Introduction à `Python`

Thibaut Lacroix
lacroix@insp.upmc.fr

MPSI - Juin 2021

> Quidquid praecipies, esto brevis.
>
> Horace, Ars Poetica, 335

Contents

1 Introduction **3**
- 1.1 Qu'est-ce que `Python` ? . . . 3
- 1.2 À quoi sert `Python` ? . . . 3

2 Installation **3**
- 2.1 Installation Simple . . . 3
  - 2.1.1 Sous Mac . . . 3
  - 2.1.2 Sous Windows . . . 4
  - 2.1.3 Sous Ubuntu . . . 4
- 2.2 Anaconda . . . 4
  - 2.2.1 Sous Windows ou Mac . . . 4
  - 2.2.2 Sous Ubuntu . . . 4

3 Éditeur et IDE **5**

4 Premiers Pas **5**
- 4.1 Interpréteur Python . . . 5
  - 4.1.1 Les opérateurs . . . 6
- 4.2 Éditeur et IDE . . . 8

5 Rédiger un script **8**
- 5.1 Commentaires . . . 9
- 5.2 Structure générale . . . 9
- 5.3 Variable . . . 9
  - 5.3.1 Types primaires . . . 10
  - 5.3.2 Listes, tuples et dictionnaires . . . 12
- 5.4 Conditions . . . 15
- 5.5 Boucle . . . 16
  - 5.5.1 For . . . 16
  - 5.5.2 While . . . 17
- 5.6 Fonction . . . 18
- 5.7 Zen of Python . . . 21
- 5.8 Complexité algorithmique . . . 21

6 Les librairies **22**
- 6.1 Première librairie : NumPy . . . 23
- 6.2 Graphiques avec PyPlot . . . 25
- 6.3 Application : Le théorème central limite . . . 28

7 Charger des données **35**
7.1 Lire et écrire des chaînes de caractères 35
7.2 Lire et écire des nombres 35
7.3 Lire et écrire des données structurées 35

8 Équations différentielles **36**
8.1 Application : l'équation de Schrödinger à $1D$ 38

9 Ce que nous n'avons pas vu **40**

1 Introduction

L'objectif de ce cours est de vous donner les bases du langage de programmation `Python` et surtout de vous rendre autonome. À la fin de ce cours, vous ne saurez pas tout sur `Python` et sur la programmation mais vous serez capable de

- comprendre et rédiger des programmes par vous-même,
- concevoir un algorithme pour résoudre un problème,
- chercher (et de comprendre !) les réponses à vos questions par vous-même.

1.1 Qu'est-ce que `Python` ?

`Python`[1] est un langage de programmation de *haut niveau*[2] simple à prendre en main, *libre* et *open source* et possédant un grand nombre de librairies[3] utilisables par tous. Ainsi, en règle générale, en `Python` vous n'aurez pas à recoder des algorithmes ou des méthodes mathématiques connus et pourrez vous concentrer sur leurs applications. `Python` est un langage *interprété* ce qui signifie que les instructions sont lues par la machine ligne par ligne. Ce qui signifie que

- le code est lisible par un humain,
- vous n'avez besoin que d'un éditeur de texte et de l'interpréteur pour exécuter un script `Python`,
- qu'un script `Python` est *portable* : il peut être lu sur toute machine peut importe son système d'exploitation.

Un autre intérêt de `Python` est qu'il ne s'agit pas d'un langage rigide : un certain nombre de chose peuvent être définie implicitement et l'interpréteur fera le reste.

1.2 À quoi sert `Python` ?

`Python` est un langage flexible largement utilisé dans au moins trois domaines d'applications :

- Résoudre des problèmes non soluble analytiquement / Simulation (mécanique des fluides, astrophysique, physique quantique, etc.)
- Analyse de données
- Programmation Web (Instagram, Netflix ou encore Dropbox utilisent `Python`)

2 Installation

2.1 Installation Simple

Vous pouvez installer `Python` seul sur Windows, MacOS et Linux en vous rendant sur cette page. Cependant je vous recommande d'installer une distribution scientifique ouverte appellée *Anaconda*, présentée en section 2.2, qui contient à la fois `Python`, un grand nombre de librairies scientifiques et des logiciels utiles pour programmation et l'analyse de données.

2.1.1 Sous Mac

Décompressez le dossier que vous avez téléchargé en double-cliquant dessus. Puis cliquez de nouveau sur le document afin de lancer l'installation:
Ou bien, ouvrez une console et tapez

```
brew install python3
```

[1] Le nom du langage provient de l'émission de la BBC « Monty Python's Flying Circus ».
[2] Ce qui signifie que sa syntaxe est plus proche de la façon dont nous nous exprimons que du langage machine utilisé par votre ordinateur.
[3] C-à-d des paquets de fonctions déjà codées et disponnible "sur étagère".

2.1.2 Sous Windows

Enregistrez le dossier à télécharger puis suivez les instructions.

2.1.3 Sous Ubuntu

La plupart des distributions Linux embarquent `Python`, vous pouvez vérifier si c'est le cas pour vous (pour Debian, Ubuntu et les apparentés) avec la commande

Ligne de commande

```
$ python3 --version
```

Sinon, tapez

Ligne de commande

```
$ sudo apt-get install python3
```

dans votre console.

2.2 Anaconda

Pour commencer, téléchargez la distribution Anaconda correspondant à votre système d'exploitation, en `Python` version 3 : https://www.anaconda.com/distribution/.

2.2.1 Sous Windows ou Mac

Téléchargez l'installeur Windows ou macOS, et double-cliquez pour lancer l'installation.

2.2.2 Sous Ubuntu

Dans un terminal, entrez l'instruction suivante :

Ligne de commande

```
bash ~/Downloads/Anaconda3-5.3.0-Linux-x86_64.h
```

L'installeur affiche : "In order to continue the installation process, please review the license agreement." Affichez les termes de la licence d'utilisation, scrollez tout en bas et cliquez sur yes pour accepter. Répondez "yes" à la question qui suit :

Ligne de commande

Do you wish the installer to prepend the Anaconda3 install location to PATH in your /home/username/.bashrc ? [yes|no]

La commande précédente a ajouté au fichier .bashrc le chemin du dossier dans lequel se trouve Anaconda. Ainsi, vous pourrez lancer Anaconda directement en tapant seulement "anaconda" dans votre terminal. Cette opération prendra effet au redémarrage de votre ordinateur, ou dès l'exécution de cette commande :

Ligne de commande

```
source .bashrc
```

3 Éditeur et IDE

En théorie, un simple éditeur de texte est suffisant pour programmer en `Python`. En pratique, utiliser un éditeur de texte possédant des fonctions pensées pour la programmation (comme la numérotation des lignes, la gestion automatique de l'indentation et des parenthèses, etc.) simplifie grandement la vie. Certain apprécient aussi ce que l'on nomme un *IDE* pour *Integrated Development Environments*, c-à-d un logiciel dans lequel on peut à la fois rédiger son code, l'exécuter, afficher les résultats et le déboguer. Une liste d'éditeurs et d'IDE peut être trouvée ici : https://wiki.python.org/moin/PythonEditors. Je recommande :

- Atom (multi-plateforme)
- Spyder (multi-plateforme et inclu dans Anaconda)
- gedit (multi-plateforme)
- VSCode (multi-plateforme)

4 Premiers Pas

4.1 Interpréteur Python

Maintenant que Python est intstallé, nous pouvons ouvrir l'interpréteur et faire nos premiers pas.
Sur Ubuntu et Mac, ouvrez un terminal et taper

Ligne de commande

```
python3
```

Sur Windows, lancer Python IDLE depuis le menu.
Cette fenêtre est ce qu'on appelle le *REPL*, qui est l'acronyme de *Read-Evaluate-Print-Loop*. Lorsque vous tapez dans la fenêtre le chiffre 5

Ligne de commande

```
>>> 5
5
```

La série suivante d'étapes se passent

1. Read: Python lit 5
2. Evaluate: Python évalue cette entré et comprend qu'il s'agit d'un nombre
3. Print: Python affiche l'éaluation de l'entrée
4. Loop: Python revient dans son état initiale et est prêt à recevoir une nouvelle commande

L'interpréteur dynamique de Python est une de ses caractéistiques les plus pratiques. Il permet de tester rapidement des idées et des commandes sans avoir à rédiger un script entier. Nénmoins, l'interpréteur par défaut est un peu limité. C'est pourquoi nous utiliseront plutôt un interpréteur plus élaboré nommé `IPython` (et plus tard des notebooks interactifs). `IPython` est pré-installé avec Anaconda.

Ligne de commande

```
ipython3
```

4.1.1 Les opérateurs

L'interpréteur Python peut être utilisé comme une calculatrice sophistiquée, ce qui va nous permettre d'introduire des commandes simples.

Arithmétique Python possède pré-implémentés les opérateurs arithmétiques habituels comme l'addition ou la multiplication.

Opérateur	Nom	Exemple
+	Addition	2 + 2
–	Soustraction	3 – 1
*	Multiplication	5 * 3
/	Division	5 / 2

Mais aussi quelques opérations plus complexes. La priorité est celle habituellement définie en mathématiques mais lorsque l'on pense qu'une instruction pourrait se révéler ambiguë, on n'hésitera pas à employer des parenthèses:

Opérateur	Nom	Exemple
%	Modulo	5 % 2
//	Division entière	9 // 2
**	Exponentiation	2 ** 4

Voici quelques exemples basiques

Ligne de commande

```
>>> 2 + 3 * 3
11
>>> (2 + 3) * 3
15
>>> 1 + 2**2
5
>>> 2/2*8
8.0
```

Exercice 1

Testez ces exemples et d'autres par vous même.

Booléen Un booléen est un type de variable à deux états (généralement notés **Vrai** et **Faux**), destiné à représenter les valeurs de vérité de la logique et l'algèbre booléenne. En Python, les valeurs booléennes sont primitives et sont représentées par

```
True
False
```

Comme les nombres, ces variables peuvent être manipulées avec des opérateurs booléens *not*, *and* et *or* qui suivent les tables de vérités suivantes

NON	0	1
	1	0

ET	0	1
0	0	0
1	0	1

OU	0	1
0	0	1
1	1	1

On remarquera que *NON* est un opérateur qui ne prend qu'un seul argument (un opérateur *unaire*) alors que *ET* et *OU* prennent deux arguments (ce sont des opérateurs binaires).

Exercice 2 Tables de vérité

Vérifiez que ces tables de vérité sont vérifiées.

