Utiliser un module

Dernière mise à jour le : 06/12/2023

Qu'est-ce qu'un module ?

Un **module** Python est un fichier d'extension `.py`, comme n'importe quel script Python, dans lequel on écrit un ensemble de fonctions ou de classes (les classes sont au programme de Terminale), on définit des constantes, etc.

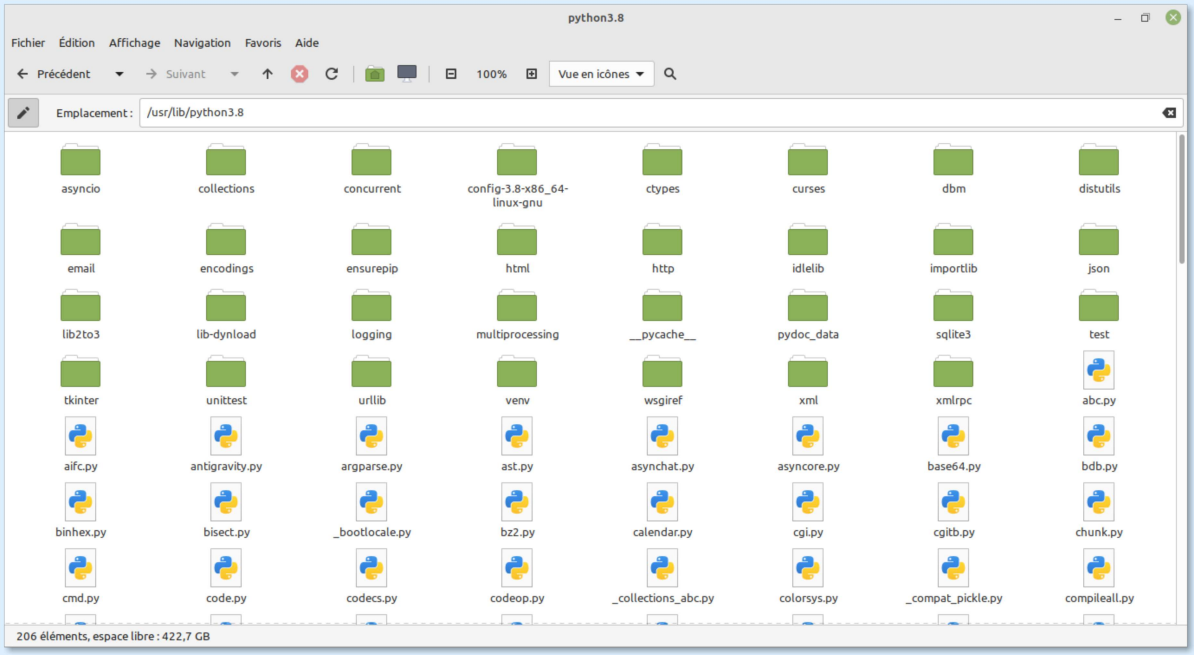
Une **bilbiothèque** est un ensemble de plusieurs modules qui portent généralement sur un même thème précis.

L'intérêt principal des modules (et bibliothèques) est leur réutilisation par d'autres programmes et d'autres programmeurs ou programmeuses. En effet, il est très simple d'*importer* un module (écrit par nous-même ou par quelqu'un d'autre) pour utiliser les fonctions (et classes, et constantes, etc.) définies dans le module. Dans ce cas, on peut voir un module comme une "brique" qui contient des fonctionnalités que l'on peut utiliser sans avoir à les coder soi-même, pour construire son propre programme (ou son propre module).

Les développeurs Python ont mis au point de très nombreux modules pour effectuer une quantité impressionnante de tâches. Beaucoup de ces modules sont déjà installés dans les versions standards de Python. On peut trouver la liste de ces modules directement dans la documentation de Python. On peut citer par exemple :

- le module `turtle` qui permet de réaliser des dessins géométriques,
- le module `random` qui permet de générer des variables aléatoires,
- le module `math` qui permet d'utiliser des fonctions et constantes mathématiques de base,
- le module `csv` qui permet de manipuler plus simplement des fichiers CSV (voir le Thème 4 sur le traitement de données en tables),
- le module `time` qui permet l'accès à l'heure de l'ordinateur et aux fonctions gérant le temps.

Sur vos machines au lycée, on trouve tous ces modules dans le répertoire `/usr/lib/python3.8` (pour la version 3.8 de Python) :



N'hésitez pas regarder le contenu de ces dossiers, vous y trouverez des dossiers ainsi que des fichiers Python au format `.py` correspondant respectivement aux bibliothèques et aux modules (simples).

Mais il y en a en réalité beaucoup (beaucoup !) d'autres :

- le dépôt tiers officiel du langage Python, appelé **PyPI** (Python Package Index), en contient des centaines de milliers : https://pypi.org/

Logo de PyPI
Crédits : PPA / PSF, GPL, via Wikimedia Commons

- il y en a aussi énormément sur des plateformes de dépôts, comme **GitHub** : https://github.com/search?q=python+module.

