

Python

**Data Science, manipuler et
visualiser les données**



Objectifs

- Posséder une vue d'ensemble de l'écosystème scientifique de Python
- Connaître les librairies scientifiques incontournables pour la science des données
- Être capable de manipuler des données volumineuses avec Python
- Comprendre l'intérêt de la data visualisation, savoir visualiser des données avec Python



Programme



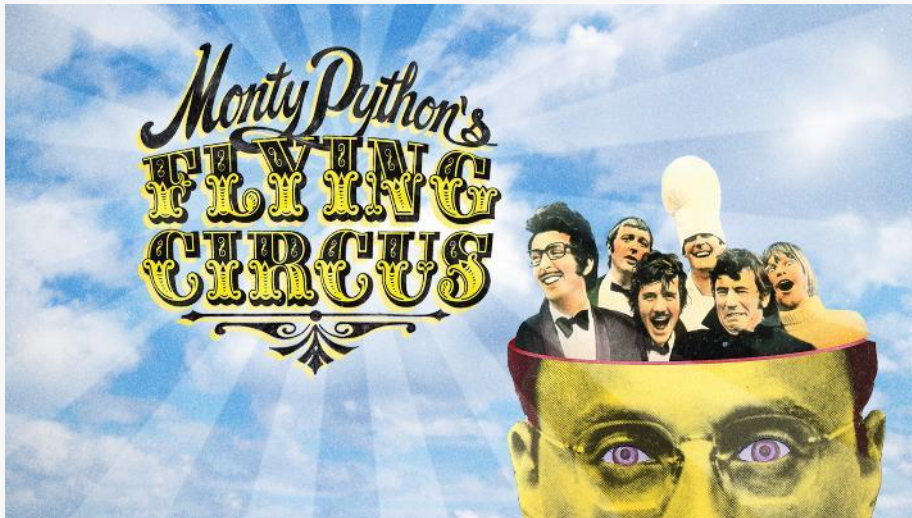
- 01** **Introduction**
- 02** **Présentation de l'écosystème scientifique Python**
- 03** **La scipy stack**
- 04** **La data visualisation et les librairies de data visualisation**
- 05** **Les formats de fichiers scientifiques et la manipulation de données volumineuses**

01

Introduction

Brève histoire de Python

- Le langage de programmation Python a été créé en 1989 par Guido van Rossum, aux Pays-Bas.
- Le nom Python est un hommage à la série télévisée Monty Python's Flying Circus, dont G. van Rossum est fan.
- La première version publique de ce langage a été publiée en 1991.



Pourquoi utiliser Python plutôt qu'un autre langage ?

Caractéristiques

Python présente de nombreuses caractéristiques intéressantes :

- Il est **multiplateforme**. En d'autres termes, il fonctionne sur un large éventail de systèmes d'exploitation : Windows, Mac OS X, Linux, Android, iOS, des mini-ordinateurs Raspberry Pi aux superordinateurs.
- Il est **gratuit**. Vous pouvez l'installer sur autant d'appareils que vous le souhaitez.
- C'est un **langage de haut niveau**. Pour l'utiliser, il faut relativement peu de connaissances sur le fonctionnement d'un ordinateur. C'est un langage interprété. Un script Python n'a pas besoin d'être compilé pour être exécuté, contrairement à des langages comme C ou C++.
- Il est **orienté objet** (mais pas seulement). En d'autres termes, il est possible de concevoir en Python des entités qui imitent celles du monde réel (une voiture, une cellule, etc.) avec un certain nombre de règles de fonctionnement et d'interactions.
- Il est **relativement facile à apprendre...**

➡ Du fait de ces caractéristiques, Python est désormais enseigné dans de nombreuses écoles et universités, et il est également largement utilisé dans les entreprises. Les entreprises utilisent Python à des fins diverses telles que le développement web, l'analyse de données, l'apprentissage automatique, l'automatisation...

Versions

- ➡ Au jour de notre formation, la dernière (sous-)version de Python est la (sous-)version [3.13](#), sortie en octobre 2024. Le dernier correctif est le 3.13.7, publiée en août 2025. La version (majeure) 2 de Python est obsolète et n'est plus maintenue... ne l'utilisez pas (vraiment !).

Python Documentation by Version

Python Documentation by Version

Some previous versions of the documentation remain available online. Use the list below to select a version to view.

For unreleased (in development) documentation, see [In Development Versions](#).

- [Python 3.12.2](#), documentation released on 6 February 2024.
- [Python 3.12.1](#), documentation released on 8 December 2023.
- [Python 3.12.0](#), documentation released on 2 October 2023.
- [Python 3.11.8](#), documentation released on 6 February 2024.
- [Python 3.11.7](#), documentation released on 4 December 2023.
- [Python 3.11.6](#), documentation released on 2 October 2023.
- [Python 3.11.5](#), documentation released on 24 August 2023.
- [Python 3.11.4](#), documentation released on 6 June 2023.
- [Python 3.11.3](#), documentation released on 5 April 2023.
- [Python 3.11.2](#), documentation released on 8 February 2023.
- [Python 3.11.1](#), documentation released on 6 December 2022.
- [Python 3.11.0](#), documentation released on 24 October 2022.



- ➡ [The Python Software Foundation](#) est l'association qui organise le développement de Python et dirige la communauté des développeurs et des utilisateurs.

Versions

Différences entre les sous-versions de Python 3 (<= 3.5)

Version	Année	Changements majeurs
3.0	2008	Rupture avec Python 2 : print() devient une fonction, Unicode par défaut, nouvelles divisions, suppression d'anciennes syntaxes.
3.1	2009	Amélioration de io, dictionnaires plus rapides, introduction de OrderedDict.
3.2	2011	Nouveau module concurrent.futures pour le parallélisme, améliorations SSL.
3.3	2012	Ajout de venv (environnements virtuels intégrés), module faulthandler, et support implicite des namespaces packages.
3.4	2014	asyncio fait son apparition, introduction de pathlib, et de enum.
3.5	2015	Nouveaux opérateurs @ (produit matriciel), introduction officielle de async/await, et unpacking amélioré (*a, b = [1,2,3]).

Versions

Différences entre les sous-versions de Python 3 (≥ 3.6)

Version	Année	Changements majeurs
3.6	2016	f-strings (<code>f"Bonjour {nom}"</code>), dictionnaires ordonnés par défaut, underscores dans les nombres (<code>1_000_000</code>).
3.7	2018	dataclasses, <code>breakpoint()</code> , ordonnancement officiel des dicts, meilleure gestion de <code>asyncio</code> .
3.8	2019	“Walrus operator” <code>:=</code> , arguments positionnels uniquement (<code>def f(a, /, b):</code>), amélioration des f-strings.
3.9	2020	Fusion de dictionnaires (<code>a b</code>)
3.10	2021	Pattern matching (<code>match/case</code>), syntaxe plus claire pour les unions de types (<code>int str</code>)
3.11	2022	Performances +25–60 %, nouveaux messages d’erreur ultra précis, meilleure gestion des coroutines.
3.12	2023	Optimisations internes, suppression de vieilles API, meilleure gestion de typing et compatibilité PEP 695 (paramètres de type).
3.13	2024	Modularisation de l’interpréteur (PEP 703 : GIL optionnel expérimental), performances accrues, simplification du cœur du langage.

Installation de Python

A partir du site officiel

- Vous pouvez installer Python directement à partir du [site officiel de Python](#). Vous pouvez par exemple suivre les étapes générales de ce [tutoriel](#), assez complet sur les différentes méthodes d'installation sur une machine « *personnelle* ».
- **Toutefois, il est essentiel de respecter les lignes directrices et les politiques établies par votre entreprise en matière d'installation et d'utilisation de logiciels, afin de garantir le respect des protocoles de sécurité et la compatibilité avec les systèmes existants.**
- Il existe également d'autres options pour travailler avec Python, en fonction de vos besoins et de vos préférences.

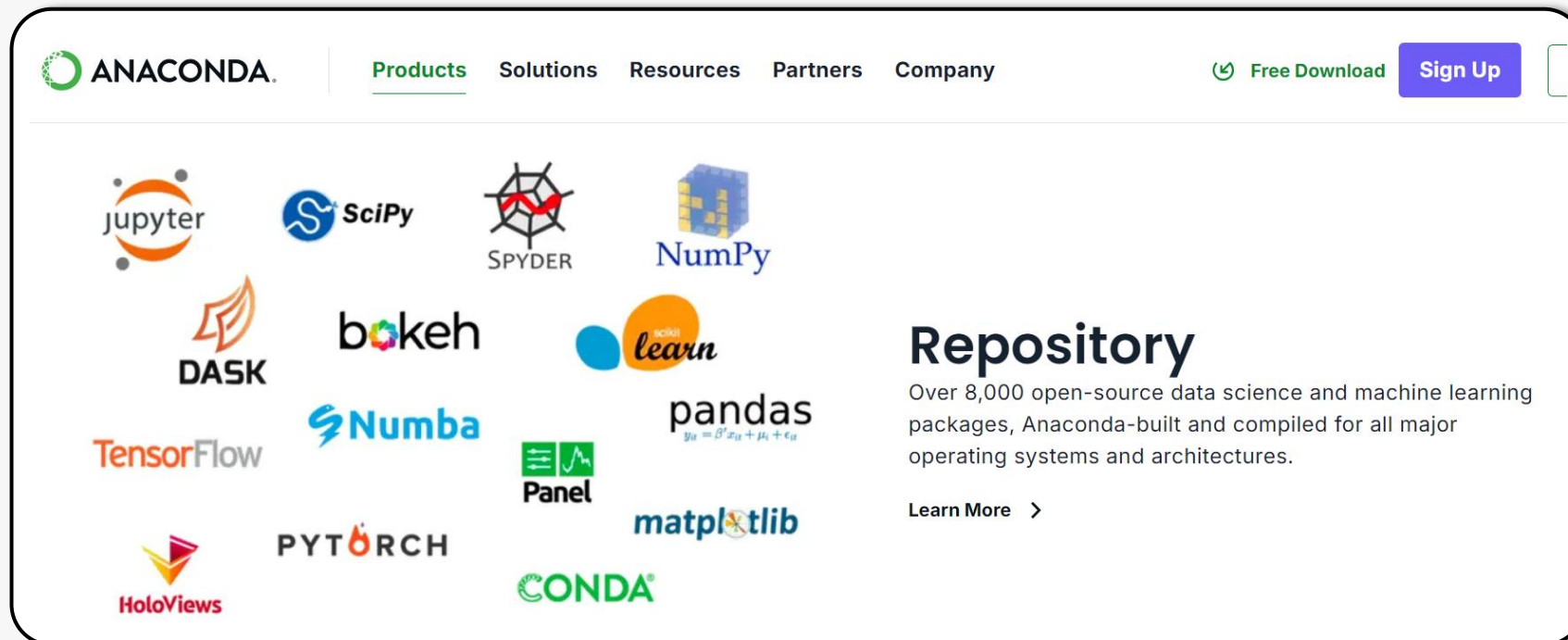


Installation de Python

Anaconda



- Anaconda est une distribution populaire des langages de programmation Python et R pour le calcul scientifique, la science des données et l'apprentissage automatique.
- Elle comprend un large éventail de paquets préinstallés et d'outils couramment utilisés dans l'analyse des données et la recherche scientifique.
- Anaconda fournit un gestionnaire de paquets appelé conda pour gérer les environnements et installer des paquets supplémentaires.



Installation de Python

Docker



- Docker est une plateforme de conteneurisation qui vous permet d'empaqueter des applications et leurs dépendances dans des conteneurs.
- Grâce à Docker, vous pouvez créer des environnements portables et reproductibles pour l'exécution d'applications Python, en garantissant la cohérence entre différents systèmes.
- Les conteneurs Docker offrent isolation et flexibilité, ce qui les rend adaptés aux flux de travail de développement, de test et de déploiement.
- Vous pouvez trouver les images Docker officielles de Python sur le [site web Docker Hub](https://hub.docker.com/_/python/).
- Pour utiliser les images Docker Python officielles, vous pouvez extraire la version Python souhaitée à l'aide de l'interface de ligne de commande (CLI) de Docker. Par exemple, pour obtenir la dernière image Python 3, vous pouvez utiliser la commande suivante :

```
docker pull python:3
```

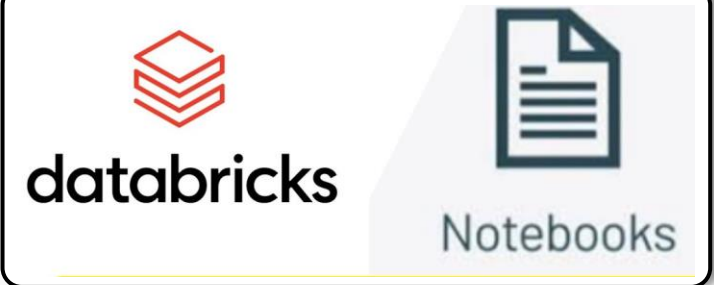
Une fois l'image téléchargée, vous pouvez créer et exécuter un conteneur basé sur l'image Python à l'aide de la commande `docker run`, en spécifiant les paramètres supplémentaires nécessaires.

- Docker fournit une documentation et des ressources complètes pour travailler avec les conteneurs et les images Docker, ce qui facilite l'intégration de Docker dans votre flux de travail de développement Python.

Installation de Python

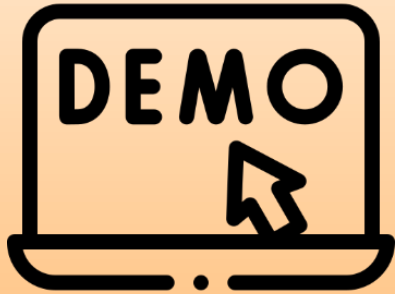
Manipuler Python sans avoir à l'installer

- Pour manipuler Python sans avoir à l'installer, vous pouvez utiliser des interpréteurs Python en ligne ou des plateformes basées sur le cloud qui offrent des environnements Python.
- Les interpréteurs Python en ligne tels que Repl.it, PythonAnywhere, IDEOne... fournissent des interpréteurs Python en ligne où vous pouvez écrire et exécuter du code Python dans votre navigateur. Ces plateformes offrent des environnements Python de base et la prise en charge des bibliothèques Python les plus courantes.
- Les notebooks Jupyter dans le cloud comme Google Colab, Microsoft Azure Notebooks ou Databricks fournissent des environnements où vous pouvez créer et exécuter des notebooks Python sans installer Python localement. Ces plateformes offrent des fonctionnalités supplémentaires telles que l'édition collaborative, le stockage sur le cloud et l'intégration avec d'autres services cloud.



Installation de Python

Setup



**Préparation de notre
environnement de travail**

Installation de Python

Hello, World avec Python installé

Voici comment créer votre premier programme « Hello, World ! » en Python, selon que vous l'avez installé localement ou que vous travaillez sur un notebook en ligne.

➡ Si vous avez Python installé localement :

- Ouvrez l'éditeur de texte de votre choix (comme Notepad sous Windows, TextEdit sous Mac, ou un éditeur de code comme Visual Studio Code, PyCharm, etc.)
- Tapez le code suivant : `print("Hello, World!")`
- Enregistrez le fichier avec l'extension « .py » (par exemple, hello_world.py).
- Ouvrez une invite de commande (terminal sur Mac/Linux, Command Prompt sur Windows).
- Naviguez jusqu'au répertoire où vous avez enregistré votre fichier hello_world.py. Tapez la commande `python hello_world.py` et appuyez sur Entrée.

Installation de Python

Hello, World avec un notebook en ligne

➡ Si vous travaillez avec un notebook en ligne

- Ouvrez votre plateforme de notebooks en ligne.
- Créez un nouveau notebook ou ouvrez un notebook existant.
- Dans une cellule de code, tapez le code suivant : `print("Hello, World!")`
- Exécutez la cellule de code. Dans la plupart des notebooks, vous pouvez le faire en appuyant sur les touches Maj + Entrée. Vous devriez voir le message « Hello, World! » affiché sous la cellule de code.

