Python, Les Fondamentaux

Darko Stankovski

16 oct. 2026

Table des matières

Présentation de la formation **8**
Pré-requis 8

Première partie **L'environnement de travail** **9**

1 **Boite à outils** **10**
1.1 L'interpréteur 10
1.2 Environnement de développement 10
1.2.1 Thonny 11
1.2.2 Visual Studio Code 11
1.2.3 PyCharm 11
1.2.4 Jupyter-lab 11
1.3 Outil de gestion de projet : uv 11

2 **L'environnement de travail** **12**
2.1 Isoler son environnement de travail 12
2.1.1 Création de l'environnement virtuel 12
2.1.2 Activation de l'environnement virtuel 13
2.1.3 Désactiver un environnement virtuel 13
2.1.4 Supprimer un environnement virtuel 13
2.1.5 Les environnements virtuels et les gestionnaires de version 13
2.1.6 Les environnements virtuels et les IDEs 14
2.2 Gérer des dépendances de manière standard 14
2.2.1 Installer une dépendance 14
2.2.2 Mettre à jour une dépendance 14
2.2.3 Supprimer une dépendance 14
2.2.4 Lister les dépendances 14
2.2.5 Gérer les dépendances d'un projet, approche historique 15
2.3 L'approche packaging standard 15
2.3.1 Installation du projet en mode éditable 15
2.3.2 Une gestion centralisée des dépendances 16

Deuxième partie **Les fondamentaux** **17**

3 **Données et variables** **18**
3.1 Les données 18
3.2 Les variables 18

3.3 Règles de nommage . . . 19
3.4 Conventions de codage : la PEP 8 . . . 19
3.5 Affecter une donnée à une variable . . . 20
3.6 Absence de déclaration du type . . . 20
3.7 Manipuler et comprendre des données . . . 20

4 Les types de base (built-in) 21
4.1 Les types numériques . . . 21
4.2 Opérateurs arithmétiques . . . 22
4.3 Les opérateurs *bitwise* . . . 22
4.4 Les collections . . . 23
4.4.1 Instructions communes . . . 24
4.4.2 Les séquences : accès aux éléments . . . 24
4.4.3 Modification d'un élément . . . 25
4.5 Slicing . . . 25
4.5.1 Usage . . . 25
4.5.2 Mais pourquoi cette exclusion ? . . . 26

5 Les structures de contrôle 28
5.1 Les structures . . . 28
5.1.1 Les blocs . . . 28
5.1.2 L'instruction `pass` . . . 28
5.1.3 Exécution conditionnelle avec `if` . . . 29
5.1.4 Les opérateurs de comparaison . . . 29
5.1.5 Les valeurs booléennes . . . 30
5.1.6 Les opérateurs logiques . . . 30
5.1.7 Chainage des opérateurs de comparaison . . . 30
5.1.8 Comparaison de tuples . . . 31

6 Les fonctions 32
6.1 Structure . . . 32
6.2 Retours . . . 32
6.3 Paramètres . . . 32
6.3.1 Paramètres positionnels . . . 32
6.3.2 Paramètres optionnels . . . 32
6.3.3 Arguments nommés . . . 33
6.4 Utilisation des type hints . . . 33

7 Les séquences 34
7.1 Création . . . 34
7.1.1 Créer une chaine de caractères . . . 34
7.1.2 Créer une liste . . . 34
7.1.3 Créer un n-uplet (*tuple*) . . . 34
7.2 Taille . . . 35
7.3 Accès à un élément . . . 35
7.4 Test élément dans séquence . . . 35

7.5 Unpacking . . . 36
7.6 Slicing . . . 36

Troisième partie La Programmation Orientée Objet 37

8 Concepts de la Programmation Orientée Objet 38
8.1 Les paradigmes de programmation . . . 38
8.2 Le concept d'objet . . . 38
8.3 La notion de classe . . . 39
8.4 Représentation : présentation d'UML . . . 39
8.4.1 Représenter une classe . . . 39
8.4.2 Représenter les attributs et méthodes . . . 39
8.4.3 Notion de visibilité des attributs et méthodes . . . 40
8.4.4 Cardinalité et contraintes . . . 41
8.4.5 La notion de relations . . . 41
8.4.6 Relation simple . . . 41
8.5 UML : les limites de la représentation graphique . . . 42

9 La Programmation Orientée Objet en Python 43
9.1 Déclarer une classe . . . 43
9.2 Les méthodes . . . 43
9.2.1 appels . . . 44
9.2.2 Les attributs . . . 44
9.2.3 En Python, un objet n'a que des attributs . . . 45
9.2.4 Méthode particulière : le constructeur . . . 45
9.2.5 Méthode particulière : le destructeur . . . 46
9.2.6 Utilisation du constructeur . . . 46
9.2.7 Méthodes et état de l'objet . . . 47
9.3 Visibilité et encapsulation en Python . . . 48
9.3.1 Principe d'encapsulation . . . 49
9.3.2 Name Mangling . . . 49
9.3.3 Les properties . . . 50
9.4 Les méthodes spéciales . . . 52
9.5 Attributs et méthodes de classes . . . 53
9.5.1 Attributs de classe . . . 53
9.5.2 Méthodes de classe . . . 54
9.6 Héritage et abstraction . . . 54
9.6.1 L'héritage . . . 54
9.6.2 Base de l'héritage avec les exceptions . . . 55
9.6.3 Base de conception de l'héritage . . . 55
9.6.4 Notion d'implémentation de l'héritage . . . 57
9.6.5 Héritage simple et héritage multiple . . . 59
9.6.6 Héritage et type des objets . . . 59
9.6.7 Abstraction : classes abstraites et classes concrètes . . . 59
9.6.8 L'héritage en Python . . . 60

9.7 Notions de polymorphisme . . . 61
9.8 Polymorphisme et duck typing . . . 62
9.8.1 Ne testez pas si c'est un canard… . . . 62
9.8.2 Ou ne testez pas du tout . . . 63
9.9 Les attributs . . . 63
9.9.1 Affecter et accéder à une valeur dans l'objet . . . 63
9.9.2 Accéder à un attribut en dehors de l'objet . . . 64
9.10 Les méthodes spéciales . . . 64
9.10.1 L'initialiseur . . . 64
9.10.2 La représentation sous forme de chaine de caractères . . . 64
9.10.3 Autres méthodes spéciales . . . 65
9.11 Visibilité des attributs . . . 66

10 Pour quelques notions de plus 67
10.1 Le paramètre `self` de la déclaration des méthodes dans la classe 67
10.2 Notion de properties . . . 68
10.2.1 Accesseurs (*getters*) . . . 68
10.2.2 Mutateurs (*setters*) . . . 69
10.2.3 Cas pratique : notre formation . . . 69

Quatrième partie La lib standard (et plus) 71

11 Les tests en Python 72
11.1 Pourquoi tester ? . . . 72
11.2 Démarche générale pour le test . . . 72
11.2.1 Glossaire . . . 73
11.2.2 Principe des tests . . . 73
11.2.3 Principe des tests unitaires . . . 73
11.3 Doctest, la documentation auto-testée . . . 74
11.4 Les tests unitaires avec unittest . . . 74
11.4.1 Mise en œuvre de Unittest . . . 74
11.4.2 Cas des tests simples . . . 75
11.4.3 Cas des tests modifiant l'état . . . 76
11.4.4 Les assertions . . . 76
11.5 Les tests unitaires avec pytest . . . 77
11.5.1 Tester la levée d'exceptions . . . 78
11.5.2 Une base de tests avec les test fixtures . . . 78
11.6 L'approche test-first . . . 79

12 Les fichiers et le système de fichiers 81
12.1 Le système de fichiers . . . 81
12.1.1 Obtenir le contenu d'un répertoire . . . 81
12.1.2 Exploiter le contenu d'un répertoire . . . 83
12.2 En résumé . . . 84
12.2.1 Obtenir le contenu d'un répertoire . . . 84

12.2.2 Créer des répertoires . . . 84
12.2.3 Supprimer des fichiers et des répertoires . . . 84
12.2.4 Copier, déplacer et renommer . . . 84
12.2.5 Informations sur le nom de fichier . . . 85
12.3 Le module pathlib . . . 85
12.3.1 Fichier ou répertoire ? . . . 86
12.3.2 Obtenir le contenu d'un répertoire . . . 86
12.4 Ouverture et fermeture de fichier . . . 87
12.4.1 Ouvrir un fichier . . . 87
12.4.2 Fermer un fichier . . . 88
12.4.3 Risques d'exceptions . . . 88
12.4.4 Usage idiomatique avec un context manager . . . 89
12.5 Lire et écrire dans un fichier . . . 89
12.5.1 Les méthodes d'écriture . . . 89
12.5.2 Les méthodes de lecture . . . 89
12.5.3 Lecture avec l'itérateur . . . 90
12.6 Manipuler le curseur . . . 90
12.7 Le module pickle . . . 90
12.8 Les fichiers CSV . . . 91
12.8.1 La méthode standard… et ancienne . . . 91
12.8.2 La méthode plus moderne même si complexe . . . 92

13 Python et les bases de données 93
13.1 Connexion à MariaDb . . . 93
13.2 Utilisation d'un Context Manager . . . 95
13.3 Les requêtes . . . 95
13.3.1 Création de tables . . . 95
13.3.2 Insertion de données : requêtes paramétrées . . . 95
13.3.3 Extractions de données . . . 96
13.4 Les transactions . . . 96
13.4.1 Illustrations par un exemple. . . . 96
13.4.2 Mécanisme des transactions . . . 97
13.4.3 Transactions et exceptions . . . 97
13.4.4 Les context managers et les transactions . . . 97
13.5 Organisation du code . . . 98

14 Les dates en Python 99
14.1 Création d'un instant avec la classe datetime . . . 99
14.2 Objet `datetime` et chaines de caractères . . . 100
14.3 Les durées . . . 101
14.4 Jours et heures . . . 101
14.5 Le temps : des objets immuables . . . 102
14.6 Les fuseaux horaires . . . 102
14.6.1 Attribuer une timezone . . . 102
14.6.2 Créer des dates *aware* . . . 103
14.6.3 Re-localiser une date . . . 103

14.6.4 Changement d'heure : une gestion calendaire . . . 103
14.6.5 Calcul des durées réelles . . . 104
14.7 Le calendrier . . . 105
14.7.1 Des fonctions et constantes utilitaires . . . 105
14.7.2 La classe Calendar . . . 106

15 Les expressions rationnelles 107
15.1 Le vocabulaire . . . 107
15.2 Définir un motif . . . 108
15.2.1 Un motif de base . . . 108
15.2.2 Les classes de caractères . . . 108
15.2.3 Nombre d'occurrences . . . 108
15.2.4 La position d'un motif . . . 109
15.2.5 Les groupes de caractères . . . 109
15.2.6 Référence précédente . . . 109
15.3 Utiliser le module re . . . 110
15.3.1 Les fonctions de recherche . . . 110
15.4 Les Match Objects et leurs attributs . . . 111
15.5 Les sous-groupes . . . 111
15.5.1 Groupes non-capturants . . . 111
15.6 Les substitutions . . . 112

Présentation de la formation

Ce document est un support de formation destiné à être utilisé suite à mes formations.

Ce document est préparé pour permettre d'accéder aux informations présentées en formation et non pour être lu comme un manuel.

Pré-requis

Cette formation est destinée à une audience ayant des connaissances de base en *programmation*. Les concepts d'instruction, donnée, variable, structure de contrôle et fonction sont supposés connus. Les notions de base sur l'utilisation du terminal/invite de commande est également nécessaire.

Licence CC-BY-NC-ND Darko Stankovski 2017-2026 version 2026.10

Première partie

L'environnement de travail

1 Boite à outils

Ce chapitre propose un aperçu de l'écosystème des outils de développement Python.

1.1 L'interpréteur

L'interpréteur est le seul outil indispensable. Pour l'installer :

— Pour Windows et Mac OS, télécharger la dernière version de Python sur le site de la fondation Pyhton : https://www.python.org
— Pour Linux, vous pouvez récupérer les sources sur python.org. Mais il sera plus simple d'utiliser le gestionnaire de paquets.

Pour être plus précis, vous n'installez pas que l'interpréteur mais la **distribution standard** (ou distribution de référence) qui vient avec l'interpréteur, la bibliothèque standard et des outils complémentaires.

Parmi ces outils, vous avez :

— **pip** : le gestionnaire de paquets (https://pypi.python.org/pypi/pip)
— **venv** : l'outil de création d'*environneemnts virtuels* qui permet d'isoler les environnements d'exécution (https://docs.python.org/3/library/venv.html)
— **IDLE** : définis comme un *IDE léger*, c'est plutôt un éditeur avancé

1.2 Environnement de développement

L'usage d'un environnement de développement n'est pas nécessaire pour Python. Il est possible d'utiliser un simple éditeur de texte en s'assurant qu'il est configuré en UTF-8. Un IDE (Environnement de Développement Intégré) apportera tout de même certaines facilitées.

Il existe un certain nombre d'environnements intégrés plus ou moins complets. La liste suivante n'est pas exhaustive.

1.2.1 Thonny

Thonny est un éditeur de code créé en 2014. Il est destiné à l'apprentissage du code Python, est simple et complet pour cette tâche. À noter qu'il vient avec sa propre version de l'interpréteur officiel ce qui permet de plonger directement dans l'apprentissage de Python.

https://thonny.org/

1.2.2 Visual Studio Code

L'éditeur de code léger de Microsoft prend en charge Python de manière relativement complète pour ce type d'éditeur basé sur plug-ins. La prise en charge s'améliore avec les versions.

https://code.visualstudio.com/

1.2.3 PyCharm

PyCharm est un IDE proposé par la société JetBrains à qui on doit IntelliJ Idea. Initialement, PyCharm était proposé sous deux licences, une commerciale et une *community*. Ces deux versions ont été fusionnées. L'usage *community* est largement suffisant.

https://www.jetbrains.com/pycharm/download/

1.2.4 Jupyter-lab

Le projet Jupyter est un projet open-source d'application web qui permet de partager des pages contenant du code exécutable. Le projet a évolué de *notebooks* (calepins) vers Jupyter Lab qui se veut proposer un IDE web. Les notebooks sont des fichiers spécifiques qui sont gérés par de nombreux IDEs.

http://jupyter.org/

1.3 Outil de gestion de projet : uv

Python n'a pas réellement d'outil de gestion de projet et de manière générale, pas standard. Plusieurs initiatives ont vu le jour et actuellement, **uv** se démarque.

Créé en février 2024 par la société Astral (à qui nous devons Ruff), `uv` tends à remplacer l'usage de `venv`, `pip` et de la création de packages tout en assurant une cohérence dans le travail d'équipe.

https://docs.astral.sh/uv/

2 L'environnement de travail

Bien que vous puissiez écrire un script Python dans un fichier n'importe où dans votre système de fichiers, travailler de manière plus robuste nécessite de bien préparer son environnement de travail.

2.1 Isoler son environnement de travail

Un projet Python, c'est un interpréteur (sa version) et un ensemble de dépendances. Plusieurs interpréteurs peuvent cohabiter sur un même système. Les dépendances vont être spécifiques à une version d'interpréteur (installer `pytest` pour Python 3.13 ne le rends pas disponible pour 3.14). Chaque interpréteur a donc **un** environnement d'exécution. Si vous avez deux projets Python utilisant la même version d'interpréteur, elles utiliseront aussi le même environnement avec les mêmes dépendances.

Pour isoler les environnement par projet, nous utilisons les **venv**. L'utilitaire pour les créer fait parti du package standard.

Ci dessous, vous avez les grandes lignes mais consultez la documentation : https://docs.python.org/3/library/venv.html pour plus de précisions.

2.1.1 Création de l'environnement virtuel

L'outil s'utilise en ligne de commande et il faut commencer par créer l'environnement virtuel. La commande est de la forme :

```
1 python3.XX -m venv [ENV_DIR]
```

où :

- `python3.XX` est la version de Python utilisée pour créer le `venv` et sera la version active de cet environnement. En fonction de votre système, vous pourrez avoir besoin de déclarer le chemin absolu.
- `-m venv` est la commande de création du `venv`.
- `[ENV_DIR]` est le nom du répertoire où sera créé le `venv`. De manière générale, il est créé à la racine du projet et est nommé `.venv`.

La commande habituelle, exécutée dans le répertoire du projet, est sous cette forme (en supposant l'usage de `python3.14`) :

```
python3.14 -m venv .venv
```

2.1.2 Activation de l'environnement virtuel

Créer un environnement virtuel consiste à créer un répertoire. Pour pouvoir l'utiliser, il faut l'activer. La commande va dépende re votre système. Les exemples suivants considèrent que l'environnement virtuel a été créé dans un répertoire `.venv`.

Sous `sh`, `bash` ou `zsh` :

```
source .venv/bin/activate
```

Sous Windows, avec le PowerShell

```
.venv/bin/Activate.ps1
```

Sous Windows, avec le `cmd.exe`

```
.venv\Scripts\activate.bat
```

2.1.3 Désactiver un environnement virtuel

Si vous avez besoin de sortir de l'environnement virtuel, c'est à dire de le désactiver dans le terminal, l'instruction est :

```
deactivate
```

2.1.4 Supprimer un environnement virtuel

Supprimer un environnement virtuel consiste à supprimer le répertoire qui le contient.

2.1.5 Les environnements virtuels et les gestionnaires de version

Le répertoire d'un environnement virtuel **n'est pas sous contrôle de version**. Ces environnement sont volumineux et peuvent contenir des packages spécifiques à votre système (OS ou architecture). Les environnements virtuels se reconstruisent.

2.1.6 Les environnements virtuels et les IDEs

Les principaux IDEs (PyCharm et Visual Studio Code) on des outils plus *graphiques* de gestion des environnements virtuels. Il ne s'agit cependant que d'une interface sur les instructions présentées ici.

2.2 Gérer des dépendances de manière standard

Les dépendances se gèrent de manière standard avec `pip`.

2.2.1 Installer une dépendance

Pour installer une dépendance :

```
pip install pytest
```

Vous pouvez installer plusieurs dépendances en une instruction

```
pip install pytest pylint ruff
```

2.2.2 Mettre à jour une dépendance

```
pip install --upgrade pytest
```

Pour information, `pip` se met à jour par lui même :

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

2.2.3 Supprimer une dépendance

```
pip uninstall pytest
```

2.2.4 Lister les dépendances

```
pip list
```

2.2.5 Gérer les dépendances d'un projet, approche historique

Historiquement, les dépendances d'un projet sont déclarées dans un fichier `requirements.txt`, ce qui facilite leur installation. Ce fichier doit suivre une certaine syntaxe. Le plus simple est de le générer, à l'aide de l'instruction :

```
pip freeze
```

Cette instruction affichera dans le terminal les bibliothèques installées avec la syntaxe suivante :

```
flake8==5.0.4
ipython==8.11.0
pylint==2.15.5
pytest==7.0.1
```

Pour créer le fichier, il suffit de rediriger la sortie standard vers le fichier :

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

Pour installer le contenu du fichier :

```
pip install -r requirements.txt
```

Il est également possible d'utiliser ce fichier pour mettre à jour des dépendances :

```
pip install --upgrade -r requirements.txt
```

2.3 L'approche packaging standard

Un programme Python, dans sa forme la plus simple, est une arborescence de fichiers. Python prévoit également la possibilité de *packager* le code pour en faite un *package*. C'est exactement le type de fichiers que vous installez avec `pip instal`. Mais créer un projet destiné au packaging a aussi un intérêt dans le cadre du développement.

Un projet destiné au packaging a une structure particulière (*src-based layout*) et est décrit dans un fichier `pyproject.toml`.

Il y a plusieurs intérêts pour le développement.

2.3.1 Installation du projet en mode éditable

```
pip install -e .
```

Le projet actuel n'est pas installé, un lien est créé dans la bibliothèque des packages vers l'arborescence. Ceci permet de ne plus avoir de problème de chemin vers les ressources.

2.3.2 Une gestion centralisée des dépendances

Lorsqu'un projet est destiné au packaging, la création d'un package ne regarde pas le fichier `requirements.txt`. Les dépendances sont décrites dans le `pyproject.toml`.

Depuis `pip 26.2`, il est possible d'utiliser le fichier `pyproject.toml` et se passer du `requirements.txt` :

```
pip install . --only-deps
```

En général, dans un `pyproject.toml`, les dépendances ne sont que les dépendances *de prod*. Les dépendances de développement sont un *groupe* habituellement nommé `dev` et que l'on peut installer par :

```
pip install --group dev
```

Deuxième partie

Les fondamentaux

3 Données et variables

La programmation consiste à manipuler des données. Pour accéder aux données, il est fait usage de *variables*. Pour le programmeur, une variable est un nom qui doit identifier la donnée manipulée.

3.1 Les données

Une donnée représente une information. Elle a un type et une valeur.

