

Python : introduction

MARS 2025



Objectifs

- Structurer des programmes selon un algorithme
- Maîtriser les éléments de lexique et de syntaxe d'un langage pour écrire un programme
- Exécuter un programme
- Déboguer et tester un programme



Présentations

- Ce qui vous semble pertinent (rôle, expérience, attentes...).
- Votre (éventuel) rapport à Python dans votre travail.
- Votre background (pour orienter vers certains axes le discours et les échanges).



Dépôt Github distant



<https://github.com/BEESPE/caceis>

Programme



- 01** **Introduction - programmes et règles de programmation**
- 02** **Les variables**
- 03** **Opérateurs et expressions**
- 04** **Les structures de contrôle**
- 05** **Les procédures et les fonctions**
- 06** **Maintenance, débogage et test des programmes**

Avant-propos

Source principale

Une partie de ce matériel pédagogique provient de l'excellent cours [Introduction to Python programming for biology](#) de Patrick Fuchs et Pierre Poulain, sous licence libre (CC BY-SA 3.0 FR).

Les adaptations faites dans ce support sont relatives à l'identité visuelle des diapositives et le développement de points particulièrement pertinents pour cette formation.

Par conséquent, les informations contenues dans ce support sont placées sous licence CC BY-SA 4.0.



01

Introduction - programmes et règles de programmation

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.



Versions

- ➡ Au jour de notre formation, la dernière version de Python est la version [3.13](#), sortie en octobre 2024. La version 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.

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...

Télécharger et installer Python

A partir du site officiel

- Vous pouvez télécharger et installer Python directement à partir du site [officiel de Python](#). Vous pouvez par exemple suivre les étapes générales de ce [tutoriel](#).
- **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.



Télécharger et installer 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 convivial appelé conda pour gérer les environnements et installer des paquets supplémentaires.



Télécharger et installer 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.

Télécharger et installer 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 (que nous utiliserons aujourd'hui pour ceux qui voudront bien créer un compte ou utiliser un compte existant) 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.



Editeurs et I.D.E.

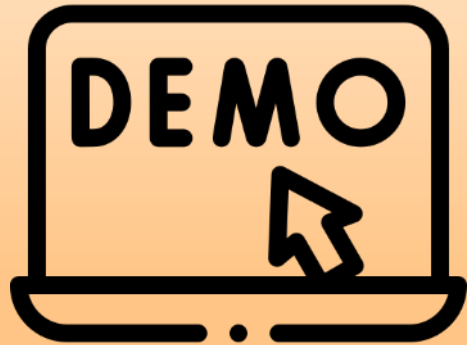
Principaux I.D.E.

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...

Editeurs et I.D.E.

Démo de VSC



Visual Studio Code

Editeurs et I.D.E.

Autres

- 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...).

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.

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).

Algorithme, programme

Définitions

Algorithme

- Un algorithme est une **suite finie et ordonnée d'étapes ou d'instructions** qui permet de **résoudre un problème** ou d'**accomplir une tâche**.
- **Clarté** : Les étapes doivent être clairement définies et compréhensibles.
- **Finitude** : L'algorithme doit se terminer après un nombre fini d'étapes.
- **Précision** : Chaque étape doit être non ambiguë.
- **Efficacité** : Il doit produire un résultat correct tout en étant optimal en temps et en espace.
- **Exemple** : Algorithme pour calculer la somme des n premiers entiers
 1. Initialiser une variable somme à 0.
 2. Pour chaque entier i de 1 à n , ajouter i à somme.
 3. Afficher le résultat somme.

Algorithme, programme

Définitions

Programme

- Un programme est une **suite d'instructions écrites dans un langage de programmation** (comme Python) qui sont **exécutées par un ordinateur** pour accomplir une tâche spécifique.
- Il contient :
 - des **instructions**, qui sont les actions que l'ordinateur doit réaliser, comme des calculs, des boucles, ou des appels à des fonctions.
 - des **structures**, telles que des conditions (if/else), des boucles (for, while) et des fonctions qui organisent le code.
 - des **données**, informations manipulées par le programme, stockées sous forme de variables, listes, dictionnaires, etc.
- Un programme Python peut être utilisé pour automatiser des tâches, effectuer des calculs, manipuler des fichiers, interagir avec des utilisateurs...



Un **programme** implémente un ou plusieurs **algorithmes** en utilisant un langage de programmation. L'algorithme est donc une sorte de plan ou de recette, tandis que le programme est la traduction informatique concrète de ce plan.

Premier programme

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.

Premier programme

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! » imprimé sous la cellule de code.

Règles de programmation

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/>

Règles de programmation

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/>

Règles de programmation

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.

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](#).



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`.

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
```

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
```

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.

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
```

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
```

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.

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).

Les environnements

Workflow typique

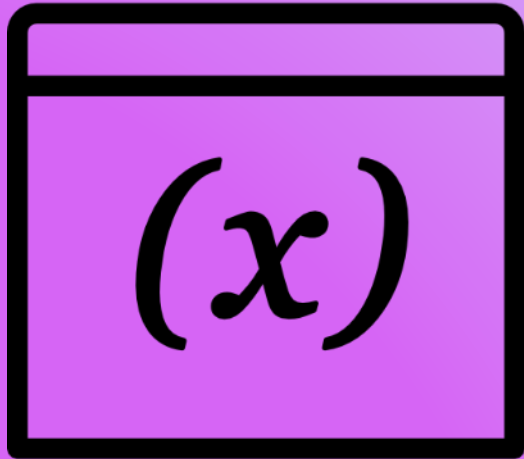
```
# Créer un environnement
python -m venv myenv

# 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
```



02

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 !

