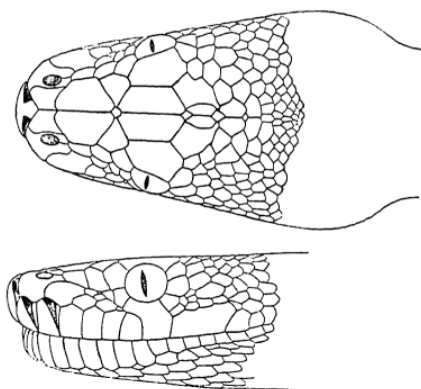


Matthieu Boileau & Vincent Legoll

Apprendre Python *pour les sciences*



Head of *Python molurus*.

Table des matières

| | | |
|----------|---|-----------|
| 1 | Généralités | 7 |
| 1.1 | Programme du cours | 7 |
| 1.1.1 | Généralités | 7 |
| 1.1.2 | Variables et types de données | 7 |
| 1.1.3 | Opérations, contrôle, fonctions et modules | 7 |
| 1.1.4 | Numpy | 8 |
| 1.1.5 | Microprojet | 8 |
| 1.1.6 | Introduction à Pandas | 8 |
| 1.2 | Prise en main des notebooks | 8 |
| 1.2.1 | Les deux modes du notebook : | 8 |
| 1.2.2 | Changement de mode | 8 |
| 1.2.3 | Mode Commande | 8 |
| 1.2.4 | Mode Edition | 8 |
| 1.2.5 | Autres commandes utiles | 9 |
| 1.3 | Le langage Python | 9 |
| 1.3.1 | Historique | 9 |
| 1.3.2 | Un langage en forte croissance | 10 |
| 1.4 | Qu'est-ce qu'un langage interprété ? | 10 |
| 1.5 | Quelques interpréteurs Python | 11 |
| 1.6 | Comment exécuter un programme Python | 11 |
| 1.6.1 | Dans la console Python interactive | 11 |
| 1.6.2 | Depuis la console système | 11 |
| 1.6.3 | Dans la console iPython | 12 |
| 1.6.4 | Au sein d'un notebook Jupyter | 12 |
| 1.6.5 | Utilisation d'un IDE (Environnement de développement intégré) | 13 |
| 1.6.6 | Pourquoi un IDE ? | 13 |
| 1.7 | Applications du langage Python | 13 |
| 1.7.1 | Grand public | 14 |
| 1.7.2 | Mais aussi | 14 |
| 1.7.3 | Intérêt pour les sciences | 14 |
| 1.7.4 | Le Python scientifique | 14 |
| 1.7.5 | Liste non exhaustive de paquets dédiés au Python scientifique : | 15 |
| 1.7.6 | Python plutôt que Matlab ? | 15 |
| 1.8 | Documentation et sources | 15 |
| 1.8.1 | La documentation | 15 |
| 1.8.2 | Les sources de ce support | 15 |
| 2 | Variables et types de données | 17 |
| 2.1 | Langage python et sa syntaxe | 17 |
| 2.1.1 | Variables et affectation | 17 |
| 2.1.2 | Le mécanisme d'affectation en détail | 18 |
| 2.1.3 | Accéder à la valeur d'une variable | 18 |

| | | |
|----------|---|-----------|
| 2.1.4 | L'affectation du point de vue de l'objet | 18 |
| 2.1.5 | Le type | 19 |
| 2.2 | Types de données | 20 |
| 2.3 | Types de base | 21 |
| 2.3.1 | None | 21 |
| 2.3.2 | Booléens | 21 |
| 2.3.3 | Numériques | 22 |
| 2.4 | Séquences | 24 |
| 2.4.1 | Chaînes de caractères | 25 |
| 2.4.2 | Listes | 30 |
| 2.4.3 | Tuples | 36 |
| 2.4.4 | Types muables et types immuables | 37 |
| 2.4.5 | Le slicing de séquences en Python | 39 |
| 2.5 | Chaînes de caractères, le retour | 42 |
| 2.6 | Les dictionnaires | 43 |
| 2.6.1 | Un exemple | 43 |
| 2.6.2 | Un autre exemple | 44 |
| 2.7 | Les ensembles | 45 |
| 2.7.1 | Exemples | 46 |
| 2.8 | Fichiers | 46 |
| 2.8.1 | Ouverture | 46 |
| 2.8.2 | Fermeture | 47 |
| 2.8.3 | Méthodes de lecture | 47 |
| 2.8.4 | Méthodes d'écriture | 47 |
| 3 | Opérations, contrôle, fonctions et modules | 49 |
| 3.1 | Opérateurs | 49 |
| 3.1.1 | Arithmétiques | 49 |
| 3.1.2 | Logiques | 50 |
| 3.1.3 | Comparaison | 51 |
| 3.1.4 | bits à bits (<i>bitwise</i>) | 52 |
| 3.1.5 | Affectation augmentée | 53 |
| 3.1.6 | Compatibilité de type, coercion de type | 54 |
| 3.1.7 | Priorité des opérateurs | 55 |
| 3.2 | Structures de contrôle | 55 |
| 3.2.1 | La mise en page comme syntaxe | 55 |
| 3.2.2 | <code>pass</code> | 55 |
| 3.2.3 | Tests conditionnels | 56 |
| 3.3 | Boucles | 57 |
| 3.3.1 | Boucle <code>while</code> | 57 |
| 3.3.2 | Boucle <code>for</code> | 57 |
| 3.3.3 | Instruction <code>break</code> | 59 |
| 3.3.4 | Instruction <code>continue</code> | 60 |
| 3.4 | Fonctions | 60 |
| 3.4.1 | Fonctions sans arguments | 60 |
| 3.4.2 | Fonctions avec arguments | 62 |
| 3.4.3 | Espace de nommage et portée des variables | 66 |
| 3.4.4 | Fonctions <i>built-in</i> | 67 |
| 3.4.5 | Exercices sur les fonctions | 68 |
| 3.5 | Exceptions | 70 |
| 3.5.1 | Exercice | 71 |
| 3.6 | Les gestionnaires de contexte | 72 |
| 3.7 | Les compréhensions de listes | 73 |
| 3.8 | Les expressions génératrices | 73 |

| | | |
|----------|---|-----------|
| 3.9 | Modules | 74 |
| 3.9.1 | Créer ses propres modules | 74 |
| 3.9.2 | Imports relatifs | 77 |
| 3.9.3 | Exercice | 78 |
| 3.9.4 | Quelques modules de la <code>stdlib</code> | 79 |
| 3.10 | Bonnes pratiques | 80 |
| 3.10.1 | Commentez votre code | 80 |
| 3.10.2 | Documentez en utilisant les <i>docstrings</i> | 80 |
| 3.10.3 | Utilisez les annotations de type | 80 |
| 3.10.4 | Conventions d'écriture | 81 |
| 3.10.5 | Organisez votre code source | 81 |
| 3.11 | Utiliser les environnements de développement intégrés : | 81 |
| 3.11.1 | Utilisez un gestionnaire de versions | 82 |
| 3.11.2 | Héberger vos dépôts de sources sur des forges logicielles | 82 |
| 3.11.3 | Vérificateurs de code source | 82 |
| 3.11.4 | Tests en Python | 82 |
| 3.11.5 | Environnements virtuels | 82 |
| 3.12 | Philosophie du langage : le zen de Python | 82 |
| 3.13 | Python 3.x vs 2.x | 83 |
| 3.13.1 | Différences notables entre Python 2 et Python 3 | 83 |
| 3.14 | Exercice de synthèse : décodage de l'ARN | 84 |
| 3.14.1 | Enoncé | 84 |
| 3.14.2 | Consignes | 84 |
| 3.14.3 | Solution complète | 85 |
| 3.15 | Exercices complémentaires | 85 |
| 3.15.1 | Chaines de caractères | 85 |
| 3.15.2 | Récursion | 86 |
| 4 | Une introduction à Numpy | 89 |
| 4.1 | Numpy | 89 |
| 4.2 | Démarrer avec Numpy | 90 |
| 4.3 | Tableaux Numpy | 90 |
| 4.3.1 | Une question de performance | 90 |
| 4.3.2 | La différence avec les listes | 91 |
| 4.3.3 | Propriétés | 91 |
| 4.4 | Création de tableaux Numpy | 91 |
| 4.4.1 | Avec des valeur constantes | 91 |
| 4.4.2 | Création à partir d'une séquence | 92 |
| 4.4.3 | Exercice : créer les tableaux suivants | 93 |
| 4.4.4 | Création de tableaux à partir de fichiers | 93 |
| 4.4.5 | Format HDF5 avec H5py | 94 |
| 4.5 | Opérations basiques sur les tableaux | 94 |
| 4.6 | Indexation et slicing | 95 |
| 4.6.1 | Indexation | 95 |
| 4.6.2 | Slicing | 95 |
| 4.6.3 | Exercice | 96 |
| 4.7 | Quelques opérations d'algèbre linéaire | 98 |
| 4.8 | Les méthodes associées aux tableaux | 99 |
| 4.9 | Broadcasting de tableaux | 99 |
| 4.9.1 | Règles de broadcasting | 100 |
| 4.9.2 | Exemple 1 : on étend un tableau | 101 |
| 4.9.3 | Exemple 2 : on étend les deux tableaux | 101 |
| 4.9.4 | Exemple 3 : broadcasting impossible | 102 |
| 4.9.5 | Exemple d'utilisation | 102 |

| | | |
|----------|---|------------|
| 4.10 | Programmation avec Numpy | 103 |
| 4.11 | Références | 103 |
| 5 | Microprojet | 105 |
| 5.1 | Exercice | 105 |
| 5.1.1 | Ouverture du fichier de prévisions | 105 |
| 5.1.2 | Chargement du fichier json ouvert | 106 |
| 5.1.3 | Exploration des données | 106 |
| 5.1.4 | Tracé de la température | 108 |
| 5.2 | Exercice sur les fonctions | 109 |
| 5.3 | Exécution avec les widgets ipython | 110 |
| 5.4 | Exécution en script | 110 |
| 5.4.1 | Utilisation de <code>if __name__ == '__main__':</code> | 110 |
| 5.4.2 | Gestion des arguments de la ligne de commande | 111 |
| 5.5 | Utilisation avancée de Spyder | 114 |
| 6 | Une introduction à Pandas | 115 |
| 6.1 | Un outil pour l'analyse de données | 115 |
| 6.2 | Les <i>Pandas series</i> | 115 |
| 6.2.1 | Illustration | 116 |
| 6.2.2 | Une série temporelle | 116 |
| 6.2.3 | Un exemple de traitement | 116 |
| 6.2.4 | Indexation et slicing | 119 |
| 6.2.5 | Ordonner la série | 120 |
| 6.3 | Les <i>Pandas Dataframes</i> | 120 |
| 6.3.1 | Un exemple avec les arbres de la ville de Strasbourg | 120 |
| 6.4 | Représentation géographique | 125 |
| 6.4.1 | Exercice | 131 |
| 6.4.2 | Utilisation des widgets ipython | 132 |
| 6.4.3 | Vers des applications web | 132 |
| 6.5 | Références | 132 |
| 6.6 | Annexe : une autre façon de représenter les occurrences de mots | 132 |

Chapitre 1

Généralités



Contenu sous licence [CC BY-SA 4.0](#)

1.1 Programme du cours

1.1.1 Généralités

- Prise en main des notebooks
- Généralités sur le langage Python

1.1.2 Variables et types de données

- Qu'est-ce qu'une variable en python ?
- Revue des types de données

1.1.3 Opérations, contrôle, fonctions et modules

- Opérateurs
- Structures de contrôle
- Fonctions
- Exceptions et gestionnaires de contexte
- Compréhensions de listes & expressions génératrices
- Modules
- Bonnes pratiques

1.1.4 Numpy

1.1.5 Microprojet

1.1.6 Introduction à Pandas

1.2 Prise en main des notebooks

Le document que vous lisez est un [notebook Jupyter](#). Il est constitué de cellules comportant :

- soit du texte en [Markdown](#) comme ici,
- soit du code Python, comme dans la cellule suivante :

```
In [1]: print('Hello world!')
```

Hello world!

Note : - `print()` est une fonction fournie par python pour afficher du texte à destination de l'utilisateur. - **Entrée** [] : ou **In** [] : indique le nombre d'exécutions du noyau.

1.2.1 Les deux modes du notebook :

- *Commande* : permet de se déplacer d'une cellule à l'autre et d'exécuter les cellules
- *Edition* : permet de modifier le contenu d'une cellule.

1.2.2 Changement de mode

- *Commande* -> *Edition* : touche `Enter` ou double-clic dans la cellule.
- *Edition* -> *Commande* :
 - Touche `Esc` pour basculer sans exécuter
 - Touches `Shift + Enter` pour *exécuter* la cellule et passer à la suivante. Exécuter une cellule en Markdown provoque le rendu visuel de celle-ci.

Exercice : Revenez en arrière, sélectionnez et exécutez (`Shift + Enter`) la cellule de code qui contient

```
print('Hello world!')
```

1.2.3 Mode Commande

- On se déplace à l'aide des flèches `↑↓` ou en cliquant avec la souris.
- On peut ajouter, effacer, déplacer, créer ou modifier le contenu des cellules à l'aide des menus déroulants en haut de la page

1.2.4 Mode Edition

- on entre dans le mode édition avec la touche `Enter` ou par un double-clic
- signalement par la petite icône “crayon” en haut à droite, dans la barre de menu
- on en sort avec `Ctrl + Enter` (ou `Shift + Enter` pour passer à la cellule suivante)

Note :

La plupart des actions à la souris peuvent se faire à l'aide des raccourcis du menu *Help > Keyboard Shortcuts* (touche H)

Exercice : Revenez en arrière, sélectionnez la cellule `print('Hello world!')`. Modifiez son contenu, par exemple en traduisant le message en français. Réexécutez-la et observez le nouveau résultat. Remarquez que le nombre d'exécutions augmente.

1.2.5 Autres commandes utiles

- Redémarrer le kernel Python : Bouton correspondant ou touche 0,0 en mode *Commande*
- Changer le type de cellule : *Code* -> *Markdown* : touche M en mode *Commande*
- Changer le type de cellule : *Markdown* -> *Code* : touche Y en mode *Commande*
- Ajouter une nouvelle cellule au dessus de l'actuelle : touche A (pour above) en mode *Commande*
- Ajouter une nouvelle cellule en dessous de l'actuelle : touche B (pour below) en mode *Commande*

Exercice 1. Suivre le tour guidé de l'interface : *Help > User Interface Tour* 2. Avancer dans le notebook avec + 3. Passer du mode **Commande** au mode **Edition** de différentes façons 4. Ouvrez la rubrique **Keyboard Shortcuts** de l'aide et tenter de reproduire les actions uniquement avec les touches du clavier

Notes :

- Vous pouvez vous servir de votre copie d'un notebook pour faire office de bloc-notes en ajoutant des cellules de texte pour vos commentaires et des cellules de code pour vos essais de résolution d'exercices.
- **Ctrl + S** pour sauver vos modifications

Exercice : 1. Ajoutez une cellule de texte ci-dessous (touches B puis M) et insérez une note. 2. Ajoutez une cellule de code ci-dessous (touche B) et insérez un exemple de code.



1.3 Le langage Python

- Langage interprété, originellement écrit en C
- Open-source, portable et disponible sur Unix, Windows, MacOS, etc.
- Syntaxe claire et simple
- Orienté objet
- Types nombreux et puissants
- Interfaces avec de nombreux autres langages et librairies
- Large spectre d'applications

Plus d'informations sur [wikipedia](https://fr.wikipedia.org/wiki/Python_(langage_de_programmation)) ou sur le site officiel de [python](https://python.org).

1.3.1 Historique

La genèse du langage date de la fin des années 80. [Guido van Rossum](#), alors à l'Institut de Recherche en Mathématiques et Informatique Hollandais (CWI) à Amsterdam a publié la version 0.9.0 de l'interpréteur en Février 1991. Il travaille maintenant pour dropbox après 7 ans chez google.

- Plus d'histoire sur [wikipedia](https://fr.wikipedia.org/wiki/Python_(langage_de_programmation)).
- L'histoire racontée par le créateur lui-même sur [son blog](#) sous forme d'anecdotes.

1.3.2 Un langage en forte croissance

Dans [cet article](#) de 2017, le site [stackoverflow.com](#) relevait la forte croissance de Python par rapport aux autres langages depuis 2012.

Depuis cet article, la tendance mensuelle des requêtes Stackoverflow s'est poursuivie. On peut la suivre sur <https://insights.stackoverflow.com/>. En 2024, elle donne :

1.4 Qu'est-ce qu'un langage interprété ?

- Ordinateur → CPU → jeu d'instructions (ISA) → langage binaire
- Un langage de programmation permet d'écrire des programmes dans des langages mieux adaptés aux humains, mais nécessite une étape de traduction.
- Comme pour une langue étrangère, il nous faut un traducteur ou un interprète...
 - Le traducteur lit l'ensemble du texte et en produit une version dans la langue étrangère.
 - L'interprète lit le texte et le traduit au fur et à mesure.
- Pour un langage informatique, il existe une analogie avec les [compilateurs](#) et les [interpréteurs](#)
 - Le compilateur traduit tout le code source en langage binaire utilisable directement par le CPU.
 - L'interpréteur lit une partie du code source, exécute directement les instructions binaires correspondantes et passe à la suite.

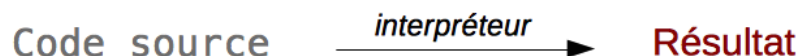
Un langage interprété sera souvent moins rapide qu'un langage compilé car les optimisations sont plus faciles à réaliser lors d'une compilation.

Cette différence a tendance à s'estomper avec l'apparition des techniques suivantes :

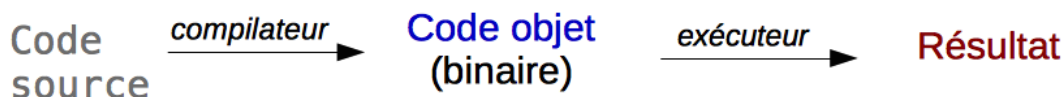
- [JIT](#) : compilation à la volée (Just In Time compilation)
- [RTTS](#) : spécialisation de types au moment de l'exécution (Run Time Type Specialization)

Une autre possibilité pour contourner la lenteur d'exécution d'un langage est de faire appel à des bibliothèques externes programmées et optimisées dans un langage compilé. C'est très efficace pour les parties du code qui sont utilisées de manière répétitive.

Langage interprété (ex : bash)



Langage compilé (ex : C, C++, Fortran)



Langage Python



Figure inspirée du livre [Apprendre à programmer en Python](#) de G. Swinnen.

1.5 Quelques interpréteurs Python

- [CPython](#) : Implémentation de référence
- [Jython](#) : Java byte code, accès aux classes java
- [IronPython](#) : CLR byte code, accès aux classes .NET
- [Pyjamas](#) : JavaScript, Ajax, GWT
- [Stackless Python](#) : pas de pile, microthreads, coroutines
- [Shed Skin](#) : C++, typage statique
- [Cython](#) : C, compilateur créant des modules python
- [Pyrex](#) : langage proche de python, C
- [PyPy](#) : JIT, RTTS, RPython → C, Java byte code, CLR byte code
- [Nuitka](#) : C, fortement compatible

1.6 Comment exécuter un programme Python

1.6.1 Dans la console Python interactive

La commande `python` lance la console interactive python dans laquelle on peut exécuter le code directement :

```
$ python
Python 3.11.6 (main, Oct 2 2023, 13:45:54) [Clang 15.0.0 (clang-1500.0.40.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> print(a)
2
>>>
```

1.6.2 Depuis la console système

On peut l'exécuter en paramètre de la ligne de commande

```
$ python -c 'a=3;print(a)'
```

Sous windows :

```
C:\> python.exe -c 'a=3;print(a)'
```

On peut exécuter un fichier (par exemple `test.py`) contenant notre code

```
$ python test.py
```

On peut exécuter directement un fichier python contenant notre code, grâce à l'utilisation du mode script avec, en rajoutant en première ligne du fichier :

```
#!/python
```

Après avoir rendu le fichier exécutable. Sous Linux :

```
$ chmod a+x test.py
```

Il peut alors être exécuté sans le précéder du nom de l'interpréteur :

```
$ ./test.py
```

1.6.3 Dans la console iPython

Le terminal interactif **iPython** peut s'utiliser comme alternative à la console Python classique pour ses fonctionnalités :

- syntaxe additionnelle
- complétion
- commandes système
- historique enrichi

Aperçu du terminal `ipython` :

```
$ ipython
Python 3.6.2 |Anaconda custom (x86_64)| (default, Sep 21 2017, 18:29:43)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.
```

In [1]:

Un exemple d'utilisation

Sauvegarde de l'historique des commandes avec la *magic function* `%save`

```
$ ipython
[...]

In [1]: print('- Hello world!')
- Hello world!

In [2]: R = 'Hello you! '

In [3]: print(R*6)
Hello you! Hello you! Hello you! Hello you! Hello you! Hello you!

In [4]: %save hello.py 1-3
The following commands were written to file `hello.py`:
print('- Hello world!')
R = 'Hello you! '
print(R*6)

In [5]: quit
```

1.6.4 Au sein d'un notebook Jupyter

Les cellules de type *code* vous donnent accès à une console qui inclut la plupart des fonctionnalités de la console iPython.

Exercice :

1. Exécuter la cellule de code ci-dessous et observez le résultat
2. Exécuter cette cellule une deuxième fois, observez la différence

```
In [2]: print('- Hello world!')
        R = '- Hello you!\n'
        print(R*6)
        %history
```

```
- Hello world!
- Hello you!
- Hello you!
- Hello you!
- Hello you!
- Hello you!
- Hello you!

print('Hello world!')
print('- Hello world!')
R = '- Hello you!\n'
print(R*6)
%history
```

1.6.5 Utilisation d'un IDE (Environnement de développement intégré)

Un grand nombre d'IDE sont disponibles pour Python (cf. la [revue wikipedia](#) et la [revue wiki.python.org](#)).

Citons simplement :

- **IDLE** : l'IDE par défaut de Python.
- **Spyder** : l'IDE qui sera utilisé pour certains exercices de ce cours.
- **Pycharm** : une alternative moins libre que Spyder mais avec de nombreuses fonctionnalités comme le support pour les outils de suivi de version ou la complétion automatique.
- **VSCode** : un IDE gratuit, non limité à Python, aux fonctionnalités très riches grâce à son système d'extensions.

1.6.6 Pourquoi un IDE ?

Dans un IDE, on dispose d'outils intégrés dans une interface (généralement) intuitive :

- débogueur
- analyseur de variables
- outils d'inspection de code
- outils de coloration et de correction syntaxique
- raccourcis d'exécution
- profileurs de code

1.7 Applications du langage Python

Python est un langage complet, utilisable dans un grand nombre de domaines.

1.7.1 Grand public



1.7.2 Mais aussi

- openstack - gestionnaire de cloud
- yum - installateur de paquets redhat, centos
- softimage - modelage et rendu 3D
- maya - modelage et rendu 3D
- inkscape - éditeur de graphiques vectoriels
- gnuradio - SDR (software defined radio) toolkit

Et encore plus d'exemples sur wikipedia.

1.7.3 Intérêt pour les sciences

Pour un usage scientifique, Python est intéressant à plusieurs titres. En effet, il est capable de réaliser de manière automatique et efficace un certain nombre de tâches qui sont le quotidien des scientifiques : - Manipuler et traiter des données de simulations ou d'expériences - Visualiser des résultats - Communiquer ses résultats sous la forme de données numériques formatées, de figures ou d'animations - Dans le domaine du **calcul scientifique**, Python est particulièrement riche en fonctionnalités grâce à la contribution importante de la communauté des mathématiques et du calcul à travers le projet SciPy.

1.7.4 Le Python scientifique

1. Développement de code de calcul

Bien que généralement moins performant que les langages compilés (C, C++ ou Fortran), Python est particulièrement intéressant et agréable à programmer dans les phases de développement pour tester rapidement de nouvelles méthodes. Une fois le prototypage terminé, il est possible de porter les parties critiques du code vers un langage compilé plus rapide, tout en gardant le reste en python.

2. Traitement de données

- langage de haut niveau produisant du code agréable à lire (par opposition à excel, par exemple...)
- nombreux modules spécialisés (algèbre, statistique, traitement d'images, etc.)
- le concept des **notebooks Jupyter** qui combinent de l'exécution de code, du texte formaté, des formules mathématiques (via LaTeX), des tracés et du contenu média

3. Tracés graphiques :

- tracés 1D, 2D voire 3D
- animations

1.7.5 Liste non exhaustive de paquets dédiés au Python scientifique :

- [NumPy](#) : calcul numérique, opérations mathématiques sur tableaux et matrices de grandes dimensions
- [SciPy](#) : ensemble d'outils scientifiques pour le traitement de signal, d'images, algèbre linéaire, etc.
- [SymPy](#) and [SAGE](#) : bibliothèques et outils mathématiques pour le calcul symbolique
- [Matplotlib](#) : tracer et visualiser des données sous forme graphique, à la matlab ou mathematica
- [Pandas](#) : analyser vos données
- [Pytorch](#) : bibliothèque d'apprentissage machine développée par Facebook
- [TensorFlow](#) : bibliothèque d'apprentissage machine développée par Google
- [Scikit-learn](#) : apprentissage machine, fouille et analyse de données
- [BioPython](#) : problèmes de biologie : génomique, modélisation moléculaire, etc.
- [AstroPy](#) : bibliothèque communautaire dédiée à l'astronomie

1.7.6 Python plutôt que Matlab ?

Constat

- On dispose ainsi d'un outil très complet et performant qui représente une alternative sérieuse aux outils commerciaux tels que Matlab, Maple, Mathematica, etc.
- Le site [scipy-lectures](#) fournit de bons pointeurs vers les applications scientifiques de Python avec Scipy.

Alors ?

- Une discussion intéressante : [Python vs Matlab](#)
- Un guide de migration : [Numpy for Matlab users : guide](#)
- Une table de conversion de la syntaxe : [NumPy for MATLAB users : syntax](#)

1.8 Documentation et sources

1.8.1 La documentation

- Officielle :
 - [L'index](#)
 - [La FAQ](#)
 - [La librairie standard](#)
 - [Des tutoriels](#)
- [Stackoverflow](#) : Forum de questions / réponses

1.8.2 Les sources de ce support

Quelques ressources qui ont inspiré le contenu de ce cours et qui pourront vous servir pour aller plus loin...

avec le langage

- Le MOOC de l'INRIA : [Python 3 : des fondamentaux à l'utilisation du langage](#) hébergé sur la plateforme FUN
- La formation du [groupe Calcul](#) : ANF "Python avancé en calcul scientifique"
- La formation de Pierre Navaro : [Python pour le calcul](#)

- Le cours de python scientifique de l’institut de science du télescope spatial : [STSCI’s Scientific Python Course 2015](#) (en anglais)

avec les notebooks Jupyter

- Ce qu’on peut écrire en Markdown et en LaTeX dans les notebooks [Jupyter](#) et ce qu’on peut faire dans les cellules de code dans cette [série de tutoriels](#)
- Pour mettre vos notebooks en ligne : [nbviewer](#), [binder](#)

en s’entraînant

- Site de tutorat social : [Python Tutor](#)

Chapitre 2

Variables et types de données



- Variables
- Types de données
- Fichiers

Contenu sous licence [CC BY-SA 4.0](#)

2.1 Langage python et sa syntaxe

2.1.1 Variables et affectation

Pour accéder à une donnée, on lui donne un nom que l'on appelle variable. Cette opération s'appelle une **affectation** (ou *assignation*) et se fait avec l'opérateur `=`.

```
variable_x = donnee_x
```

signifie qu'on affecte la `donnee_x` à la `variable_x`.

Exécutez la cellule suivante pour définir trois variables nommées : `age`, `prenom` et `taille`.

```
In [1]: # Par exemple: donnons la valeur 23 à la variable 'age'
age = 23
# Les variables peuvent se référer à divers types de données : des chaînes de caractères...
prenom = 'Julien'
# Des nombres réels, etc...
taille = 1.83
```

2.1.2 Le mécanisme d'affectation en détail

Lors de l'affectation `x = donnee_x`, Python :

1. crée et mémorise le nom de variable `x`
2. crée un objet dont le type dépend de la nature de la donnée et y stocke la valeur particulière `donnee_x`
3. établit une référence entre le nom de la variable `x` et l'emplacement mémoire de `donnee_x`. Cette référence est mémorisée dans l'espace de nom.

2.1.3 Accéder à la valeur d'une variable

Pour se servir de la donnée référencée par une variable, il suffit d'utiliser le nom cette variable.

La fonction `print()` affiche à l'écran ce qui lui est passé en paramètre. On peut lui donner plusieurs paramètres séparés par des virgules : `print()` les affichera tous, séparés par un espace.

```
In [2]: print('Julien', 23, 1.83)
        print(prenom, 'a', age, 'ans, et mesure', taille, 'mètre')
```

```
Julien 23 1.83
Julien a 23 ans, et mesure 1.83 mètre
```

Les variables peuvent changer et ainsi se référer à une autre donnée.

```
In [3]: age = 23
        print(age)
        age = 24
        print(age)
```

```
23
24
```

2.1.4 L'affectation du point de vue de l'objet

La notion d'objet

En python, toute donnée est contenue dans un **objet**. En programmation, un objet est un conteneur qui regroupe :

- des données
- les méthodes pour interagir avec ces données

Exemple de l'objet de type entier relatif (`int`) :

- sa donnée : la valeur du nombre entier
- ses méthodes : les opérations mathématiques, la conversion vers les chaînes de caractères, etc.

Affecter une valeur à une variable revient à nommer un objet, c'est-à-dire référencer cet objet par un nom.

Exemple avec l'entier 7 :

- pour Python, 7 est l'objet de type entier (`int`) dont la donnée est le nombre 7
- lorsqu'on écrit 7 dans un programme, Python crée l'objet correspondant et lui affecte un identifiant unique dans l'espace de nom courant.
- cet identifiant est renvoyé par la fonction intrinsèque `id()` :

```
In [4]: print(id(7))
```

```
94777227912296
```

```
x = 7
```

signifie qu'on nomme l'objet 7 avec la variable x. Autrement dit, x devient **une référence vers cet objet**.

```
In [5]: print(id(7))
        x = 7
        print(id(x))
        y = x
        print(id(y))  # y référence le même objet que x
```

```
94777227912296
```

```
94777227912296
```

```
94777227912296
```

2.1.5 Le type

Les variables n'ont pas de type propre : c'est la donnée qui est typée, pas la variable qui la référence. Une variable peut donc faire référence à une donnée d'un autre type après une nouvelle affectation.