Installation de Python

Editeurs et IDE

Principaux environnements de développement intégré (IDE) pour Python :

- **Visual Studio Code (VSCode)**
Développé par Microsoft.
Éditeur de code léger mais riche en fonctionnalités, avec un vaste écosystème d'extensions.
Prend en charge Python à travers ces extensions.
- **PyCharm**
Développé par JetBrains.
IDE puissant avec des fonctionnalités telles que la complétion de code, la coloration syntaxique, des outils de débogage, un contrôle de version intégré...
- **Spyder**
IDE open-source.
Conçu spécifiquement pour le calcul scientifique et l'analyse de données avec Python.
Inclut des fonctionnalités telles que l'explorateur de variables, le débogage, l'intégration avec des bibliothèques scientifiques...

Installation de Python

Editeurs et IDE

- Il est également possible d'utiliser **des environnements de développement interactifs basés sur le web**, comme JupyterLab pour les notebooks Jupyter, qui sont largement utilisés dans la science des données et la recherche, et qui prennent en charge Python ainsi que d'autres langages.
- Enfin, vous pouvez utiliser l'**éditeur de texte** de votre choix (Sublime Text, Atom, Vim, Emacs, Notepad++, Geany, Komodo Edit, Brackets, TextMate, GNU nano, Kate, gedit, Bluefish...).

Installation de Python

Les environnements

En Python, les environnements sont des espaces isolés qui permettent de gérer des dépendances et des bibliothèques pour des projets spécifiques. Ils jouent un rôle crucial dans la gestion des versions et des conflits de dépendances entre différents projets.

Pourquoi utiliser des environnements ?

➡ Isolation des projets

Chaque projet a ses propres bibliothèques et versions sans interférer avec d'autres projets. Par exemple, un projet utilise Django 3.2 tandis qu'un autre utilise Django 4.0.

➡ Éviter les conflits de dépendances

Les environnements isolés garantissent que les dépendances spécifiques à un projet ne perturbent pas d'autres projets.

➡ Facilité de partage et de reproductibilité

Il est possible de partager l'environnement exact d'un projet via un fichier comme `requirements.txt` ou `environment.yml`.

Installation de Python

Les environnements

Environnements virtuels avec venv ou virtualenv

- Ce sont des environnements locaux créés pour un projet spécifique.

- Créer un environnement :

```
python -m venv myenv
```

```
python3.12 -m venv myenv
```

- Activer l'environnement :

Windows :

```
myenv\Scripts\activate
```

macOS/Linux :

```
source myenv/bin/activate
```

- Installer des bibliothèques dans l'environnement :

```
pip install <package>
```

- Désactiver l'environnement :

```
deactivate
```

Installation de Python

Les environnements

Environnements Conda

- Conda est un outil plus large utilisé pour gérer les environnements et les dépendances, souvent utilisé avec des projets scientifiques
- Créer un environnement :

```
conda create -n myenv
```

```
conda create -n myenv python=3.12
```
- Activer l'environnement :

```
conda activate myenv
```
- Installer des bibliothèques dans l'environnement :

```
conda install <package>
```
- Désactiver l'environnement :

```
conda deactivate
```

Installation de Python

Les environnements

Environnements basés sur Docker



- Utilise des conteneurs pour isoler complètement un projet, y compris son environnement système.
- Avantages : isolation totale, partage facile, intégration avec des configurations système spécifiques.
- On utilise un fichier `Dockerfile` pour définir l'environnement.

Installation de Python

Les environnements

Gestionnaires de versions et d'environnements comme pyenv

- Permettent de gérer différentes versions de Python sur une machine.
- Peuvent être combinés avec des outils comme pyenv-virtualenv pour créer des environnements spécifiques à des versions de Python.

- Installer une version de Python :

```
pyenv install 3.12.0
```

- Créer un environnement virtuel basé sur cette version spécifique :

```
pyenv virtualenv 3.12.0 myenv
```

Installation de Python

Les environnements

Fichiers associés aux environnements

➡ requirements.txt (pour pip) :

```
pip freeze > requirements.txt
```

```
pip install -r requirements
```

➡ environment.yml (pour Conda)

```
conda env export > environment.yml
```

```
conda env create -f environment.yml
```

```
numpy==1.21.2  
pandas>=1.3.0
```

```
name: myenv  
dependencies:  
  - python=3.12  
  - numpy  
  - pandas
```


Installation de Python

Les environnements

- ➡ **conda dans venv** : Ce n'est pas la méthode idéale, bien que techniquement possible, et cela risque de créer des problèmes de gestion des dépendances.
- ➡ **pip dans conda** : Tout à fait possible, et souvent utilisé pour ajouter des paquets non présents dans les canaux Conda. Cependant, il est préférable de gérer les dépendances principales avec Conda et d'utiliser pip pour les paquets supplémentaires.

Installation de Python

Les environnements

Bonnes pratiques

- Créer **un environnement pour chaque projet** (cela garantit l'isolation et évite les conflits).
- Versionner les fichiers de dépendances (i.e. inclure requirements.txt ou environment.yml dans le contrôle de version).
- Désactiver un environnement lorsqu'il n'est plus nécessaire (évite des comportements inattendus liés aux dépendances).
- Utiliser des gestionnaires de versions Python comme pyenv (pour tester facilement le code sur différentes versions de Python).

Installation de Python

Les environnements

Workflow typique (Linux)

```
# Créer un environnement
python -m venv myenv

# Activer l'environnement
source myenv/bin/activate

# Installer les dépendances
pip install flask

# Générer un fichier des dépendances
pip freeze > requirements.txt

# Désactiver l'environnement
deactivate
```

Fonctionnement de l'interpréteur Python

1/2

- Python est un langage interprété, ce qui signifie que chaque ligne de code est lue et interprétée pour être exécutée par l'ordinateur. Pour voir cela en action, ouvrez un shell et entrez la commande suivante :

```
>>> python
```

- La commande ci-dessus lancera l'interpréteur Python. Vous devriez voir quelque chose comme ceci :

```
C:\Users\bspeziale>python
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct 2 2024, 13:03:39) [MSC v.1935 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Capture d'écran sur Windows. Elle serait similaire sur Mac et Linux.

Fonctionnement de l'interpréteur Python

2/2

- L'interpréteur Python est un système interactif dans lequel vous pouvez entrer des commandes que Python exécutera sous vos yeux (lorsque vous validez la commande en appuyant sur la touche Entrée). A ce stade, vous pouvez entrer une autre commande ou quitter l'interpréteur Python. Pour quitter, vous pouvez taper la commande **exit()** et appuyer sur Entrée.
- Il existe de nombreux autres langages interprétés comme Perl ou R. Le grand avantage de ce type de langage est que vous pouvez **immédiatement tester une commande** à l'aide de l'interpréteur, ce qui est très utile pour le débogage (trouver et corriger les erreurs potentielles d'un programme).

Les conventions de codage

Les PEP (Python Enhancement Proposals)

- Documents officiels utilisés dans la communauté Python pour proposer, discuter et documenter les modifications ou évolutions du langage Python, des bibliothèques, ou des processus qui régissent son développement. Ils permettent de structurer et de standardiser la manière dont les idées sont présentées et débattues, tout en servant de référence pour les décisions futures.
- **Documentation formelle** : Les PEP offrent une description claire et détaillée des nouvelles fonctionnalités ou changements proposés.
- **Transparence** : Ils rendent le processus de décision participatif et ouvert à la communauté Python.
- **Normalisation** : Ils établissent des standards (comme le style de codage ou l'utilisation de certaines conventions) pour améliorer la cohérence et la lisibilité du code.
- **Références historiques** : Ils gardent une trace des décisions passées, permettant de comprendre pourquoi certaines fonctionnalités ont été adoptées, modifiées ou rejetées.

<https://peps.python.org/>

Les conventions de codage

Le PEP 8

- **"Style Guide for Python Code"** : l'un des documents les plus emblématiques de la communauté Python. Il définit les conventions de style que les développeurs Python doivent suivre pour écrire un code propre, lisible et cohérent.
- **Lisibilité du code** : Le principal objectif est d'améliorer la lisibilité du code Python, facilitant ainsi sa compréhension par d'autres développeurs.
- **Uniformité** : Promouvoir des conventions communes pour que tous les développeurs Python puissent travailler sur des projets avec un style homogène.
- **Faciliter la collaboration** : En suivant les mêmes standards, les équipes peuvent collaborer plus efficacement sans perdre de temps à reformater ou interpréter des styles de code variés.
- **Professionalisation** : Adopter un style cohérent et professionnel améliore la qualité globale des projets Python.

<https://peps.python.org/pep-0008/>



Nous reviendrons sur les recommandations du PEP8 au fil de la formation.

Debugage

Panneau "Exécuter et déboguer", points d'arrêt



La 4^e icône dans la barre d'icônes verticale à gauche dans VSCode est consacrée au debugging. Il est possible d'ouvrir le **Debugger** en cliquant dessus (ou avec le raccourci Ctrl-Shift-D) :

- Affiche les panneaux **Variables**, Surveiller (**Watch**), Pile d'appels (**Call Stack**) et **Points d'arrêt**.
- Permet de choisir une configuration de débogage ou de modifier les options de lancement.

Un **point d'arrêt** (*breakpoint* en anglais) est un outil de débogage dans la plupart des IDEs qui permet de mettre son programme en pause à un endroit précis pour pouvoir l'analyser.

Il est alors possible de lancer le debugage (raccourci : F5).

02

Présentation de l'écosystème scientifique Python

Présentation de l'écosystème scientifique Python

Savoir où trouver de nouvelles librairies...

PyPI (Python Package Index)	Le principal dépôt pour les bibliothèques Python.	https://pypi.org
Github	Server Git. De nombreuses bibliothèques Python sont hébergées sur GitHub. C'est une excellente plateforme pour découvrir des projets open source et voir les contributions en temps réel	https://github.com/search
Awesome Python	Une liste collaborative et bien entretenue des meilleures bibliothèques Python. C'est un excellent point de départ pour trouver des outils spécifiques et éprouvés.	https://awesomepython.org/
Forums	<p>Le subreddit r/Python propose régulièrement des discussions autour de nouvelles bibliothèques Python, avec des retours d'expérience de la communauté</p> <p>Stack Overflow : les réponses aux questions de développement incluent souvent des recommandations pour des bibliothèques populaires ou émergentes. C'est aussi un bon indicateur de la communauté autour d'une bibliothèque</p>	<p>https://www.reddit.com/r/Python</p> <p>https://stackoverflow.com/</p>
Newsletters	Les (multiples) newsletters partagent des découvertes de nouvelles bibliothèques Python, majoritairement à fréquence hebdomadaire, ainsi que des discussions autour des outils tendances.	

Présentation de l'écosystème scientifique Python





... et comment juger de leur pérennité

- **Popularité/adoption** (nombre de téléchargements dans la section "Downloads" sur PyPI, nombre de forks et d'étoiles sur GitHub, mentions dans des articles et utilisation dans des projets populaires/des entreprises bien établies)
- **Fréquence des mises à jour** (date de la dernière mise à jour, historique des commits, cycle de versions régulières, changements documenté)
- **Support de la communauté et réactivité** (issues, temps de réponse des mainteneurs)
- **Documentation** (complétude, exactitude, clarté, structure...) Tests et intégration continue (couverture de tests automatisé, badges d'intégration)
- **Dépendances et compatibilité** (dépendances minimales et bien gérées, suivi des versions récentes de Python)

Et, même si c'est de nature différente, ne pas oublier de se renseigner sur la licence et les conditions d'utilisation !

Présentation de l'écosystème scientifique Python

Au-delà de l'écosystème scientifique, l'écosystème technique pour la data science

IDE :  (Visual Studio Code)  (PyCharm)  / Jupyterlab  spyder




Manipulation et traitement de données :  pandas  dask  PySpark  (Vaex) et bien d'autres

Visualisation de données :  (matplotlib)  seaborn  plotly  bokeh  + a b l e a u  Power BI

Machine Learning et Deep Learning :  scikit-learn  XGBoost  LightGBM  TensorFlow  K Keras 

Gestion des bases de données :  MySQL  PostgreSQL  MongoDB  SQLite

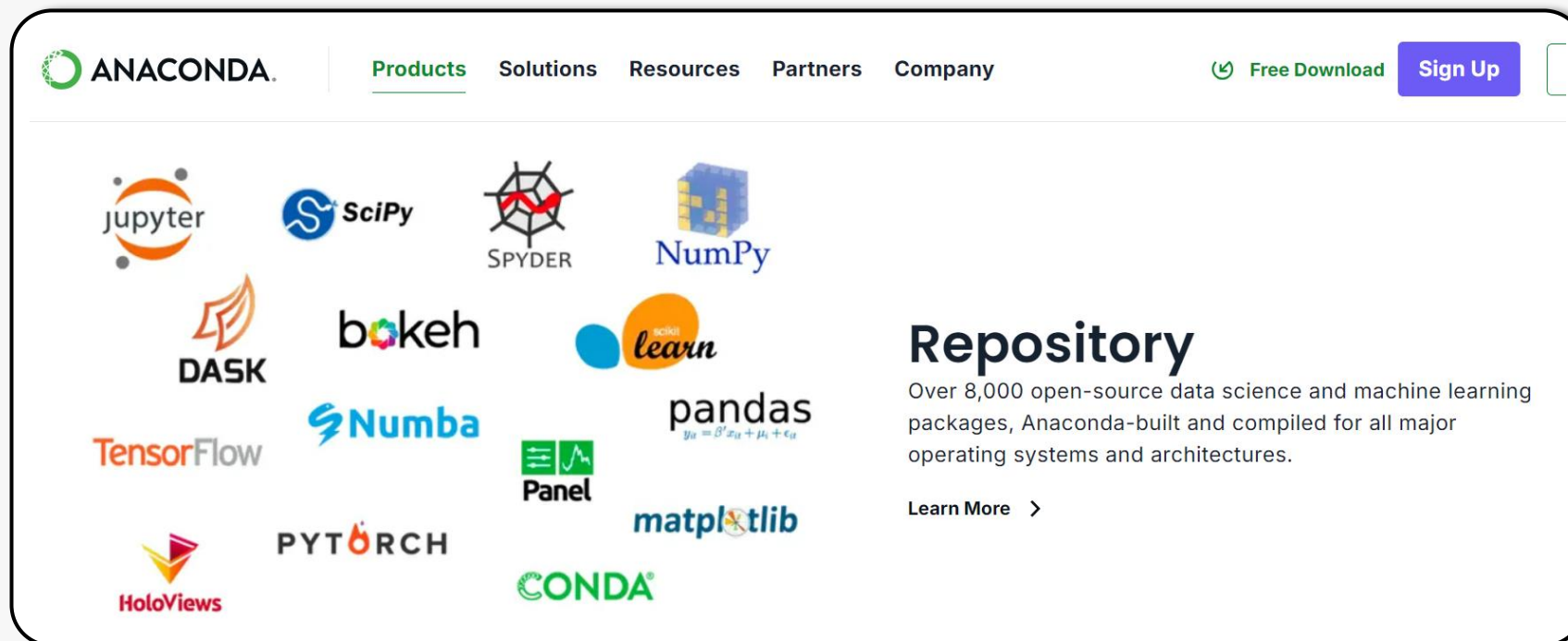
Big Data :  hadoop  PySpark  dask  kafka

Gestion des versions et collaboration :  git  GitLab  GitHub

Présentation de l'écosystème scientifique Python

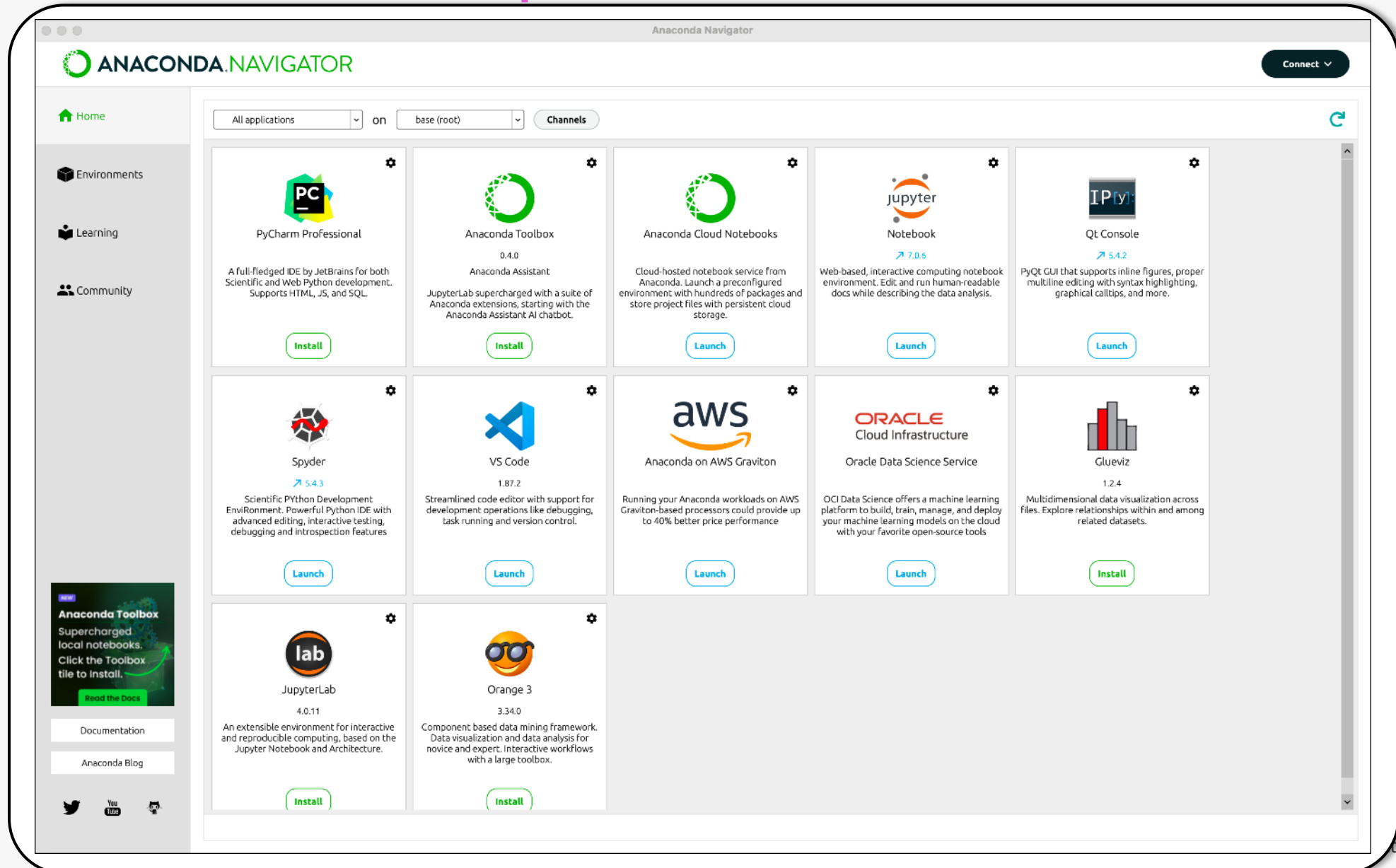
Utiliser une distribution scientifique comme Anaconda

- Il existe des distributions de Python qui simplifient l'installation, la gestion de packages et les environnements virtuels pour les projets de Data Science et d'apprentissage automatique.
- Pour un scientifique, il peut être intéressant d'utiliser une telle distribution.
- Anaconda est une de ces distributions (c'est aussi une distribution de R). Elle est open-source.



Présentation de l'écosystème scientifique Python

Utiliser une distribution scientifique comme Anaconda



Présentation de l'écosystème scientifique Python

Pourquoi utiliser une distribution scientifique, sur l'exemple d'Anaconda

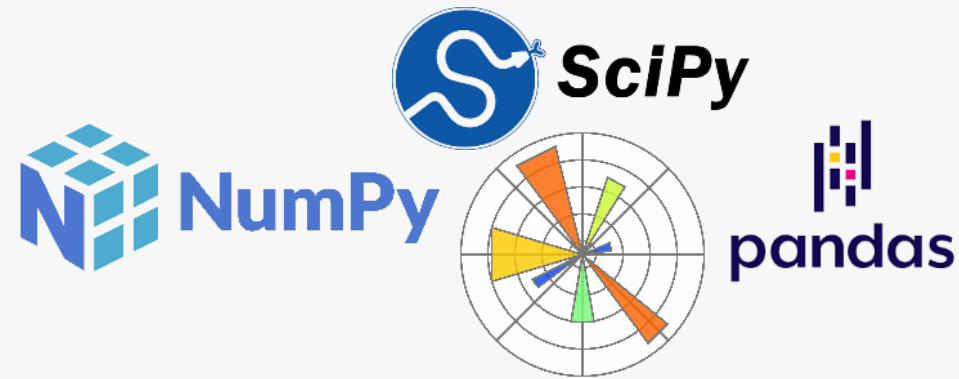
- Une distribution comme Anaconda va **faciliter l'installation des bibliothèques scientifiques** complexes qui, autrement, nécessiteraient plusieurs dépendances difficiles à gérer. Lors de l'installation d'Anaconda, il précharge plus de 1 500 packages scientifiques populaires (NumPy, Pandas, Scikit-learn, Matplotlib, Jupyter, SciPy...) Cela réduit les efforts pour configurer un environnement scientifique complet (gain de temps, résolution automatique des dépendances).
- Anaconda inclut un **gestionnaire de paquets et d'environnements** appelé Conda. Il permet de créer facilement des environnements virtuels isolés avec différentes versions de Python et des packages spécifiques à chaque projet.
- Anaconda inclut par défaut **Jupyter Notebook** et **JupyterLab**, des outils interactifs très populaires en data science. Ils permettent d'exécuter du code, de visualiser des graphiques, et de documenter des résultats dans des notebooks réutilisables.
- Anaconda fonctionne sur **Windows, macOS, et Linux**, ce qui rend le développement et le partage de projets entre différentes plateformes simple.
- Contrairement à pip, qui ne gère que les packages Python, Conda peut gérer des **packages non Python** (comme des bibliothèques C ou des compilateurs), ce qui est peut parfois être utile pour des outils dépendant de composants en C++, CUDA...

03

La scipy stack

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne



Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - pandas



- Pandas est une bibliothèque de **manipulation** et d'**analyse de données** en Python. Elle fournit des structures de données performantes et faciles à utiliser, comme les DataFrames, qui permettent de travailler avec des données tabulaires.
- Lancée en 2008 par Wes McKinney, Pandas s'est rapidement imposée comme un outil incontournable pour la science des données. Elle est utilisée pour importer, nettoyer, transformer et analyser des ensembles de données hétérogènes.
- Pandas est optimisée pour la **manipulation de grands ensembles de données** grâce à ses capacités d'intégration avec NumPy et d'autres bibliothèques Python.

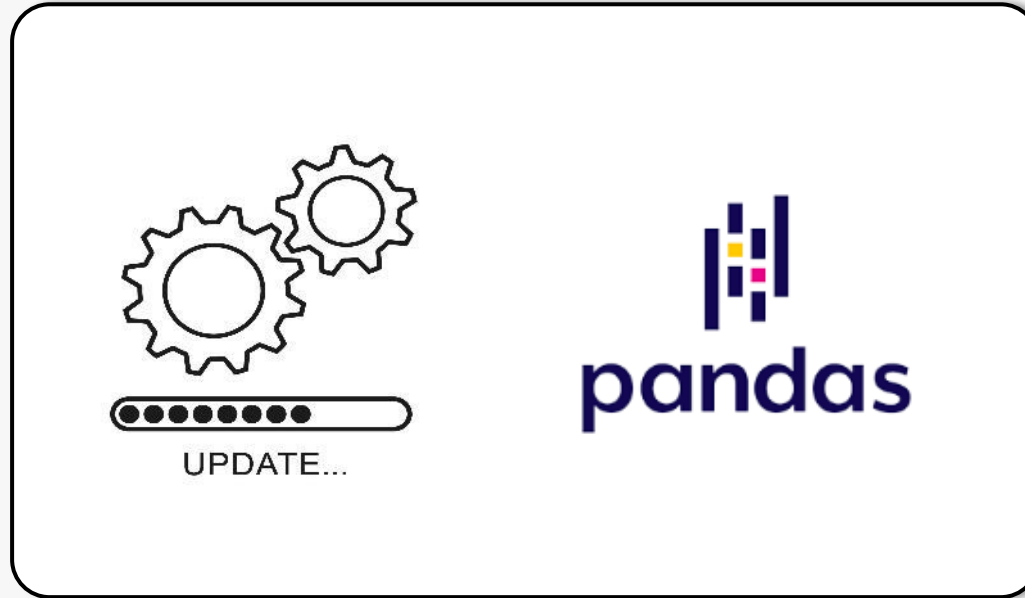
Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - pandas

- Les **DataFrames** de Pandas permettent de manipuler des données organisées en lignes et colonnes, semblables à des feuilles de calcul Excel. Pandas prend également en charge des index, ce qui facilite l'accès, le tri, et la sélection des données.
- La bibliothèque est idéale pour les tâches de **nettoyage de données**, de **fusion d'ensembles de données**, et de **calcul de statistiques descriptives**. Elle est souvent utilisée en tandem avec d'autres outils comme Matplotlib pour la visualisation ou scikit-learn pour l'apprentissage automatique.

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - pandas