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



Manipuler des variables

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 :

```
>>> ani_1 = ['giraffe', 'tiger']
>>> ani_2 = ['monkey', 'mouse']
>>> ani_1 + ani_2
['giraffe', 'tiger', 'monkey', 'mouse']
>>> ani_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.

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




Manipuler des listes

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
```



Autres collections

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()`.

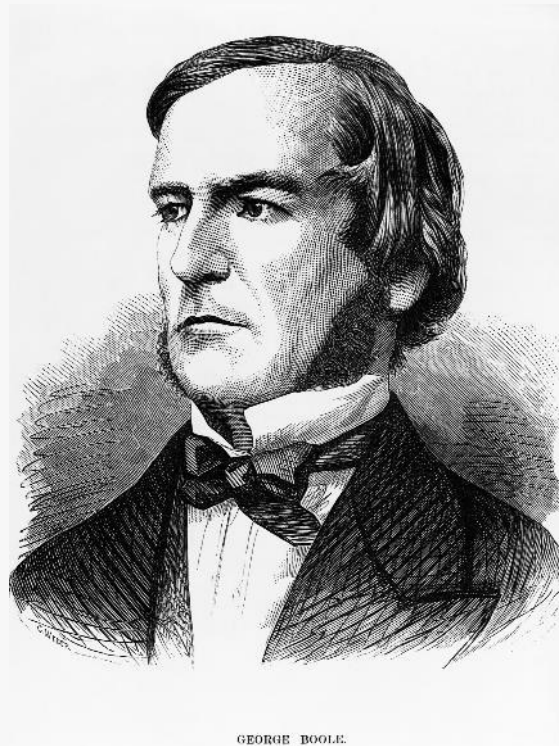
03

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
```



04

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]
```



Manipuler des listes (en plus fortiche) et des boucles

$f(x)$

05

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

Portée, variables locales et globales

- When manipulating functions, it is essential to understand how variables behave.
- A variable is called **local** when it is created within a function. It will only exist and be visible during the execution of that function.
- A variable is called **global** when it is created in the main program. It will be visible throughout the program.

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



[Python Tutor](https://pythontutor.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
```




Manipuler des fonctions

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

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](#)



06

Maintenant, débogage et test des programmes

Tests unitaires

Ce que les tests unitaires ne sont pas



Tests unitaires

2 parties



Pourquoi faire des tests unitaires ?



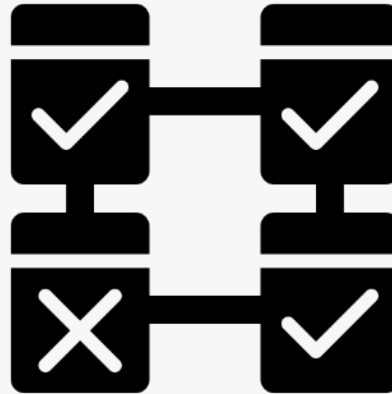
Les éléments utiles pour écrire ses tests unitaires

Tests unitaires

Définition



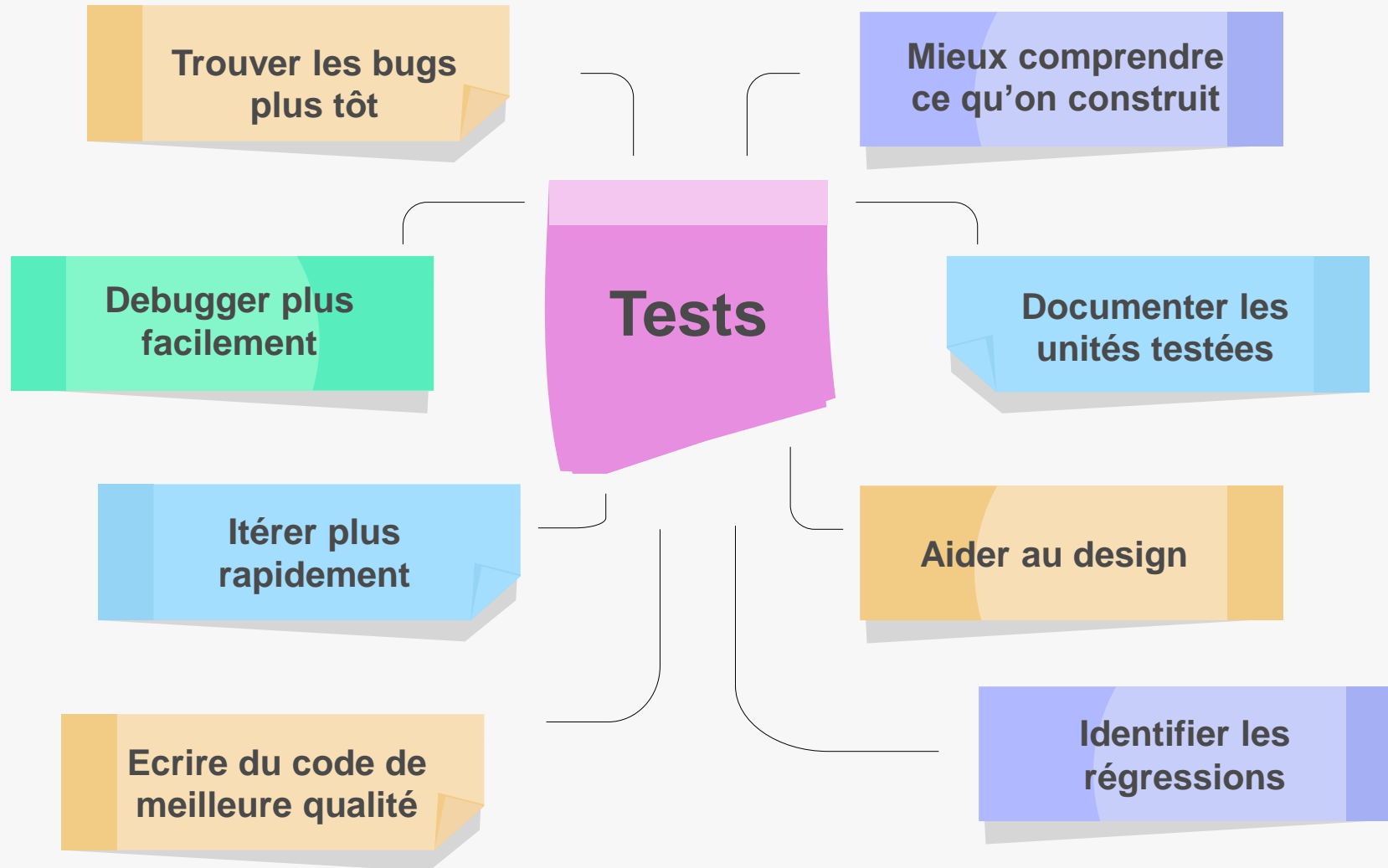
Tester des morceaux de code dans un context isolé...



... pour s'assurer que votre code fait ce que vous attendez de lui.

Tests unitaires

Avantages



Tests unitaires

Pourquoi tout le monde n'en fait pas ?



- C'est un double investissement :
 - ➡ cela prend du temps pour les écrire,
 - ➡ cela prend du temps pour apprendre à le faire correctement.
- L'analyse des données suit parfois des modèles différents de ceux de l'ingénierie logicielle.



Tests unitaires

Vocabulaire

Cas de test (test case) :

- Strictement parlant : une série d'instructions qui valident qu'une partie du logiciel fonctionne comme prévu.
- Dans certaines sources : unité de test individuelle.
- Vérifie une réponse spécifique à un ensemble particulier d'entrées.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                           'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                           'wrong size after resize')
```

