakes E)`. Handwritten annotations in green and yellow are present on the right side of the code editor:

- A green arrow points from the handwritten text "Librairie math" to the `import math` statement.
- A green arrow points from the handwritten text "importé mais" to the `# 'math' imported but unused (pyflakes E)` annotation.
- A green arrow points from the handwritten text "Mon utilité." to the `# 'math' imported but unused (pyflakes E)` annotation.
- A yellow bracket on the left side of the code editor highlights the first two lines of code.

En plus de cette vérification en temps réel, Spyder propose un outil qui permet d'arrêter un programme en cours d'exécution pour examiner la valeur des différentes variables et détecter d'éventuelles erreurs. A cet effet, des points d'arrêt doivent être créés au préalable dans le(s) fichier(s) .py. Pour créer un point d'arrêt (breakpoint), il suffit de cliquer à droite du numéro de la ligne correspondante et un point rouge apparaît à gauche de la ligne courante.

L'exécution du programme est alors réalisée via le bouton Debug file. Le programme s'arrête au point d'arrêt indiqué par une flèche. Vous pouvez examiner le contenu des variables du Workspace, avancer d'une ligne, continuer l'exécution jusqu'au prochain point d'arrêt ou arrêter le programme. Lorsqu'une ligne fait appel à une fonction, il est possible de rentrer à l'intérieur de cette fonction ou de sauter directement à la ligne suivante.

Lancer le mode
Debug

→ points d'arrêt

Examen des variables
en cours d'exécution

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Wed Jan 20 11:19:08 2021
5
6 @author: fred
7 """
8 import numpy as np
9 import matplotlib.pyplot as plt
10
11 size = 20
12 z = np.zeros(size)
13 for i in range(size):
14     z[i] = i**2 + 4
15
16 plt.plot(range(size),z)

```

In [54]: debugfile('/Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py', wdir='/Users/fred/Dropbox/Codes/Python101/Tutoriel')
> /Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py(3)<module>()
 1 #!/usr/bin/env python3
 2 # -*- coding: utf-8 -*-
--> 3 """
 4 Created on Wed Jan 20 11:19:08 2021
 5

IPdb [1]: !continue
> /Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py(14)<module>()
 12 z = np.zeros(size)
 13 for i in range(size):
--> 14 z[i] = i**2 + 4
 15
 16 plt.plot(range(size),z)

IPdb [2]: |

Handwritten notes in the screenshot:

- Suivi de z pour le premier élément n° 0 : $(0^2 + 4) \Rightarrow 4$
- point d'arrêt suivant

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Wed Jan 20 11:19:08 2021
5
6 @author: fred
7 """
8 import numpy as np
9 import matplotlib.pyplot as plt
10
11 size = 20
12 z = np.zeros(size)
13 for i in range(size):
14     z[i] = i**2 + 4
15
16 plt.plot(range(size),z)

```

```

In [54]: debugfile('/Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py', wdir='/Users/fred/Dropbox/Codes/Python101/Tutoriel')
> /Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py(3)<module>()
  1 #!/usr/bin/env python3
  2 # -*- coding: utf-8 -*-
  3 """
  4 Created on Wed Jan 20 11:19:08 2021
  5
  6
  7
  8
  9
  10
  11
  12
  13
  14
  15
  16

IPdb [1]: !continue
> /Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py(14)<module>()
  12 z = np.zeros(size)
  13 for i in range(size):
  14     z[i] = i**2 + 4
  15
  16 plt.plot(range(size),z)

IPdb [2]: !continue
IPdb [2]:

```

Des fonctionnalités de débogage sont également accessibles lorsqu'un programme produit une erreur. Dans la console, le message d'erreur en rouge indique alors la ligne à laquelle l'erreur s'est produite. On peut y accéder directement en cliquant sur le numéro de la ligne.

```

Spyder (Python 3.8)
/Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py
HelloWorld.py PolarToCartesian.py chgtcoord.py debug.py

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Wed Jan 20 11:19:08 2021
5
6 @author: fred
7 """
8 import numpy as np
9 import matplotlib.pyplot as plt
10
11 size = 20
12 z = np.zeros(size)
13 for i in range(size)    ← il manque :
14     z[i] = ix**2 + iy
15
16 plt.plot(range(size), z)

```

Il manque :
après la for

cliquez ici vous
conduira

Console 1/A

```

In [59]: %run debug.py
File "/Users/fred/Dropbox/Codes/Python101/Tutoriel/debug.py", line 13
    for i in range(size)
               ^
SyntaxError: invalid syntax

In [60]:

