Les librairies

Introduction

Une bibliothèque, ou librairie, en Python désigne un ensemble cohérent de modules préécrits destinés à étendre les fonctionnalités du langage sans qu'il soit nécessaire de tout redévelopper soi-même.

Chaque bibliothèque regroupe des fonctions, des classes et parfois des constantes permettant de résoudre des problèmes spécifiques, tels que le calcul scientifique, la manipulation de fichiers, le traitement de données ou le développement web, comme nous l'avons défini dans le précédents chapitre.

Python repose sur un écosystème très vaste : à côté de sa bibliothèque standard intégrée, qui couvre les besoins fondamentaux (comme `os`, `math`, `datetime`, `json` ou `csv`), des milliers de bibliothèques tierces sont disponibles sur le dépôt officiel *PyPI* (Python Package Index), librement installables par les utilisateurs.

L'existence de ces bibliothèques participe à la popularité du langage. Elle transforme Python en un véritable environnement modulaire : plutôt que d'être limité à un domaine particulier, il peut s'adapter à la quasi-totalité des usages modernes.

Par exemple, on trouve `pandas` et `numpy` pour l'analyse et le calcul numérique, `matplotlib` pour la visualisation, `Flask` ou `Django` pour le développement web, `requests` pour les communications HTTP, `pytest` pour les tests, ou encore `pytz` et `dateutil` pour la manipulation avancée du temps.

Chacune de ces bibliothèques a été conçue pour répondre à un besoin précis, en respectant la philosophie de clarté et de simplicité du langage.

Sur le plan technique, une bibliothèque n'est rien d'autre qu'un ensemble de modules Python, c'est-à-dire de fichiers `.py` structurés dans un répertoire muni d'un fichier `__init__.py` signalant qu'il s'agit d'un package. Lorsqu'on écrit `import math` ou `from datetime import date`, Python charge dynamiquement le module concerné dans la mémoire de l'interpréteur et rend accessibles ses fonctions et classes. Ce mécanisme d'importation, accompagné d'un système de gestion de dépendances (aujourd'hui géré efficacement par UV ou pip), permet de composer librement ses outils pour bâtir des programmes robustes et évolutifs.

Ainsi, comprendre les bibliothèques Python revient à comprendre la logique d'un écosystème dans lequel tout est modulaire : chaque module est une brique que l'on assemble pour répondre à un besoin précis, et la bibliothèque constitue la collection organisée de ces briques, prêtes à être réutilisées et partagées.

Pypi.org

Le site **PyPI.org**, abréviation de *Python Package Index*, est le répertoire officiel où sont stockées toutes les bibliothèques Python disponibles publiquement. C'est un peu comme un magasin d'applications, mais pour du code Python. Il joue le même rôle que *npm* pour JavaScript ou *Maven Central* pour Java. Sur PyPI, les développeurs du monde entier peuvent publier et partager leurs bibliothèques (qu'on appelle aussi "paquets" ou "packages"). Chaque paquet contient du code réutilisable, de la documentation, et la liste des autres bibliothèques dont il a besoin pour fonctionner.

Techniquement, PyPI est l'endroit où les outils d'installation comme **pip** ou **uv** vont chercher les bibliothèques quand on veut les installer. Par exemple, quand on tape la commande `uv add requests` ou `pip install requests`, voici ce qui se passe en coulisses : l'outil contacte PyPI.org, cherche la dernière version du paquet *requests*, télécharge un fichier compressé (généralement au format `.whl` ou `.tar.gz`), puis installe les fichiers dans l'environnement virtuel du projet. Tout cela se fait automatiquement. PyPI est maintenu par la Python Software Foundation, ce qui garantit que les bibliothèques sont fiables et sécurisées.

Chaque paquet publié sur PyPI possède une **page d'information** qu'on peut consulter sur le site. On y trouve le nom du paquet, sa version actuelle, les versions de Python avec lesquelles il est compatible, les autres bibliothèques dont il dépend, une description de ce qu'il fait, la commande pour l'installer, et souvent un lien vers son code source sur GitHub. Ces informations viennent du fichier `pyproject.toml` ou `setup.cfg` du projet. Cela permet de comprendre rapidement à quoi sert un paquet et de vérifier qu'il est bien maintenu avant de l'installer.

PyPI est donc bien plus qu'une simple liste de bibliothèques : c'est un système qui permet de distribuer et de partager du code Python à grande échelle. Il facilite la réutilisation du code, encourage la collaboration entre développeurs, et garantit que tout le monde utilise le même système de distribution. Grâce à PyPI, Python a pu construire un écosystème très riche, où chaque développeur peut contribuer en publiant ses propres bibliothèques et en faire profiter la communauté entière, le tout directement depuis la ligne de commande.