Tests unitaires

Vocabulaire

Suite de tests (test suite) :

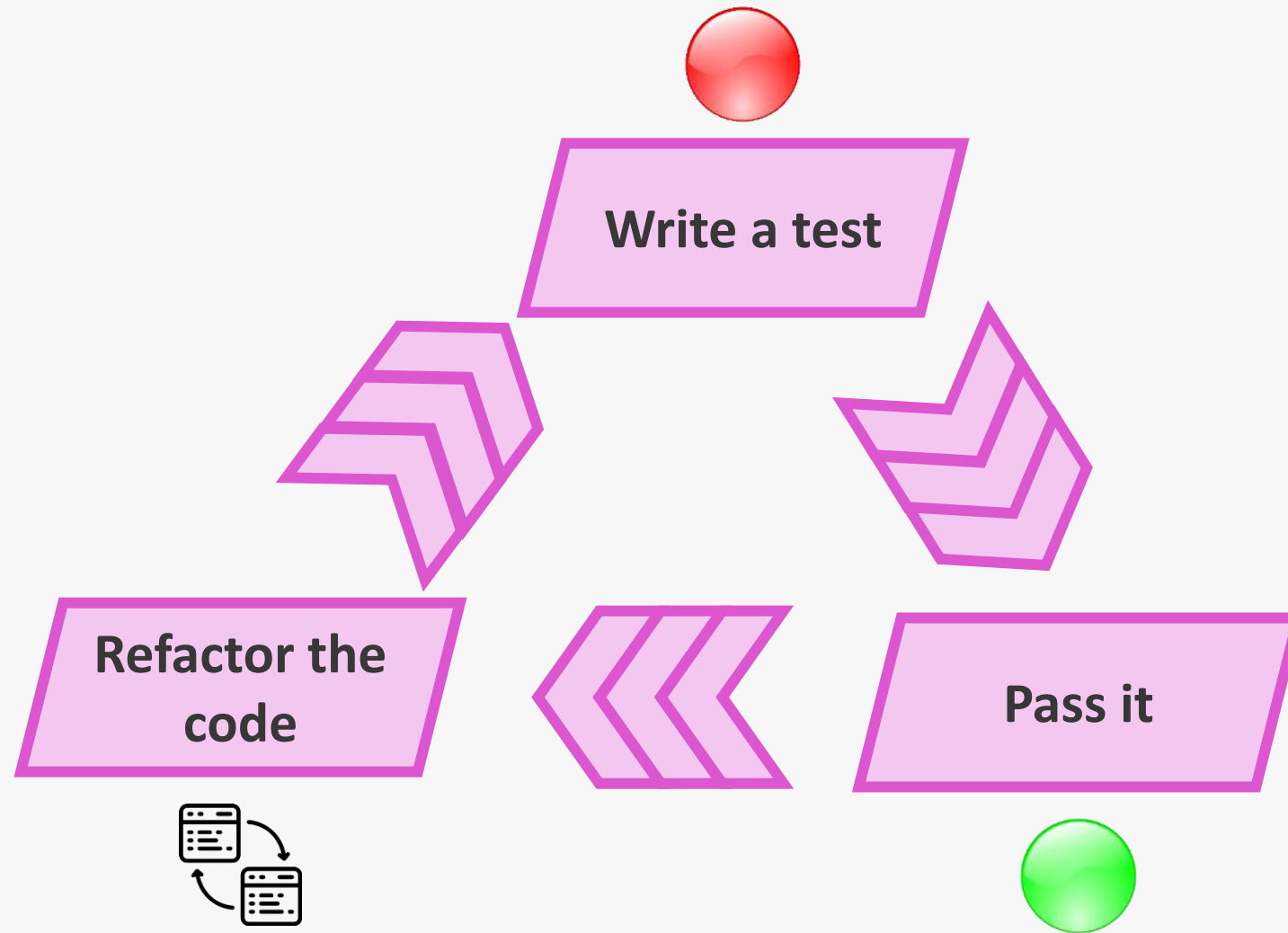
- Ensemble de cas de test, suites de tests ou les deux.
- Utilisée pour regrouper les tests qui doivent être exécutés ensemble.