La fonction `type()` retourne le type effectif de la donnée passée en paramètre.

```
In [6]: # Des nombres
        age = 23
        print(type(age))
```

```
<class 'int'>
```

```
In [7]: # Les variables peuvent aussi changer de type : chaîne de caractères
        age = 'vingt quatre ans'
        print(type(age))
```

```
<class 'str'>
```

```
In [8]: # Sans limites...
        age = 24.5
        print(type(age))
```

```
<class 'float'>
```

```
In [9]: # Attention aux pièges...
        age = '25'
        print(type(age))
```

```
<class 'str'>
```

Une variable peut être initialisée avec des valeurs constantes, comme vu précédemment, mais aussi à partir d'autres variables ou de valeurs retournées par des opérations, des fonctions, etc.

Exemple : La fonction `max()` retourne le plus grand de ses paramètres.

```
In [10]: a = 1
         b = a
         c = a * 2
         d = max(a, 2, 3, c)
         print(a, b, c, d)
```

1 1 2 3

Les variables, une fois définies dans une cellule **exécutée**, continuent d'exister dans les suivantes car toutes les cellules sont connectées à la même instance du noyau python.

Note : En cas de redémarrage du notebook, toutes les variables existantes sont détruites, il faut donc ré-exécuter les cellules qui les définissent si l'on veut de nouveau pouvoir les utiliser.

```
In [11]: abcd = 1234
```

Ici, la variable nommée `abcd` survit d'une cellule à la suivante...

```
In [12]: print(abcd)
```

1234

Si on veut effacer une variable de l'espace de nom, on peut utiliser l'assertion `del`.

```
In [13]: # Cette cellule génère une erreur
         a = 2
         print(a)
         del a
         print(a)
```

2

```
-----
NameError                                Traceback (most recent call last)
Cell In[13], line 5
      3 print(a)
      4 del a
----> 5 print(a)

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

Note : `del` est aussi utilisé pour enlever des éléments dans des conteneurs modifiables (listes, dictionnaires). Nous aborderons ce sujet plus tard.

2.2 Types de données

Types de base

- None
- Booléens
- Numériques
 - entiers
 - flottants
 - complexes

Séquences

- Chaines de caractères
- listes
- tuples

Conteneurs

- Dictionnaires
- Ensembles

Fichiers

2.3 Types de base

2.3.1 None

- Il existe dans python un type de données particulier : `None`.
- `None` représente un objet sans valeur. On s'en sert comme valeur de retour en cas d'erreur ou pour représenter un cas particulier.
- `None` est équivalent à `NULL` en Java et en C.

```
In [14]: a = None
         print(a)
         print(type(a))
```

```
None
<class 'NoneType'>
```

2.3.2 Booléens

Les booléens ne peuvent avoir que deux valeurs :

`True`, `False`

On peut utiliser la fonction `bool()` pour construire un booléen.

Dans un contexte booléen, toutes ces valeurs sont équivalentes à `False` :

- `None`
- le zéro des types numériques, par exemple : `0`, `0.0`, `0j`.
- les séquences vides, par exemple : `''`, `()`, `[]`.
- les dictionnaires vides, par exemple, `{}`.

Tout le reste est équivalent à `True` :

- une valeur numérique différente de zéro
- une séquence non vide
- un dictionnaire non vide

Exemples de constructions de valeurs booléennes à partir d'autres types

```
In [15]: print(bool(None), bool())
         print(bool(0), bool(1))
         print(bool(0.0), bool(0.5))
         print(bool(0j), bool(3j))
         print(bool(''), bool('abc'))
         print(bool(()), bool((1, 0.5, 'toto')))
         print(bool([]), bool([None]))
         print(bool({}), bool({'ane': True, 'chat': True}))
```

```
False False
False True
False True
False True
```

```
False True
False True
False True
False True
```

Exemple d'utilisation d'un contexte booléen

```
In [16]: Am_I_OK = True
         if Am_I_OK:
             print('OK')
         else:
             print('KO')
         print(Am_I_OK)
         print(type(Am_I_OK))
```

```
OK
True
<class 'bool'>
```

Exercice : 1. Supposons que vous soyez malade, mettez `Am_I_OK` à la valeur `False` puis réexécutez la cellule. 2. Essayez avec d'autres types : `list`, `None`, nombres, etc...

2.3.3 Numériques

Entiers

La taille est illimitée.

```
In [17]: # Il n'y a plus de distinction entre les entiers "courts" et les entiers "longs"
         entier = 4
         print(entier)
         print(type(entier))
         # Ce nombre nécessite 131 bits
         entier = 1415926535897932384626433832795028841971
         print(entier)
         print(type(entier))

4
<class 'int'>
1415926535897932384626433832795028841971
<class 'int'>
```

Ou plus exactement, la taille n'est pas limitée par le langage mais par la mémoire disponible sur le système.

Note : En python version < 3, il existait deux types distincts `int` et `long`...

- On peut utiliser la fonction interne `int()` pour créer des nombres entiers.
- `int()` peut créer des entiers à partir de leur représentation sous forme de chaîne de caractères
- On peut aussi spécifier la base :

```
In [18]: entier = int(12)
         print(entier)
         print(type(entier))
         entier = int('13')
         print(entier)
```

```

    print(type(entier))
    entier = int('0xFF', 16)
    print(entier)
    print(type(entier))

12
<class 'int'>
13
<class 'int'>
255
<class 'int'>

```

Flottants (réels 64 bits)

Pour créer des nombres réels (en virgule flottante), on peut utiliser la fonction interne `float()`. La précision est limitée à 16 chiffres significatifs.

```

In [19]: pi_approx = 3.1415926535897932
         print(pi_approx)
         print(type(pi_approx))
         print('{:.16f}'.format(pi_approx))
         tropgros = float('Infinity')
         print(type(tropgros))
         print(tropgros)

3.141592653589793
<class 'float'>
3.1415926535897931
<class 'float'>
inf

```

Flottants

Attention! Python autorise un affichage plus long que la précision des flottants mais tous les chiffres après le 16ème chiffre significatif sont faux :

```

In [20]: # On demande 20 chiffres après la virgule, alors que 16 seulement sont exacts
         print('{:.20f}'.format(pi_approx))

3.14159265358979311600

```

Attention à ne pas effectuer des opérations interdites...

```

In [21]: # Cette cellule génère une erreur
         print(3.14 / 0)

```

```

-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[21], line 2
      1 # Cette cellule génère une erreur
----> 2 print(3.14 / 0)

ZeroDivisionError: float division by zero

```

```
In [22]: # Cette cellule génère une erreur
         print(34 % 0)
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[22], line 2
      1 # Cette cellule génère une erreur
----> 2 print(34 % 0)

ZeroDivisionError: integer modulo by zero
```

```
In [23]: # Cette cellule génère une erreur
         print(27 // 0.0)
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[23], line 2
      1 # Cette cellule génère une erreur
----> 2 print(27 // 0.0)

ZeroDivisionError: float floor division by zero
```

```
In [24]: # Cette cellule génère une erreur
         print(1 + None)
```

```
-----
TypeError                                         Traceback (most recent call last)
Cell In[24], line 2
      1 # Cette cellule génère une erreur
----> 2 print(1 + None)

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Complexes

Les nombres complexes, à deux dimensions, peuvent être créés en utilisant le suffixe `j` à la fin d'un entier ou réel ou en utilisant la fonction interne `complex()`

```
In [25]: c1 = 1 + 2j
         print('représentation :', c1)
         print(type(c1))
         c2 = .3j
         print(c2)
```

```
représentation : (1+2j)
<class 'complex'>
0.3j
```

2.4 Séquences

Les séquences sont des conteneurs d'objets où les objets référencés sont ordonnés. Python supporte nativement trois types de séquences :

- les chaînes de caractères
- les listes
- les tuples

2.4.1 Chaînes de caractères

Pour le traitement de données textuelles, python utilise les chaînes de caractères.

Pour délimiter le texte on utilise des guillemets " (double-quote) ou des apostrophes ' (single-quote).

```
In [26]: mois1 = 'janvier'
        mois2 = "février"
        print(mois1, mois2)
```

```
janvier février
```

Les deux formes sont très utiles car elles permettent d'utiliser des guillemets ou des apostrophes dans des chaînes de caractères de manière simple.

```
In [27]: arbre = "l'olivier"
        print(arbre)
        cuisson_pates = "huit minutes (8')"
        print(cuisson_pates)
        record_100m = 'neuf secondes et 58 centièmes (9"58)'
        print(record_100m)
```

```
l'olivier
huit minutes (8')
neuf secondes et 58 centièmes (9"58)
```

Sinon, pour avoir une apostrophe ou un guillemet dans une chaîne de caractères, il faut le faire précéder d'un \ (*backslash*). Ce qui est beaucoup moins lisible, mais parfois obligatoire (par exemple une chaîne avec à la fois des guillemets et des apostrophes).

```
In [28]: arbre = 'l\'alisier'
        pates = '8\''
        record = "9\"58"
        print(arbre, pates, record)
        duel = 'guillemet: " et apostrophes: \'' peuvent être utilisés dans une même chaîne...'
        print(duel)
        multi = '''Dans une "triplequoted" (voire plus loin), on peut (presque) tout utiliser: `"" ",'"«»'"" et ça
        print(multi)
```

```
l'alisier 8' 9"58
guillemet: " et apostrophes: ' peuvent être utilisés dans une même chaîne...
Dans une "triplequoted" (voire plus loin), on peut (presque) tout utiliser: `"" ",'"«»'"" et ça ne pose
```

Caractères spéciaux

Il est possible de définir des chaînes de caractères qui contiennent des caractères spéciaux. Ils sont introduits par des séquences de deux caractères dont le premier est \. On l'appelle le caractère d'échappement.

- retour à la ligne : \n
- tabulation : \t
- *backslash* : \\
- un caractère unicode avec son code : \uXXXX (où les XXXX sont le code hexadécimal représentant ce caractère)

Plus d'information dans la [documentation officielle](#)

```
In [29]: print("Une belle présentation, c'est :\n\t- bien ordonné\n\t- aligné\n\t- même s'il y en a plein\nEt c'est
print("Une belle présentation, c'est :")
print('\t- bien ordonné')
print('\t- aligné')
print("\t- même s'il y en a plein\nEt c'est plus joli.")
```

```
Une belle présentation, c'est :
- bien ordonné
- aligné
- même s'il y en a plein
```

```
Et c'est plus joli.
```

```
Une belle présentation, c'est :
- bien ordonné
- aligné
- même s'il y en a plein
```

```
Et c'est plus joli.
```

Chaînes multilignes

Pour écrire plusieurs lignes d'une façon plus lisible, il existe les chaînes multilignes :

```
In [30]: # L'équivalent est :
print("""\
Une belle présentation, c'est :
\t- bien ordonné
\t- aligné
\t- même s'il y en a plein
Et c'est plus joli.""")
```

```
Une belle présentation, c'est :
- bien ordonné
- aligné
- même s'il y en a plein
```

```
Et c'est plus joli.
```

Exercice : enlevez le caractère \ de la 2ème ligne dans la cellule ci-dessus, et comparez le résultat à celui de la cellule de code précédente (3 cellules plus haut)

Les deux formes de délimiteurs sont aussi utilisables : guillemets triples ou apostrophes triples.

```
In [31]: multi1 = '''m
a
r
s est un mois "multiligne"""
print(multi1, '\n')

multi2 = """a
v
r
i
l l'est aussi"""
print(multi2)
```

```
m
a
```

```

r
s est un mois "multiligne"

a
v
r
i
l l'est aussi

```

Unicode

En python 3, toutes les chaînes de caractères sont unicode, ce qui permet d'utiliser des alphabets différents, des caractères accentués ou des pictogrammes, etc.

Exemples de caractères spéciaux :

```

In [32]: unicode_str = "Les échecs ( ), c'est \u263A"
         print(unicode_str)
         japanese_str = 'Du japonais : '
         print(japanese_str)
         mix_str = ' kollha '
         print('"' + mix_str + '"', 'veut dire: "bonjour tout le monde"')

```

```

Les échecs ( ), c'est
Du japonais :
" kollha " veut dire: "bonjour tout le monde"

```

Pour une liste de caractères unicode, voir [ici](#).

Chaînes brutes (*raw strings*)

Il s'agit de chaînes dans lesquelles les séquences d'échappement ne sont pas remplacées : `r'...'`, `r'''...'''`, etc.

```

In [33]: normal_str = "chaîne normale: \n est un retour à la ligne, \t est une tabulation"
         print(normal_str)

         raw_str = r"chaîne RAW: \n est un retour à la ligne, \t est une tabulation"
         print(raw_str)

```

```

chaîne normale:
  est un retour à la ligne,          est une tabulation
chaîne RAW: \n est un retour à la ligne, \t est une tabulation

```

Concaténation

Plusieurs chaînes de caractères contiguës sont rassemblées.

```

In [34]: b = 'a' 'z' 'e' 'r' 't' 'y'
         print(b)

```

```
azerty
```

On peut mélanger les genres.

```
In [35]: a = 'une chaine qui est \t' r'''la somme de \n ''' ""plusieurs morceaux...""
         print(a)
```

```
une chaine qui est          la somme de \n plusieurs morceaux...
```

On peut utiliser la fonction `str()` pour créer une chaîne de caractère à partir d'autres objets.

```
In [36]: a = 23
         ch_a = str(a)
         print(type(a), repr(a))
         print(type(ch_a), repr(ch_a))
         a = 3.14
         ch_a = str(a)
         print(type(a), a)
         print(type(ch_a), repr(ch_a))
```

```
<class 'int'> 23
<class 'str'> '23'
<class 'float'> 3.14
<class 'str'> '3.14'
```

On ne peut pas mélanger les guillemets et les apostrophes pour délimiter une chaîne de caractères.

```
In [37]: # Cette cellule génère une erreur
         a = "azerty'
```

```
Cell In[37], line 2
```

```
    a = "azerty'
```

```
    ^
```

```
SyntaxError: unterminated string literal (detected at line 2)
```

```
In [38]: # Cette cellule génère une erreur
         a = 'azerty"
```

```
Cell In[38], line 2
```

```
    a = 'azerty"
```

```
    ^
```

```
SyntaxError: unterminated string literal (detected at line 2)
```

Exercice : corrigez les deux cellules ci dessus.

Le formatage de chaîne de caractère avec `format()`

On peut formater du texte, c'est à dire utiliser une chaîne de caractères qui va servir de modèle pour en fabriquer d'autres. Historiquement, il existe plusieurs méthodes. Nous ne voyons ici qu'une utilisation basique de la méthode `format()`.

`format()` remplace les occurrences de '{}' par des valeurs qu'on lui spécifie en paramètre. Quelque soit le type des valeurs passées, une représentation sous forme de chaîne de caractère sera automatiquement créée, avec la fonction `str()`.

```
'Bonjour {} !'.format('le monde')
```

```
In [39]: variable_1 = 27
         variable_2 = 'vingt huit'
         ch_modele = 'Une chaine qui contient un {} ainsi que {} et là {}'
         ch_modele.format('toto', variable_1, variable_2)

Out[39]: 'Une chaine qui contient un toto ainsi que 27 et là vingt huit'
```

Les spécifications de format

Des indicateurs peuvent être fournis à la méthode `format()` pour spécifier le type de donnée à intégrer et la manière de formater sa représentation.

- `:d` pour des entiers
- `:s` pour des chaînes de caractères
- `:f` pour des nombres flottants
- `:x` pour un nombre entier affiché en base hexadécimale
- `:o` pour un nombre entier affiché en base octale
- `:e` pour un nombre affiché en notation exponentielle

```
In [40]: import math
         a = 27
         ch = """un entier : {:d},
         une chaîne : {:s},
         un flottant avec une précision spécifiée : {:.02f}"""
         print(type(ch))
         print(ch.format(a, 'Arte', math.pi))
         print('Des hexadécimaux: {:x} {:x} {:x} {:x}'.format(254, 255, 256, 257))
         print('Des octaux: {:o} {:o} {:o} {:o}'.format(254, 255, 256, 257))
         print('Des nombres en notation exponentielle {:e}'.format(2**64))
```

```
<class 'str'>
un entier : 27,
une chaîne : Arte,
un flottant avec une précision spécifiée : 3.14
Des hexadécimaux: fe ff 100 101
Des octaux: 376 377 400 401
Des nombres en notation exponentielle 1.844674e+19
```

Attention : Si le type de la donnée passée ne correspond pas à la séquence de formatage, python va remonter une erreur.

```
In [41]: # Cette cellule génère une erreur
         variable_3 = 'une chaîne de caracteres'
         # Exemple d'erreur de type : on fournit une chaîne de caractères
         # alors que la méthode attend un entier
         print('on veut un entier : {:d}'.format(variable_3))
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[41], line 5
      2 variable_3 = 'une chaîne de caracteres'
      3 # Exemple d'erreur de type : on fournit une chaîne de caractères
      4 # alors que la méthode attend un entier
----> 5 print('on veut un entier : {:d}'.format(variable_3))

ValueError: Unknown format code 'd' for object of type 'str'
```

On peut se servir de cette fonctionnalité pour indenter du texte de taille variable.

```
In [42]: print("""
    Alignons à droite les animaux:
    Un animal : {:>8s}.
    Un animal : {:>8s}.
    Un animal : {:>18s}, qui se croit plus malin que les autres.
    Un animal : {:>8s}.
    Un animal : {:>8s}.""").format('âne', 'becasse', 'chat', 'dinde', 'elephant'))
```

```
Alignons à droite les animaux:
Un animal :      âne.
Un animal :  becasse.
Un animal :                chat, qui se croit plus malin que les autres.
Un animal :      dinde.
Un animal : elephant.
```

Exercice : remettez le chat à sa place.

f-string

Depuis Python 3.6, il existe une autre syntaxe appelée *f-string* qui ajoute de la concision et de la lisibilité :

```
f"La {<variable>:<format>} est nommée dans la chaîne."

In [43]: print('Soit {:.03f} la valeur de pi sur 3 décimales.'.format(math.pi))
    # devient
    print(f'Soit {math.pi:.03f} la valeur de pi sur 3 décimales.')
```

```
Soit 3.142 la valeur de pi sur 3 décimales.
Soit 3.142 la valeur de pi sur 3 décimales.
```

Remarques :

- cette syntaxe n'est pas toujours utilisable
- le code sera incompatible avec python < 3.6

Pour plus d'informations sur le formatage de chaînes de caractères, voir la [doc Python](#) correspondante.

2.4.2 Listes

- Une liste est un objet pouvant contenir d'autres objets
- Ces objets, appelés éléments, sont ordonnés de façon séquentielle, les uns à la suite des autres
- C'est un conteneur dynamique dont le nombre et l'ordre des éléments peuvent varier

On crée une liste en délimitant par des crochets [] les éléments qui la composent :

```
In [44]: L = ['egg', 'spam', 'spam', 'spam', 'bacon']
    print(f"{L} est de type {type(L)}")

['egg', 'spam', 'spam', 'spam', 'bacon'] est de type <class 'list'>
```

Une liste peut contenir n'importe quel type d'objets.

```
In [45]: L0 = [1, 2]
        L1 = []
        L2 = [None, True, False, 0, 1, 2**64, 3.14, '', 0+1j, 'abc']
        L3 = [[1, 'azerty'], L0]
        print(L0, L1)
        print(L2)
        print(L3)

[1, 2] []
[None, True, False, 0, 1, 18446744073709551616, 3.14, '', 1j, 'abc']
[[1, 'azerty'], [1, 2]]
```

On peut utiliser la fonction `list()` pour créer une liste à partir d'autres séquences ou objets.

```
In [46]: a = list()
        b = list('bzzzzzt')
        print(a)
        print(b)

[]
['b', 'z', 'z', 'z', 'z', 'z', 't']
```

On accède aux éléments d'une liste grâce à un indice. Le premier élément est **indiqué 0**.

```
In [47]: print(L)
        print(L[0])
        print(L[4])

['egg', 'spam', 'spam', 'spam', 'bacon']
egg
bacon
```

Un dépassement d'indice produit une erreur :

```
In [48]: # Cette cellule génère une erreur
        print(L[10])
```

```
-----
IndexError                                Traceback (most recent call last)
Cell In[48], line 2
      1 # Cette cellule génère une erreur
----> 2 print(L[10])

IndexError: list index out of range
```

Les listes sont dites *muables* : on peut modifier la séquence de ses éléments. Je remplace le deuxième élément :

```
In [49]: L[1] = 'tomatoes'
        print(L)
        L[3] = 9
        print(L)

['egg', 'tomatoes', 'spam', 'spam', 'bacon']
['egg', 'tomatoes', 'spam', 9, 'bacon']
```

Méthodes associées aux listes

Méthodes ne modifiant pas la liste

- La longueur d’une liste est donnée par fonction `len()`
- `L.index(elem)` renvoie l’indice de l’élément `elem` (le 1er rencontré)

Méthodes modifiant la liste

- `L.append()` ajoute un élément à la fin
- `L.pop()` retourne le dernier élément et le retire de la liste
- `L.sort()` trie
- `L.reverse()` inverse l’ordre

Plus d’infos dans la [doc officielle](#).

Exemples

```
In [50]: L = ['egg', 'spam', 'spam', 'spam', 'bacon']
         print('len() renvoie:', len(L))
         L.append('spam') # Ne renvoie pas de valeur
         print('Après append():', L)
         print('len() renvoie:', len(L))
```

len() renvoie: 5

Après append(): ['egg', 'spam', 'spam', 'spam', 'bacon', 'spam']

len() renvoie: 6

```
In [51]: print('pop() renvoie:', L.pop())
         print('Après pop():', L)
```

pop() renvoie: spam

Après pop(): ['egg', 'spam', 'spam', 'spam', 'bacon']

```
In [52]: L.reverse() # Ne renvoie pas de valeur
         print('Après reverse():', L)
```

Après reverse(): ['bacon', 'spam', 'spam', 'spam', 'egg']

```
In [53]: print('index() renvoie:', L.index('egg'))
```

index() renvoie: 4

```
In [54]: L.remove('spam') # Ne renvoie pas de valeur
         print('Après remove:', L)
```

Après remove: ['bacon', 'spam', 'spam', 'egg']

Dans les exemples précédents, remarquez la syntaxe qui permet d’appliquer une méthode à un objet :

`objet.methode()`

Ici, `.methode()` est une fonction propre au type de `objet`. Si `.methode()` existe pour un autre type, elle n’a pas forcément le même comportement.

Pour obtenir la liste des méthodes associées aux listes, on peut utiliser la fonction interne `help()` :


```
In [55]: help(L) # ou aussi help([])
```

Help on list object:

```
class list(object)
| list(iterable=(), /)
|
| Built-in mutable sequence.
|
| If no argument is given, the constructor creates a new empty list.
| The argument must be an iterable if specified.
|
| Methods defined here:
|
| __add__(self, value, /)
|     Return self+value.
|
| __contains__(self, key, /)
|     Return key in self.
|
| __delitem__(self, key, /)
|     Delete self[key].
|
| __eq__(self, value, /)
|     Return self==value.
|
| __ge__(self, value, /)
|     Return self>=value.
|
| __getattr__(self, name, /)
|     Return getattr(self, name).
|
| __getitem__(...)
|     x.__getitem__(y) <==> x[y]
|
| __gt__(self, value, /)
|     Return self>value.
|
| __iadd__(self, value, /)
|     Implement self+=value.
|
| __imul__(self, value, /)
|     Implement self*=value.
|
| __init__(self, /, *args, **kwargs)
|     Initialize self. See help(type(self)) for accurate signature.
|
| __iter__(self, /)
|     Implement iter(self).
|
| __le__(self, value, /)
|     Return self<=value.
|
| __len__(self, /)
```

```

|     Return len(self).
|
|     __lt__(self, value, /)
|         Return self<value.
|
|     __mul__(self, value, /)
|         Return self*value.
|
|     __ne__(self, value, /)
|         Return self!=value.
|
|     __repr__(self, /)
|         Return repr(self).
|
|     __reversed__(self, /)
|         Return a reverse iterator over the list.
|
|     __rmul__(self, value, /)
|         Return value*self.
|
|     __setitem__(self, key, value, /)
|         Set self[key] to value.
|
|     __sizeof__(self, /)
|         Return the size of the list in memory, in bytes.
|
|     append(self, object, /)
|         Append object to the end of the list.
|
|     clear(self, /)
|         Remove all items from list.
|
|     copy(self, /)
|         Return a shallow copy of the list.
|
|     count(self, value, /)
|         Return number of occurrences of value.
|
|     extend(self, iterable, /)
|         Extend list by appending elements from the iterable.
|
|     index(self, value, start=0, stop=9223372036854775807, /)
|         Return first index of value.
|
|         Raises ValueError if the value is not present.
|
|     insert(self, index, object, /)
|         Insert object before index.
|
|     pop(self, index=-1, /)
|         Remove and return item at index (default last).
|
|         Raises IndexError if list is empty or index is out of range.

```

```

| remove(self, value, /)
|     Remove first occurrence of value.
|
|     Raises ValueError if the value is not present.
|
| reverse(self, /)
|     Reverse *IN PLACE*.
|
| sort(self, /, *, key=None, reverse=False)
|     Sort the list in ascending order and return None.
|
|     The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
|     order of two equal elements is maintained).
|
|     If a key function is given, apply it once to each list item and sort them,
|     ascending or descending, according to their function values.
|
|     The reverse flag can be set to sort in descending order.
|
| -----
| Class methods defined here:
|
| __class_getitem__(...) from builtins.type
|     See PEP 585
|
| -----
| Static methods defined here:
|
| __new__(*args, **kwargs) from builtins.type
|     Create and return a new object.  See help(type) for accurate signature.
|
| -----
| Data and other attributes defined here:
|
| __hash__ = None

```

On peut créer facilement des listes répétitives grâce à l'opération de multiplication.

```

In [56]: a = ['a', 1] * 5
         print(a)

['a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1]

```

Mais on ne peut pas 'diviser' une liste.

```

In [57]: # Cette cellule génère une erreur
         a = [1, 2, 3, 4]
         print(a / 2)

```

```

-----
TypeError
Cell In[57], line 3

```

Traceback (most recent call last)

```

1 # Cette cellule génère une erreur
2 a = [1, 2, 3, 4]
----> 3 print(a / 2)

```

TypeError: unsupported operand type(s) for /: 'list' and 'int'

Exercice : Manipulez la liste L ci-dessous avec les méthodes associées aux listes.

```

In [58]: L = ['egg', 'spam', 'spam', 'spam', 'bacon']
         # Votre code ci-dessous

```

Vous trouverez la documentation complète sur les listes [ici](#).

2.4.3 Tuples

Les Tuples (ou n-uplets en Français) sont des séquences *immuables* : on ne peut pas les modifier après leur création.

On les initialise ainsi :

```

In [59]: T = ('a', 'b', 'c')
         print(T, 'est de type', type(T))
         T = 'a', 'b', 'c' # une autre façon, en omettant les parenthèses
         print(T, 'est de type', type(T))
         T = tuple(['a', 'b', 'c']) # à partir d'une liste
         print(T, 'est de type', type(T))
         T = ('a') # ceci n'est pas un tuple
         print(T, 'est de type', type(T))
         T = ('a',) # Syntaxe pour initialiser un tuple contenant un seul élément
         print(T, 'est de type', type(T))
         # Syntaxe alternative pour initialiser un tuple contenant un seul élément.
         # Préférez celle avec parenthèses.
         T = 'a',

```

```

('a', 'b', 'c') est de type <class 'tuple'>
('a', 'b', 'c') est de type <class 'tuple'>
('a', 'b', 'c') est de type <class 'tuple'>
a est de type <class 'str'>
('a',) est de type <class 'tuple'>

```

Une fois créée, cette séquence ne peut être modifiée.

```

In [60]: T = ('a', 'b', 'c')
         print(T[1]) # On peut utiliser un élément

```

b

```

In [61]: # Cette cellule génère une erreur
         T[1] = 'z' # mais on ne peut pas le modifier

```

```

-----
TypeError                                Traceback (most recent call last)
Cell In[61], line 2
      1 # Cette cellule génère une erreur
----> 2 T[1] = 'z' # mais on ne peut pas le modifier

TypeError: 'tuple' object does not support item assignment

```

Intérêt des tuples par rapport aux listes : - plus rapide à parcourir que les listes - immuables donc “protégés” - peuvent être utilisés comme clé de dictionnaires (cf. plus loin)

On peut créer des tuples à partir d’autres séquences ou objets grâce à la fonction `tuple()`.

```
In [62]: a = [1, 2, 3, 'toto']
         print(type(a), a)
         b = tuple(a)
         print(type(b), b)
         a = 'azerty'
         print(type(a), a)
         b = tuple(a)
         print(type(b), b)

<class 'list'> [1, 2, 3, 'toto']
<class 'tuple'> (1, 2, 3, 'toto')
<class 'str'> azerty
<class 'tuple'> ('a', 'z', 'e', 'r', 't', 'y')
```

Manipulation des tuples

Construire d’autres tuples par concaténation et multiplication

```
In [63]: T1 = 'a', 'b', 'c'
         print('T1 =', T1)
         T2 = 'd', 'e'
         print('T2 =', T2)
         print('T1 + T2 =', T1 + T2)
         print('T2 * 3 =', T2 * 3)

T1 = ('a', 'b', 'c')
T2 = ('d', 'e')
T1 + T2 = ('a', 'b', 'c', 'd', 'e')
T2 * 3 = ('d', 'e', 'd', 'e', 'd', 'e')
```

2.4.4 Types muables et types immuables

Avant d’aller plus loin dans la revue des types, il est important de comprendre le mécanisme d’affectation en fonction du caractère muable ou immuable de l’objet.

