Les modules

Introduction

Un module, c'est simplement un fichier Python (`.py`) qui contient du code : des fonctions, des variables, des constantes, et parfois des classes. L'idée est de regrouper tout ce qui traite du même sujet dans un seul fichier, pour pouvoir l'utiliser facilement ailleurs dans le programme en faisant un `import`.

Pourquoi est-ce utile ? D'abord, cela rend le code plus clair : au lieu d'avoir tout dans un énorme fichier, on peut organiser le code par thème. Ensuite, c'est réutilisable : si on a besoin de la même fonctionnalité dans un autre projet, on copie simplement le fichier. On peut même le partager avec d'autres développeurs.

Un module crée ce qu'on appelle un "espace de noms". Concrètement, cela signifie que si on a une fonction `calculer()` dans le module `maths.py` et une autre fonction `calculer()` dans `stats.py`, elles ne vont pas entrer en conflit. Quand on fait `import maths`, on sait exactement d'où vient chaque fonction : `maths.calculer()`. Cela évite les bugs et rend le code plus facile à comprendre.

Un point important à comprendre : quand on fait `import mon_module` pour la première fois, Python exécute tout le code du fichier de haut en bas, puis met le résultat en mémoire (on dit qu'il est "mis en cache"). Les fois suivantes, Python réutilise ce qui est déjà en mémoire. C'est rapide et cela garantit que tout le monde utilise la même version. Par contre, attention : tout ce qu'on écrit au niveau principal du module (pas dans une fonction) va s'exécuter à l'import. Il faut donc éviter de mettre de la logique métier à ce niveau, et garder juste des définitions de fonctions et de classes.

Dans un module, on a accès à quelques variables spéciales comme `__name__`, `__doc__` ou `__file__`. Elles donnent des informations sur le module. On peut aussi définir `__all__` : c'est une liste qui dit explicitement ce qui fait partie de l'API publique (ce que les autres peuvent utiliser). Tout ce qui n'est pas dans cette liste est considéré comme "interne", et les autres développeurs ne devraient pas y toucher. C'est important pour travailler en équipe : cela évite qu'une personne utilise une fonction temporaire qui va changer plus tard.

Quand le projet grandit, on va créer des packages : ce sont des dossiers qui contiennent plusieurs modules liés entre eux (avec un fichier `__init__.py` dedans). Cela permet de structurer le code en sous-dossiers logiques. Par exemple : `mon_app/models/`, `mon_app/views/`, etc. Il vaut mieux utiliser des imports absolus (du genre `from mon_app.models import User`) plutôt que relatifs, c'est plus clair et plus robuste. Il faut aussi faire attention aux imports circulaires (quand A importe B et B importe A) : cela cause des bugs difficiles à trouver.

Enfin, il est recommandé d'ajouter toujours un petit commentaire au début du module (une docstring) qui explique à quoi il sert et donne un exemple d'utilisation. C'est très utile pour se relire plus tard, et pour les autres développeurs qui vont lire le code. Un bon module, c'est un module qu'on comprend sans avoir à tout lire.

Exercice 1

On va commencer par la première étape de notre module.

Créez un nouveau fichier nommé `module_csv.py` dans votre espace de travail. Ce fichier représentera votre premier module personnalisé en Python.

À l'intérieur, ajoutez la première fonction du module, celle qui permet de **créer un dossier vide**. Cette fonction doit utiliser le module standard `os`, qui sert à interagir avec le système de fichiers (création, suppression, navigation dans les répertoires, etc.).

Voici la marche à suivre :

1. Importez le module `os`, car c'est lui qui fournit la fonction `mkdir`.
2. Définissez une fonction nommée `create_folder` qui prend deux paramètres : le chemin du dossier parent (`path`) et le nom du dossier à créer (`name`).
3. À l'intérieur, construisez le chemin complet en combinant le dossier parent et le nom du dossier avec `os.path.join()`.
4. Vérifiez si le dossier existe déjà à l'aide de `os.path.exists()`.
5. Si ce n'est pas le cas, créez-le avec `os.mkdir()`.

Une fois la fonction écrite, sauvegardez le fichier. On testera ensuite son fonctionnement dans un second temps pour comprendre comment importer et exécuter une fonction depuis un module.

Correction

```
# module_csv.py
import os

def create_folder(path, name):
    chemin = os.path.join(path, name)
    if not os.path.exists(chemin):
        os.mkdir(chemin)
```

La fonction `os.path.join()` permet de **composer un chemin complet de manière sûre et portable**.