Exécuteur de tests (test runner) :

- Composant qui orchestre l'exécution des tests et fournit les résultats à l'utilisateur.
- Peut utiliser une interface graphique, une interface textuelle ou retourner une valeur spéciale pour indiquer les résultats de l'exécution des tests.

Tests unitaires

Test driven development (TDD)



Tests unitaires

Test driven development (TDD) en analyse de données

Nay!

Analyse exploratoire

POC simple / étude de faisabilité

**Données complètes et
faciles à utiliser**

**Travail en solo, par
exemple analyses ad hoc**

Yea!

Pipeline analytique

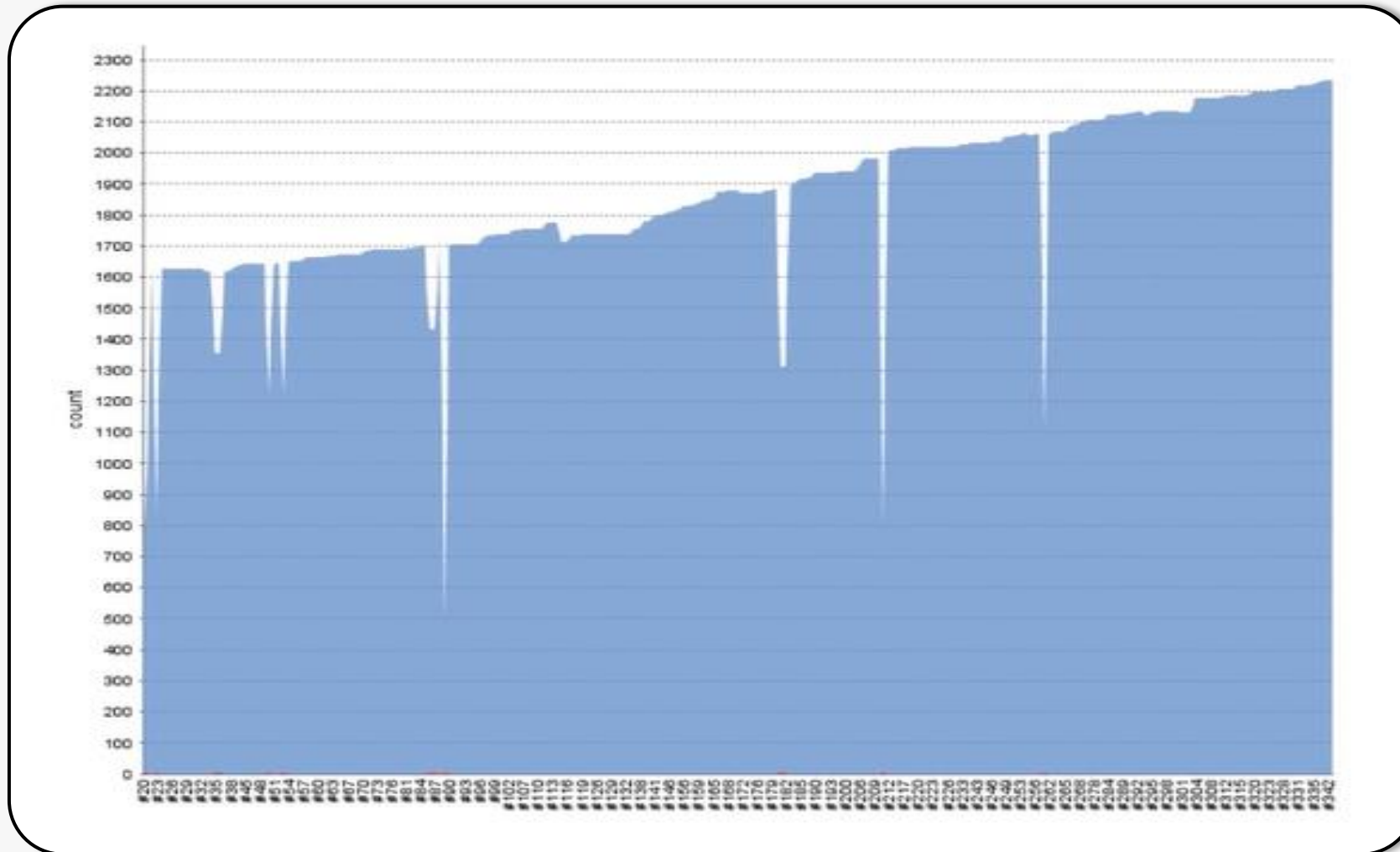
POC complexe

Données non exhaustives

Travail collaboratif

Tests unitaires

Intégration continue



Tests unitaires

Intégration continue

Makefile 774 bytes

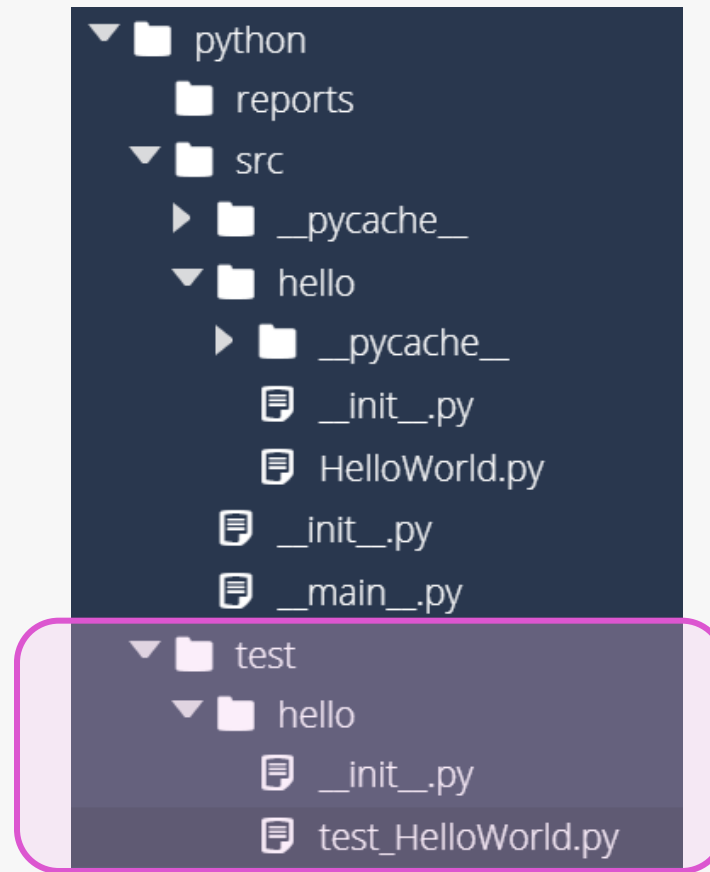
Open in Web IDE

Lock Replace Delete