— Le **type** définit la nature des valeurs que peut prendre la donnée ainsi que les opérateurs qui peuvent lui être appliqués et leur effet.
— La **valeur** représente l'information.

Les données sont manipulées avec des instructions qui peuvent également être des symboles ou des fonctions.

3.2 Les variables

Une variable est **un nom** avec lequel on va manipuler une donnée. Une variable est créée par affectation.

```
1 >>> question = 'Meaning of life'  ①
2 >>> answer = 42                  ②
```

① Création de la variable `question` référençant la donnée `'Meaning of life'`
② Création de la variable `answer` référençant la donnée `42`

En Python, les variables **référencent** des données. Une variables ne contient pas la valeur mais une référence vers une valeur.

Python est un langage **non déclaratif** : il n'y a pas de déclaration du type des données référencées par les variables. Une variable peut donc référencer des données de type différents au cours de son existence. **Le type est déterminé dynamiquement** par l'interpréteur.

Python est un langage de haut niveau qui va gérer la mémoire par un **mécanisme de comptage de références**. Le développeur n'a pas à se soucier de la gestion de la mémoire. Les données seront détruites lorsqu'elles ne seront plus accessibles, c'est à dire lorsqu'elles ne seront plus référencées.

```
>>> answer = 42                                                    ①
>>> print(type(answer))
<class 'int'>
>>> del answer                                                     ②
```

① Création de la variable par affectation.
② Destruction de la variable et en conséquence de la valeur.

Une instruction peut se limiter au nom d'une variable. Il s'agit alors d'une instruction qui retourne la donnée référencée par la variable.

```
>>> answer = 42
>>> answer
42
```

Une variable dans le code doit donc systématiquement être considérée comme la valeur qu'elle référence.

3.3 Règles de nommage

Les variables en Python peuvent contenir des caractères alphanumérique sans diacritique (tel que les accents, cédille…) ou souligné `_` (underscore). Le premier caractère ne peut pas être un caractère numérique (de 0 à 9). Python est sensible à la casse (capitale ou minuscule).

L'expression rationnelle représentant un nom de variable Python peut s'écrire : `[a-zA-Z_][a-zA-Z0-9_]`

En plus de la règle de nommage, vous ne pouvez pas utiliser les mots réservés.

3.4 Conventions de codage : la PEP 8

Tout développement nécessite de s'accorder sur des *conventions de codage* afin de rendre l'ensemble du code lisible et maintenance. Dans l'univers Python, ces règles et conventions sont généralisés dans la **PEP 8**. Une **PEP**, pour **Python Enhancement Proposal**, est un document destiné à proposer une évolution du langage puis, si accepté, à documenter cette évolution. L'explication plus détaillée est publiée dans la PEP 1 et la liste de toutes les PEPs dans la PEP 0.

La PEP 8 est le document formalisant les règles de codage adoptées par la communauté. La PEP 8 propose des règles à props des variables. Elle ne concerne que la question du formatage et tient en 3 cas (plus un lié à la lisibilité) :

— Évitez les caractères l, O ou I comme nom de variable à une lettre (en général des indices) car en fonction des polices, ils ne sont pas lisibles et peuvent être confondus avec les valeurs 0 et 1.

— snake_case : lettres minuscules pour les noms de modules, variables et fonctions. Pour représenter plusieurs mots, les séparer par le caractère souligné (exemple : ma_variable).
— Lettres capitales et soulignés pour les constantes (exemple : MA_CONSTANTE)
— CamelCase (ou CapWords) pour les noms de classes (exemple : MaClasse)

Principes et conventions

Au delà des règles de formatage des noms de variables, il y a quelques principes et conventions.

> Explicit is better than implicit

Python étant un langage non déclaratif, le nom des variables doit expliquer ce qu'elles contiennent. Cependant, ne laissez pas supposer du type de la donnée puisque le type peut changer.

3.5 Affecter une donnée à une variable

Le symbole *égal* = est **l'opérateur d'affectation** et permet donc d'affecter une donnée à une variable.

```
1 >>> answer = 42
```

3.6 Absence de déclaration du type

Python est un langage non-déclaratif, c'est à dire que lors de la création d'une variable, on ne déclare pas le type de donnée qui sera référencé par cette variable. La conséquence est que le type de la variable référencée par l'interpréteur pourra varier. Le type de la donnée sera défini par l'initialisation de la donnée (voir les différents types).

3.7 Manipuler et comprendre des données

Python propose plusieurs outils permettant de comprendre les variables, donc les données. Soit une variable ma_variable :

— `print(ma_variable)` affiche la valeur référencée par la variable
— `print(type(ma_variable))` affiche le type de la valeur référencée par la variable
— `id(ma_variable)` affiche l'identifiant unique de la donnée référencée par la variable
— `dir(ma_variable)` retourne la liste des attributs du type de la donnée référencée par la variable.
— `help(ma_variable)` affiche *l'aide* sur le type de la données référencée par la variable.

4 Les types de base (built-in)

Python propose plusieurs types, ou plutôt familles de types, de base (built-in).

Les scalaires sont représentés par les types numériques.

Les types de données qui représentent plusieurs données sont les collections.

4.1 Les types numériques

En Python, nous avons 3 types numériques :

- `int` qui représente les entiers
- `float` pour les réels
- `complex` pour les nombres complexe.

Le type booléens (`bool`) représente les notions de vrai et faux et prennent deux valeurs : `True` et `False`.

De manière générale en Python, le type de la donnée est déduite par l'interpréteur. Les types *built-in* ont la particularité d'être déclarés par leur valeur. Néanmoins, il existe aussi des fonctions pour initialiser ces données. Ces fonctions peuvent également convertir un type dans le type correspondant au nom de la fonction : `int(var)`, `float(var)` ou `complex(var1, var2)`.

Ci-dessous, vous avez un exemple de déclaration de types numérique

```
1 answer = 42                                                    ①
2 answer = 42.0                                                  ②
```

① Déclaration d'un entier (`int`)
② Déclaration d'un réel (`Float`)

```
1 In [10]: var_int = int(42.2)
2 In [11]: var_int = 42
3
4 In [12]: var_float = float(3)
5 In [13]: var_float = 3.
6 In [14]: var_float = 3.14
7
8 In [15]: var_cpx = complex(42, 2)
```

```
In [16]: var_cpx = 42 + 2j
In [17]: var_cpx.real
Out[17]: 42.0
In [18]: var_cpx.imag
Out[18]: 2.0
```

En Python, il n'y a pas de taille maximum des entiers. La curiosité est que si sa valeur est comprise entre `-2^(n-1)` et `2^(n-1)-1`, il est géré en registre, sinon en mémoire. Les réels ont une taille correspondant à un double en C.

4.2 Opérateurs arithmétiques

Python propose les opérateurs arithmétiques classiques ainsi que quelques autres.

x + y	Addition
x - y	Soustraction
x * y	Multiplication
x / y	Division
x // y	Division entière
x % y	Reste
-x	Opposé
+x	Identité
x ** y	Puissance

La priorité des opérateurs est la priorité mathématique. À priorité égale, les opérations sont résolues de gauche à droite. Les parenthèses ont la plus grande priorité et permettent de grouper les opérations.

Ces opérateurs sont utilisés entre les types numériques en tant qu'opérateurs mathématiques mais certains peuvent être utilisés avec d'autres types. Ainsi, l'addition de deux chaines de caractères (voir partie sur les collections) produira une chaine de caractères correspondant à la concaténation de ces deux chaines.

4.3 Les opérateurs *bitwise*

Aussi appelés opérateurs bits à bits, ils sont souvent à tord appelés *opérateurs binaires*.

Python propose les opérateurs suivants :

x | y	Ou binaire
x ^ y	Ou exclusif
x & y	Et binaire

x « y	Décalage à gauche
x » y	Décalage à droite
~ x	Inversion

Ces opérateurs permettent de manipuler la représentation binaire des valeurs, par exemple :

```
In [1]: x = 5

In [2]: y = 6

In [3]: res = x | y

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

5 en binaire est `0b101`

6 en binaire est `0b110`

Donc `0b101` OU BINAIRE `0b110` donne `0b111` soit `7`.

Leur intérêt se retrouve surtout dans les opérations usuelles comme dans l'exemple suivant, déterminer si un entier est une puissance de deux (dans cet exemple, 0 est également considéré comme une puissance de 2) :

```
In [5]: (value & (value - 1)) == 0
```

4.4 Les collections

Les collections représentent des types qui regroupent une **collection** de données.

Les collections qui contiennent des éléments ordonnées sont des **séquences**. Python propose trois types pour les séquences :

— les chaines de caractères (`str`)
— les listes (`list`)
— les N-uplets (`tuple`)

Les collections qui ne sont pas des séquences sont :

— les ensembles (`set`)
— les dictionnaires (`dict`)

Les collections sont des objets. En plus des opérateurs communs, chaque type a ses méthodes spécifiques qui permettent des actions spécifiques. Souvenez-vous que vous pouvez consulter la documentation de chaque type avec la fonction `help()`.

Toutes les collections peuvent être initialisées par une fonction du nom du type qui, sans paramètre, retournera une collection vide.

4.4.1 Instructions communes

Les différents types de collection, soit `str`, `list`, `tuple` et `dict`, ont des comportements communs. Les instructions suivantes sont communes à ces différents types. Ainsi, soit `my_collection` une collection d'un de ces types, `x` une valeur et `n` un entier (`int`).

- `x in my_collection` : retourne `True` si `my_collection` contient `x`, sinon `False`
- `x not in my_collection` : retourne `False` si `my_collection` contient `x`, sinon `True`
- `my_collection1 + my_collection2` : Retourne une concaténation les deux collections du même type.
- `my_collection * n` : où `n` est un entier, retourne une nouvelle collection multiple de l'originale
- `len(my_collection)` : retourne la taille de la collection

4.4.2 Les séquences : accès aux éléments

Concerne : `str`, `list`, `tuple`

Les éléments d'une séquence sont identifiés par leur position. La position d'un élément est représentée par son indice qui est la position en commençant à 0.

Pour accéder à un élément d'une séquence, on utilise son indice entre crochet. Python utilise les indices négatifs afin d'accéder aux éléments à partir de la fin.

```
In [30]: knights = ["Arthur", "Lancelot", "Robin", "Bedevere", "Galaad"]
In [31]: antagonist = 'Killer Rabbit of Caerbannog'

In [32]: print(knights[2])
Robin

In [33]: print(antagonist[2])
l

In [34]: print(knights[-1])
Galaad

In [35]: print(antagonist[-5])
a
```

Ceci permet d'illustrer un des adages de Python : "There should be one– and preferably only one –obvious way to do it" ou « Il devrait y avoir une, et de préférence une seule, manière évidente de faire une chose ». Ainsi, une demande du type « quel est le dernier élément de la liste » se traduit par `ma_liste[-1]`.

4.4.3 Modification d'un élément

Concerne : `list`

Un élément d'une liste peut être remplacé en attribuant une valeur à un indice existant. La valeur référencée par cet indice est alors remplacée par la nouvelle.

```
In [50]: knights = ["Arthur", "Robin", "Bedevere", "Galaad"]

In [51]: knights[1] = 'Lancelot'
In [52]: print(knights)
['Arthur', 'Lancelot', 'Bedevere', 'Galaad']
```

Les tuples et les chaines de caractères sont des types immuables. Il est impossible de replacer un de leurs éléments.

4.5 Slicing

Concerne : `str`, `list`, `tuple`

Le slicing permet d'extraire une *sous-séquence*. Cette opérateur va retourner une nouvelle séquence du même type avec les éléments correspondants à la plage indiquée.

4.5.1 Usage

L'opérateur de slicing ainsi que ses différents usages sont les suivants (seq étant une séquence) :

- `seq[x:y]` : extrait une séquence de l'indice `x` à l'indice `y` exclu
- `seq[:y]` : extrait une séquence de l'indice `0` à l'indice `y` exclu
- `seq[x:]` : extrait une séquence de l'indice `x` à la fin de la séquence `seq`
- `seq[x:y:z]` : extrait une séquence de l'indice `x` à l'indice `y` exclu avec un pas de `z`.

Les valeurs `x`, `y` et `z` sont toutes optionnelles, `seq[:]` retournera ainsi une copie de la séquence.

Si les bornes sont en dehors des limites de la séquence, le slicing retournera une séquence vide.

```
In [2]: knights = ["Arthur", "Lancelot", "Robin", "Bedevere", "Galaad"]

In [3]: knights[2:4]
Out[3]: ['Robin', 'Bedevere']

In [4]: # Récupérons les deux premiers éléments

In [5]: knights[:2]
Out[5]: ['Arthur', 'Lancelot']

In [6]: # Récupérons les éléments à partir du troisième

In [7]: knights[2:]
Out[7]: ['Robin', 'Bedevere', 'Galaad']

In [8]: # Récupérons les deux derniers éléments avec les indices négatifs

In [9]: knights[-2:]
Out[9]: ['Bedevere', 'Galaad']

In [10]: antagonist = 'Killer Rabbit of Caerbannog'

In [11]: antagonist[::2]
Out[11]: 'Kle abto arang'
```

4.5.2 Mais pourquoi cette exclusion ?

La position des éléments dans une séquence en Python est représenté par leur indice, le premier élément est à l'indice 0. On parle de 0-based indexing. ABC, Algol, Fortran ou Pascal utilisaient le 1-based indexing.

Le slicing permet d'obtenir selon la définition mathématique, un intervalle semi-ouvert à droite. Cela se noterait `[x, y[`. Pourquoi ce choix d'implémentation ?

Guido Van Rossum s'est exprimé sur ce choix. Il faut savoir que c'est le slicing qui a conduit au choix entre une indexation 0-based ou 1-based. Ce choix repose sur les cas d'usage les plus courants :

— Obtenir les n premiers éléments
— Obtenir les n éléments à partir de l'indice i
— Obtenir des tranches adjacentes à l'aide d'un invariant.

Ces cas sont illustrés ci-dessous

```
In [12]: n = 2

In [13]: i = 1

In [14]: knights[:2]  # Les deux premiers
Out[14]: ['Arthur', 'Lancelot']

In [15]: knights[i: i + n]  # Les deux après l'indice i
Out[15]: ['Lancelot', 'Robin']

In [16]: # Les deux premiers et les suivants

In [17]: knights[:2]
Out[17]: ['Arthur', 'Lancelot']

In [18]: knights[2:]
Out[18]: ['Robin', 'Bedevere', 'Galaad']
```

5 Les structures de contrôle

Les *structures de contrôle* servent à **structurer** l'exécution du code. Elles permettent de définir des **blocs** qui vont être exécutés de manière **conditionnelle** ou **en boucle**.

Le bloc est un ensemble d'instructions qui est précédé d'un **en tête**. Un en tête est une ligne qui commence par un mot clef et se termine par deux points (`:`).

En Python, il n'y a pas de délimiteur pour le bloc d'instructions comme les accolades. Un bloc est défini par l'**indentation** par rapport à la déclaration. Cette indentation est un retrait de 4 espaces (PEP8).

5.1 Les structures

5.1.1 Les blocs

Les blocs sont donc un ensemble d'instructions exécutées en fonction de l'en tête qui le précède. Ces instructions sont indentées par rapport à la déclaration, c'est à dire qu'ils sont en retrait. Ce retrait est défini par la pep 8 comme étant de 4 espaces.

Veillez donc à vous assurer que votre éditeur de code convertisse les tabulations en 4 espaces.

5.1.2 L'instruction `pass`

L'instruction `pass` est une instruction que ne fait… rien. Après une déclaration de bloc (que ce soit une structure de contrôle, une déclaration de fonction ou de classe), l'interpréteur attends une ligne indentée. Si vous n'avez aucune instruction à déclarer dans le bloc (voir les cas lors de la formation), l'instruction `pass` permet de ne pas avoir d'erreur d'interprétation.

```
1 if True:
2     pass
```

5.1.3 Exécution conditionnelle avec `if`

L'instruction `if` permet d'exécuter un bloc d'instructions de manière conditionnelle.

```
1 if knight == "Arthur":
2     print("Is the King")
```

Une ou plusieurs conditions alternatives peuvent être déclarées avec l'instruction `elif`. L'instruction `else` permet de gérer tous les autres cas.

```
1 if knight == "Arthur":
2     print("Is the King")
3  elif knight == "Robin":
4     print("He's not that brave")
5  else:
6     print("Who's that lad ?")
```

Les différentes conditions sont exclusives. À la première condition évaluée à vrai (`True`), l'interpréteur exécute le bloc puis sort de la structure conditionnelle.

La condition, aussi appelée structure de contrôle, doit retourner une expression booléenne. Cette expression peut aussi bien être le retour d'une fonction que le résultat d'un opérateur de comparaison ou d'un opérateur logique.

5.1.4 Les opérateurs de comparaison

Les opérateurs de comparaisons entre deux déclarations sont :

<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Python permet également l'évaluation de l'identité par le mot réservé is. Attention, is doit être utilisé avec des valeurs comme `None`, `True` ou `False` car il s'agit de comparer avec ces types et non des valeurs.

5.1.5 Les valeurs booléennes

N'importe quel objet en Python peut être évalué à *vrai* ou *faux*. Le type booléen (`bool`) est le seul à représenter cette notion, mais pour tous les autres types, Python peut reconnaitre que certaines valeurs auront le sens de *faux*. Dans le cas des types de base, voici le comportement :

Type	**FAUX**
Bool	False
int	0
float	0.0
string	" "
tuple	()
list	[]
dict	{}
None Type	None

Ainsi en Python, il n'est pas nécessaire d'écrire certaines comparaisons.

5.1.6 Les opérateurs logiques

Les opérateurs logiques, nom utilisé pour les valeurs de vérité de l'algèbre de Bool, sont la conjonction (et), la disjonction (ou) et la négation (non). En Python, les mots-clef respectifs sont `and`, `or` et `not`.

Soient X et Y deux expressions qui retournent une valeur booléenne.

- Conjonction : X ET Y est vrai si et seulement si X est vrai et Y est vrai
  En python, elle s'écrit `X and Y`
- Disjonction : X OU Y est vrai si et seulement si X est vrai ou Y est vrai
  En python, elle s'écrit `X or Y`
- Négation : la négation de X est vrai si et seulement si X est faux
  En python, elle s'écrit `not X`

En python, les fonctions de conjonction et de disjonction sont des opérations *paresseuses*. Si l'évaluation du premier terme ne laisse aucun doute sur le résultat de l'opération (si X est faux pour une conjonction ou X est vrai pour une disjonction), le second terne n'est pas évalué. Gardez ceci à l'esprit si les expressions sont des appels de fonctions.

5.1.7 Chainage des opérateurs de comparaison

Au lieu d'utiliser l'opérateur logique and, il est possible dans certains cas de chainer les opérateurs.

```
In [1]: x = 10

In [2]: x > 0 and x < 10
Out[2]: False

In [3]: 0 < x < 10
Out[3]: False
```

Ceci permet une syntaxe similaire à ce que nous écrivons.

5.1.8 Comparaison de tuples

Dans le même ordre d'idée, les tuples pouvant être comparés, la comparaison est similaire à l'opérateur logique `and`.

```
In [8]: x1 == y1 and x2 == y2
Out[8]: True

In [9]: (x1, x2) == (y1, y2)
Out[9]: True
```

6 Les fonctions

6.1 Structure

```
def nom_fonction(param1, param2):
    do_somerhing()
    return value

```

6.2 Retours

Une fonction retourne

- `return` est optionnel
- `return` avec ou sans paramètre interrompt l'exécution de la fonction
- `return value` retourne la valeur associée à `value`. La fonction *retourne* cette valeur.
- L'absence de `return` entraine un `return None` implicite
- Un `return` sans paramètre correspond à un `return none`

6.3 Paramètres

6.3.1 Paramètres positionnels

```
def func(param1, param2):
    pass
```

```
func('val1', 'val 2')
```

6.3.2 Paramètres optionnels

```
def func(param="value"):
    pass
```

```
func()
func('val')
```

Les paramètres optionnels sont en dernier (un paramètre positionnel ne peut suivre un paramètre optionnel).