```
pip install --upgrade pandas
```

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - pandas

`df = pandas.DataFrame()` crée une table de données.

`df.head()` affiche les premières lignes, `df.tail()` affiche les dernières lignes.

`df.describe()` fournit un résumé statistique des colonnes numériques.

`df.groupby()` regroupe les données pour appliquer des opérations agrégées.

`df.merge()` fusionne plusieurs DataFrames.

`df.isna()` Identifie les valeurs manquantes, `df.fillna()` les remplit.

Pandas est compatible avec divers formats (CSV, Excel, JSON, SQL, HDF5...).

La documentation de pandas est disponible à l'adresse suivante :
<https://pandas.pydata.org/docs/>

Présentation de l'écosystème scientifique Python

TP pandas #1



**Utiliser pandas pour
manipuler des données
tabulaires temporelles
(données Vélib)**

Présentation de l'écosystème scientifique Python

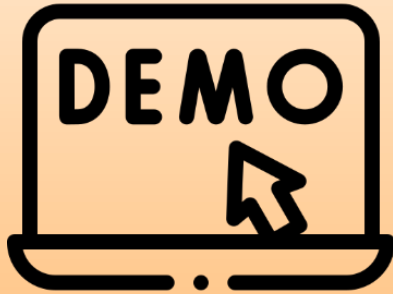
TP pandas #2



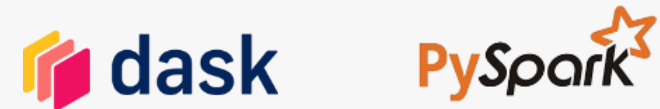
**Identifier des données
aberrantes**

Présentation de l'écosystème scientifique Python

Distribuer des traitements type « pandas »



Traitement équivalent
avec dask et pyspark



Présentation de l'écosystème scientifique Python

TP bilan pandas



**Calcul de deltas dans un
portefeuille boursier**

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - NumPy



- NumPy, abréviation de "Numerical Python", est une bibliothèque fondamentale en Python pour le **calcul numérique** et la **manipulation de tableaux multidimensionnels**. Elle est largement utilisée dans divers domaines, y compris le traitement d'images.
- Elle a été créée en 2005 par Travis Oliphant en se basant sur des travaux précédents. NumPy est open source et a connu une adoption rapide parmi les communautés scientifiques et d'ingénierie. Elle a posé les bases pour de nombreuses autres bibliothèques de calcul scientifique en Python (scipy, matplotlib, scikit-learn...).

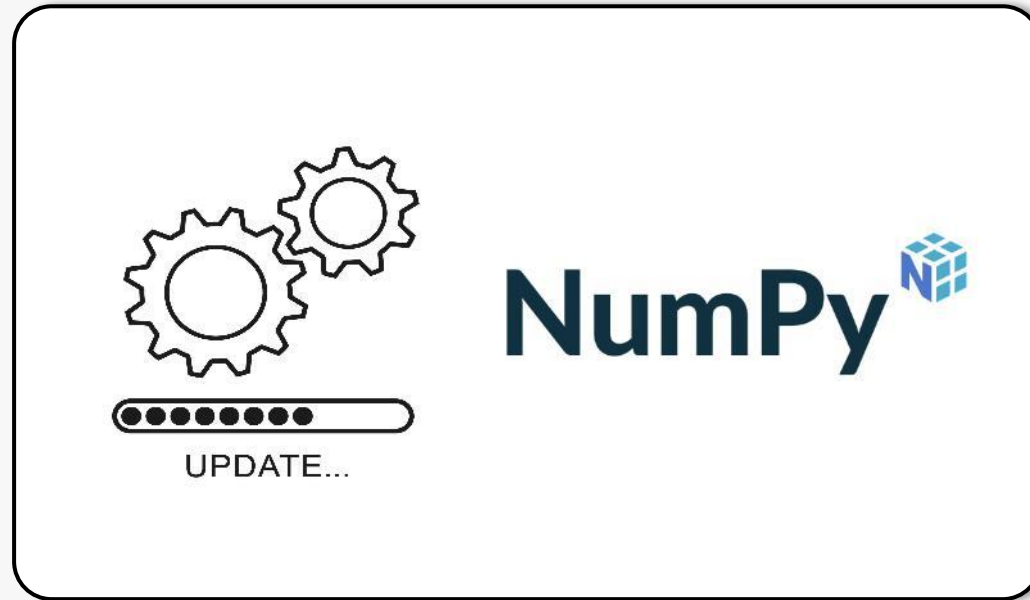
Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - NumPy

- NumPy est construite autour du concept de tableaux NumPy (ndarrays), qui sont des structures de données multidimensionnelles homogènes et efficaces pour stocker et manipuler des données numériques.
- Les **tableaux NumPy** (arrays) sont similaires aux listes Python, mais ils sont optimisés pour les opérations numériques. Ils peuvent avoir des dimensions multiples et prennent en charge des opérations vectorielles, ce qui les rend idéaux pour le calcul numérique.
- NumPy offre des fonctions pour effectuer des **opérations vectorielles** sur des tableaux, ce qui permet d'effectuer des calculs efficaces et parallèles sur de grandes quantités de données.
- NumPy est couramment utilisée pour **traiter et analyser des données**, notamment dans les domaines de la science des données, de l'apprentissage automatique et du traitement d'images.

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - NumPy



```
pip install --upgrade numpy
```

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - NumPy

`numpy.array()` crée un tableau NumPy à partir d'une séquence de données.

`numpy.zeros()`, `numpy.ones()`, `numpy.empty()` créent des tableaux remplis de zéros, de uns ou de valeurs vides. Il faut en préciser la taille.

`numpy.arange()`, `numpy.linspace()` génèrent des séquences de nombres.

`numpy.shape()`, `numpy.reshape()`, `numpy.dim()` permettent de manipuler la forme et les dimensions des tableaux.

`numpy.mean()`, `numpy.max()`, `numpy.min()`, `numpy.std()` calculent des statistiques sur les données.

Il est possible d'accéder aux et de manipuler les éléments d'un tableau en utilisant des **indices** et des opérations de découpage (**slicing**).

La documentation de numpy est disponible à l'adresse suivante :

<https://numpy.org/doc/>

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - SciPy



- SciPy, abréviation de "Scientific Python", est une bibliothèque conçue pour le **calcul scientifique et technique**. Elle repose sur NumPy et fournit une large gamme d'algorithmes pour des tâches telles que l'intégration, l'optimisation, l'interpolation, l'algèbre linéaire et le traitement du signal.
- Créée en 2001 par Travis Oliphant, Eric Jones, et Pearu Peterson, SciPy a joué un rôle crucial dans l'essor de Python en tant que langage phare pour la science des données et la recherche. Elle est open source et bénéficie d'une communauté active de développeurs et d'utilisateurs.
- SciPy est conçue pour être modulaire, avec plusieurs sous-modules spécialisés comme `scipy.integrate` (intégration), `scipy.optimize` (optimisation), ou `scipy.spatial` (géométrie et distances).

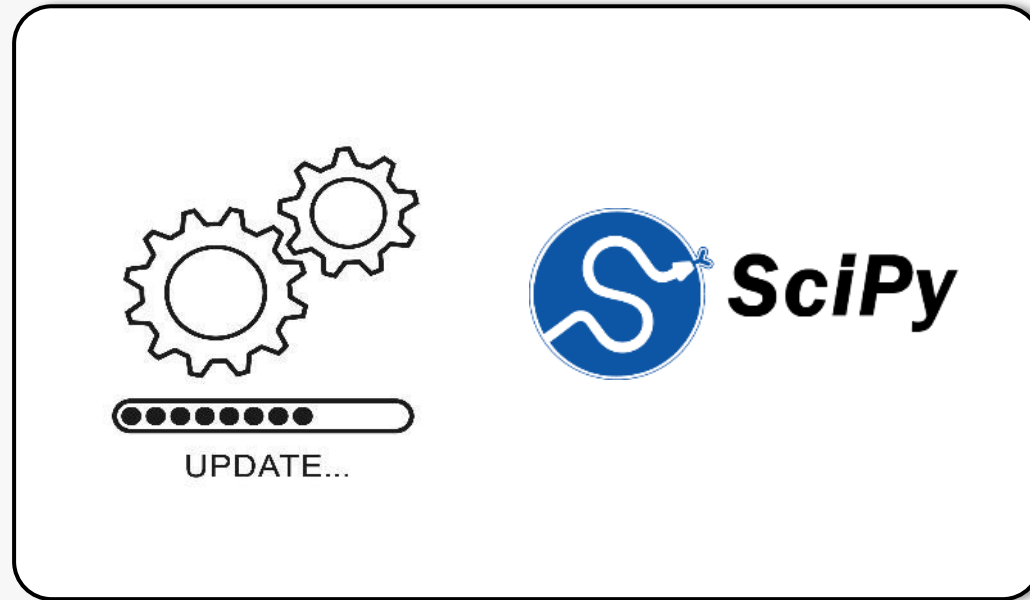
Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - SciPy

- SciPy s'appuie sur NumPy pour manipuler les données et se concentre sur les outils d'analyse avancés. Elle est couramment utilisée dans des domaines variés comme l'analyse des données expérimentales, la physique, la bioinformatique et l'ingénierie.
- Ses modules permettent de résoudre des équations différentielles, d'effectuer des décompositions matricielles avancées, ou encore d'analyser des distributions statistiques. Par exemple, la fonction `scipy.stats` fournit des outils pour réaliser des tests d'hypothèse ou ajuster des distributions.
- Avec ses performances optimisées, SciPy permet de résoudre des problèmes complexes de manière efficace.

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - SciPy