```
1 PROJECT_DIRECTORY = 
2 
3 init-prod:
4     pipenv install
5 
6 init-dev:
7     pipenv install --dev
8 
9 test:
10     pipenv run \
11     pytest --cov-report xml:coverage.xml --cov-report term --cov ${PROJECT_DIRECTORY} \
12     --junitxml test_results.xml tests/
13 
14 build:
15     pipenv install
16     pipenv run python setup.py bdist_wheel
17 
18 test-no-cov:
19     pipenv run pytest tests/
20 
21 clean: clean-build clean-pyc clean-test
22 
23 clean-build:
24     rm -rf dist/
25     rm -rf build/
26     rm -rf *.egg-info
27 
28 clean-pyc:
29     find . -name '*.pyc' -exec rm -f {} +
30     find . -name '*.pyo' -exec rm -f {} +
```


Tests unitaires

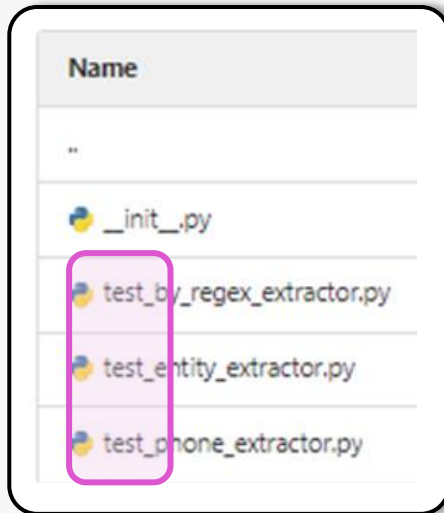
Structure d'un projet



Tests unitaires

Convention

Convention : Préfixer tous les noms de fichiers et les fonctions de test par "test_".