Cas d’un objet muable

- Deux noms de variables différents peuvent référencer le même objet
- Si cet objet est muable, les modifications faites par l’intermédiaire d’une des variables sont visibles par toutes les autres.

```
In [64]: a = ['spam', 'egg']  # on initialise la variable a
         b = a                # b référence le même objet que a
```

a et b possèdent la même référence :

```
In [65]: print(id(a))
         print(id(b))
         print(a is b)
```

```
140577237405696
140577237405696
True
```

a et b contiennent la même donnée :

```
In [66]: print(a)
          print(b)
          a == b

['spam', 'egg']
['spam', 'egg']
```

Out[66]: True

Si on modifie la donnée de a, la donnée de b est aussi modifiée !

```
In [67]: a.append('bacon')
          print(a)
          print(b)

['spam', 'egg', 'bacon']
['spam', 'egg', 'bacon']
```

Cas d'un objet immuable

```
In [68]: t = 'spam', 'egg'
          u = t
          print(id(t))
          print(id(u))
          print(t is u)
```

```
140576995467200
140576995467200
True
```

t et u sont deux variables qui référencent le même objet **tuple** donc leur donnée ne peut être modifiée. Tout ce qu'on peut faire, c'est affecter une nouvelle valeur :

```
In [69]: t = 'bacon', 'egg'
```

Dans ce cas, t référence un nouvel objet alors que u référence toujours l'objet initial :

```
In [70]: print(id(t))
          print(id(u))
          print(t is u)
```

```
140576997323200
140576995467200
False
```

Et bien sûr leurs données sont différentes :

```
In [71]: print(t)
          print(u)
          t == u

('bacon', 'egg')
('spam', 'egg')
```

Out[71]: False

Exercice : analyser ce qu'il se passe dans cette série d'instructions avec [Python Tutor](#) .

Un peu plus loin...

Bien que sa séquence soit immuable, si un **tuple** est constitué d'éléments muables, alors ces éléments-là peuvent être modifiés.

Illustration avec un **tuple** dont un des éléments est une **list** :

```
In [72]: T = ('a', ['b', 'c']) # le deuxième élément est une liste donc il est muable
          print('T =', T)
          L = T[1]
          print('L =', L)
          L[0] = 'e'
          print('L =', L)
          print('T =', T)
```

```
T = ('a', ['b', 'c'])
L = ['b', 'c']
L = ['e', 'c']
T = ('a', ['e', 'c'])
```

```
In [73]: # Ici on fait exactement la même chose...
          T = ('a', ['b', 'c'])
          print('T =', T)
          T[1][0] = 'z'
          print('T après =', T)
```

```
T = ('a', ['b', 'c'])
T après = ('a', ['z', 'c'])
```

```
In [74]: # Cette cellule génère une erreur
          T[0] = 'A' # Ici, on essaye de modifier le tuple lui même...
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[74], line 2
      1 # Cette cellule génère une erreur
----> 2 T[0] = 'A' # Ici, on essaye de modifier le tuple lui même...

TypeError: 'tuple' object does not support item assignment
```

Exercice : analyser ce qu'il se passe dans cette série d'instructions avec [Python Tutor](#) .

2.4.5 Le slicing de séquences en Python

- Cela consiste à extraire une sous-séquence à partir d'une séquence.
- Le slicing fonctionne de manière similaire aux intervalles mathématiques : `[début:fin[`
- La borne de fin ne fait pas partie de l'intervalle sélectionné.
- La syntaxe générale est `L[i:j:k]`, où :
 - `i` = indice de début
 - `j` = indice de fin, le premier élément qui n'est pas sélectionné
 - `k` = le "pas" ou intervalle (s'il est omis alors il vaut 1)
- La sous-séquence sera donc composée de tous les éléments de l'indice `i` jusqu'à l'indice `j-1`, par pas de `k`.
- La sous-séquence est un nouvel objet.

Dans le sens normal (le pas `k` est positif)

- Si `i` est omis alors il vaut 0
- Si `j` est omis alors il vaut `len(L)`

Dans le sens inverse (le pas `k` est négatif)

- Si `i` est omis alors il vaut `-1`
- Si `j` est omis alors il vaut `-len(L)-1`

Illustrons ça en créant une liste à partir d'une chaîne de caractères.

La fonction `split()` découpe une chaîne de caractères en morceaux, par défaut en 'mots'.

```
In [75]: L = 'Dans le Python tout est bon'.split()
         print(L)

['Dans', 'le', 'Python', 'tout', 'est', 'bon']
```

Pour commencer, on extrait de la liste `L` un nouvel objet liste qui contient tous les éléments de `L` \Leftarrow copie de liste

```
In [76]: print(L[0:len(L):1]) # Cette notation est inutilement lourde car :
         print(L[:])          # i = 0, j=len(L) et k=1 donc i, j et k peuvent être omis
         print(L[:])          # on peut même omettre le 2ème ":"

['Dans', 'le', 'Python', 'tout', 'est', 'bon']
['Dans', 'le', 'Python', 'tout', 'est', 'bon']
['Dans', 'le', 'Python', 'tout', 'est', 'bon']
```

On extrait une sous-liste qui ne contient que les 3 premiers éléments :

```
In [77]: print(L[0:3:1]) # Notation complète
         print(L[:3:1]) # Le premier indice vaut i=0, donc on peut l'omettre
         print(L[:3])  # Le pas de slicing vaut 1, donc on peut l'omettre, ainsi que le ":"

['Dans', 'le', 'Python']
['Dans', 'le', 'Python']
['Dans', 'le', 'Python']
```

J'extrais une sous-liste qui exclut les trois premiers éléments :

```
In [78]: print(L[3:len(L):1]) # Cette notation est inutilement lourde car :
         print(L[3:])          # j et k peuvent être omis, ainsi que le ":"

['tout', 'est', 'bon']
['tout', 'est', 'bon']
```

Les indices peuvent être négatifs, ce qui permet traiter les derniers éléments :

```
In [79]: # Je veux exclure le dernier élément :
         print(L[0:-1:1]) # Notation complète
         print(L[:-1:1]) # Le premier indice vaut i=0, donc on peut l'omettre
         print(L[:-1])  # Le pas de slicing vaut 1, donc on peut l'omettre

['Dans', 'le', 'Python', 'tout', 'est']
['Dans', 'le', 'Python', 'tout', 'est']
['Dans', 'le', 'Python', 'tout', 'est']
```


On ne garde que les deux derniers éléments

```
In [80]: print(L[-2:])

['est', 'bon']
```

Note

`L[1]` n'est pas équivalent à `L[1:2]`, ni à `L[1:]`, ni à `L[:1]`.
Illustration :

```
In [81]: a = L[1]
         print(type(a), a) # Je récupère le deuxième élément de la liste
         a = L[1:2]
         print(type(a), a) # Je récupère une liste composée du seul élément L[1]
         a = L[1:]
         print(type(a), a) # Je récupère une liste
         a = L[:1]
         print(type(a), a) # Je récupère une liste

<class 'str'> le
<class 'list'> ['le']
<class 'list'> ['le', 'Python', 'tout', 'est', 'bon']
<class 'list'> ['Dans']
```

Exercice : Retourner une liste composée des éléments de `L` *en ordre inverse* avec une opération de slicing. Toute utilisation de `[]`.`reverse()` ou `reversed()` est interdite.

```
In [82]: L = 'Dans le Python tout est bon'.split()
         print(L)
         # <- votre code ici

['Dans', 'le', 'Python', 'tout', 'est', 'bon']
```

```
In [83]: # Décommentez la ligne suivante pour voir la solution
         # %load exos/snippets/02_langage/reverse.py
```

Le slicing peut être utilisé pour **modifier** une séquence **muable** grâce à l'opération d'affectation.

```
In [84]: L = 'Dans le Python tout est bon'.split()
         print(L)
         L[2:4] = ['nouvelles', 'valeurs', 'et encore plus...', 1, 2, 3]
         print(L)

['Dans', 'le', 'Python', 'tout', 'est', 'bon']
['Dans', 'le', 'nouvelles', 'valeurs', 'et encore plus...', 1, 2, 3, 'est', 'bon']
```

Le slicing peut être utilisé sur des chaînes de caractères.

```
In [85]: alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

Exercice : 1. Découpez l'alphabet en deux parties égales 2. Prenez une lettre sur deux

```
In [86]: # Votre code ici
```

```
In [87]: # Décommentez la ligne suivante pour voir la solution
         # %load exos/snippets/02_langage/alphabet.py
```

2.5 Chaînes de caractères, le retour

Les chaînes de caractères sont des séquences **immuables** donc on les manipule comme telles.

```
In [88]: # Cette cellule génère une erreur
         immutable = 'abcdefgh'
         immutable[3] = 'D'
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[88], line 3
      1 # Cette cellule génère une erreur
      2 immutable = 'abcdefgh'
----> 3 immutable[3] = 'D'

TypeError: 'str' object does not support item assignment
```

Il faut construire une nouvelle chaîne de caractère. En concaténant des morceaux (slices) de la chaîne originale :

```
In [89]: nouvelle_chaine = immutable[:3] + 'D' + immutable[4:]
         print(nouvelle_chaine)
```

abcDefgh

Ou alors en utilisant une transformation en liste, puis à nouveau en chaîne :

```
In [90]: a = list(immutable)
         print(a)
         a[3] = 'D'
         print(a)
         print(''.join(a))

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
['a', 'b', 'c', 'D', 'e', 'f', 'g', 'h']
abcDefgh
```

On peut savoir si une chaîne se trouve dans une autre

```
In [91]: print('123' in 'azerty_123_uiop')
         print('AZE' in 'azerty_123_uiop')
         print('aze' in 'azerty_123_uiop')
```

True
False
True

La longueur d'une chaîne s'obtient avec `len()`.

```
In [92]: print(len(immutable))
```

Exercice, dans la cellule ci-dessous : 1. **Insérez** le caractère `#` au milieu de la chaîne donnée. 2. Idem mais **coupez** la chaîne en 3 parties, et insérez le caractère `@` entre chacune d'elles. 3. **Insérez** le caractère `|` entre chaque caractère de la chaîne.

```
In [93]: chaine_donnee = 'azertyuiiooiuytreza'
        # Votre code ici
```

```
In [94]: # Décommentez la ligne suivante pour voir la solution
        # %load exos/snippets/02_langage/azerty.py
```

2.6 Les dictionnaires

Les dictionnaires ou listes associatives sont des conteneurs où les objets ne sont **pas** ordonnés ni accessibles par un indice mais sont associés à une clé d'accès.

L'accès aux éléments se fait comme pour les listes ou tuples, avec les `[]`.

```
dico = {cle1: valeur1, cle2: valeur2, ...}
```

Les clés peuvent avoir n'importe quelle valeur à condition qu'elles soient de type immuable : les listes ne peuvent pas servir de clés alors que les chaînes de caractères et les tuples le peuvent.

Dans `dico`, on accède à `valeur1` avec la syntaxe `dico[cle1]`.

2.6.1 Un exemple

```
In [95]: dic_animaux = {'ane': True, 'arbre': False, 'chat': True, 'lune': False, 'humain': True}
        cle = 'chat'
        valeur = dic_animaux[cle]
        print(valeur)
        print('{} est un animal: {}'.format(cle, valeur))
        # Ou encore
        print('{} est un animal: {}'.format('chat', dic_animaux['chat']))
        # En utilisant les f-strings :
        print(f"{'chat'} est un animal: {dic_animaux['chat']}")
```

```
True
chat est un animal: True
chat est un animal: True
chat est un animal: True
```

Exercice : essayez de savoir si un arbre est un animal

```
In [96]: # Votre code ici
        dic_animaux["toto"]
```

```
-----
KeyError                                Traceback (most recent call last)
Cell In[96], line 2
      1 # Votre code ici
----> 2 dic_animaux["toto"]

KeyError: 'toto'
```

Les différentes manières de créer des dictionnaires, en particulier grâce à la fonction interne `dict()` :

```
In [97]: a = {'un': 1, 'deux': 2, 'trois': 3}           # Les accolades comme syntaxe
        b = dict(un=1, deux=2, trois=3)              # La méthode dict()
        c = dict(zip(['un', 'deux', 'trois'], [1, 2, 3])) # On "zippe" deux listes
        d = dict([('deux', 2), ('un', 1), ('trois', 3)]) # On transforme une liste de 2-tuples
        e = dict({'trois': 3, 'un': 1, 'deux': 2})
        a == b == c == d == e
```

Out[97]: True

Accéder à un élément qui n'est pas dans le dictionnaire génère une erreur. Il existe la méthode `{}.get()` ou l'opérateur `in` qui permettent d'éviter ce problème.

```
In [98]: # Cette cellule génère une erreur
        err = a['quatre']
```

```
-----
KeyError                                Traceback (most recent call last)
Cell In[98], line 2
      1 # Cette cellule génère une erreur
----> 2 err = a['quatre']

KeyError: 'quatre'
```

Ici on utilise `.get()` et cela ne remonte pas d'erreur.

```
In [99]: print(a.get('quatre'))      # La valeur par défaut est 'None' lorsque .get() ne trouve pas l'élément
        print(a.get('quatre', 5))    # On peut spécifier une autre valeur par défaut
```

```
None
5
```

2.6.2 Un autre exemple

```
In [100]: tup_mois = ('jan', 'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov', 'dec')
        tup_long = (31, (28, 29), 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
        dic_mois = dict(zip(tup_mois, tup_long))
        mois_naissance = 'jul'
        print(f'Il y a {dic_mois[mois_naissance]:d} jours dans votre mois de naissance')
```

Il y a 31 jours dans votre mois de naissance

Exercice :

1. Modifiez la cellule précédente pour y changer `mois_naissance` (les 3 premiers caractères de votre mois de naissance, en anglais). Ré-exécutez la cellule et vérifiez la réponse.
2. Un problème s'est glissé dans la cellule, lequel ?
3. Corrigez-le (il y a plusieurs manières de faire).

Les dictionnaires sont muables.

```
In [101]: ages = {'albert': 62, 'bob': 34, 'charlie': 1, 'daphne': 67}
        print(ages)
        print(f'Albert a {ages["albert"]} ans.')
        # C'est l'anniversaire de charlie, il a un an de plus...
        ages['charlie'] += 1 # equivalent à : ages['charlie'] = ages['charlie'] + 1
        print(ages)
        # Bob est parti, enlevons-le
        del ages['bob']
        print(ages)
```

```
{'albert': 62, 'bob': 34, 'charlie': 1, 'daphne': 67}
Albert a 62 ans.
{'albert': 62, 'bob': 34, 'charlie': 2, 'daphne': 67}
{'albert': 62, 'charlie': 2, 'daphne': 67}
```

- Savoir si une clé est présente dans un dictionnaire est une opération rapide. On utilise, comme pour les séquences, l'opérateur `in`.
- La fonction interne `len()` est utilisable pour savoir combien d'objets sont référencés dans le dictionnaire.

```
In [102]: ages = {'albert': 62, 'bob': 34, 'charlie': 1, 'daphne': 67}
          print('Charlie est dedans ?', 'charlie' in ages)
          print('Zoé est dedans ?', 'zoé' in ages)
          print('Bob est dedans ?', 'bob' in ages)
          print(f'Il y a {len(ages):d} personnes.')
          # Bob est parti, enlevons-le
          del ages['bob']
          print('Bob est dedans ?', 'bob' in ages)
          print(f'Il y a {len(ages):d} personnes.')
```

```
Charlie est dedans ? True
Zoé est dedans ? False
Bob est dedans ? True
Il y a 4 personnes.
Bob est dedans ? False
Il y a 3 personnes.
```

On peut itérer sur les clés ou les objets référencés, ou vider un dictionnaire, en comparer deux, etc. Pour plus d'informations sur les dictionnaires, voir [ici](#).

Exercice :

1. Créez un dictionnaire qui va traduire des chiffres (de 1 à 3) écrits en toutes lettres entre deux langues. Par exemple : `trad_num['un'] → 'one'`
2. Modifiez ce dictionnaire, pour qu'il fonctionne dans les deux sens de traduction (Fr → En et En → Fr)
3. Modifiez ce dictionnaire, pour qu'il fonctionne aussi avec les chiffres sous forme d'entiers. Par exemple : `trad_num[1] → 'un'`

```
In [103]: # Votre code ici
```

```
In [104]: # Décommentez la ligne suivante pour voir la solution
          # %load exos/snippets/02_langage/trad_num.py
```

2.7 Les ensembles

La fonction `set()` permet de créer des ensembles. Les ensembles sont des conteneurs qui n'autorisent pas de duplication d'objets référencés, contrairement aux listes et tuples.

On peut créer des ensembles de cette façon :

```
ensemble = set(iterable)
```

Où `iterable` peut être n'importe quel objet qui supporte l'itération : liste, tuple, dictionnaire, un autre `set` (pour en faire une copie), vos propres objets itérables, etc...

Tout comme pour les dictionnaires, l'opérateur `in` est efficace.

2.7.1 Exemples

```
In [105]: tupl1 = ('un', 'un', 'deux', 1, 3)
          list1 = [1, 1, 2]

          a = set(tupl1)
          b = set(list1)

          print(a, b)

          print("La chaîne 'un' est elle dans l'ensemble ?", 'un' in a)
          a.remove('un')
          print("La chaîne 'un' est-elle toujours dans l'ensemble ?", 'un' in a)

          print(a, b)

{'deux', 1, 'un', 3} {1, 2}
La chaîne 'un' est elle dans l'ensemble ? True
La chaîne 'un' est-elle toujours dans l'ensemble ? False
{'deux', 1, 3} {1, 2}
```

Des opérations supplémentaires sont possibles sur des ensembles. Elles sont calquées sur les opérations mathématiques :

- union
- intersection
- etc...

```
In [106]: print('intersection :', a & b)
          print('union :', a | b)

intersection : {1}
union : {1, 2, 3, 'deux'}
```

Pour plus d'informations sur les ensembles, voir [ici](#)

2.8 Fichiers

2.8.1 Ouverture

L'instruction :

```
f = open('interessant.txt', mode='r')
```

ouvre le fichier `interessant.txt` en mode lecture seule et le renvoie dans l'objet `f`.

- On peut spécifier un chemin d'accès complet ou relatif au répertoire courant
- Le caractère de séparation pour les répertoires peut être différent en fonction du système d'exploitation (/ pour unix et \ pour windows), voir le module `os.path` ou mieux la bibliothèque `pathlib`.

Modes d'ouverture communs

- 'r' : lecture seule
- 'w' : écriture seule
- 'a' : ajout à partir de la fin du fichier

Note : Avec 'w' et 'a', le fichier est créé s'il n'existe pas.

Pour plus d'informations sur les objets fichiers, voir [ici](#), pour la documentation de la fonction `open()`, voir [là](#).

2.8.2 Fermeture

On ferme le fichier `f` avec l'instruction :

```
f.close()
```

2.8.3 Méthodes de lecture

— `f.read()` : retourne tout le contenu de `f` sous la forme d'une chaîne de caractères.

```
In [107]: f = open('exos/interessant.txt', mode='r')
          texte = f.read()
          f.close()
          print(f'"texte" est un objet de type {type(texte)} de longueur {len(texte)} caractères:')
          print(texte)
          print('Contenu en raw string:')
          print(repr(texte))
          %pycat exos/interessant.txt
```

"texte" est un objet de type <class 'str'> de longueur 74 caractères:

Si vous lisez ce texte alors

...

vous savez lire un fichier avec Python !

Contenu en raw string:

'Si vous lisez ce texte alors\n...\nvous savez lire un fichier avec Python !\n'

— `f.readlines()` : retourne toutes les lignes de `f` sous la forme d'une liste de chaînes de caractères.

```
In [108]: f = open('exos/interessant.txt', mode='r')
          lignes = f.readlines()
          f.close()
          print(f'"lignes" est un objet de type {type(lignes)} contenant {len(lignes)} éléments:')
          print(lignes)
```

"lignes" est un objet de type <class 'list'> contenant 3 éléments:

['Si vous lisez ce texte alors\n', '...\n', 'vous savez lire un fichier avec Python !\n']

2.8.4 Méthodes d'écriture

— `f.write('du texte')` : écrit la chaîne 'du texte' dans `f`

```
In [109]: chaine = 'Je sais écrire\n...\navec Python !\n'
          # mode 'w' : on écrase le contenu du fichier s'il existe
          f = open('pas_mal.txt', mode='w')
          f.write(chaine)
          f.close()
          %pycat pas_mal.txt
```

Note : du point de vue du système, rien n'est écrit dans le fichier avant l'appel de `f.close()`

— `f.writelines(ma_sequence)` : écrit la séquence `ma_sequence` dans `f` en mettant bout à bout les éléments

```
In [110]: sequence = ['Je sais ajouter\n', 'du texte\n', 'avec Python !\n']
          f = open('pas_mal.txt', mode='a')
          # mode 'a' : on ajoute à la fin du fichier
          f.writelines(sequence)
          f.close()
          %pycat pas_mal.txt
```

Exercice :

1. écrire le contenu de la liste `mystere` dans le fichier `coded.txt` puis fermer ce dernier
2. lire le fichier `coded.txt` et le stocker dans une chaîne `coded`
3. Décoder la chaîne `coded` avec les instructions suivantes :

```
import codecs  
decoded = codecs.decode(coded, encoding='rot13')
```

4. écrire la chaîne `decoded` dans le fichier `decoded.txt` et fermer ce dernier
5. visualiser le contenu du fichier `decoded.txt` dans un éditeur de texte

```
In [111]: mystere = ['Gur Mra bs Clguba, ol Gvz Crgref\n\n',  
                    'Ornhgvshy vf orggre guna htyl.\n',  
                    'Rkcyvpvg vf orggre guna vzcypvg.\n']  
        # Votre code ci-dessous  
  
In [112]: # Décommentez la ligne suivante pour voir la solution  
        # %load exos/snippets/02_langage/chaîne_mystere.py
```


Chapitre 3

Opérations, contrôle, fonctions et modules



- Opérateurs
- Structures de contrôle
- Fonctions
- Exceptions & gestionnaires de contexte
- Compréhensions de listes & expressions génératrices
- Modules
- Bonnes pratiques

Contenu sous licence [CC BY-SA 4.0](#)

3.1 Opérateurs

3.1.1 Arithmétiques

`+, -, *, /, //, %, **`

sont des opérateurs classiques qui se comportent de façon habituelle.

Particularités de la division

```
In [1]: # Avec des nombres entiers
        print(16 / 3) # Quotient de la division euclidienne (produit un réel)
        print(16 // 3) # Quotient de la division euclidienne (produit un entier)
```

```

print(16 % 3) # Reste de la division euclidienne (produit un entier)

# Avec des nombres flottants
print(16.0 / 3) # Division (produit un réel)
print(16.0 // 3) # Quotient de la division (produit un réel)
print(16.0 % 3) # Reste de la division ou modulo (produit un réel)

```

```

5.333333333333333
5
1
5.333333333333333
5.0
1.0

```

Puissance

```

In [2]: print(2**10)
        # On peut aussi utiliser la fonction pow() du module math, mais celle-ci renvoie un réel.
        import math

        print(math.pow(2, 10))

1024
1024.0

```

3.1.2 Logiques

`and`, `or`, `not`

retournent une valeur booléenne.

```

In [3]: print(True or False)
        print(True and False)
        print(not True)
        print(not False)
        print(not [])
        print(not (1, 2, 3))

```

```

True
False
False
True
True
False

```

Attention, ce sont des opérateurs *court-circuit* :

```

In [4]: a = True
        b = False and a # b vaut False sans que a soit évalué
        c = True or a   # c vaut True, sans que a soit évalué

```

Pour s'en convaincre :

```

In [5]: True or print("nicht a kurz schluss")
        False and print("not a short circuit")
        print('on a prouvé que ce sont des opérateurs "court-circuit"...')

```

on a prouvé que ce sont des opérateurs "court-circuit"...

Exercice : Modifiez les valeurs `True` et `False` dans la cellule précédente, pour visualiser le fonctionnement de ces opérateurs.

3.1.3 Comparaison

python `==`, `is`, `!=`, `is not`, `>`, `>=`, `<`, `<=` L'évaluation de ces opérateurs retourne une valeur booléenne.

```
In [6]: print(2 == 2)
        print(2 != 2)
        print(2 == 2.0)
        print(type(2) is int)
```

True
False
True
True

On peut utiliser ces opérateurs avec des variables et des appels à des fonctions.

```
In [7]: x = 3
        print(1 > x)
        y = [0, 1, 42, 0]
        print(x <= max(y))
        print(x <= min(y))
```

False
True
False

On peut chaîner ces opérateurs, mais ils fonctionnent en mode *court-circuit* et l'opérande centrale n'est évaluée qu'une seule fois.

```
In [8]: x = 3
        print(2 < x <= 9)  # équivalent à 2 < x and x <= 9
```

True

Attention : comparer des types **non numériques** peut avoir des résultats surprenants.

```
In [9]: # Chaînes de caractères
        print("aaa" < "abc")
        print("aaa" < "aaaa")
        print("22" > "3")
```

True
True
False

```
In [10]: # Listes
        print([1, 2, 3, 4] > [42, 42])
        print([666] > [42, 42])
```

False
True

Attention : comparer des types **incompatibles** peut avoir des résultats surprenants.

```
In [11]: # Cette cellule génère des erreurs
print("chaîne:\t", "a" < 2)
print("liste:\t", ["zogzog"] > 42)
print("vide:\t", [] > 1)
print("tuple:\t", [23, 24] >= (23, 24))
print("dict:\t", [23, 24] >= {23: True, 24: "c'est pas faux"})
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[11], line 2
      1 # Cette cellule génère des erreurs
----> 2 print("chaîne:\t", "a" < 2)
      3 print("liste:\t", ["zogzog"] > 42)
      4 print("vide:\t", [] > 1)

TypeError: '<' not supported between instances of 'str' and 'int'
```

Attention, l'égalité de valeur n'implique pas forcément que l'identité des objets comparés est la même.

```
In [12]: a = [] # a est une liste vide
        b = [] # b est une AUTRE liste vide
```

```
In [13]: print(a == b) # test de valeur
print(a is b) # test d'identité
print(f"{id(a) = }")
print(f"{id(b) = }")
```

True
False
id(a) = 140298911160384
id(b) = 140298911265600

Mais, comme vu précédemment, des variables différentes peuvent référencer le même objet.

```
In [14]: c = a
print(a == c) # test de valeur
print(a is c) # test d'identité
```

True
True

3.1.4 bits à bits (*bitwise*)

python |, ^, &, <<, >>, ~ - Ces opérateurs permettent de manipuler individuellement les bits d'un entier. - Ce sont des opérations bas-niveau, souvent utilisées pour piloter directement du matériel, pour implémenter des protocoles de communication binaires (par exemple réseau ou disque). - L'utilisation d'entiers comme ensemble de bits permet des encodages de données très compacts, un booléen (True, False) ne prendrait qu'un bit en mémoire, c'est à dire que l'on peut encoder 64 booléens dans un entier.

Description complète [ici](#).

```
In [15]: val = 67 # == 64 + 2 + 1 == 2**6 + 2**1 + 2**0 == 0b1000011
print(bin(val))

mask = 1 << 0 # On veut récupérer le 1er bit
print("le 1er bit vaut", (val & mask) >> 0)

mask = 1 << 1 # On veut récupérer le 2ème bit
print("le 2ème bit vaut", (val & mask) >> 1)

mask = 1 << 2 # On veut récupérer le 3ème bit
print("le 3ème bit vaut", (val & mask) >> 2)

mask = 1 << 6 # On veut récupérer le 7ème bit
print("le 7ème bit vaut", (val & mask) >> 6)

# Si on positionne le 4ème bit a 1 (on rajoute 2**3 = 8)
newval = val | (1 << 3)
print(newval)

# Si on positionne le 6ème bit a 0 (on soustrait 2**7 = 64)
print(newval & ~(1 << 6))

0b1000011
le 1er bit vaut 1
le 2ème bit vaut 1
le 3ème bit vaut 0
le 7ème bit vaut 1
75
11
```

Exercice : Retournez une chaîne de caractères représentant le nombre contenu dans x écrit en notation binaire. Par exemple :

$$5 \rightarrow 1016 \rightarrow 1107 \rightarrow 111$$

```
In [16]: x = 7
# votre code ici

In [17]: # Décommentez puis exécutez pour afficher le corrigé :
# %load exos/snippets/03_langage/bits_a_bits.py
```

3.1.5 Affectation augmentée

python +=, -=, *=, /=, **= # etc.

```
In [18]: a = 4
a += 1 # <=> a = a + 1
print(a)
a /= 2
print(a)
a **= 3
print(a)
a %= 2
print(a)
```

5
2

8
0

3.1.6 Compatibilité de type, coercition de type

Python effectue certaines conversions implicites quand on ne perd pas d'information (par exemple d'entier vers flottant).

```
In [19]: print(1 + 0b1)
          print(1 + 1.0)
          print(1.0 + 2 + 0b11 + 4j)
```

2
2.0
(6+4j)

Mais dans d'autres cas, la conversion doit être explicite.

```
In [20]: # Cette cellule génère une erreur
          a = 1
          b = "1"
          a + b
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[20], line 4
      2 a = 1
      3 b = "1"
----> 4 a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Exercice :

Sans toucher au +, corrigez la ligne 4 de la cellule ci-dessus afin d'afficher :

1. la chaîne '11'
2. l'entier 2
3. l'entier 11

```
In [21]: a = 1
          b = "1"
          a + b
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[21], line 3
      1 a = 1
      2 b = "1"
----> 3 a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
In [22]: # Décommentez puis exécutez pour afficher le corrigé :
          # %load exos/snippets/03_langage/compatibilite.py
```

3.1.7 Priorité des opérateurs

Les opérateurs ont des priorités classiques en python.

Par exemple, dans l'ordre :

1. puissance : ******
2. multiplication, division : ***** et **/**
3. addition, soustraction : **+** et **-**

etc.

Utilisez des parenthèses quand elles aident à la lisibilité et à la clarté.

Pour plus d'informations, voir [ici](#).