```
def func(param_pos, param_opt="value"):
    pass
```

```
func('val opsitionnelle')
func('val opsitionnelle', 'val optionnelle')
```

6.3.3 Arguments nommés

```
def func(param_pos, param_opt="value"):
    pass
```

```
func('val opsitionnelle', param_opt='val optionnelle')
```

6.4 Utilisation des type hints

```
def func(param_pos:str, param_opt:int = 1) -> str:
    return param_pos * param_opt
```

7 Les séquences

Une séquence est une collection d'éléments ordonnés.

Parmi les types de bases, les séquences sont :

- `str` : chaines de caractères, sont des séquences immuables de chaines d'un seul caractère
- `list` : listes, sont des séquences d'objets
- `tuple` : n-uplets, sont des séquences immuables d'objets

7.1 Création

7.1.1 Créer une chaine de caractères

```
1 my_string = ''
2 my_string = ""
3 my_string = str()
4
5 king = "Arthur"
```

7.1.2 Créer une liste

```
1 my_list = []
2 my_list = list()
3
4 camelot = ['Arthur', 'Merlin', 'Lancelot',
5            'Robin', 'Bohort', 'Perceval', 'Karradoc']
```

7.1.3 Créer un n-uplet (*tuple*)

```
1 my_tuple = tuple() # tuple vide
2
3 wizards = 'Merlin',
4 tavern = 'Perceval', 'Karradoc',
```

7.2 Taille

Avec la fonction `len(seq)`

```
>>> len(king)
6
>>> len(camelot)
7
>>> len(wizards)
1
```

7.3 Accès à un élément

Par indexation (*subscript*)

```
>>> king[2]
't'
>>> camelot[1]
'Merlin'
>>> tavern[0]
'Perceval'
```

```
>>> king[-1]
'r'
>>> camelot[-2]
'Perceval'
>>> tavern[-1]
'Karradoc'
```

7.4 Test élément dans séquence

Avec le mot-clef `in`

```
>>> 't' in king
True
>>> 'Mordred' in camelot
False
>>> 'Tavernier' in tavern
False
```

7.5 Unpacking

Fonctionne avec toute séquence

```
>>> quest = ['Rabbit of Caerbannog', 'cave',
...          'Location of the Holly Grail']
>>> opponent_name, location, reward = quest
>>> opponent_name
'Rabbit of Caerbannog'
>>> location
'cave'
>>> reward
'Location of the Holly Grail'
```

7.6 Slicing

Structure : ‘seq[start:stop:step]

```
>>> king[2:5]
'thu'
>>> camelot[2:6]
['Lancelot', 'Robin', 'Bohort', 'Perceval']
```

```
>>> camelot[:2]
['Arthur', 'Merlin']
>>> camelot[3:]
['Robin', 'Bohort', 'Perceval', 'Karradoc']
>>> camelot[-2:]
['Perceval', 'Karradoc']
```

Troisième partie

La Programmation Orientée Objet

8 Concepts de la Programmation Orientée Objet

8.1 Les paradigmes de programmation

Il s'agit des différentes façons de raisonner et d'implémenter une solution à un problème en programmation. Nous pouvons citer trois grandes familles :

- La programmation impérative
  paradigme originel et le plus courant
- La programmation orientée objet (POO)
  consistant en la définition et l'assemblage de briques logiciel appelées objets
- La programmation déclarative
  consistant à déclarer les données du problème, puis à demander au programme de le résoudre

8.2 Le concept d'objet

Le problème à traiter va être modélisé par des objets. Un objet est une entité caractérisée par :

- Un état, représenté par des données spécifiques à l'objet que nous appellerons des *attributs*.
- Des comportements, représentés par des fonctions spécifiques au type d'objet que nous appelleront des *méthodes*.

Un objet est donc un composant qui regroupe au sein d'une même entité des données et des traitements.

Vous avez déjà manipulé des objets. Si nous prenons les listes, l'état correspond aux données de la liste et les comportements sont les méthodes comme sort, split, append…

Un objet sera un composant du traitement de notre problème et plusieurs objets devront interagir pour traiter notre problème. Un principe pour la conception des objets est que **chaque objet devra avoir une seule responsabilité**. Par exemple, un objet qui gérera un compte ne s'occupera pas de son affichage.

8.3 La notion de classe

En programmation orientée objet, nous utilisons des objets. La structure de ces objets est définie dans des classes. Une classe peut être vue comme le moule qui permet de produire les objets. Dans le vocabulaire orienté objet, les objets sont des instances d'une classe. Les objets étant en soi des données, ils ont un type qui est leur classe.

Bien qu'un objet soit l'instance d'une classe, nous allons voir qu'il s'agit de deux composants différents dans le mécanisme de la programmation orientée objet.

8.4 Représentation : présentation d'UML

Afin de pouvoir représenter les notions de classes et objets, nous allons utiliser UML, Unified Modeling Language. UML est un langage de modélisation graphique basé sur des diagrammes et des pictogrammes. Il est standardisé et se veut être une synthèse des méthodes de modélisation. Pour cela, en version 2.5, il propose 14 type de diagrammes qui vont bien au delà de la modélisation objet.

8.4.1 Représenter une classe

La classe la plus basique peut être représentée par le diagramme ci-dessous. Cette classe n'a ni état (pas d'attributs) ni de comportement (pas de méthode). Notez que le nom de la classe correspond à la règle de nommage de la PEP8 et est donc en CamelCase.



Si cette classe n'est pas censée exister d'un point de vue paradigme, elle nous permet de voir le code qui permet de déclarer une classe dans la partie suivante.

8.4.2 Représenter les attributs et méthodes

Pour illustrer le concept de l'orienté objet, nous utiliserons un exemple de suivi de formations. Nous allons commencer par définir ce qu'est une formation et les actions sur celle-ci.

L'*état* d'une formation est représentée par son nom, sa durée, le nombre de places et la liste des stagiaires (pour l'instant représentés par leur nom). Une formation a donc les attributs `subject`, `duration`, `students` et `available_seats`.

À une formation, nous pouvons ajouter ou supprimer des participants, nous aurons donc des méthodes pour cela.

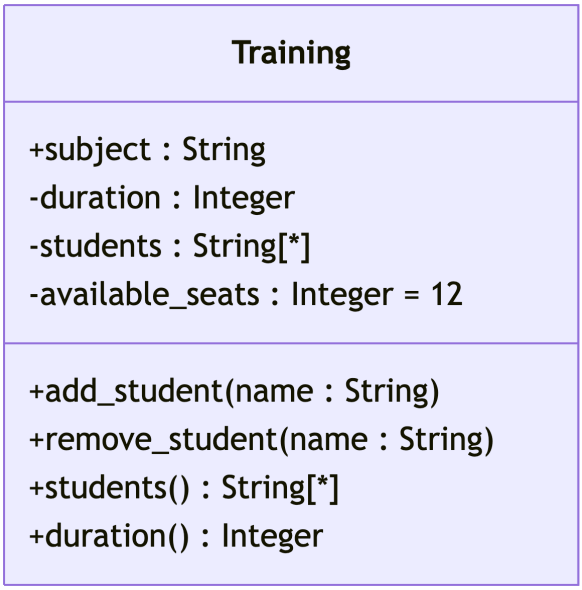
UML permet de représenter dans un diagramme de classe ses attributs et méthodes. Le diagramme ci-dessous représente cette notion de formation.

Une classe est représentée en 3 blocs. Le premier est le nom de la classe, le second est l'ensemble des attributs de la classe et la troisième les méthodes.

Pour les attributs, il est possible de spécifier le type et la valeur par défaut.

Les méthodes spécifient les paramètres attendus entre parenthèses ainsi que le type du retour.

Attributs et méthodes sont précédés d'un symbole qui représente leur visibilité.



— `Training` est le nom de la classe.
— `title`, `duration`, `students` et `seats` sont des attributs.
— `add_student(name)`, `remove_student(name)` et `students()` sont des méthodes.

8.4.3 Notion de visibilité des attributs et méthodes

La visibilité est un concept important de la conception objet puisqu'elle définit *qui* pourra accéder à l'attribut ou à la méthode et permet de mettre en œuvre le principe de l'encapsulation. Accéder à un attribut signifie avoir la possibilité de consulter la valeur mais aussi la modifier. UML permet de représenter 4 visibilité :

— **Public** représenté par le symbole `+`. Un attribut ou une méthode publique pourra être accédé par n'importe quel composant.
— **Package** représenté par le symbole `~`. Un attribut ou une méthode package pourra être accédé par n'importe quel composant déclaré dans le même package.

— **Protected** représenté par le symbole `#`. Un attribut ou une méthode protégée ne pourra être accédé que par le composant qui l'a déclaré ou par tout composant spécialisé (voir notions d'héritage)
— **Private** représenté par le symbole `-`. Un attribut ou une méthode privée ne pourra être accédé que par le composant qui l'a déclaré.

Dans notre exemple, la durée, la liste des participants et le nombre de places disponibles sont des attributs privés car ils ne doivent pas être modifiés par un composant extérieur. La liste de participants évoluera par appel aux méthodes d'ajout et suppression de participants. Des méthodes permettent également de consulter ces attributs privés (`students()` et `duration()`).

Vous avez remarqué que ce modèle ne permet pas en l'état de consulter le nombre de places disponibles.

8.4.4 Cardinalité et contraintes

La liste des stagiaires est définie comme pouvant contenir de 0 à un nombre indéfini de stagiaires. Mais une formation a une limite de places. Cette limite dépend de la formation. Il est donc impossible de l'imposer dans le modèle, ce qui nous conduit à cette cardinalité de 0 ou plusieurs.

UML ne permet pas de modéliser les contraintes, celles-ci sont modélisées par un autre langage, OCL (Object Constraint Language). Cette contrainte doit être appliquée à notre modèle et c'est pourquoi l'ajout de participants doit passer par une méthode. Celle-ci connait la règle métier qui sera d'ajouter un stagiaire uniquement si le nombre de participants est inférieur au nombre de places.

8.4.5 La notion de relations

Lorsque nous définissons des objets, ceux-ci peuvent être liés les uns aux autres. Nous parlons de relations.

Les relations permettent de représenter les associations entre les classes.

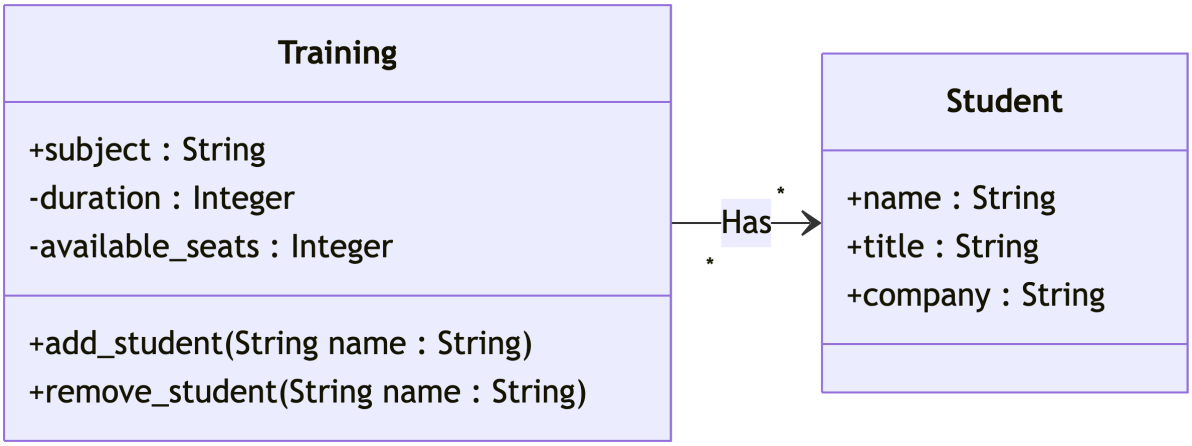
Nous pouvons compléter notre modèle de formation. Un stagiaire n'est pas limité à une chaine de caractères, nous pouvons définir une classe pour le représenter. Il faudra ensuite l'associer à la formation.

8.4.6 Relation simple

En UML, la relation est symbolisée par un trait, en général horizontal, reliant les deux classes. Une flèche permet d'orienter la relation. Cette dernière est importante car elle détermine où (dans quelle classe) sera exprimée la relation.

UML nous permet également de préciser la cardinalité de cette relation. Les indications possibles sont :

— `n` où n est un nombre signifiant exactement n instances
— `*` signifie un nombre indéterminé d'instances
— `m..n` où m est un entier et n est soit un entier soit `*` et signifie de `m` à `n` instances.



Dans notre modèle, une formation est dispensée à plusieurs stagiaires et un même stagiaire peut suivre plusieurs formations.

La flèche ainsi que la définition de la relation indiquée au dessus de cette flèche nous renseigne sur l'implémentation attendue. Ainsi, une formation possède des stagiaires. L'implémentation devra donc définir une liste de stagiaires dans la classe `Training`. Vous remarquerez qu'il n'y a pas d'attribut `students:Student[*]` déclarée dans ce dernier schéma, contrairement au premier. C'est parce cette relation implique implicitement cet attribut.

8.5 UML : les limites de la représentation graphique

Nous avons vu que nous pouvons représenter beaucoup d'informations sous forme de diagrammes en UML. Mais la représentation graphique a des limites. Dans le cas de nos formations, techniquement, la liste des stagiaires n'a pas de limite supérieure. Cependant, cette limite est définie par un autre paramètre qui, en soi, est une variable.

Il est donc impossible de représenter cette contrainte de manière graphique et nous devrions exprimer certaines contraintes en langage naturel.

Cependant, il existe depuis la version 1.1 d'UML un langage formel, OCL (Object Constraint Language). Le langage naturel est évidemment le plus facile à mettre en œuvre, à formuler et à comprendre. Mais il est également source d'ambiguïtés qu'OCL essaye d'éviter.

Cette formation n'est pas une formation UML/OCL, ce sujet ne sera pas abordé ici.

9 La Programmation Orientée Objet en Python

9.1 Déclarer une classe

En Python, une classe est déclarée par la directive `class`. Elle est suivie du nom de la classe puis de deux points. Le corps de la classe doit évidemment être indenté.

Une classe est un *callable*. Un appel à cette classe retournera une instance de cette classe et pour maintenir une référence à cette instance, nous allons l'affecter a une variable.

Le code suivant est le code minimum permettant de déclarer et d'instancier une classe. La variable `myInstance` référence alors une donnée de type `MyClass`.

```
1 class MyClass:
2     pass
3
4 myInstance = MyClass()
```

Les instances de cette classe sont des objets sans état et sans comportement. Ils ne servent pas à grand chose mais nous avons tout de même créé un nouveau type dans notre système.

9.2 Les méthodes

En Python, les méthodes sont des fonctions. Il s'agit donc des traitements. Lors de la conception d'un objet, les méthodes doivent être caractérisées par des verbes et représentent des actions.

Les méthodes peuvent recevoir des paramètres et retourner une donnée. Deux caractéristiques vont faire qu'une fonction est une méthodes :

- Les méthodes sont définies dans les classes.
- Vous devez déclarer un premier paramètre nommé par convention `self`. Ce paramètre est destiné à recevoir l'instance en cours. Vous pouvez déclarer d'autres paramètres en plus de ce paramètre `self`.

Pour appeler une méthode, il faut la préfixer par le nom de l'objet à laquelle elle appartient en ignorant le paramètre attendu `self`.

```python
class MyClass:
    def doSomething(self):
        print("I did something")

    def doSomethingWith(self, value):
        print("I will do something with {}".format(value))


myObject = MyClass()

myObject.doSomething()                                    ①
myObject.doSomethingWith('a value').                      ②
```

① Affiche `I did something`
② Affiche `I will do something with a value`

9.2.1 appels

```python
my_object = MyClass()                                     ①
my_object.my_method_without_param()                       ②
my_object.my_method_with_param(42)                        ③
```

① Instanciation de l'objet
② Appel d'une méthode sans arguments
③ Appel d'une méthode avec arguments

9.2.2 Les attributs

Un attribut représente une donnée propre à un objet. En soi, c'est une variable qui référence donc une donnée. Lors de la conception d'un objet, les attributs sont caractérisés par des noms.

Pour distinguer les attributs des variables au sein d'un objet (dans le code des méthodes de la classe), un attribut est préfixé par le namespace `self` (ce n'est pas qu'une convention). Pour accéder à un attribut en dehors de l'objet, il faut le préfixer par la référence de son instance.

```python
class Training:
    def reset_students(self):
        self.students = []

    def get_available_seats(self):
        return self.max_seats - len(self.students)
```

```
7
8
9  training = Training("Python")
10 print(training.students)
```

Dans cet exemple, la liste des participants, `students`, est un attribut public accessible à l'extérieur de l'objet comme le montre la dernière ligne.

9.2.3 En Python, un objet n'a que des attributs

Si vous consultez la documentation de la fonction dir(), vous verrez qu'elle liste tous les attributs d'un objet. Or, dans cette liste, il y a des méthodes.

Rappelons qu'en Python, tout est objet. Une méthode peut donc être considérée comme un attribut qui a la particularité d'être un *callable*, donc de pouvoir être exécuté. Cette propriété permet la mise en œuvre de l'introspection.

9.2.4 Méthode particulière : le constructeur

Pour créer un objet, l'interpréteur va appeler une méthode particulière. Cette méthode porte habituellement le nom de *constructeur*. Le constructeur est appelée à l'instanciation de l'objet et permet de définir son état initial. Elle est disponible pour toutes les classes et par défaut ne définit aucun état.

Python a une particularité : lors de l'instanciation de l'objet, deux méthodes sont appelées successivement :

— La méthode `__new__(self)` qui est le **constructeur** et s'assure de la création technique de l'objet
— La méthode `__init__(self)` qui est un **initialiseur** et qui définit l'état initial de l'objet.

En Python, l'état initial d'un objet et plus généralement tout son comportement fonctionnel à l'initialisation sont définis dans l'initialiseur. Par abus de langage, on parle de constructeur.

Le constructeur de base est le suivant.

```
1 class UselessClass:
2     def __init__(self):
3         pass
```

Le comportement de cet initialiseur est le comportement par défaut, il est inutile d'écrire la fonction `__init__()` de ce type. En son absence, un initialiseur équivalent par défaut est appelé.

Nous pouvons tester le comportement du constructeur par le code suivant.

Nous pouvons voir que le message est affiché à l'instanciation et donc que cette méthode est bien appelée à ce moment.

```
class DemoClass:
    def __init__(self):
        print('I have been initilized')

obj = DemoClass()                                                    (1)
```

(1) Cette expression va créer un objet de type `DemoClass` et le retourner mais auparavant, elle va afficher la chaine de caractères `I have been initilized`.

9.2.5 Méthode particulière : le destructeur

Il existe une autre méthode particulière communément appelée *destructeur*. En fait, c'est une méthode appelée lorsque l'objet perdra sa dernière référence soit juste avant sa destruction.

Le comportement peut être illustré par le code suivant.

```
class DemoClass:
    def __init__(self):
        print('I have been initilized')

    def __del__(self):
        print('I will be destroyed')


c = DemoClass()                                                      (1)

del c                                                                (2)
```

(1) Affichera `I have been initilized` en créant l'objet.
(2) Affichera `I will be destroyed` car la suppression de la variable entraine la suppression de la dernière référence à l'objet donc sa destruction.

L'usage est donc la libération de ressources (c'est le rôle des destructeurs dans les langages bas niveau) et *faire le ménage*.

9.2.6 Utilisation du constructeur

Le constructeur est indispensable pour initialiser un objet, c'est à dire définir son état initial. Évidemment, des paramètres doivent être passés à l'initialisation de l'objet. Ces paramètres sont ceux de l'initialiseur `__init__()`. Ainsi, le code suivant permet d'instancier une formation selon la spécification du diagramme UML.

```python
class Training:
    def __init__(self, subject: str, duration: int, capacity: int = 12):
        self.subject = subject
        self.duration = duration
        self.available_seats = capacity
        self.students = []

```

Vous remarquerez plusieurs points ici.

En premier lieu, les paramètres de l'initialiseur et les attributs n'ont pas à avoir le même nom. En effet, il s'agit de choses très différentes. Les paramètres ne sont que des informations pour construire l'état de l'objet. Dans notre cas, nous utilisons directement les données pour les affecter aux attributs, mais ce ne sera pas toujours le cas. Vous avez cependant une illustration du changement de concept entre l'attente d'information sur la capacité en terme de places de la formation devenant le nombre de places disponibles pour l'inscription.

L'initialiseur étant une fonction, nous pouvons utiliser les même propriétés à savoir l'argument par défaut.

Retenez également qu'il n'y a pas de symétrie entre les paramètres et les attributs. Nous venons de voir en sens, mais aussi en nombre. L'attribut `self.students` est une liste qui sera alimentée au fur et à mesure de l'inscription des stagiaires. Sa création n'attends pas d'information spécifique lors de l'instanciation, elle est donc créée dans dépendance à un paramètre.

l'initialiseur est une méthode appelée à l'instanciation de l'objet et dont le rôle est de s'assurer de l'état initial de l'objet. C'est au sein de cette méthode que vous devez vous assurer de l'intégrité de l'objet. Par principe, mettez-y le tout le code assurant l'intégrité de l'objet. Déclarez également tous les attributs qui font l'objet. Mais assurez-vous aussi de la cohérence de l'état de l'objet. Ainsi, nous pouvons par exemple ajouter la validation suivante (le code se limite au code d'intérêt).