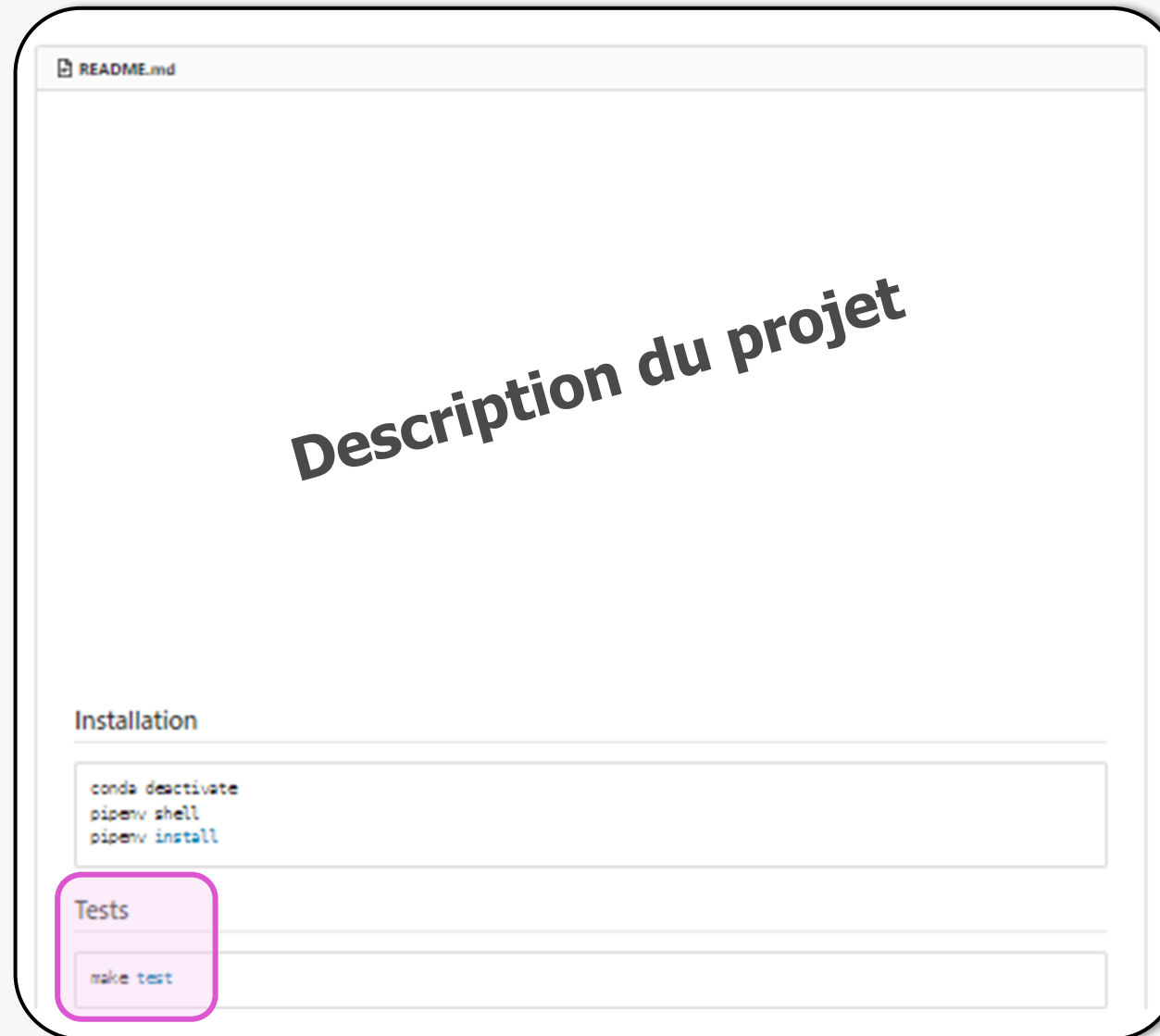
```
def test_extract_regex_from_empty_dataframe(by_regex_email_extractor, by_regex_siret_extractor):  
    assert by_regex_email_extractor.transform(pd.DataFrame({})).equals(pd.DataFrame({}))  
    assert by_regex_siret_extractor.transform(pd.DataFrame({})).equals(pd.DataFrame({}))  
  
def test_extract_from_empty_body(by_regex_email_extractor, by_regex_siret_extractor):  
    empty_body_df: pd.DataFrame = pd.DataFrame({"test_id": [0], "test_body": [None]})  
    email_expected_output: pd.DataFrame = pd.DataFrame(  
        {"test_id": [0], "test_body": [None], "test_email": [[]]}  
    )  
    siret_expected_output: pd.DataFrame = pd.DataFrame(  
        {"test_id": [0], "test_body": [None], "test_siret": [[]]}  
    )  
    assert by_regex_email_extractor.transform(empty_body_df).equals(email_expected_output)  
    assert by_regex_siret_extractor.transform(empty_body_df).equals(siret_expected_output)
```



Les frameworks comme pytest recherchent les fichiers et les répertoires dont les noms correspondent à des motifs spécifiques. Par défaut, pytest identifie les fichiers de test dont les noms commencent ou se terminent par test (par exemple, test_example.py ou example_test.py).