Logo de la plateforme GitHub
Crédits : GitHub Inc., MIT, via Wikimedia Commons

Puisqu'il est généralement destiné à d'autres programmeuses et programmeurs, un module s'accompagne d'une **documentation** qui détaille la façon de l'utiliser.

Importer et utiliser un module

Dans la très grande majorité des cas, les modules sont importés en tout début de programme. Il existe plusieurs manières de procéder. La façon d'importer un module aura une influence sur la façon dont on doit utiliser les fonctionnalités du module, autrement dit sur la façon d'écrire notre code Python par la suite.

> On prendra l'exemple du module `math` mais tout ce qui sera dit est bien entendu vrai quel que soit le module.

Méthode 1 : Utiliser un simple `import`

Pour importer un module, on peut simplement écrire

```
>>> import math
```

> Cette ligne n'est à écrire qu'une seule fois, généralement tout au début du programme. Vous constaterez que l'on n'écrit pas l'extension `.py` lors de l'import.
>
> Si le module que vous souhaitez importer n'est pas trouvé, un message vous l'indiquera, comme ci-dessous :
>
> ```
> >>> import maths  # avec un s en trop
> Traceback (most recent call last):
>   File "<pyshell>", line 1, in <module>
> ModuleNotFoundError: No module named 'maths'
> ```

On peut dès lors utiliser les fonctionnalités de ce module (ses fonctions, ses constantes, etc.) :

```
>>> import math
>>> math.sqrt(36)  # sqrt est la fonction racine carré
6.0
>>> math.pi  # le nombre pi
3.141592653589793
>>> math.cos(math.pi)  # cos est la fonction cosinus
-1.0
```

Vous noterez qu'avec cette première méthode, il est **nécessaire de préfixer les fonctions ou les constantes du module par le nom du module**.

Pour connaître toutes les fonctionnalités d'un module, on peut :

- soit utiliser la fonction `help` (après avoir importé le module):

```
>>> help(math)
```

- soit consulter la documentation en ligne :
  - en français : https://docs.python.org/fr/3/library/math.html
  - ou en anglais : https://docs.python.org/3/library/math.html

Pour obtenir accéder à la chaîne de documentation d'une fonction d'un module importé, on peut exécuter

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```

ou

```
>>> math.sqrt.__doc__
'Return the square root of x.'
```

Ai-je bien compris ?

Utilisez une console Python pour faire les questions suivantes.

1. Importez le module `random` (disponible dans la distribution standard de Python) avec `import`.
2. Quelle instruction permet d'accéder à la documentation de la fonction `randint` de ce module ?
3. Écrivez l'instruction qui permet d'utiliser cette fonction pour générer un nombre entier aléatoire compris entre 0 et 10 inclus.

Méthode 2 : Tout importer avec `from ... import *`

On peut également importer un module avec l'instruction :

```
>>> from math import *
```

Le caractère `*` indique que l'on importe toutes les fonctions (et constantes, etc.) du module (ici `math`). Cette instruction se traduit littéralement par *importe tout ce qu'il y a dans le module* (ici `math`).

Dans ce cas, différence fondamentale avec la première méthode, **on n'a plus besoin de préfixer le nom des fonctionnalités par le nom du module** pour les utiliser :

```
>>> from math import *
>>> sqrt(36)
6.0
>>> pi
3.141592653589793
```

D'ailleurs, si on le fait, il y a un problème :

```
>>> math.sqrt(36)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'math' is not defined
```

> C'est super ça ! Mais pourquoi diable utiliser la première méthode quand on sait que la seconde permet d'écrire des instructions plus courtes ? 🤔

Inconvénient

En réalité, cette méthode est souvent à éviter car elle présente un inconvénient majeur : elle peut facilement mener à un conflit si plusieurs fonctions portent le même nom (et on ne s'en rendra pas forcément compte !). Par exemple, imaginons la situation où on importe toutes les fonctionnalités de deux modules avec un `import *` :

```
from module1 import *
from module2 import *
```

Si chacun des deux modules contient une fonction portant un nom *identique*, par exemple `bidule`, alors une seule des deux fonctions ne peut être utilisée. C'est en réalité celle correspondant au *dernier* module importé qui sera disponible (et pas l'autre) : dans notre cas, ce serait la fonction `bidule` de `module2` (car son importation viendrait écraser la fonction `bidule` de `module1`).

De la même manière, si on définit nous-même une fonction avec un nom correspondant à celui d'une fonction importée d'un module, on se retrouve avec un conflit :

```
>>> from math import *  # la fonction sqrt est donc importée
>>> def sqrt(x):  # on définit notre propre fonction sqrt (complètement idiote)
        print('Salut !')
>>> sqrt(36)
Salut !
```

> **Moralité** : Cette méthode (`from ... import *`) est déconseillée car elle rend plus difficile de déterminer l'origine d'une fonction, ce qui rend l'accès aux documentations et le débogage plus difficiles, et provoque potentiellement des conflits.