Dans l'exemple :

```
chemin = os.path.join(path, name)
```

on combine deux parties d'un chemin, le dossier parent (`path`) et le nom du nouveau dossier (`name`), en laissant **Python s'occuper automatiquement du séparateur de répertoire** (`/` sur macOS et Linux, `\` sur Windows).

Sans `os.path.join()`, on serait tenté d'écrire manuellement quelque chose comme :

```
chemin = path + '/' + name
chemin = f"{path}/{name}"
```

ou encore :

```
chemin = path + '\\' + name
chemin = f"{path}\{name}"
```

Mais ces écritures sont fragiles, car elles dépendent du système d'exploitation utilisé. Sous Windows, le séparateur est la barre oblique inversée (`\`), alors que sous macOS et Linux, c'est la barre oblique (`/`).

En utilisant `os.path.join()`, Python choisit automatiquement le bon séparateur pour le système sur lequel le code s'exécute. Cela rend votre module **multiplateforme** et évite de nombreux bugs de chemin incorrect.

Ainsi, la variable `chemin` contient ici le **chemin complet du dossier à créer**, qu'on vérifie ensuite avec `os.path.exists()` avant de réellement le créer à l'aide de `os.mkdir()`. Cette combinaison assure à la fois **la portabilité** et **la robustesse** de la fonction.

Exercice 2

On va maintenant enrichir notre module **`module_csv.py`** en y ajoutant une deuxième fonction : **`create_empty_csv_file`**, qui permettra de créer un fichier CSV vide avec des colonnes bien définies.

> **Robert C. Martin**, auteur de *Coder proprement* rappelle que *« Si vous devez commenter et dire ce que fait la fonction, c'est que vous l'avez mal nommée »*. Autrement dit, le nom d'une fonction doit être suffisamment explicite pour que son rôle soit compris sans avoir besoin d'explication supplémentaire. Ici, **`create_empty_csv_file`** exprime clairement son intention : créer un fichier CSV vide.

Étape 1 : Installation de la bibliothèque `pandas`

Cette fonction utilisera la bibliothèque **pandas**, très populaire pour la manipulation de tableaux et de fichiers CSV. Installez-la directement avec la commande suivante :

```
uv add pandas
```

Étape 2 : Recherche personnelle

Avant d'écrire la fonction, prenez le temps de faire une **courte recherche** sur les deux points suivants :

1. Comment créer un objet `DataFrame` vide avec **pandas**.
2. Comment sauvegarder ce `DataFrame` dans un fichier CSV à l'aide de la méthode `.to_csv()`, tout en précisant une liste de colonnes.

Vous pouvez consulter la documentation officielle de pandas ou des ressources pédagogiques comme **W3Schools**, **Real Python** ou **GeeksforGeeks**.

Étape 3 : Implémentation

Une fois ces recherches terminées, ajoutez la fonction dans votre module. Elle devra :

- recevoir en paramètres le **chemin** (`path`) et le **nom du fichier** (`filename`) ;
- accepter une **liste de colonnes** à utiliser comme en-tête du CSV ;
- vérifier si le fichier existe déjà et que c'est bien un fichier CSV ;
- et utiliser `pandas.DataFrame` puis `.to_csv()` pour générer le fichier vide.

Cette fonction constituera la base pour les prochaines manipulations de données dans vos exercices.

Correction

Voici une proposition de correction (plusieurs possibilités sont possibles) :

```
# module_csv.py
import os
import pandas

...

def create_empty_csv_file(path: str, filename: str, /, *, colonnes: list,
separator= ','):
    if not filename.endswith('.csv'):
        liste = filename.split('.') # exemple ['donnees', 'txt']
        filename = liste[0] + '.csv'

    chemin = os.path.join(path, filename)
    if os.path.exists(chemin):
        return

    tableau = pandas.DataFrame(columns= colonnes)
    tableau.to_csv(chemin, sep= separator, index= False)
```

La fonction `create_empty_csv_file` montre plusieurs concepts que nous avons vu dans la première semaine et de nouveaux concepts à comprendre lorsqu'on travaille avec les fichiers CSV en Python.

D'abord, la signature de la fonction :

```
def create_empty_csv_file(path: str, filename: str, /, *, colonnes: list,
separator= ','):
```

Ici, la présence du symbole `/` indique que les paramètres `path` et `filename` doivent être passés **positionnellement**, tandis que `*` impose que `colonnes` et `separator` soient passés **par mot-clé**. Cette écriture renforce la clarté de l'appel de fonction et empêche les confusions d'ordre lors de son utilisation. Par exemple, on devra écrire :

```
create_empty_csv_file("/Users/votre_nom/Documents", "donnees", colonnes=
["nom", "age"])
```

et non en passant tous les arguments de manière positionnelle.