Ligne de commande

```
>>> # Négation avec not
>>> not True
False
>>> not False
True
>>> # On note que "and" et "or" sont sensibles à la casse
>>> True and False
False
>>> False or True
True
```

Python interprète également les chiffres 0 et 1 comme des synonymes de `False` et `True`.

Ligne de commande

```
>>> # Utilisation des opérations booléennes avec des entiers :
>>> 0 and 2
0
>>> -5 or 0
-5
>>> 0 == False
True
>>> 2 == True
False
>>> 1 == True
True
```

Comparaisons Les nombres peuvent être comparés entre eux avec des opérateurs de comparaison comme *égale*, *différent*, *plus grand que*, etc. Ces opérateurs sont binaires et renvoient un booléen donnant le résultat de la comparaison.

Ligne de commande

```
>>> # On vérifie une égalité avec ==
>>> 1 == 1
True
>>> 2 == 1
False
>>> # On vérifie une inégalité avec !=
>>> 1 != 1
False
>>> 2 != 1
True

>>> # Autres opérateurs de comparaison
>>> 1 < 10
True
>>> 1 > 10
False
>>> 2 <= 2
True
>>> 2 >= 2
True
>>> # On peut enchaîner les comparaisons
>>> 1 < 2 < 3
True
>>> 2 < 3 < 2
False
```

4.2 Éditeur et IDE

En théorie, un simple éditeur de texte est suffisant pour programmer en Python. En pratique, utiliser un éditeur de texte possédant des fonctions pensées pour la programmation (comme la numérotation des lignes, la gestion automatique de l'indentation et des parenthèses, etc.) simplifie grandement la vie. Certain apprécient aussi ce que l'on nomme un *IDE* pour *Integrated Development Environments*, c-à-d un logiciel dans lequel on peut à la fois rédiger son code, l'exécuter, afficher les résultats et le déboguer. Une liste d'éditeurs et d'IDE peut être trouvée ici : https://wiki.python.org/moin/PythonEditors. Je recommande :

- Atom (multi-plateforme)
- Spyder (multi-plateforme et inclu dans Anaconda)
- gedit (multi-plateforme)
- VSCode (multi-plateforme)

5 Rédiger un script

Vous allez pouvoir écrire votre premier programme dans un unique document (un script). Celui-ci, comme le veut la tradition en programmation, affichera un simple texte de salutation. Dans un nouveau document de votre éditeur de texte ou de votre IDE, tapez :

```python
print("Hello world!")
```

puis enregistrez le fichier sous le nom `main.py`. Vous pouvez l'exécuter directement depuis votre IDE ou en ouvrant un interpréteur dans le répertoire où se trouve le ficher puis en entrant


```
>>> run main.py
```

On en déduit que la fonction `print()` sert à afficher du texte et que les textes (ou plus précisément les *chaînes de caractères*) commencent et se terminent par des guillemets droits ".

5.1 Commentaires

Les commentaires sont des chaînes de caractères qui ne sont pas interprétées par Python. Ils sont essentiels pour assurer la compréhension du code par d'autres personnes que celle le rédigeant et par les rédacteurs eux-mêmes dans le future (ce qui est évident àl'instant t ne l'est plus forcément à t+ 3 mois). Les commentaires sont donc la pour expliquer le fonctionnement de l'algorithme ou des parties du code particulièrement complexes ou opaques. Un code non-commenté est un mauvais code, cependant attention à ne pas sur-commenter en expliquant ce qui n'a pas besoin d'être expliqué.

```python
# Un commentaire d'une ligne commence par un dièse

"""
Les commentaires de plusieurs lignes
sont placés entre une paire de 3 guillemets doubles (")
et sont souvent utilisées comme explications d'algorithme
ou définitions de fonction.
"""
```

5.2 Structure générale

1. Librairies 2. Méthodes 3. Définition des variables 4. Appels et operations

```python
import numpy as np
from qutip import sigmax, sigmaz
from qutip.bloch_redfield import bloch_redfield_tensor

delta = 0.2 * 2*np.pi
eps0 = 1.0 * 2*np.pi
gamma1 = 0.5

H = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()

def ohmic_spectrum(w):
  if w == 0.0: # dephasing inducing noise
    return gamma1
  else: # relaxation inducing noise
    return gamma1 / 2 * (w / (2 * np.pi)) * (w > 0.0)


R, ekets = bloch_redfield_tensor(H, [[sigmax(), ohmic_spectrum]])

print(R)
```

5.3 Variable

Les varaibles sont des objets auxquels vous attribuez une valeur que celle-ci soit un nombre entier, un nombre réel, un booléen...

```python
a = 5.0
```

```python
b = "Hello World!"
```

```
a = b
```

Dans l'exemple précédent, nous avons définit deux variables `a` et `b`. La première est un nombre réel et la seconde une chaine de caractères. `Python` est flexible, nous n'avons pas eu besoin de préciser que `a` serait un réel. Nous lui avons directement assigné un nombre réel avec l'opérateur "=" et Python à fait le reste. Même principe pour `b`. Vous remarquerez que Python est tellement flexible, que nous avons pu "*recycler*" la variable réelle `a` en une variable chaine de caractères en lui assignant simplement la valeur de `b`.

5.3.1 Types primaires

Il existe quatre types primaires de variables : les booléens (`bool`), les entiers (`int`), les nombres réels (`float`) et les chaînes de caratères (`string`).

Ligne de commande

```
>>> v = True
>>> type(v)
bool
>>> a = 5
>>> type(a)
int
>>> b = 5.0
>>> type(b)
float
>>> c = "Hello world!"
str
```

Les nombres flottants Les nombres réels sont représentés de façon approximative car il ne peuvent pas être stocké avec une précision infinie. Ils sont donc représenté sous un forme de notation scientifique en base dite *flottante* car la position de la virgule dépend de la puissance de deux considérée. On a donc

$$\text{nombre} = \text{signe} \times \text{mantisse} \times 2^{\text{exposant}} \tag{1}$$

où la *mantisse* est l'ensemble de chiffres significatifs retenus pour approximer le nombre. Par exemple, le nombre 0.001_2 sera représenté comme

$$0.001_2 = +1 \times 0.1 \times 2^{-2}\ . \tag{2}$$

L'imprecision inérente à cette représentation expose à une erreur dite de *représentation*. Prenons le nombre 1/10, son expression en binaire n'est pas finie et est donc sa représentation est approximative. Bbeaucoup de nombres décimaux qui partagent une même approximation en fraction binaire. Par exemple, 0.1, 0.10000000000000001 et $0.1000000000000000055511151231257827021181583404541015625$ ont tous pour approximation $3602879701896397 \times 2^{-55}$. Une autre conséquence du fait que 0,1 n'est pas exactement stocké 1/10 est que la somme de trois valeurs de 0,1 ne donne pas 0,3 non plus :

Ligne de commande

```
>>> .1 + .1 + .1 == .3
False
```

Ou encore :

Ligne de commande

```
>>> x = 1000.2
>>> b = x - 1000.0
>>> b
0.20000000000004547
```

Un autre type d'erreur associé à la précision finie de la représentation des nombre réél est l'*erreur d'arrondi*. Tous les nombres réels, même ceux représentés exactement sur la machine, sont soumis à cette erreur qui advient lorsque des opérations arithmétiques sont effectuées. Par exemple, si vous ajouter un grand nombre et un petit nombre ensemble. En effet, il existe un nombre flottant minimal qui lorsqu'ajouté au nombre flottant 1.0 donne un résultat différent de 1.0; il est nommé *préision machine* et est noté ϵ. Globalement, ϵ est la précision fractionnelle à laquelle les nombres flottants sont représentés et correspond à un changement de un sur le bit le moins signifiant de la mantisse. Les opérations arithmétiques sur les flottants introduisent une erreur fractionelle d'au moins ϵ en valeur absolue. Si N opérations arithmétiques sont effectuées sur un nombre flottant, l'erreur d'arrondi la plus favorable que l'on puissent obtenir est donc $\sqrt{N}\epsilon$.
Cependant les régularités de certaines d'une méthode de calcul peuvent mener les erreurs d'arrondi à s'accumuler dans une direction préféentielle et à être de l'ordre de $N\epsilon$. Ce cas de figure peut mener à une *erreur de stabilité* où une méhode de calcul valide accumule tant d'erreurs d'arrondi que la valeur calculée est très éloignée de la valeur attendue. Un exemple d'une telle erreur est le calcul de la puissance n du nombre d'or $\phi = \frac{\sqrt{5}-1}{2}$ par récurence :

$$\phi^n = \phi^{n-2} - \phi^{n-1} \,. \tag{3}$$

Après seulement une quinzaine d'itérations, l'erreur relative sur la valeur de ϕ^n calculée par récurence par rapport au calcul direct est de l'ordre de 10^6 comme le montre la figure 1.

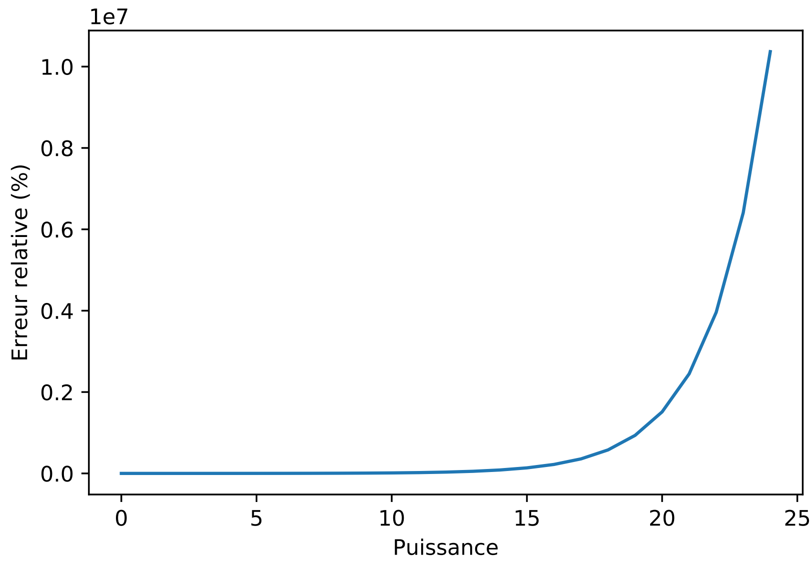


Figure 1: Erreur relative sur la puissance n du nombre d'or par récurrence par rapport au calcul direct.