3.2 Structures de contrôle

- La mise en page comme syntaxe
- **pass**
- tests conditionnels : **if/elif/else**
- boucles
 - **for** <element> in <iterable>
 - **while**
 - **break**
 - **continue**

3.2.1 La mise en page comme syntaxe

- La mise en page est importante en python, c'est une différence majeure avec les autres langages (Java, C/C++, etc.)
- Python utilise l'indentation du code avec des caractères blancs plutôt que des mots clés (**begin/end** en pascal) ou des symboles (**{}** en java et C/C++). Cela permet de rendre le code plus compact.
- Elle va servir à délimiter des blocs de code sur lesquels les structures de contrôle comme les boucles ou les tests de conditions vont s'appliquer.
- De toute façon, dans les autres langages, on indente aussi le code pour l'aspect visuel et la lisibilité.
- L'indentation faisant partie de la syntaxe du langage, il faut être rigoureux et suivre la règle suivante :
 - 4 espaces pour passer au niveau suivant
 - éviter les tabulations et préférer les espaces et surtout ne pas les mélanger !

```
In [23]: if True:
          print("toutes")
          print("les")
          print("lignes")
          print("au même niveau d'indentation forment un bloc de code")
          print('et quand on remonte, on "termine" un bloc de code')
```

```
toutes
les
lignes
au même niveau d'indentation forment un bloc de code
et quand on remonte, on "termine" un bloc de code
```

Exercice : changez le **True** en **False**, et observez quelles lignes de code ne sont plus exécutées.

3.2.2 **pass**

En cas de besoin d'un bloc de code qui ne fait rien, on utilise le mot clé **pass** (équivalent à NOP ou NO-OP)

Exemple : une boucle infinie

```
while True:
    pass
```

(à ne pas exécuter dans une cellule sous peine de bloquer le noyau de ce notebook)

3.2.3 Tests conditionnels

Les instructions if/elif/else permettent d'exécuter des blocs d'instructions en fonction de conditions :

```
if test1:
    <bloc d instructions 1>
elif test2:
    <bloc d instructions 2>
else:
    <bloc d instructions 3>
```

elif est la contraction de "else if".

```
In [24]: if True:
        print("c'est vrai!")
```

c'est vrai!

```
In [25]: if False:
        print("je suis caché!")
    else:
        print("mais moi je suis en pleine lumière...")
```

mais moi je suis en pleine lumière...

```
In [26]: # Pour cet exemple, on itère sur les éléments d'un tuple (cf. boucle for plus loin)
        for position in 2, 9, 3, 1, 8:
            if position == 1:
                print(position, "Or")
            elif position == 2:
                print(position, "Argent")
            elif position == 3:
                print(position, "Bronze")
            else:
                print(position, "Vestiaires")
```

```
2 Argent
9 Vestiaires
3 Bronze
1 Or
8 Vestiaires
```

```
In [27]: taille = 1.50
        if taille >= 1.70: # La taille moyenne en France
            print("grand")
        else:
            print("petit")
```

petit

Exercices :

1. Editez la cellule pour y mettre votre taille et exécutez-la pour savoir si vous êtes grand ou petit.
2. Gérez le cas des gens de taille moyenne.

In [28]: *# Décommentez puis exécutez pour afficher le corrigé :*
#!/load exos/snippets/03_langage/condition.py

3.3 Boucles

Les boucles sont les structures de contrôle permettant de répéter l'exécution d'un bloc de code plusieurs fois.

3.3.1 Boucle while

La plus simple est la boucle de type `while` :

```
while <condition>:
    <bloc 1>
<bloc 2>
```

Tant que `<condition>` est `True`, le `<bloc 1>` est exécuté, quand la condition passe à `False`, l'exécution continue au `<bloc 2>`.

```
In [29]: compteur = 3
        while compteur > 0:
            print("le compteur vaut :", compteur)
            compteur -= 1
        print("le compteur a été décrémenté 3 fois et vaut maintenant", compteur)
```

```
le compteur vaut : 3
le compteur vaut : 2
le compteur vaut : 1
le compteur a été décrémenté 3 fois et vaut maintenant 0
```

Exercice :

C'est la soirée du réveillon. Ecrivez une boucle `while` qui décompte les secondes à partir de 5 pour souhaiter la bonne année.
 Allez voir [par ici](#) pour passer le temps...

```
In [30]: import time
        # Votre code ici
```

In [31]: *# Décommentez puis exécutez pour afficher le corrigé :*
#!/load exos/snippets/03_langage/while.py

3.3.2 Boucle for

Une boucle plus complexe : `for/in`

```
for <variable> in <iterable>:
    <bloc 1>
<bloc 2>
```

A chaque tour de boucle, la variable `<variable>` va référencer un des éléments de l'`<iterable>`. La boucle s'arrête quand tous les éléments de l'itérable ont été traités. Il est fortement déconseillé de modifier l'itérable en question dans le `<bloc 1>`.

```
In [32]: invites = ("Aline", "Bernard", "Céline", "Dédé")
         for invite in invites:
             print(f"Bonjour {invite}, bienvenue à la soirée de gala !")
             print("Maintenant, tout le monde a été bien accueilli.")
```

```
Bonjour Aline, bienvenue à la soirée de gala !
Bonjour Bernard, bienvenue à la soirée de gala !
Bonjour Céline, bienvenue à la soirée de gala !
Bonjour Dédé, bienvenue à la soirée de gala !
Maintenant, tout le monde a été bien accueilli.
```

Exercice : Rajoutez des invités à la fête. Vérifiez que tout le monde est accueilli correctement.

```
In [33]: # Maintenant, si nous avons reçu des réponses à notre invitation, utilisons un dictionnaire :
         invites = {"Aline": True, "Bernard": False, "Céline": True, "Dédé": True}
         for personne, presence in invites.items():
             if presence:
                 print(f"Bonjour {personne}, bienvenue à la soirée de gala !")
             else:
                 print(f"Malheureusement, {personne} ne sera pas avec nous ce soir.")
         print("Maintenant, tout le monde a été bien accueilli ou excusé.")
```

```
Bonjour Aline, bienvenue à la soirée de gala !
Malheureusement, Bernard ne sera pas avec nous ce soir.
Bonjour Céline, bienvenue à la soirée de gala !
Bonjour Dédé, bienvenue à la soirée de gala !
Maintenant, tout le monde a été bien accueilli ou excusé.
```

La méthode `.items()` renvoie une vue itérable des éléments du dictionnaire.

Exercice : Rajoutez des invités à la fête. Certains ayant répondu qu'ils ne pourraient pas venir. Vérifiez que tout le monde est accueilli ou excusé correctement.

Si on souhaite itérer sur une liste et que l'on a besoin de l'indice de ses éléments, on utilise une combinaison de `range()` et `len()`.

```
In [34]: nombres = [2, 4, 8, 6, 8, 1, 0, 1j]
         for i in range(len(nombres)):
             nombres[i] **= 2

         print("valeurs de i:", list(range(len(nombres))))
         print("carrés      :", nombres)
```

```
valeurs de i: [0, 1, 2, 3, 4, 5, 6, 7]
carrés      : [4, 16, 64, 36, 64, 1, 0, (-1+0j)]
```

Il existe une forme raccourcie pour faire ce genre de choses, la fonction interne `enumerate()`

```
In [35]: nombres = [2, 4, 8, 6, 8, 1, 0]
         impairs = nombres[:] # On copie la liste nombres
         for i, nombre in enumerate(nombres):
             impairs[i] = bool(nombre % 2)
         # Les impairs
         print(impairs)
```

```
[False, False, False, False, False, True, False]
```

Note

La fonction interne `range()` retourne un itérateur. C'est un objet qui se comporte comme une liste sans pour autant allouer la mémoire nécessaire au stockage de tous ses éléments. Le coût de création d'une vraie liste augmente avec sa taille (son empreinte mémoire aussi!).

- une liste (ou un tuple) contient des données
- un itérateur possède la méthode qui permet de calculer l'élément suivant dans une boucle `for`.

Note de la note

En Python version 2.x, Il existait deux versions de cette fonctionnalité : `range()` et `xrange()`. La première renvoyait une vraie liste, allouée complètement, alors que `xrange()` renvoyait un itérateur.

```
In [36]: print(type(range(3))) # comme xrange() en python2
         print(repr(range(3))) # comme xrange() en python2
         print(type(list(range(3)))) # comme range en python2
         print(repr(list(range(3)))) # comme range en python2
```

```
<class 'range'>
range(0, 3)
<class 'list'>
[0, 1, 2]
```

3.3.3 Instruction break

Il est possible d'arrêter prématurément une boucle grâce à l'instruction `break`.

L'instruction `break` est utilisable indifféremment dans les boucles `for` ou `while`.

```
In [37]: compteur = 3
         while True: # Notre boucle infinie
             compteur -= 1
             print(f"Dans la boucle infinie! {compteur = }")
             if compteur <= 0:
                 break # On sort de la boucle while immédiatement
             print(f"On continue, {compteur = }")
             print("C'était pas vraiment une boucle infinie...")
```

```
Dans la boucle infinie! compteur = 2
On continue, compteur = 2
Dans la boucle infinie! compteur = 1
On continue, compteur = 1
Dans la boucle infinie! compteur = 0
C'était pas vraiment une boucle infinie...
```

En cas d'imbrication de plusieurs boucles, l'instruction `break` sort de la plus imbriquée (la plus proche).

```
In [38]: for i in (1, 2, 3):
         for j in (1, 2, 3, 4):
             if i == 2:
                 break
             print(f"{i, j = }")
```

```
i, j = (1, 1)
i, j = (1, 2)
i, j = (1, 3)
```

```

i, j = (1, 4)
i, j = (3, 1)
i, j = (3, 2)
i, j = (3, 3)
i, j = (3, 4)

```

3.3.4 Instruction continue

Si, dans une boucle, on veut passer immédiatement à l'itération suivante, on utilise l'instruction `continue`.

```

In [39]: compteur = 9
         while compteur > 0:
             compteur -= 1
             if compteur % 2:
                 compteur /= 2
                 print("impair, on divise :", compteur)
                 continue # retourne immédiatement au début de la boucle
             print("pair, RAS")
         print("c'est fini...")

pair, RAS
impair, on divise : 3.5
impair, on divise : 1.25
impair, on divise : 0.125
impair, on divise : -0.4375
c'est fini...

```

3.4 Fonctions

Les fonctions permettent de réutiliser des blocs de code à plusieurs endroits différents sans avoir à copier ce bloc.

En python, il n'y a pas de notion de sous-routine. Les procédures sont gérées par les objets de type fonctions, avec ou sans valeur de retour.

```

def <nom fonction>(arg1, arg2, ...):
    <bloc d instructions>
    return <valeur> # Instruction optionnelle

```

On distingue :

- les fonctions avec `return` des fonctions sans `return`
- les fonctions sans arguments (pour lesquelles () est vide) des fonctions avec arguments (`arg1, arg2, ...`)

`<nom fonction>(arg1, arg2, ...)` est appelé **signature** de la fonction.

3.4.1 Fonctions sans arguments

Fonctions sans return

Pour définir une fonction :

```

In [40]: def func(): # Definition de la fonction
         print("You know what?")
         print("I'm happy!")

```

Pour utiliser une fonction que l'on a défini :

```
In [41]: func()  # 1er Appel de la fonction
         func()  # 2eme appel
         func()  # 3eme appel, etc...
         print(func())  # On l'appelle et on affiche sa valeur de retour
```

```
You know what?
I'm happy!
You know what?
I'm happy!
You know what?
I'm happy!
You know what?
I'm happy!
None
```

Exercice : Ecrivez une fonction nommée `rien` qui ne fait rien et appelez là deux fois.

```
In [42]: # Votre code ici

In [43]: # Décommentez puis exécutez pour afficher le corrigé:
         # %load exos/snippets/03_langage/rien.py
```

Fonctions avec return

```
In [44]: def func():  # Definition de la fonction
         return "I'm happy"  # La fonction retourne une chaine de caractère

         print("1er appel:")
         func()  # 1er Appel de la fonction : la valeur retournée n'est pas utilisée
         print("2eme appel:")
         ret_val = func()  # Le retour du 2eme appel est stocké
         print("La fonction func() nous a renvoyé la valeur:", ret_val)

1er appel:
2eme appel:
La fonction func() nous a renvoyé la valeur: I'm happy
```

Exercice : Ecrivez une fonction nommée `elogie_de_rien` qui retourne la chaine de caractères `rien`. Appelez-la et affichez sa valeur de retour.

```
In [45]: # Votre code ici

In [46]: # Décommentez puis exécutez pour afficher le corrigé:
         # %load exos/snippets/03_langage/elogie_de_rien.py
```

Important : l'instruction `return` provoque une sortie de la fonction. Dans le code suivant, la ligne qui appelle la fonction `print()` n'est pas exécutée.

```
In [47]: def func():
         return "je sors"
         print("après return")

         func()

Out[47]: 'je sors'
```

3.4.2 Fonctions avec arguments

Pour définir une fonction qui prend des arguments, on nomme ces arguments entre les parenthèses de la ligne `def`. Ces paramètres seront définis comme des variables à l'intérieur de la fonction et recevrons les valeurs passées lors des appels de celle-ci.

```
In [48]: def somme(x, y):
         return x + y
```

Pour utiliser cette fonction avec diverses valeurs, il suffit de l'appeler plusieurs fois :

```
In [49]: print(somme(1, 2))
         print(somme(4, 7))
         print(somme(2 + 2, 7))
         print(somme(somme(2, 2), 7))
```

```
3
11
11
11
```

Exercice : - Définissez une fonction nommée `chevalier` qui prend un paramètre `n` et qui affiche `n` fois (avec `print()`) la chaîne de caractères `Nee!` - appelez cette fonction pour vérifier que `chevalier(3)` dit bien `Nee!` trois fois comme il convient !

Voici quelques exemples montrant comment cette fonction doit se comporter :

```
chevalier(1)
Nee!
chevalier(3)
Nee!Nee!Nee!
chevalier(6)
Nee!Nee!Nee!Nee!Nee!Nee!
```

```
In [50]: def chevalier(n):
         # Votre code ici
         pass
```

```
In [51]: # Vérifions que tout fonctionne bien:
         chevalier(1)
         chevalier(3)
         chevalier(6)
```

```
In [52]: # Décommentez puis exécutez pour afficher le corrigé:
         # %load exos/snippets/03_langage/chevalier.py
```

Exercice : Ecrivez une autre fonction, nommée `chevalier_ret` : - qui prend 2 paramètres : un entier `n` et un booléen - qui retourne une chaîne de caractères de la chaîne `nee!` ou `NEE!` en fonction du paramètre booléen, chaîne répétée `n` fois.

Appelez cette fonction et affichez sa valeur de retour.

Voici quelques exemples montrant comment cette fonction doit se comporter :

```
a = chevalier_ret(1, True)
print(a)
NEE!
print(chevalier_ret(3, False))
nee!nee!nee!
print(chevalier_ret(6, True))
NEE!NEE!NEE!NEE!NEE!NEE!
```

```
In [53]: # Votre code ici
def chevalier_ret(n: int, cri: bool):
    return n * ("NEE!" if cri else "nee!")

In [54]: # Vérifions que tout fonctionne bien:
a = chevalier_ret(1, True)
print(a)
a = chevalier_ret(3, False)
print(a)
a = chevalier_ret(6, True)
print(a)

NEE!
nee!nee!nee!
NEE!NEE!NEE!NEE!NEE!NEE!

In [55]: # Décommentez puis exécutez pour afficher le corrigé:
# %load exos/snippets/03_langage/chevalier_ret.py
```

Utilisation de valeurs par défaut

```
In [56]: def somme(x, y=1):
    return x + y

print(somme(1, 2))
```

3

Si sa valeur n'est pas spécifiée lors de l'appel, le paramètre y prend la valeur par défaut (ici : 1)

```
In [57]: print(somme(4))
```

5

Note : Les arguments ayant une valeur par défaut doivent être placés **en dernier**.

Utilisation des arguments par leur nom

Si les arguments sont explicitement nommés lors de l'appel, leur ordre peut être changé :

```
In [58]: def diff(x, y):
    return x - y

print(diff(4, 7))
print(diff(y=7, x=4))
```

-3

-3

Capture d'arguments non définis

Arguments positionnels dans un tuple On définit une fonction dont l'argument est `*args` :

```
In [59]: def fonc(*args):
         # args est un tuple :
         print(type(args))
         # ses éléments sont les arguments passés lors de l'appel :
         print(args)
```

On l'appelle avec n'importe quelle séquence d'arguments :

```
In [60]: fonc("n'importe", "quel nombre et type de", "paramètres", 5, [1, "toto"], None)

<class 'tuple'>
('n'importe', 'quel nombre et type de', 'paramètres', 5, [1, 'toto'], None)
```

Arguments nommés dans un dictionnaire On définit une fonction dont l'argument est `**kwargs` :

```
In [61]: def fonc(**kwargs):
         # kwargs est un dictionnaire :
         print(type(kwargs))
         # ses éléments sont les arguments nommés passés lors de l'appel
         print(kwargs)
```

On l'appelle en nommant les arguments :

```
In [62]: fonc(x=1, y=2, couleur="rouge", epaisseur=2)

<class 'dict'>
{'x': 1, 'y': 2, 'couleur': 'rouge', 'epaisseur': 2}
```

On peut combiner ce type d'arguments pour une même fonction :

```
In [63]: def fonc(n, *args, **kwargs): # cet ordre est important
         print(f"{n = }")
         print(f"{args = }")
         print(f"{kwargs = }")

         print("appel 1")
         fonc(3)
         print("appel 2")
         fonc(3, "bacon")
         print("appel 3")
         fonc(2, "spam", "egg", x=1, y=2, couleur="rouge", epaisseur=2)
```

```
appel 1
n = 3
args = ()
kwargs = {}
appel 2
n = 3
args = ('bacon',)
kwargs = {}
appel 3
n = 2
args = ('spam', 'egg')
kwargs = {'x': 1, 'y': 2, 'couleur': 'rouge', 'epaisseur': 2}
```


Remarques :

- les noms `args` et `kwargs` ne sont que des conventions (à respecter, toutefois!), seul le caractère `*` est déterminant
- l'ordre (`arg1`, `arg2`, ..., `*args`, `**kwargs`) doit être strictement respecté

Packing/unpacking

- La syntaxe `*args` dans la définition de la fonction correspond à une opération de *packing* : Python transforme une séquence de variables en tuple.
- L'inverse existe : ça s'appelle l'*unpacking*.
- Le *packing/unpacking* se pratique déjà par la manipulation des tuples :

```
In [64]: trio = "sax", "drums", "bass" # packing
          print(f"{trio = }")
          you, her, him = trio # unpacking
          print(f"{you = }")
          print(f"{her = }")
          print(f"{him = }")
```

```
trio = ('sax', 'drums', 'bass')
you = 'sax'
her = 'drums'
him = 'bass'
```

Il peut également se pratiquer dans le passage d'arguments de fonction

```
In [65]: def fonc(*args):
          print(args)

          fonc(you, her, him) # ici on liste directement les arguments
          fonc(*trio) # là on "unpack" un tuple

('sax', 'drums', 'bass')
('sax', 'drums', 'bass')
```

Et de la même façon pour un dictionnaire :

```
In [66]: def fonc(**kwargs):
          print(kwargs)

          trio_dict = {"sax": "you", "drums": "her", "bass": "him"}
          fonc(
              sax="you", drums="her", bass="him"
          ) # ici on liste directement les arguments nommés
          fonc(**trio_dict) # là on "unpack" un dictionnaire

{'sax': 'you', 'drums': 'her', 'bass': 'him'}
{'sax': 'you', 'drums': 'her', 'bass': 'him'}
```

3.4.3 Espace de nommage et portée des variables

1er exemple

On veut illustrer le mécanisme de l'espace de nommage des variables :

```
In [67]: def func1():
          a = 1
          print(f"Dans func1(), {a = }")

          def func2():
              print(f"Dans func2(), {a = }")

          a = 2
          func1()
          func2()
          print(f"Dans l'espace englobant, {a = }")
```

```
Dans func1(), a = 1
Dans func2(), a = 2
Dans l'espace englobant, a = 2
```

Cet exemple montre deux comportements :

1. Une variable définie localement à l'intérieur d'une fonction cache une variable du même nom définie dans l'espace englobant (cas de `func1()`).
2. Quand une variable n'est pas définie localement à l'intérieur d'une fonction, Python va chercher sa valeur dans l'espace englobant (cas de `func2()`).

2ème exemple

On veut illustrer le mécanisme de portée des variables au sein des fonctions :

```
In [68]: def func():
          a = 1
          bbb = 2
          print(f"Dans func() : {a = }")

          a = 2
          print(f"Avant func() : {a = }")
          func()
          print(f"Après func() : {a = }")
```

```
Avant func() : a = 2
Dans func() : a = 1
Après func() : a = 2
```

Les variables définies localement à l'intérieur d'une fonction sont détruites à la sortie de cette fonction. Ici, la variable `bbb` n'existe pas hors de la fonction `func()`, donc Python renvoie une erreur si on essaye d'utiliser `bbb` depuis l'espace englobant :

```
In [69]: # Cette cellule génère une erreur
          print(bbb)
```

```

-----
NameError                                Traceback (most recent call last)
Cell In[69], line 2
      1 # Cette cellule génère une erreur
----> 2 print(bbb)

NameError: name 'bbb' is not defined

```

3.4.4 Fonctions *built-in*

Ces fonctions sont disponibles dans tous les contextes. La liste complète est détaillée [ici](#). En voici une sélection :

- `dir(obj)` : retourne une liste de toutes les méthodes et attributs de l'objet `obj`
- `dir()` : retourne une liste de tous les objets du contexte courant
- `eval(expr)` : analyse et exécute la chaîne de caractère `expr`

```

In [70]: a = 1
          b = eval("a + 1")
          print(f"type(b) = {type(b)} ; {b} = {b}")

```

```

type(b) = <class 'int'> ; b = 2

```

- `globals()` : retourne un dictionnaire des variables présentes dans le contexte global
- `locals()` : idem `globals()` mais avec le contexte local
- `help(obj)` : affiche l'aide au sujet d'un objet
- `help()` : affiche l'aide générale (s'appelle depuis l'interpréteur interactif)
- `input(prompt)` : retourne une chaîne de caractère lue dans la console après le message `prompt`

```

In [71]: reponse = input("Ca va ? ")

if reponse.lower() in ("o", "oui", "yes", "y", "ok", "da", "jawohl", "ja"):
    print("Supercalifragilisticexpialidocious")
else:
    print("Faut prendre des vacances...")

```

```

-----
StdinNotImplementedError                Traceback (most recent call last)
Cell In[71], line 1
----> 1 reponse = input("Ca va ? ")
      3 if reponse.lower() in ("o", "oui", "yes", "y", "ok", "da", "jawohl", "ja"):
      4     print("Supercalifragilisticexpialidocious")

File /opt/conda/lib/python3.11/site-packages/ipykernel/kernelbase.py:1201, in Kernel._input(self, prompt, *, parent_ident, parent_password, password)
    1199 if not self._allow_stdin:
    1200     msg = "raw_input was called, but this frontend does not support input requests."
-> 1201     raise StdinNotImplementedError(msg)
    1202 return self._input_request(
    1203     str(prompt),
    1204     self._parent_ident["shell"],
    1205     self.get_parent("shell"),
    1206     password=False,
    1207 )

```

```
StdinNotImplementedError: raw_input was called, but this frontend does not support input requests.
```

- `len(seq)` : retourne la longueur de la séquence `seq`
- `max(seq)` : retourne le maximum de la séquence `seq`
- `min(seq)` : retourne le minimum de la séquence `seq`
- `range([start=0], stop[, step=1])` : retourne un itérateur d'entiers allant de `start` à `stop - 1`, par pas de `step`.

```
In [72]: print(list(range(10)))
         print(list(range(5, 10, 2)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 7, 9]
```

- `repr(obj)` : affiche la représentation de l'objet `obj`.
- `reversed(seq)` : retourne l'inverse de la séquence `seq`
- `sorted(seq)` : retourne une séquence triée à partir de la séquence `seq`
- `sum(seq)` : retourne la somme des éléments de la séquence `seq`

3.4.5 Exercices sur les fonctions

Exercice 1

Ecrire une fonction `stat()` qui prend en argument une liste d'entiers et retourne un tuple contenant : - la somme - le minimum - le maximum

des éléments de la liste

```
In [73]: def stat(a_list: list[int]) -> tuple[int]:
         # votre fonction
         pass

In [74]: # Décommentez puis exécutez pour afficher le corrigé:
         # %load exos/snippets/03_langage/stat.py
```

Exercice 2 : écriture d'un *wrapper* de fonction
Noël arrive vite... Amusez-vous avec les boules de décoration!



Soit une fonction `boule()` capable d'accrocher une boule de couleur à la position (x, y) d'un sapin.

Exercice inspiré du Mooc de l'INRIA [Python : des fondamentaux à l'utilisation du langage](#)

```
In [75]: def boule(x, y, couleur="bleue"):
          print(f"J'accroche une boule en ({x}, {y}) de couleur {couleur}")

          # On place la première boule sur le sapin
          boule(1, 2)
          # Puis une autre, etc.
          boule(3, 4, couleur="rouge")
```

J'accroche une boule en (1, 2) de couleur bleue

J'accroche une boule en (3, 4) de couleur rouge

Ecrire une fonction *wrapper* `boule_or()` qui crée des boules dorées en appelant la fonction `boule()`. Dans le futur, on souhaite modifier la fonction `boule()` pour lui faire accepter un nouvel argument `rendu` (`brillant`, `mat`, etc.). La fonction `boule_or()` devra continuer à fonctionner après cette modification de la fonction `boule()` et intégrer la nouvelle fonctionnalité `rendu` sans qu'il soit nécessaire de modifier `boule_or()`.

```
In [76]: # Cellule à modifier
          def boule_or(*args, **kwargs):
              # Votre code ici
              pass

          # On place une boule en or sur le sapin
          boule_or(1, 2)
```

Maintenant, on met à jour la fonction `boule()` :

```
In [77]: def boule(x, y, couleur="bleue", rendu="mat"):
          print(
              f"J'accroche une boule en ({x}, {y}) de couleur {couleur} et de rendu {rendu}."
          )

          boule(1, 3, couleur="jaune", rendu="brillant")
```

J'accroche une boule en (1, 3) de couleur jaune et de rendu brillant.

Vérifier que votre fonction `boule_or()` marche encore et gère la nouvelle fonctionnalité :

```
In [78]: boule_or(3, 1, rendu="brillant") # doit retourner :
          # J'accroche une boule en (3, 1) de couleur or et de rendu brillant.

In [79]: # Décommentez puis exécutez pour afficher le corrigé:
          # %load exos/snippets/03_langage/boule.py
```

3.5 Exceptions

Pour signaler des conditions particulières (erreurs, événements exceptionnels), Python utilise un mécanisme de levée d'exceptions.

```
In [80]: # Cette cellule génère une erreur
          raise Exception
```

```
-----
Exception                                Traceback (most recent call last)
Cell In[80], line 2
      1 # Cette cellule génère une erreur
----> 2 raise Exception

Exception:
```

Ces exceptions peuvent embarquer des données permettant d'identifier l'événement producteur.

```
In [81]: # Cette cellule génère une erreur
          raise Exception("Y a une erreur")
```

```
-----
Exception                                Traceback (most recent call last)
Cell In[81], line 2
      1 # Cette cellule génère une erreur
----> 2 raise Exception("Y a une erreur")

Exception: Y a une erreur
```

La levée d'une exception interrompt le cours normal de l'exécution du code et "remonte" jusqu'à l'endroit le plus proche gérant cette exception.

Pour intercepter les exceptions, on écrit :

```
try:
    <bloc de code 1>
except Exception:
    <bloc de code 2>
```

```
In [82]: try:
        print("ici ca fonctionne")
        # ici on détecte une condition exceptionnelle, on signale une exception
        raise Exception("y a un bug")
        print("on n'arrive jamais ici")
    except Exception as e:
        # L'exécution continue ici
        print(
            f"ici on peut essayer de corriger le problème lié à l'exception : Exception({e})"
        )
    print("et après, ça continue ici")
```

```
ici ca fonctionne
ici on peut essayer de corriger le problème lié à l'exception : Exception(y a un bug)
et après, ça continue ici
```

Exemple illustrant le mécanisme de remontée des exceptions d'un bloc à l'autre :

```
In [83]: def a():
        raise Exception("coucou de a()")

        def b():
            print("b() commence")
            a()
            print("b() finit")

        try:
            b()
        except Exception as e:
            print("l'exception vous envoie le message :", e)

b() commence
l'exception vous envoie le message : coucou de a()
```

3.5.1 Exercice

Ecrivez une fonction `openfile()` :

- qui demande à l'utilisateur un nom de fichier à ouvrir
- qui gère correctement les fichiers inexistants.
- qui affiche la première ligne du fichier ouvert
- qui retourne une valeur booléenne indiquant que le fichier a été ouvert ou non.

```
In [84]: # Votre code ici
        def openfile(filename: str):
            try:
                f = open(filename, 'r')
                print(f.readline())
                f.close()
                return True
```

```

except FileNotFoundError:
    return False

print(openfile("exos/interessant.txt"))

```

Si vous lisez ce texte alors

True

```

In [85]: # Décommentez puis exécutez pour afficher le corrigé:
        # %load exos/snippets/03_langage/openfile.py

```

Pour plus d'informations sur les exceptions, lire ce [tutoriel](#).

3.6 Les gestionnaires de contexte

Pour faciliter la gestion des obligations liées à la libération de ressources, la fermeture de fichiers, etc... Python propose des gestionnaires de contexte introduits par le mot-clé `with`.

```

In [86]: with open("exos/interessant.txt", "r") as fichier_ouvert:
        # Dans ce bloc de code le fichier est ouvert en lecture, on peut l'utiliser normalement
        print(fichier_ouvert.read())
        # Ici, on est sorti du bloc et du contexte : le fichier a été fermé automatiquement

```

Si vous lisez ce texte alors

...

vous savez lire un fichier avec Python !

```

In [87]: # Cette cellule génère une erreur
        print(fichier_ouvert.read())

```

```

-----
ValueError                                Traceback (most recent call last)
Cell In[87], line 2
      1 # Cette cellule génère une erreur
----> 2 print(fichier_ouvert.read())

ValueError: I/O operation on closed file.

```

Exercice : Reprenez le code de l'exercice précédent, et utilisez `with` pour ne pas avoir à utiliser la méthode `close()`.

```

In [88]: # Votre code ici

```

```

In [89]: # Décommentez puis exécutez pour afficher le corrigé:
        # %load exos/snippets/03_langage/openfile2.py

```

Il est possible de créer de nouveaux gestionnaires de contexte, pour que vos objets puissent être utilisés avec `with` et que les ressources associées soient correctement libérées.