```python
class Training:
    def __init__(self, subject: str, duration: int, capacity: int = 12):
        self.available_seats = int(capacity)

        if self.available_seats < 1:
            raise ValueError(f"Capacity must be positive, got {self.available_seats}")

```

9.2.7 Méthodes et état de l'objet

Nous avons vu plus haut comment déclarer des méthodes. Une méthode *voit* les attribut de l'objet et peut donc les interroger ou les modifier.

C'est donc par les méthodes que l'état d'un objet doit changer. Ainsi, d'après notre modèle, l'objet de type Training doit avoir une méthode `add_student(name)`. Cette méthode ajoutera un stagiaire, dont le nom est passé en paramètre, à la formation. Comme précédemment, je ne m'intéresse qu'au code d'intérêt qui est le suivant.

```
class Training:
    def __init__(self, subject: str, duration: int, capacity: int = 12):
        self.available_seats = int(capacity)
        self.students = []

    def add_student(self, name:str):
        self.students.append(name)

```

L'intérêt de la méthode est d'avoir les plein contrôles sur le changement d'état de l'objet. Ainsi, il est de la responsabilité de cette méthode de s'assurer de l'absence de doublons (non illustré ici) et du respect de la contrainte de places disponibles. La méthode `add_student(name)` devrait donc aller dans le sens suivant :

```
class Training:
    def __init__(self, subject: str, duration: int, capacity: int = 12):
        self.available_seats = int(capacity)
        self.students = []

    def add_student(self, name:str):
        if len(self.students) >= self.available_seats:
            raise ValueError('Training full')

        self.students.append(name)

```

9.3 Visibilité et encapsulation en Python

Jusque'ici, nous ne nous sommes pas soucié de la notion de visibilité. Aucun mot clef ne a déclare. Vous remarquerez que vous accédez librement à tous les attributs et méthodes.

Et pour cause, en Python, tous les attributs et les méthodes **ont une visibilité publique**.

C'est tout à fait voulu par Guido Van Rossum.

> We are all grown up adults here

Il existe cependant une convention forte : lorsqu'un nom d'attribut ou de méthode commence par le caractère souligné, l'underscore, il doit être considéré comme privé. Ainsi :

— `mon_attribut` est un attribut publique

— `_mon_attribut` est un attribut privé.

Pour être plus cohérent, on parlera souvent d'attribut **non-public**.

La visibilité est un concept à associer à celui de l'encapsulation.

9.3.1 Principe d'encapsulation

L'encapsulation est le principe de la programmation orientée objet consistant à masquer l'implémentation et à communiquer avec les interfaces publiques.

Contrairement à des idées reçues, l'objectif est avant tout de maitriser le comportement et l'évolution de l'implémentation. Et surtout de décolérer ce qui est exposé de l'implémentation.

Pour une formation, nous ne voulons pas nous contenter du nombre total de places mais aussi du nombre de places disponibles. La méthode ici peut laisser entendre l'existence d'un attribut mais calcul cette valeur :

```
1 class Training:
2     def __init__(self, subject: str, duration: int, capacity: int = 12):
3         self.available_seats = int(capacity)
4         self.students = []
5 
6     def remaining_seats(self):
7         return self.available_seats - len(self._students)
```

L'encapsulation permet de mettre en œuvre le principe Open/Closed : ouvert à l'extension, fermé à la modification.

9.3.2 Name Mangling

Le Name Mangling est une technique permettant de s'assurer qu'un attribut ne sera pas renommé dans une classe *fille* lors de la mise en œuvre de l'héritage.

En Python, des attributs dont le nom commence par au moins deux underscores et se termine par au plus un underscore basculent sur un mécanisme de résolution qui semble les faire apparaitre comme *réellement* privés. Ce n'est pas le cas…

Le code suivant semble pourtant montrer ce comportement à première vue.

```
1 class Training:
2     def __init__(self, subject: str, duration: int, capacity: int = 12):
3         self.available_seats = int(capacity)
4         self.__students_ = []
5 
```

```
t = Training('Python', 5)
t.__students_                                                    ①
```

① Provoque une `AttributeError`

Oui mais…

```
T._Training__students_
```

Retourne bien une liste.

Cet attribut est donc bien visible mais avec un nom modifié. Il ne s'agit pas de `__attribut_` mais `_classe__attribut_`.

La méconnaissance de ce mécanisme conduit de nombreux dévelopeur Python à utiliser cette convention de nom avec l'illusion de rendre l'attribut invisible de l'extérieur. Ce n'est pas le cas (puisqu'il ne le fait pas, faites un `dir(t)`…), ce mécanisme sert à éviter qu'un attribut soit renommé par accident dans une classe spécialisée lors de la mise en œuvre de l'héritage.

9.3.3 Les properties

La visibilité des attributs d'une classe est *absolue* : elle définit tous les accès à l'attribut. Il n'est pas possible de définir un attribut qui sera accédé de manière publique en lecture mais privée en écriture. Dans notre exemple, nous avons besoin de limiter l'accès à l'attribut `students`. En effet, nous devons certes y accéder pour consulter la liste des inscrits, mais nous ne devrions pas y avoir accès en modification. L'objet doit être le seul maître de l'évolution de son état.

Pour cette raison, la visibilité est définie en privé. Pour y accéder, nous aurons une méthode.

```
class Training:
    def __init__(self, subject: str, duration: int, capacity: int = 12):
        self.subject = subject
        self._students = []

    def students(self):
        return self._students.copy()

```

Mais cette implémentation entraine une violation du **principe d'accès uniforme** : certains attributs sont obtenus par appel à l'attribut et d'autres par appel à des méthodes, ce qui provoque une complexité : comment l'utilisateur saura comment accéder à un attribut donné ? Dans beaucoup de langages, l'uniformisation consiste à définir tous les attributs en privé et à définir des méthodes dites *accesseurs*.

En outre, une évolution d'un attribut publique vers un accès par méthode, même si le nom est conservé, entraine une modification du code utilisant cet objet. En effet, il faudra modifier tous les accès pour ajouter des parenthèses. Ceci est particulièrement dangereux en Python où accéder à un méthode retourne sa référence. Déclarer tous les attributs en privé et définir des accesseurs permet également, dans beaucoup de langages, d'éviter cette situation.

Python permet le respect du principe objet en proposant un descripteur de données, les `property`.

Les `property` s'utilisent de deux manières différentes : sous forme de décorateurs ou sous forme de classe *interne*. Ce document présentera l'usage du décorateur (dont le principe est détaillé plus loin). Pour plus d'informations, voir : https://docs.python.org/3/howto/descriptor.html#properties

L'usage du décorateur `property` est simple, il suffit de le rajouter avant la méthode accesseur. Le nom de la méthode devient donc un *attribut*.

Voici un extrait du de la déclaration de la fonction limité au code qui nous intéresse.

```
class Training:
    def __init__(self, subject: str, duration: int, capacity: int = 12):
        self.subject = subject
        self._students = []

    @property
    def students(self):
        return self._students.copy()
```

L'usage sera le suivant.

```
In [2]: t_python = training.Training("Python", 5)

In [3]: print(t_python.students)
[]
```

Vous voyez qu'à l'usage, il semble y avoir un attribut `students` qui retourne bien la valeur attendue. Mais cet attribut n'existe pas, c'est une property qui masque l'accès à une fonction. Cette fonction n'accepte pas de paramètre et retourne une donnée.

Avec cette implémentation, la liste des stagiaires peut être accédée comme un attribut mais cet attribut ne pourra pas être modifié par affectation (cela lèvera un `AttributeError`). Ainsi, cette liste est accédée par la `property` et évolue par appel aux méthodes `add_student()` et `remove_student()`.

Il est possible grâce aux properties de masquer une fonction accesseur pour y accéder comme une affectation de valeur à un attribut ou pour supprimer un attribut. Ces usages sortent du périmètre de cette formation.

Notez que le descripteur `property` permet ainsi la mise en œuvre du principe de la Programmation Orientée Objet qu'est l'encapsulation. Vous allez masquer l'implémentation et ne communiquer que par les attributs et méthodes publiques.

Voici ci-dessous l'exemple à l'accès à la propriété *places restantes.* que bous avons exprimée plus haut sous forme de méthode :

```
1  class Training:
2      def __init__(self, subject: str, duration: int, capacity: int = 12):
3          self.subject = subject
4          self.available_seats = int(capacity)
5          self._students = []
6
7      @property
8      def remaining_seats(self):
9          return self.available_seats - len(self._students)
10
```

Cette propriété apparait comme un attribut d'un objet de type `Training` mais est en pratique déduit à partir d'attributs réels.

9.4 Les méthodes spéciales

Nous avons vu jusqu'ici une méthode spéciale : `__init__(self)`. Les méthodes spéciales sont des méthodes qui sont encadrés de double-underscores. Leur particularité est qu'elle ne sont jamais (rarement) appelées directement. `__init__` est appelée par l'interpréteur lors de l'instanciation d'un objet. Python propose un ensemble de méthodes spéciales :

- `__init__(self)` : initialiseur appelé juste après l'instanciation d'un objet
- `__del__(self)` : destructeur, appelé juste avant la destruction de l'objet
- `__str__(self) -> str` : est appelé par la fonction de conversion de type `str()` et par la fonction print(). Elle doit donc retourner une chaine de caractères représentant l'objet.
- `__repr__(self) -> str` : est appelé par la fonction `repr()` et doit retourner une chaine de caractères contenue entre des chevrons et contenant nom, type de l'objet et informations additionnelles.

Il est conseillé de redéfinir (surcharger, vois l'héritage) ces méthodes lorsque vous créez vos classes.

Python propose également des méthodes de comparaison entre objets. Celles-ci sont appelées par les opérations de comparaison correspondantes. Dans la colonne Opération du tableau suivant, x représente l'objet courant et y l'objet *other* avec qui il est comparé.

Méthode	**Opération**
`__lt__(self, other)`	x < y

`__le__(self, other)`	x <= y
`__eq__(self, other)`	x == y
`__ne__(self, other)`	x != y
`__ge__(self, other)`	x >= y
`__gt__(self, other)`	x > y

Ces méthodes ne doivent être redéfinies que si elles sont un sens. Si vos objets doivent être triés, vous devez redéfinir la méthode `__gt__(self, other)`.

Python propose également des méthodes spéciales qui permettent des opérations mathématiques entre les objets.

Méthode	**Opération**
`__neg__`	-x
`__add__`	x + y
`__sub__`	x - y
`__mul__`	x * y
`__div__`	x / y

Python propose d'autres méthodes spéciales, pour plus d'informations, vous pouvez consulter :

https://docs.python.org/3/reference/datamodel.html

9.5 Attributs et méthodes de classes

Les attributs et les méthodes de classe ne sont pas liés à une instance mais à une classe.

9.5.1 Attributs de classe

Un attribut de classe est déclaré au niveau de la classe :

```
1 class MyClass:
2     class_attr = None
```

D'usage, un tel attribut est accédé par `MyClass.class_attr`. Dans une instance de cette classe, vous pouvez y accéder par `self.class_attr` mais **attention** : une consultation retournera bien la valeur de l'attribut de classe mais une modification (l'affectation d'une valeur) créera un attribut d'instance qui masquera l'attribut de classe.

Les attributs de classe ont un usage technique en Python comme pour les descripteurs ou les properties.

9.5.2 Méthodes de classe

Une méthode de classe est déclarée avec un décorateur :

```
class MyClass:

    @classmethod
    def class_method(cls):
        pass
```

Ce sont des méthodes qui seront appelées préfixées par la classe, qui connaissent la classe (notez le premier paramètre `cls` et non `self`) mais pas d'instance.

En Python, elles sont utilisées pour proposer des constructeurs alternatifs. Ainsi, pour notre classe `Training`, nous pouvons proposer un constructeur qui accepte les informations sur les participants et crée une formation avec des participants. Pour l'exemple suivant, `students_data` est une liste d'informations extraite d'une base de données ou d'un CSV.

```
class Training:
    @classmethod
    def with_students(cls, subject: str, duration: int,
                      capacity: int = 12, students_data: list = None):
        training = cls(subject, duration, capacity)

        if students_data:
            for student_data in students_data:
                training.add_student(*student_data)

        return training
```

Et son usage :

```
Training.with_students('Python', 5, students_data=['Nicolas', 'Alceste',
                                                   'Agnan', 'Rufus'])
```

9.6 Héritage et abstraction

9.6.1 L'héritage

L'héritage définit la relation entre classes comme « est un ». Le vocabulaire associé est la généralisation et la spécialisation qui représentent les sens opposés de la relation.

Au delà de la définition, il y a une conséquence technique : la spécialisation est et peut faire tout ce qu'est ou sait faire la définition plus générale.

Ainsi, la généralisation consiste à avoir une définition universelle, donc réutilisable. La spécialisation consiste à adapter les caractéristiques et les comportements pour une définition plus spécifique.

9.6.2 Base de l'héritage avec les exceptions

Le premier bénéfice de la spécialisation est d'avoir un nouveau type qui est également le type de la classe héritée (et ainsi en cascade). Une première application concerne les exceptions.

En effet, pour créer nos propres exceptions, nous devrons créer des classes qui héritent de `Exception`. Vous avez vu dans le chapitre concerné qu'il existe toute une arborescence, en fait, nous pouvons spécialiser un type d'exception plus spécifique que `Exception`.

Dans notre code d'illustration, nous levons une exception de type `ValueError` lorsque nous essayons d'ajouter un stagiaire à une formation qui ne peut en accepter d'avantage. Ce n'est évidemment pas le *bon* type de message.

Nous pouvons créer un nouveau type d'exception, `TrainingFullException` qui spécialise directement `Exception`.

```
1 class TraininfFullException(Exception):
2     ...
3
```

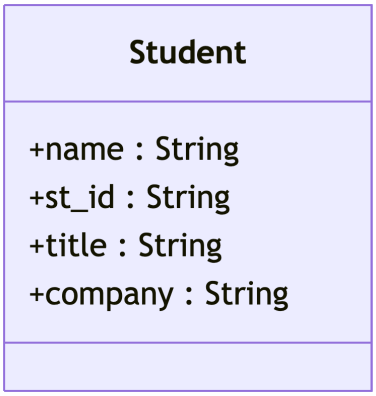
Et c'est tout. Cette simple instruction permet de créer un nouveau type d'exception en se basant sur le type existant. `TrainingFullException` se comporte exactement comme `Exception`, mais c'est un type, donc un message, en plus.

Nous allons pouvoir l'utiliser de la manière suivante.

```
1 class Training:
2     def __init__(self, subject: str, duration: int, capacity: int = 12):
3         self.available_seats = int(capacity)
4
5         if self.available_seats < 1:
6             raise TraininfFullException(f"Capacity must be positive, "
7                                         f"got {self.available_seats}")
8
```

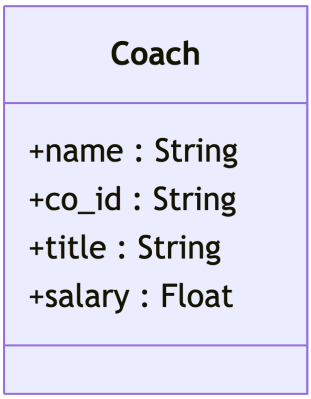
9.6.3 Base de conception de l'héritage

La relation d'héritage apparait en général lors de la conception. Si nous reprenons notre modèle, nous avons défini une classe Student pour représenter les stagiaires. Dans un soucis de simplification, je ne représente ci-dessous que les attributs de cette classe. Ces attributs ont été complétés par un identifiant et l'information de la société qui emploi le stagiaire.



Mais il nous manque une notion pour animer cette formation : l'intervenant…

Nous allons donc définir une nouvelle classe que nous allons appeler `Coach`. Cette notion nécessite une nouvelle classe pour plusieurs raisons.



D'une part, il s'agit d'un autre concept, je dois pouvoir identifier ce que je manipule.

D'autre part, l'état de l'objet est différent. L'identifiant n'est pas le même que pour les stagiaires, je n'ai pas besoin d'information de société et, indépendamment de la formation, l'intervenant a un salaire.

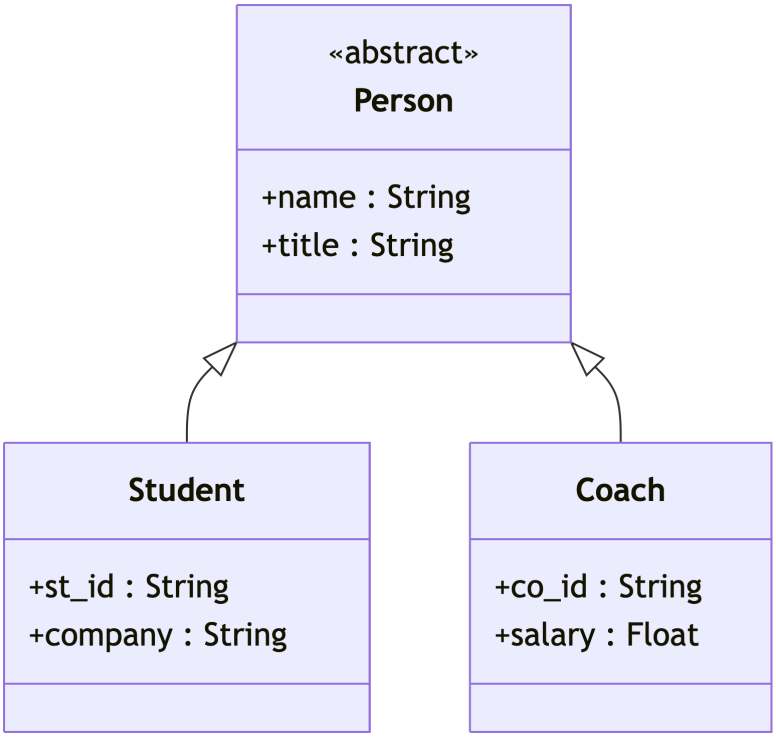
Définir un autre objet est tout à fait nécessaire. Mais bien qu'il s'agisse de deux notions différentes, elles partagent des similitudes.

En premier lieu, elles partagent la même notion : un stagiaire et un intervenant peuvent être, toutes deux, qualifiées de *personnes*. Nous pouvons perler d'elles de manière plus abstraite.

En conséquence, elles partagent des caractéristiques communes : le nom et la civilité.

Nous allons donc pouvoir **généraliser** le concept de *personne*.

La représentation de la relation de ces trois classes est représentée ci-dessous. Les classes `Student` et `Coach` héritent de `Person`. `Student` et `Coach` sont des **spécialisations** de `Person`, `Person` est une **généralisation** de `Student` ou de `Coach`.



Vous remarquerez que `Person` définit les attributs `name` et `title`. En même temps, ces attributs sont absents de la définition de `Student` et `Coach`. Mais comme `Student` ou `Coach` **sont** également de type `Person`, les objets posséderont évidemment ces attributs.

9.6.4 Notion d'implémentation de l'héritage

Commençons de manière simple. Nous avions une classe Student qui avait un constructeur que nous pouvons rappeler ici.

```
class Student:
    def __init__(self, name:str, title:str, company:str=None):
        if len(name) < 2:
            raise ValueError("Name too short")

        self.name = name
        self.title = title
        self.company = company

```

Ce constructeur possède une validation sur la taille du nom.

La classe `Coach` a un comportement similaire. Nous décidons de généraliser la notion de personne ce qui nous conduit à la définition suivante.

```python
class Person:
    def __init__(self, name:str, title:str):
        if len(name) < 2:
            raise ValueError("Name too short")

        self.name = name
        self.title = title
```

Nous retrouvons évidemment le même code que pour la classe `Student`. Code qui est, pour cette partie, identique à la classe `Coach` (non représenté ici).

La première étape de la généralisation consiste donc à définir une classe reprenant les notions *générales* de nos implémentations.

Nous allons donc maintenant pouvoir considérer que `Student` et `Coach` sont des spécialisations de `Person`. Qu'elles héritent de `Person`.

En premier lieu, pour déclarer que `Student` hérite de `Person`, il faut modifier la déclaration de la classe et ajouter, entre parenthèses, le nom des classes héritées. Oui, pluriel car Python prends en charge l'héritage multiple ce qui sera détaillé plus loin.

La base sera la suivante.