Unit testing

Documentation



Tests unitaires

Setup and teardown

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                          'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                          'wrong size after resize')
```

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

Tests unitaires

Frameworks

Bibliothèques Python pour écrire des tests unitaires (frameworks fonctionnant comme exécuteurs de tests) :



Tests unitaires

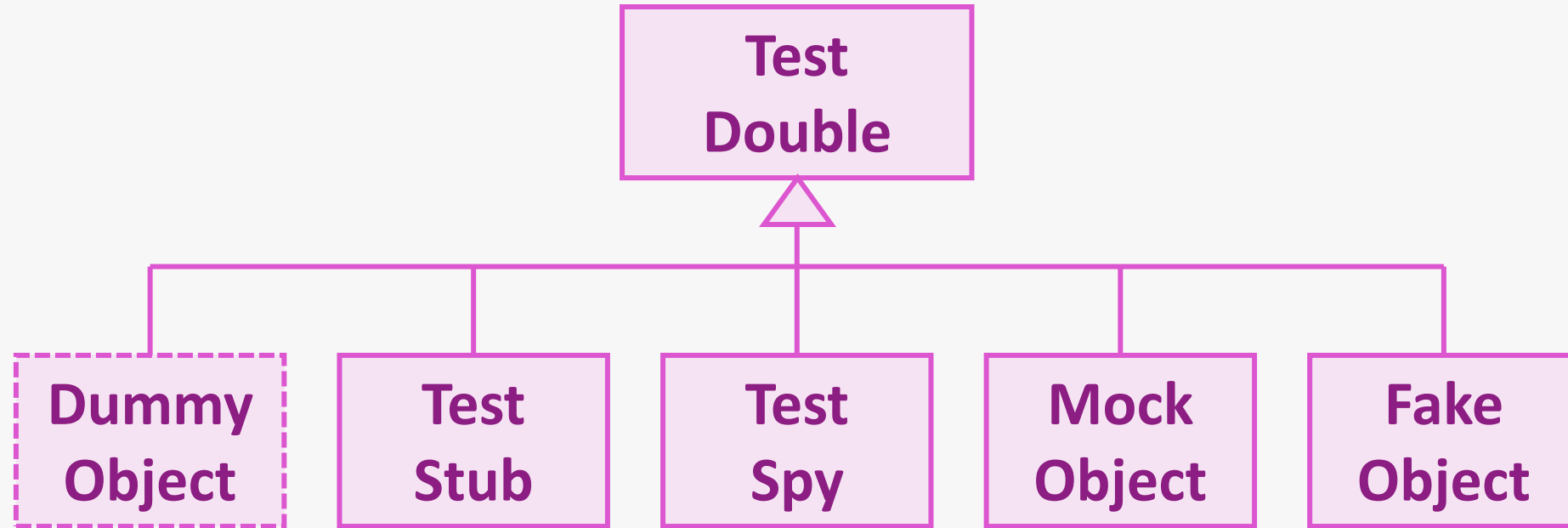
Les fixtures (de pytest)

- Principalement, pour créer un objet de test et le supprimer automatiquement (mais pas uniquement !).
- Alternative à une (grande) partie des méthodes setup et teardown, mais sans être identique.
- En termes absolus, il est encore parfois nécessaire de gérer certains éléments manuellement.
- Certaines fixtures permettent de travailler avec des dossiers temporaires (tmpdir, tmp_path).
- Les fixtures peuvent demander d'autres fonctionnalités (en particulier dépendre d'autres fixtures).

```
@pytest.fixture
def local_airport() -> Airport:
    """
    Pytest fixture acting as default local airport.
    :return: a default test airport with the is_local attribute set to True.
    """
    return Airport(
        name="Paris Charles de Gaulle",
        iata_airport_code="CDG",
        icao_airport_code="LFPG",
        is_local=True,
    )
```

Tests unitaires

Les test doubles



Tests unitaires

Les stubs

```
from sensor import Sensor

class Alarm:

    def __init__(self):
        self._low_pressure_threshold = 17
        self._high_pressure_threshold = 21
        self._sensor = Sensor()
        self._is_alarm_on = False

    def check(self):
        pressure = self._sensor.sample_pressure()
        if pressure < self._low_pressure_threshold \
            or self._high_pressure_threshold < pressure:
            self._is_alarm_on = True

    @property
    def is_alarm_on(self):
        return self._is_alarm_on
```

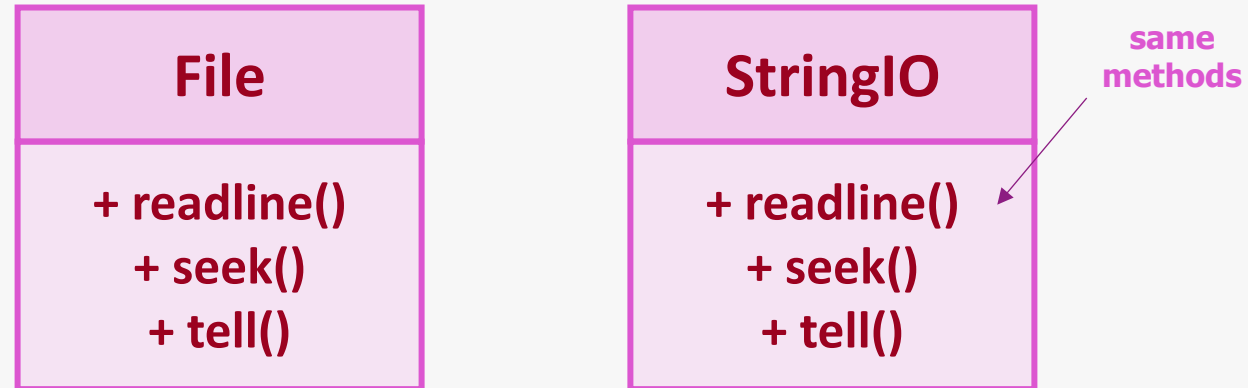


```
class StubSensor:
    def sample_pressure(self):
        return 15

def test_low_pressure_activates_alarm():
    alarm = Alarm(sensor=StubSensor())
    alarm.check()
    assert alarm.is_alarm_on
```


Tests unitaires

Les fakes



```
def test_access_second_page():
    fake_file = io.StringIO("""\
page one
PAGE_BREAK
page two
PAGE_BREAK
page three
""")
    converter = HtmlPagesConverter(open_file=fake_file)
    converted_text = converter.get_html_page(1)
    assert converted_text == "page two<br />"
```

Tests unitaires

Les fakes



Fichiers

A remplacer par StringIO



Bases de données

A remplacer par des bases de données en mémoire



Serveurs web

A remplacer par des serveurs web légers

Tests unitaires

Les mocks