```
pip install --upgrade scipy
```


Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - SciPy

`scipy.optimize.minimize()` résout des problèmes d'optimisation.

`scipy.integrate.quad()` calcule l'intégrale d'une fonction.

`scipy.interpolate.interp1d()` interpole des points de données.

`scipy.linalg.eig()` calcule les valeurs propres d'une matrice.

`scipy.stats.norm.pdf()` donne la densité de probabilité d'une loi normale.

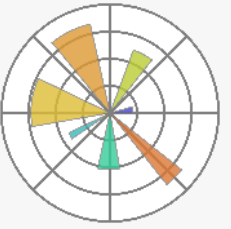
SciPy permet également de manipuler des structures de données comme les matrices creuses via `scipy.sparse`.

La documentation de scipy est disponible à l'adresse suivante :

<https://docs.scipy.org/doc/scipy/>

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - Matplotlib



- **Matplotlib** est une bibliothèque de **visualisation** de données flexible pour Python. Elle est utilisée pour créer une grande variété de graphiques, de tracés et de visualisations, que ce soit pour des analyses de données, des présentations ou des rapports scientifiques. Elle est également capable d'afficher et de traiter des images de base, ce qui en fait un outil polyvalent pour la visualisation et le traitement d'image.
- Elle a été créée par John D. Hunter en 2003. Il a développé cette bibliothèque pour aider les scientifiques et les ingénieurs à visualiser leurs données de manière efficace et esthétique.

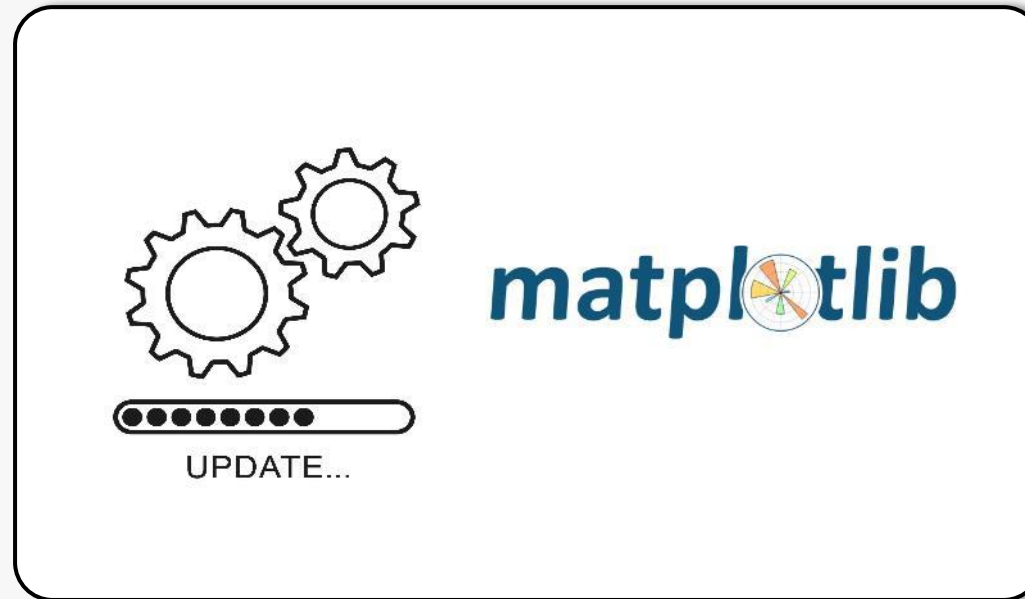
Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - Matplotlib

- Matplotlib fonctionne en fournissant une **interface orientée objet** pour la création de graphiques et de tracés en Python. Le sous-module `pyplot` fournit une interface simplifiée pour construire des visualisations rapidement.
- Matplotlib permet aux utilisateurs de créer des **figures** et des **axes** qui peuvent être personnalisés de manière approfondie. Les utilisateurs peuvent ajouter des éléments tels que des lignes, des points, des légendes, des titres... aux graphiques de manière « programmatique ».
- Matplotlib prend en charge une **grande variété de types de graphiques**, notamment les graphiques en nuages de points, les histogrammes, les graphiques à barres, les graphiques en boîte, les graphiques en courbes, les graphiques de dispersion, les graphiques en secteurs, etc.
- Matplotlib permet une **personnalisation avancée** des graphiques, avec la possibilité de contrôler chaque aspect visuel, y compris les couleurs, les styles de ligne, les polices, les étiquettes, les axes et les échelles.
- Elle est souvent utilisée avec Pandas pour tracer des données tabulaires ou avec NumPy pour visualiser des données numériques brutes.

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - Matplotlib



```
pip install --upgrade matplotlib
```

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - Matplotlib

`plt.figure()`, `plt.subplot()` permettent de créer des figures et des sous-graphiques.

`plt.plot()`, `plt.scatter()`, `plt.bar()` créent différents types de graphiques.

`plt.xlabel()`, `plt.ylabel()`, `plt.title()`, `plt.legend()` ajoutent des éléments de légende/titres à la visualisation

`plt.savefig()` sauvegarde le graphique dans un fichier image.

La documentation de matplotlib est disponible à l'adresse suivante :
<https://matplotlib.org/stable/index.html>

Présentation de l'écosystème scientifique Python

Le stack scientifique plus au complet



Où est Charlie ?

Présentation de l'écosystème scientifique Python

Le stack scientifique plus au complet



Apple stock price data

Présentation de l'écosystème scientifique Python

L'écosystème scientifique Python moderne - scikit-learn

scikit-learn est une (excellente) bibliothèque Python pour le Machine Learning, offrant un ensemble d'outils **standardisés** pour l'apprentissage supervisé et non supervisé. Elle est couramment utilisée comme **catalogue** d'algorithmes implémentés, et plus généralement pour **construire des modèles**, transformer les données et **évaluer** les performances des algorithmes de Machine Learning.



Prétraitement des données

scikit-learn inclut des méthodes pour normaliser, transformer et manipuler les données afin de les préparer pour le modèle

Algorithmes de Machine Learning

scikit-learn propose un (large) éventail de modèles pour des tâches de classification, régression, clustering, réduction de dimensionnalité, et sélection de modèles

Évaluation des modèles

scikit-learn fournit des fonctions pour évaluer la performance des modèles grâce à des métriques standard et des techniques de validation croisée

Optimisation de modèles

scikit-learn permet de rechercher les meilleurs hyperparamètres pour un modèle à l'aide de techniques telles que GridSearchCV et RandomizedSearchCV

04

La data visualisation et les librairies de data visualisation

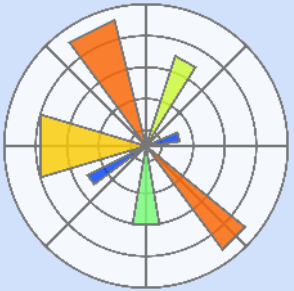
Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Visualisation 2D/3D



Matplotlib



C'est la librairie de visualisation la plus populaire pour les graphiques 2D en Python. Elle permet de créer des graphiques classiques tels que des histogrammes, des graphiques en courbes, des diagrammes en boîte, etc.

Ex. : Graphiques linéaires, scatter plots, histogrammes.



Seaborn



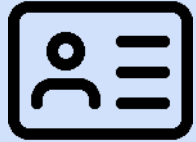
Construite sur Matplotlib, Seaborn simplifie la création de visualisations statistiques avancées. Elle est particulièrement adaptée aux visualisations basées sur des ensembles de données complexes.

Ex. : Cartes thermiques, boxplots, pairplots, distribution de données.

Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Visualisation 2D/3D



Plotly



Offre des graphiques interactifs et des visualisations en 3D. Permet de créer des graphiques très esthétiques et interactifs qui peuvent être intégrés à des sites web.

Ex. : Graphiques interactifs en 2D/3D, cartes, visualisations de séries temporelles.



Bokeh



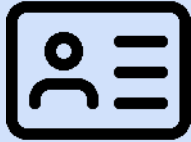
Permet de créer des visualisations interactives en 2D et en 3D, souvent utilisées pour des dashboards. Prend en charge les web apps en temps réel.

Ex. : Visualisation de séries temporelles, diagrammes en barres interactifs.

Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Visualisation 2D/3D



Altair



Librairie déclarative pour la visualisation de données. Utilise un langage basé sur JSON pour décrire les visualisations, ce qui la rend plus intuitive

Ex. : Graphiques interactifs, visualisations statistiques simples.

Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Visualisation web (interactive et dashboards)



Dash (Plotly)



Permet de créer des applications web interactives. Très populaire pour les dashboards et la visualisation en temps réel des données.

Ex. : Dashboards interactifs, visualisations en temps réel.



Panel (HoloViz)



Librairie basée sur Bokeh pour créer des dashboards interactifs. Elle offre une grande flexibilité pour intégrer des widgets et des contrôles interactifs.

Ex. : Interface pour ajuster des paramètres de visualisation en temps réel.

Présentation de l'écosystème scientifique Python

Les bibliothèques de visualisation

Visualisation de données statistiques



ggplot (ggplot2)



Un port de la fameuse librairie ggplot2 de R. Elle suit une approche déclarative pour créer des visualisations basées sur la grammaire des graphiques.

Ex. : Graphiques statistiques élégants.



Statsmodels



Bien que principalement une librairie pour l'analyse statistique, Statsmodels propose également des visualisations pour analyser des régressions et des séries temporelles

Ex. : Graphiques de résidus, autocorrélation, analyse de séries temporelles.

Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Cartographie et visualisation géospatiale



Geopandas



Extension de Pandas, Geopandas permet de travailler avec des données géospatiales (formes géométriques et géographiques) et de les visualiser facilement.

Ex. : Cartes, cartes choroplèthes, visualisation de zones géographiques.



Folium



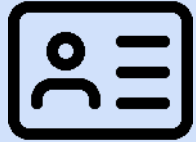
Basée sur la librairie JavaScript Leaflet, Folium permet de créer des cartes interactives dans Python. Idéale pour la visualisation de données géospatiales interactives.

Ex. : Cartes interactives, données géospatiales en temps réel.

Présentation de l'écosystème scientifique Python

Les bibliothèques de visualisation

Cartographie et visualisation géospatiale



Cartopy

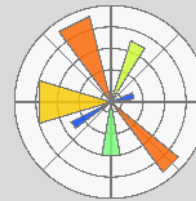


Librairie pour la cartographie géospatiale et la visualisation des données climatiques et environnementales.

Ex. : Cartes des données climatiques, géospatiales.



Basemap



Extension de Matplotlib pour créer des cartes géographiques et des visualisations de données spatiales.

Ex. : Cartes topographiques, cartes des températures, cartes de trajectoires.

Présentation de l'écosystème scientifique Python

Les bibliothèques de visualisation

Big Data et visualisation de données volumineuses



Datashader (HoloViz)



Datashader

Librairie conçue pour la visualisation de très grandes quantités de données. Elle permet de créer des graphiques en temps réel pour des ensembles de données massifs.

Ex. : Visualisation de données massives, comme les données spatiales ou temporelles en temps réel.



HoloViews (HoloViz)



HoloViews

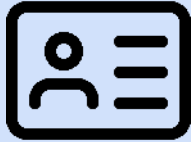
Conçue pour visualiser de très grands ensembles de données (Big Data), HoloViews se base sur des abstractions de haut niveau qui permettent une visualisation rapide.

Ex. : Graphiques interactifs pour données massives, visualisation de séries temporelles.

Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Big Data et visualisation de données volumineuses



Vaex



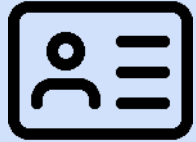
Librairie spécialisée dans la visualisation et l'analyse de très grands ensembles de données. Elle est optimisée pour travailler sur des jeux de données volumineux sans nécessiter beaucoup de mémoire.

Ex. : Visualisation de grands ensembles de données dans des graphiques interactifs.

Présentation de l'écosystème scientifique Python

Les librairies de visualisation

Autres librairies et outils utiles pour la visualisation



Pyvis



Librairie permettant de visualiser des graphes (réseaux) interactifs. Elle est utilisée pour représenter des graphes de manière intuitive et interactive.

Ex. : Visualisation de réseaux sociaux, graphes relationnels.



WordCloud



Spécialement conçue pour créer des nuages de mots (word clouds), une représentation visuelle simple mais percutante des fréquences de mots dans un texte.

Ex. : Analyse de texte, visualisation de mots-clés dans des corpus.

Visualisation

TP



**Représentation de
l'emplacement des
aéroports français et
américains sur une carte**

05

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Panorama des principaux formats de fichiers scientifiques

Ces formats sont optimisés pour stocker, échanger et analyser des données massives ou complexes.

Format	Description	Domaines d'application	Avantages
NetCDF (Network Common Data Form)	Conçu pour la manipulation de données multidimensionnelles (variables, temps, espace).	Climatologie, océanographie, modélisation atmosphérique.	Compact, largement utilisé, compatible avec Xarray.
HDF5 (Hierarchical Data Format v5)	Format hiérarchique conçu pour les données volumineuses et complexes.	Simulations numériques, bioinformatique, machine learning.	Flexible, rapide, compatible avec plusieurs langages.
GRIB (GRIdded Binary)	Format binaire spécialisé pour les données météorologiques.	Prévisions météorologiques, climatologie.	Compact, optimisé pour les grilles spatiales.
MATLAB (.mat)	Format natif pour stocker les données MATLAB.	Recherche scientifique, ingénierie.	Pour le partage entre utilisateurs MATLAB.
CGNS (CFD General Notation System)	Format destiné à la dynamique des fluides numériques.	Simulation en mécanique des fluides.	Standardisé, structuré.
JSON (JavaScript Object Notation)	Format léger pour représenter des objets de données.	APIs, échanges de données, stockage léger.	Lisible par l'humain, très populaire.
PARQUET	Format binaire orienté colonnes pour les données analytiques.	Big Data, machine learning.	Optimisé pour la compression et les requêtes rapides.

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Manipuler des données volumineuses

- ➔ Les bibliothèques modernes permettent de traiter efficacement des datasets volumineux sans charger entièrement les données en mémoire.



Vaex



Les formats de fichiers scientifiques et la manipulation de données volumineuses

Manipuler des données volumineuses



- **Dask** : bibliothèque Python permettant de travailler avec des données volumineuses grâce à une parallélisation simple.
- Utilise une API similaire à Pandas et NumPy, mais traite les données en morceaux pour réduire l'utilisation de la mémoire.
- Compatible avec Pandas, NumPy et scikit-learn.
- Supporte des workflows sur des clusters de calcul.