La première partie du corps de la fonction :

```
if not filename.endswith('.csv'):
    liste = filename.split('.')
    filename = liste[0] + '.csv'
```

sert à **garantir que le fichier ait bien l'extension** `.csv`. On découpe le nom du fichier avec `split('.')` pour isoler la partie avant le point (par exemple `donnees` dans `donnees.txt`), puis on reconstruit un nom conforme à l'attendu (`donnees.csv`). Cela évite les erreurs dues à des extensions manquantes ou incorrectes.

Vient ensuite la construction du **chemin complet du fichier** :

```
chemin = os.path.join(path, filename)
```

Pour assembler le dossier et le nom du fichier de façon compatible avec le système d'exploitation. On obtient ainsi un chemin valide quel que soit l'environnement (Windows, macOS ou Linux).

La condition suivante :

```
if os.path.exists(chemin):
    return
```

évite de recréer un fichier qui existe déjà. La fonction s'interrompt silencieusement dans ce cas pour prévenir toute perte de données.

La ligne centrale :

```
tableau = pandas.DataFrame(columns= colonnes)
```

construit un **tableau vide** avec uniquement les noms de colonnes fournis par l'utilisateur. `pandas.DataFrame` est ici utilisé comme structure temporaire pour générer un fichier CSV vide mais correctement formaté.

Enfin :

```
tableau.to_csv(chemin, sep= separator, index= False)
```

écrit ce tableau vide dans le fichier CSV. Le paramètre `sep` permet de choisir le **séparateur** utilisé entre les colonnes (par défaut la virgule, mais on pourrait utiliser `;` dans certains contextes). Le paramètre

`index=False` empêche pandas d'ajouter une colonne d'index automatique.

En somme, cette fonction illustre une conception propre et claire : elle **prépare le nom du fichier**, **vérifie son existence**, **crée un DataFrame vide** et **le sauvegarde au bon endroit**.

Son nom explicite, la typage des paramètres et le découpage logique des étapes traduisent l'esprit prôné par Robert C. Martin : le code se lit comme une phrase et ne nécessite aucun commentaire superflu pour être compris.

Exercice 3

On va maintenant passer à la dernière étape de notre module : écrire la fonction `write_to_csv`.

Cette fois, on n'utilisera pas **pandas**, mais la **librairie native** `csv` de Python. Elle fait partie de la bibliothèque standard et permet de lire et d'écrire dans des fichiers CSV sans dépendance externe.

Avant d'écrire la fonction, prenez le temps de **consulter la documentation officielle** ou un site de référence pour comprendre :

- comment ouvrir un fichier avec la fonction `open()` ;
- comment utiliser la classe `csv.DictWriter` pour écrire des dictionnaires dans un fichier CSV.

Petit rappel : tout comme il faut un **stylo** pour écrire dans un cahier, il faut préparer un **writer** pour écrire dans un fichier CSV. Ce writer saura traduire vos dictionnaires Python en lignes correctement formatées.

Les données à écrire prendront cette forme :

```
donnees = [
    {'id': 1, 'firstname': 'John', 'lastname': 'Doe'},
    {'id': 2, 'firstname': 'Alice', 'lastname': 'Smith'},
    {'id': 3, 'firstname': 'Bob', 'lastname': 'Johnson'}
]
```

À partir de là, à vous de rédiger la fonction `write_to_csv` et d'expérimenter par vous-même : comment construire le chemin, ouvrir le fichier, créer le writer et écrire ces données dans votre CSV ?

Correction

```
def write_to_csv(path: str, filename: str, data: list[dict], colonnes:
list[str], separator= ','):
    chemin = os.path.join(path, filename)
    if not os.path.exists(chemin):
        return

    with open(chemin, 'a') as fichier:
        writer = csv.DictWriter(fichier, colonnes, delimiter= separator,
lineterminator= '\r')
        writer.writerows(data)
```