Chaînes de caractères

Ligne de commande

```
>>> # Les chaînes (ou strings) sont créées avec " ou '
>>> "Ceci est une chaine"
"Ceci est une chaine"
>>> 'Ceci est une chaine aussi.'
'Ceci est une chaine aussi.'
>>> # On peut concatener les chaînes avec +
>>> "Hello " + "world!"
"Hello world!"
>>> # On peut traîter une chaîne comme une liste de caractères
>>> "This is a string"[0]
'T'
>>> # .format peut être utilisé pour formatter des chaînes, comme ceci:
>>> "{} peuvent etre {}".format("Les chaînes", "interpolées")
"Les chaînes peuvent etre interpolées"
>>> # On peut aussi réutiliser le même argument pour gagner du temps.
>>> "{0} be nimble, {0} be quick, {0} jump over the {1}".format("graham", "candle stick")
"graham be nimble, graham be quick, graham jump over the candle stick"
>>> # On peut aussi utiliser des mots clés pour éviter de devoir compter.
>>> "{name} wants to eat {food}".format(name="Bob", food="lasagna")
"Bob wants to eat lasagna"
```

5.3.2 Listes, tuples et dictionnaires

Listes Les listes sont des collections d'objets qui peuvent être différent types. Elles sont délimités par des crochets "[" et les éléments sont séparés par des virgules ",".

Ligne de commande

```
>>> # Les listes permettent de stocker des séquences
>>> li = []
>>> # On peut initialiser une liste pré-remplie
>>> other_li = [4, 5, 6]
>>> # On ajoute des objets à la fin d'une liste avec .append
>>> li.append(1)    # li vaut maintenant [1]
>>> li.append(2)    # li vaut maintenant [1, 2]
>>> li.append(4)    # li vaut maintenant [1, 2, 4]
>>> li.append(3)    # li vaut maintenant [1, 2, 4, 3]
>>> # On enlève le dernier élément avec .pop
>>> li.pop() # li vaut maintenant [1, 2, 4]
3
>>> # Et on le remet
>>> li.append(3)    # li vaut de nouveau [1, 2, 4, 3]
>>> # On peut accéder aux éléments de la liste par un indice.
>>> # Les indices commencent à 0 :
>>> li[0]
1
>>> # Accès au dernier élément :
>>> li[-1]
3
>>> # Accéder à un élément en dehors des limites lève une IndexError
>>> li[4]
IndexError: list index out of range
>>> # On peut accéder à une intervalle avec la syntaxe "slice" (intervalle)
>>> li[1:3]
[2, 4]
>>> # Omettre les deux premiers éléments
>>> li[2:]
[4, 3]
>>> # Prendre les trois premiers
>>> li[:3]
[1, 2, 4]
>>> # Sélectionner un élément sur deux
>>> li[::2]
[1, 4]
>>> # Avoir une copie de la liste à l'envers
>>> li[::-1]
[3, 4, 2, 1]
>>> # Pour des "slices" plus élaborées : li[debut:fin:pas]
>>> # Enlever des éléments arbitrairement d'une liste
>>> del li[2]
>>> li
[1, 2, 3]
>>> # On peut additionner des listes : les valeurs de li et other_li ne sont pas modifiées
>>> li + other_li
[1, 2, 3, 4, 5, 6]
>>> # Concaténer des listes avec "extend()"
>>> li.extend(other_li)
[1, 2, 3, 4, 5, 6]
>>> # Vérifier la présence d'un objet dans une liste avec "in"
>>> 1 in li
True
>>> # Examiner la longueur avec "len()"
>>> len(li)
6
```

Il est possible d'imbriquer des listes (c.-à-d. créer des listes contenant d'autres listes). Par exemple :

Ligne de commande

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Tuple Les tuples sont similaires aux listes à la différence que leurs éléments sont immuables. Ils consistent en différentes valeurs séparées par des virgules.

Ligne de commande

```
>>> t = (12345, 54321, 'hello!')
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Les tuples peuvent être imbriquées
... u = (t, (1, 2, 3, 4, 5))
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Les tuples sont immuables
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # Mais ils peuvent contenir des objets muables
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Les tuples vides sont construits par une paire de parenthèses sans élément et un tuple avec un unique élément est construit en faisant suivre celui-ci par une virgule (il n'est pas suffisant de placer cette valeur entre parenthèses).

Ligne de commande

```
>>> empty = ()
>>> singleton = 'hello',
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```


Dictionnaire Les dictionnaires sont un type natif en `Python` qui contrairement aux listes n'est pas indexé par un indice mais par une clef qui peut être une chaîne de caractères, un nombre ou un objet de n'importe quel type immuable. Les dictionnaires sont des ensembles de paires clé: valeur, les clés devant être uniques (au sein d'un dictionnaire). Une paire d'accolades crée un dictionnaire vide : .

Ligne de commande

```
>>> tel = {'graham': 4098, 'john': 4139}
>>> tel['terry'] = 4127
>>> tel
{'graham': 4098, 'john': 4139, 'terry': 4127}
>>> tel['graham']
4098
>>> del tel['john']
>>> tel['eric'] = 4127
>>> tel
{'graham': 4098, 'terry': 4127, 'eric': 4127}
>>> list(tel)
['graham', 'terry', 'eric']
>>> sorted(tel)
['terry', 'eric', 'graham']
>>> 'terry' in tel
True
>>> 'graham' not in tel
False
```

5.4 Conditions

Une action peut être conditionnée grâce à l'instruction `if`. Si la proposition suivant `if` est vrai, alors une certaine opération sera réalisée.

Ligne de commande

```
>>> x = int(input("Entrez votre âge : "))
Entrez votre âge : -6
>>> if x < 0:
...     print('Negatif, vous n\'êtes pas coopératif')
... elif x == 0:
...     print('Un peu jeune pour la prépa...')
... else:
...     print('x')
...
Negatif, vous n'êtes pas coopératif
```

Comme vous le voyez, les conditions peuvent s'enchaîner les unes après les autres.

Exercice 3 **Nombre pair**

Rédigez une condition qui teste si un nombre est pair ou impair.

5.5 Boucle

5.5.1 For

La boucle `for` est une instruction qui est réalisée pour chaque élément d'une liste (ou plus généralement d'un itérable).

Ligne de commande

```
>>> liste = [0, 1, 2, 3, 4]
>>> for i in liste:
...     print(i)
...
0
1
2
3
4
```

La fonction `range(n)` permet de créer une suite arithmétique de 0 à `n-1` inclu. La boucle précédente peut donc se réécrire :

Ligne de commande

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Exercice 4 **Liste des carrés**

Écrire une boucle `for` créant une liste de carrés des entiers de 1 à 10.

Il existe un façon très concise de créer une liste sans passer directement par une boucle `for` où l'instruction est donnée directement à l'intérieur de la liste. On la nomme *liste en compréhension*. Au lieu de la boucle `for` suivante :

Ligne de commande

```
>>> carres = []
>>> for x in range(10):
...      carres.append(x**2)
...
>>> carres
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On écrira

Ligne de commande

```
carres = [x**2 for x in range(10)]
```

Exercice 5 Tri d'une liste

Écrire un script qui tri une liste par ordre d'éléments décroissant.

5.5.2 While

L'instruction `while` bouclera tant qu'une condition reste vraie.

Ligne de commande

```
>>> i = 0
>>> while i < 10:
...     i += 1
...     print(i)
...
1
2
3
4
5
6
7
8
9
10
```

Exercice 6 Le crible d'Ératosthène

Le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N. L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier (autres que lui-même).

Codez un script qui donne tous les nombres premiers inférieurs à N

Exercice 7 Série de Fibonacci paire *(À la maison)*

La suite de Fibonacci est générée par la somme des deux précédents termes. Les deux premiers termes sont 0 et 1, et les 10 termes suivants sont donc : 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.

En considérant les termes de la suite de Fibonacci dont la valeur ne dépasse pas quatre million, trouvez la somme des termes pairs.

Exercice 8 Le plus grand palindrome *(À la maison)*

Un nombre palindrome se lit de la même façon à l'endroit et à l'envers. Le plus grand palindrome obtenu par un produit de deux nombres à deux chiffres est $9009 = 91 \times 99$.

Trouvez le plus grand palindrome obtenu à partir du produit de deux nombres à trois chiffres.

5.6 Fonction

Une fonction se défini de la façon suivante

```python
def nom_de_la_fonction(argument1, argument2):
    """
    Commentaire de documentation de la fonction
    """
    Instructions...
    resultat = ... # On définit la varibale résultat qui sera renvoyée par la fonction
    return resultat # si la fonction renvoie un objet
```

On remarquera qu'il n'est pas nécessaire de définir le type des arguments de la fonction. La fonction n'est pas délimitépar des parenthèses ou des crochets mais simplement par *indentation*. Par exemple, si l'on souhaite une fonction qui calcule la somme de deux nombres :

```python
def somme(a, b):
    """
    somme(a, b) renvoie a+b
    """
    return a + b
```

Un exemple plus complexe : on calcule la moyenne d'une liste