Pour plus d'informations sur la création de gestionnaires de contexte, voir [la documentation](#).

3.7 Les compréhensions de listes

Python a introduit une facilité d'écriture pour les listes qui permet de rendre le code plus lisible car plus concis.

On construit par exemple la liste `[0, 1, 2, ..., 9]` :

```
In [90]: Liste1 = list(range(10))
         print(Liste1)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On veut maintenant une liste ne contenant que les éléments pairs de la liste `Liste1`.

```
In [91]: ListePaire = []
         for i in Liste1:
             if (i % 2) == 0:
                 ListePaire.append(i)
         print(ListePaire)
```

```
[0, 2, 4, 6, 8]
```

À présent, on fait la même chose en compréhension de liste

```
In [92]: ListePaire = [i for i in Liste1 if (i % 2) == 0]
         print(ListePaire)
```

```
[0, 2, 4, 6, 8]
```

Cette concision peut être utile, mais n'en abusez pas, si vous commencez à avoir une compréhension de liste trop complexe à écrire en une simple ligne, utilisez plutôt des boucles et conditions explicites. Plus d'informations dans [ce tutoriel](#).

3.8 Les expressions génératrices

C'est une forme d'écriture, très proche des compréhensions de listes, mais qui ne crée pas de nouvel objet liste immédiatement. Les items sont produits à la demande, plus loin dans le code, là où ils sont nécessaires, ce qui économise du temps et de la mémoire.

```
In [93]: generateur_paires = (i for i in Liste1 if (i % 2) == 0)
```

Le générateur ne contient pas de données à proprement parler :

```
In [94]: print(generateur_paires)
```

```
<generator object <genexpr> at 0x7f99e2c1eb50>
```

Pour visualiser son comportement, on peut l'utiliser pour créer une liste :

```
In [95]: print(list(generateur_paires))
```

```
[0, 2, 4, 6, 8]
```

Plus d'informations dans [ce tutoriel](#).

3.9 Modules

- Python fournit un système de modularisation du code qui permet d'organiser un projet contenant de grandes quantités de code et de réutiliser et de partager ce code entre plusieurs applications.
- L'instruction `import` permet d'accéder à du code situé dans d'autres fichiers. Cela inclut les nombreux modules de la bibliothèque standard, tout comme vos propres fichiers contenant du code.
- Les objets (variables, fonctions, classes, etc.) du module sont accessibles de la manière suivante :

```
module.variable
module.fonction(parametre1, parametre2, ...)
```

Ou plus généralement :

```
module.attribut
```

```
In [96]: # Pour utiliser les fonctions mathématiques du module 'math'
import math

print(f"{math.pi:.2f}")
print(f"{math.sin(math.pi):.2f}")
```

```
3.14
0.00
```

3.9.1 Créer ses propres modules

- Il suffit de placer votre code dans un fichier avec l'extension `.py`
- Le module stocké dans le fichier `mon_module.py` s'importe avec la syntaxe :

```
import mon_module
```

Cas 1 : le fichier module est directement importable

- **Cas 1.a :** le module est déjà installé dans l'environnement d'exécution
 - soit il fait partie de la bibliothèque standard
 - soit il a été installé (avec `conda`, avec `pip`, etc.)
- **Cas 1.b :** le fichier module se trouve :
 - dans le même répertoire que le script qui l'importe
 - dans un répertoire référencé par la variable d'environnement `PYTHONPATH`

Le fichier `exos/mon_module.py` contient du code définissant `ma_variable` et `ma_fonction()`. On copie ce fichier dans le répertoire courant.

```
In [97]: import shutil # shutil fait partie de la bibliothèque standard

shutil.copy(
    "exos/mon_module.py", "."
) # On copie le fichier dans le répertoire d'exécution du notebook
%pycat mon_module.py
```

On peut maintenant importer `mon_module` depuis le notebook :

```
In [98]: import mon_module # Notez la syntaxe: nom_du_fichier sans l'extension.

print(mon_module.ma_variable)
mon_module.ma_fonction() # On accède ainsi à l'attribut ma_fonction() du module mon_module
```

27

un appel à `ma_fonction()`

On peut importer un module sous un autre nom (pour le raccourcir, en général) :

```
In [99]: import mon_module as mm
```

```
mm.ma_fonction()
```

un appel à `ma_fonction()`

On peut importer un objet particulier d'un module :

```
In [100]: from mon_module import ma_fonction
```

```
ma_fonction()
```

un appel à `ma_fonction()`

Ou encore en définissant un alias avec `as`

```
In [101]: from mon_module import ma_fonction as ma_fonc
```

```
ma_fonc()
```

un appel à `ma_fonction()`

Note :

Dans le code :

```
import mon_module
[...]  
import mon_module
```

Le deuxième `import` n'a pas d'effet car le module n'est importé qu'une seule fois au sein d'une même instance Python. Toutefois, il existe des moyens de le réimporter de force avec la bibliothèque [importlib](#).

Exercice : Modifiez le code contenu dans le fichier `mon_module.py`, et reexécutez la cellule ci-dessus. Que remarquez-vous ?

Cas 2 : le fichier module se trouve ailleurs

On peut distinguer deux cas d'usage :

- **Cas 2.a :** on veut aller chercher du code dans un autre projet python qui n'est pas installé dans l'environnement courant
- **Cas 2.b :** on travaille sur un gros projet structuré en modules stockés dans une arborescence de sous-répertoires.

Cas 2.a : exemple avec le fichier `exos/ext_dir/module_externe.py` Le fichier `module_externe.py` contient :

```
In [102]: %pycat exos/ext_dir/module_externe.py
```

On ajoute le répertoire `exos/ext_dir` à la liste des répertoires scannés par Python :

```
In [103]: import sys
```

```
    sys.path.append("./exos/ext_dir")
    print(sys.path)
```

```
['/builds/urfist/cours-python/notebooks', '/opt/conda/lib/python311.zip', '/opt/conda/lib/python3.11',
```

Le module s'importe alors directement avec :

```
In [104]: import module_externe
```

Je suis dans le module `module_externe`

Cas 2.b : un projet structuré en sous-répertoires Dans ce cas, on parle de paquets (*packages*) et de sous-paquets :

- n'importe quel répertoire contenant un fichier `__init__.py` est un **paquet python**
- chaque sous-répertoire est un sous-paquet du répertoire (ou paquet) parent
- une arborescence de paquets permet d'organiser le code de manière hiérarchique.
- On accède aux sous-paquets avec la notation :

```
import paquet.sous_paquet.sous_sous_paquet...
```

Exemple avec le répertoire `exos` :

```
In [105]: #!tree exos -P "*.py" -I __pycache__
```

```
exos
[...]
```

```
  sous_paquet
    __init__.py
    module_a.py
    module_b.py
  sous_paquet2
    __init__.py
    module_c.py
[...]
```

Dans ce cas, le module `exos/sous_paquet/module_a.py` contenant :

```
In [106]: %pycat exos/sous_paquet/module_a.py
```

s'importe de la façon suivante :

```
In [107]: import exos.sous_paquet.module_a
```

```
    # On appelle fonction()
    exos.sous_paquet.module_a.fonction()
```

Je suis dans le module `exos.sous_paquet.module_a`
 Je suis dans fonction() du module `exos.sous_paquet.module_a`

ou encore :

```
In [108]: from exos.sous_paquet import module_a

          # On appelle fonction()
          module_a.fonction()
```

Je suis dans fonction() du module `exos.sous_paquet.module_a`

Exercice : Importer directement la fonction `fonction` de `exos/sous_paquet/module_b.py` sous le nom `func` et appelez-là.

```
In [109]: %pycat exos/sous_paquet/module_b.py

In [110]: # Votre code ici
          # func()

In [111]: # Décommentez puis exécutez pour afficher le corrigé:
          # %load exos/snippets/03_langage/import_module.py
```

Remarque

Si `__init__.py` existe, le code qu'il contient est exécuté lors de l'import du paquet.

```
In [112]: %pycat exos/sous_paquet/__init__.py
```

Dans ce cas, on peut importer directement `module_a` et `module_b` :

```
In [113]: from exos.sous_paquet import *

          module_b.fonction()
```

Je suis dans le module `exos.sous_paquet.module_b`
 Je suis dans fonction() du module `exos.sous_paquet.module_b`

3.9.2 Imports relatifs

Depuis le module

`exos/sous_paquet2/module_c.py`

on peut importer le module

`exos/sous_paquet/module_b.py`

en utilisant la syntaxe :

```
from .. import sous_paquet
from ..sous_paquet import module_b
```

```
In [114]: %pycat exos/sous_paquet2/module_c.py
```

On importe `module_c` et on appelle ses attributs :

```
In [115]: from exos.sous_paquet2 import module_c
```

```
    module_c.fonction()
    module_c.module_b.fonction()
```

Je suis dans le module `exos.sous_paquet2.module_c`

Je suis dans `fonction()` du module `exos.sous_paquet2.module_c`

Je suis dans `fonction()` du module `exos.sous_paquet.module_b`

Pour plus d'informations sur les modules et paquets, voir [ce tutoriel](#).

3.9.3 Exercice

1. dans le répertoire `exos/sous_paquet2/` copiez la définition de la fonction `boule()` (cf. Section ??) dans un fichier nommé `deco.py`

```
def boule(x, y, couleur='bleue', rendu='mat'):
    print(f"J'accroche une boule en ({x}, {y}) de couleur {couleur} et de rendu {rendu}.")
```

2. dans le répertoire `exos/sous_paquet/` éditez un fichier nommé `noel.py` contenant :

- l'import de la fonction `boule()` depuis `exos/sous_paquet2/deco.py`
- la définition de la fonction `boule_or()` (cf. corrigé de l'exercice)

```
def boule_or(*args, **kwargs):
    # seule instruction qui fait une hypothèse
    # sur la signature de la fonction boule() :
    kwargs['couleur'] = "or"
    return boule(*args, **kwargs)
```

3. Décommentez la cellule ci-dessous, et exécutez-là pour vérifier que la fonction `boule_or()` est bien importée.

```
In [116]: %pycat exos/sous_paquet/noel.py
```

Error: no such file, variable, URL, history range or macro

```
In [117]: # from exos.sous_paquet.noel import boule_or
          # boule_or(1, 2)
```

```
In [118]: # Solution : décommentez les lignes suivantes pour positionner le corrigé
```

```
#!cp exos/sous_paquet/deco.py exos/sous_paquet2/
#!cp exos/sous_paquet2/noel.py exos/sous_paquet/
```

```
from exos.sous_paquet.noel import boule_or
```

```
boule_or(1, 2)
```

ModuleNotFoundError

Traceback (most recent call last)

Cell In[118], line 6

```
1 # Solution : décommentez les lignes suivantes pour positionner le corrigé
2
3 #!cp exos/sous_paquet/deco.py exos/sous_paquet2/
4 #!cp exos/sous_paquet2/noel.py exos/sous_paquet/
----> 6 from exos.sous_paquet.noel import boule_or
```

```
8 boule_or(1, 2)
```

```
ModuleNotFoundError: No module named 'exos.sous_paquet.noel'
```

3.9.4 Quelques modules de la stdlib

La bibliothèque standard de Python est incluse dans toute distribution de Python. Elle contient en particulier une panoplie de modules à la disposition du développeur.

string, pour manipuler les chaînes de caractères

```
— find()
— count()
— split()
— join()
— strip()
— upper()
— replace()
```

math, pour les fonctions mathématiques

```
— log()
— sqrt()
— cos()
— pi
— e
```

os, pour les fonctions liées au système d'exploitation

```
— listdir()
— getcwd()
— getenv()
— chdir()
— environ()
— os.path : exists(), getsize(), isdir(), join()
```

pathlib, pour manipuler les chemins de fichiers

```
— Path()
— exists()
— is_dir()
— is_file()
— mkdir()
— unlink()
— rename()
— glob()
```

sys

```
— argv
— exit()
— path
```

Mais bien plus sur la [doc officielle de la stdlib](#) !

3.10 Bonnes pratiques

3.10.1 Commentez votre code

- pour le rendre plus lisible
- pour préciser l'utilité des fonctions, méthodes, classes, modules, etc...
- pour expliquer les parties complexes

3.10.2 Documentez en utilisant les *docstrings*

- Juste après la signature de la fonction, on utilise une chaîne de caractère appelée *docstring* délimitée par `"""` et `"""`
- la *docstring* permet de documenter la fonction au plus près de sa définition
- cette *docstring* est affichée par `help(fonction)`
- la *docstring* est utilisée par les outils de documentation automatique comme [Sphinx](#).

```
In [119]: def nrien(n):
           """Retourne n fois 'rien'"""

           Args:
               n (int): nombre de répétitions
           """

           return "rien" * n
```

```
help(nrien)
# ou dans ipython:
nrien?
```

Help on function nrien in module __main__:

```
nrien(n)
    Retourne n fois 'rien'

    Args:
        n (int): nombre de répétitions
```

Plus d'information sur les Docstrings dans la [documentation officielle](#) et l'extension [Napoleon](#) pour [Sphinx](#).

3.10.3 Utilisez les annotations de type

Dans la version 3.5, Python a introduit le mécanisme d'[annotation de type](#) :

```
In [120]: def nrien(n: int) -> str:
           """Retourne n fois 'rien'"""
           return "rien" * n
```

```
nrien(3)
```

```
Out[120]: 'rienrienrien'
```

Ces annotations ne modifient pas l'exécution du code, mais elles présentent des avantages :

- elles indiquent au lecteur le type des arguments et de valeurs de retour,
- elles sont utilisées par les vérificateurs de code statique comme [mypy](#) et par [Sphinx](#).

Cela permet d'éviter de nombreux bugs !

3.10.4 Conventions d'écriture

Habituez-vous assez tôt aux conventions préconisées dans la communauté des utilisateurs de python. Cela vous aidera à relire plus facilement le code écrit par d'autres, et aidera les autres (et vous-même!) à relire votre propre code.

Ces conventions sont décrites dans le document [PEP n°8](#) (Python Enhancement Proposal). Les outils comme [pep8](#) ou [ruff](#) permettent de formater automatiquement le code source pour respecter ces règles.

Exercice :

1. Lisez le PEP8, et corrigez toutes les fautes commises dans ce notebook
2. Envoyez le résultat à votre enseignant

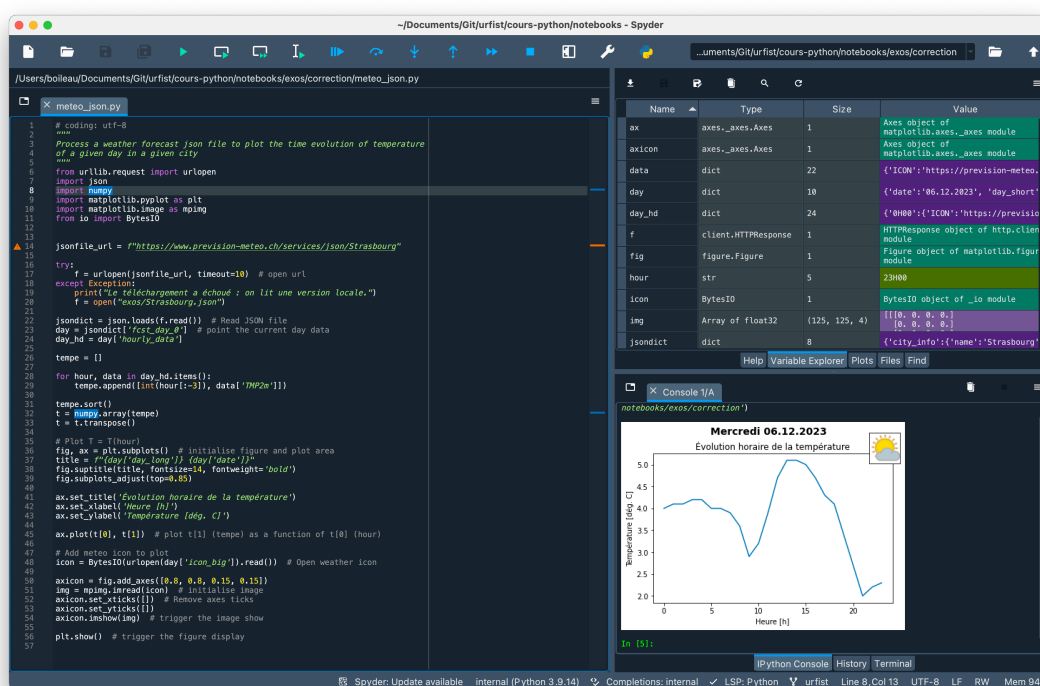
3.10.5 Organisez votre code source

Pour la lisibilité, la réutilisation et la maintenance, le principe à retenir est **d'évitez le copier-coller** de code :

- placez dans une même fonction les portions de code exécutées plusieurs fois
- placez dans un même module les variables, fonctions et classes partagées entre plusieurs parties ou programmes
- dans les projets importants, regrouper vos modules en packages

3.11 Utiliser les environnements de développement intégrés :

- spyder
- pycharm
- VSCode



3.11.1 Utilisez un gestionnaire de versions

L'outil incontournable depuis de nombreuses années pour gérer des versions est [Git](#).

Git est conçu pour faciliter la création et la fusion des branches. Un conseil général est de faire des enregistrements (*commits*) fréquents et cohérents.

3.11.2 Héberger vos dépôts de sources sur des forges logicielles

- [github](#) : ~100 millions d'utilisateurs, +400 millions de dépôts
- [gitlab](#) : un concurrent moins visible mais aux fonctionnalités très intéressantes. Disponible notamment sous forme d'instances privées.

3.11.3 Vérificateurs de code source

- [autopep8](#), [ruff](#) ou [black](#) : pour normaliser la mise en page
- [pylint](#) ou [ruff](#) : pour vérifier que la syntaxe est correcte
- [mypy](#) : pour la vérification de type

Ces modules sont généralement disponibles dans les IDE avancées.

3.11.4 Tests en Python

En programmation, on utilise des tests de non régression pour vérifier que les nouveaux développements et les corrections de bugs n'entraînent pas de pertes de fonctionnalités et de nouveaux bugs. On distingue généralement : - les tests unitaires (comportement de fonctions prises séparément) - les tests d'intégration (interaction entre plusieurs parties de programme) - les tests du système complet

Python dispose de nombreux modules dédiés aux deux premières catégories. Quelques exemples :

- [unittest](#) : fait partie de la bibliothèque standard
- [doctest](#) : le test est basé sur des sorties de sessions interactives stockées généralement dans la *docstring* de la fonction testée (alourdit le code...)
- [nose](#) : une extension de [unittest](#)
- [pytest](#) : syntaxe simple et nombreuses fonctionnalités

Une synthèse des outils existants sur [cette page](#).

3.11.5 Environnements virtuels

Les environnements virtuels sont très utiles pour le développement car ils permettent d'isoler les installations et les exécutions.

- [virtualenv](#)
- [conda environments](#)

3.12 Philosophie du langage : le zen de Python



PEP 20

```
In [121]: import this
```

The Zen of Python, by Tim Peters

```
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!
```

3.13 Python 3.x vs 2.x

C'est le futur, et incidemment aussi le présent voire même le passé...

- Quoi : une version qui casse la compatibilité ascendante
- Pourquoi : nettoyage de parties bancales du langage accumulées au fil du temps
- Python 3.0 est sorti en 2008
- Python 2.7 est sorti en 2010 : EOL, fin de vie, (mal-)heureusement longue à venir
- Un certain nombre de choses n'a pas encore été converti pour fonctionner avec
- Les distributions linux majeures proposent encore la 2.X par défaut, mais la 3 est disponible en parallèle
- Une partie, la moins disruptive, a quand même été portée vers 2.6 et 2.7 pour aider à la transition
- Les tutoriels, et autres documentations disponibles sur internet ne sont pas forcément migrées
- Pour un nouveau projet, recherchez un peu avant de vous lancer, pour vérifier les besoins en librairies externes
- Les implémentations tierces d'interpréteurs python peuvent avoir des degrés variables de compatibilité avec les versions 3.x
- Les modules comportant des extensions en C sont plus compliqués à porter.

3.13.1 Différences notables entre Python 2 et Python 3

- Division entière
- `print()`
- variable à durée de vie plus stricte (boucles, etc...)
- toutes les classes sont du nouveau type
- Les chaînes de caractères sont en UTF-8 par défaut & `encoding(s)` & `byte()` interface
- `stdlib` changée
- `range()` vs `xrange()`
- outils 2to3.py, 3to2, python-modernize, futurize

— `pylint --py3k`
 — module de compatibilité : `six`

Plus d'informations sur le [wiki officiel](#).

3.14 Exercice de synthèse : décodage de l'ARN

3.14.1 Enoncé

On souhaite traduire une séquence d'ARN stockée dans le fichier `exos/arn.txt` en une séquence d'acides aminés. Pour ce faire, on dispose du code génétique (simplifié) stocké dans `exos/code_genetique.csv` (tableau tiré de [wikipedia](#)).

Le fichier `code_genetique.csv` contient pour chaque ligne :

Nom de l'acide aminé;lettre unique;nom abrégé;codon1,codon2,...

où `codon1` et `codon2` sont des codons synonymes : ils codent pour le même acide.

In [122]: `%pycat exos/code_genetique.csv`

3.14.2 Consignes

1. Ecrire le code qui :

— ouvre le fichier code génétique
 — lit son contenu avec le module `csv` pour générer le dictionnaire suivant :

```
code = {letter: {name: ...,
                 abrv: ...,
                 codons: [..., ..., ]}}
```

In [123]: *# votre code ici*

In [124]: *# Décommentez la ligne suivante pour charger la solution*
%load exos/snippets/03_langage/arn_1.py

2. Construire le dictionnaire inverse `icode = {codon: letter}` qui traduit un codon en acide aminé représenté par sa lettre symbole.

In [125]: *# votre code ici*

In [126]: *# Décommentez la ligne suivante pour charger la solution*
%load exos/snippets/03_langage/arn_2.py

3. Ecrire la fonction `decode()` qui admet comme argument la chaîne de caractères représentant une séquence d'ARN et retourne la séquence d'acides aminés (appelée peptide) correspondante sous forme de chaîne de caractères. En fonction de l'argument optionnel `form`, cette fonction retournera :

— soit une séquence de symboles (lettres)
 — soit des abréviations d'acides aminés (séparées par des -).

In [127]: *# votre code ici*

In [128]: *# Décommentez la ligne suivante pour charger la solution*
%load exos/snippets/03_langage/arn_3.py

In [129]: `s = ''`
`help(s.join)`

Help on built-in function join:

```
join(iterable, /) method of builtins.str instance
Concatenate any number of strings.
```

The string whose method is called is inserted in between each given string.
The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

4. Appliquer la fonction `decode()` à la chaîne de caractères représentant l'ARN contenue dans `exos/arn.txt`.

```
In [130]: # votre code ici
```

```
In [131]: # Décommentez la ligne suivante pour charger la solution
          # %load exos/snippets/03_langage/arn_4.py
```

3.14.3 Solution complète

```
In [132]: # Décommentez la ligne suivante pour charger la solution
          # %load exos/snippets/03_langage/arn.py
```

3.15 Exercices complémentaires

3.15.1 Chaines de caractères

Ecrivez les fonctions suivantes, sans utiliser `upper()` ni `lower()` :

- `majuscule('azERtyUI') → 'AZERTYUI'`
- `minuscule('azERtyUI') → 'azertyui'`
- `inverse_casse('azERtyUI') → 'AZerTYui'`
- `nom_propre('azERtyUI') → 'Azertyui'`

Indice : cet exercice s'apparente à de la “traduction”...

```
In [133]: def majuscule(chaine: str) -> str:
          # Votre code ici
          pass

          def minuscule(chaine: str) -> str:
              # Votre code ici
              pass

          def inverse_casse(chaine: str) -> str:
              # Votre code ici
              pass

          def nom_propre(chaine: str) -> str:
              # Votre code ici
              pass
```

```

assert majuscule("azERtyUI") == "AZERTYUI"
assert minuscule("azERtyUI") == "azertyui"
assert inverse_casse("azERtyUI") == "AZerTYui"
assert nom_propre("azERtyUI") == "Azertyui"

```

```

-----
AssertionError                                Traceback (most recent call last)
Cell In[133], line 21
     16 def nom_propre(chaine: str) -> str:
     17     # Votre code ici
     18     pass
--> 21 assert majuscule("azERtyUI") == "AZERTYUI"
     22 assert minuscule("azERtyUI") == "azertyui"
     23 assert inverse_casse("azERtyUI") == "AZerTYui"

AssertionError:

```

```

In [134]: # Décommentez pour révéler la solution
          # %load exos/snippets/03_langage/casse.py

```

3.15.2 Récursion

Les fonctions dites récursives sont des fonctions qui font appel à elles-mêmes, en résolvant une partie plus petite du problème à chaque appel, jusqu'à avoir un cas trivial à résoudre.

Par exemple, pour calculer *la somme de tous les nombres de 0 jusqu'à x*, on peut utiliser une fonction récursive. Par exemple, pour $x = 10$, on a :

— $\sum_{n=0}^{10} n = 10 + \sum_{n=0}^9 n$
 — $\sum_{n=0}^9 n = 9 + \sum_{n=0}^8 n$
 — etc.

```

In [135]: def sum_to(x: int) -> int:
           if x == 0:
               return 0
           return x + sum_to(x - 1)

```

```

In [136]: print(sum_to(9))

```

45

La fonction mathématique factorielle est similaire, mais calcule *le produit de tous les nombres de 1 jusqu'à x*.

```

In [137]: def fact(x: int) -> int:
           if x == 1:
               return 1
           return x * fact(x - 1)

```

```

In [138]: print(fact(5), fact(9))

```

120 362880

La fonction mathématique qui calcule *la suite des nombres de Fibonacci*, peut être décrite de la façon suivante :

— `fibonacci(0) = 0`
— `fibonacci(1) = 1`

Et pour toutes les autres valeurs :

— `fibonacci(x) = fibonacci(x - 1) + fibonacci(x - 2)`

Exercice : écrivez une fonction récursive `fibonacci(x)` qui renvoie le x-ième nombre de la suite de Fibonacci.