```

Exercice: Déboguer le code suivant:

```

In [ ]: %reset
a == 2
b == 3

def somme(a,b)
    c = a + b
    return c

somme(12 14)

if a==2;
    print(a est égale à deux)
else
    print(a, "est différent de deux")

```

Analyse des performances

Des outils de performance avancée existent pour analyser l'efficacité des programmes sur Python : <https://docs.python.org/3/library/profile.html> (<https://docs.python.org/3/library/profile.html>)

Le module `cProfile` qui permet de regarder combien de temps le programme exécuté passe dans chaque fonction et le nombre d'appels des fonctions:

`ncalls` : le nombre d'appel à une fonction

`tottime` : temps total passé dans une fonction sans compté le temps passé dans les appels aux sous-fonctions

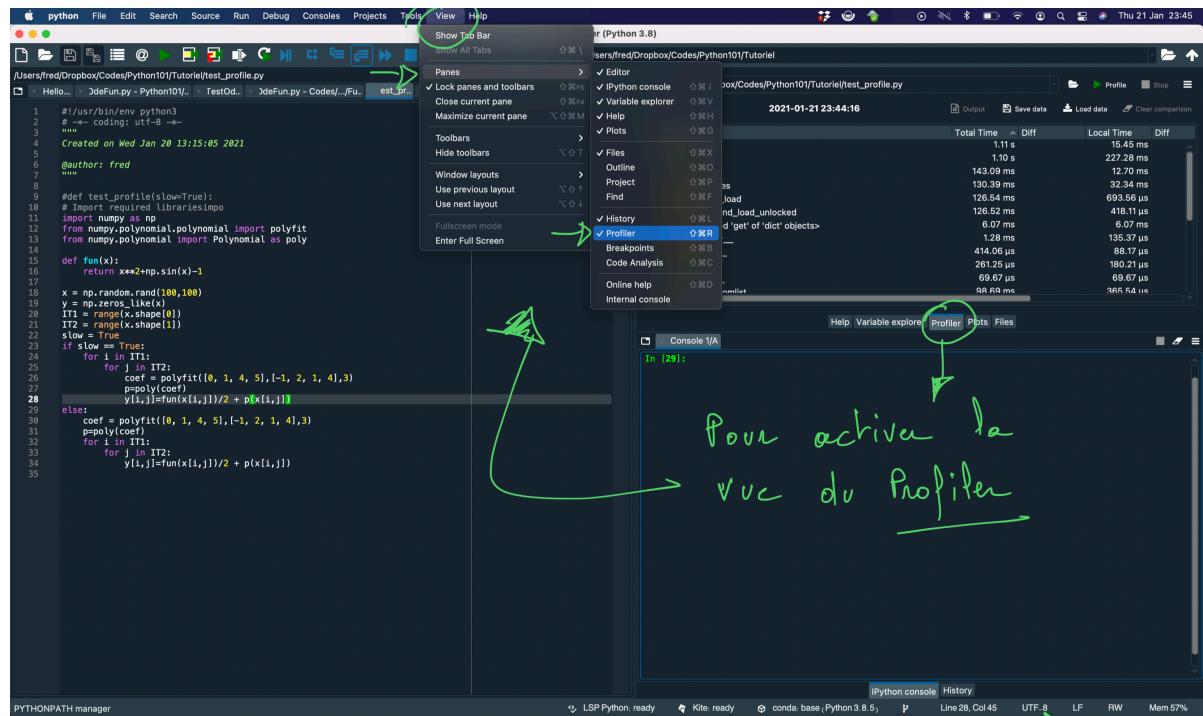
`percall` : temps passé pour les appels

`cumtime` : temps cumulé passé dans la fonction et les sous-fonctions

`percall` : `cumtime` divisé par les appels

`filename:lineno(function)` : données des fonctions

L'IDE Spyder nous offre une interface graphique lisible nous permettant de rapidement analyser le temps pris nos scripts Python. Pour y accéder, il faut tout d'abord s'assurer d'avoir activé le Profiler à travers le menu View > Panes > Profiler