```python
def moyenne(liste):
    """
    Calcule la moyenne des éléments de liste
    """
    S = 0 # Accumulateur de somme
    L = len(liste) # Longueur de la liste = nombre d'éléments
    for i in range(L):
        S += liste[i]
    S = S/L
    return S
```

Exercice 9 **Méthode d'Euler**

La méthode d'Euler est une méthode numérique élémentaire de résolution d'équations différentielles du premier ordre. Considérons l'équation différentielle portant sur la fonction $x(t)$

$$\frac{\mathrm{d}}{\mathrm{d}\,t}x(t) = f(x(t),t)\ ,$$

avec la condition initiale $x(t_0) = x_0$. La méthode d'Euler permet d'obtenir un ensemble de valeur x_n pour chaque instant $t_n = t_0 + n\delta t$ (avec n un entier dans $[0, N]$, t_0 le temps initial et δt le pas de temps séparant ces instants) grâce à l'approximation suivante :

$$x_n = x_{n-1} + f(x_{n-1}, t_{n-1})\delta t\ .$$

Écrivez une fonction implémentant la méthode d'Euler pour $f(x(t), t) = x(t)$, prenant comme arguments la condition initiale, le pas d'intégration et le temps final $t_f \overset{\text{def.}}{=} t_0 + N\delta t$, renvoyant une liste de valeurs intégrées.
Commentez le résultat en fonction de δt.
Bonus : estimez l'erreur commise par la méthode d'Euler.

Exercice 10 **Les suites de Syracuse**

En mathématiques, on appelle suite de Syracuse une suite d'entiers naturels définie de la manière suivante. On part d'un nombre entier strictement positif :

- s'il est pair, on le divise par 2,
- s'il est impair, on le multiplie par 3 et on ajoute 1.

En répétant l'opération, on obtient une suite d'entiers strictement positifs dont chacun ne dépend que de son prédécesseur. Par exemple, à partir de 14, on construit la suite des nombres : 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2...C'est ce qu'on appelle la suite de Syracuse du nombre 14. Après que le nombre 1 a été atteint, la suite des valeurs (1,4,2,1,4,2...) se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial. A priori, il serait possible que la suite de Syracuse de certaines valeurs de départ n'atteigne jamais la valeur 1, soit qu'elle aboutisse à un cycle différent du cycle trivial, soit qu'elle diverge. Or, on n'a jamais trouvé d'exemple de suite qui n'aboutisse pas à 1. La conjecture de Syracuse est l'hypothèse selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1. On note u_n les éléments de cette suite et u_0 l'entier qui l'initialise.

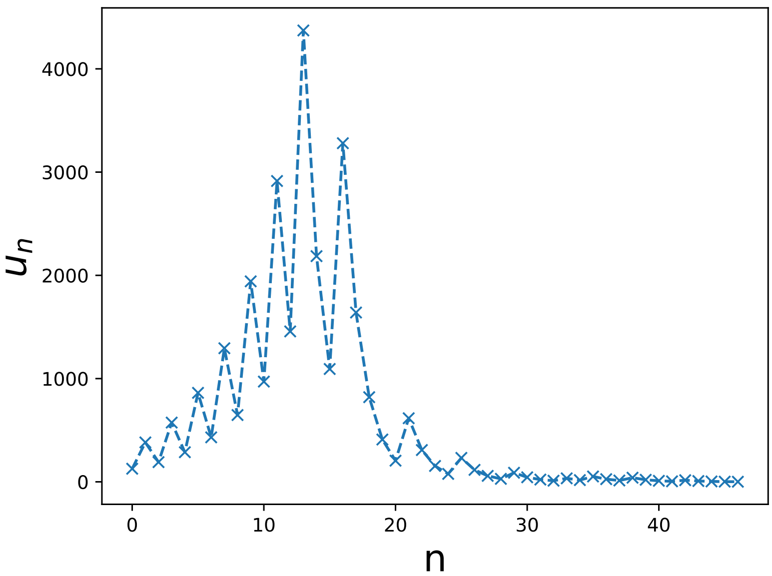


Figure 2: Suite de Syracuse de $u_0 = 127$.

On définit alors :

- le *temps de vol* : c'est le plus petit indice n tel que $u_n = 1$. (e.g. il est de 17 pour la suite de Syracuse 15 et de 46 pour la suite de Syracuse 127),
- le *temps de vol en altitude* : c'est le plus petit indice n tel que $u_n + 1 < u_0$ (e.g il est de 10 pour la suite de Syracuse 15 et de 23 pour la suite de Syracuse 127),
- l'*altitude maximale* : c'est la valeur maximale de la suite. (e.g. elle est de 160 pour la suite de Syracuse 15 et de 4 372 pour la suite de Syracuse 127).

(1) Rédigez une fonction qui calcule le temps de vol de la suite et prenant pour argument un entier correspondant à la valeur initiale de la suite u_0.

(2) Rédigez une fonction qui calcule l'altitude maximale de la suite et prenant pour argument un entier correspondant à la valeur initiale de la suite u_0.

Exercice 10 **Les suites de Syracuse** (Suite)

(3) Rédigez une fonction qui calcule le temps de vol en altitude de la suite et prenant pour argument un entier correspondant à la valeur initiale de la suite u_0.

(4) Rédigez une fonction qui vérifie si la conjecture est vérifiée pour tous les entiers inférieurs à N. Qu'obtient-on pour $N = 1000000$?

(5) Quelle est l'altitude maximale atteinte pour $u_0 \in [2, N]$ et quel est le plus petit u_0 y menant ?

(6) Quel est le plus long temps de vol en altitude pour $u_0 \in [2, N]$ et quel est le plus petit u_0 y menant ?

5.7 Zen of Python

Tapez dans l'interpréteur :

```
>>> import this
```

Et le Zen de Python s'affichera. Il s'agit d'aphorisme donnant des recommandation de haut niveau sur la façon de rédiger un programme.

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one– and preferably only one –obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!

Plus généralement des conseils de style dans la rédaction de codes ont été édités par la communauté des programmeurs en Python : PEP 8.

5.8 Complexité algorithmique

Il existe souvent plusieurs façons de résoudre un problème avec différents algorithmes, néanmoins toutes les méthodes ne sont pas équivalentes. Certaines sont plus rapides que d'autres, certaines sont plus gourmande en espace mémoire que d'autres. Le temps de calcul et la mémoire disponnible étant limité, il convient de tenter de minimiser leur utilisation par nos algorithme. Pour cela, il faut être capable de les quantifier à partir de l'algorithme avant même son implémentation dans un langage de programmation particulier. Cette quantification s'appelle la *complexité en temps* ou bien la *complexité en espace*. La complexité en temps est exprimée sous la forme d'un ordre de grandeur du nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaire au fonctionnement de l'algorithme dans le pire des cas pour une entrée de taille n. La complexité en espace est l'ordre de grandeur de l'espace mémoire minimal utilisé par l'algorithme dans le pire des cas.

Nom	Complexité en temps	Temps de calcul ($n \sim 10^3$)	Exemple d'algorithmes
Constant	O(1)	10 ns	Accès à un élément d'une liste
Logarithmique	O(log(n))	30 ns	Recherche dichotomique
Linéaire	O(n)	10 µs	Parcours d'une liste
Linéarithmique	O(n log(n))	30 µs log n	Tri fusion
Quadratique	$O(n^2)$	10 ms	Parcours matrice
Cubique	$O(n^3)$	10 s	Multiplication matricielle
Exponentiel	$2^{O(n)}$	$\sim \infty$	Problème du voyageur de commerce

Table 1: Exemples de classes de complexité en temps.

Exercice 11 **Classe de complexité**

À quelles classes de complexité (en temps) appartiennent les algorithmes suivant ?

```python
def permutation(A,i,j):
    """ Permute les éléments i et j de la liste A"""
    temp = A[i]
    A[i] = A[j]
    A[j] = A[i]
```

```python
def tri(A,n):
    """ Tri par ordre croissant les éléments de la liste A de longueur n"""
    for i in reverse(range(2,n)):
        for j in range(i):
            if A[j]>A[j+1]:
                permutation(A,j,j+1)
```

```python
def factorielle(n):
    """ Renvoie n! """
    R = 1
    i = 1
    while i<n:
        i += 1
        R = R*i

    return R
```

6 Les librairies

Pour éviter d'avoir des scripts trop volumineux, donc illisibles, et de devoir réinventer la roue à chaque fois qu'on en a besoin, il est possible d'importer des fonctions d'un script à un autre. Par exemple, si vous avez enregistré votre fonction `crible(N)` renvoyant l'ensemble des nombres premiers inférieurs à N dans un script nommé `eratosthene.py`, vous pouvez l'importer dans un nouveau script avec l'instruction suivante

```python
from eratosthene import crible
```

Vous pouvez ensuite l'utiliser directement en appelant `crible(un_entier)` dans le script. Si `eratosthene.py` contient plusieurs fonctions, vous pouvez toutes les importer avec l'instruction

```python
import eratosthene
```

Dans ce cas il faudra taper `eratosthene.crible(un_entier)` pour appeler la fonction.
Vous pouvez racourcir l'appelle d'une fonction importée en donnant un surnom à sa librairie d'origine, par exemple

```python
import eratosthene as er
```

Il vous suffira de taper taper `er.crible(un_entier)` pour appeler la fonction.

6.1 Première librairie : NumPy

`NumPy` pour *Numerical Python* est une librairie dédiée au calcul scientifique. Il contient un grand nombre de fonctions mathématiques de base comme $\cos, \sin, \exp \ldots$ et de fonctions pour le calcul matriciel. On l'importe avec l'instruction

```python
import numpy as np
```

NumPy possède un large éventail de manières efficaces et rapides de créer des tableaux et de manipuler les données numériques qu'ils contiennent. Contrairement aux listes de Python, tous les éléments d'un tableau de NumPy doivent êre du même type. Un tableau est plus rapide et plus compact qu'une liste, utilise moins de mémoire et est pratique d'utilisation.

Ligne de commande

```python
>>> a = np.array( [20,30,40,50] ) # transforme la liste en un tableau
>>> b = np.arange( 4 ) # créé un tableau de quatre éléments
>>> b
array([0, 1, 2, 3])
>>> c = a-b # soustraction élément par élément
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True,  True, False, False])
```

Des fonctions permettent de créer des tableaux avec des valeurs prédéfinis.

Ligne de commande

```
>>> np.zeros((3, 4)) # Initialise une matrice 3x4 ne contenant que des 0
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> # Initialise un tableau 2x3x4 ne contenant que des 1
>>> np.ones( (2,3,4), dtype=np.int16 )
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

Des fonctions intégrés aux tableaux permettent de connaître leurs caractéristiques

Ligne de commande

```
>>> a = array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
```

Vous pouvez en apprendre plus avec la documentation et le guide pour débutants.

Exercice 12 Le jeu de la vie de Conway

Le jeu de la vie est un automate cellulaire, un modèle où chaque état conduit mécaniquement à l'état suivant à partir de règles pré-établies. Le jeu se déroule sur une grille à deux dimensions dont les cases — qu'on appelle des «cellules», par analogie avec les cellules vivantes — peuvent prendre deux états distincts : «vivante» ou «morte».
Une cellule possède huit voisins, qui sont les cellules adjacentes horizontalement, verticalement et diagonalement. À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

- une cellule morte possédant exactement trois voisines vivantes devient vivante
- une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt

Implémentez le jeu de la vie.
Vous aurez besoin de la fonction `sleep(nombre_de_seconde)` de la librairie `time` qui met Python en pause pendant `nombre_de_seconde`.

Exercice 13 Multiplication Matricielle

Rédigez une fonction qui prend en argument deux matrices A et B et renvoie la matrice $C = A \cdot B$. Cette fonction vérifiera toutes les conditions nécessaires pour que la multiplication soit définie avant de la réaliser.