```
import dask.dataframe as dd

df = dd.read_csv('large_file.csv')
mean_value = df['column'].mean().compute()
print(mean_value)
```


Les formats de fichiers scientifiques et la manipulation de données volumineuses

Manipuler des données volumineuses



- **Vaex** : conçu pour analyser rapidement des datasets massifs (plusieurs téraoctets) sans les charger en mémoire.
- Support pour les fichiers HDF5, CSV, et Parquet.
- Calcul rapide grâce à l'utilisation de techniques de mémoire partagée et différée

```
import vaex

df = vaex.open('large_file.hdf5')
print(df.mean('column'))
```

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Manipuler des données volumineuses



- **Xarray** : Xarray est conçu pour manipuler des données multidimensionnelles comme celles issues des fichiers NetCDF ou HDF5.
- Idéal pour les grilles régulières (données spatiales ou temporelles).*
- Intégration avec Dask pour le traitement distribué.

```
import xarray as xr

dataset = xr.open_dataset('data.nc')
temperature = dataset['temperature'].mean(dim='time')
print(temperature)
```

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Les accès aux bases de données relationnelles, le fonctionnement de la DB API

DB API 2.0 (PEP 249) : norme Python qui définit une interface commune pour interagir avec **toutes** les bases relationnelles (SQLite, PostgreSQL, MySQL, Oracle...).

Objectif : écrire du code Python indépendant du SGBD autant que possible.

Avantages :

- **Uniformité** : même interface pour SQLite, PostgreSQL, MySQL...
- **Sécurité** : placeholders pour éviter les injections SQL
- **Contrôle des transactions** : commit() et rollback()
- **Facilité d'usage** avec curseurs et méthodes fetch

SGBD	Module Python
SQLite	sqlite3
PostgreSQL	psycopg2
MySQL	mysql-connector-python
Oracle	cx_Oracle

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Les accès aux bases de données relationnelles, le fonctionnement de la DB API

Étapes générales :

1. Importer le module Python correspondant au SGBD (SQLite, PostgreSQL, MySQL, etc.).
2. Créer une connexion à la base de données.
3. Créer un curseur à partir de la connexion.
4. Exécuter des commandes SQL via le curseur (SELECT, INSERT, UPDATE, DELETE, etc.).
5. Récupérer les résultats des requêtes si nécessaire (fetchone(), fetchall()).
6. Valider les modifications avec commit() pour les commandes modifiant la base.
7. Gérer les erreurs et éventuellement effectuer un rollback() en cas d'exception.
8. Fermer le curseur et la connexion à la base.

Les formats de fichiers scientifiques et la manipulation de données volumineuses

Les accès aux bases de données relationnelles, le fonctionnement de la DB API

Concept	Description
Connection	Objet représentant la connexion à la base. Permet de créer des curseurs et de gérer les transactions.
Cursor	Objet pour exécuter des requêtes SQL et récupérer les résultats.
Placeholders	Permettent d'éviter l'injection SQL. Syntaxe : ? pour SQLite, %s pour PostgreSQL/MySQL.
commit()	Valide les changements pour INSERT/UPDATE/DELETE.
rollback()	Annule les changements non commités en cas d'erreur.
fetchone() / fetchall()	Méthodes pour récupérer les résultats d'un SELECT.

Les formats de fichiers scientifiques et la manipulation de données volumineuses

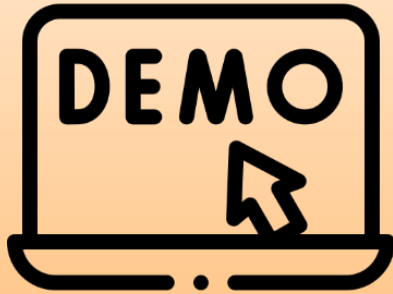
BDD



sqlite3 pour SQLite

Les formats de fichiers scientifiques et la manipulation de données volumineuses

BDD



Psycopg2 pour POSTGRESQL

Conclusion

Projets finaux

Projet final

Tabulaire finance



Apple stock data

Projet final

Machine Learning



**Préparer des données
de consommation
énergétique de
bâtiments pour un futur
modèle prédictif de
machine learning**

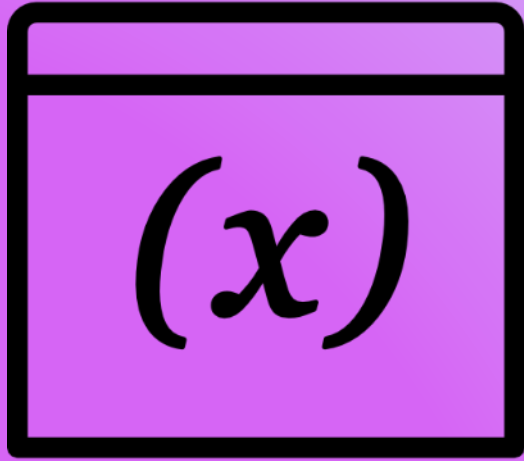


Fin de la formation

Merci !

Annexe

Bases et syntaxe de Python



01

Les variables

Variables, noms et références

- En Python, une variable est un **emplacement dans la mémoire de l'ordinateur où une valeur est stockée**. Pour le programmeur, cette variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une **adresse**, qui fait référence à une zone spécifique de la mémoire.
- En Python, la déclaration d'une variable et son initialisation (c'est-à-dire l'attribution de la première valeur) se font simultanément.

```
>>> x = 2
>>> x
2
```

- À la ligne 1 de cet exemple, nous avons déclaré et initialisé la variable x avec la valeur 2. Notez que plusieurs choses se sont produites en « coulisses » :
 1. ➡ Python a « deviné » que la variable était un entier. Python est connu comme un langage à **typage dynamique**.
 2. ➡ Python a **alloué** (réservé) de l'espace **mémoire** pour accueillir un entier. Chaque type de variable prend plus ou moins de place en mémoire. Python s'est également assuré que la variable pouvait être trouvée sous le nom x.
 3. ➡ Enfin, Python a **assigné** (attribué) la valeur 2 à la variable x.
- Dans d'autres langages (comme le C, par exemple), ces différentes étapes doivent être codées une à une. Python étant un langage de haut niveau, la seule instruction `x = 2` suffit à accomplir les 3 étapes en une seule fois !

Variables, noms et références

- Chaque objet a donc une identité unique, qui correspond à son adresse en mémoire (ou un identifiant équivalent). Cette identité peut être obtenue grâce à la fonction intégrée `id()`.
- Cette fonction retourne un entier unique qui identifie l'objet en mémoire. Dans CPython (l'implémentation standard de Python), cet entier correspond en pratique à l'adresse mémoire de l'objet (même si ce n'est pas garanti par la spécification du langage).

```
x = 42  
print(id(x))
```

- **Valeur** : ce que contient l'objet ([1, 2, 3])
- **Référence** : le "nom" ou la variable qui pointe vers l'objet (a, b, c)
- **Identité** : l'adresse mémoire interne de l'objet (id(a))

```
x = [1, 2, 3]  
y = [1, 2, 3]  
  
print(x == y)  # True  
print(x is y)  # False
```

Types de variables

- Le type d'une variable correspond à sa nature.
- Nous utilisons principalement les types suivants pour représenter des valeurs de base :

<code>int</code>	entier (integer)
<code>float</code>	flottant/nombre réel (floating-point)
<code>str</code>	chaîne de caractères (string)
<code>bool</code>	booléen (boolean)
- Nous aborderons des types plus élaborés plus tard. Pour un aperçu complet des types existants, vous pouvez consulter ce [lien](#).
- La fonction intégrée `type()` permet de déterminer le type d'une variable.

```
>>> x = 5
>>> print(type(x))
<class 'int'>

>>> y = 3.14
>>> print(type(y))
<class 'float'>

>>> z = "Hello, World!"
>>> print(type(z))
<class 'str'>
```



Pour afficher un résultat à l'écran en Python, vous pouvez utiliser la fonction `print()`.

La fonction `print()` prend une ou plusieurs expressions séparées par des virgules, les convertit en chaînes de caractères si nécessaire, puis les affiche à l'écran.

Types de variables

Pour convertir une variable Python en un autre type, vous pouvez utiliser les fonctions de conversion intégrées vers le type cible. Le nom de ces fonctions correspond au nom du type associé, comme dans cet exemple :

```
>>> x = 10
>>> y = float(x)
>>> print(y)
10.0
```

Variables : opérations

Opérations avec les types numériques

- Les quatre opérations arithmétiques de base sont effectuées de manière simple sur les types numériques (entiers et floats) :

```
>>> x = 45
>>> x + 2
47
>>> x - 2
43
>>> x * 3
135
>>> y = 2.5
>>> x - y
42.5
>>> (x * 10) + y
452.5
```

- Notez toutefois que si vous mélangez des types entiers et floats, le résultat est renvoyé sous forme de float (car ce type est plus général). Les parenthèses peuvent également être utilisées pour gérer les priorités.

Variables : opérations

Opération avec les types numériques

- L'opérateur / effectue une division. Contrairement aux opérateurs +, - et *, il renvoie systématiquement un float :

```
>>> 3 / 4
0.75
>>> 4 / 2
2.0
```

- L'opérateur de puissance s'écrit ** :

```
>>> 2 ** 3
8
>>> 2 ** 4
16
```

- Pour obtenir le quotient et le reste d'une division entière, on utilise respectivement les symboles // et % (modulo) :

```
>>> 9 // 4
2
>>> 9 % 4
1
```

Variables : opérations

Opérations avec les types numériques

- Les symboles `+`, `-`, `*`, `/`, `**`, `//` et `%` sont appelés **opérateurs**, simplement parce qu'ils effectuent des opérations sur des variables.
- Il existe des **opérateurs combinés** qui effectuent une opération et une affectation en une seule étape. Par exemple, l'opérateur `+=` effectue une addition puis assigne le résultat à la même variable. Cette opération est appelée *incrément* :

```
>>> i = 0
>>> i = i + 1
>>> i
1
>>> i += 1
>>> i
2
>>> i += 2
>>> i
4
```

- Les opérateurs `-=`, `*=` et `/=` se comportent de manière similaire pour la soustraction, la multiplication et la division.

Variables : opérations

Opérations sur les chaînes de caractères

Pour les chaînes de caractères, deux opérations sont possibles : l'addition et la multiplication.

```
>>> word = "Hi"  
>>> word  
'Hi'  
>>> word + " Python"  
'Hi Python'  
>>> word * 3  
'HiHiHi'
```

- ➡ L'opérateur d'addition + concatène (assemble) deux chaînes de caractères.
- ➡ L'opérateur de multiplication * entre un entier et une chaîne de caractères duplique (répète) une chaîne plusieurs fois.

Variables : opérations

Opérations sur les chaînes de caractères : en coulisses

- En Python, la méthode `__add__` est une méthode spéciale qui permet aux objets de définir comment ils se comportent lorsque l'opérateur `+` est utilisé avec eux. Cette méthode est appelée la méthode "d'addition" ou "de concaténation", selon le contexte dans lequel elle est utilisée.
- Lorsque vous utilisez l'opérateur `+` entre deux objets, Python appelle en interne la méthode `add` de l'opérande de gauche (l'objet à gauche de l'opérateur `+`) et passe l'opérande de droite en tant qu'argument à cette méthode. La méthode `add` effectue alors l'opération nécessaire et renvoie le résultat.
- Par exemple, si vous avez deux objets `a` et `b` et que vous écrivez `a + b`, Python traduit cela en interne par `a.__add__(b)`.

Variables : opérations

Opérations sur les chaînes de caractères

Si vous effectuez une opération illicite, vous recevrez un message d'erreur :

```
>>> "toto" * 1.3
Traceback (most recent call last) :
File "<stdin>", line 1, in <module>
TypeError : can't multiply sequence by non-int of type 'float'
>>> "toto" + 2
Traceback (most recent call last) :
File "<stdin>", line 1, in <module>
TypeError : can only concatenate str (not "int") to str
```



Notez que Python vous donne des informations dans son message d'erreur. Dans le deuxième exemple, il indique que vous devez utiliser une variable de type str, c'est-à-dire une chaîne de caractères, et non un int, c'est-à-dire un entier.

Variables

TP



Nous allons réaliser un premier TP, mais avant cela, spoilons la partie 5...

Listes

Définition

- Une liste est une structure de données contenant une **série de valeurs**.
- Python permet la construction de listes contenant des **valeurs de différents types** (par exemple, entier et chaîne de caractères), ce qui leur confère une grande flexibilité.
- Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets simples ou doubles si des chaînes sont impliquées) séparées par des virgules, et l'ensemble est enfermé dans des **crochets**.
- Voici quelques exemples :

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> dimensions = [5, 2.5, 1.75, 0.15]
>>> mix = ['giraffe', 5, 'mouse ', 0.15]
>>> animals
['giraffe', 'tiger', 'monkey', 'mouse']
>>> dimensions
[5, 2.5, 1.75, 0.15]
>>> mix
['giraffe', 5, 'mouse ', 0.15]
```



Lorsqu'une liste est affichée, Python la rend telle qu'elle a été saisie.

Listes

Utilisation

- L'un des grands avantages d'une liste est que vous pouvez accéder à ses éléments par leur position. Ce nombre est appelé l'index de la liste.

```
list: ['giraffe', 'tiger', 'monkey', 'mouse']  
index: 0 1 2 3
```

- Veuillez noter que les indices d'une liste de n éléments commencent à 0 et se terminent à $n - 1$.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']  
>>> animals[0]  
'giraffe'  
>>> animals[1]  
'tiger'  
>>> animals[3]  
'mouse'
```

- Par conséquent, si nous appelons l'élément d'index 4 de notre liste, Python renverra un message d'erreur :

```
>>> animals[4]  
Traceback (innermost last):  
  File "<stdin>", line 1, in ?  
IndexError : list index out of range
```

Listes

Opérations sur les listes

Comme les chaînes de caractères, les listes prennent en charge l'opérateur + pour la concaténation, ainsi que l'opérateur * pour la duplication :

```
>>> animals_1 = ['giraffe', 'tiger']
>>> animals_2 = ['monkey', 'mouse']
>>> animals_1 + animals_2
['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals_1 * 3
['giraffe', 'tiger', 'giraffe', 'tiger', 'giraffe', 'tiger']
```

Listes

Opérations sur les listes

Vous pouvez également utiliser la méthode `.append()` pour ajouter un élément unique à la fin d'une liste.

```
>>> a = []
>>> a
[]
>>> a = a + [15]
>>> a
[15]
>>> a = a + [-5]
>>> a
[15, -5]
>>> a.append(13)
>>> a
[15, -5, 13]
>>> a.append(-3)
>>> a
[15, -5, 13, -3]
```

Listes

Indexation négative

```
list:           ['giraffe', 'tiger', 'monkey', 'mouse']
positive index:      0         1         2         3
negative index:    -4        -3        -2        -1
```

➡ Il est donc possible d'accéder au dernier élément d'une liste sans utiliser sa longueur.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[-1]
'mouse'
>>> animals[-2]
'monkey'
```

Listes

Slicing

Il est possible de sélectionner une partie d'une liste en utilisant un index construit selon le modèle [m:n] pour récupérer tous les éléments, de l'élément m **inclus** à l'élément n **exclu**. Dans ce cas, nous disons qu'une **tranche** (slice) de la liste est récupérée.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[0:2]
['giraffe', 'tiger']
>>> animals[:3]
['giraffe', 'tiger', 'monkey']
>>> animals[:]
['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[1:]
['tiger', 'monkey', 'mouse']
>>> animals[1:-1]
['tiger', 'monkey']
```

Listes

Slicing

Vous pouvez également spécifier la taille du **pas** (step) en ajoutant un symbole ":" supplémentaire et en indiquant la taille du pas avec un entier.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> animals[:3:2]
['giraffe', 'monkey']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
>>> x[6:2:-3]
[6, 3]
```

Listes

Slicing, en résumé

L'accès au contenu d'une liste est basé sur le modèle

`list[start:end:step]`

où start est inclus et end est exclu.