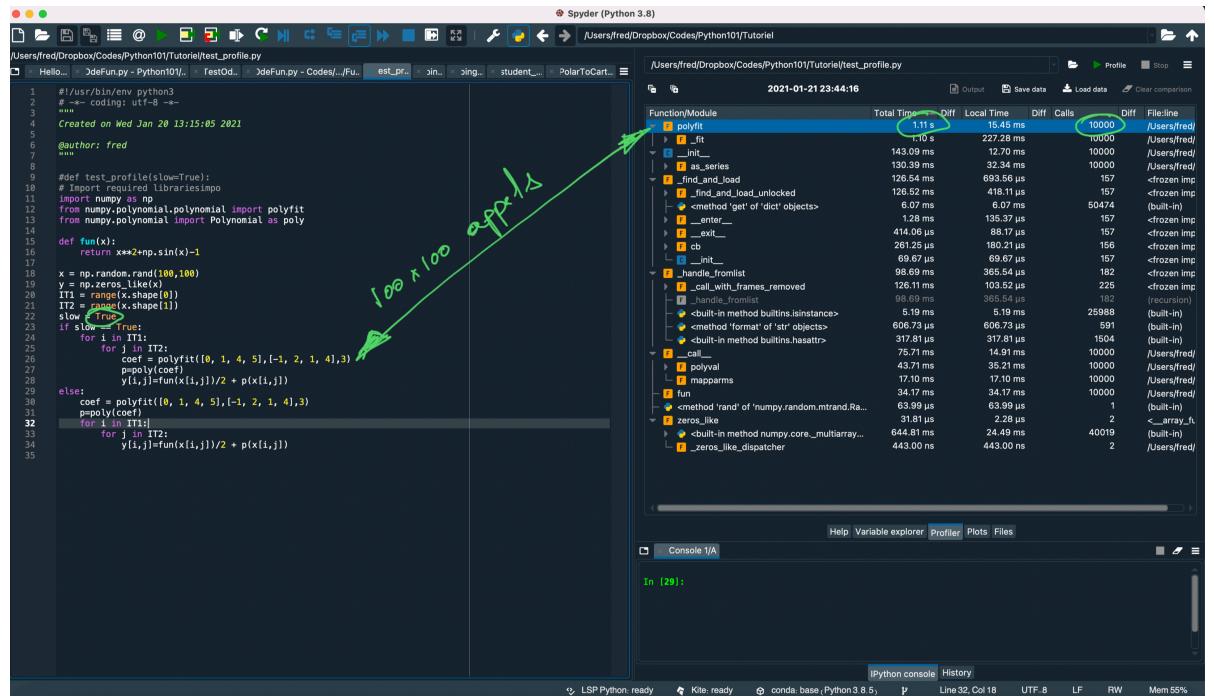
Considérons maintenant le script `test_profile.py` ci-dessous qui comprend deux boucles, une fonction propre `fun` et qui utilise des fonctions de la librairie NumPy pour réaliser une interpolation d'une fonction définie en quelques points.

```
In [ ]: import numpy as np
from numpy.polynomial.polynomial import polyfit
from numpy.polynomial import Polynomial as poly

def fun(x):
    return x**2+np.sin(x)-1

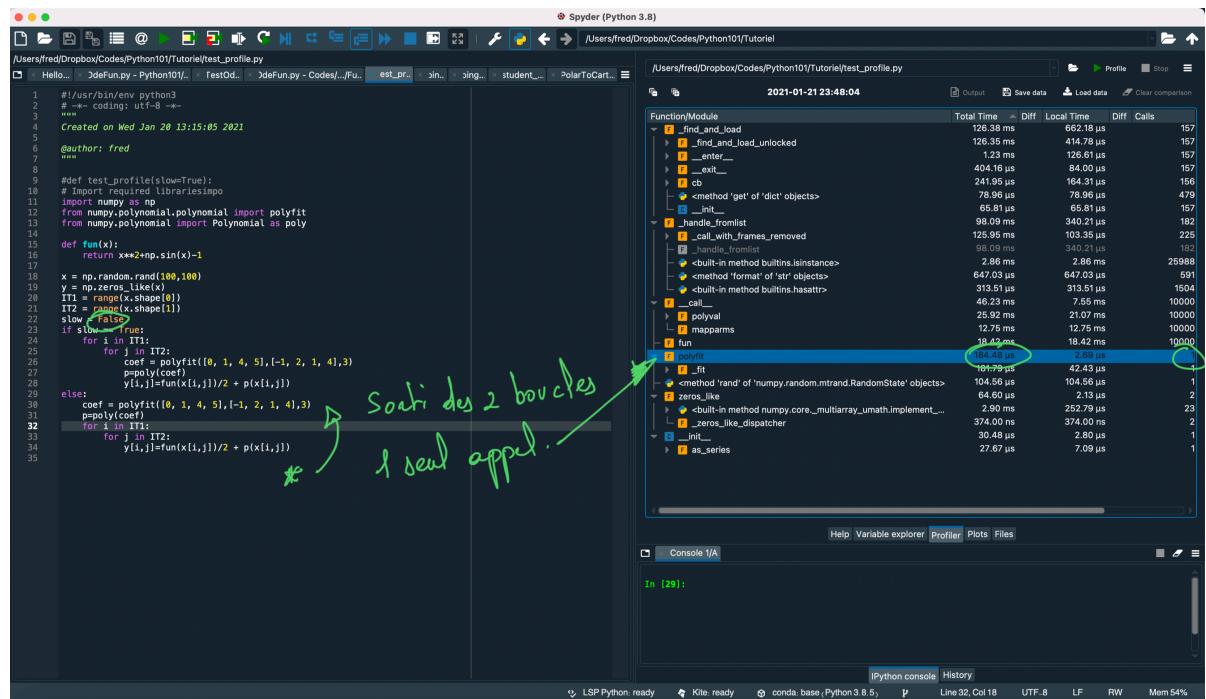
x = np.random.rand(100,100)
y = np.zeros_like(x)
IT1 = range(x.shape[0])
IT2 = range(x.shape[1])
slow = False
if slow == True:
    for i in IT1:
        for j in IT2:
            coef = polyfit([0, 1, 4, 5],[-1, 2, 1, 4],3)
            p=poly(coef)
            y[i,j]=fun(x[i,j])/2 + p(x[i,j])
else:
    coef = polyfit([0, 1, 4, 5],[-1, 2, 1, 4],3)
    p=poly(coef)
    for i in IT1:
        for j in IT2:
            y[i,j]=fun(x[i,j])/2 + p(x[i,j])
```