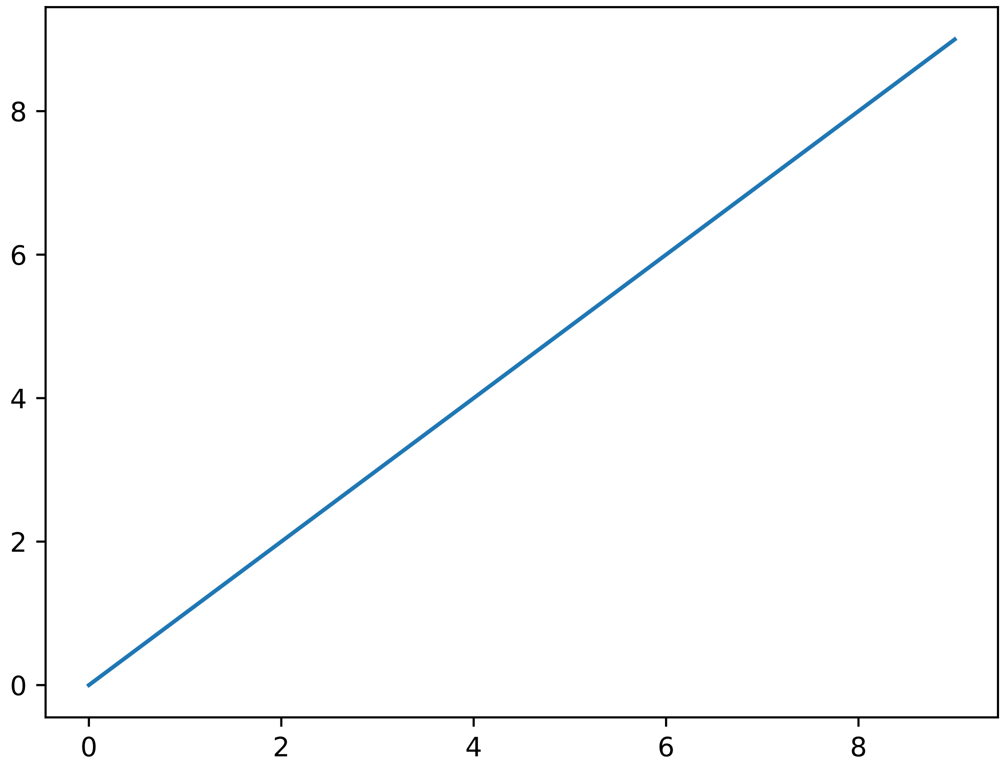
6.2 Graphiques avec PyPlot

La librairie `PyPlot` contient un ensemble de fonctions permettant d'afficher des courbes, des histogrammes, etc.

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

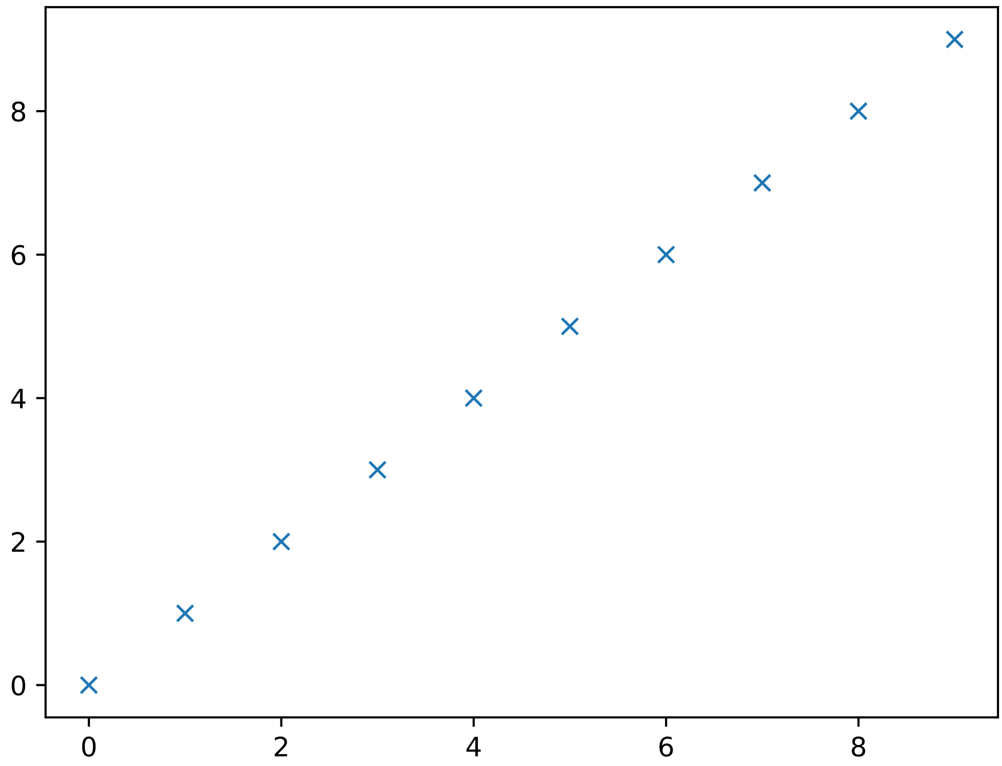
Ligne de commande

```
>>> y = np.arange(10)
>>> plt.plot(y)
```



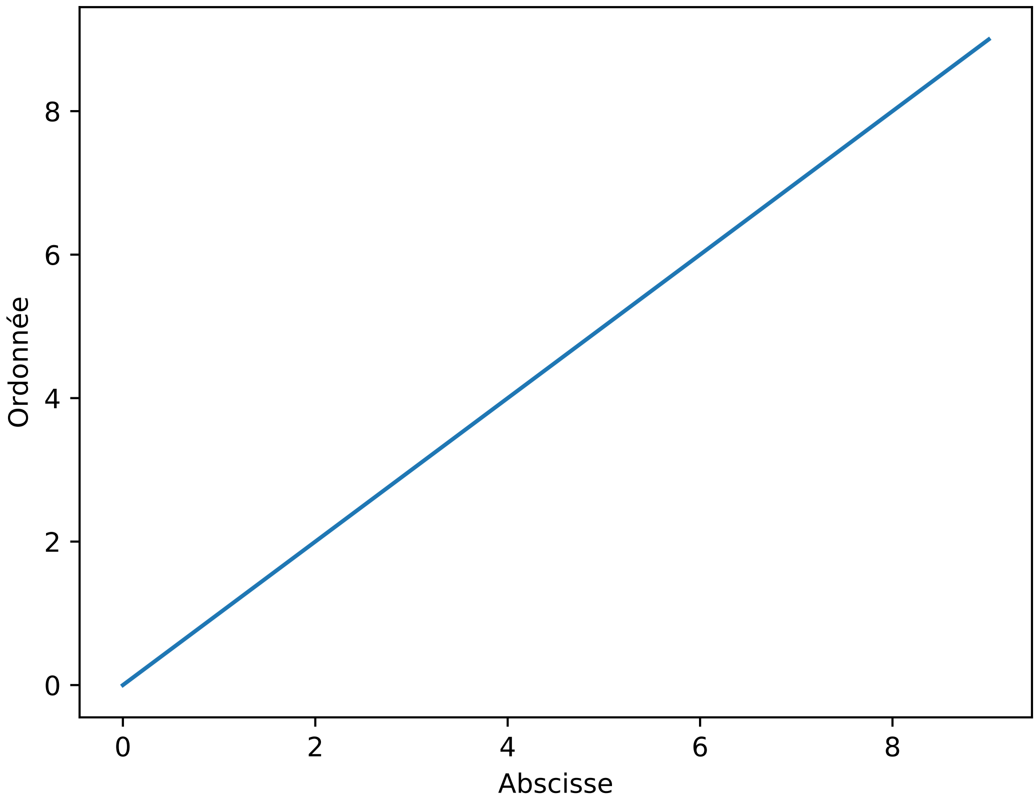
Ligne de commande

```
>>> y = np.arange(10)
>>> plt.plot(y, 'x')
```



Ligne de commande

```
>>> y = np.arange(10)
>>> plt.plot(y)
>>> plt.xlabel("Abscisse")
>>> plt.xlabel("Ordonnée")
```



6.3 Application : Le théorème central limite

Soit $X_1, X_2 \ldots$ une suite de variables aléatoires réelles définies sur le même espace de probabilité, indépendantes et identiquement distribuées suivant la même loi. Supposons que l'espérance μ et l'écart-type σ de existent et soient finis.
Considérons la somme $S_n = X_1 + X_2 + \ldots + X_n$. Alors, l'espérance de S_n est $n\mu$ et son écart-type vaut $\sigma\sqrt{n}$.

Nous nous propsons de vérifier cela numériquement et d'afficher le résultat. Premièrement nous devons réliser un ensemble de M tirages aléatoires de nos n variables aléatoires X_i. Pour cela nous alons utiliser la fonction `rand()` de la librairie `numpy.random`. D'après sa page dans la documentation de NumPy :

`random.rand(d0, d1, ..., dn)`
Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

Parameters: d0, d1, . . . , dn : int, optional
The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns: out: ndarray, shape (d0, d1, ..., dn)
Random values.