```
from datetime import datetime

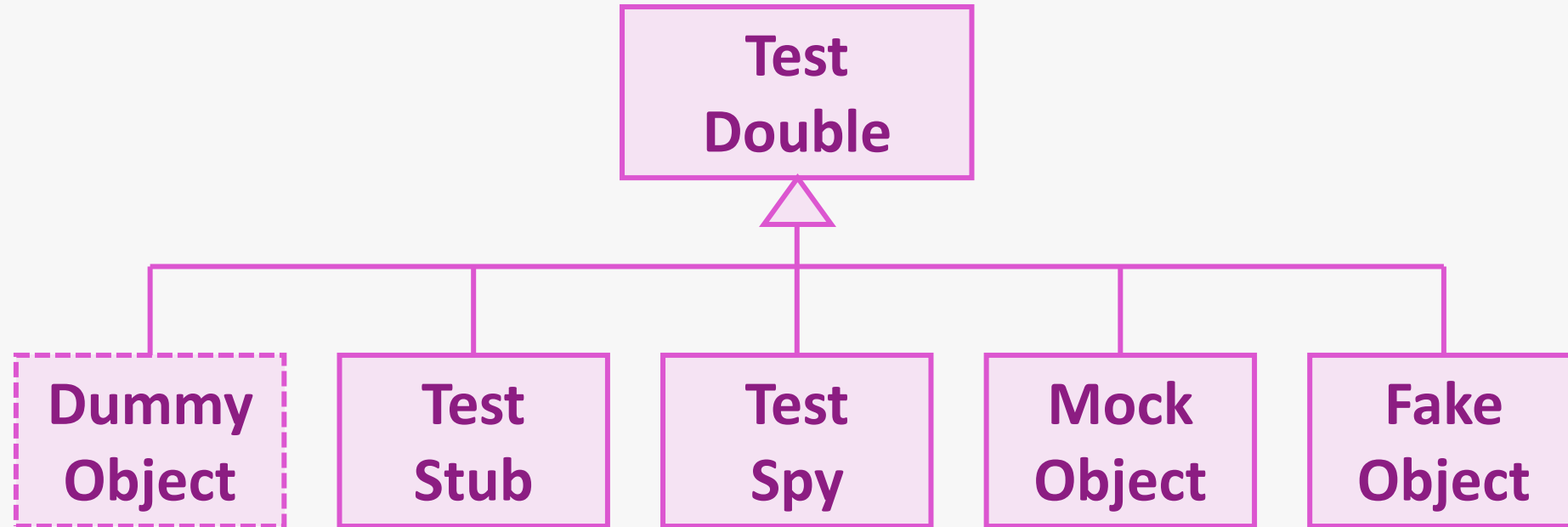
def is_weekday():
    today = datetime.today()
    # Python's datetime library treats Monday as 0 and Sunday as 6
    return (0 <= today.weekday() < 5)

# Test if today is a weekday
assert is_weekday()
```

```
MockBlabla = Blabla
MockBlabla.any_method = Mock(return_value=any_value)
```

Tests unitaires

Les test doubles



Tests unitaires


Paramétrer ses tests

```
from tennis import score_tennis
```

```
def test_0_0_love_all():  
    assert score_tennis(0, 0) == "Love-All"
```

```
def test_1_1_fifteen_all():  
    assert score_tennis(1, 1) == "Fifteen-All"
```

```
def test_2_2_thirty_all():  
    assert score_tennis(2, 2) == "Thirty-All"
```



```
@pytest.mark.parametrize("player1_points, player2_points, expected_score",  
                           [(0, 0, "Love-All"),  
                            (1, 1, "Fifteen-All"),  
                            (2, 2, "Thirty-All"),  
                           ])  
  
def test_score_tennis(player1_points, player2_points, expected_score):  
    assert score_tennis(player1_points, player2_points) == expected_score
```

Tests unitaires

Astuce

Il est possible de combiner les fixtures et la paramétrisation (c'est-à-dire paramétrer avec des fixtures), en utilisant `request.getfixturevalue(...)` pour accéder au paramètre.

Tests unitaires

Couverture de test (test coverage)

- Couverture des tests : simplement la partie du code qui est testée.
- Outils principaux : pytest-cov, Coverage.

```
8  
9 test:  
10     pipenv run \  
11     pytest --cov-report xml:coverage.xml --cov-report term --cov ${PROJECT_DIRECTORY} \  
12     --junitxml test_results.xml tests/  
13
```

Coverage for **tennis.py** : 62%

16 statements 10 run 6 missing 2 excluded

```
1  
2  
3 def score_tennis(player1_points, player2_points):  
4     score_names = ["Love", "Fifteen", "Thirty", "Forty"]  
5     if _end_game(player1_points, player2_points):  
6         leader = "Player 2"  
7         if player1_points > player2_points:  
8             leader = "Player 1"  
9  
10        if abs(player2_points - player1_points) == 1:  
11            return f"Advantage {leader}"  
12        else:  
13            return f"Win for {leader}"  
14        player1_score = score_names[player1_points]  
15        if player1_points == player2_points:  
16            return f"{player1_score}-All"  
17    else:
```

Notez que pour une fonction, une couverture de 100 % ne signifie pas nécessairement qu'un seul test est suffisant.

C'est particulièrement vrai dans le cas des boucles IF.



FINAL BOSS

Fin de la formation

Merci !