```python
class Student(Person):
    def __init__(self, name:str, title:str, company:str=None):
        self.company = company

```

Nous venons ici de définir une classe `Student` qui, en héritant de `Person`, est du même type et accède donc aux mêmes méthodes. Cependant, en déclarant la méthode `__init__()`, nous remplaçons son contenu pour la classe `Student`. Ainsi, si vous instanciez un objet de type `Student`, c'est ce constructeur qui sera exécuté et lui seul. Donc, les attributs `name` et `title` ne seront pas créés.

Pour que la classe Student délègue à la classe Person la gestion de ce qu'elle sait faire, il faut appeler sa méthode. Ceci est réalisé par une fonction spéciale, `super()`.

Notre code devient :

```python
class Student(Person):
    def __init__(self, name:str, title:str, company:str=None):
        super().__init__(name, title)
        self.company = company

```

Lorsque le constructeur sera appelé, la fonction `super()` appellera le constructeur du parent puis continuera l'exécution. Cela permet donc de déléguer une partie du traitement à la classe parent avant de réaliser la partie spécifique.

9.6.5 Héritage simple et héritage multiple

Le schéma précédent présente un héritage simple : chaque classe spécialisée n'hérite que d'une seule classe générale. Certains langages permettent l'héritage multiple, c'est à dire qu'une classe peut hériter de plusieurs classes. Elle acquiert alors les caractéristiques de l'ensemble des classes héritées.

En général, l'héritage multiple est difficile à mettre en œuvre efficacement car il est source d'erreurs et est assez difficile à maintenir. Même si le langage le permet, évitez un héritage multiple.

9.6.6 Héritage et type des objets

Le type d'un objet est sa classe. Mais dans le mécanisme de l'héritage, un objet est également du type d'une classe *parente*. Ainsi, une instance de `Student` est aussi bien de type `Student` que `Person`.

Cette propriété est fondamentale dans les langages déclaratifs afin de manipuler plusieurs types spécialisés pour leur caractéristique commune. C'est le premier pas vers les notions d'abstraction.

9.6.7 Abstraction : classes abstraites et classes concrètes

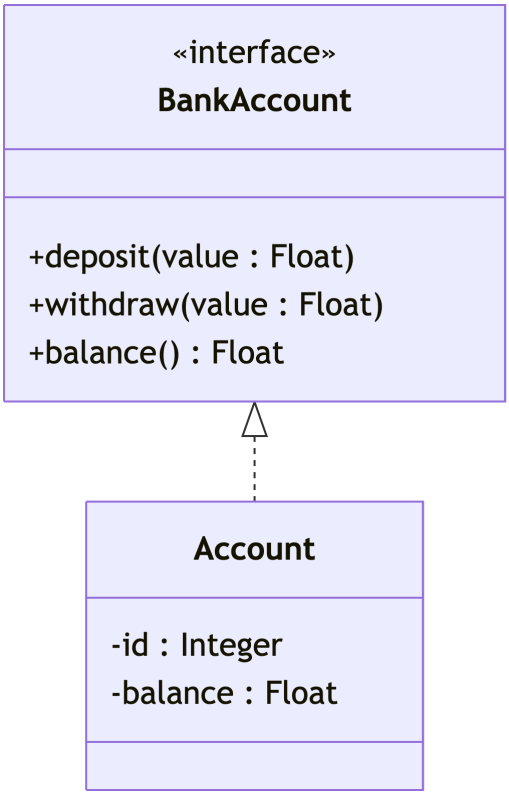
Dans l'exemple précédent, la classe `Person` est une classe abstraite. Une classe abstraite est une classe qui ne peut être instanciée. Elle ne peut être que spécialisée.

Cette propriété permet de s'assurer qu'une classe qui n'a pas de sens dans notre système ne pourra exister en elle même. Mais qu'il sera possible de la spécialiser. ### Abstraction : les interfaces

Une interface peut être définie de la manière la plus simple comme un contrat.

Il s'agit d'un composant abstrait qui ne fait que définir des méthodes publiques que les classes qui implémentent l'interface doivent déclarer. Une interface garantit donc un comportement commun entre des objets de types distincts sans aucune filiation. Une interface est représentée par un trait vertical discontinu.

L'autre propriété d'une interfaces est que, même si elle n'est pas une classe, elle confère son type à la classe qui l'implémente. Dans l'exemple ci-dessous, la classe Account est également de type `BankAccount`.



La notion d'interface est apparue avec les langages déclaratifs interdisant l'héritage multiple. Elles permettent un comportement et un type commun entre objets sans filiation.

Python et les abstractions

Par rapport à ce qui a été présenté ci-dessus, Python

— permet l'héritage
— supporte l'héritage multiple
— ne possède pas (nativement) la notion de classes abstraites
— ne possède pas la notion d'interfaces

Python ne propose pas nativement la notion de classe abstraite. Cependant, la PEP-3119 propose la notion d'*Abstract Base Class* (ABC). Ce mécanisme n'est pas abordé dans cette formation.

9.6.8 L'héritage en Python

Pour définir qu'une classe est une spécialisation d'une autre en Python, on l'indique dans la déclaration de la class en faisant suivre le nom de la classe par des parenthèses contenant le ou les nom(s) de la ou des classe(s) héritée(s).

La déclaration de la classe `Client` ci-dessous suffit pour déclarer un nouveau type spécialisant la notion de `Person`. Ce nouveau type déclare tous les états et comportements du type `Person`.

```
class Client(Person):
    pass

```

Spécialiser une classe signifie lui donner un comportement spécifique, mais cela peut également signifier lui attribuer des comportements complémentaires.

Dans ce dernier cas, une classe spécialisée peut déléguer à la classe générale ce qu'elle sait gérer. Lors de l'instanciation d'une classe Client, la classe Person sait gérer l'initialisation de nom et prénom. Nous pouvons lui déléguer cette gestion dans le constructeur.

```
class Client(Person):
    def __init__(self, nom, prenom, nid):
        super().__init__(nom, prenom)
        self._identifiant = nid
        self._accounts = []

```

La fonction `super()` permet d'appeler les *parents* de la classe `Client` et donc leur méthode. Dans le cas de l'héritage simple (`Client` est la spécialisation d'un seul parent), c'est donc la méthode du parent qui est appelée. Dans le cas de l'héritage multiple (`Client` est la spécialisation de plusieurs parents), la fonction `super()` gérera l'ordre d'appel des méthodes des parents.

Les attributs créés par l'initialiseur de la classe Person seront évidemment accessibles par l'objet Client.

9.7 Notions de polymorphisme

Le polymorphisme est la capacité d'un système à choisir l'implémentation réelle d'une méthode. Nous la mettons déjà en œuvre en surchargeant la méthode `__init__(self)`.

Spécialiser une classe peut ainsi conduire à redéfinir le comportement de certaines méthodes. Si nous avons un nouveau type de compte `DeferedAccount` qui est compte à débit différé, l'opération de débit ne peut être la même que la classe Account dont il hérite. Nous allons donc implémenter cette classe de la manière suivante :

```
class DeferedAccount(Account):

    def __init__(self, nid, balance=100):
        Account.__init__(self, nid, balance)
```

```
5           self._operations = []
6
7       def withdraw(self, value):
8           self._operations.append(value)
9
```

Pour les objets de type `DeferedAccount`, l'interpréter ne fera pas appel à la méthode `withdraw()` de la classe parent mais bien à celle redéfinie (surchargée) dans la déclaration de `DeferedAccount`.

9.8 Polymorphisme et duck typing

En Python, la notion de polymorphisme repose sur la notion de duck typing.

> Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard

Ce qui nous intéresse en Python, ce n'est pas quel est le type réel de l'objet mais est-ce qu'il a la caractéristique nécessaire à l'action.

Ce principe peut être résumé par cette citation :

> Don't check whether it IS-a duck : check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with. If the argument fails this speci c-ducklyhood-subset-test, then you can shrug, ask « why a duck ? »
>
> *Alex Martelli*

Ce qui nous conduit à un principe clef en Python :

> It's easier to ask for forgiveness than permission

9.8.1 Ne testez pas si c'est un canard...

Il existe une fonction qui permet de vérifier si un objet est une instance d'une classe : `isinstance(objet, classe)`. En Python, nous allons rarement l'utiliser. Plutôt que de tester si l'objet est de cette instance, nous testerons si il a les propriétés qui nous intéressent. Pour cela, nous utiliserons la fonction `hasattr(objet, nom)` qui permet de vérifier si un objet a un certain attribut. Le paramètre nom est une chaine de caractère du nom de la propriété.

```
In [1]: hasattr(my_account, 'deposit')
Out[1]: True

In [2]: hasattr(my_account, 'upper')
Out[2]: False
```

9.8.2 Ou ne testez pas du tout

Si vous tentez d'appeler une méthode qui n'existe pas, cet appel va provoquer une exception de type `AttributeError`.

```
In [3]: my_account.upper()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-436618faa434> in <module>()
----> 1 my_account.upper()

AttributeError: 'DebtAccount' object has no attribute 'upper'
```

Vous pouvez donc directement capturer et gérer l'exception comme le décrit le chapitre suivant.

9.9 Les attributs

9.9.1 Affecter et accéder à une valeur dans l'objet

Un attribut dans la classe a un syntaxe `self.nom_attribut`.

On affecte une valeur à un attribut en déclarant celui-ci préfixé par `self` :

```
class MyClass:
    def assign_value(self, value:int):
        self.attribute = value

```

On y accède en le référant de la même manière.

```
class MyClass:
    def access_value(self) -> int:
        return self.attribute

```

9.9.2 Accéder à un attribut en dehors de l'objet

On peut accéder à un attribut en dehors de l'objet

```
>>> my_object = MyClass()
>>> my_object.assign_value(42)
>>> my_object.value
42
```

On peut modifier la valeur par assignation avec la même syntaxe. Note que si l'attribut n'existe pas, il est créé.

```
>>> my_object = MyClass()
>>> my_object.value = 42
>>> my_object.value
42
```

9.10 Les méthodes spéciales

Documentation : https://docs.python.org/3/reference/datamodel.html

9.10.1 L'initialiseur

```
class MyClass:
    def __init__(self, param_1, param_2):
        self.attribute_1 = param_1
        self.attribute_2 = param_2

```

9.10.2 La représentation sous forme de chaine de caractères

9.10.2.1 Représentation informelle avec `__str__`

```
class MyClass:
    def __init__(self, param_1, param_2):
        self.attribute_1 = param_1
        self.attribute_2 = param_2

    def __str__(self):
        return f'Objet {self.attribute_1} {self.attribute_2}'

```

9.10.2.2 Représentation *officielle* avec `__repr__`

The target audience for the string representation returned by `.__repr__()` is the programmer developing and maintaining the program. In general, it provides detailed and unambiguous information about the object. Another important property of the official string representation is that a programmer can normally use it to re-create an object equal to the original one.

One way to implement the method is the following :.

```
class MyClass:
    def __init__(self, param_1, param_2):
        self.attribute_1 = param_1
        self.attribute_2 = param_2

    def __repr__(self):
        class_name = type(self).__name__
        return f"{class_name}(param_1={self.attribute_1!r},
                              param_2={self.attribute_2!r})"

```

Note : si `__repr__`est surchargé mais pas `__str__`, le retour de `__repr__` sera utilisé lors des requêtes à `__str__`.

9.10.3 Autres méthodes spéciales

9.10.3.1 Opérateurs de comparaison

Méthode spéciale	Opérateur
`__lt__(self, other)`	`self < other`
`__gt__(self, other)`	`self > other`
`__le__(self, other)`	`self <= other`
`__ge__(self, other)`	`self >= other`
`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`

9.10.3.2 Opérateurs mathématiques

Méthode spéciale	Opérateur
`__neg__(self)`	`-self`
`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`

Méthode spéciale	Opérateur
`__truediv__(self, other)`	`self / other`

9.11 Visibilité des attributs

En Python, tout a une visibilité publique. Par convention, un attribut est considéré comme *non-public* si son nom commence par un underscore (_) :

```
1 class MyClass:
2     def __init__(self):
3         self._non_publi_attribute = 42
4 
5     def _non_public_method(self):
6         pass
```

10 Pour quelques notions de plus

Cette partie présente quelques *outils* que propose Python pour une meilleur implémentation de l'Objet.

10.1 Le paramètre `self` de la déclaration des méthodes dans la classe

Lorsque vous écrivez une méthode, vous êtes tenus de déclarer un premier paramètre qui par convention, aura pour nom `self`. Ce paramètre est destiné à recevoir l'instance. Explication…

Les objets sont des instances de classes. Le principe de la programmation orientée objet consiste à définir des entités qui regroupent un état et des comportements. Il paraît évident de se représenter les objets comme des composants construits sur le modèle d'une classe et qui embarquent données (les attributs) et comportements. Mais en Python, l'implémentation de la POO est par *prototype*, ce n'est donc pas tout à fait ça.

Commençons par définir une classe et créons une instance.

```
In [1]: class Demo:
   ...:     def __init__(self, name):
   ...:         self.name = name
   ...:
   ...:     def say_hello(self):
   ...:         print(f"Hello, I'm {self.name}")
   ...:

In [2]: d = Demo("One")
```

Puisque les méthodes sont des fonctions, si nous faisons appel à `d.say_hello`, nous devons avoir en retour une référence à la méthode (fonction) `say_hello` de l'objet `d`.

```
In [3]: d.say_hello
Out[3]: <bound method Demo.say_hello of <__main__.Demo object at 0x102d2ceb0>>
```

D'après ce retour, nous pouvons vérifier que nous pouvons accéder à `Demo.say_hello`.

```
In [4]: Demo.say_hello
Out[4]: <function __main__.Demo.say_hello(self)>
```

Python voit donc `say_hello(self)` comme une fonction définie dans la classe `Demo`, perçue comme un namespace.

La méthode `say_hello()` de l'objet (instance) d est donc une *méthode liée* à cette fonction.

Nous pouvons donc exécuter la fonction `Demo.say_hello()`… Ce qui va lever une exception car il manque un arguments. Cet argument correspond au paramètre `self`.

Le paramètre `self` attends en effet une instance de `Demo`.

```
In [5]: Demo.say_hello(d)
Hello, I'm One
```

Cette instruction, c'est ce que l'interpréteur exécute réellement. Nous écrivons une instruction qui fait appel à la méthode de l'instance et l'interpréteur va exécuter la méthode de la classe en fournissant en premier paramètre l'instance. Ce paramètre `self` est donc un paramètre de la méthode de la classe destiné à recevoir l'instance sur laquelle exécuter la méthode.

10.2 Notion de properties

Lorsqu'un attribut est déclaré comme *non_public*, pour accéder à sa valeur, li

Cette partie ne présente les properties que en tant que décorateurs

10.2.1 Accesseurs (*getters*)

```
class MyClass:

    @property
    def my_property(self):
        return 42
```

Usage :

```
>>> my_object = MyClass()
>>> my_object.my_property
42
```

10.2.2 Mutateurs (*setters*)

Le mutateur nécessite qu'un accesseur ai été défini

```
class MyClass:
    def __init__(self):
        self._property_value = 42

    @property
    def my_property(self):
        return self._property_value

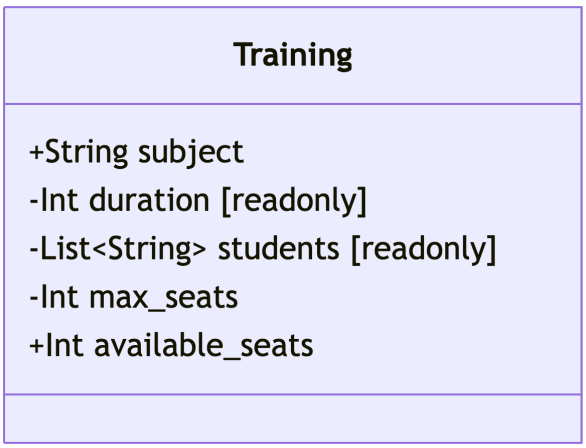
    @my_property.setter
    def my_property(self, new_value):
        self._property_value = new_value
```

Usage :

```
>>> my_object = MyClass()
>>> my_object.my_property
42
>>> my_object.my_property = 24
>>> my_object.my_property
24
```

10.2.3 Cas pratique : notre formation

Si je reprends notre formation en ne m'intéressant qu'aux attributs, sa conception *logique* doit être :



Note : l'outil "mermaid" utilisé pour générer ce schéma ne supporte pas la représentation standard qui devrait être `-List<String> students {readonly}`.

```python
class Training:
    def __init__(self, subject:str, duration:int, seats:int):
        self.subject = subject
        self._duration = duration
        self._seats = seats
        self.students = []
```

Pour les attributs en *lecture seule* `duration` et `students`, l'ajout de properties accesseurs seules est évidente. Attention à ne pas retourner la référence vers `_students`.

```python
@property
def duration(self):
    return self._duration

@property
def students(self):
    return self._students.copy()
```

Le diagramme UML indique que nous devons avoir un attribut `max_seats` en lecture seule représentant le nombre maximum de places. Sauf que nous avons un attribut *non_public* `_seats` et non `_max_seats`. Et ceci n'est pas un problème : rien n'oblige à ce que les attributs exposés (publiques) référencent un attribut *privé* du même nom. Donc, on ajoute la property

```python
@property
def max_seats(self):
    return self._seats
```

Enfin, il y a un attribut en lecture seule `available_seats`. Celui-ci représente la différence entre le nombre de places de la formation et le nombre d'inscrits. Au lieu de retourner la valeur d'un attribut *privé*, nous pouvons calculer cette valeur :

```python
@property
def available_seats(self):
    return self.seats - len(self._students
```

Et ceci illustre en quoi les properties sont un outil fondamental de l'encapsulation : elles permettent de décorréler l'implémentation de l'interface de l'objet. De l'extérieur, on voit un état cohérent pour l'utilisateur de l'objet mais l'implémentation utilise un état cohérent pour son traitement.

Quatrième partie

La lib standard (et plus)

11 Les tests en Python

Les tests font partie intégrante du développement logiciel. Le test le plus proche du développeur est le test unitaire. Cette partie traitera que des tests unitaires.

11.1 Pourquoi tester ?

Les tests ne font pas parti du code *fonctionnel*. Ils sont toutefois fondamentaux dans le développement logiciel. Les tests permettent de

— Montrer que le code fonctionne
— Montrer que le code répond aux attentes
— Illustrer l'usage du code
— Montrer que le code fonctionne **toujours**

Python propose deux outils pour les tests : `doctest` et `unittest`. L'intérêt de ces deux outils est qu'ils sont présents dans la distribution standard de Python. Cependant, une bibliothèque tierce rend en Python les tests beaucoup plus simples : `PyTest`.

11.2 Démarche générale pour le test

Tester consiste à s'assurer que le système a le comportement attendu en fonction d'une condition donnée. Tester se fait en 3 étapes, les 3A :

— **Arrange** : Initialiser le système dans un état de départ. Ce système est le système testé habituellement appelé SUT (System Under Test).
— **Act** : Exécuter le code que nous souhaitons tester.
— **Assert** : Observer le système et s'assurer qu'il est dans l'état attendu. ## Notions de tests unitaires

Le code relatif à cette partie est à voir dans les sources avec le projet.

Tester un programme fait parti du processus de réalisation du programme. Vous testez naturellement vos programmes en vérifiant leur fonctionnement. Python en particulier permet de vérifier des parties de code grâce à l'interpréteur interactif.

Si cette pratique permet de valider le bon comportement du code, elle a le défaut d'être manuelle et donc ne nécessiter un effort d'analyse du résultat de vos instructions. Le principe des tests n'est pas très différent de votre pratique et va surtout permettre l'automatisation.

11.2.1 Glossaire

Dans la partie suivante, le **code fonctionnel** fera référence au code destiné à la production. C'est le code qui répond à votre besoin. Le **code de test** correspond au code destiné à tester le code fonctionnel.

11.2.2 Principe des tests

L'automatisation des tests consiste à écrire du code de code qui comporte deux parties :

— Du code qui exécute le code fonctionnel
— Du code qui *valide* le comportement du code fonctionnel.

La première partie, vous l'écrivez déjà lorsque vous écrivez du code de validation manuelle. Nous allons devoir ajouter du code de *validation*. Une validation consistera en une *assertion* d'état. Les bibliothèques de tests fournissent un ensemble d'assertions que nous pourrons utiliser.

Ainsi, nous pourrons de manière automatique *jouer* et *re-jouer* ces tests. Cette automatisation a 4 intérêts :

— Elle permet de montrer que le code fonctionne
— Elle permet de montrer que le code répond aux attentes
— Elle permet d'illustrer l'usage du code
— Elle permet de montrer que le code fonctionne toujours

Ce dernier point est important. Un code peut ne plus fonctionner suite à une modification. On appelle cela une **régression**.

Pour éviter la régression, nous devons avoir une stratégie de tests afin d'identifier le plus rapidement et le plus précisément toute régression. C'est pour cela que les **tests unitaires** ont été créés.

11.2.3 Principe des tests unitaires

Le principe des tests unitaires consiste à tester de manière **isolée** les plus petites parties possible du programme afin d'identifier le plus précisément les régressions ou les défaillances.

Tester de manière isolée doit permettre de s'assurer un comportement déterministe, c'est à dire que le test produira toujours le même résultat. Un test en échec doit mettre en évidence une *erreur* du code et non une influence extérieur.

Exemple : lorsque vous testez la récupération d'une base de données réseau, vous n'êtes pas isolé. Votre attente sera de récupérer certaines données que vous ne récupérerez pas si votre code est mal écrit, mais aussi si le réseau est absent, que le serveur n'est pas accessible, que la connexion à la base est impossible…

11.3 Doctest, la documentation auto-testée

Doctest est un test inclus dans la documentation proposé par le module doctest. Le principe est d'illustrer par un code simple le composant documenté.