```
In [139]: def fibonacci(x: int) -> int:
          # Votre code ici
          pass

          print(fibonacci(9))
```

None

```
In [140]: # Décommentez pour révéler la solution
          # %load exos/snippets/03_langage/fibonacci.py
```


Chapitre 4

Une introduction à Numpy



- Tableaux Numpy
- Création de tableaux
- Opérations basiques
- Indexation et slicing
- Algèbre linéaire
- Méthodes sur les tableaux
- Broadcasting de tableaux

Contenu sous licence [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/), largement inspiré de <https://github.com/pnavaro/python-notebooks> et <https://github.com/jakevdp/PythonDataScienceHandbook>

4.1 Numpy

Le Python pur est peu performant pour le calcul. Les listes ne sont pas des objets efficaces pour représenter les tableaux numériques de grande taille. **Numpy** a été créé à l'initiative de développeurs qui souhaitaient combiner la souplesse du langage python et des outils de calcul algébrique performants.

Numpy se base sur :

- le **ndarray** : tableau multidimensionnel
- des objets dérivés comme les *masked arrays* et les matrices
- les **ufuncs** : opérations mathématiques optimisées pour les tableaux
- des méthodes pour réaliser des opérations rapides sur les tableaux :
 - manipulation des *shapes*
 - tri
 - entrées/sorties

- FFT
- algèbre linéaire
- statistiques
- calcul aléatoire
- et bien plus!

Numpy permet de calculer *à la Matlab* en Python. Il est un élément de base de l'écosystème [SciPy](#)

4.2 Démarrer avec Numpy

```
In [1]: import numpy as np
        print(np.__version__)
```

1.26.4

Utilisez l'auto-complétion pour lister les objets numpy :

```
In [2]: #np.nd<TAB>
```

La rubrique d'aide est également précieuse

```
In [3]: np.ndarray?
```

4.3 Tableaux Numpy

4.3.1 Une question de performance

- Les listes Python sont trop lentes pour le calcul et utilisent beaucoup de mémoire
- Représenter des tableaux multidimensionnels avec des listes de listes devient vite brouillon à programmer

Exemple avec le produit scalaire entre les vecteurs **a** et **b** :

- en python pur avec des listes :

```
In [4]: from random import random
        from operator import mul

        a = [random() for i in range(100000)]
        b = [random() for i in range(100000)]
        %timeit s = sum(map(mul, a, b))
        print(sum(map(mul, a, b)))
```

3.23 ms ± 62.9 s per loop (mean ± std. dev. of 7 runs, 100 loops each)
25049.973798307085

- en numpy avec des tableaux 1D :

```
In [5]: a_np = np.array(a)
        b_np = np.array(b)
        %timeit s = a_np.dot(b_np)
        print(a_np.dot(b_np))
```

20.9 s ± 6.02 s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
25049.973798306608

4.3.2 La différence avec les listes

Les différences entre tableaux Numpy et listes Python : - un `ndarray` a une taille fixée à la création - un `ndarray` est composé d'éléments du même type - les opérations sur les tableaux sont réalisées par des routines C pré-compilées et optimisées (éventuellement parallèles)

```
In [6]: a = np.array([0, 1, 2, 3]) # list
        b = np.array((4, 5, 6, 7)) # tuple
        c = np.matrix('8 9 0 1')  # string (syntaxe matlab)

In [7]: print(a, b, c)

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

4.3.3 Propriétés

```
In [8]: a = np.array([1, 2, 3, 4, 5]) # On crée un tableau

In [9]: type(a) # On vérifie son type

Out[9]: numpy.ndarray

In [10]: a.dtype # On affiche le type de ses éléments

Out[10]: dtype('int64')

In [11]: a.itemsize # On affiche la taille mémoire (en octets) de chaque élément

Out[11]: 8

In [12]: a.shape # On retourne un tuple indiquant la longueur de chaque dimension

Out[12]: (5,)

In [13]: a.size # On retourne le nombre total d'éléments

Out[13]: 5

In [14]: a.ndim # On retourne le nombre de dimensions

Out[14]: 1

In [15]: a.nbytes # On retourne l'occupation mémoire

Out[15]: 40
```

- Toujours utiliser `shape` ou `size` pour les tableaux numpy plutôt que `len`
- `len` est réservé aux tableaux 1D

4.4 Création de tableaux Numpy

4.4.1 Avec des valeur constantes

```
In [16]: x = np.zeros((5, 3)) # On ne précise pas le type : on crée des flottants
        print(x)
        print(x.dtype)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
float64
```

```
In [17]: x = np.zeros((5, 3), dtype=int) # On explicite le type
        print(x)
        print(x.dtype)
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
int64
```

On dispose d'une panoplie de fonctions pour allouer des tableaux avec des valeurs constantes ou non initialisées (empty) :

empty, empty_like, ones, ones_like, zeros, zeros_like, full, full_like

4.4.2 Création à partir d'une séquence

arange

C'est l'équivalent de `range` pour les listes.

```
In [18]: np.arange(5) # entiers de 0 à 4
```

```
Out[18]: array([0, 1, 2, 3, 4])
```

```
In [19]: np.arange(5, dtype=np.double) # flottants de 0. à 4.
```

```
Out[19]: array([0., 1., 2., 3., 4.])
```

```
In [20]: np.arange(2, 7) # entiers de 2 à 6.
```

```
Out[20]: array([2, 3, 4, 5, 6])
```

```
In [21]: np.arange(2, 7, 0.5) # flottants avec incrément de 0.5.
```

```
Out[21]: array([2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

linspace et logspace

```
In [22]: # 5 éléments régulièrement espacés entre 1 et 4, 1 et 4 inclus
        np.linspace(1, 4, 5)
```

```
Out[22]: array([1. , 1.75, 2.5 , 3.25, 4.  ])
```

```
In [23]: # 5 éléments régulièrement espacés selon une progression géométrique entre 10^1 et 10^4
        np.logspace(1, 4, 5)
```

```
Out[23]: array([ 10.          ,  56.23413252,  316.22776602, 1778.27941004,
                10000.         ])
```

Consulter l'aide contextuelle pour plus de fonctionnalités

```
In [24]: np.logspace?
```

4.4.3 Exercice : créer les tableaux suivants

```
[100 101 102 103 104 105 106 107 108 109]
```

Astuce : `np.arange()`

```
In [25]: # Votre code ici
```

```
In [26]: # Solution
         np.arange(100, 110)
```

```
Out[26]: array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109])
```

```
[-2. -1.8 -1.6 -1.4 -1.2 -1. -0.8 -0.6 -0.4 -0.2 0.
 0.2 0.4 0.6 0.8 1. 1.2 1.4 1.6 1.8]
```

Astuce : `np.linspace()`

```
In [27]: # Votre code ici
```

```
In [28]: # Décommenter la ligne suivante pour voir la solution
         # %load exos/snippets/04_numpy/linspace.py
```

```
[ 0.001 0.00129155 0.0016681 0.00215443 0.00278256
 0.00359381 0.00464159 0.00599484 0.00774264 0.01]
```

Astuce : `np.logspace()`

```
In [29]: # Votre code ici
```

```
In [30]: # Décommenter la ligne suivante pour voir la solution
         # %load exos/snippets/04_numpy/logspace.py
```

```
[[ 0.  0. -1. -1. -1.]
 [ 0.  0.  0. -1. -1.]
 [ 0.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

Astuce : `np.tri()`, `np.ones()`

```
In [31]: # Votre code ici
```

```
In [32]: # Décommenter la ligne suivante pour voir la solution
         # %load exos/snippets/04_numpy/tri.py
```

4.4.4 Création de tableaux à partir de fichiers

Afin d'illustrer la création d'un tableau numpy à partir de données lues dans un fichier texte, on commence par sauvegarder un tableau dans un fichier.

```
In [33]: x = np.arange(0.0, 5.0, 1.0)
         y = x * 10
         z = x * 100
```

```
In [34]: np.savetxt('test.out', (x, y, z))
         %pycat test.out
```

```
In [35]: np.savetxt('test.out', (x, y, z), fmt='%1.4e')  # Notation exponentielle
         %pycat test.out
```

```
In [36]: np.loadtxt('test.out')
```

```
Out[36]: array([[ 0.,  1.,  2.,  3.,  4.],
                [ 0., 10., 20., 30., 40.],
                [ 0., 100., 200., 300., 400.]])
```

`savetxt` et `loadtxt` ont leurs correspondants binaires :

- `save` : enregistrer un tableau dans un fichier binaire au format `.npy`
- `load` : créer un tableau numpy à partir d'un fichier binaire numpy

4.4.5 Format HDF5 avec H5py

Le format `.npy` n'est lisible que par Numpy. À l'inverse, le format HDF5 est partagé par un grand nombre d'applications. De plus, il permet de structurer des données binaires : - en les nommant - en ajoutant des métadonnées - en assurant une portabilité indépendante de la plateforme

voir le [manuel utilisateur](#)

H5py est une interface Python pour HDF5.

On écrit dans le fichier `test.h5`

```
In [37]: import h5py as h5

         with h5.File('test.h5', 'w') as f:
             f['x'] = x
             f['y'] = y
             f['z'] = z
```

On lit le fichier `test.h5` (qui pourrait provenir d'une autre application)

```
In [38]: with h5.File('test.h5', 'r') as f:
         for field in f.keys():
             print(f"{field}: {f[field][()]}")
```

```
x: [0. 1. 2. 3. 4.]
y: [ 0. 10. 20. 30. 40.]
z: [ 0. 100. 200. 300. 400.]
```

4.5 Opérations basiques sur les tableaux

Par défaut, Numpy réalise les opérations arithmétiques éléments par éléments

```
In [39]: a = np.array([0, 1, 2, 3])
         b = np.array([4, 5, 6, 7])

         print(a * b)  # Produit éléments par éléments : pas le produit matriciel !
         print(a + b)
```

```
[ 0  5 12 21]
[ 4  6  8 10]
```

```
In [40]: print(a**2)
```

```
[0 1 4 9]
```

```
In [41]: print(5 * a)
         print(5 + a)
```

```
[ 0  5 10 15]
[ 5  6  7  8]
```

```
In [42]: print(a < 2)
```

```
[ True  True False False]
```

```
In [43]: print(np.cos(a*np.pi)) # Utilisation de ufunc
```

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

De nombreuses ufunc dans la [doc officielle](#).

4.6 Indexation et slicing

4.6.1 Indexation

Les règles diffèrent légèrement des listes pour les tableaux multi-dimensionnels

```
In [44]: # Indexation d'un tableau numpy
         a = np.arange(9).reshape(3, 3) # Notez l'effet de la méthode reshape()
         print(a)
         print(a[1, 2])
```

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

```
In [45]: # Indexation de la liste équivalente
         liste = a.tolist()
         print(liste)
         print(liste[1][2])
```

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

4.6.2 Slicing

Pour les tableaux unidimensionnels, les règles de slicing sont les mêmes que pour les séquences. Pour les tableaux multidimensionnels numpy, le slicing permet d'extraire des séquences dans n'importe quelle direction.

```
In [46]: print(a)
         print(a[2, :]) # Retourne la 3ème ligne
         print(a[:, 1]) # Retourne la deuxième colonne
```

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

Attention : contrairement aux listes, le slicing de tableaux ne renvoie pas une copie mais constitue une *vue* du tableau.

```
In [47]: b = a[:, 1]
         b[0] = -1
         print(a)
```

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

`a` est aussi une vue du tableau `np.arange(9)` obtenue avec la méthode `reshape()` donc `a` et `b` sont deux vues différentes du même tableau :

```
In [48]: b.base is a.base # b.base retourne le tableau dont b est la vue
```

```
Out[48]: True
```

Si on veut réaliser une copie d'un tableau, il faut utiliser la fonction `copy()`

```
In [49]: b = a[:, 1].copy()
```

ici `b` n'est pas une vue mais une copie donc `b.base` vaut `None` et `a` n'est pas modifié.

```
In [50]: print(b.base)
         b[0] = 200
         print(a)
```

```
None
```

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

4.6.3 Exercice

Approcher la dérivée de $f(x) = \sin(x)$ par la méthode des différences finies centrées :

$$\frac{\partial f}{\partial x} \approx \frac{f\left(x + \frac{\Delta x}{2}\right) - f\left(x - \frac{\Delta x}{2}\right)}{\Delta x}$$

Les valeurs seront calculées au milieu de deux abscisses discrètes successives.

```
In [51]: # On crée un tableau de 40 points d'abscisse
         x, dx = np.linspace(0., 4.*np.pi, 40, retstep=True)
         y = np.sin(x)
```

- `x` représente les abscisses discrètes x_i sur l'intervalle $x_i \in [0, 4\pi]$ et `dx` le pas de discrétisation.
- `y` représente les valeurs de la fonction $f(x_i)$.

1. Calculer les valeurs de `x_mil` correspondant au milieu de l'intervale $[x_i, x_{i+1}]$.

In [52]: *# Votre code ici*

2. Calculer les valeurs de dy_dx correspondant à la dérivée de y par différences finies centrées.

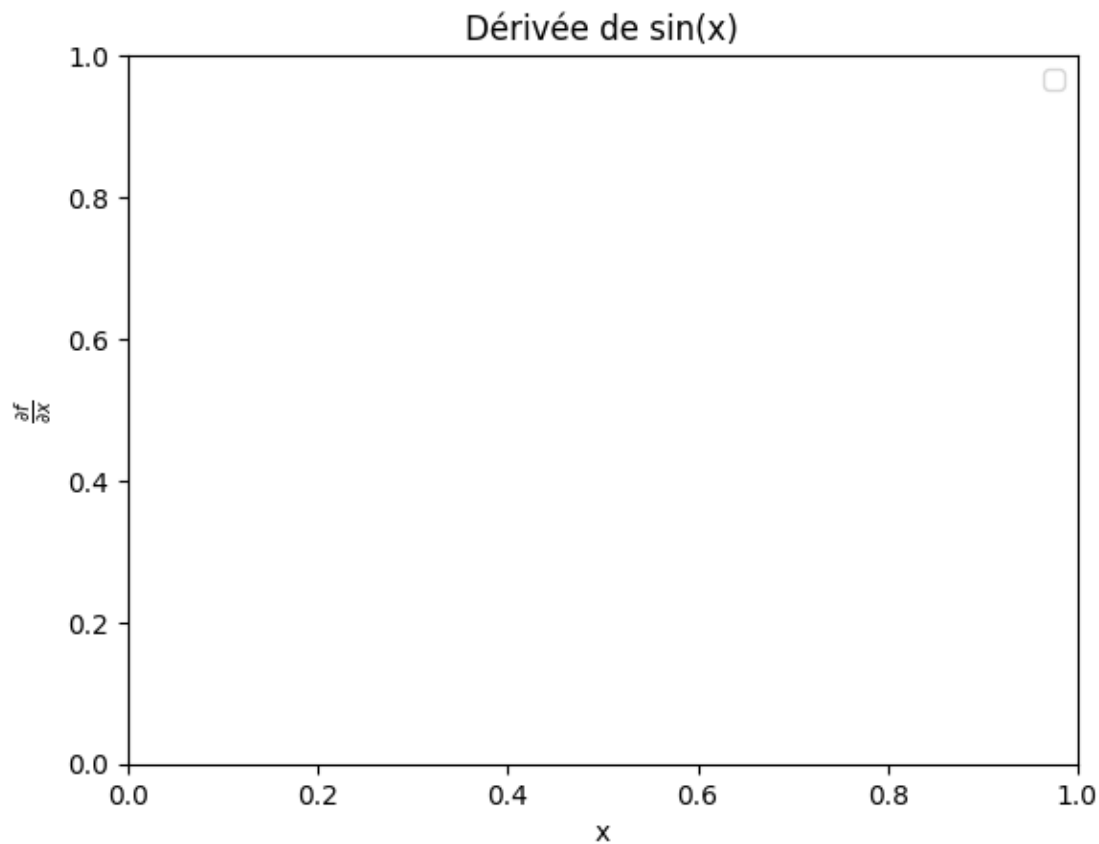
In [53]: *# Votre code ici*

3. Tracer la solution analytique $\cos(x)$ et dy_dx en fonction de x_mil .

```
In [54]: %matplotlib inline
import matplotlib.pyplot as plt

plt.title("Dérivée de sin(x)")
plt.xlabel(r"x")
plt.ylabel(r"$\frac{\partial f}{\partial x}$");
# Décommentez les lignes ci-dessous pour tracer la dérivée numérique dy_dx
# plt.plot(x_mil, np.cos(x_mil), label="cos(x)") # la dérivée analytique de sin() est cos()
# plt.plot(x_mil, dy_dx, 'rx', label="différences finies") # x_mil est le milieu de deux abscisses
plt.legend();
```

No artists with labels found to put in legend. Note that artists whose label start with an underscore



```
In [55]: # Décommenter la ligne suivante pour voir la solution complète
# %load exos/snippets/04_numpy/diff_finies.py
```

4.7 Quelques opérations d’algèbre linéaire

```
In [56]: A = np.array([[1, 1],
                      [0, 1]])
          B = np.array([[2, 0],
                      [3, 4]])
```

```
In [57]: # Produit élément par élément
          print(A * B)
```

```
[[2 0]
 [0 4]]
```

```
In [58]: # Produit matriciel (3 syntaxes équivalentes)
          print(A.dot(B))
          print(np.dot(A, B))
          print(A @ B)
```

```
[[5 4]
 [3 4]]
[[5 4]
 [3 4]]
[[5 4]
 [3 4]]
```

Soit la matrice :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
In [59]: a = np.array([[1., 2.],
                      [3., 4.]])
          print(a)
```

```
[[1. 2.]
 [3. 4.]]
```

```
In [60]: # Deux syntaxes équivalentes pour la transposition
          print(a.transpose())
          print(a.T)
```

```
[[1. 3.]
 [2. 4.]]
[[1. 3.]
 [2. 4.]]
```

```
In [61]: print(np.linalg.inv(a)) # inversion de matrice
          print(np.linalg.inv(a) @ a) # on vérifie que l'on obtient la matrice identité
```

```
[[ -2.   1. ]
 [ 1.5 -0.5]]
[[1.00000000e+00 0.00000000e+00]
 [1.11022302e-16 1.00000000e+00]]
```

```
In [62]: print(np.trace(a)) # trace
```

5.0

```
In [63]: print(np.eye(2))      # "eye" représente "I", la matrice identité
```

```
[[1.  0.]
 [0.  1.]]
```

```
In [64]: y = np.array([[5.0], [7.0]]) # Résoudre A*x = y
         x = np.linalg.solve(a, y)
         print(x)
         print(a @ x == y) # On vérifie que x est solution
```

```
[[ -3.]
 [  4.]]
[[ True]
 [ True]]
```

```
In [65]: j = np.array([[0.0, -1.0],
                       [1.0, 0.0]])
         print(np.dot(j, j)) # produit matriciel
         print(np.linalg.eig(j)) # Extraction des valeurs propres
```

```
[[ -1.  0.]
 [  0. -1.]]
```

```
EigResult(eigenvalues=array([0.+1.j, 0.-1.j]), eigenvectors=array([[0.70710678+0.j, 0.70710678-0.j],
                        [0. -0.70710678j, 0. +0.70710678j]]))
```

4.8 Les méthodes associées aux tableaux

Elles sont très nombreuses : impossible de toutes les lister dans le cadre de ce cours. Citons brièvement :

- min, max, sum
- sort, argmin, argmax
- statistiques basiques : cov, mean, std, var

À chercher dans la [doc officielle](#).

4.9 Broadcasting de tableaux

Le *broadcasting* est un mécanisme qui permet de faire des opérations sur des tableaux de différentes tailles ou *shapes*.

Pour rappel, les opérations sur les tableaux de même taille sont réalisées élément par élément :

```
In [66]: a = np.array([0, 1, 2])
         b = np.array([1, 1, 1])
         a + b
```

```
Out[66]: array([1, 2, 3])
```

Si on remplace `b` par le scalaire `1`, numpy déclenche le mécanisme de *broadcasting* où ce scalaire est virtuellement transformé en tableau de même taille que `a` en répétant la valeur, ce qui aboutit au même résultat :

```
In [67]: a + 1
```

```
Out[67]: array([1, 2, 3])
```

Si on additionne maintenant un tableau 1D et un tableau 2D :

```
In [68]: A = np.ones((3, 3), dtype=int)
         print(A)
```

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

```
In [69]: print(a)
```

```
[0 1 2]
```

```
In [70]: A + a
```

```
Out[70]: array([[1, 2, 3],
                [1, 2, 3],
                [1, 2, 3]])
```

`a` est étendu suivant la 2e dimension jusqu'à couvrir la *shape* de `A`.
Considérons maintenant un tableau ligne et un tableau colonne :

```
In [71]: a = np.arange(3)
         print(a)
```

```
[0 1 2]
```

```
In [72]: b = np.arange(3).reshape(3, 1)
         print(b)
```

```
[[0]
 [1]
 [2]]
```

```
In [73]: a + b
```

```
Out[73]: array([[0, 1, 2],
                [1, 2, 3],
                [2, 3, 4]])
```

`a` et `b` sont étendus simultanément pour couvrir la *shape* (3, 3).

4.9.1 Règles de broadcasting

1. Si les deux tableaux n'ont pas la même dimension, la *shape* de celui avec la plus faible dimension est complétée avec des 1 sur son bord gauche.
2. Si les *shapes* des tableaux n'ont aucune valeur commune, le tableau dont la *shape* vaut 1 est étendue jusqu'à la valeur de l'autre tableau.
3. Si les tailles sont différentes et non-unitaires dans toutes les dimensions, alors le broadcasting est impossible et on lève une erreur.

4.9.2 Exemple 1 : on étend un tableau

```
In [74]: A = np.ones((2, 3), dtype=int)
         a = np.arange(3)
         print(f"A =\n {A}")
         print(f"a = {a}")
         print(f"{A.shape = }")
         print(f"{a.shape = }")
```

```
A =
[[1 1 1]
 [1 1 1]]
a = [0 1 2]
A.shape = (2, 3)
a.shape = (3,)
```

- Selon la règle 1., la *shape* de **a** devient (1, 3).
- Selon la règle 2., la première dimension de **a** est étendue à (2, 3).

Maintenant que les *shapes* correspondent, l'addition terme à terme est possible :

```
In [75]: A + a

Out[75]: array([[1, 2, 3],
                [1, 2, 3]])
```

4.9.3 Exemple 2 : on étend les deux tableaux

```
In [76]: a = np.arange(3).reshape((3, 1))
         print(a)
         print(f"{a.shape = }")
```

```
[[0]
 [1]
 [2]]
a.shape = (3, 1)
```

```
In [77]: b = np.arange(3)
         print(b)
         print(f"{b.shape = }")
```

```
[0 1 2]
b.shape = (3,)
```

Dans ce cas, les deux tableaux doivent être étendus.

- Selon la règle 1., la *shape* de **b** doit être étendue en (1, 3).
- Selon la règle 2., on étend la *shape* de **a** et **b** en prenant le maximum de la taille dans chaque dimension soit (3, 3).

```
In [78]: a + b

Out[78]: array([[0, 1, 2],
                [1, 2, 3],
                [2, 3, 4]])
```

4.9.4 Exemple 3 : broadcasting impossible

```
In [79]: a = np.ones((3, 2), dtype=int)
         print(a)
         print(f"{a.shape = }")
```

```
[[1 1]
 [1 1]
 [1 1]]
a.shape = (3, 2)
```

```
In [80]: b = np.arange(3)
         print(b)
         print(f"{b.shape = }")
```

```
[0 1 2]
b.shape = (3,)
```

- Selon la règle 1., la *shape* de *b* devient (1, 3).
- Selon la règle 2., la *shape* de *b* devient (3, 3).
- Selon la règle 3., le broadcasting est alors impossible :

```
In [81]: a + b
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[81], line 1
----> 1 a + b

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

4.9.5 Exemple d'utilisation

Un cas classique d'utilisation du broadcasting consiste à retirer la moyenne dans certaines dimensions :

```
In [82]: rng = np.random.default_rng(seed=0)
         X = rng.random((6, 3))
         X
```

```
Out[82]: array([[0.63696169, 0.26978671, 0.04097352],
                [0.01652764, 0.81327024, 0.91275558],
                [0.60663578, 0.72949656, 0.54362499],
                [0.93507242, 0.81585355, 0.0027385 ],
                [0.85740428, 0.03358558, 0.72965545],
                [0.17565562, 0.86317892, 0.54146122]])
```

On calcule la moyenne selon la première dimension :

```
In [83]: Xmean = X.mean(0)
         Xmean
```

```
Out[83]: array([0.5380429 , 0.58752859, 0.46186821])
```

On peut maintenant calculer la valeur centrée de *X* en retirant la moyenne :

```
In [84]: X_centered = X - Xmean
         X_centered
```

```
Out[84]: array([[ 0.09891878, -0.31774188, -0.42089469],
                [-0.52151527,  0.22574164,  0.45088737],
                [ 0.06859287,  0.14196797,  0.08175678],
                [ 0.39702952,  0.22832496, -0.45912971],
                [ 0.31936137, -0.55394302,  0.26778724],
                [-0.36238728,  0.27565033,  0.07959301]])
```

On vérifie que `X_centered` est à moyenne nulle suivant la première dimension :

```
In [85]: X_centered.mean(0)
```

```
Out[85]: array([ 5.55111512e-17,  0.00000000e+00, -5.55111512e-17])
```

4.10 Programmation avec Numpy

- Les opérations sur les tableaux sont rapides, les boucles python sont lentes => **Éviter les boucles !**
- C'est une gymnastique qui nécessite de l'entraînement
- Le résultat peut devenir difficile à lire et à déboguer, par exemple dans le cas de boucles contenant de multiples conditions
- Dans certains cas, l'écriture de boucles est inévitable : des solutions performantes existent comme Cython, Pythran, Numba, etc.

4.11 Références

- [NumPy reference](#)
- [Numpy by Konrad Hinsén](#)
- [Cours de Pierre Navaro](#)
- [Python Datascience Handbook](#), par Jake VanderPlas
- [Scipy Lecture notes](#)

Chapitre 5

Microprojet



- Utiliser les modules de la bibliothèque standard pour récupérer des données via un service web.
- Manipuler les dictionnaires et les chaînes de caractères
- Utiliser la bibliothèque de tracés graphiques matplotlib
- Utiliser un IDE (Spyder)
- Exécuter un fichier script en gérant les arguments de la ligne de commande

5.1 Exercice

Exploiter les données du site <https://www.prevision-meteo.ch> pour tracer l'évolution horaire de la température à Strasbourg aujourd'hui.



5.1.1 Ouverture du fichier de prévisions

Le site <https://www.prevision-meteo.ch> fournit des prévisions sous forme de fichier au format `json`. On veut récupérer les données relatives à Strasbourg avec la méthode `urlopen()` du module `urllib.request`.

```
In [1]: %config InlineBackend.figure_format = 'retina'
        %matplotlib inline
        from urllib.request import urlopen

        jsonfile_url = "https://www.prevision-meteo.ch/services/json/Strasbourg"
        try:
            f = urlopen(jsonfile_url, timeout=10) # open url
        except Exception as e:
            print(f"Erreur: {e}")
            print("Le téléchargement a échoué : on lit une version locale.")
            f = open("exos/Strasbourg.json")
```

5.1.2 Chargement du fichier json ouvert

La méthode `json.loads()` permet de charger un fichier json comme un dictionnaire python :

```
In [2]: import json
        jsondict = json.loads(f.read()) # Read JSON file
        f.close() # Close file
```

5.1.3 Exploration des données

On commence naïvement par afficher le contenu du fichier :

```
In [3]: print(type(jsondict))
        print(jsondict)
```

```
<class 'dict'>
{'city_info': {'name': 'Strasbourg', 'country': 'France', 'latitude': '48.5844421', 'longitude': '7.755'}}
```

On essaie de faire mieux en affichant uniquement les clés du dictionnaire :

```
In [4]: for k in jsondict:
        print(repr(k))
```

```
'city_info'
'forecast_info'
'current_condition'
'fcst_day_0'
'fcst_day_1'
'fcst_day_2'
'fcst_day_3'
'fcst_day_4'
```

On est intéressé par le temps d'aujourd'hui :

```
In [5]: day = jsondict['fcst_day_0']
        print(day)
```

```
{'date': '28.01.2025', 'day_short': 'Mar.', 'day_long': 'Mardi', 'tmin': 5, 'tmax': 10, 'condition': 'E'}
```

Là aussi, on cherche les clés :

```
In [6]: for k in day:
        print(repr(k))
```

```
'date'  
'day_short'  
'day_long'  
'tmin'  
'tmax'  
'condition'  
'condition_key'  
'icon'  
'icon_big'  
'hourly_data'
```

Vérifions qu'il s'agit d'aujourd'hui :

```
In [7]: print(day['day_long'], day['date'])
```

Mardi 28.01.2025

C'est bon ! Maintenant, une entrée particulière nous intéresse :

```
In [8]: day_hd = day['hourly_data']  
        for k in day_hd:  
            print(repr(k))
```

```
'0H00'  
'1H00'  
'2H00'  
'3H00'  
'4H00'  
'5H00'  
'6H00'  
'7H00'  
'8H00'  
'9H00'  
'10H00'  
'11H00'  
'12H00'  
'13H00'  
'14H00'  
'15H00'  
'16H00'  
'17H00'  
'18H00'  
'19H00'  
'20H00'  
'21H00'  
'22H00'  
'23H00'
```

Regardons ce que contient une `hourly_data` :

```
In [9]: for k in day_hd['8H00']:  
        print(repr(k))
```

```
'ICON'
'CONDITION'
'CONDITION_KEY'
'TMP2m'
'DPT2m'
'WNDCHILL2m'
'HUMIDEX'
'RH2m'
'PRMSL'
'APCPsfc'
'WNDSPD10m'
'WNDGUST10m'
'WNDDIR10m'
'WNDDIRCARD10'
'ISSNOW'
'HCDC'
'MCDC'
'LCDC'
'HGTOC'
'KINDEX'
'CAPE180_0'
'CIN180_0'
```

La clé qui nous intéresse est la chaîne 'TMP2m' qui correspond à la température à 2m du sol.

```
In [10]: hour = '12H00'
         print(f"Aujourd'hui à {hour}, il fera : {day_hd[hour]['TMP2m']} deg. C.")
```

Aujourd'hui à 12H00, il fera : 8.2 deg. C.

Sauver ces lignes de commandes dans le fichier `today_stras.py` en allant de l'exécution 1 au compteur d'exécution courant indiqué dans la cellule de code ci-dessus In [XX]. Dans le cas présent :

```
In [11]: # Décommenter la ligne ci-dessous
         #%save today_stras.py 1-10
```

5.1.4 Tracé de la température

1. Ouvrir le fichier `today_stras.py` dans Spyder et nettoyer les `print` inutiles.
2. Exécutez le code dans Spyder et utilisez la fenêtre “Variable explorer” en haut à droite pour parcourir les données de votre dictionnaire.
3. Extraire :
 - `hours`, la listes des heures (entiers)
 - `temperatures`, la liste des températures (réels)
4. réaliser un tracé matplotlib en suivant [ce tutoriel](#) ou en intégrant les lignes de code suivantes :

```
In [12]: import matplotlib.pyplot as plt # To be placed at the top of python file

         # [Your previous code...]

         # Décommentez les lignes ci-dessous
         # Plot T = T(hour)
         # fig, ax = plt.subplots() # initialize figure and plot area
         # title = f"{day['day_long']} {day['date']} à {city_name.title()}"
```

```
# fig.suptitle(title, fontsize=14, fontweight="bold")
# fig.subplots_adjust(top=0.85)
#
# ax.set_title("Évolution horaire de la température")
# ax.set_xlabel("Heure [h]")
# ax.set_ylabel("Température [deg. C]")
#
# ax.plot(hours, temperatures) # plot temperatures vs hours
```

Option : intégrer l’icone de la météo du jour en utilisant le module `matplotlib.image`.

In [13]: # Décommentez les lignes ci-dessous

```
# # Add meteo icon to plot
# from io import BytesIO
# import matplotlib.image as mpimg
#
# icon = BytesIO(urlopen(day["icon_big"]).read()) # Open weather icon
#
# axicon = fig.add_axes([0.8, 0.8, 0.15, 0.15])
# img = mpimg.imread(icon) # initialise image
# axicon.set_xticks([]) # Remove axes ticks
# axicon.set_yticks([])
# axicon.imshow(img) # trigger the image show
```

Solution : [exos/correction/meteo_json.py](#).