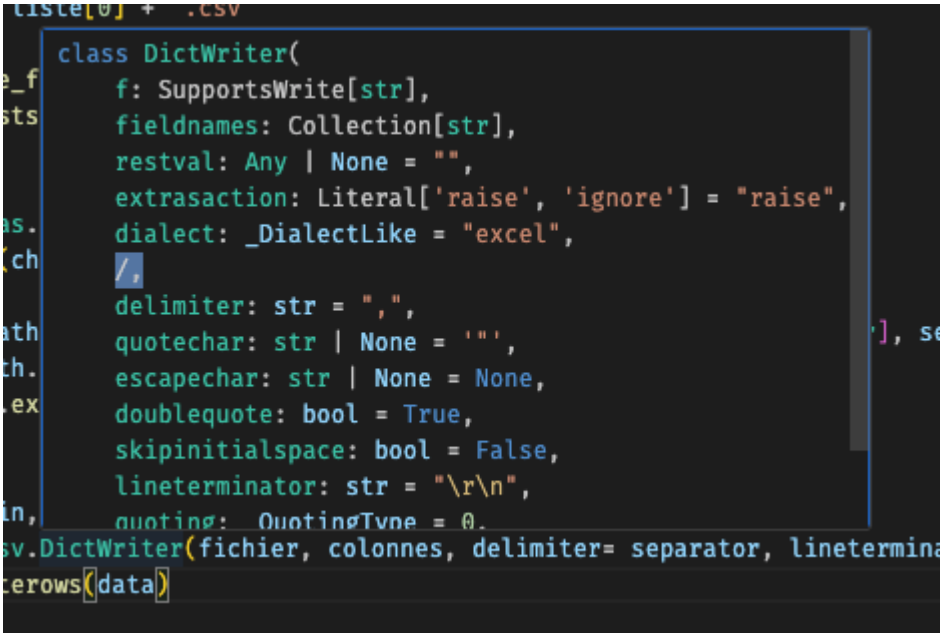
La première partie du code construit le **chemin complet du fichier** grâce à `os.path.join`. On vérifie ensuite que ce fichier existe avec `os.path.exists`, car on ne veut pas écrire dans un fichier inexistant.

La ligne clé se trouve ici :

```
writer = csv.DictWriter(fichier, colonnes, delimiter= separator,
lineterminator= '\r')
```

Cette classe `DictWriter` agit comme un “stylo” spécialisé pour écrire des dictionnaires dans un fichier CSV. Chaque clé du dictionnaire correspond à une colonne du fichier.

Ce qu’il faut remarquer, c’est que `fichier` et `colonnes` sont passés **sans nom de paramètre**, alors que `delimiter` et `lineterminator` sont passés **avec leur nom**. Cela provient directement de la définition de la classe `csv.DictWriter`.



Ici, `f` et `fieldnames` (le fichier et la liste de colonnes) sont des **paramètres positionnels**. Autrement dit, on doit les fournir dans cet ordre, sans préciser leur nom.

Les autres paramètres comme `delimiter`, `lineterminator` ou `quotechar` font partie des **arguments nommés** : ils servent à configurer le comportement du writer (choix du séparateur, fin de ligne, guillemets, etc.).

`writer.writerows(data)` parcourt un itérable (liste, tuple, set...) d’éléments et écrit, pour chacun d’eux, une ligne dans le fichier CSV en appelant implicitement `writer.writerow(...)` à répétition.

Avec un `DictWriter`, on s’attend à ce que chaque élément de `data` soit un dictionnaire dont les clés correspondent aux noms de colonnes déclarés lors de la construction du writer (`fieldnames`). L’ordre des valeurs écrites suit strictement l’ordre de ces `fieldnames`, indépendamment de l’ordre des clés dans chaque dictionnaire, ce qui garantit une structure de fichier stable.

`writer.writerows(data)` ne produit pas d’en-tête; si l’on souhaite écrire la ligne de titres, on utilise explicitement `writer.writeheader()` avant l’appel à `writerows`. L’appel ne renvoie rien de significatif:

il écrit dans le flux associé au fichier. Dans le contexte `with open(...):`, le vidage des buffers et la fermeture du fichier sont garantis à la sortie du bloc, ce qui scelle sur disque toutes les lignes générées par l'itération.

Utilisation du module

À présent que notre module **`module_csv.py`** contient plusieurs fonctions réutilisables, `create_folder`, `create_empty_csv_file`, et `write_to_csv`,, il est temps d'apprendre à **les utiliser dans un autre fichier**.

L'idée est de séparer le **code fonctionnel** (celui qu'on veut réutiliser) du **code d'exécution** (celui qu'on lance). C'est un principe fondamental en programmation : un module contient des outils, tandis qu'un fichier principal, souvent appelé `main.py`, orchestre ces outils pour accomplir une tâche concrète.

Créer le fichier principal

Dans le même dossier que votre module, créez un nouveau fichier nommé **`main.py`**. Ce sera le point d'entrée de votre programme, celui que vous exécuterez pour tester vos fonctions.

Importer le module

Pour utiliser les fonctions écrites dans `module_csv.py`, il faut d'abord **importer le module** dans `main.py`. Si le fichier est dans le même dossier, on l'importe simplement ainsi :

```
import module_csv
```

ou, si l'on veut accéder directement à certaines fonctions :

```
from module_csv import create_folder, create_empty_csv_file, write_to_csv
```

Les deux approches sont valides : la première est plus explicite (`module_csv.create_folder()`), la seconde plus concise (`create_folder()` directement).