Listes

La fonction len()

L'instruction `len()` est utilisée pour connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que la liste contient.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> len(animals)
4
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
8
```

Listes

Les fonctions list() et range()

```
>>> iterable = (1, 2, 3, 4, 5)
>>> new_list = list(iterable)
>>> print(new_list)
[1, 2, 3, 4, 5]
```

En Python, l'instruction **list()** est utilisée pour créer un nouvel objet liste (à partir d'un Iterable, que nous verrons plus tard).

L'instruction **range()** est une fonction spéciale de Python qui génère des entiers dans une plage donnée. Lorsqu'elle est utilisée en combinaison avec la fonction **list()**, elle produit une liste d'entiers.

L'instruction **range()** fonctionne selon le modèle **range([start,] end[, step])**. Les arguments entre crochets sont optionnels. Si un seul argument est fourni, il correspond à la fin, auquel cas le début est 0 et le pas est 1. Si deux arguments sont fournis, ils correspondent au début et à la fin, et le pas est 1.

```
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
>>> list(range(15, 20))
[15, 16, 17, 18, 19]
>>> list(range(0, 1000, 200))
[0, 200, 400, 600, 800]
>>> list(range(2, -2, -1))
[2, 1, 0, -1]
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Listes

Un exemple de liste de listes

```
>>> exhibit_1 = ['giraffe', 4]
>>> exhibit_2 = ['tiger', 2]
>>> exhibit_3 = ['monkey', 5]
>>> zoo = [exhibit_1, exhibit_2, exhibit_3]
>>> zoo
[['giraffe', 4], ['tiger', 2], ['monkey', 5]]
```

```
>>> zoo[1]
['tiger', 2]
>>> zoo[1][0]
'tiger'
>>> zoo[1][1]
2
```

Tuples

Comparaison avec les listes

Les tuples sont similaires aux listes, avec quelques différences fondamentales. Ils se déclarent avec des parenthèses au lieu de crochets, et sont immutables.

Caractéristique	Tuple	Liste
Mutabilité	Immutable (ne peut pas être modifié après création)	Mutable (peut être modifié après création)
Syntaxe	Parenthèses : (1, 2, 3)	Crochets : [1, 2, 3]
Performance	Plus rapide en termes d'accès et de traitement	Légèrement plus lent en raison de la mutabilité
Taille	Taille fixe une fois créé	Taille variable (ajout et suppression d'éléments possibles)
Utilisation courante	Lorsque l'intégrité des données est importante (par ex. coordonnées GPS d'un point de référence)	Lorsque la manipulation des données est nécessaire (par ex. liste d'achats)
Fonctions disponibles	Moins de méthodes disponibles (ex. : pas d'append(), remove())	Plus de méthodes disponibles (append(), remove(), etc.)

Tuples

Comparaison avec les listes

En particulier, la syntaxe suivante renverra une erreur :

```
my_tuple = (1, 2, 3)
my_tuple[0] = 4 # lèvera une erreur
>>> TypeError: 'tuple' object does not support item assignment
```

Complément sur les chaînes de caractères

Les chaînes de caractères en tant que séquences de caractères

Les chaînes de caractères peuvent être considérées comme des "listes" (de caractères) d'un genre particulier :

```
>>> animal = "big tiger"
>>> animal[:5]
'big t'
>>> animal[6:]
'ger'
>>> animal[ :-2]
'big tig'
>>> animal[1:-2:2]
'i i'
```

Complément sur les chaînes de caractères

Les chaînes de caractères en tant que séquences de caractères

Cependant, contrairement aux listes, les chaînes de caractères présentent une différence notable : elles ne peuvent pas être modifiées. Une fois qu'une chaîne a été définie, vous ne pouvez plus modifier aucun de ses éléments.

```
>>> animal = "big tiger"
>>> animal[5]
't'
>>> animal[5] = "T"
Traceback (most recent call last) :
  File "<stdin>", line 1, in <module>
TypeError : 'str' object does not support item assignment
```

- ➡ Pour cette raison, nous pourrions plutôt les désigner comme des "séquences" de caractères, un terme plus large que "listes" qui n'implique pas certaines propriétés, notamment la mutabilité.
- ➡ Si vous souhaitez modifier une chaîne, vous devrez en construire une nouvelle. Pour ce faire, n'oubliez pas que les opérateurs de concaténation (+) et de duplication (*) peuvent être utiles. Vous pouvez également générer une liste, qui peut être modifiée, puis revenir à une chaîne.

Complément sur les listes

Autres méthodes des listes : la méthode `.insert()`

La méthode `.insert()` insère un objet dans une liste à une position (index) donnée :

```
>>> a = [1, 2, 3]
>>> a.insert(2, -15)
>>> a
[1, 2, -15, 3]
```


Complément sur les listes

Autres méthodes des listes : del

L'instruction `del` supprime un élément d'une liste à un index donné. Contrairement aux autres méthodes associées aux listes, `del` est une instruction générale de Python, qui peut être utilisée pour des objets autres que les listes. Elle ne nécessite pas de parenthèses.

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> a
[1, 3]
```

Complément sur les listes

Autres méthodes des listes : la méthode `.remove()`

La méthode `.remove()` supprime un élément d'une liste en fonction de sa valeur. Plus précisément, elle supprime la première occurrence de l'élément spécifié.

```
>>> a = [1, 2, 3]
>>> a.remove(3)
>>> a
[1, 2]
```

```
>>> b = [1, 2, 3, 2]
>>> b.remove(2)
>>> b
[1, 3, 2]
>>> b.remove(2)
>>> b
[1, 3]
```

Complément sur les listes

Autres méthodes des listes : les méthodes `.sort()` et `reverse()`

La méthode `.sort()` trie une liste, par défaut, du plus petit au plus grand.

```
>>> a = [4, 1, 2]
>>> a.sort()
>>> a
[1, 2, 4]
```

La méthode `.reverse()` inverse une liste.

```
>>> a = [4, 1, 2]
>>> a.reverse()
>>> a
[2, 1, 4]
```

Notez que la séquence `.sort()` puis `.reverse()` trie du plus grand au plus petit. Par souci de simplicité, `reverse` peut également être passé comme argument à `.sort()` pour trier du plus grand au plus petit.

```
>>> a = [4, 1, 2]
>>> a.sort(reverse=True)
>>> a
[4, 2, 1]
```

Complément sur les listes

Tests d'appartenance

L'opérateur **in** teste si un élément fait partie d'une liste.

```
my_list = [1, 3, 5, 7, 9]
>>> 3 in my_list
True
>>> 4 in my_list
False
```

La variation avec **not** permet, à contrario, de vérifier qu'un élément n'est pas dans une liste.

```
>>> 3 not in my_list
False
>>> 4 not in my_list
True
```

Si vous vous souvenez des diapositives sur les booléens, il serait également possible d'inverser le résultat booléen ainsi obtenu.

```
>>> not (3 in my_list)
False
>>> not 4 not in my_list
True
```

Complément sur les listes

Mutabilité des listes et copies

Objets mutables

Lorsque vous avez un objet mutable, comme une liste, un dictionnaire ou une instance de classe personnalisée, vous pouvez modifier directement son contenu. Cela signifie que si vous avez deux variables qui font référence au même objet mutable, les modifications apportées via une variable seront reflétées dans l'autre variable, puisqu'elles pointent toutes deux vers le même objet en mémoire.

Copie par référence

Lorsque vous effectuez une copie par référence, toutes les modifications ultérieures apportées à l'objet original seront visibles à travers la référence copiée, et vice versa. L'opposé serait une copie par valeur.

➡ Lorsque vous effectuez une opération de copie sur un objet mutable en Python (comme les listes) en utilisant la méthode `=`, vous créez une nouvelle référence vers le même objet.



```
>>> x = [1, 2, 3]
>>> y = x
>>> y
[1, 2, 3]
>>> x[1] = -15
>>> x
[1, -15, 3]
>>> y
[1, -15, 3]
```

Complément sur les listes

Mutabilité des listes et copies

- L'utilisation de la méthode `.copy()` sur une liste en Python crée une copie superficielle (shallow) de cette liste, ce qui signifie qu'elle copie les éléments de la liste dans un nouvel objet liste. Par conséquent, les modifications apportées à la liste originale n'affecteront pas la liste copiée, et vice versa.

- Cependant, une copie superficielle signifie que la structure de premier niveau de la liste est dupliquée, mais les éléments eux-mêmes ne sont pas copiés de manière récursive. En d'autres termes, si la liste contient des références à d'autres objets mutables (comme d'autres listes ou dictionnaires), ces références sont copiées, mais les objets eux-mêmes ne sont pas dupliqués.

```
>>> x = [1, 2, 3]
>>> y = x.copy()
>>> x[1] = -15
>>> x
[1, -15, 3]
>>> y
[1, 2, 3]
```

```
>>> x = [1, 2, [7, 5]]
>>> y = x.copy()
>>> x[1].append(3)
>>> x
[1, 2, [7, 5, 3]]
>>> y
[1, 2, [7, 5, 3]]
```

Complément sur les listes

Mutabilité des listes et copies

Pour effectuer une copie profonde d'une liste en Python, vous pouvez utiliser la fonction **copy.deepcopy()** du module `copy`. Cette fonction crée un nouvel objet et copie récursivement l'objet original ainsi que tous ses objets imbriqués. Cela garantit que les modifications apportées à l'objet original n'affectent pas l'objet copié, même si l'objet original contient des objets mutables imbriqués.

```
>>> import copy
>>> x = [[1, 2], [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> y = copy.deepcopy(x)
>>> x[1][1] = 99
>>> x
[[1, 2], [3, 99]]
>>> y
[[1, 2], [3, 4]]
```

Collections

Dictionnaires

- Utilisés pour stocker des collections de données sous forme de **paires clé-valeur**.
- Chaque élément dans un dictionnaire est accessible par sa clé, plutôt que par son index comme dans les listes ou les tuples.
- Les dictionnaires sont **mutables**, ce qui signifie qu'ils peuvent être modifiés après leur création.

```
>>> animal_1 = {}
>>> animal_1["name"] = "giraffe"
>>> animal_1["height"] = 5.0
>>> animal_1["weight"] = 1100
>>> animal_1
{'name' : 'giraffe', 'height' : 5.0, 'weight' : 1100}
```

```
>>> animal_1["height"]
5.0
>>> animal_1["sex"]
KeyError: 'sex'
>>> animal_1.get("height")
5.0
>>> animal_1.get("sex")
None
```

```
>>> animal_2 = {"name": "monkey", "weight": 70, "height": 1.75}
>>> animal_2["age"] = 15
```


Collections

Dictionnaires : `.keys()`, `.values()`

```
>>> animal_2.keys()
dict_keys(['weight', 'name', 'height'])
>>> animal_2.values()
dict_values([70, 'monkey', 1.75])
```

Les mentions **dict_keys** et **dict_values** indiquent que nous avons affaire à des objets quelque peu spéciaux.

Ils ne sont pas indexables (nous ne pouvons pas récupérer un élément par index, par ex. `some_dict.keys()[0]` renverra une erreur).

Si nécessaire, nous pouvons les convertir en liste en utilisant la fonction `list()`. Cependant, ce sont des objets "itérables", donc ils peuvent être directement utilisés dans une boucle (nous y reviendrons dans la section suivante dédiée aux boucles).

```
>>> animal_2.keys()[0]
... TypeError: 'dict_keys' object is not subscriptable
```

```
>>> for val in animal_2.values():
...     print(val)
...
70
'monkey'
1.75
```

Collections

Dictionnaires : .items()

La méthode `.items()` renvoie un objet de vue qui affiche une liste des paires clé-valeur d'un dictionnaire. Ces paires sont présentées sous forme de tuples, où chaque tuple contient une clé et sa valeur correspondante. Cette méthode permet d'accéder et d'itérer à la fois sur les **clés** et les **valeurs** du dictionnaire **simultanément**.

```
>>> a = {0 : 't', 1 : 'o', 2 : 't', 3 : 'o'}  
>>> a.items()  
dict_items([(0, 't'), (1, 'o'), (2, 't'), (3, 'o')])
```

Collections

Dictionnaires : listes de dictionnaires

- Lorsque nous créons une liste de dictionnaires avec des clés partagées, nous construisons effectivement une structure semblable à une **base de données**. Contrairement aux listes, les dictionnaires offrent un moyen plus explicite de gérer des structures complexes. Chaque entrée de dictionnaire représente un **enregistrement**, où **les clés servent de noms de champs et les valeurs correspondantes contiennent les données associées à chaque champ**.
- Cette approche fournit clarté et organisation, facilitant l'accès et la manipulation des données au sein de la structure. Les dictionnaires offrent une flexibilité dans la représentation des entités du monde réel, permettant une gestion des données plus intuitive dans les programmes Python.

```
>>> animals = [animal_1, animal_2]
>>> animals
[{'name': 'giraffe', 'weight': 1100, 'height': 5.0}, {'name': 'monkey',
'weight': 70, 'height': 1.75}]
>>>
>>> for animal in animals :
...     print(animal['name'])
...
giraffe
monkey
```

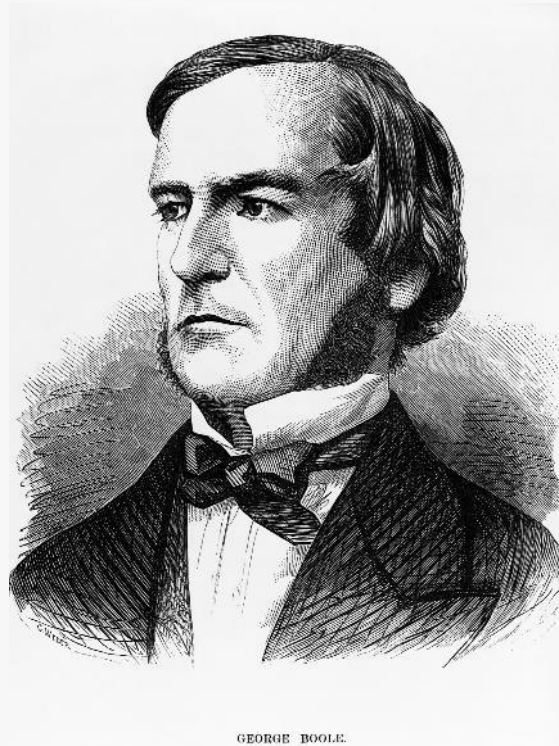
02

Opérateurs et expressions

Booléens

Le type de données « booléen »

- Les booléens sont un type de données qui représente l'une des deux valeurs possibles : vrai ou faux.
- Les booléens portent le nom du mathématicien George Boole, qui a d'abord formulé l'algèbre booléenne, une branche de l'algèbre où les variables sont soit vraies, soit fausses.
- En Python, le type de données booléen est désigné par les mots-clés **True** et **False**. Ces mots-clés sont sensibles à la casse.



Booléens

Utilisation

- ➡ Les booléens peuvent être combinés à l'aide d'**opérateurs logiques** tels que and, or et not pour effectuer des opérations logiques.
- ➡ Les booléens sont souvent utilisés dans des expressions impliquant des **opérateurs de comparaison** tels que == (égal à), != (différent de), < (inférieur à), > (supérieur à), <= (inférieur ou égal à) et >= (supérieur ou égal à).
- ➡ Les booléens sont couramment utilisés dans des **instructions conditionnelles** telles que if, elif et else pour contrôler le flux d'exécution du programme en fonction de certaines conditions.

Booléens

Comparaisons

Syntaxe Python	Signification
<code>==</code>	est égal à
<code>!=</code>	est différent de
<code>></code>	est strictement supérieur à
<code>>=</code>	est supérieur ou égal à
<code><</code>	est strictement inférieur à
<code><=</code>	est inférieur ou égal à

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
```

```
>>> animal = "tiger"
>>> animal == "tig"
False
>>> animal != "tig"
True
>>> animal == "tiger"
True
```

```
>>> "a" < "b"
True
>>> "ali" < "alo"
True
>>> "abb" < "ada"
True
```

Booléens

Expressions conditionnelles

Les tests conditionnels sont une partie essentielle de tout langage informatique, si l'on souhaite lui donner une certaine complexité, car ils permettent à l'ordinateur de prendre des décisions. Pour cela, Python utilise l'instruction **if** ainsi qu'une comparaison.

```
>>> x = 2
>>> if x == 2:
...     print("The test is true!")
...
The test is true!
```

```
>>> x = "mouse"
>>> if x == "tiger":
...     print("The test is true!")
...
```

- ➡ Dans le premier exemple, comme le test est vrai, l'instruction `print("Le test est vrai !")` est exécutée.
- ➡ Dans le second exemple, le test est faux et rien n'est affiché.
- ➡ Les blocs d'instructions dans les tests doivent être indentés. L'indentation indique la portée des instructions à exécuter si le test est vrai.
- ➡ La ligne contenant l'instruction `if` se termine par le caractère deux-points ":".