Ligne de commande

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

Nous pouvons donc créer une matrice où chaque ligne correspond à une des n varible aléatoire X_i, chaque colonne à un des M tirages. Ensuite, nous construisons S_n en sommant tous les éléments d'une colonne. Nous obtiendrons donc un vecteurs contenant M tirages de la variables aléatoire S_n, que nous pourrons donc représenter comme un histogramme pour avoir la distribution de probabilité empirique de S_n.

```python
import numpy as np
import matplotlib.pyplot as plt

n = 10000 # nombre de variables aléatoires distribuées uniformément sur [0,1[
M = 20000 # nombre de tirage de ces variables

tirages = np.random.rand(n,M)
variable_somme = np.sum(tirages,0) # on somme les colonnes pour former M tirages d'une nouvelle VA

plt.hist(variable_somme, bins=100) # On affiche la distribution de cette variable
plt.xlabel("S_n")
plt.ylabel("Occurences")
```

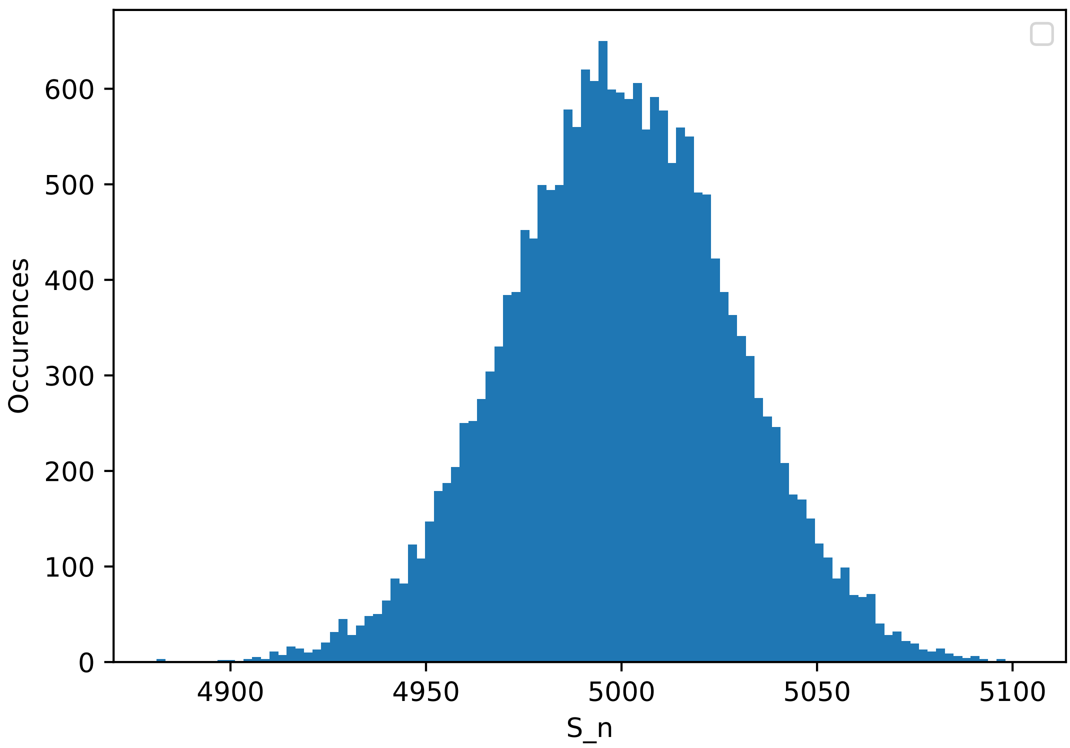


Figure 3: Distribution empirique de $S_n = X_1 + X_2 + \ldots + X_n$.

L'allure obtenue est bien une gaussienne. On peut vérifier que sa moyenne m et son écart-type s concordent avec le théorème central limite. On s'attend à $m = n\mu = 10000 \times \frac{1}{2} = 5000$ et $s^2 = 10000 \times \frac{1}{12} \approx 833.33$.

Ligne de commande

```
>>> variable_somme.mean()
4999.90037391763
>>> variable_somme.var()
841.7700137855464
```

On peut aussi comparer l'histogramme à la distribution attendue

```python
import numpy as np
import matplotlib.pyplot as plt

n = 10000 # nombre de variables aléatoires distribuées uniformément sur [0,1[
M = 20000 # nombre de tirage de ces variables

tirages = np.random.rand(n,M)
variable_somme = np.sum(tirages,0) # on somme les colonnes pour former M tirages d'une nouvelle VA

plt.hist(variable_somme, bins=100) # On affiche la distribution de cette variable
plt.xlabel("S_n")
plt.ylabel("Occurences")

mu = 5000 # moyenne de la Gaussienne
sigma = np.sqrt(833.33) # écart-type

f = lambda x : np.exp(-(x-mu)**2/(2*sigma**2)) # délaration la gaussienne par fonction anonyme

domaine = range(4900,5100)

g = [600*f(x) for x in domaine]

plt.plot(domaine, g, 'r', label="Gaussienne : m = 5 000, s^2 = 833.33")
```

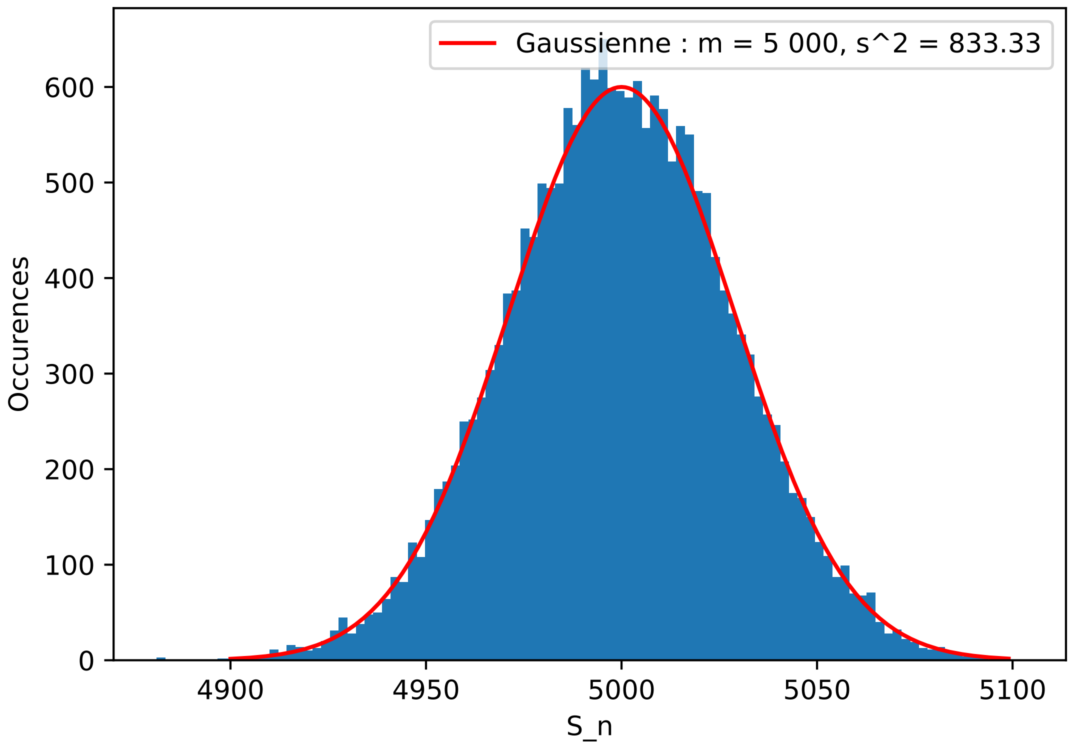


Figure 4: Distribution empirique de $S_n = X_1 + X_2 + \ldots + X_n$. En rouge, la distribution théoriquement attendue

Exercice 14 **Diagramme de Hertzsprung-Russell**

En astronomie, le diagramme de Hertzsprung-Russell, en abrégé diagramme HR, est un graphique dans lequel est indiquée la luminosité d'un ensemble d'étoiles en fonction de leur température effective. Ce type de diagramme a permis d'étudier les populations d'étoiles et d'établir la théorie de l'évolution stellaire. Un diagramme de Hertzsprung-Russell représente soit la luminosité intrinsèque d'une étoile en fonction de sa température (utilisée par les théoriciens), soit la magnitude absolue en fonction de l'indice de couleur (ce qui découle immédiatement de données photométriques). Un diagramme de Hertzsprung-Russell est toujours présenté de la manière suivante :

- la luminosité est en ordonnée, le plus brillant étant en haut
- la température effective, ou l'indice de couleur, est en abscisse, le plus chaud étant à gauche

En utilisant les données de `stars.txt` tracez un diagramme HR tel que définit ci-dessus. Vous aurez besoin de la fonction `loadtxt` de la librairie NumPy qui prend comme argument une chaîne de caractères donnant le chemin du fichier `stars.txt` et le type des données que vous souhaitez charger, et retourne un tableau contenant les données.

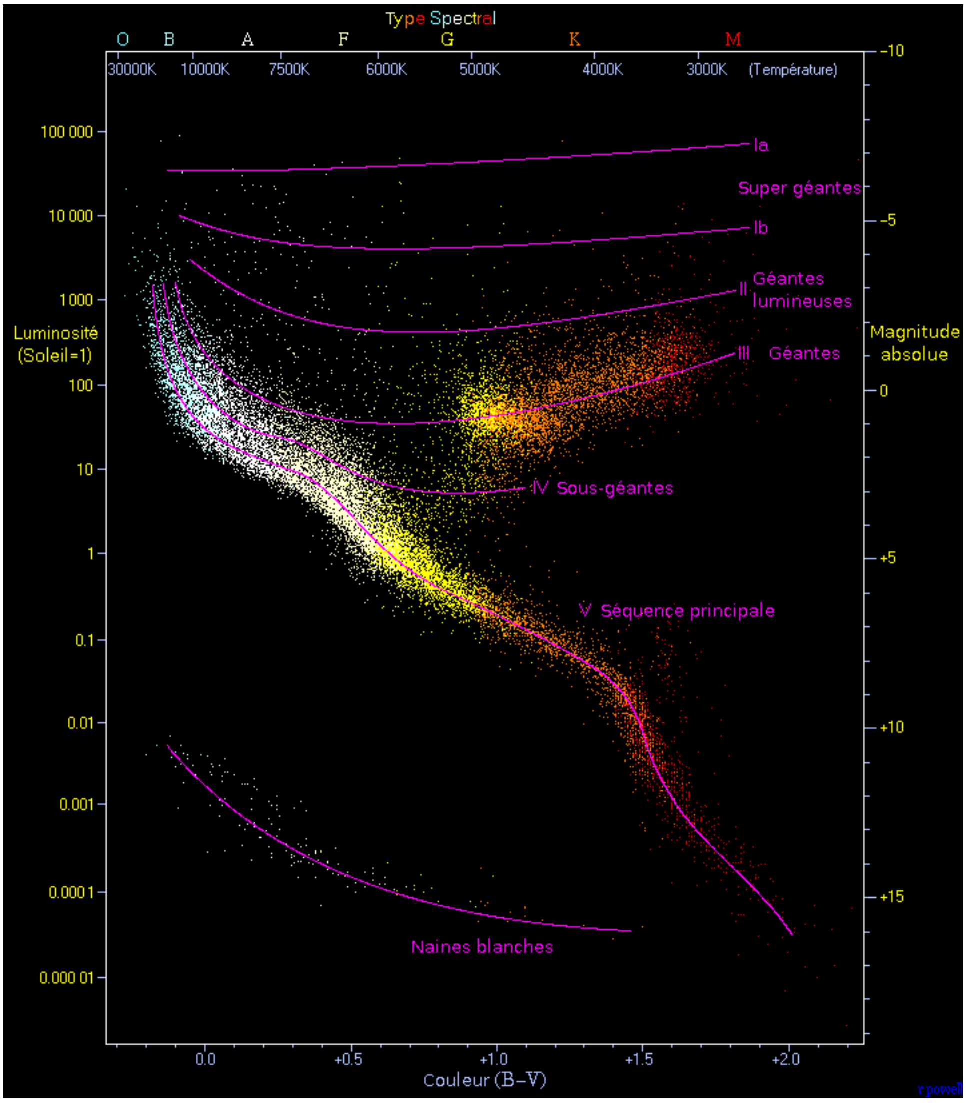


Figure 5: Diagramme de Hertzsprung-Russell créé par Richard Powell, à partir de 22 000 étoiles du catalogue Hipparcos et 1 000 étoiles du catalogue Gliese. Le Soleil se trouve sur la séquence principale et a pour luminosité 1 (magnitude absolue 4,8) et température 5 780 K (type spectral G2).