```
def add(a, b):
    """
        :Example:

        >>> add(1, 1)
        2
        >>> add(2.1, 3.4)
        5.5

    """
    return a + b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Dans cet exemple, l'exécution du module exécutera doctest qui recherchera tout code dans la documentation, l'exécutera et comparera le retour obtenu avec le retour attendu. Doctest ne signale que les erreurs.

Doctest est un outil pratique permettant d'illustrer l'usage mais limité à des exemples simples. N'oubliez pas que ces exemples vont se trouver dans la documentation.

11.4 Les tests unitaires avec unittest

Unittest, aussi appelé PyUnit, est la version Python de JUnit.

Junit est le framework de test créé par Kent Beck et Erich Gamma pour le langage Java en 1997. Il est lui même inspiré de SUnit créé par Kent Beck en 1994 et est considéré comme le framework fondateur des framework de type xUnit.

La documentation complète est disponible à l'adresse : https://docs.python.org/3/library/unittest.html

11.4.1 Mise en œuvre de Unittest

Le principe des tests unitaires est de tester un comportement et un seul. Pour cela, le système testé doit être isolé au maximum afin qu'aucun composant externe n'influe sur le résultat du test.

Unittest permet l'isolation et l'indépendance des tests.

Le principe de mise en œuvre est le suivant :

— On crée une classe héritant d'unittest par fonctionnalité testée
— Au sein de cette classe, on crée une méthode de test par cas testé. Le nom de ces méthodes doit commencer par `test`, unittest détermine par introspection si une méthode est une méthode de test.
— `Unittest` met à disposition deux méthodes, `setup()` et `teardown()` qui sont exécutées respectivement avant et après chaque méthode de test et qui sont destinés à préparer et à nettoyer l'environnement de test.

Un test (implanté dans une méthode de test) consiste à exécuter un appel à la fonctionnalité et à faire une assertion sur l'état après cet appel.

Classes et méthodes doivent avoir un nom sans ambiguïté sur leur comportement. En cas d'échec d'un test, le reporting indiquera le nom de ces classes et méthodes. Un nom adapté facilitera la correction.

Les méthodes `setup()` et `teardown()` vous permettent de garantir l'indépendance des tests. La première doit s'assurer de la préparation de l'environnement pour le test. La seconde s'assure de nettoyer l'environnement pour le test suivant. N'oubliez pas que **vous ne pouvez faire aucune spéculation sur l'ordre d'exécution des tests**.

11.4.2 Cas des tests simples

Le code suivant test la création d'une formation. Nous testons les différents cas sans oublier le comportement lorsque la durée ou le nombre de places est d'un autre type.

```
1  import unittest
2  import academy.training as tr
3
4  class TestCreateTraining(unittest.TestCase):
5
6      def testCreateTraining(unittest.TestCase):
7          training = tr.Training('Python, fondamentaux', 5)
8          self.assertEqual('Python, fondamentaux', training.subject)
9          self.assertEqual(5, training.duration)
10         self.assertEqual(12, training.seats)
11
12     def testCreateTrainingWithSeats(unittest.TestCase):
13         training = tr.Training('Python, fondamentaux', 5, 8)
14         self.assertEqual(5, training.duration)
15         self.assertEqual(8, training.seats)
16
17     def testCreateTrainingWithValuesAsStrings(unittest.TestCase):
18         training = tr.Training('Python, fondamentaux', '5', '8')
```

```
19          self.assertEqual(5, training.duration)
20          self.assertEqual(8, training.seats)
```

11.4.3 Cas des tests modifiant l'état

Le code suivant l'ajout de participants. Tester un ajout de participant nécessite l'existence d'un objet de type formation, donc sa création qui est réalisée dans la méthode `setUp()`. Cet objet étant modifié par l'appel à la méthode `.add_student()`, il est détruit par la méthode `tearDown()`.

```
1  class TestNewStudents(unittest.TestCase):
2
3      def setUp(self):
4          self.training = tr.Training('Python, fondamentaux', 5)
5
6      def testAddFirstStudent(self):
7          self.training.add_student('Eliott Alderson')
8          self.assertEqual(1, len(self.training.students)
9
10     def tearDown(self):
11         del self.training
```

Le bon fonctionnement de la méthode `.add_student()` nécessite que l'objet soit correctement créé. La bonne création de l'objet est assurée par les tests précédents.

11.4.4 Les assertions

Les principales assertions sont :

Methode	**Vérifie que**
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Dans le cas de notre formation, ajouter un doublon (représenté par le même nom) conduit à la levée d'une exception. Il existe une assertion spécifique pour ce cas, `assertRaises()` qui prend en paramètre l'exception attendue. Elle est utilisée de la manière suivante.

```
def testDuplicateStudent(self):
    self.training.add_student('Eliott Alderson')

    with self.assertRaises(ValueError):
        self.training.add_student('Eliott Alderson')
```

Pour discriminer l'origine de la levée de l'exception, nous nous baserons sur les arguments et paramètres de l'exception. La source de l'exception doit pouvoir être identifiée non pas pour des questions de *testabilité* mais de facilité de maintenance.

```
def testDuplicateStudent(self):
    self.training.add_student('Eliott Alderson')

    with self.assertRaises(ValueError) as context:
        self.training.add_student('Eliott Alderson')
        self.assertTrue('Duplicate student' in context.exception)
```

11.5 Les tests unitaires avec pytest

Pytest, ou Py.Test, est une bibliothèque qui se veut plus *Pythonique* que unittest avec une philosophie « we-don't-need-no-stinking-API unit test suite ». L'objectif est de simplifier l'écriture des tests sans en perdre la puissance des libs xUnit. Ainsi, PyTest n'utilise pour les tests que de simples fonctions et assertions. Le premier exemple de tests unittest se transpose en Pytest de la manière suivante :

```
import academy.training as tr

def test_create_training(self):
    training = tr.Training('Python, fondamentaux', 5)
    assert 'Python, fondamentaux', training.subject
    assert training.duration == 5
    assert training.seats == 12

def test_create_training_with_seats(self):
    training = tr.Training('Python, fondamentaux', 5, 8)
    assert training.duration == 5
    assert training.seats == 8

def test_create_training_with_values_as_strings(self):
```

```
15      training = tr.Training('Python, fondamentaux', '5', '8')
16      assert training.duration == 5
17      assert training.seats == 8
```

Avec pytest, les tests ne sont que de simples fonctions et l'assertion est vérifiée avec la directive native de Python `assert`. Vous pouvez remarquer que le code précédent se suffit à lui même. Il n'y besoin d'aucune dépendance à la bibliothèque de tests.

Pour exécuter les tests, il suffit d'exécuter l'instruction `py.test` sur l'invite de commande à la racine du projet. Pytest suit les conventions de découverte des tests et exécutera donc tous les tests de l'arborescence. Il évidemment possible de fournir un chemin en paramètre afin de spécifier un répertoire ou un module pour cette exécution ou même de le configurer dans le `pyproject.toml`.

11.5.1 Tester la levée d'exceptions

Il est indispensable de tester que certaines exceptions sont bien levées dans des conditions spécifiques. Afin de conserver un code concis et lisible, Pytest fournit un context manager. En déclarant ce context manager, il faut indiquer quelle exception doit être levée.

```
1  import academy.training as tr
2  import pytest
3
4  def test_duplicate_student_must_raise(create_account):
5      training = tr.Training('Python, fondamentaux', 5)
6      training.add_student('Eliott Alderson')
7
8      with pytest.raises(ValueError):
9          training.add_student('Eliott Alderson')
```

Le test de l'appel de la méthode `deposit()` nécessite une instance d'objet. Cependant, nous aurons besoin de tester ce même système dans différentes conditions. Il sera donc nécessaire de pouvoir accéder à cet objet de manière uniforme.

11.5.2 Une base de tests avec les test fixtures

Les test fixtures offrent une base fixe afin de répéter de manière fiable un contexte de tests. L'usage le plus simple est de créer un objet lorsque l'on veut tester un appel de méthode.

Un objet doit être initialisé de la même manière pour l'ensemble des tests de la fonctionnalité. Une fixture permet de s'assurer que nous aurons toujours le même objet dans les différents tests. Nous pouvons donc reprendre le test de l'ajout de participants dans l'ensemble de tests ci-dessous.

```python
from academy.training as tr
import pytest

@pytest.fixture()
def training:
    return tr.Training('Python, fondamentaux', 5)

def test_add_student(training):
    training.add_student('Eliott Alderson')

    assert len(training.students) == 1
    assert training.students[-1] == "Eliott Alderson"

def test_duplicate_student_must_raise(training):
    training.add_student('Eliott Alderson')

    with pytest.raises(ValueError):
        training.add_student('Eliott Alderson')
```

Vous remarquerez qu'une fixture se présente comme un paramètre de la fonction. Il est donc possible de passer plusieurs fixtures pour un même test.

Le mécanisme des fixtures est de l'injection de dépendances.

11.6 L'approche test-first

Il est indispensable pour la qualité du code que celui-ci soit accompagné de tests. La démarche habituelle consiste à écrire son code en fonction des spécifications puis les tests. Cette démarche à plusieurs défauts :

— Reportés à la fin de la tâche, les tests ne sont finalement pas écrits. Après tout, le code fonctionne et il est urgent de passer à la tâche suivante.
— Les tests sont écrits en fonction du code existant et montrent surtout que le code répond aux attentes.
— Cette stratégie confirme le reproche fait à l'écriture des tests qui est que cela prends du temps.

L'idéal serait d'écrire les tests en même temps que le code en diminuant au maximum le temps nécessaire à leur écriture. C'est ce que promet l'approche TDD pour Test Driven Development. Ce document n'a pas l'ambition d'être une une formation à TDD (qui mérite une formation dédiée). Mais en résumé, si vous souhaitez mettre en œuvre cette approche, vous avez 3 règles à respecter scrupuleusement :

1. Vous ne pouvez écrire du code de production que pour corriger un test en échec
2. Vous ne devez écrire qu'un minimum de code de test pour que celui-ci soit en échec

3. Vous ne devez pas écrire plus de code de production que pour faire passer un test en échec

Vous y verrez que

— Les intentions sont écrites avant leur réalisation
— Les intentions sont exhaustives
— Les tests tendent à être plus exhaustifs comparé à leur écriture à postériori
— Le temps passé à écrire les tests est compensé par l'absence d'écriture de code inutile.

12 Les fichiers et le système de fichiers

La manipulation de fichiers comporte deux aspects :

— Manipuler le contenu du fichier.
— Manipuler les fichiers, c'est à dire le système de fichiers.

Nous allons voir dans un premier temps ce que Python propose pour la manipulation des fichiers avant de nous intéresser au contenu des fichiers.

12.1 Le système de fichiers

Il y a fondamentalement 3 modules qui vont offrir des fonctionnalités pour la manipulation du système de fichiers :

— `os` : https://docs.python.org/3/library/os.html
Permet d'utiliser des fonctions dépendant du système d'exploitation
— `os.path` : https://docs.python.org/3/library/os.path.html#module-os.path
Permet de manipuler les *chemins*. os.path est un module virtuel dont l'implémentation réelle est chargée en fonction du système sur lequel s'exécute le code.
— `shutil` : https://docs.python.org/3/library/shutil.html#module-shutil
Permet des manipulations de haut-niveau des fichiers. Certaines fonctions peuvent appeler des fonctions du module os.

À ces trois là s'ajoute un module plus récent, pathlib, plus récent et qui propose une approche objet pour manipuler le système de fichiers. Il tend à remplacer principalement `os.path`.

Consultez les documentations en lien de chaque modules pour en apprendre d'avantage sur ce que propose chaque module. Dans ce document, nous allons plutôt nous intéresser à des cas pratiques.

12.1.1 Obtenir le contenu d'un répertoire

Il existe plusieurs méthodes pour obtenir le contenu d'un répertoire. La manière *historique* utilise la fonction `os.listdir(path)` qui retourne une liste de chaines de caractères.

```
In [2]: import os
In [3]: os.listdir("assets")
Out[3]:
['Paris', 'bbts12.csv','movieslist.csv', 'ile_de_france',
'showslist_bad.csv', 'ministere_culture', 'showsfiles.txt',
'files', 'movieslistdest.csv', 'showslist.csv', 'SNCF']
```

Bien entendu, vous ne manipulerez jamais réellement la liste mais les éléments de la liste.

La fonction `listdir()` ne prends pas en charge les glob (les wildcards utilisés pour filtrer le contenu d'un répertoire). Historiquement, il a été ajouté un module `glob` qui contient une fonction `glob()`.

```
In [4]: import glob
In [5]: glob.glob('assets/*.csv')
Out[5]:
['assets/bbts12.csv', 'assets/movieslist.csv',
'assets/showslist_bad.csv', 'assets/movieslistdest.csv',
'assets/showslist.csv']
```

Depuis Python 3.5, il existe une nouvelle manière de manipuler le contenu d'un répertoire à l'aide de la fonction `os.scandir()`. La fonction `scandir()` retourne un itérateur sur des objets de type `os.DirEntry`.

```
In [6]: for file in os.scandir("assets"):
   ...:     print(file)
   ...:
<DirEntry 'Paris'>
<DirEntry 'bbts12.csv'>
<DirEntry 'movieslist.csv'>
<DirEntry 'ile_de_france'>
…
```

L'usage conseillé utilise un contexte manager de la manière suivante :

```
In [7]: with os.scandir("assets") as entries:
   ...:     for file in entries:
   ...:         print(file)
   ...:
<DirEntry 'Paris'>
<DirEntry 'bbts12.csv'>
<DirEntry 'movieslist.csv'>
<DirEntry 'ile_de_france'>
…
```

La raison est que le Context Manager s'assurera de la bonne fermeture de l'itérateur et de la libération des ressources.

12.1.2 Exploiter le contenu d'un répertoire

À nouveau, tout va dépendre de la fonction que vous allez utiliser, `listdir()` ou `scandir()`. Dans le premier cas, vous avez des chaines de caractères représentant les noms *basename* des fichiers, dans le second, des objets de type `os.DirEntry`.

Les objets de type os.DirEntry possèdent des attributs et méthodes qui vont faciliter la manipulation des fichiers. Nous avons ainsi :

- `DirEntry.name` : retourne le nom de base du fichier (sans l'arborescence)
- `DirEntry.path` : retourne le nom du fichier avec l'arborescence relative (celle qui a été utilisée pour `scandir`).
- `DirEntry.is_dir()` : retourne True si il s'agit d'un répertoire sinon `False`
- `DirEntry.is_file()` : retourne True si il s'agit d'un fichier sinon `False`
- `DirEntry.stat()` : retourne un objet de type `stat_result` comportant un ensemble d'informations sur le fichier. Ne sera pas couvert dans cette formation.

Ces attributs et méthodes permettent déjà d'obtenir des informations sur les éléments contenus dans un répertoire.

Ainsi, si nous souhaitons lister uniquement les fichier d'un répertoire avec un chemin relatif, avec `scandir()`, il suffit d'écrire le code suivant :

```
In [8]: basepath = "assets"
In [9]: with os.scandir(basepath) as entries:
   ...:     for file in entries:
   ...:         if file.is_file():
   ...:             print(file.path)
   ...:
assets/bbts12.csv
assets/movieslist.csv
assets/showslist_bad.csv
assets/showsfiles.txt
assets/movieslistdest.csv
assets/showslist.csv
```

Avec `listdir()`, il faudra quelques éléments en plus.

Il est possible de tester si un chemin est un fichier ou un répertoire grâce aux fonctions `os.path.isdir(path)` et `os.path.isfile(path)`.

Mais il faudra fournir un chemin à ces fonctions. Or `listdir()` ne retourne que le nom *basename* du fichier. Il faudra donc le reconstruire grâce à la fonction `os.path.join(path, *paths)`.

```
In [10]: basepath = "assets"

In [11]: for entry in os.listdir(basepath):
    ...:     if os.path.isfile(os.path.join(basepath, entry)):
    ...:         print(entry)
    ...:
bbts12.csv
movieslist.csv
showslist_bad.csv
showsfiles.txt
movieslistdest.csv
showslist.csv
```

12.2 En résumé

12.2.1 Obtenir le contenu d'un répertoire

— `os.listdir(path)` : retourne le contenu d'un répertoire sous forme d'une liste de chaînes de caractères représentant le nom `basename` du contenu du répertoire.
— `os.scandir(path)` : retourne un itérateur sur des objets de type `DirEntry` représentant le contenu du répertoire.
— `glob.glob(path)` : liste le contenu du répertoire en fonction du pattern et permet ainsi l'utilisation de wildcards.

12.2.2 Créer des répertoires

— `os.mkdir(path, mode)` : crée répertoire, mode UNIX
— `os.makedirs(path, mode)` : crée une arborescence de répertoires, mode UNIX

12.2.3 Supprimer des fichiers et des répertoires

— `os.remove(path)` : supprime un fichier
— `os.rmdir(path)` : supprime un répertoire
— `os.removedirs(name)` : supprime répertoires récursivement à partir de la fin (leaf)
— `shutil.rmtree(path)` : supprime le répertoire et tout son contenu

12.2.4 Copier, déplacer et renommer

— `shutil.copy(src, dest)` : copie un fichier. `dest` peut être un répertoire, dans ce cas, le fichier sera copié dans ce répertoire.
— `shutil.copy2(src, dest)` : copie un fichier avec les métadonnées. `dest` peut être un répertoire, dans ce cas, le fichier sera copié dans ce répertoire.

— `shutil.copytree(src, dest)` : copie un répertoire et son contenu.
— `shutil.move(src, dest)` : déplace ou renomme un fichier ou un répertoire
— `os.rename(src, dest)` : renomme fichier ou répertoire. Si `dest` est une arborescence, déplace le fichier ou le répertoire.
— `os.renames(src, dest)` : renomme fichier ou répertoire en créant les répertoires si ils n'existent pas. Si `dest` est une arborescence, déplace le fichier ou le répertoire.

12.2.5 Informations sur le nom de fichier

Dans les exemples suivants, `path` est un nom de fichier ou répertoire contenant ou non une arborescence.

— `os.path.exists(path)` : est-ce que le fichier ou répertoire existe
— `os.path.isdir(path)` : est-ce un répertoire
— `os.path.isfile(path)` : est-ce un fichier
— `os.path.join(path, *paths)` : retourne une chaine de caractères représentant un chemin, concaténation des paramètres.
— `os.path.dirname(path)` : retourne l'arborescence de répertoires
— `os.path.basename(path)` : retourne le nom du fichier
— `os.path.split(path)` : retourne un tuple des deux précédents
— `os.path.splitext(path)` : retourne un tuple pour obtenir l'extension

12.3 Le module pathlib

Le module `pathlib`, ajouté à partir de Python 3.4 (voir https://www.python.org/dev/peps/pep-0428/), permet de manipuler les chemins avec une approche objet. La classe principale est la classe `Path` que nous allons importer directement.

```
1 In [1]: from pathlib import Path
```

Cette classe permet de créer un objet *chemin* en lui passant en paramètre une chaîne de caractères représentant ce chemin. Ce chemin peut être absolu ou relatif. Dans l'exemple de code plus bas, vous pouvez voir que le type de l'objet est `PosixPath`. `Path()` est une classe abstraite (ou plus exactement une *factory* et donc une *abstract factory*) . Elle retourne l'implémentation concrète selon le système d'exploitation : `PosixPath` pour les systèmes posix, `WindowsPath` pour Windows.

La classe `Path()` permet de créer facilement un objet sur le répertoire courant. Ce chemin reste abstrait. Vous pouvez créer un nouvel objet avec le chemin réel grâce en *résolvant* le chemin par la méthode du même nom.

Cet objet possède un attribut `.parent` qui retourne un nouvel objet `PosixPath`, utilisé communément pour remonter l'arborescence.

```
In [2]: p = Path('.')

In [3]: p
Out[3]: PosixPath('.')

In [4]: p.resolve()
Out[4]: PosixPath('/Users/dad3zero/formation-Python_fondamentaux')

In [5]: p.parent
Out[5]: PosixPath('.')

In [6]: p.resolve().parent
Out[6]: PosixPath('/Users/dad3zero')
```

12.3.1 Fichier ou répertoire ?

L'objet `Path` possède des méthodes `.is_dir()` et `.is_file()` qui permettent de tester si le chemin correspond à un fichier ou un répertoire.

```
In [7]: [px for px in p.iterdir() if px.is_file()]
Out[7]:
[PosixPath('LICENSE'),
 PosixPath('requirements.txt'),
 PosixPath('README.md'),
 PosixPath('.gitignore')]
```

Attention, un retour `False` de ces méthodes peut signifier que le chemin n'existe pas.