Utiliser les fonctions

L'objectif du fichier `main.py` est de tester le bon fonctionnement de votre module. On peut par exemple :

1. Créer un dossier nommé **data** ;
2. Créer un fichier CSV vide à l'intérieur ;
3. Écrire quelques lignes dans ce fichier.

Voici la logique à mettre en place :

```
from module_csv import create_folder, create_empty_csv_file, write_to_csv

path = "."
```

```
folder_name = "data"
filename = "personnes.csv"
colonnes= ['id', 'firstname', 'lastname']

create_folder(path, folder_name)

create_empty_csv_file(
    path,
    filename,
    colonnes= colonnes
)

donnees = [
    {'id': 1, 'firstname': 'John', 'lastname': 'Doe'},
    {'id': 2, 'firstname': 'Alice', 'lastname': 'Smith'},
    {'id': 3, 'firstname': 'Bob', 'lastname': 'Johnson'}
]

write_to_csv(
    path= folder_name,
    filename= filename,
    data= donnees,
    colonnes= colonnes
)
```

L'intérêt de cette séparation

Cette manière de travailler prépare votre code à la **réutilisation**. Votre module peut maintenant être importé dans **n'importe quel projet**, sans copier-coller les fonctions. Et `main.py` devient votre **script de test**, où l'on orchestre les appels pour vérifier que tout fonctionne comme prévu.

C'est exactement ainsi qu'on structure des projets professionnels : les modules définissent des fonctionnalités réutilisables, et le fichier principal sert de point d'entrée pour exécuter la logique métier du programme.

Revenons sur l'écriture suivante :

```
from module_csv import *
```

on demande à Python d'importer **toutes les fonctions, classes et variables publiques** définies dans le module `module_csv`. Mais attention : le mot-clé `*` ne signifie pas "tout sans distinction". On peut **filtrer** par une variable spéciale nommée `__all__`, si elle est présente dans le module.

Rôle de `__all__`

Dans notre fichier `module_csv.py`, on va définir tout en haut du fichier :

```

```

```
__all__ = [
    'create_folder',
    'create_empty_csv_file',
    'write_to_csv'
]
```

Cette liste indique explicitement à Python quelles fonctions seront **visibles** lors d'un import global (`from module_csv import *`). Ainsi, seules ces trois fonctions seront importées automatiquement. Toutes les autres fonctions, même si elles existent dans le module, resteront **internes** et ne seront pas accessibles depuis l'extérieur.

Cette technique permet de **contrôler l'interface publique** d'un module. Autrement dit, on décide quelles fonctions font partie de son API officielle et lesquelles doivent rester cachées, car elles servent uniquement à l'interne. C'est une pratique de conception propre et professionnelle : elle empêche l'utilisateur du module d'utiliser accidentellement une fonction auxiliaire ou expérimentale.

Exemple

Supposons qu'on ait ajouté cette fonction dans le module :

```
def _sanitize_filename(filename: str) -> str:
    return filename.strip().replace(' ', '_')
```

> Le terme "**sanitize**" est souvent utilisé pour décrire une fonction qui supprime des caractères spéciaux ou qui normalise un nom de fichier. Une fonction qui nettoie.

Cette fonction est utile **uniquement à l'intérieur du module** pour s'assurer qu'un nom de fichier est propre avant sa création. Elle n'a pas vocation à être utilisée directement par un autre fichier comme `main.py`.

> Le préfixe `_` signale aux autres développeurs qu'il s'agit d'une **fonction privée**, réservée à un usage interne. C'est une convention de programmation classique, mais elle n'est pas obligatoire.

Son utilisation dans le code:

```
def create_empty_csv_file(path: str, filename: str, /, *, colonnes: list,
separator= ','):
    filename = _sanitize_filename(filename)
    ...
```

Si on écrit :

```
from module_csv import *
```

alors `_sanitize_filename` **ne sera pas importée**, car elle n'est pas listée dans `__all__`. Et même sans cette liste, le préfixe `_` signale déjà à Python (et aux autres développeurs) qu'il s'agit d'une **fonction privée**, réservée à un usage interne.

En résumé

- `from module_csv import *` importe uniquement ce que le module décide de rendre public via `__all__`.
- `__all__` définit la frontière entre l'interface publique (ce que l'utilisateur du module peut utiliser) et les détails internes (ce que seul le module doit manipuler).
- Les fonctions internes comme `_sanitize_filename` permettent d'alléger le code principal du module sans encombrer son interface d'utilisation.