5.2 Exercice sur les fonctions

À partir de [exos/correction/meteo_json.py](#), écrivez le programme `meteo_json_func.py` qui contient une fonction `plot_day_tempe()` admettant deux arguments :

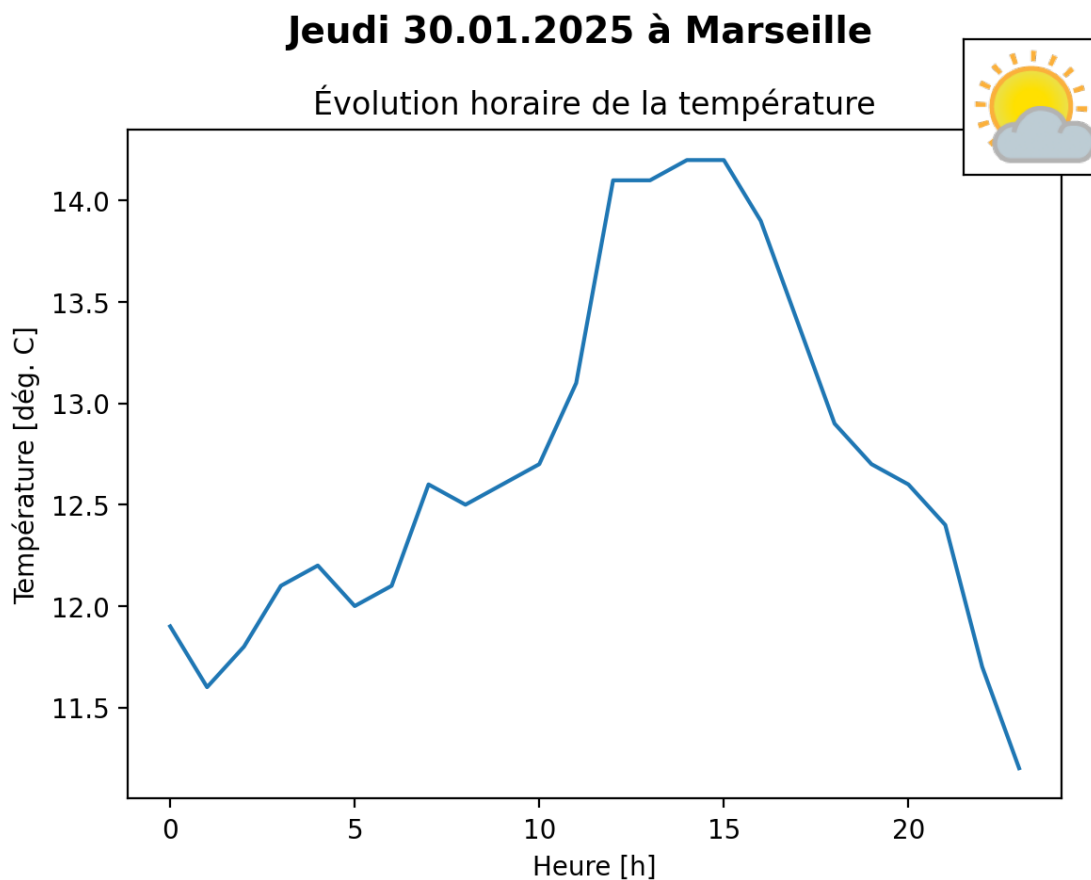
- `day_key` : un entier représentant le jour visé (0 : aujourd’hui, 1 : demain, 2 : après-demain...)
- `city_name` : une chaîne de caractère de la ville recherchée

In [14]: # Pour tester votre script dans cette cellule, décommenter les lignes suivantes
et redémarrer le noyau avant chaque modification :

```
# from meteo_json_func import plot_day_tempe
# plot_day_tempe(2, city_name='Marseille')
```

Solution dans [exos/correction/meteo_json_func.py](#). Dans cette correction, nous allons plus loin que l’énoncé en découplant le script en quatre fonctions.

In [15]: # Correction
from exos.correction.meteo_json_func import main
main(2, city_name='Marseille')



5.3 Exécution avec les widgets ipython

Jupyter [ipywidgets](#) permet de créer très facilement des menus interactifs pour faciliter l'exécution de code dans les notebooks.

Un exemple avec notre courbe de température :

```
In [16]: import exos.correction.meteo_json_func as meteo
         from ipywidgets import interact

         interact(meteo.main,
                   day_key=list(range(5)),
                   city_name=["Marseille", "Paris", "Toulouse", "Strasbourg"]
                   );

interactive(children=(Dropdown(description='day_key', options=(0, 1, 2, 3, 4), value=0), Dropdown(descr
```

5.4 Exécution en script

5.4.1 Utilisation de `if __name__ == '__main__':`

Dans un fichier python `test_module.py`, on souhaite généralement différencier : - le code exécuté lors de l'import du fichier **comme un module** depuis un autre programme python ou depuis un notebook Jupyter avec :

```
import test_module
```

Dans ce cas, la variable interne `__name__` vaut le nom du module (ici `test_module`).

— le code exécuté lorsque le fichier est appelé directement **comme un script** depuis le terminal système :

```
python test_module.py
```

Dans ce cas, la variable interne `__name__` vaut la chaîne de caractère `__main__`.

Prenons comme exemple la cellule suivante que l'on sauvegarde dans le fichier `test_module.py`.

```
In [17]: %%writefile test_module.py
def main():
    print(f'je suis dans {__name__}')

    if __name__ == '__main__':
        print("Je suis appelé comme programme principal")
        main()
    else:
        # En mode module importé, on ne fait rien de plus
        pass
```

Writing test_module.py

Appelons le fichier `test_module.py` comme un script python directement depuis le système :

```
In [18]: %run test_module.py
```

```
Je suis appelé comme programme principal
je suis dans __main__
```

<Figure size 640x480 with 0 Axes>

La variable `__name__` vaut `__main__`.

Importons maintenant le fichier comme un module.

```
In [19]: import test_module
```

Le bloc qui appelle la fonction `main()` n'est pas exécuté. En revanche cette fonction est accessible à la demande :

```
In [20]: test_module.main()
```

```
je suis dans test_module
```

Cette fois-ci, la variable `__name__` vaut `test_module`, c'est-à-dire le nom du module importé.

Plus d'information sur `__name__` dans la [doc officielle](#).

5.4.2 Gestion des arguments de la ligne de commande

Afin de positionner les paramètres d'un script à exécuter (noms de fichier, taille du problème, etc.), on a le choix entre :

- Editer le script là où les variables sont définies : si c'est envisageable pour des tests ou dans le contexte d'un notebook, ça ne l'est pas pour un programme destiné à être exécuté plusieurs fois avec des paramètres variables.

- Positionner des variables d’environnement qui peuvent être lues avec la fonction `os.getenv()` : le risque est de dissocier la définition des paramètres de l’exécution du programme.
- lire un fichier d’entrée, par exemple avec `configparser`. C’est particulièrement utile lorsque les paramètres sont nombreux et variés et que l’on souhaite faciliter la reproductibilité des exécutions mais ça demande d’éditer le fichier d’entrée à chaque changement de paramètre.
- interpréter les arguments de la ligne de commande : c’est la façon la plus souple d’exécuter un script avec des paramètres variables.

Une façon très simple et très rapide créer une interface de ligne de commande (CLI en anglais) est d’utiliser la bibliothèque `fire` développée par Google.

C’est une bibliothèque externe : commençons par l’installer avec pip.

```
In [21]: %pip install fire
```

```
Requirement already satisfied: fire in /home/jovyan/.local/lib/python3.11/site-packages (0.7.0)
Requirement already satisfied: termcolor in /home/jovyan/.local/lib/python3.11/site-packages (from fire)
```

Note: you may need to restart the kernel to use updated packages.

À titre d’exemple, le fichier `exos/correction/meteo_json_func.py` appelle `fire` dans son bloc final :

```
if __name__ == '__main__':
    import fire
    fire.Fire(main)
```

`fire` utilise la signature et la docstring de la fonction `main()` :

```
In [22]: from exos.correction.meteo_json_func import main
         help(main)
```

Help on function main in module exos.correction.meteo_json_func:

```
main(day_key: int = 0, city_name: str = 'strasbourg')
    Plot the time evolution of predicted temperature
```

pour créer la CLI suivante :

```
In [23]: %run exos/correction/meteo_json_func.py -h
```

```
INFO: Showing help with the command 'meteo_json_func.py -- --help'.
```

NAME

```
meteo_json_func.py - Plot the time evolution of predicted temperature
```

SYNOPSIS

```
meteo_json_func.py <flags>
```

DESCRIPTION

```
Plot the time evolution of predicted temperature
```

FLAGS

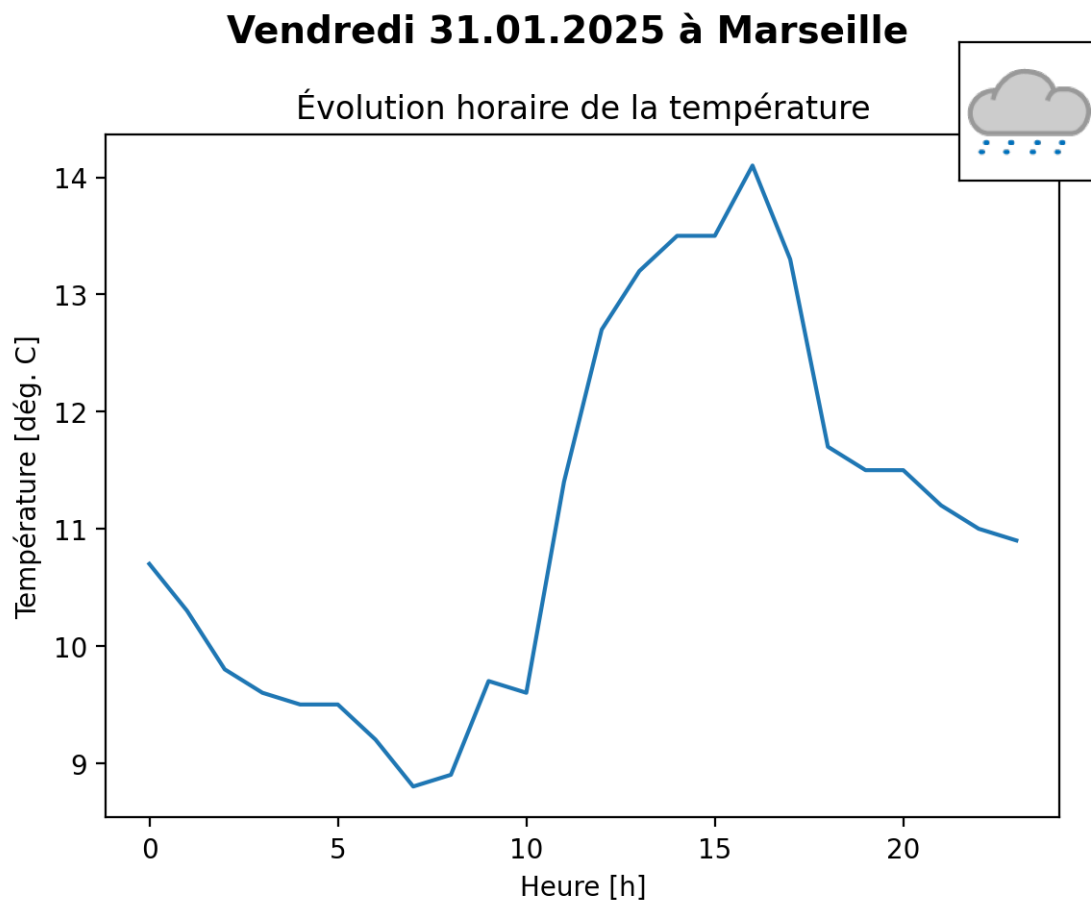
```
-d, --day_key=DAY_KEY
    Type: int
```



```
Default: 0  
-c, --city_name=CITY_NAME  
Type: str  
Default: 'strasbourg'
```

De sorte qu'on peut tracer la température à Marseille dans 3 jours en tapant :

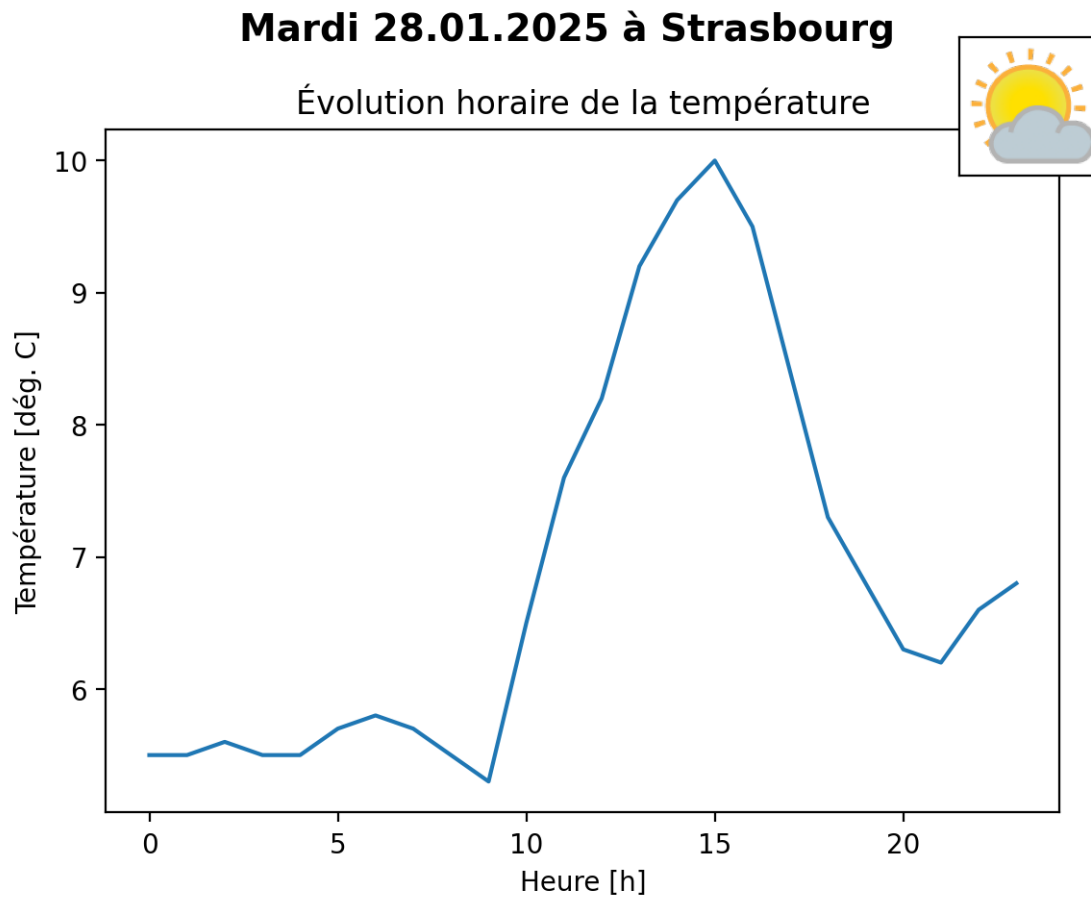
```
In [24]: %run exos/correction/meteo_json_func.py --day_key=3 --city_name=Marseille
```



<Figure size 640x480 with 0 Axes>

Ou avec les valeurs par défaut :

```
In [25]: %run exos/correction/meteo_json_func.py
```



<Figure size 640x480 with 0 Axes>

Si on se limite ici à la présentation de `fire`, il faut mentionner l'existence du module `argparse` qui fait partie de la bibliothèque standard. Moins immédiate mais aussi plus souple, l'utilisation d'`argparse` est décrite dans ce [tutoriel](#).

5.5 Utilisation avancée de Spyder

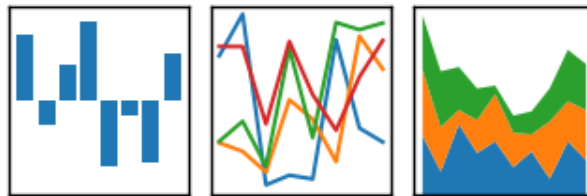
- explorer le système de [debugging](#)
- explorer le [profilier](#)

Chapitre 6

Une introduction à Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



- les *Series*
- les *Dataframes*
- Des exemples de traitement de données publiques

Contenu sous licence [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/), inspiré de <https://github.com/pnavaro/big-data>

6.1 Un outil pour l'analyse de données

- première version en 2011
- basé sur NumPy
- largement inspiré par la toolbox R pour la manipulation de données
- structures de données auto-descriptives
- Fonctions de chargement et écriture vers les formats de fichiers courants
- Fonctions de tracé
- Outils statistiques basiques

6.2 Les *Pandas series*

[Documentation officielle](https://pandas.pydata.org/pandas-docs/stable/10min.html)

- Une *series* Pandas :
 - un tableau 1D de données (éventuellement hétérogènes)
 - une séquence d'étiquettes appelée *index* de même longueur que le tableau 1D
- l'index peut être du contenu numérique, des chaînes de caractères, ou des dates-heures.
- si l'index est une valeur temporelle, alors il s'agit d'une *time series*
- l'index par défaut est `range(len(data))`

6.2.1 Illustration

```
In [1]: import pandas as pd
import numpy as np
pd.set_option("display.max_rows", 8) # Pour limiter le nombre de lignes affichées

In [2]: print(pd.Series([10, 8, 7, 6, 5]))
print(pd.Series([4, 3, 2, 1, 0.]))
```

```
0    10
1     8
2     7
3     6
4     5
dtype: int64
0     4.0
1     3.0
2     2.0
3     1.0
4     0.0
dtype: float64
```

6.2.2 Une série temporelle

Par exemple, les jours qui nous séparent du nouvel an.

```
In [3]: today = pd.Timestamp.today()
next_year = today.year + 1
time_period = pd.period_range(today, f'01/01/{next_year}', freq="D")
pd.Series(index=time_period, data=range(len(time_period) - 1, -1, -1))
```

```
Out[3]: 2025-01-28    338
        2025-01-29    337
        2025-01-30    336
        2025-01-31    335
        ...
        2025-12-29     3
        2025-12-30     2
        2025-12-31     1
        2026-01-01     0
        Freq: D, Length: 339, dtype: int64
```

6.2.3 Un exemple de traitement

On exploite un texte tiré de ce site non officiel : <http://www.sacred-texts.com/neu/mphg/mphg.htm>

```
In [4]: with open("exos/nee.txt") as f:
        nee = f.read()

        print(nee)
```

```
HEAD KNIGHT: Nee!
Nee!
Nee!
Nee!
ARTHUR: Who are you?
```

HEAD KNIGHT: We are the Knights Who Say... Nee!
 ARTHUR: No! Not the Knights Who Say Nee!
 HEAD KNIGHT: The same!
 BEDEMIR: Who are they?
 HEAD KNIGHT: We are the keepers of the sacred words: Nee, Pen, and
 Nee-wom!
 RANDOM: Nee-wom!
 ARTHUR: Those who hear them seldom live to tell the tale!
 HEAD KNIGHT: The Knights Who Say Nee demand a sacrifice!
 ARTHUR: Knights of Nee, we are but simple travellers who seek the
 enchanter who lives beyond these woods.
 HEAD KNIGHT: Nee! Nee! Nee! Nee!
 ARTHUR and PARTY: Oh, ow!
 HEAD KNIGHT: We shall say 'nee' again to you if you do not appease us.
 ARTHUR: Well, what is it you want?
 HEAD KNIGHT: We want... a shrubbery!
 [dramatic chord]
 ARTHUR: A what?
 HEAD KNIGHT: Nee! Nee!
 ARTHUR and PARTY: Oh, ow!
 ARTHUR: Please, please! No more! We shall find a shrubbery.
 HEAD KNIGHT: You must return here with a shrubbery or else you will
 never pass through this wood alive!
 ARTHUR: O Knights of Nee, you are just and fair, and we will return
 with a shrubbery.
 HEAD KNIGHT: One that looks nice.
 ARTHUR: Of course.
 HEAD KNIGHT: And not too expensive.
 ARTHUR: Yes.
 HEAD KNIGHTS: Now... go!

Dénombrer les occurrences de mots

On supprime la ponctuation

```
In [5]: for s in '.', '!', ',', '?', ':', '[', ']', 'ARTHUR', 'HEAD KNIGHT', 'PARTY':
        nee = nee.replace(s, '')
```

On transforme en minuscule et on découpe en une liste de mots

```
In [6]: nees = nee.lower().split()
        print(nees)
```

```
['nee', 'nee', 'nee', 'nee', 'who', 'are', 'you', 'we', 'are', 'the', 'knights', 'who', 'say', 'nee', 'I']
```

On crée un objet compteur

```
In [7]: from collections import Counter
        c = Counter(nees)
```

On ne retient que les mots qui apparaissent plus de 2 fois

```
In [8]: c = Counter({x: c[x] for x in c if c[x] > 2})
        c
```

```
Out[8]: Counter({'nee': 16,
                'who': 8,
                'the': 8,
                'you': 7,
                'we': 7,
                'are': 6,
                'and': 6,
                'a': 6,
                'knights': 5,
                'say': 4,
                'of': 4,
                'shrubby': 4,
                'not': 3})
```

Création d'une série Pandas à partir de l'objet c

Notons que la série est ordonnée avec un index croissant (dans l'ordre alphabétique).

```
In [9]: words = pd.Series(c)
        words
```

```
Out[9]: nee      16
        who      8
        are      6
        you      7
        ..
        of       4
        and      6
        a        6
        shrubby   4
        Length: 13, dtype: int64
```

Représentation dans un histogramme

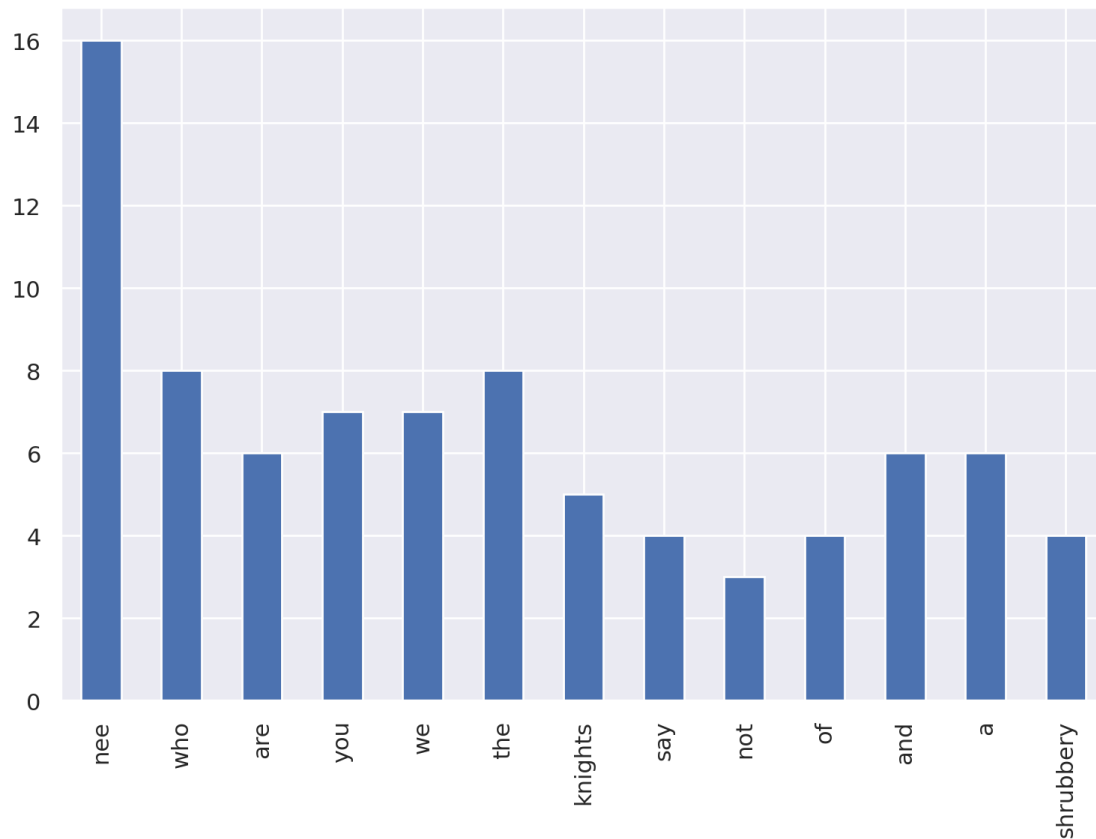
On commence par positionner certains paramètres de tracé

```
In [10]: %matplotlib inline
         %config InlineBackend.figure_format = 'retina'

         # Pour un rendu plus abouti https://seaborn.pydata.org/introduction.html
         import seaborn as sns
         sns.set()

         import matplotlib.pyplot as plt
         plt.rcParams['figure.figsize'] = (9, 6) # Pour obtenir des figures plus grandes

In [11]: words.plot(kind='bar');
```



6.2.4 Indexation et slicing

L'indexation et le slicing est une sorte de mélange entre les listes et les dictionnaires :

- `series[index]` pour accéder à la donnée correspondant à `index`
- `series[i]` où `i` est un entier qui suit les règles de l'indexation en python

Nombre d'occurrences de la chaîne `nee`

```
In [12]: print(words.index) # Pour rappel
         words["nee"]
```

```
Index(['nee', 'who', 'are', 'you', 'we', 'the', 'knights', 'say', 'not', 'of',
      'and', 'a', 'shrubbery'],
      dtype='object')
```

```
Out[12]: 16
```

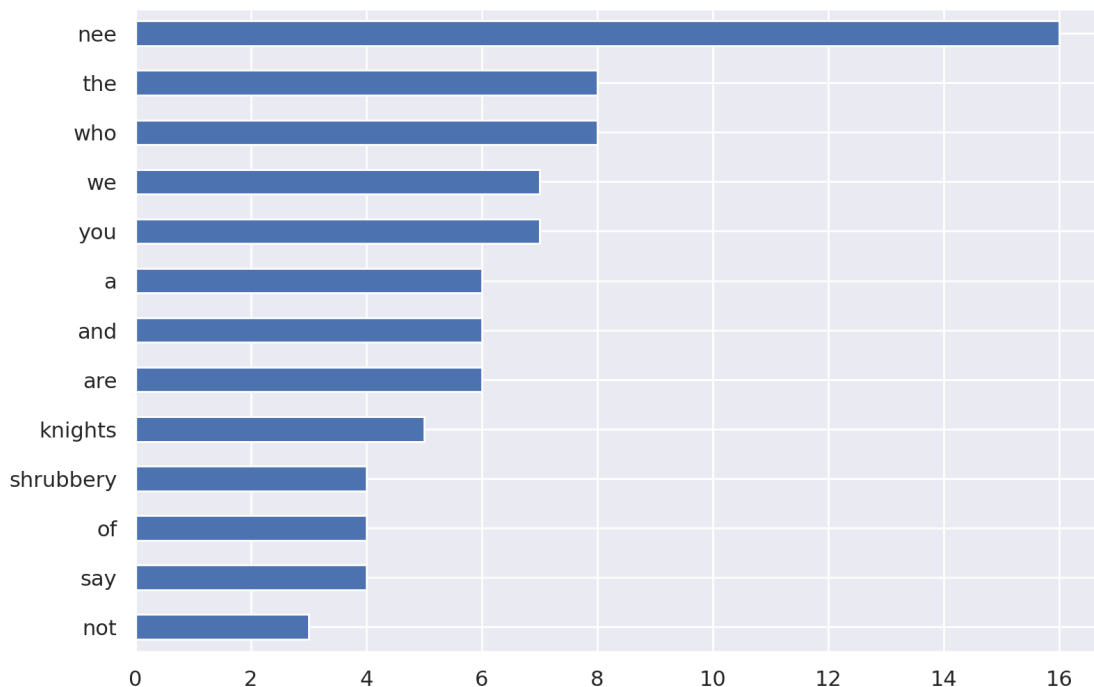
Trois dernières données de la série

```
In [13]: words[-3:]
```

```
Out[13]: and      6
         a        6
         shrubbery 4
         dtype: int64
```

6.2.5 Ordonner la série

```
In [14]: words.sort_values(inplace=True)
         words.plot(kind='barh'); # On change pour un histogramme horizontal
```



6.3 Les *Pandas Dataframes*

- C'est la structure de base de Pandas
- un *Dataframe* est une structure de données tabulées à deux dimensions, potentiellement hétérogène
- un *Dataframe* est constitué de lignes et colonnes portant des étiquettes
- C'est en quelque sorte un “dictionnaire de *Series*”.

6.3.1 Un exemple avec les arbres de la ville de Strasbourg

Conformément à l'[ordonnance du 6 juin 2005](#) (qui prolonge la loi CADA), la ville de Strasbourg a commencé à mettre en ligne ses données publiques.