12.3.2 Obtenir le contenu d'un répertoire

À partir d'un objet `Path`, il est possible d'obtenir son contenu si il s'agit d'un répertoire. Les éléments retournés seront également de type `Path` (`PosixPath` pour être plus précis).

L'objet de type `Path` permet également d'utiliser les `glob`. Dans ce cas, c'est un objet de type générateur qui est retourné.

```
In [8]: list(p.glob('p*/*.py'))
Out[8]:
[PosixPath('pylib/templating.py'),
 PosixPath('pylib/__init__.py'),
 PosixPath('pylib/datasource.py'),
 PosixPath('pylib/media_db.py')]
```

L'objet de type `Path` a un usage particulier de l'opérateur divisé `/`. Celui-ci peut être utilisé en tant que réparateur avec d'autres chaines afin de créer un nouvel objet de type `PosixPath`. Attention, cet objet nouvellement créé n'est pas un élément existant. Un objet `PosixPath` est plus proche de ce qui est manipulé par la lib `os.path`. Vous pouvez évidemment tester l'existence de cette ressource créée.

```
In [9]: new_path = p / 'stage'

In [10]: new_path
Out[10]: PosixPath('stage')

In [11]: new_path.exists()
Out[11]: True

In [12]: new_path = p / 'toto'

In [13]: new_path
Out[13]: PosixPath('toto')

In [14]: new_path.exists()
Out[14]: False
```

Cet opérateur, bien qu'utilisé, est plus une curiosité inadapté à certains cas. Il existe également une méthode `Path.joinpath(*pathsegments)` qui permet d'ajouter autant de segment que nécessaire.

```
In [15]: p.joinpath('stage', 'python', 'exercices')
Out[15]: PosixPath('stage/python/exercices')
```

Si le chemin est un fichier, vous pouvez l'ouvrir par la méthode `.open()` exactement comme la fonction que nous allons voir.

```
In [29]: my_file = (p / "README.md").open()

In [30]: my_file
Out[30]: <_io.TextIOWrapper name='README.md' mode='r' encoding='UTF-8'>
```

12.4 Ouverture et fermeture de fichier

12.4.1 Ouvrir un fichier

Pour manipuler le contenu d'un fichier avec Python, aucune dépendance n'est nécessaire. La fonction `open(path, mode)` est une fonction de base de Python (*built-in*). Elle prends

deux paramètres : le nom fichier à ouvrir et le mode. Le nom du fichier s'entends au sens UNIX, c'est à dire comprenant ou non le chemin relatif ou absolu. Le mode est une chaine de caractères optionnelle avec les options sont les suivantes.

Le mode :

— `r` : lecture seule, paramètre par défaut
— `w` : écriture, écrase un fichier existant, crée un fichier inexistant
— `x` : ouverture pour création (levée d'exception si fichier existe)
— `a` : écriture en mode ajout, écrit en fin de fichier, le crée si inexistant

Le type de fichier :

— `b` : ajouté au précédent ouverture en mode binaire
— `t` : ajouté au précédent ouverture en mode texte (défaut)

En plus…

— `+` : ajouté au précédent lecture et écriture

La fonction `open()` retourne un objet de *type fichier*.

Si vous utilisez le module `pathlib`, vous pouvez aussi passer un objet de type `Path` à la fonction `open()`. Cet objet possède méthode `.open(mode)` qui se comporte comme la fonction.

12.4.2 Fermer un fichier

Vous devez toujours prévoir de fermer un fichier ouvert. En effet, Python délègue les opérations sur les fichiers au système d'exploitation. Le système d'exploitation limite le nombre de fichiers qu'un process peut ouvrir (il est de l'ordre du millier). De plus, vous pouvez subir une perte de données en écriture principalement parce que l'écriture est asynchrone. Il est donc important de toujours refermer un fichier par la méthode `.close()`.

12.4.3 Risques d'exceptions

La manipulation d'un fichier risque toujours de lever une exception, elle se fait donc dans une clause `try`. Une bonne pratique consiste à toujours fermer le fichier dans une clause `finally`.

```
In [1]: try:
   ...:     mfile = open('showsfiles.txt', 'r')
   ...: except OSError:
   ...:     if not mfile.closed():
   ...:         mfile.close()
   ...:

In [2]: mfile
```

```
Out[2]: <_io.TextIOWrapper name='showsfiles.txt' mode='r' encoding='UTF-8'>

In [3]: mfile.close()
```

12.4.4 Usage idiomatique avec un context manager

Afin de simplifier l'écriture, nous pouvons utiliser un context manager avec le pattern `with open()`. Le context manager ferme le fichier proprement même si une exception est levée.

```
In [7]: with open('showsfiles.txt', 'r') as mfile:
   ...:     mfile.read()
   ...:
```

Nous allons dans cette partie traiter uniquement des fichiers texte. Depuis Python 3, les fichiers text sont de type `_io.TextIOWrapper`.

12.5 Lire et écrire dans un fichier

Nous avons plusieurs méthodes pour lire et écrire dans un fichier.

12.5.1 Les méthodes d'écriture

Pour écrire dans un fichier, le fichier doit être ouvert en écriture. L'objet fichier propose les méthodes suivantes :

- `write(str)` : écrit la chaine `str` dans le fichier et retourne le nombre de caractères écrits.
- `writelines(sequences)` : écrit à la suite dans le fichier les chaines de la liste passée en paramètre.

Attention, aucun caractère n'est inséré en plus du paramètre. C'est donc à vous de vous assurer de la présence des caractères de fin de ligne.

12.5.2 Les méthodes de lecture

Pour lire un fichier, celui-ci doit être ouvert en lecture. L'objet fichier propose plusieurs méthodes pour lire dans un fichier :

- `read([size])` : sans argument, retourne le contenu du fichier sous forme d'une chaine de caractères. L'argument optionnel est la taille maximum lue en nombre de caractères.
- `readline([size])` : retourne le contenu du fichier jusqu'au prochain saut de ligne (inclut). Si l'argument optionnel size est renseigné, la lecture sera limitée à ce nombre de caractères.

— `readlines([hint])` : Sans argument, retourne le contenu du fichier sous la forme d'une liste de chaines de caractères correspondant aux lignes contenues dans le fichier. Le paramètre hint spécifie le nombre de caractères lus. Le nombre total de caractères de la liste ne pourra donc pas être supérieur à ce paramètre.

12.5.3 Lecture avec l'itérateur

Cependant, le plus facile et le plus efficace est d'utiliser un **itérateur** qui facilite le parcours et le traitement d'un fichier ligne à ligne. L'usage est le suivant :

```
In [1]: for line in file:
   ...:     print(line[:-1])
```

12.6 Manipuler le curseur

La position dans un fichier est définie par un curseur. Il est donc possible de manipuler le curseur sur un fichier. Pour cela, nous avons 2 méthodes :

— `tell()` : retourne la position du curseur
— `seek(offset[, whence])` : permet de déplacer le curseur de la valeur offset en fonction du paramètre `whence`. Le paramètre `whence` peut prendre trois valeurs :
— `0` ou `os.SEEK_SET` : déplace le curseur par rapport au début du fichier. C'est le comportement par défaut. Le paramètre offset est un entier de valeur 0 ou positif.
— `1` ou `os.SEEK_CUR` : déplace le curseur par rapport à la position courante du curseur. Pour un fichier texte, le paramètre offset ne peut avoir que la valeur de 0.
— `2` ou `os.SEEK_END` : déplace le curseur par rapport à la fin du fichier. Pour un fichier texte, le paramètre offset ne peut avoir que la valeur de 0.

Attention : si vous utilisez l'itérateur, depuis Python 3, le curseur est inaccessible. Ceci est justifié parce que libérateur utilise un buffer (tampon) et que donc la position du curseur ne correspond pas à la position réellement lue.

12.7 Le module pickle

Pour plus d'informations : https://docs.python.org/3/library/pickle.html

Le module pickle est le module permettant la stérialisation et désérialisation d'objets en Python. Sérialiser un objet consiste à convertir un objet en flux d'octets ce qui permet de sauvegarder une donnée et son type. Pour éviter les confusions entre *serialization*, *marshalling*, et *flattening*, on parlera de *pickling* et *unpickling*.

Le module pickle propose deux fonctions principales :

— `dump(objet, file)` : sauvegarde la donnée

— `load(file)` : charge une donnée

Le module pickle propose également deux classes :

— `Pickler(file)` qui possède une méthode `dump(object)` qui sauvegarde une donnée
— `Unpickler(file)` qui possède une méthode `load()` qui charge une donnée

L'usage est le suivant :

```
pile_name = "ma pile"
pile_value = [42, 'answer']

import pickle

f = open("pile_backup", w)
pickle.dump(pile_name, f)
pickle.dump(pile_value, f)
f.close
```

```
import pickle

f = open("pile_backup", r)
pile_name = pickle.load(f)
pile_value = pickle.load(f)
f.close
```

Comme vous pouvez le constater, il n'y a aucune sécurité sur les données serialisées et déserialisées. Ne désérialisez donc que des données dont vous êtes à 200 % sûr.

12.8 Les fichiers CSV

12.8.1 La méthode standard... et ancienne

Python propose un module permettant de gérer les fichiers CSV. Pour la documentation complète, voir : https://docs.python.org/3/library/csv.html

Le principe général de ce module est de fournir des objets permettant la lecture ou l'écriture de fichiers CSV. L'utilisation de base fera appel aux fonctions `csv.reader(csvfile)` et `csv.writer(csvfile)` qui retourneront un objet de type `csv.reader` et `csv.writer`. Un paramètre optionnel `delimiter` permet de spécifier le séparateur de colonnes et un paramètre optionnel `quotechar` permet de spécifier le caractère encadrant les chaînes. Un exemple de lecture est proposé dans le code suivant :

```
In [10]: with open('csvfile.csv') as csvfile:
    ...:     csvreader = csv.reader(csvfile, delimiter=';',
    ...:                            quotechar='"')
    ...:     for row in csvreader:
    ...:         print(row)
    ...:
```

Un exemple d'écriture est proposé dans le code suivant :

```
In [24]: with open('csvfile.csv', 'w', newline='') as csvfile:
    ...:     csvwriter = csv.writer(csvfile, delimiter=';')
    ...:     for row in list_elements:
    ...:         csvwriter.writerow(row)
    ...:
```

Un comparatif de code est disponible dans le projet de référence et vous pouvez voir que pour un traitement de base, le module `csv` apporte peu de choses, en tout cas pour la lecture.

Il peut cependant être intéressant d'utiliser la classe `DictReader` qui instancie un objet. Cet objet est un itérateur qui retourne la ligne suivante sous forme d'un dictionnaire. Sans paramètre `fieldnames`, qui permet de définir les clefs, la première ligne (l'en-tête) est utilisée pour définir les clefs.

```
In [10]: with open('csvfile.csv') as csvfile:
    ...:     csvreader = csv.DictReader(csvfile, delimiter=';',
    ...:                                quotechar='"')
    ...:     for row in csvreader:
    ...:         print(row)
    ...:
```

12.8.2 La méthode plus moderne même si complexe

Bien qu'initialement destinée à la *datascience* la bibliothèque **Pandas** est souvent utilisée pour charger des fichiers csv. Pandas retournera un objet de type `DataFrame`. Ce type d'objet est un *tableau* qui possèdes des fonctionnalités pour sélectionner ou filtrer des données.

13 Python et les bases de données

Cette partie va présenter les principes de base de la connexion aux bases de données, à l'envoi de requêtes et au traitement des données retournées. Cette partie traite des bases de données relationnelles. Les outils tel que les ORMs ne sont pas abordés.

Le principe général indépendant de la base de données est :

— Établir une connexion
— Créer un *curseur* sur cette connexion
— Attribuer une requête à ce curseur
— Récupérer les données si il y a lieu
— Fermer la connexion

13.1 Connexion à MariaDb

Cette partie illustre le principe précédent à l'aide d'une base MariaDb. Cet exemple est évidemment transposable directement à MySQL et plus généralement à tout système de gestion de bases de données (PostgreSql, Oracle…).

Nous aurons besoins d'un connecteur ou pilote. Pour MariaDb, qui est un fort de MySQL, il s'agit de celui de MySQL disponible par pip.

Le code suivant montre comment récupérer la version de MariaDb.

```
1  import MySQLdb
2
3  try:
4      conn = MySQLdb.connect(host='localhost', user='test user',
5                             passwd='test pass', db='test')
6      cursor = conn.cursor()
7      cursor.execute("SELECT VERSION()")
8      row = cursor.fetchone()
9      print('server version', row[0])
10
11 finally:
12     if conn:
13         conn.close()
```

Notez que cet exemple utiliser pour se connecter les paramètres host, user, passwd et db. Les trois premiers sont indispensables. Le dernier vous permet de vous connecter à une base de donnée. Vous pouvez l'omettre pour vous connecter au serveur de base de données en dehors de toute base. Dans ce contexte, vous pouvez effectuer les tâches d'administration tel que la création de tables.

Si vous vous connectez sur le serveur pour créer une base, pour utiliser cette base il faudra créer une nouvelle connexion. ## Connexion à SQLite

Pour les formations et l'apprentissage, il y a une base de données plus adaptée, SQLite. Ses caractéristiques sont :

- Le pilote SQLite est disponible dans l'installation standard de Python. Il n'est pas nécessaire d'installer d'autres dépendances.
- SQLite est une base de données fichier. Il n'y a pas de notion de base de donnée à laquelle se connecter ni d'utilisateurs.
- Lors de la connexion à la base, si elle n'existe pas, elle est créée.
- Il est possible de travailler avec une base en mémoire.

SQLite est une base que vous pouvez utiliser à chaque fois que vous avez besoin de gérer des données de manière relationnelle. En tant que base en mémoire, SQLite est la base à utiliser lors des tests.

Le code suivant montre comment récupérer la version de SQLite.

```
1  import sqlite3 as lite
2
3  con = lite.connect('testdb.db')
4
5  try:
6      cursor = con.cursor()
7      cursor.execute("SELECT SQLITE_VERSION()")
8      row = cursor.fetchone()
9      print('server version', row[0])
10
11 finally:
12     if con:
13         con.close()
```

Pour travailler avec une base en mémoire, il faut spécifier une chaine de caractères spécifique :

```
1  con = sqlite3.connect(":memory:")
```

13.2 Utilisation d'un Context Manager

Les connecteurs aux bases de données proposent des *Context Managers*. Celui-ci prends en charge la fermeture de la connexion. La contre-partie est que la connexion étant fermée à la sortie du bloc suivant le with, l'ensemble des traitements doit être réalisé dans ce bloc.

```
import sqlite3 as lite

con = lite.connect('testdb.db')

with con:
    cursor = con.cursor()
    cursor.execute('SELECT SQLITE_VERSION()')
    row = cursor.fetchone()
    print('server version', row[0])
```

13.3 Les requêtes

Les requêtes sont exécutées sur des curseurs. Une requête doit être une requête SQL valide.

13.3.1 Création de tables

Les requêtes de création de tables sont en principe prédéfinies. Elles consistent en une chaine de caractères que nous allons passer à la méthode execute() du curseur. La requête suivante crée une table dans la base SQLite.

```
cur = conn.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS trainings ("
            "subject TEXT NOT NULL, "
            "duration INT NOT NULL, "
            "seats INT NOT NULL")
```

13.3.2 Insertion de données : requêtes paramétrées

Les requêtes d'insertion et d'extraction nécessiteront des paramètres (insertion, filtre…). Ces paramètres ont une origine non fiable. Il est indispensable de s'assurer de la sécurité de la base de données en particulier face à l'injection SQL. Il faut alors utiliser la fonctionnalité des requêtes paramétrées. Pour cela, l'endroit de la requête SQL où un paramètre est attendu est marqué par un ?. Les paramètres attendus seront ajoutés à l'appel de la méthode exécuté.

```
title = "Python, l'essentiel"
duration = 5
seats = 12

cur.execute("INSERT INTO trainings VALUES (?, ?, ?)",
            (title, duration, seats))
```

Vous pouvez également utiliser les requêtes paramétrées avec une syntaxe de dictionnaire. Ci-dessous, un exemple avec une requête d'extraction.

```
cur.execute("SELECT * FROM trainings WHERE duration=:duration",
            {duration:5})
```

Utilisez toujours les requêtes paramétrées qui ont entre autre la fonction d'échapper tout caractère qui ferait parti d'une requête et ainsi limiter les risques d'injection SQL.

13.3.3 Extractions de données

Une requête d'extraction de données retourne des enregistrements. Un enregistrement n'est autre qu'une liste de données. Pour obtenir ces enregistrements, nous avons plusieurs méthodes à not disposition après l'exécution de la requête :

- `fetchone()` : renvoi l'enregistrement suivant ou None si il n'y en a plus.
- `fetchmany([size])` : retourne un nombre d'enregistrements correspondant au paramètre size. En cas d'itération, size doit rester constant.
- `fetchall()` : envoi tous les enregistrements correspondant à la requête sous la forme d'une liste (dans la limite du paramètre cursor.arraysize de la base de données

13.4 Les transactions

Un système de gestion de base de données qui est transactionnel doit respecter un ensemble de règles baptisé **A.C.I.D.** pour **Actomicity**, **Coherence**, **Isolation** et **Durability**.

Une transaction est un ensemble de modifications indivisible qui doit se réaliser en assurant la cohérence de la base de données en isolation des autres requêtes.

13.4.1 Illustrations par un exemple.

Soit un transfert de compte à compte où les comptes ne doivent pas basculer en négatif. Nous avons 2 écritures : un débit et un crédit. La première contrainte est donc de ne pas faire passer en négatif le compte à débiter. La démarche intuitive est de vérifier si le solde est suffisant et si oui, débiter le compte. Cependant, cela n'empêche pas un débit entre la consultation et le retrait effectif conduisant à un état incohérent de la base.

La bonne démarche consiste donc à effectuer l'écriture de débit puis à vérifier que le solde est toujours positif ou nul avant de continuer. Si ce n'est pas le cas, alors on informera la base d'annuler ces requêtes, c'est à dire cette transaction. Le système reviendra dans son état initial. Parce que la base gère les transactions en isolation (tant qu'elles ne sont pas validées, elles sont invisibles de l'extérieur), cette opération est sans conséquences sur les autres requêtes.

13.4.2 Mécanisme des transactions

Du point de vue de la base de données, une transaction débute par une commande BEGIN et se poursuit jusque'à une commande COMMIT qui valide l'ensemble des modifications ou ROLLBACK qui annule l'ensemble des modifications.

Du point de vue Python, la transaction va débuter avec la première requête de modification et se poursuit jusqu'à l'appel de la méthode commit() ou de la méthode rollback().

Les enregistrements concernés seront verrouillés le temps de la transaction. Il est donc important d'exécuter une transaction *aussi vite que possible*. Pour éviter un verrou de la base, celles-ci ont en général un mécanisme de *Deadlock* qui annule une transaction en cas d'inactivité après un certain délais. Dans le cas de SQLite, ce délais est de 5 secondes par défaut.

13.4.3 Transactions et exceptions

Transactions et exceptions sont deux mécanismes distincts. Il appartient au développeur de réaliser un rollback suite à une exception ou de déclencher une exception après un rollback. Si nécessaire, l'un et l'autre ne sont pas systématiques.

En reprenant l'exemple du compte bancaire, si le compte débité est négatif, le rollback annulera la transaction mais aucune information ne remonte au code appelant. Il sera nécessaire de lever une exception.

13.4.4 Les context managers et les transactions

Les objects connexion peuvent être utilisés comme des context managers si le pilote le permet (c'est le cas de SQLite ou PostgreSQL). Celui-ci déclenche automatiquement les commit et rollback. Dans le cas où une exception se produit, le contexte manager provoquera un rollback. Une sortie *normale* du context manager provoquera un commit.

En sortie du context manager, l'exception est toujours levée, c'est donc l'information que la transaction ne s'est pas achevée normalement. Elle doit donc être capturée.

Des exemples d'illustrations vous seront montrés en formation sur des exemples d'illustration.

13.5 Organisation du code

L'organisation du code (connexion, exécution des requêtes, traitement des données…) dépendra évidemment de votre application. Une manière d'organiser son code de connexion à une base de données est de créer une classe de type *dao*. Vous avez quelques exemples dans des modules du package `training.projects` (les modules se terminant par _db). Le principe général proposé dans ce code consiste à :

— Établir la connexion dans le constructeur. Pour une connexion à une base SQLite, les tables sont également créées dans le constructeur.
— Le destructeur s'occupe de fermer la connexion.
— Des méthodes encapsulent les requêtes et le chargement des données.

14 Les dates en Python

Travailler avec les dates est systématiquement compliqué. Tous les langages proposent une bibliothèque pour simplifier la tâche.