Booléens

Expressions conditionnelles : else

Parfois, il est utile de tester si une condition est vraie ou fausse dans la même instruction if :

```
>>> if x == 2:
...     print("The test is true!")
... else:
...     print("Test is false!")
...
The test is true!
>>> x = 3
>>> if x == 2:
...     print("Test is true!")
... else :
...     print("Test is false!")
```

```
>>> import random
>>> base = random.choice(["a", "t", "c", "g"])
>>> if base == "a":
...     print("Choice: adenine")
... elif base == "t":
...     print("Choice: thymine")
... elif base == "c":
...     print("Choice: cytosine")
... elif base == "g":
...     print("Choice: guanine")
...
Choice: cytosine
```

Booléens

Tests multiples

Condition 1	Opérateur	Condition 2	Résultat
True	OR	True	True
True	OR	False	True
False	OR	True	True
False	OR	False	False

Condition 1	Opérateur	Condition 2	Résultat
True	AND	True	True
True	AND	False	False
False	AND	True	False
False	AND	False	False

Booléens

Tests multiples

En Python, nous utilisons le mot réservé **and** pour l'opérateur AND et le mot réservé **or** pour l'opérateur OR. Veillez à la casse des opérateurs `and` et `or`, qui sont écrits en minuscules dans Python.

```
>>> x = 2
>>> y = 2
>>> if x == 2 and y == 2:
...     print("x and y are both 2")
...
x and y are both 2
```

Le même résultat serait obtenu en utilisant deux instructions `if` imbriquées :

```
>>> x = 2
>>> y = 2
>>> if x == 2:
...     if y == 2:
...         print("x and y are both 2")
...
x and y are both 2
```

Booléens

Tests multiples

Vous pouvez également tester directement l'effet de ces opérateurs en utilisant True et False (sensibles à la casse).

```
>>> True or False
True
```

Enfin, vous pouvez utiliser l'opérateur de négation logique **not**, qui inverse le résultat d'une condition :

```
>>> not True
False
>>> not False
True
>>> not (True and True)
False
```

Booléens

Test de valeur des floats dans l'analyse de données (1/3)

Lorsque vous souhaitez tester la valeur d'une variable flottante, le réflexe initial serait d'utiliser l'opérateur d'égalité, comme :

```
>>> 1/10 == 0.1  
True
```

Cependant, il n'est vraiment pas conseillé de le faire. Python stocke les valeurs numériques des floats sous forme de nombres à virgule flottante (d'où le nom !), ce qui entraîne certaines limitations.

```
>>> (3 - 2.7) == 0.3  
False  
>>> 3 - 2.7  
0.2999999999999998
```

Booléens

Test de valeur des floats dans l'analyse de données (2/3)

- En fait, ce problème ne vient pas de Python, mais plutôt de la façon dont un ordinateur gère les nombres à virgule flottante (comme un rapport de nombres binaires).
- Cela signifie que certaines valeurs flottantes ne peuvent être que des approximations.
- Une façon de s'en rendre compte est d'utiliser une écriture formatée en demandant l'affichage d'un grand nombre de décimales :

```
>>> 0.3
0.3
>>> "{:.5f}".format(0.3)
'0.30000'
>>> "{:.60f}".format(0.3)
'0.299999999999999988897769753748434595763683319091796875000000'
>>> "{:.60f}".format(3.0 - 2.7)
'0.299999999999999982236431605997495353221893310546875000000000'
```

Booléens

Test de valeur des floats dans l'analyse de données (3/3)

Pour ces raisons, vous ne devriez pas tester si un float est égal à une certaine valeur. La meilleure pratique consiste à vérifier si un flottant se situe dans un intervalle d'une certaine précision.

```
>>> delta = 0.0001
>>> var = 3.0 - 2.7
>>> 0.3 - delta < var < 0.3 + delta
True
>>> abs(var - 0.3) < delta
True
```



03

Les structures de contrôle

Boucles FOR

Principe

- En programmation, vous devez souvent répéter une instruction plusieurs fois.
- Les boucles sont une partie essentielle de tout langage de programmation et nous aident à le faire de manière compacte et efficace.
- Imaginez, par exemple, que vous souhaitez afficher les éléments d'une liste les uns après les autres. Avec ce que nous avons vu jusqu'à présent dans cette formation, vous devrez taper quelque chose comme :

```
animals = ['giraffe', 'tiger', 'monkey', 'mouse']  
print(animals[0])  
print(animals[1])  
print(animals[2])  
print(animals[3])
```

- ➡ Si votre liste ne contient que 4 éléments, cela reste faisable, mais imaginez si elle en contenait 1000...
- ➡ Pour remédier à cela, on utilise les boucles.

Boucles FOR

Principe

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...     print(animal)
...
giraffe
tiger
monkey
mouse
```

- La variable `animal` est appelée la **variable d'itération** et prend les différentes valeurs de la liste `animals` à chaque itération de la boucle.
- Vous pouvez choisir le nom que vous souhaitez pour cette variable. Elle est créée par Python la première fois que la ligne contenant celle-ci est exécutée (si elle existait déjà, son contenu serait écrasé). Une fois la boucle terminée, cette variable d'itération `animal` ne sera pas détruite et contiendra donc la dernière valeur de la liste `animals` (dans ce cas, la chaîne de caractères `mouse`).

Boucles FOR

Principe

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...     print(animal)
...
giraffe
tiger
monkey
mouse
```

- Notez les types de variables utilisés ici : `animals` est une liste sur laquelle nous itérons, et `animal` est une chaîne de caractères, car chaque élément de la liste est une chaîne de caractères.
- La variable d'itération peut être de n'importe quel type, en fonction de la liste sur laquelle on itère. En Python, une boucle itère toujours sur un objet dit séquentiel (c'est-à-dire un objet composé d'autres objets) comme une liste.
- Il est possible d'itérer sur d'autres objets séquentiels avec une boucle.

Boucles FOR

Principe

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...     print(animal)
...
giraffe
tiger
monkey
mouse
```

- Notez également le ":" à la fin de la ligne commençant par for. Cela signifie que la boucle for attend un bloc d'instructions, dans ce cas, toutes les instructions que Python répétera à chaque itération de la boucle. Ce bloc d'instructions est appelé le corps de la boucle.

- Comment indiquons-nous à Python où ce bloc commence et se termine ? Cela est indiqué uniquement par l'indentation, c'est-à-dire en décalant la ou les lignes du bloc d'instructions vers la droite.

Itérables

Quelques définitions

List

Une liste est une structure de données intégrée de Python qui contient une collection ordonnée d'éléments.

Vous pouvez itérer sur une liste en utilisant une boucle for pour accéder à chaque élément de manière séquentielle.

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
```

Sequence

En Python, une séquence est un terme générique désignant tout objet itérable qui prend en charge l'accès indexé, comme les chaînes de caractères, les listes, les tuples et les plages (ranges). En conséquence, elle a une longueur connue.

Certaines propriétés, comme la mutabilité, ne sont pas fixes.

Vous pouvez itérer sur les séquences en utilisant une boucle for pour accéder à chaque élément.

```
my_string = "Hello,
world!"
for char in my_string:
    print(char)
```

Iterable

Un itérable est tout objet en Python qui peut être itéré, ce qui signifie qu'il peut produire séquentiellement un élément à la fois. En plus des séquences, d'autres exemples d'itérables incluent les dictionnaires, les ensembles, les objets de fichier et les objets générateurs.

```
my_dict = {"a": 1, "b": 2,
"c": 3}
for key in my_dict:
    print(key,
my_dict[key])
```

Itérables

Vue d'ensemble

Une boucle en Python peut itérer sur diverses structures de données, toutes très utiles dans l'analyse de données :

Lists

Une boucle peut itérer sur les **éléments** d'une liste.

Tuples

De même, une boucle peut itérer sur les **éléments** d'un tuple.

Strings

Une boucle peut itérer sur les **caractères** d'une chaîne de caractères.

Ranges

Une boucle peut itérer sur les **éléments** d'un objet range.

Dictionaries

Une boucle peut itérer sur les **clefs**, les **valeurs** ou les **paires clef-valeur** d'un dictionnaire.

Sets

Une boucle peut itérer sur les **éléments** d'un ensemble.

File objects

Une boucle peut itérer sur les **lignes** d'un objet fichier lors de la lecture d'un fichier.

En réalité, une boucle en Python peut itérer sur tout itérable, c'est-à-dire tout objet qui prend en charge le protocole d'itération : si un objet implémente la méthode `__iter__()` pour renvoyer un itérateur, une boucle peut itérer sur les éléments produits par cet itérateur.

Boucles WHILE

Description

Une alternative à l'instruction `for` couramment utilisée en informatique est la boucle `while`. Une série d'instructions est exécutée tant qu'une condition est vraie.

```
>>> i = 1
>>> while i <= 4:
...     print(i)
...     i += 1
...
1
2
3
4
```

Une boucle `while` nécessite généralement trois éléments pour fonctionner correctement :

1. ➡ L'initialisation de la variable d'itération avant la boucle (ligne 1).
2. ➡ Le test de la variable d'itération associé à l'instruction `while` (ligne 2).
3. ➡ La mise à jour de la variable d'itération dans le corps de la boucle (ligne 4).

Boucles WHILE

Exemple

Une boucle `while` combinée avec la fonction `input()` peut être très utile lorsque vous souhaitez demander à l'utilisateur une valeur numérique.

```
>>> i = 0
>>> while i < 10:
...     response = input("Enter an integer greater than 10:")
...     i = int(response)
...
Enter an integer greater than 10: 4
Enter an integer greater than 10: -3
Enter an integer greater than à 10: 15
>>> i
15
```

La fonction `input()` prend comme argument un message (sous forme de chaîne de caractères), demande à l'utilisateur de saisir une valeur et la renvoie sous forme de chaîne de caractères. Cette valeur doit ensuite être convertie en entier (en utilisant la fonction `int()`).

Boucles

De l'importance de l'indentation

Code 1 :

```
numbers = [4, 5, 6]
for nb in numbers:
    if nb == 5:
        print("The test is true")
        print(f"because nb is {nb}")
```

Résultat 1 :

```
The test is true
because nb is 5
```

Code 2 :

```
numbers = [4, 5, 6]
for nb in numbers:
    if nb == 5:
        print("The test is true")
    print(f"because nb is {nb}")
```

Résultat 2 :

```
because nb is 4
The test is true
because nb is 5
because nb is 6
```

Boucles

De l'importance de l'indentation

Une erreur entraîne souvent des "boucles infinies" (qui ne s'arrêtent jamais).

Vous pouvez toujours arrêter l'exécution d'un script Python avec la combinaison de touches Ctrl-C (c'est-à-dire en appuyant simultanément sur les touches Ctrl et C).

```
i = 0
while i < 10:
    print("Python is cool!")
```

Ici, nous avons omis de mettre à jour la variable `i` dans le corps de la boucle. En conséquence, la boucle ne s'arrêtera jamais (sauf en appuyant sur Ctrl-C), puisque la condition `i < 10` sera toujours vraie.

Boucles

Instructions BREAK et CONTINUE

Ces deux instructions peuvent être utilisées pour modifier le comportement d'une boucle (for ou while) avec un test.

➡ L'instruction **break** « sort de » (arrête) la boucle.

```
>>> for i in range(5):  
...     if (i > 2) and (i < 4):  
...         break  
...     print(i)  
...  
0  
1  
2
```

➡ L'instruction **continue** passe à l'itération suivante, sans exécuter le reste du bloc d'instructions de la boucle.

```
>>> for i in range(5):  
...     if (i > 2) and (i < 4):  
...         continue  
...     print(i)  
...  
0  
1  
2  
4
```

Boucles : retour aux listes

Itération sur les listes : itération sur les indices

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for i in range(4):
...     print(animals[i])
...
giraffe
tiger
monkey
mouse
```

Boucles : retour aux listes

Itération sur les listes : itération directe

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for animal in animals:
...     print(animal)
...
giraffe
tiger
monkey
mouse
```

Boucles : retour aux listes

Itération sur les listes : enumerate

- La méthode la plus efficace est la seconde, qui itère directement sur les éléments.
- Cependant, il peut y avoir des cas pendant une boucle où on a besoin des indices. Dans ce cas, il faudra itérer sur les indices.
- Python fournit la fonction **enumerate()**, qui permet d'itérer à la fois sur les indices et sur les éléments eux-mêmes. Cela offre un moyen pratique d'accéder aux éléments et à leurs indices correspondants dans une boucle.

```
>>> animals = ['giraffe', 'tiger', 'monkey', 'mouse']
>>> for i, animal in enumerate(animals):
...     print("Animal {} is a(n) {}".format(i, animal))
...
Animal 0 is a(n) giraffe
Animal 1 is a(n) tiger
Animal 2 is a(n) monkey
Animal 3 is a(n) mouse
```

Boucles : retour aux listes

Itération sur les listes : list comprehension

Les compréhensions de liste en Python offrent plusieurs avantages, rendant le code plus concis et lisible (une fois qu'on y est habitué) d'une part, et plus efficace d'autre part.

```
>>> x = []
>>> for i in range(21):
...     if i % 2 == 0:
...         x.append(i)
...
>>> print(x)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```



```
>>> x = [i for i in range(21) if i % 2 == 0]
>>> print(x)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

$f(x)$

04

Les procédures et les fonctions

Fonctions

Principes et généralités

- En programmation, les fonctions sont très utiles pour **effectuer plusieurs fois la même opération** au sein d'un programme. Elles rendent également le code **plus clair** et **plus lisible** en le divisant en blocs logiques.
- Lors de son utilisation, une fonction peut être vue comme une sorte de "boîte noire" :
 1. ➡ À laquelle vous transmettez aucune, une ou plusieurs variable(s) entre parenthèses. Ces variables sont appelées des arguments. Elles peuvent être de n'importe quel type d'objet Python.
 2. ➡ Qui exécute une action.
 3. ➡ Et qui renvoie un objet Python ou rien du tout.

Fonctions

Exemples

```
>>> len([0, 1, 2])  
3
```

1. ➡ On appelle la fonction `len()`, en lui passant une liste en argument (dans ce cas, la liste `[0, 1, 2]`).
2. ➡ La fonction calcule la longueur de cette liste (dans ce cas, 3).
3. ➡ Elle renvoie un entier égal à cette longueur.

Fonctions

Exemples

```
>>> some_list.append(5)
```

1. ➡ Vous passez l'entier 5 en argument.
2. ➡ La méthode `.append()` ajoute l'entier 5 à l'objet `some_list`.
3. ➡ Elle ne renvoie rien.

Fonctions

Modularité

- Une fonction accomplit une tâche. Pour cela, elle peut recevoir des arguments et renvoyer un résultat. L'algorithme utilisé à l'intérieur de la fonction n'a pas d'intérêt direct pour l'utilisateur.
- ➡ Par exemple, il n'est pas nécessaire de savoir comment la fonction `math.cos()` calcule un cosinus. Tout ce que vous devez savoir, c'est qu'il faut lui fournir un angle en radians comme argument et qu'elle renverra le cosinus de cet angle. Ce qui se passe à l'intérieur de la fonction est uniquement l'affaire du programmeur.

En général, chaque fonction réalise une **tâche unique et précise**. Si les choses deviennent complexes, il est recommandé d'écrire plusieurs fonctions (qui peuvent également s'appeler entre elles). Cette **modularité** améliore la qualité globale du code ainsi que sa lisibilité.