En particulier des données sur ses arbres : <https://www.strasbourg.eu/arbres-alignements-espaces-verts>
On veut exploiter ces données. Pour ce faire, on va :

1. télécharger les données
2. les charger dans un *Dataframe*
3. les nettoyer/filtrer
4. les représenter graphiquement

On télécharge et on nettoie

On commence par définir une fonction qui télécharge et extrait une archive zip.


```
In [15]: from io import BytesIO
         from urllib.request import urlopen
         from zipfile import ZipFile

         def download_unzip(zipurl, destination):
             """Download zipfile from URL and extract it to destination"""
             with urlopen(zipurl) as zipresp:
                 with ZipFile(BytesIO(zipresp.read())) as zfile:
                     zfile.extractall(destination)
```

On l'utilise pour télécharger l'archive des données ouvertes de la ville de Strasbourg.

```
In [16]: download_unzip("https://www.strasbourg.eu/documents/976405/1168331/CUS_CUS_DEPN_ARBR.zip", "arbres")
```

On liste le contenu de l'archive

```
In [17]: %ls -R arbres
```

```
arbres:
CUS_CUS_DEPN_ARBR.csv
```

On charge le fichier csv comme un *Dataframe*.

```
In [18]: arbres_all = pd.read_csv("arbres/CUS_CUS_DEPN_ARBR.csv",
                                   encoding='latin', # Pour prendre en compte l'encoding qui n'est pas utf-8
                                   delimiter=";",    # Le caractère séparateur des colonnes
                                   decimal=',')      # Pour convertir les décimaux utilisant la notation ,
```

```
arbres_all
```

```
Out[18]:
```

| | Num point | vert point | vert NOM_USUEL | point vert | ADRESSE \ |
|-------|-----------|------------|----------------|------------|-----------------------------|
| 0 | | 450.0 | Rue du Houblon | | Houblon (rue du) |
| 1 | | 450.0 | Rue du Houblon | | Houblon (rue du) |
| 2 | | 450.0 | Rue du Houblon | | Houblon (rue du) |
| 3 | | 450.0 | Rue du Houblon | | Houblon (rue du) |
| ... | | ... | ... | | ... |
| 79134 | | 859.0 | Krummerort | | Oberjaegerhof (route de 1') |
| 79135 | | 859.0 | Krummerort | | Oberjaegerhof (route de 1') |
| 79136 | | 859.0 | Krummerort | | Oberjaegerhof (route de 1') |
| 79137 | | 859.0 | Krummerort | | Oberjaegerhof (route de 1') |

| | point vert | VILLE | Point vert | Quartier usuel \ |
|-------|------------|------------|------------|------------------|
| 0 | | STRASBOURG | | CENTRE |
| 1 | | STRASBOURG | | CENTRE |
| 2 | | STRASBOURG | | CENTRE |
| 3 | | STRASBOURG | | CENTRE |
| ... | | ... | | ... |
| 79134 | | STRASBOURG | | STOCKFELD |
| 79135 | | STRASBOURG | | STOCKFELD |
| 79136 | | STRASBOURG | | STOCKFELD |
| 79137 | | STRASBOURG | | STOCKFELD |

| | point vert | TYPLOGIE | n°arbre | SIG \ |
|---|------------|--------------------------------------|---------|-------|
| 0 | | ACCE - Accompagnement de cours d'eau | | 15783 |
| 1 | | ACCE - Accompagnement de cours d'eau | | 15784 |
| 2 | | ACCE - Accompagnement de cours d'eau | | 15785 |
| 3 | | ACCE - Accompagnement de cours d'eau | | 15786 |

```

...
79134 ACJF - Accompagnement de jardins familiaux      87652
79135 ACJF - Accompagnement de jardins familiaux      87653
79136 ACJF - Accompagnement de jardins familiaux      87654
79137 ACJF - Accompagnement de jardins familiaux      87655

```

```

      Libellé_Essence  Diam fût à 1m  Hauteur arbre
0      Tilia x 'Euchlora'      25.0      8.0
1      Tilia x 'Euchlora'      8.0      6.5
2      Tilia x 'Euchlora'      33.0      7.5
3      Tilia x 'Euchlora'      23.0      9.0
...
79134      Picea abies      30.0      10.0
79135      Picea abies      30.0      10.0
79136      Picea abies      30.0      10.0
79137      Picea abies      30.0      10.0

```

```
[79138 rows x 10 columns]
```

```
In [19]: print(f"{len(arbres_all)} arbres recensés !")
```

```
79138 arbres recensés !
```

On commence par lister les villes citées.

```
In [20]: print(set(arbres_all['point vert VILLE']))
```

```
{'REICHSTETT', 'HOLTZHEIM', 'OBERHAUSBERGEN', 'SCHILTIGHEIM', 'BLAESHEIM', 'WANTZENAU (La)', nan, 'LIPS'
```

On ne s'intéresse qu'à la ville de Strasbourg

```
In [21]: arbres = arbres_all[arbres_all['point vert VILLE'] == "STRASBOURG"]
        print(f"Il ne reste plus que {len(arbres)} arbres.")
```

```
Il ne reste plus que 64624 arbres.
```

On enlève les données incomplètes.

```
In [22]: arbres = arbres.dropna(axis=0, how='any')
        print(f"Il ne reste plus que {len(arbres)} arbres.")
```

```
Il ne reste plus que 61382 arbres.
```

On veut comptabiliser les essences

On extrait la série des essences.

```
In [23]: essences = set(arbres['Libellé_Essence'])
        print(f"Il y a {len(essences)} essences différentes !")
```

```
Il y a 456 essences différentes !
```

Les 5 premières dans l'ordre alphabétique :

```
In [24]: sorted(list(essences))[:5]
```

```
Out[24]: ['Abies (sp non déterminée)',
          'Abies alba',
          'Abies cephalonica',
          'Abies concolor',
          'Abies grandis']
```

C'est bientôt Noël, on se limite aux sapins!

```
In [25]: sapins = arbres[arbres['Libellé_Essence'].str.match("^Abies")]
        sapins
```

```
Out[25]:
```

| | Num point vert | | point vert NOM_USUEL \ |
|-------|----------------|---|------------------------|
| 2656 | 620.0 | | Parc des Contades |
| 2657 | 620.0 | | Parc des Contades |
| 9235 | 704.0 | | Groupe scolaire Ampère |
| 9575 | 1151.0 | Groupe scolaire Charles Adolphe Wurtz | |
| ... | ... | | ... |
| 75935 | 318.0 | Parc de la Citadelle -(01)- Secteur Centre et Est | |
| 75940 | 318.0 | Parc de la Citadelle -(01)- Secteur Centre et Est | |
| 75941 | 318.0 | Parc de la Citadelle -(01)- Secteur Centre et Est | |
| 78276 | 997.0 | | Parc de Pourtalès |

| | point vert ADRESSE | point vert VILLE | Point vert Quartier usuel \ |
|-------|-------------------------|------------------|-----------------------------|
| 2656 | Hirschler (rue René) | STRASBOURG | CONSEIL-XV |
| 2657 | Hirschler (rue René) | STRASBOURG | CONSEIL-XV |
| 9235 | Wattwiller (39, rue de) | STRASBOURG | NEUDORF |
| 9575 | Rieth (51, rue du) | STRASBOURG | CRONENBOURG |
| ... | ... | ... | ... |
| 75935 | Belges (quai des) | STRASBOURG | ESPLANADE |
| 75940 | Belges (quai des) | STRASBOURG | ESPLANADE |
| 75941 | Belges (quai des) | STRASBOURG | ESPLANADE |
| 78276 | Mélanie (rue) | STRASBOURG | ROBERTSAU |

| | point vert TYPOLOGIE | n°arbre SIG \ |
|-------|---|---------------|
| 2656 | PARC - Parcs | 20379 |
| 2657 | PARC - Parcs | 20380 |
| 9235 | EESE2 - Espaces des établissements sociaux et ... | 25143 |
| 9575 | EESE2 - Espaces des établissements sociaux et ... | 44237 |
| ... | ... | ... |
| 75935 | PARC - Parcs | 11419 |
| 75940 | PARC - Parcs | 11424 |
| 75941 | PARC - Parcs | 11425 |
| 78276 | PARC - Parcs | 40616 |

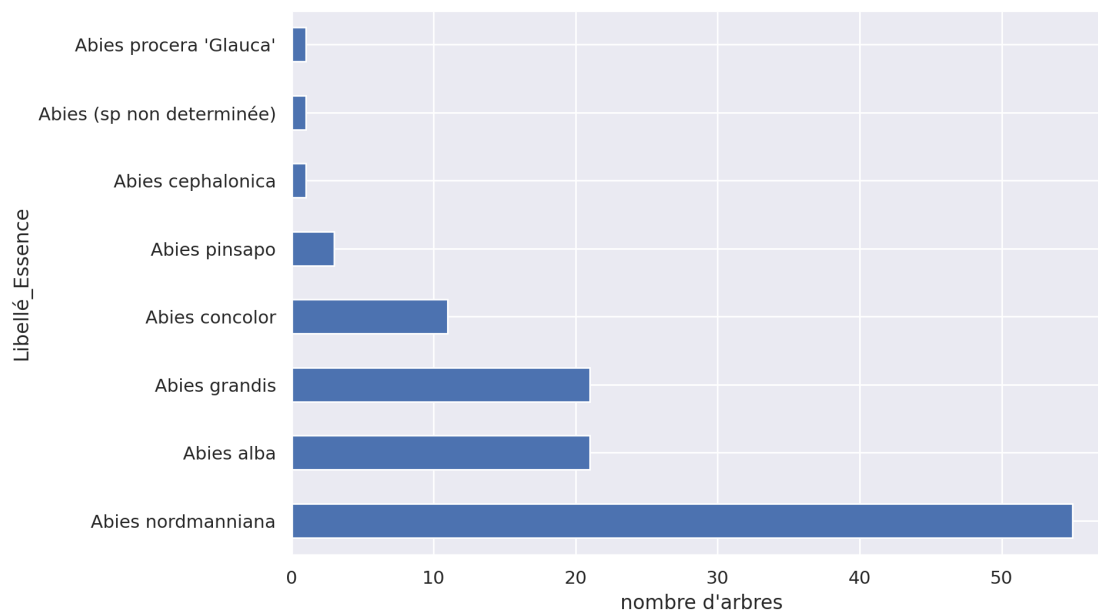
| | Libellé_Essence | Diam fût à 1m | Hauteur arbre |
|-------|--------------------|---------------|---------------|
| 2656 | Abies concolor | 26.0 | 14.0 |
| 2657 | Abies concolor | 23.0 | 13.5 |
| 9235 | Abies alba | 10.0 | 6.0 |
| 9575 | Abies nordmanniana | 10.0 | 4.0 |
| ... | ... | ... | ... |
| 75935 | Abies nordmanniana | 38.0 | 18.6 |
| 75940 | Abies concolor | 28.0 | 11.6 |
| 75941 | Abies concolor | 18.0 | 5.4 |

```
78276      Abies alba      29.0      16.0

[114 rows x 10 columns]
```

On trace leur répartition

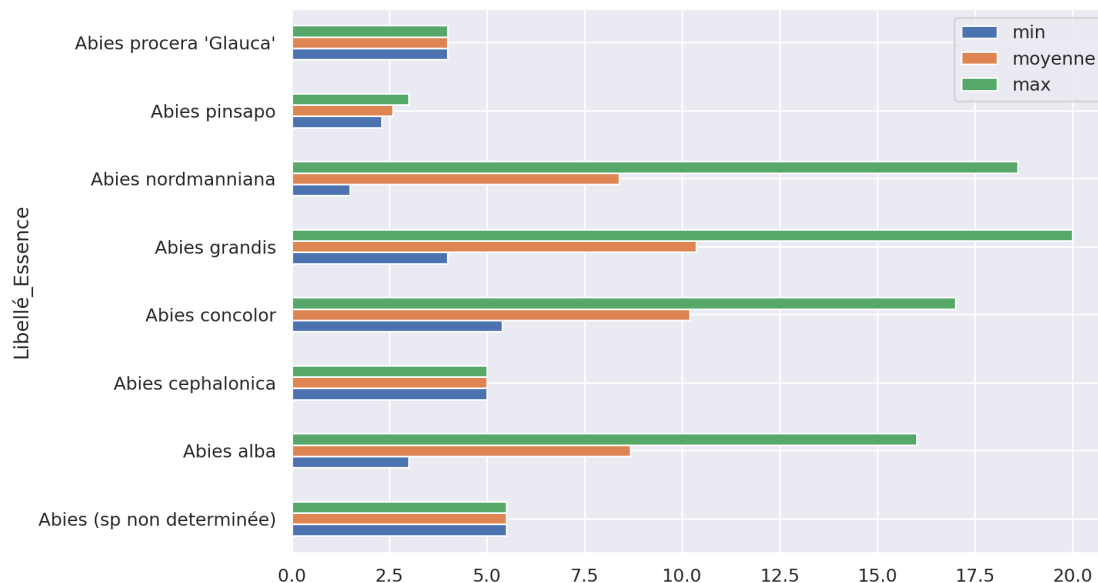
```
In [26]: ax = sapins['Libellé_Essence'].value_counts().plot(kind="barh");
         ax.set_xlabel("nombre d'arbres");
```



On veut faire des statistiques par essence

On veut connaître la hauteur moyenne par essence pour chaque type *Abies*.

```
In [27]: hauteurs_sapins = sapins.groupby(['Libellé_Essence'])["Hauteur arbre"]
         pd.concat([hauteurs_sapins.min().rename('min'),
                    hauteurs_sapins.mean().rename('moyenne'),
                    hauteurs_sapins.max().rename('max')],
                    axis=1).plot(kind='barh');
```



6.4 Représentation géographique

On voudrait maintenant représenter la répartition des arbres par quartiers.

On utilise à nouveau les données ouvertes de la ville de Strasbourg, cette fois-ci concernant les quartiers :

<https://data.strasbourg.eu/explore/dataset/strasbourg-15-quartiers/information/>

On télécharge, on extrait l'archive et on liste son contenu.

```
In [28]: download_unzip(r"https://data.strasbourg.eu/api/explore/v2.1/catalog/datasets/strasbourg-15-quartiers/expo
%ls -R quartiers
```

quartiers:

```
strasbourg-15-quartiers.cpg  strasbourg-15-quartiers.shp
strasbourg-15-quartiers.dbf  strasbourg-15-quartiers.shx
strasbourg-15-quartiers.prj
```

C'est le fichier `.shp` qui nous intéresse.

À ce stade, nous avons besoin des bibliothèques `GeoPandas` et `Folium` que l'on installe avec conda.

```
In [29]: %pip install geopandas folium
#%conda install -c conda-forge geopandas folium
```

```
Requirement already satisfied: geopandas in /home/jovyan/.local/lib/python3.11/site-packages (1.0.1)
Requirement already satisfied: folium in /home/jovyan/.local/lib/python3.11/site-packages (0.19.4)
Requirement already satisfied: numpy>=1.22 in /home/jovyan/.local/lib/python3.11/site-packages (from ge
Requirement already satisfied: pyogrio>=0.7.2 in /home/jovyan/.local/lib/python3.11/site-packages (from
Requirement already satisfied: packaging in /opt/conda/lib/python3.11/site-packages (from geopandas) (2
```

```
Requirement already satisfied: pandas>=1.4.0 in /home/jovyan/.local/lib/python3.11/site-packages (from
Requirement already satisfied: pyproj>=3.3.0 in /opt/conda/lib/python3.11/site-packages (from geopandas
Requirement already satisfied: shapely>=2.0.0 in /opt/conda/lib/python3.11/site-packages (from geopandas
```

```
Requirement already satisfied: branca>=0.6.0 in /opt/conda/lib/python3.11/site-packages (from folium) (0.8.0)
Requirement already satisfied: Jinja2>=2.9 in /home/jovyan/.local/lib/python3.11/site-packages (from folium) (3.1.2)
Requirement already satisfied: requests in /opt/conda/lib/python3.11/site-packages (from folium) (2.31.0)
Requirement already satisfied: xyzservices in /home/jovyan/.local/lib/python3.11/site-packages (from folium) (2023.10.3)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/conda/lib/python3.11/site-packages (from Jinja2>=2.9->folium) (2.1.2)
```

```
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/conda/lib/python3.11/site-packages (from pandas>=1.5.3) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /opt/conda/lib/python3.11/site-packages (from pandas>=1.5.3) (2023.3)
Requirement already satisfied: tzdata>=2022.7 in /opt/conda/lib/python3.11/site-packages (from pandas>=1.5.3) (2023.3)
```

```
Requirement already satisfied: certifi in /opt/conda/lib/python3.11/site-packages (from requests>=2.31.0->folium) (2023.7.22)
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/conda/lib/python3.11/site-packages (from requests>=2.31.0->folium) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.11/site-packages (from requests>=2.31.0->folium) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/conda/lib/python3.11/site-packages (from requests>=2.31.0->folium) (2.0.7)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.11/site-packages (from python-dateutil>=2.8.2->pandas>=1.5.3) (1.16.0)
```

Note: you may need to restart the kernel to use updated packages.

On charge le fichier comme un GeoDataFrame :

```
In [30]: import geopandas as gpd
         gdf_quartiers = gpd.read_file("quartiers/strasbourg-15-quartiers.shp")
         print(f"gdf_quartiers est de type {type(gdf_quartiers)}.")
         gdf_quartiers

gdf_quartiers est de type <class 'geopandas.geodataframe.GeoDataFrame'>.
```

```
Out[30]:
```

| | code_sct | libelle \ | |
|----|----------|----------------------------------|--|
| 0 | 8 | ELSAU | |
| 1 | 2B | MONTAGNE-VERTE | |
| 2 | 6 | ORANGERIE-CONSEIL DES XV | |
| 3 | 2A | KOENIGSHOFFEN | |
| .. | ... | ... | |
| 11 | 5 | BOURSE-ESPLANADE-KRUTENAU | |
| 12 | 11A | NEUHOF1 | |
| 13 | 9 | MEINAU | |
| 14 | 3 | TRIBUNAL-GARE-PORTE DE SCHIRMECK | |

| | geometry |
|----|---|
| 0 | POLYGON ((7.73488 48.56871, 7.73459 48.56614, ... |
| 1 | POLYGON ((7.72644 48.57736, 7.72666 48.57717, ... |
| 2 | POLYGON ((7.78101 48.57854, 7.78127 48.57923, ... |
| 3 | POLYGON ((7.72644 48.57736, 7.72633 48.5773, 7... |
| .. | ... |
| 11 | POLYGON ((7.75002 48.57454, 7.74983 48.57463, ... |
| 12 | POLYGON ((7.75904 48.56022, 7.75924 48.55992, ... |
| 13 | POLYGON ((7.73488 48.56871, 7.73494 48.5695, 7... |
| 14 | POLYGON ((7.75513 48.59156, 7.75506 48.59133, ... |

[15 rows x 3 columns]

Avec Folium, on commence par représenter ces données géographiques sur un fond de carte.

```
In [31]: import folium

# On crée une carte initialement centrée sur Strasbourg
STRASBOURG_COORD = (48.58, 7.75)
stras_map = folium.Map(STRASBOURG_COORD, zoom_start=11, tiles='cartodbpositron')

# On ajoute les données des quartiers
folium.GeoJson(gdf_quartiers).add_to(stras_map)

# On enregistre dans un fichier html
stras_map.save('stras_map.html')

# On trace dans le notebook
display(stras_map)

<folium.folium.Map at 0x7f49dc5a9290>
```

À l'emplacement de ces quartiers, on souhaite représenter une échelle de couleur en fonction de la densité d'arbres.

On constate que les noms de quartiers sont différents de ceux du jeu de données sur les arbres.

```
In [32]: from pprint import pprint

set_quartiers = set(gdf_quartiers['libelle'])
set_arbres = set(arbres['Point vert Quartier usuel'])

def print_set_data(s: set):
    """Print set and its length"""
    print(f"{pprint(s)} -> {len(s)}")

print_set_data(set_quartiers)
print_set_data(set_arbres)

{'BOURSE-ESPLANADE-KRUTENAU',
 'CENTRE',
 'CRONENBOURG',
 'ELSAU',
 'HAUTÉPIERRE',
 'KOENIGSHOFFEN',
 'MEINAU',
 'MONTAGNE-VERTE',
 'NEUDORF',
 'NEUHOF1',
 'NEUHOF2',
 'ORANGERIE-CONSEIL DES XV',
 'PORT DU RHIN',
 'ROBERTSAU',
 'TRIBUNAL-GARE-PORTE DE SCHIRMECK'} -> 15
{'BOURSE',
 'CENTRE',
 'CONSEIL-XV',
 'CRONENBOURG',
 'ELSAU',
 'ESPLANADE',
 'GARE',
 'HAUTÉPIERRE',
 'KOENIGSHOFFEN',
```

```
'KRUTENAU',
'MEINAU',
'MONTAGNE VERTE',
'MUSAU',
'NEUDORF',
'NEUHOF',
'ORANGERIE',
'PLAINE DES BOUCHERS',
'POLYGONE',
'PORT DU RHIN',
'PORTE DE SCHIRMECK',
'ROBERTSAU',
'STOCKFELD',
'TRIBUNAL',
'WACKEN'} -> 24
```

Certains noms figurent dans les deux jeux de données :

```
In [33]: intersection = set_quartiers.intersection(set_arbres)
         print_set_data(intersection)
```

```
{'CENTRE',
 'CRONENBOURG',
 'ELSAU',
 'HAUTÉPIERRE',
 'KOENIGSHOFFEN',
 'MEINAU',
 'NEUDORF',
 'PORT DU RHIN',
 'ROBERTSAU'} -> 9
```

D'autres sont différents :

```
In [34]: difference = set_arbres.difference(set_quartiers)
         print_set_data(difference)
```

```
{'BOURSE',
 'CONSEIL-XV',
 'ESPLANADE',
 'GARE',
 'KRUTENAU',
 'MONTAGNE VERTE',
 'MUSAU',
 'NEUHOF',
 'ORANGERIE',
 'PLAINE DES BOUCHERS',
 'POLYGONE',
 'PORTE DE SCHIRMECK',
 'STOCKFELD',
 'TRIBUNAL',
 'WACKEN'} -> 15
```

Afin d'obtenir de faire correspondre parfaitement les noms des deux jeux de données, on convertit les noms dans le Dataframe `arbres` en supposant les correspondances ci-dessous.


```

In [35]: conversion_dict = {
    'BOURSE': 'BOURSE-ESPLANADE-KRUTENAU',
    'CONSEIL-XV': 'ORANGERIE-CONSEIL DES XV',
    'ESPLANADE': 'BOURSE-ESPLANADE-KRUTENAU',
    'GARE': 'TRIBUNAL-GARE-PORTE DE SCHIRMECK',
    'KRUTENAU': 'BOURSE-ESPLANADE-KRUTENAU',
    'MONTAGNE VERTE': 'MONTAGNE-VERTE',
    'MUSAU': 'NEUDORF',
    'NEUHOF': 'NEUHOF1',
    'ORANGERIE': 'ORANGERIE-CONSEIL DES XV',
    'PLAINE DES BOUCHERS': 'MEINAU',
    'POLYGONE': 'NEUHOF1',
    'PORTE DE SCHIRMECK': 'TRIBUNAL-GARE-PORTE DE SCHIRMECK',
    'STOCKFELD': 'NEUHOF2',
    'TRIBUNAL': 'TRIBUNAL-GARE-PORTE DE SCHIRMECK',
    'WACKEN': 'ROBERTSAU'
}

for k, v in conversion_dict.items():
    arbres['Point vert Quartier usuel'] = \
        arbres['Point vert Quartier usuel'].replace(to_replace=k, value=v)

arbres

```

```

Out[35]:

```

| | Num point vert | point vert | NOM_USUEL | point vert | ADRESSE \ |
|-------|----------------|----------------|-----------------------------|------------------|-----------|
| 0 | 450.0 | Rue du Houblon | | Houblon (rue du) | |
| 1 | 450.0 | Rue du Houblon | | Houblon (rue du) | |
| 2 | 450.0 | Rue du Houblon | | Houblon (rue du) | |
| 3 | 450.0 | Rue du Houblon | | Houblon (rue du) | |
| ... | ... | ... | ... | ... | |
| 79134 | 859.0 | Krummerort | Oberjaegerhof (route de 1') | | |
| 79135 | 859.0 | Krummerort | Oberjaegerhof (route de 1') | | |
| 79136 | 859.0 | Krummerort | Oberjaegerhof (route de 1') | | |
| 79137 | 859.0 | Krummerort | Oberjaegerhof (route de 1') | | |

| | point vert | VILLE | Point vert | Quartier usuel \ |
|-------|------------|-------|------------|------------------|
| 0 | STRASBOURG | | CENTRE | |
| 1 | STRASBOURG | | CENTRE | |
| 2 | STRASBOURG | | CENTRE | |
| 3 | STRASBOURG | | CENTRE | |
| ... | ... | | ... | |
| 79134 | STRASBOURG | | NEUHOF2 | |
| 79135 | STRASBOURG | | NEUHOF2 | |
| 79136 | STRASBOURG | | NEUHOF2 | |
| 79137 | STRASBOURG | | NEUHOF2 | |

| | point vert | TYPLOGIE | n°arbre | SIG \ |
|-------|--|----------|---------|-------|
| 0 | ACCE - Accompagnement de cours d'eau | | 15783 | |
| 1 | ACCE - Accompagnement de cours d'eau | | 15784 | |
| 2 | ACCE - Accompagnement de cours d'eau | | 15785 | |
| 3 | ACCE - Accompagnement de cours d'eau | | 15786 | |
| ... | ... | | ... | |
| 79134 | ACJF - Accompagnement de jardins familiaux | | 87652 | |
| 79135 | ACJF - Accompagnement de jardins familiaux | | 87653 | |
| 79136 | ACJF - Accompagnement de jardins familiaux | | 87654 | |
| 79137 | ACJF - Accompagnement de jardins familiaux | | 87655 | |

| | Libellé_Essence | Diam fût à 1m | Hauteur arbre |
|-------|--------------------|---------------|---------------|
| 0 | Tilia x 'Euchlora' | 25.0 | 8.0 |
| 1 | Tilia x 'Euchlora' | 8.0 | 6.5 |
| 2 | Tilia x 'Euchlora' | 33.0 | 7.5 |
| 3 | Tilia x 'Euchlora' | 23.0 | 9.0 |
| ... | ... | ... | ... |
| 79134 | Picea abies | 30.0 | 10.0 |
| 79135 | Picea abies | 30.0 | 10.0 |
| 79136 | Picea abies | 30.0 | 10.0 |
| 79137 | Picea abies | 30.0 | 10.0 |

[61382 rows x 10 columns]

On vérifie que l'ensemble des quartiers est le même pour les deux Dataframes `quartiers` et `arbres`.

```
In [36]: set(arbres['Point vert Quartier usuel']) == set_quartiers
```

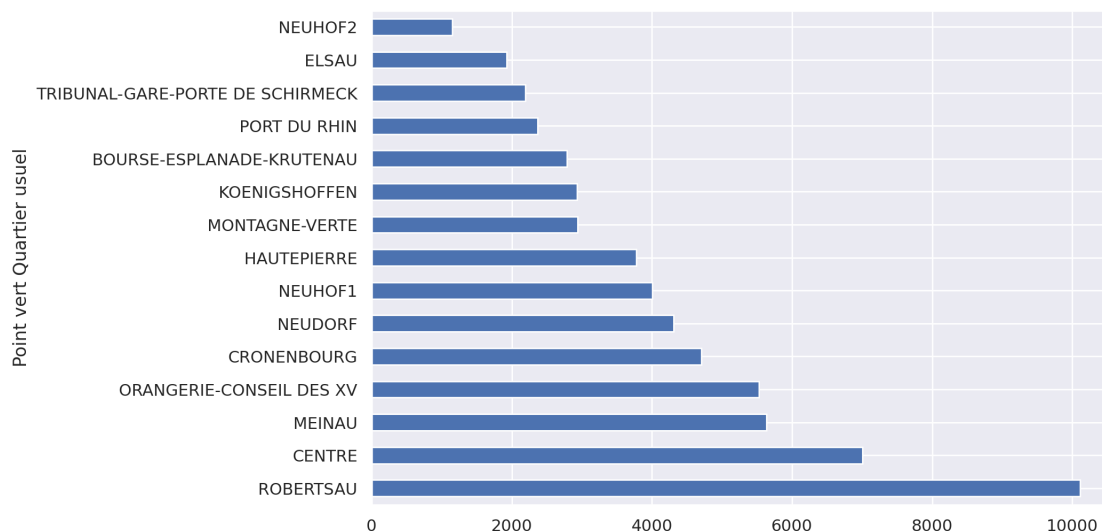
```
Out[36]: True
```

On construit une série qui contient le nombre d'arbres par quartier.

```
In [37]: arbres_quartiers = arbres['Point vert Quartier usuel'].value_counts()
```

On trace le graphique en barres correspondant.

```
In [38]: arbres_quartiers.plot(kind='barh');
```



On construit une nouvelle *Series* correspondant à l'aire de chaque quartier en m^2 . Pour que le calcul des aires soit fiables, les données de `gdf_quartier` doivent être projetées. Pour la France métropolitaine, on utilise la projection EPSG :2154, c'est-à-dire Lambert 93.

```
In [39]: aires = gdf_quartiers.to_crs(2154).area
aires.index = gdf_quartiers["libelle"]
print(aires)
print(f"\nAire totale = {aires.sum() / 10**6:.03f} km^2")
```

```

libelle
ELSAU                1.895918e+06
MONTAGNE-VERTE       2.103470e+06
ORANGERIE-CONSEIL DES XV  2.857609e+06
KOENIGSHOFFEN        2.717087e+06
...
BOURSE-ESPLANADE-KRUTENAU 1.820240e+06
NEUHOF1              3.947680e+06
MEINAU               4.777885e+06
TRIBUNAL-GARE-PORTE DE SCHIRMECK 3.359467e+06
Length: 15, dtype: float64

```

Aire totale = 78.225 km²

On calcule la densité d'arbres par hectare.

```

In [40]: densite = arbres_quartiers / aires * 10000
         densite

```

```

Out[40]: BOURSE-ESPLANADE-KRUTENAU    15.316666
         CENTRE                       39.367782
         CRONENBOURG                 12.038048
         ELSAU                       10.169214
         ...
         ORANGERIE-CONSEIL DES XV    19.344847
         PORT DU RHIN                4.874460
         ROBERTSAU                   5.607219
         TRIBUNAL-GARE-PORTE DE SCHIRMECK 6.536751
         Length: 15, dtype: float64

```

On trace une carte colorée par la densité d'arbres avec l'objet `Choropleth`.

```

In [41]: folium.Choropleth(geo_data=gdf_quartiers,
                           data=densite,
                           key_on='feature.properties.libelle',
                           fill_color='YlGn',
                           fill_opacity=0.5,
                           line_opacity=0.2,
                           legend_name=r"Nombre d\'arbres par hectare").add_to(stras_map)
stras_map.save('stras_tree.html')
display(stras_map)

```

<folium.folium.Map at 0x7f49dc5a9290>

6.4.1 Exercice

Ecrire la fonction `plot_essence()` qui prend en argument une essence d'arbres et qui trace le nombre d'arbres correspondant par quartier en utilisant `choropleth`.

```

In [42]: def plot_essence(essence):
         pass
         # Votre code ici

         plot_essence("Acer")

```

```

In [43]: # Décommentez puis exécutez pour afficher le corrigé :
         #%load exos/snippets/plot_essence.py

```

6.4.2 Utilisation des widgets ipython

On souhaite proposer à l'utilisateur un menu de sélection pour afficher le nombre d'essences par quartier. Pour limiter la taille du menu, on regroupe les essences par genre (première partie du nom latin).

```
In [44]: genres = set([nom.split()[0] for nom in essences])
```

La bibliothèque `ipywidgets` permet de générer très facilement un menu déroulant. La fonction `plot_essence()` est alors appelée avec comme

```
In [45]: from ipywidgets import interact
```

```
        interact(plot_essence, essence=sorted(genres));
```

```
interactive(children=(Dropdown(description='essence', options=('Abies', 'Acer', 'Aesculus', 'Ailanthus'
```

6.4.3 Vers des applications web

`ipywidgets` permet de faire beaucoup plus que l'exemple ci-dessus. De plus, on peut transformer facilement un notebook en application avec [voilà](#). Par exemple, transformons le notebook [exos/stras_arbres.ipynb](#) :

Dans un terminal, installer `voilà` :

```
pip install voilà
```

Exécuter `voilà` sur le notebook :

```
voilà exos/stras_arbres.ipynb
```

6.5 Références

- La [documentation officielle](#)
- Le [cours de Pierre Navaro](#)
- Le [cours de Jake Vanderplas](#)
- Des sites personnels de développeurs :
 - <http://wesmckinney.com/>
 - <https://matthewrocklin.com/>

6.6 Annexe : une autre façon de représenter les occurrences de mots

Cette fois, on n'utilise pas `pandas` mais le module `wordcloud`.

```
In [46]: from wordcloud import WordCloud
```

```
        # On crée un objet Wordcloud
```

```
        wcloud = WordCloud(background_color="white", width=480, height=480, margin=0).generate(nee)
```

```
        # On affiche l'image avec matplotlib
```

```
        plt.imshow(wcloud, interpolation='bilinear')
```

```
        plt.axis("off")
```

```
        plt.margins(x=0, y=0)
```