La distribution standard Python fournit le module `datetime`. Celui-ci contient fondamentalement deux classes qui permettent de représenter les concepts fondamentaux :

— `datetime` qui permet de représenter la notion d'instant
— `timedelta` qui permet de représenter la notion de durée

14.1 Création d'un instant avec la classe datetime

Les paramètres du constructeur sont : l'année, le mois, le jour, l'heure, les minutes, les secondes, les microsondes et le *timezone*. Les paramètres autre que le timezone sont des entiers et seuls l'année, mois et jour sont nécessaires.

La gestion des fuseaux horaires est assez complexe, elle sera abordée dans une partie dédiée.

```
In [1]: from datetime import datetime

In [2]: datetime(2015, 12, 16)
Out[2]: datetime.datetime(2015, 12, 16, 0, 0)
```

L'objet créé avec uniquement l'année, mois et jour est donc un instant à la date donnée à minuit.

La classe `datetime` contient des méthodes de classe pour créer un instant autrement que par le passage de tous les paramètres. Ainsi, pour représenter l'instant présent, nous avons la méthode `now()`.

```
In [3]: datetime.now()
Out[3]: datetime.datetime(2015, 12, 16, 14, 30, 13, 835376)
```

Notez qu'il existe également une méthode `utcnow()` qui crée un objet `datetime` en UTC. Pour cela, l'interpréteur se reposera sur les informations du système. L'objet `datetime` créé sera naïf.

```
In [3]: datetime.now()
Out[3]: datetime.datetime(2015, 12, 16, 14, 30, 13, 835376)
In [4]: datetime.utcnow()
Out[4]: datetime.datetime(2015, 12, 16, 13, 30, 13, 835376)
```

Consultez la documentation pour les autres méthodes de classe.

14.2 Objet `datetime` et chaines de caractères

Pour convertir une chaine de caractères représentant une date en objet de type `datetime` ou inversement, la classe `datetime` dispose de deux méthodes : `.strptime(date, format)` et `.strftime(format)`.

Le paramètre `format` est une chaine de caractère décrivant le format à lire ou à produire. Sa syntaxe est décrite dans la documentation :

https://docs.python.org/3.12/library/datetime.html#strftime-strptime-behavior

Pour lire une date à partir d'une chaine de caractères et générer un objet `datetime`, nous allons utiliser la méthode `strptime(string, format)`.

```
In [4]: date_object = datetime.strptime('Jun 1 2005  1:33PM',
                                        '%b %d %Y %I:%M%p')
```

Pour créer une chaine de caractères représentant un objet `datetime`, nous utiliserons la méthode `strftime(format)`.

```
In [5]: date_object.strftime('%Y-%m-%d %H:%M:%S.%f')
Out[5]: '2005-06-01 13:33:00.000000'
```

Par défaut, la date est en anglais. Pour travailler avec les informations de date en français, il faut modifier la locale :

```
In [6]: date_object.strftime('%a %d %B %Y')
Out[6]: 'Wed 01 June 2005'

In [7]: import locale

In [8]: locale.setlocale(locale.LC_TIME, 'fr_FR')
Out[8]: 'fr_FR'

In [9]: date_object.strftime('%a %d %B %Y')
Out[9]: 'Mer 01 juin 2005'
```

14.3 Les durées

Travailler avec des dates, c'est travailler avec les différences entre ces dates soit des durées. Elles sont représentées par des objets `datetime.timedelta`. La différence entre deux instants permet d'obtenir une durée.

```
In [10]: duration = datetime(2017, 12, 15) - datetime(2015, 12, 16)

In [11]: print(duration)
730 days, 0:00:00

In [12]: duration
Out[12]: datetime.timedelta(730)
```

On peut évidemment instancier un objet de type `timedelta` et ainsi déterminer des instants à partir d'autres instants.

```
In [13]: cuisson_oeuf = timedelta(seconds=180)

In [14]: datetime.now() + cuisson_oeuf
Out[14]: datetime.datetime(2017, 9, 10, 12, 12, 52, 776483)
```

Des opérations mathématiques sont donc possibles entre des objets de type `datetime` (soustraction), `timedelta` (addition et soustraction), `datetime` et `timedelta` (addition et soustraction) et `timedelta` et des valeurs numériques (multiplication et division).

Les objets de type `timedelta` ne sont pas adaptés à la représentation des durées (conversion en chaines de caractères human-friendly). Elles ne proposent pas de méthodes de parsing ou de formatage de la donnée. L'information de durée ne peut être extraite qu'en jour, secondes et microsecondes.

14.4 Jours et heures

Le module `datetime` propose des objets de type `datetime.date`et `datetime.time` pour représenter une date (jour) et un instant dans une journée. Il est possible soit d'instancier spécifiquement ces données, soit de les extraire d'un objet de type `datetime`.

```
In [15]: ep8 = datetime(2017, 12, 15)

In [16]: ep8.date()
Out[16]: datetime.date(2017, 12, 15)

In [17]: from datetime import time
```

```
In [18]: show_time = time(20, 30)

In [19]: print(show_time)
20:30:00
```

14.5 Le temps : des objets immuables

Les informations liées au temps sont des données critiques. Afin de ne pas les altérer par accident, les objets sont immuables, ils ne peuvent pas être modifiés. La méthode `replace()` de la classe `datetime` permet de modifier un attribut de l'objet mais celle-ci retourne une copie de l'objet.

```
In [20]: ep8.replace(year=2019, day=20)
Out[20]: datetime.datetime(2019, 12, 20, 0, 0)

In [21]: print(ep8)
2017-12-15 00:00:00
```

Si vous souhaitez modifier une donnée, il faut donc le faire explicitement.

14.6 Les fuseaux horaires

Jusqu'ici, nous avons utilisé des objets `datetime` sans fuseau horaire. Nous parlons alors de dates *naïves*. Des dates naïves ne peuvent pas réellement être comparées entre elles. La notion de *timezone* ou fuseau horaire, c'est le décalage de date en fonction du lieu géographique. Mais il y a également les notions politique dont le plus simple est le changement d'heure. Il faut donc ajouter la notion de fuseau horaire et le gérer.

14.6.1 Attribuer une timezone

Pour rendre les dates *aware* (traduit par *avisées*), il faut leur associer un *timezone*. Python propose depuis la version 3.9 une implémentation concrète de la base IANA issue de la PEP 615 : zoneinfo.

Note : avant Python 3.9, il fallait passer par des projets tiers et en particulier pytz. Ce projet est toujours maintenu (en 2026) pour fournir le support concret aux codes utilisant une version de Python antérieur à la 3.9.

14.6.2 Créer des dates *aware*

La première étape est de créer un objet `ZoneInfo`.

```
from zoneinfo import ZoneInfo

paris_tz = ZoneInfo('Europe/Paris')
new_york_tz = ZoneInfo('America/New_York')
```

Pour connaitre les arguments possibles, `zoneinfo` fournit une fonction `available_timezones()` qui en retourne la liste.

Nous pouvons maintenant créer une date *aware*. Il y a pour cela 2 possibilités :

— assigner le timezone à la création
— assigner le timezone à une date *naïve*.

Dans le code suivant, la ligne 3 crée une date localisée, la ligne 6 localise une date naïve créée précédemment.

```
from datetime import datetime, timedelta

summer_time = datetime(2021, 10, 30, tzinfo=paris_tz)

naive_summer_time = datetime(2021, 10, 30)
summer_time = naive_summer_time.replace(tzinfo=paris_tz)
```

14.6.3 Re-localiser une date

Si une date *avisée* est *localisée* dans un timezone. Pour la *convertir* à un autre timezone, il y a la méthode `.astimezone()`.

```
meeting_fr = datetime(2021, 3, 29, 9, tzinfo=paris_tz)
meeting_us = meeting_fr.astimezone(new_york_tz)
print(meeting_us) #  affiche 2021-03-29 03:00:00-04:00
```

14.6.4 Changement d'heure : une gestion calendaire

Python réalise ses calculs de temps entre dates avisées de manière calendaire :

```
>>> summer_time = datetime(2026, 10, 24, tzinfo=paris_tz)

>>> winter_time = datetime(2026, 10, 26, tzinfo=paris_tz)

# L'été est bien en UTC+2
>>> print(summer_time)
2026-10-24 00:00:00+02:00

# L'hiver est bien en UTC+1
>>> print(winter_time)
2026-10-26 00:00:00+01:00

# 2 jours calendaires sont ajoutés et le timezone adapté
>>> print(summer_time + timedelta(days=2))
2026-10-26 00:00:00+01:00

# La différence est calendaire et non absolue
>>> print(winter_time - summer_time)
2 days, 0:00:00
```

14.6.5 Calcul des durées réelles

Pour calculer les durées réelles, il faut passer par une référence : UTC.

```
>>> utc_tz = ZoneInfo("UTC")

>>> summer_time_utc = summer_time.astimezone(utc_tz)
>>> winter_time_utc = winter_time.astimezone(utc_tz)

>>> print(summer_time_utc)
2026-10-23 22:00:00+00:00

>>> print(winter_time_utc)
2026-10-25 23:00:00+00:00

# La différence est la durée réelle
>>> print(winter_time_utc - summer_time_utc)
2 days, 1:00:00

# La nouvelle date est à 48 heures réels
>>> print(summer_time_utc + timedelta(days=2))
2026-10-25 22:00:00+00:00

# Et pour finir, une illustration de la relocalisation
```

```
>>> print((summer_time_utc + timedelta(days=2)).astimezone(paris_tz))
2026-10-25 23:00:00+01:00
```

14.7 Le calendrier

14.7.1 Des fonctions et constantes utilitaires

Le module `calendar` offre des fonctionnalités pour gérer un calendrier ou une date dans un calendrier. Le module propose avant tout un certain nombre de constantes et de fonction afin d'accéder à des données usuelles.

```
In [22]: import calendar
In [23]: calendar.EPOCH
Out[23]: 1970
In [24]: calendar.MONDAY
Out[24]: 0
In [25]: calendar.TUESDAY
Out[25]: 1
In [26]: calendar.January
Out[26]: 1
```

Vous pouvez voir avec le code ci-dessus que pour Python, lundi est représenté par la valeur 0 et janvier par 1. Les constantes proposées par le module sont donc utiles pour éviter toute confusion.

Le module `calendar` propose également des constantes et méthodes liées aux dates ou aux mois :

```
In [27]: calendar.mdays
Out[27]: [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

In [28]: calendar.isleap(2018) # annee bissextile
Out[28]: False

In [29]: calendar.leapdays(2000, 2018)
Out[29]: 5

In [30]: calendar.weekday(2017, 12, 13)
Out[30]: 2
```

Le module `calendar` intègre également de quoi mettre en forme des informations de calendrier.

```
In [31]: calendar.prmonth(2017, 12)
   décembre 2017
Lu Ma Me Je Ve Sa Di
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Ces fonctions sont similaires à l'instruction `cal` du bash. L'instruction ci-dessus est équivalente à `print(calendar.month(2017, 12))`.

14.7.2 La classe Calendar

Le module `calendar` permet également de manipuler un calendrier sous forme d'objet, il propose en effet une classe `Calendar`. Celle-ci prend un paramètre qui est le premier jour de la semaine, par défaut c'est 0 soit lundi.

Cette classe fournit surtout des données liées au formatage. Elle ne fait pas de formatage en elle même, ceci est à déléguer aux classes spécialisées.

```
In [43]: cal = calendar.Calendar()

In [44]: print(list(cal.itermonthdays(2020, 3)))
[0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

In [45]: print(list(cal.itermonthdays2(2020, 3)))
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 6), (2, 0), (3, 1), (4, 2), (5, 3), (

In [46]: print(list(cal.itermonthdates(2020, 3)))
[datetime.date(2020, 2, 24), datetime.date(2020, 2, 25), datetime.date(2020, 2, 26), datet
```

15 Les expressions rationnelles

Une expression rationnelle, expressions régulières (anglicisme de *Regular Expressions*) ou simplement **regex**, décrit la forme d'un texte recherché.

Elle sert notamment à :

- rechercher une information dans un texte ;
- vérifier qu'une valeur respecte un format ;
- extraire ou remplacer les parties reconnues.

Une regex devient utile quand une partie peut varier :

```
CMD-2048    CMD-0007    CMD-9312
    même forme, chiffres différents
```

Utiliser les regex consiste avant tout à définir un motif décrivant la chaine de caractères recherchée. C'est la partie la plus complexe. Ce document n'est qu'une introduction à la définition de ces motifs.

En Python, les expression rationnelles sont gérées par le module re

15.1 Le vocabulaire

```
texte : "Commande CMD-2048 expédiée"
motif : r"CMD-\d{4}"
                    correspondance : "CMD-2048"
```

- Le **texte** contient les données à examiner.
- Le **motif** décrit ce que l'on cherche.
- Une **correspondance** est une portion du texte reconnue par le motif.

Un motif combine des caractères ordinaires, comme `CMD-`, et des symboles spéciaux, comme `\d`.

15.2 Définir un motif

Utiliser les RegEx consiste avant tout à définir un motif décrivant la chaine de caractères à rechercher ou valider. Ce motif est lui-même une chaine de caractères. Le moteur de RegEx cherchera à identifier ce motif dans une chaine de caractères.

La suite de ce document présente une introduction à la définition d'un motif. La documentation citée en référence est bien plus exhaustive.

15.2.1 Un motif de base

Le motif le plus simple est une chaine de caractères strict. `'CMD-'` par exemple peut être sera identifié dans la chaine de caractères `'CMD-'` mais aussi `'CMD-2048'` ou `'Commande CMD-2048'`.

15.2.2 Les classes de caractères

Une classe représente **un caractère parmi plusieurs possibilités**.

Motif	Signification	Exemple reconnu
`[abc]`	`a`, `b` ou `c`	`b`
`[A-Z]`	une capitale de `A` à `Z`	`M`
`\d`	un chiffre	`7`
`\w`	une lettre, un chiffre ou `_`	`p`
`\s`	un espace, une tabulation ou un saut de ligne	espace
`[^0-9]`	un caractère qui n'est pas un chiffre	`X`

`\D`, `\W` et `\S` sont les formes inverses de `\d`, `\w` et `\s`.

Le point `'.'` représente presque n'importe quel caractère.

15.2.3 Nombre d'occurrences

Un quantificateur indique combien de fois l'élément précédent peut apparaître.

Motif	Nombre attendu	Exemple
`\d?`	zéro ou une fois	`7` ou rien
`\d*`	zéro fois ou plus	`123` ou rien
`\d+`	une fois ou plus	`123`
`\d{4}`	exactement 4 fois	`2048`
`\d{2,4}`	de 2 à 4 fois	`20`, `204`, `2048`

```
CMD-\d{4}
       exactement quatre chiffres
      texte attendu tel quel
```

15.2.4 La position d'un motif

Les ancres indiquent **où** le motif doit se trouver.

— `^` en début de motif représente le début de la chaîne.
— `$` en fin de motif représente la fin de la chaîne.

```
pattern = r"^CMD-\d{4}$"

bool(re.search(pattern, "CMD-2048"))           # True
bool(re.search(pattern, "Commande CMD-2048"))  # False
```

15.2.5 Les groupes de caractères

Les parenthèses regroupent plusieurs éléments :

```
re.fullmatch(r"(ha)+", "hahaha")  # correspondance
```

Le symbole `|` signifie « ou ».

```
re.fullmatch(r"oui|non", "oui")  # correspondance
```

15.2.6 Référence précédente

À l'aide d'un metacaractère appelé *backreference*, nous pouvons matcher un groupe capturé précédemment. Ce metacaractère s'écrit `\n` où `n` est un entier qui représente le match du nième groupe capturé.

Ce metacaractère permet d'éviter le problème avec un pattern comme `([rf]i)+` qui capture `'riri'`, `'fifi'` mais aussi `'rifi'` et `'firi'`.

Pour identifier spécifiquement `'riri'` ou `'fifi'` mais ni `'rifi'` ou `'firi'`, le pattern est `([rf]i)\1`.

En passant, pour ajouter `'loulou'` aux matchs possibles, nous pouvons ajouter le ou : `([rf]i|lou)\1`.

15.3 Utiliser le module re

Pour utiliser les RegEx, Python fournit le module `re`.

15.3.1 Les fonctions de recherche

Le module `re` propose les fonctions de recherche suivantes :

Fonction	Zone examinée
`re.search(motif, texte)`	n'importe où dans le texte
`re.match(motif, texte)`	uniquement au début
`re.fullmatch(motif, texte)`	toute la chaîne

En pratique, utilisez `search()` pour trouver et `fullmatch()` pour valider.

`re.search()` retournera la première occurence trouvée. Pour recherobtenir toutes les occurences, vous avez :

Fonction	Renvoie
`re.findall(motif, texte)`	La liste des chaines de caractères trouvées
`re.finditer(motif, texte)`	Un itérateur sur des *Match Objects*

Une RegEx peut également être utilisée comme un objet. Un motif utilisé plusieurs fois peut être compilé :

```
1 cmd_ref = re.compile(r"CMD-\d{4}")
2
3 cmd_ref.search("Commande CMD-2048")
4 cmd_ref.fullmatch("CMD-0007")
```

La variable `cmd_ref` regroupe le motif et ses options. L'objet propose les mêmes méthodes de recherche que les fonctions du module.

Notez qu'un motif compilé n'est pas plus performant qu'utiliser la fonction. Si la *compilation optimise* la recherche, il en va de même au niveau du module pour les fonctions.

15.4 Les Match Objects et leurs attributs

Un Match Object expose principalement les 3 méthodes suivantes. Le paramètre optionnel `subgroup` est un entier ou une chaine de caractères.

- `group([subgroup])` : utilisé sans paramètre ou avec la valeur `0`, retourne le motif trouvé, sinon le motif correspondant au sous-groupe.
- `start([subgroup])` : utilisé sans paramètre ou avec la valeur `0`, retourne l'indice de début du motif trouvé, sinon celui correspondant au sous-groupe.
- `end([subgroup])` : utilisé sans paramètre ou avec la valeur `0`, retourne l'indice de fin du motif trouvé, sinon celui correspondant au sous-groupe.

Un *sous-groupe* représente un groupe défini par des parenthèses.

15.5 Les sous-groupes

```
texte = "Livraison prévue le 07/11/2026"
pattern = r"(\d{2})/(\d{2})/(\d{4})"

m = re.search(motif, texte)

if m:
    print(m.group(0))  # 07/11/2026 : motif complet
    print(m.group(1))  # 07         : premier groupe
    print(m.groups())  # ('07', '11', '2026')
```

La seconde fonction des groupes est donc d'isoler des parties d'intérêt.

Utiliser un nom rend l'extraction plus explicite qu'un numéro. Nous définissons une étiquette grâce à la syntaxe `(?P<name>…)`.

```
pattern = (r"(?P<jour>\d{2})/(?P<mois>\d{2})/(?P<annee>\d{4})")

m = re.search(pattern, "Livraison le 07/11/2026")

if m:
    print(m.group("annee"))  # 2026
    print(m.groupdict())     # {'jour': '07', 'mois': '11', 'annee': '2026'}
```

15.5.1 Groupes non-capturants

Quand un groupe sert seulement à structurer le motif, `(?:...)` évite de le capturer :

```python
pattern = r"(?:M|Mme) Dupont"
```

`(?:M|Mme)` choisit une alternative sans créer de groupe à extraire.

15.6 Les substitutions

Les RegEx peuvent être utilisées pour substituer du texte grâce à :

- La fonction `sub(pattern, repl, string, count=0, flags=0)`
- la méthode `sub(repl, string, count=0, flags=0)`

Les paramètres sont :

- `repl` peut-être soit une chaine de substitution, soit un *callable*, c'est à dire une fonction. Celle-ci prendra en paramètre un match object et devra retourner une chaine de caractères.
- `count` permet de limiter le nombre de substitutions.

Le code suivant illustre la substitution avec une chaine de caractères fixe.

```python
ma_chaine = 'pi vaut 3.14 et e vaut 2.72'

float_regex.sub('quelque chose', ma_chaine)
# Retourne : 'pi vaut quelque chose et e vaut quelque chose'
```

Le code suivant illustre une substitution plus intelligente avec une fonction.

```python
temp_pattern = r"(?P<temp>[0-9]+\.[0-9]{2})°(?P<value>c|F)"

t_regex = re.compile(temp_pattern)

def simplify(temp_match):
    temp = float(temp_match.group('temp'))
    if temp > 30:
        return "Trop chaud"
    elif temp < 15:
        return "Trop froid"
    else:
        return temp_match.group()

t_regex.sub(simplify, r"Il fait 17.08°c")
# Retourne: 'Il fait 17.08°c'

t_regex.sub(simplify, r"Il fait 32.08°c")
# Retourne: 'Il fait Trop chaud'
```

```
19
20 t_regex.sub(simplify, r"Il fait 12.08°c")
21 # Retourne: 'Il fait Trop froid'
```