Exercice 15 Les moindres carrés

La méthode des moindres carrés est une méhode d'ajustement linéaire de données expérimentales $\{x_i, y_i\}_{i=1...N}$. La pente de l'ajustement est donnée par $\hat{a} = \mathrm{cov}(x,y)/\mathrm{var}(x)$ et $\hat{b} = \bar{y} - \hat{a}\bar{x}$, où

$$\mathrm{cov}(x,y) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}) \ ,$$
$$\mathrm{var}(x) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 \ ,$$
$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \ ,$$
$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i \ .$$

Implémentez la méhode des moindres carrés et testez la sur les données générées par le code suivant.

```
import numpy as np
from random import gauss

a = 100*np.random.rand()
b = 10*np.random.rand()

x = np.arange(-5,5) # Intensité en mA
y = np.sort(np.array([a*gauss(0,1)+b for i in range(10)])) # Tension en V
```

Bonus :

- que fait le code ci-dessus ?
- quelle est la valeur de la résistance ?

Exercice 16 Lignes de champs d'un dipôle électrostatique *(À la maison)*

Deux charges q et $-q$ sont placées aux positions $\vec{r}_+ = \frac{a}{2}\vec{u}_y$ et $\vec{r}_- = -\frac{a}{2}\vec{u}_y$ dans le planet forment un dipôle électrostatique $\vec{d} = q(\vec{r}_+ - \vec{r}_-)$. Les composantes du champ électrostatique sont

$$E_x = \frac{q}{4\pi\epsilon_0}\left[\frac{x}{(x^2+(y-\frac{a}{2})^2)^{\frac{3}{2}}} - \frac{x}{(x^2+(y+\frac{a}{2})^2)^{\frac{3}{2}}}\right]$$
$$E_y = \frac{q}{4\pi\epsilon_0}\left[\frac{y-\frac{a}{2}}{(x^2+(y-\frac{a}{2})^2)^{\frac{3}{2}}} - \frac{y+\frac{a}{2}}{(x^2+(y+\frac{a}{2})^2)^{\frac{3}{2}}}\right]$$

Rédigez un script calculant et traçant les lignes de champ de ce dipôle.

Exercice 17 **Percolation** *(À la maison)*

Le café peut être obtenu par percolation, en utilisant un percolateur où l'eau se fraye un chemin entre les particules de café. C'est la méthode utilisée dans la préparation de l'*espresso*. La percolation (du latin *percolare*, « filtrer », « passer au travers ») désigne communément le passage d'un fluide à travers un milieu plus ou moins perméable. Plus généalement, en physique et en mathématiques ce terme désigne une transition d'un état vers un autre avec un phénomène de seuil associé à la transmission d'une « information » par le biais d'un réseau de sites et de liens qui peuvent, selon leur état, relayer ou non l'information aux sites voisins. Cette théorie est régulièrement utilisée pour la modélisation de phénomènes naturels, comme les incendies.

Figure 6: Les cafés *espresso* sont obtenus par percolation de l'eau à travers le café moulu.

Nous allons considérer une grille carré de n cases de côté. Ces cases peuvent être libre avec une probabilité p ou obstruée. La percolation est possible s'il existe un chemin de cases libres joignant la ligne supérieure de la grille à la ligne inférieure.

La grille sera repréentée par un tableau. Les cases libres par la valeur 1, les cases obstruées par la valeur 0 et les cases emplies de café par la valeur $1/2$.
Pour visualiser la grille, nous utiliserons la fonction
`matshow(grille, cmap=ListedColormap(['black','brown','white']))`
de la librairie `matplotlib.pyplot`. On importera `ListedColormap` de `matplotlib.colors`

(1) Rédigez une fonction générant une grille de taille $n \times n$ avec une probabilité $1 - p$ d'obstruction d'une case.

(2) Rédigez une fonction remplissant la grille de café en commençant par la ligne supérieure. Une fois les cases libres de la lignes supérieure remplies, les cases libres adjacentes sont remplies à leur tour. Cela est répété jusqu'à ce qu'il n'y ait plus de cases libres adjacentes à remplir.

(3) Rédigez une fonction qui génère une grille et teste si la percolation a lieu ou non.

(4) Rédiger un script qui calcule la distribution de probabilité $P(p)$ que la percolation ait lieu en fonction du paramètre p.

7 Charger des données

Python est en mesure de lire et d'écrire des informations dans des fichiers comme nous l'avons vu lors de l'exercice du diagramme HR. On fonction du type de données avec lesquelles vous travaillez, plusieurs possibilités s'offrent à vous.

7.1 Lire et écrire des chaînes de caractères

La fonction `open(chemin, mode)` permettant de lire et d'écrire dans un fichier texte est native. L'argument `chemin` est une chaîne de caractères contenant le chemin du fichier et l'argument `mode` détermine si celui-ci est accédé en *lecture* ou en *écriture* :

- `r` permet d'accéder en lecture,
- `w` permet d'accéder en écriture en écrasant le fichier,
- `a` permet d'accéder en écriture en ajoutant au fichier.

Pour lire le contenu du fichier, il suffit d'écrire

```
with open('chemin\_du\_fichier') as f:
    read_data = f.read()
```

Ici le mode d'ouverture du fichier n'a pas été précisé. Par défaut, le fichier est donc ouvert en lecture. Pour écire dans un fichier, il suffit d'écrire

```
f = open("fichier",w)
f.write('Always look on the bright side of life!\n')
f.close()
```

Lorsque vous qvez terminé d'utiliser le fichier `f` il faut le fermer avec la fonction `close()`. Lorsqu'on utilise la condition `with`, le fichier est automatiquement fermé à la fin de celle-ci.
La fonction `open` permet aussi d'ouvrir d'autres documents que des texte (comme des images par exemples) mais il faut alors lire et écrire le fichier en binaire en accolant 'b' à la fin du mode.

7.2 Lire et écire des nombres

Pour lire des tableaux de données à partir d'un document texte, nous avons déjà vu que l'on peut utiliser la fonction `loadtxt` de NumPy.

7.3 Lire et écrire des données structurées

Si vous souhaitez stocker des listes ou des dictionnaires, écrire un simple document texte n'est pas le plus efficace. Dans ce cas il vaut mieux utiliser un format de données qui conservent la structure de l'objet que vous souhaitez enregistrer. Inversement, si vous souhaitez lire ces données cela serait fastidieux depuis un simple fichier texte car vous auriez à reconstruire manuellement la structure.
Vous pouvez putôt utiliser le format `json` à partir de la librairie éponyme. L'ériture se fait simplement avec la fonction `dump(liste, fichier)`

```
import json
x = [1, 'simple', 'list']
f = open("mon_fichier")
json.dump(x, f)
f.close()
```

La lecture quant à elle se fait simplement avec la fonction `json.load(fichier)`.

Exercice 18 Loi de Hubble-Lemaître

En astronomie, la loi de Hubble-Lemaître énonce que les galaxies s'éloignent les unes des autres à une vitesse proportionnelle à leur distance. Autrement dit, plus une galaxie est loin de nous, plus elle semble s'éloigner rapidement. La constante de proportionalité, notée H_0, s'appelle *constante de Hubble* et son inverse est une estimation de l'âge de l'Univers. À partir du fichier `hubble.csv` contenant les données récoltées par Edwin Hubble dans son article de 1929, remontez à H_0 et donnez une estimation de l'âge de l'Univers. Que pouvez-vous en dire ?
Si vous avez trouvé une ordonnée à l'origine, quelle est sa signification ?

8 Équations différentielles

Les équations différentielles sont au cœur de la description dynamique des systèmes physiques mais ne sont solubles analytiquement que dans un nombre restreint de cas. Des méthodes de résolutions numériques des équations différentielles, comme la méthode d'Euler que nous avons rencontré précédemment, sont donc nécessaires.
Cependant la méthode d'Euler ne résoud que les EDO du premier ordre. Cette difficulté est contourné en ramenant les équations d'ordres supérieurs à un système d'équations du premier ordre. Considérons l'EDO d'ordre n suivante

$$a_n \frac{\mathrm{d}^n f(t)}{\mathrm{d}t^n} + a_{n-1}\frac{\mathrm{d}^{n-1} f(t)}{\mathrm{d}t^{-1}n} + \cdots + a_1 \frac{\mathrm{d}f(t)}{\mathrm{d}t} + a_0 = 0\,.$$

En introduisant les nouvelle varaibles $f_1(t) = \frac{\mathrm{d}f(t)}{\mathrm{d}t}$, $f_2(t) = \frac{\mathrm{d}f_1(t)}{\mathrm{d}t}$, ..., $f_{n-1}(t) = \frac{\mathrm{d}f_{n-2}(t)}{\mathrm{d}t}$ l'EDO devient

$$\begin{aligned}
\frac{\mathrm{d}f(t)}{\mathrm{d}t} &= f_1(t)\,,\\
\frac{\mathrm{d}f_1(t)}{\mathrm{d}t} &= f_2(t)\,,\\
&\vdots\\
\frac{\mathrm{d}f_{n-1}(t)}{\mathrm{d}t} &= -\frac{a_{n-1}}{a_n} f_{n-1} - \cdots - \frac{a_1}{a_n} f_1(t) - \frac{a_0}{a_n}\,.
\end{aligned}$$

Pour réoudre ce système nous ferons appel à une nouvelle librairie scientifique, `scipy.integrate`, dans laquelle se trouve la fonction `solve_ivp` dont voici un extrait de la documentation :