Fonctions

Définition d'une fonction

- Pour définir une fonction, Python utilise le mot-clé **def**.
- Si vous souhaitez que la fonction renvoie quelque chose, utilisez le mot-clé **return**. Lorsqu'une instruction **return** est rencontrée, l'exécution de l'appel de la fonction s'arrête immédiatement.

```
>>> def square(x):  
...     return x ** 2  
...  
>>> print(square(3))  
9
```

```
>>> res = square(3)  
>>> print(res)  
9
```

```
>>> def hello():  
...     print("hi")  
...  
>>> hello()  
hi
```

```
>>> var = hello()  
hi  
>>> print(var)  
None
```

Fonctions

Renvoyer plusieurs résultats

- Les fonctions peuvent renvoyer plusieurs objets à la fois.
- En réalité, Python ne renvoie qu'un seul objet, mais cet objet peut être séquentiel, c'est-à-dire qu'il peut contenir d'autres objets. Dans notre exemple, Python renvoie un objet de type tuple. Nous aurions tout aussi bien pu écrire notre fonction pour qu'elle renvoie une liste.
- Le retour d'un tuple ou d'une liste contenant deux éléments ou plus est compatible avec l'affectation multiple, ce qui permet de récupérer plusieurs valeurs renvoyées par une fonction et de les assigner à différentes variables en une seule opération.

```
>>> def square_cube(x):  
...     return x**2, x**3  
...  
>>> square_cube(3)  
(9, 27)
```

```
>>> def square_cube(x):  
...     return [x**2, x**3]  
...  
>>> square_cube(3)  
[9, 27]
```

```
>>> z1, z2 = square_cube(3)  
>>> z1  
9  
>>> z2  
27
```

Fonctions

Définition d'une fonction avec des valeurs d'argument par défaut

```
>>> def sum(a, b=1):  
...     return a + b  
...  
>>> sum(3)  
4  
>>> sum(3, 3)  
6
```

Fonctions

Appels de fonction positionnels ou par mot-clé

```
>>> def fct(a, b):  
...     return a - b  
...  
>>> fct(3, 2)  
1  
>>> fct(a=7, b=2)  
5  
>>> fct(b=12, a=14)  
2
```


Fonctions

Appeler une fonction dans une autre fonction

- Lors de la manipulation des fonctions, il est essentiel de comprendre le comportement des variables.
- Une variable est dite **locale** lorsqu'elle est créée à l'intérieur d'une fonction. Elle n'existe et n'est visible que pendant l'exécution de cette fonction.
- Une variable est dite **globale** lorsqu'elle est créée dans le programme principal. Elle est visible partout dans le programme.

```
def square(x):  
    y = x ** 2  
    return y  
  
z = 5  
result = square(z)  
print(result)
```



[Python Tutor](https://www.python-tutor.com/)

Fonctions

Règle LGI

- Lorsque Python rencontre une variable, il traitera la résolution de son nom avec des priorités particulières :
- 1. Tout d'abord, il vérifiera si la variable est locale.
- 2. Ensuite, s'il ne trouve pas la variable localement, il vérifiera si elle est globale.
- 3. Enfin, si elle n'est pas globale, il vérifiera si elle est interne (par exemple, la fonction len() est considérée comme une fonction interne à Python, elle existe à chaque fois que vous démarrez Python).

LGI = Local > Global > Internal

```
>>> def my_function():  
...     x = 4  
...     print(f'Function: x = {x}')
```



```
>>> x = -15  
>>> my_function()  
Function: x = 4  
>>> print('Module: x = {x}')
```

```
Module: x = -15
```

Fonctions

Expressions lambda

- Une fonction lambda est une fonction **anonyme**, c'est-à-dire une fonction qui n'a pas besoin d'être définie avec le mot-clé `def` et n'a pas de nom propre. Elle est souvent utilisée pour des opérations simples, en particulier lorsqu'il est inutile de définir une fonction complète pour une tâche qui ne sera utilisée qu'une seule fois.
- La syntaxe est la suivante :

```
lambda arguments: expression
```

Exemple d'utilisation :

```
items = [(1, 'banane'), (2, 'pomme'), (3, 'poire')]  
sorted_items = sorted(items, key=lambda item: item[1])  
print(sorted_items)
```

Fonctions

Expressions lambda

Intérêt :

- **Concision** : Elles permettent d'écrire des fonctions courtes et simples de manière compacte, sans avoir à utiliser plusieurs lignes pour une définition complète.
- **Usage temporaire** : Elles sont plus optimales dans des cas où la fonction ne sera utilisée qu'une fois ou dans une situation locale (par exemple, dans des fonctions comme `map()`, `filter()`, ou `sorted()`), car elles n'impliquent pas le stockage dans une variable.
- **Clarté** (dans certaines situations) : En particulier lors de l'utilisation de fonctions qui acceptent d'autres fonctions en arguments.

Limites :

- **Lisibilité réduite** : Les fonctions lambda peuvent rendre le code moins lisible si elles deviennent trop complexes.
- **Pas de documentation** : Contrairement aux fonctions classiques, les fonctions lambda ne peuvent pas avoir de docstrings (commentaires explicatifs).

A noter qu'il est possible de stocker dans une variable une fonction définie par une expression lambda, mais l'usage le plus courant est de basculer alors vers la syntaxe avec `def`.

```
square = lambda x: x ** 2
```

Fonctions

Documentation

- En Python, la documentation d'une fonction est une chaîne de caractères placée juste sous la définition de celle-ci.
- Elle est appelée **docstring** (abréviation de documentation string).
- Elle sert à expliquer ce que fait le code, ses paramètres et sa valeur de retour.
- Elle peut être consultée avec la fonction `help()` ou l'attribut `.__doc__`.
- Au-delà d'une fonction, une docstring peut aussi être rattachée à une classe ou à un module.

Les bonnes pratiques :

- Toujours commencer par une phrase concise décrivant le but de la fonction. Cette phrase doit pouvoir tenir sur une ligne.
- Si besoin, détailler ensuite les explications (optionnel).
- Détailler ensuite les arguments, valeurs de retour, exceptions éventuelles.
- Respecter un format standard (pour les outils de documentation automatique).

Fonctions

Documentation

Format NumPy :

```
def normalize(data, mean=None, std=None):
    """
    Normalize a NumPy array using the provided mean and standard deviation.

    Parameters
    -----
    data : numpy.ndarray
        Input array to be normalized.
    mean : float, optional
        Mean value to use for normalization. If None, the mean of `data` is used.
    std : float, optional
        Standard deviation to use for normalization. If None, the standard deviation of `data` is used.

    Returns
    -----
    numpy.ndarray
        The normalized array with zero mean and unit variance.

    Raises
    -----
    ValueError
        If `std` is zero or negative.

    Examples
    -----
    >>> import numpy as np
    >>> x = np.array([1, 2, 3, 4, 5])
    >>> normalize(x)
    array([-1.2649, -0.6324,  0.0000,  0.6324,  1.2649])
    """
    import numpy as np
    if std is None:
        std = np.std(data)
    if mean is None:
        mean = np.mean(data)
    if std <= 0:
        raise ValueError("Standard deviation must be positive.")
    return (data - mean) / std
```

Fonctions

Documentation

Format Google :

```
def normalize(data, mean=None, std=None):
    """
    Normalizes a NumPy array using the provided mean and standard deviation.

    Args:
        data (numpy.ndarray): Input array to be normalized.
        mean (float, optional): Mean value to use for normalization.
            If None, the mean of `data` is used.
        std (float, optional): Standard deviation to use for normalization.
            If None, the standard deviation of `data` is used.

    Returns:
        numpy.ndarray: The normalized array with zero mean and unit variance.

    Raises:
        ValueError: If `std` is zero or negative.

    Examples:
        >>> import numpy as np
        >>> x = np.array([1, 2, 3, 4, 5])
        >>> normalize(x)
        array([-1.2649, -0.6324,  0.0000,  0.6324,  1.2649])
    """
    import numpy as np
    if std is None:
        std = np.std(data)
    if mean is None:
        mean = np.mean(data)
    if std <= 0:
        raise ValueError("Standard deviation must be positive.")
    return (data - mean) / std
```

Fonctions

Documentation

Format reStructuredText (Sphinx) :

```
def normalize(data, mean=None, std=None):
    """
    Normalize a NumPy array using the provided mean and standard deviation.

    :param data: Input array to be normalized.
    :type data: numpy.ndarray
    :param mean: Mean value to use for normalization. If None, the mean of ``data`` is used.
    :type mean: float, optional
    :param std: Standard deviation to use for normalization. If None, the standard deviation
    of ``data`` is used.
    :type std: float, optional

    :returns: The normalized array with zero mean and unit variance.
    :rtype: numpy.ndarray

    :raises ValueError: If ``std`` is zero or negative.

    :example:
        >>> import numpy as np
        >>> x = np.array([1, 2, 3, 4, 5])
        >>> normalize(x)
        array([-1.2649, -0.6324,  0.0000,  0.6324,  1.2649])
    """
    import numpy as np
    if std is None:
        std = np.std(data)
    if mean is None:
        mean = np.mean(data)
    if std <= 0:
        raise ValueError("Standard deviation must be positive.")
    return (data - mean) / std
```


05

Les générateurs

Les générateurs

Les générateurs en Python sont des fonctions spéciales qui permettent de produire une séquence de valeurs paresseusement (c'est-à-dire à la demande) plutôt que de les calculer et stocker en mémoire d'un coup. Cela les rend très efficaces pour manipuler de grandes quantités de données.



Les générateurs

Caractéristiques des générateurs

- ➡ **Itérables** : les générateurs peuvent être parcourus avec une boucle for, ils implémentent les protocoles itérateurs de Python, avec les méthodes `__iter__()` et `__next__()`.
- ➡ **Paresseux** : les valeurs ne sont produites qu'une seule à la fois, au fur et à mesure de la demande, ce qui permet d'économiser de la mémoire, surtout pour des séquences volumineuses.

Les générateurs

Création d'un générateur

```
def count_up_to(n):  
    i = 1  
    while i <= n:  
        yield i  
        i += 1  
  
gen = count_up_to(5)
```



Que fait ce code ?

```
print(next(gen))  
print(next(gen))  
for number in gen:  
    print(number)
```

```
gen_2 = (x**2 for x in range(10))  
print(next(gen_2))  
print(next(gen_2))
```

Les générateurs

	Générateur	Liste
Mémoire	Calcul paresseux, faible utilisation	Tous les éléments en mémoire
Performance	Calcul à la demande, rapide	Initialisation complète requise
Mutabilité	Non mutable	Mutable
Utilisation	Grands ensembles, flux de données	Petites séquences, accès à un élément à une position précise (ou aléatoire)

Les générateurs



Que font ces exemples de générateurs ?

```
def read_lines(file_path):  
    with open(file_path) as file:  
        for line in file:  
            yield line.strip()
```

```
def double(numbers):  
    for n in numbers:  
        yield n * 2  
  
nums = (x for x in range(10))  
doubled = double(nums)
```

```
def fibonacci():  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b
```

Les générateurs

- Une fois qu'un générateur a produit toutes ses valeurs, il ne peut plus être utilisé. S'il l'est alors qu'il n'a plus rien à produire, il lève une erreur de type `StopIteration`.
- Un générateur peut recevoir des valeurs via la méthode `send()`.
- Un générateur est terminé proprement grâce à `close()`.

06

Les modules

Modules

Que sont-ils ?

- Les modules en Python sont des collections de fonctions et de code réutilisables que les développeurs peuvent utiliser pour effectuer diverses tâches. En réalité, ce sont simplement des fichiers Python contenant du code Python.
- Bien que la bibliothèque standard de Python inclue de nombreux modules couvrant une large gamme de fonctionnalités, les développeurs ont également créé d'innombrables modules tiers disponibles pour utilisation.
- Il est fortement conseillé de vérifier si la fonctionnalité dont vous avez besoin existe déjà sous forme de module avant d'écrire votre propre code. La [documentation officielle](#) de Python fournit des informations détaillées sur les modules standards, et une documentation supplémentaire pour les modules tiers peut souvent être trouvée en ligne.

c	
calendar	Functions for working with calendars, including some emulation of the Unix cal program.
cgi	Obsolète: Helpers for running Python scripts via the Common Gateway Interface.
cgitb	Obsolète: Configurable traceback handler for CGI scripts.
chunk	Obsolète: Module to read IFF chunks.
cmath	Mathematical functions for complex numbers.
cmd	Build line-oriented command interpreters.
code	Facilities to implement read-eval-print loops.
codecs	Encode and decode data and streams.
codeop	Compile (possibly incomplete) Python code.
* collections	Container datatypes
colorsys	Conversion functions between RGB and other color systems.
compileall	Tools for byte-compiling all Python source files in a directory tree.
* concurrent	
configparser	Configuration file parser.
contextlib	Utilities for with-statement contexts.
contextvars	Context Variables
copy	Shallow and deep copy operations.



Bien que la bibliothèque standard de Python contienne à elle seule plus de 300 modules, l'écosystème Python inclut des milliers de modules supplémentaires disponibles sur des plateformes comme le Python Package Index (PyPI) et d'autres dépôts.

Modules

Importer des modules

```
>>> import random
>>> random.randint(0, 10)
4
```

- ➡ Dans la ligne 1, l'instruction **import** donne accès à toutes les fonctions du module `random`.
- ➡ Dans la ligne 2, nous utilisons la fonction **randint()** du module `random`. Cette fonction retourne un entier tiré aléatoirement entre 0 et 10.

```
>>> import math
>>> math.cos(math.pi / 2)
6.123233995736766e-17
>>> math.sin(math.pi / 2)
1.0
```

Modules

Importer des modules

Il existe une autre façon d'importer une ou plusieurs fonctions à partir d'un module. En utilisant le mot-clé **from**, vous pouvez importer une fonction spécifique d'un module donné. Dans ce cas, vous n'avez pas besoin de répéter le nom du module lorsque vous l'utilisez, mais seulement le nom de la fonction concernée.

```
>>> from random import randint
>>> randint(0, 10)
6
```

Modules

Importer des modules

Enfin, vous pouvez également importer toutes les fonctions d'un module en utilisant `import *`.

```
>>> from random import *  
>>> randint(0, 10)  
3  
>>> uniform(0, 2.5)  
0.74943174760727951
```

C'est une syntaxe que vous pouvez rencontrer, mais qui est généralement **déconseillée dans le code de production** car elle peut entraîner une **pollution de l'espace de noms** (namespace) et rendre le code **moins lisible et maintenable**. Elle importe tous les noms définis dans le module dans l'espace de noms actuel, ce qui peut provoquer des conflits et rendre difficile la traçabilité de l'origine des fonctions et des variables.



Utilisez les deux autres syntaxes.

Modules

Obtenir de l'aide sur les modules importés

Vous pouvez utiliser la fonction intégrée `help()`.



La commande `help()` est en réalité une commande plus générale qui permet d'obtenir de l'aide sur n'importe quel objet chargé en mémoire.

```
>>> import random
>>> help(random)
```

```
Help on module random :
```

```
NAME
```

```
    random - Random variable generators.
```

```
MODULE REFERENCE
```

```
    https ://docs.python.org/3.7/library/random
```

```
    The following documentation is automatically generated from the Python
    source files. It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations. When in doubt, consult the module reference at the
    location listed above.
```

```
DESCRIPTION
```

```
    integers
```

```
    -----
```

```
        uniform within range
```

```
    sequences
```

```
    -----
```

```
        pick random element
```

```
        pick random sample
```

Modules

Quelques modules courants

- Il existe un certain nombre de modules que vous êtes susceptibles d'utiliser si vous programmez en Python.
 - `math` fonctions mathématiques de base et constantes (sin, cos, exp, pi...)
 - `sys` interaction avec l'interpréteur Python, notamment pour passer des arguments en ligne de commande et accéder à des informations sur l'environnement d'exécution,
 - `os` facilite le dialogue avec le système d'exploitation, permettant des opérations telles que la manipulation de fichiers et de répertoires, ainsi que l'accès aux variables d'environnement,
 - `random` génération de nombres aléatoires,
 - `time` accès à l'heure de l'ordinateur et fonctions de gestion du temps,
 - `urllib` récupération de données sur Internet directement depuis Python,
 - `re` gestion des expressions régulières,
- N'hésitez pas à consulter la [page des modules](#) sur le site officiel Python.
- Il existe de nombreux autres modules externes qui ne sont pas installés par défaut avec Python mais qui sont largement utilisés dans la pratique : Flask, Requests, TensorFlow, scikit-learn, Matplotlib, seaborn, SQLAlchemy...

Modules

Modules, packages, bibliothèques, frameworks

Dans l'univers du développement en Python, il existe quelques concepts distincts qui constituent les fondations de la structure du code et influencent la manière dont les développeurs organisent et réutilisent le code pour construire des applications robustes et modulaires : les modules, packages, bibliothèques (parfois appelées librairies par anglicisme) et frameworks (pour le coup très rarement traduits en français).

Je vous suggère la lecture de cet excellent article de LearnPython qui explique l'utilité de et les différences entre ces quatre concepts : [Python Modules, Packages, Libraries, and Frameworks](#).



Fin de l'annexe