Nous allons la tester avec l'aide du Profiler et classer les résultats par temps cumulé afin de voir quelle partie du code prend le plus de temps à tourner. Tout d'abord, nous faisons tourner une version "lente" du code avec la variable `slow = True` et nous obtenons le résultat suivant:



On peut voir qu'à l'exécution du script avec `slow = True`, nous calculons les coefficients d'un polynôme à l'aide de `polyfit` à chaque fois dans les deux boucles, alors que ces coefficients sont indépendants des boucles. Le temps passé à appeler 10000 la fonction `polyfit` est de 1.11 secondes.

Afin d'améliorer l'efficacité de cette fonction, nous pouvons sortir le calcul des coefficients des boucles et ne les calculer qu'une seule fois (`slow = False`). On peut alors vérifier que `polyfit` n'est appelée qu'une seule fois et que le temps mis pour exécuter le script est beaucoup plus faible.



Il existe également des fonctions permettant de chronométrier l'exécution de Python dans Spyder comme `%time` ou `%timeit`.

Exercice: Créez une script de votre choix pour comparer l'utilisation de la fonction `sum()` de NumPy avec une votre propre fonction qui devra utiliser une boucle `for` pour calculer la somme des éléments d'un tableau à 1 dimensions de type `ndarray`. Analysez la performance de votre fonction par rapport à la fonction de NumPy .

In []:

Fin du tutoriel

Nous voici au bout de ce tutoriel. Quelques conseils avant de débuter le projet. Lancez-vous avant toute chose à plein pieds dans le langage Python, essayez les différentes fonctionnalités présentées ici et n'hésitez pas à vous renseigner ailleurs et de partager entre vous vos bonnes pratiques.

Complétez ce tutoriel avant de démarrer le projet, et soyez certain.e.s de le maîtriser avant de passer l'examen oral ;)

In []: `%reset`

Print dependencies

In [1]: `%load_ext watermark`

In [2]: `%watermark -v -m -p numpy,scipy,matplotlib,time,watermark
print("")
%watermark -u -n -t -z`

```
Python implementation: CPython
Python version       : 3.8.5
IPython version     : 7.19.0

numpy      : 1.19.2
scipy      : 1.5.2
matplotlib: 3.3.2
time       : unknown
watermark  : 2.1.0

Compiler    : Clang 10.0.0
OS          : Darwin
Release     : 20.2.0
Machine     : x86_64
Processor   : i386
CPU cores   : 16
Architecture: 64bit
```

Last updated: Fri Jan 22 2021 19:36:16CET

Type *Markdown* and *LaTeX*: α^2