> `scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **options)`
> *Solve an initial value problem for a system of ODEs.*
> *This function numerically integrates a system of ordinary differential equations given an initial value:*
>
> ```
> dy / dt = f(t, y)
> y(t0) = y0
> ```
>
> *Here t is a 1-D independent variable (time), y(t) is an N-D vector-valued function (state), and an N-D vector-valued function f(t, y) determines the differential equations. The goal is to find y(t) approximately satisfying the differential equations, given an initial value y(t0)=y0.*
>
> Parameters:
>
> - fun: callable
>   *Right-hand side of the system. The calling signature is fun(t, y). Here t is a scalar, and there are two options for the ndarray y: It can either have shape (n,); then fun must return array_like with shape (n,). Alternatively, it can have shape (n, k); then fun must return an array_like with shape (n, k), i.e., each column corresponds to a single column in y. The choice between the two options is determined by vectorized argument (see below). The vectorized implementation allows a faster approximation of the Jacobian by finite differences (required for stiff solvers).*

- t_span: 2-tuple of floats
  Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.
- y0: array_like, shape (n,) *Initial state. For problems in the complex domain, pass y0 with a complex data type (even if the initial value is purely real).*

Returns:
Bunch object with the following fields defined:

- t: ndarray, shape (n_points,)
  Time points.
- y: ndarray, shape (n, n_points)
  Values of the solution at t.
- sol: OdeSolution or None
  Found solution as OdeSolution instance; None if dense_output was set to False.
- t_events: list of ndarray or None
  Contains for each event type a list of arrays at which an event of that type event was detected. None if events was None.
- y_events: list of ndarray or None
  For each value of t_events, the corresponding value of the solution. None if events was None.
- nfev: int
  Number of evaluations of the right-hand side.
- njev: int
  Number of evaluations of the Jacobian.
- nlu: int
  Number of LU decompositions.
- status: int
  Reason for algorithm termination:
  -1: Integration step failed.
  0: The solver successfully reached the end of tspan.
  1: A termination event occurred.
- message: string
  Human-readable description of the termination reason.
- success: bool
  True if the solver reached the interval end or a termination event occurred (status >= 0).

Exercice 19 **Pendule simple**

L'équation différentielle régissant la dynamique du pendule simple n'est pas soluble analytiquement

$$\ddot{\theta}(t) + \omega_0^2 \sin(\theta(t)) = 0 \ .$$

Cependant son approximation aux petits angles l'est

$$\ddot{\theta}(t) + \omega_0^2 \theta(t) = 0 \ .$$

En utilisant la méthode d'Euler que vous avez implémentée, puis la fonction `solve_ivp`, intégrez ces deux équations différentielles.
Les deux méthodes donnent-elles le même résultat ?
Commentez les solutions des deux équations pour la même condition initiale.

Pour les grands angles, l'approximation de Borda est utilisée pour corriger la dynamique

$$\omega(\theta_0)^2 = \omega_0^2 \left(1 - \frac{\theta_0^2}{8}\right) \ .$$

Commentez numériquement la validité de cette approximation.

8.1 Application : l'équation de Schrödinger à $1D$

L'*amplitude de probabilité* d'un système quantique est donnée par la fonction $\psi(x,t)$ qui satisfait une équation d'évolution appelée *équation de Schrödinger*

$$\mathrm{i}\hbar\frac{\partial\psi(x,t)}{\partial t} = \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} - V(x)\right)\psi(x,t)$$

où $V(x)$ est une énergie potentielle. Ici nous allons considérer une énergie potentielle harmonique

$$V(x) = \frac{1}{2}m\omega_0^2(x-x_0)^2 \; . \tag{4}$$

Cette amplitude est reliée à la densité de probabilité $p(x,t)$ de mesurer le système quantique à la position x à l'instant t

$$p(x,t) = |\psi(x,t)|^2 \; ,$$

avec la condition de normalisation

$$\int_{\mathbb{R}} p(x,t)\mathrm{d}x = 1 \; .$$

Nous allons résoudre cette équation différentielle pour t. Pour cela nous allons discrétiser l'espace avec un pas δx et remplacer la fonction $\psi(x,t)$ par un ensemble de fonctions $\psi_n(t)$. La dérivée seconde devient

$$\begin{aligned}
\frac{\partial^2\psi(x,t)}{\partial x^2} &= \frac{1}{\delta x}\left(\frac{\psi_{n+1}(t)-\psi_n(t)}{\delta x} - \frac{\psi_n(t)-\psi_{n-1}(t)}{\delta x}\right) \\
&= \frac{\psi_{n+1}(t) - 2\psi_n(t) + \psi_{n-1}(t)}{\delta x^2} \\
\frac{\partial^2\psi(x,t)}{\partial x^2} &= \frac{1}{\delta x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & \dots \\ 0 & 1 & -2 & 1 & \dots \\ \vdots & \vdots & \vdots & \ddots & \end{pmatrix} \cdot \begin{pmatrix} \psi_1(t) \\ \psi_2(t) \\ \psi_3(t) \\ \vdots \end{pmatrix}
\end{aligned}$$

Cette matrice peut être écrite simplement avec la fonction `sparse.diags` de la librairie `scipy`

```python
dx = 0.01 # Pas de discrétisation
x = np.arange(0, 10, dx) # Liste des positions discrétisée
D2 = scipy.sparse.diags([1, -2, 1], # éléments de la diagonale inférieur, la diagonal et la diagonale sup.
                        [-1, 0, 1], # -1 = diagonale inf., 0 = diagonale, etc.
                        shape=(x.size, x.size)) / dx**2
```

Une fonction pour définir la dérivée temporelle

```python
def ddt_psi(t, psi):
    """définition de la dérivée temporelle de psi_n(t) """
    return -1j * (- 0.5 * hbar / m * D2.dot(psi) + V / hbar * psi)
```

où `1j` correspond au nombre imaginaire i, `dot()` fait un produit matriciel et `V` est une matrice.

```python
from scipy import integrate
from scipy import sparse

import matplotlib.pyplot as plt
import numpy as np

dx    = 0.02                     # spatial separation
x     = np.arange(0, 10, dx)     # spatial grid points

```

```python
kx    = 0.1                       # wave number
m     = 1                         # mass
sigma = 0.1                       # width of initial gaussian wave-packet
x0    = 3.0                       # center of initial gaussian wave-packet
hbar = 1

# Potential V(x)
x_Vmin = 5          # center of V(x)
T      = 1          # peroid of SHO
omega = 2 * np.pi / T
k = omega**2 * m
V = 0.5 * k * (x - x_Vmin)**2

# Dérivée seconde
D2 = sparse.diags([1, -2, 1], [-1, 0, 1], shape=(x.size, x.size)) / dx**2

# Paramètre pour la résolution de l'EDO
dt = 0.005  # time interval for snapshots
t0 = 0.0    # initial time
tf = 1.0    # final time
t_eval = np.arange(t0, tf, dt)  # recorded time shots

# Initial Wavefunction
A = 1.0 / (sigma * np.sqrt(np.pi)) # normalization constant
psi0 = np.sqrt(A) * np.exp(-(x-x0)**2 / (2.0 * sigma**2)) * np.exp(1j * kx * x)

def ddt_psi(t, psi):
    """définition de la dérivée temporelle de psi_n(t) """
    return -1j * (- 0.5 * hbar / m * D2.dot(psi) + V / hbar * psi)

# Make a plot of psi0 and V
plt.plot(x, V*0.01, "k--", label=r"$V(x) = \frac{1}{2}m\omega^2 (x-5)^2$ (x0.01)")
plt.plot(x, np.abs(psi0)**2, "r", label=r"$\vert\psi(t=0,x)\vert^2$")
plt.legend(loc=1, fontsize=8, fancybox=False)
print("Total Probability: ", np.sum(np.abs(psi0)**2)*dx)

# Solve the Initial Value Problem
sol = integrate.solve_ivp(psi_t, t_span = [t0, tf], y0 = psi0, t_eval = t_eval, method="RK23")

fig = plt.figure(figsize=(6, 4))
for i, t in enumerate(sol.t):
    plt.plot(x, np.abs(sol.y[:,i])**2)             # Plot Wavefunctions
#     print(np.sum(np.abs(sol.y[:,i])**2)*dx)        # Print Total Probability (Should = 1)
plt.plot(x, V * 0.01, "k--", label=r"$V(x) = \frac{1}{2}m\omega^2 (x-5)^2$ (x0.01)")    # Plot Potential
plt.legend(loc=1, fontsize=8, fancybox=False)
fig.savefig('sho@2x.png')
```

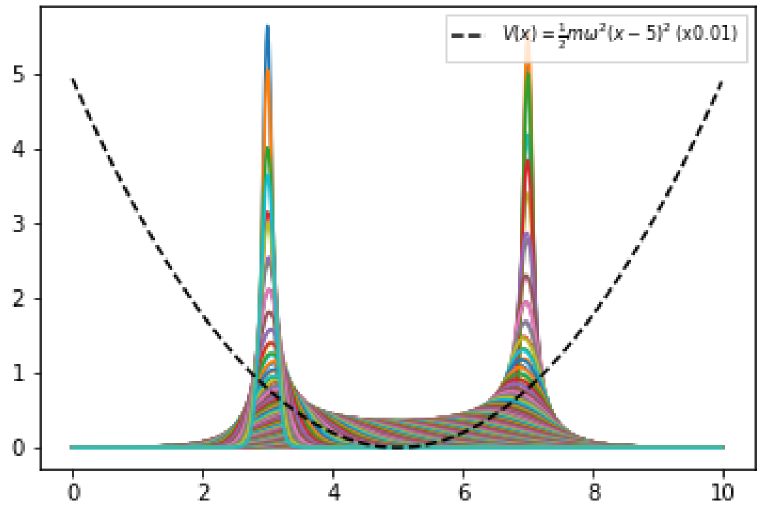


Figure 7: Densité de probabilité au cours du temps dans un potentiel harmonique.

9 Ce que nous n'avons pas vu

Bien entendu il reste beaucoup de choses à dire sur la programmation en général, sur `Python` en particulier et sur les méthodes numériques utiles en sciences. Voici une liste (non-exhaustives) de thèmes non-abordés :

- la gestion des erreurs,
- la programation orientée objet,
- l'intégration numérique,
- l'algèbre linéaire.

Voici quelques liens pour continuer à explorer ces sujets :

- le tutoriel `Python`,
- Hack in Science - Recueil d'exercices pour Python,
- Project Euler - Série d'exercices mathématiques à résoudre numériquement,
- Numerical Recipes - Manuel de référence sur les méthodes numériques.