Pathlib

La bibliothèque **pathlib**, introduite officiellement dans la bibliothèque standard de Python à partir de la version 3.4, a profondément renouvelé la manière de manipuler les chemins de fichiers et de répertoires.

Avant son apparition, la gestion des chemins reposait sur le module `os` et ses sous-modules `os.path`, qui traitaient les chemins comme de simples chaînes de caractères. Cela entraînait souvent des erreurs de concaténation, des différences de séparateurs entre systèmes d'exploitation et un manque de lisibilité dans les scripts. Avec `pathlib`, Python adopte une approche orientée objet qui transforme les chemins en véritables objets manipulables, cohérents et portables.

L'élément central de cette bibliothèque est la classe **`Path`**. Elle représente un chemin de fichier ou de dossier comme un objet doté de méthodes et d'attributs spécifiques. Lorsqu'on instancie un objet `Path`, celui-ci devient capable de décrire, interroger et manipuler l'arborescence du système de fichiers de manière naturelle. Par exemple :

```
from pathlib import Path

chemin = Path("/Users/Jean/Documents/rapport.txt")
print(chemin.name)        # rapport.txt
print(chemin.parent)      # /Users/Jean/Documents
print(chemin.suffix)      # .txt
print(chemin.stem)        # rapport
```

Ces propriétés permettent d'obtenir immédiatement les différentes composantes d'un chemin sans avoir à recourir à des découpages de chaînes. L'objet `Path` ne se contente pas de représenter un chemin : il fournit

également des **méthodes utilitaires** puissantes pour interagir avec le système de fichiers. On peut vérifier l'existence d'un fichier avec `exists()`, tester s'il s'agit d'un répertoire avec `is_dir()`, ou créer des dossiers avec `mkdir()`. On peut également lire et écrire des fichiers en une seule ligne grâce à `read_text()` et `write_text()` :

```
chemin = Path("notes.txt")
chemin.write_text("Introduction à pathlib")
print(chemin.read_text())
```

Le grand avantage de `pathlib` est sa **portabilité**. L'objet `Path` adapte automatiquement le format du chemin à la plateforme sous-jacente : `PosixPath` pour macOS et Linux, `WindowsPath` pour Windows. Ainsi, le même code s'exécute sans modification sur différents systèmes, en respectant les conventions de chacun (barres obliques `/` ou antislash `\`). De plus, `pathlib` s'intègre parfaitement avec les modules de la bibliothèque standard : on peut, par exemple, combiner `Path` avec `open()`, `json`, `csv` ou `shutil` sans conversion préalable.

L'un des aspects les plus pratiques de `Path` est qu'il unifie la création, la lecture et la manipulation des fichiers sous une syntaxe fluide. On peut ainsi construire des chemins à l'aide de l'opérateur `/`, créer des répertoires ou des fichiers, parcourir des dossiers, ou encore modifier des extensions de manière expressive.

Considérons le code suivant, qui illustre ces usages fondamentaux :

```
from pathlib import Path

# Créer un répertoire images
images_dir = Path("images")
images_dir.mkdir(exist_ok=True) # Créer le répertoire si il n'existe pas
sinon il ne fait rien
print("Répertoire 'images' créé.")

# Créer un fichier image.jpg dans le répertoire images
image_file = images_dir / "image.jpg"
image_file.touch()
print("Fichier 'image.jpg' créé dans le répertoire 'images'.")

# Changer l'extension de image.jpg en image.png
image_png = image_file.with_suffix(".png")
image_file.rename(image_png)
print(f"L'extension du fichier a été modifiée en : {image_png}")

# Lister tous les fichiers dans le répertoire images
print("Fichiers dans le répertoire 'images' :")
for fichier in images_dir.iterdir():
    if fichier.is_file():
        print(fichier.name)

# Vérifier si nouveau_nom.txt existe dans le répertoire courant
fichier_nouveau = Path("nouveau_nom.txt")
```

```
if fichier_nouveau.exists():
    print(f"Le fichier {fichier_nouveau} existe.")
else:
    print(f"Le fichier {fichier_nouveau} n'existe pas.")

# Créer un répertoire documents et ajouter des fichiers .txt
documents_dir = Path("documents")
documents_dir.mkdir(exist_ok=True)

# Créer quelques fichiers .txt
(documents_dir / "file1.txt").touch()
(documents_dir / "file2.txt").touch()
(documents_dir / "file3.txt").touch()

# Lister tous les fichiers .txt dans le répertoire documents
print("Fichiers .txt dans le répertoire 'documents' :")
for fichier in documents_dir.glob("*.txt"):
    print(fichier.name)
```

Dans cet exemple, `Path("images")` crée un objet représentant un répertoire virtuel nommé *images*.

La méthode `mkdir()` le crée effectivement sur le disque, tandis que `exist_ok=True` évite une erreur si le dossier existe déjà.

L'opérateur `/` permet de composer un chemin en combinant naturellement des segments, ici `images_dir / "image.jpg"`.

La méthode `touch()` crée un fichier vide, et `with_suffix()` génère un nouveau chemin identique, mais avec une extension différente, avant que `rename()` ne le renomme physiquement.

Les méthodes `iterdir()` et `glob()` permettent de parcourir les fichiers d'un répertoire.

`iterdir()` renvoie tous les éléments (fichiers et sous-dossiers), tandis que `glob("*.txt")` applique un filtrage selon un motif. L'appel à `exists()` permet de vérifier la présence d'un fichier avant toute opération, renforçant la robustesse du code.

Par cette approche, `pathlib` offre une syntaxe expressive et sécurisée, transformant les manipulations de fichiers en opérations naturelles, intuitives et indépendantes du système sous-jacent. On ne travaille plus avec des chaînes de caractères mais avec des objets cohérents, capables de se combiner, de s'interroger et d'agir directement sur le système de fichiers.

Pendulum

La bibliothèque **Pendulum** constitue une alternative moderne et élégante au module standard `datetime`. Conçue pour corriger ses faiblesses et en simplifier l'usage, elle offre une interface plus intuitive, un comportement cohérent face aux fuseaux horaires, ainsi qu'une syntaxe naturelle pour les calculs et les formats de dates. Là où `datetime` requiert souvent la combinaison de plusieurs modules (`pytz`, `dateutil`, `calendar`), Pendulum réunit toutes ces fonctionnalités dans une seule librairie, tout en garantissant une compatibilité totale avec les objets `datetime` natifs de Python.

```
uv add pendulum
```

L'un des atouts majeurs de Pendulum réside dans sa gestion automatique des fuseaux horaires. Lorsqu'on crée un objet de date, il est *timezone-aware* par défaut, c'est-à-dire qu'il contient déjà une information de fuseau, ce qui évite les erreurs fréquentes de conversion ou de comparaison entre instants localisés différemment. De plus, la création, la manipulation et la mise en forme des dates y deviennent beaucoup plus expressives. Par exemple :

```python
import pendulum

# Obtenir la date et l'heure actuelles
maintenant = pendulum.now()
print("Heure locale :", maintenant)

# Spécifier un fuseau horaire
paris = pendulum.now("Europe/Paris")
tokyo = pendulum.now("Asia/Tokyo")

print("Heure à Paris :", paris)
print("Heure à Tokyo :", tokyo)

# Créer une date spécifique
evenement = pendulum.datetime(2025, 10, 19, 14, 30, tz="Europe/Paris")
print("Événement :", evenement)

# Ajouter ou soustraire du temps
print("Dans 10 jours :", evenement.add(days=10))
print("Il y a 3 heures :", evenement.subtract(hours=3))

# Calculer la différence entre deux dates
diff = evenement.diff(paris)
print("Différence :", diff.in_days(), "jours")

# Afficher une durée dans un format humain
print("Événement dans :", evenement.diff_for_humans())
```

Dans cet exemple, chaque opération se lit presque comme une phrase : `add(days=10)` ou `diff_for_humans()` expriment directement l'intention. La méthode `diff_for_humans()` illustre bien cette approche orientée lisibilité : elle renvoie une durée exprimée dans un langage naturel, tel que *"il y a 2 heures"* ou *"dans 3 jours"*, ce qui s'avère extrêmement utile pour des applications affichant du temps relatif.

Pendulum excelle aussi dans la **précision et la cohérence des calculs calendaires**. Contrairement à `datetime`, il gère correctement les mois, années, transitions d'heure d'été, et conversions entre zones. La méthode `add(months=1)` ajoute réellement un mois calendaire, et non trente jours arbitraires. De plus, il introduit des objets spécialisés comme `Duration` ou `Period` qui permettent de raisonner sur des intervalles de temps avec une granularité fine.

Enfin, Pendulum améliore l'expérience développeur : ses objets sont immuables, ses méthodes chaînables, et sa syntaxe minimaliste rappelle l'élégance des langages déclaratifs. On peut ainsi écrire :

```
pendulum.parse("2025-10-
19T14:30:00Z").in_timezone("Europe/Paris").add(days=5)
```

ce qui produit une opération complète en une seule ligne, lisible et robuste.

En somme, **Pendulum** se distingue par une conception claire et pragmatique : rendre la manipulation du temps expressive, fiable et exempte d'ambiguïtés. Là où `datetime` reste fidèle à la logique interne du langage, Pendulum s'oriente vers l'expérience humaine du temps, conciliant rigueur technique et lisibilité, dans le plus pur esprit "Pythonic".