On peut néanmoins utiliser `from` pour importer plus "proprement" des fonctionnalités d'un module, c'est ce qui est expliqué dans la troisième méthode.

Méthode 3 : Utiliser `from ... import ...`

On peut décider de n'importer que certaines fonctionnalités d'un module. Cela peut se faire ainsi :

```
>>> from math import sqrt, pi  # on importe uniquement sqrt et pi
>>> sqrt(36)
6.0
>>> pi
3.141592653589793
```

Remarquez qu'avec cette façon de faire, il n'est **pas nécessaire de préfixer les fonctions (ou constantes) par le nom du module**.

En revanche, si on essaie d'utiliser une fonction non importée, on obtient une erreur :

```
>>> cos(pi)  # la fonction cos n'a pas été importée donc est indisponible
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'cos' is not defined
```

Avec cette méthode, il est nécessaire de nommer précisément les fonctions et constantes importées, ce qui présente un avantage certain par rapport à un `from ... import *`. En effet, cela permet à la fois au programmeur et à l'utilisateur de voir directement quelles sont les fonctionnalités du module qui seront utiles (ayez en tête que certains modules contiennent des dizaines voire des centaines de fonctions). Ainsi, il sera plus facile d'identifier les noms des fonctions importées et faire en sorte d'éviter les conflits évoqués dans le paragraphe précédent.

Néanmoins, on ne pourra toujours pas importer de cette manière deux fonctions portant le même nom dans des modules distincts :

```
from module1 import bidule
from module2 import bidule  # va écraser la fonction bidule de module1
```

Pour contourner ce problème, il n'y a pas d'autre choix que d'importer le module comme dans la première méthode :

```
import module1
import module2
```

On peut alors utiliser les instructions `module1.bidule()` et `module2.bidule()` pour accéder respectivement aux fonctions `bidule` des modules `module1` et `module2`.

Ai-je bien compris ?

Utilisez une console Python pour faire les questions suivantes.

1. Importez *uniquement* les fonctions `random` et `randint` du module `random` (disponible dans la distribution standard de Python).
2. Quelle instruction permet alors d'utiliser la fonction `randint` pour générer un nombre entier aléatoire compris entre 0 et 10 inclus ?

Méthode 4 : importer un module en le renommant avec `as`

Lorsque l'on trouve que le fait de devoir préfixer chaque fonctionnalité par le nom de son module est un peu long à écrire, il est possible de renommer le module au moment de son import. On utilise pour cela le mot clé `as`, qui permet de définir un *alias* pour le module (on utilise alors un nom plus court, sinon cela n'a pas vraiment d'intérêt) :

```
import module1 as m1  # m1 est l'alias de module1
import module2 as m2  # m2 est l'alias de module2
```

Dans ce cas, on écrira `m1.bidule()` ou `m2.bidule()` pour accéder aux fonctions `bidule` de chaque module.

Ai-je bien compris ?

Utilisez une console Python pour faire les questions suivantes.

1. Importez le module `math` en le renommant `m` avec le mot clé `as`.
2. Quelle instruction permet alors d'utiliser la fonction `sqrt` pour calculer la racine carré d'un nombre ?

Bilan

- Un **module** Python est un simple fichier d'extension `.py`. On peut regrouper plusieurs modules (qui portent généralement sur une même thématique) dans une **bibliothèque**.
- N'importe qui peut écrire ses propres modules, basés ou non sur d'autres, et les rendre disponibles à l'ensemble de la communauté des développeurs et développeuses.
- L'importation des modules se fait la plupart du temps tout en haut d'un script.
- Il existe différentes méthodes pour importer un module, chacune ayant des avantages et des inconvénients :
  - `import module` est la manière standard de le faire, mais nécessite de préfixer toutes les fonctions du module par le nom du module, ce qui permet d'indiquer sans ambiguité de quel module provient la fonction ;
  - `import module as m` permet de renommer `module` par l'alias `m`, afin de raccourcir les écritures préfixés de la première méthode
  - `from module import *` permet d'importer toutes les fonctionnalités d'un module et on peut les utiliser sans les préfixer par le nom du module ;
  - `from module import fonction_1, fonction_2` permet de n'importer que certaines fonctions du module, sans avoir besoin de les préfixer par le nom du module pour les utiliser
- L'utilisation de `from module import *` est souvent déconseillée pour éviter les conflits et faciliter le bébogage, principalement lorsque l'on travaille sur des projets conséquents.
- L'utilisation de `from module import fonction_1` a l'avantage de nommer les fonctions importées, pour repérer plus facilement d'éventuels conflits, mais ne règle pas le problème de l'importation de fonctions portant des noms identiques et provenant de

modules différents. Il faudrait dans ce dernier cas, utiliser les méthodes `import module` ou `import module as m`.

Références :

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe DECLERCQ & Christophe JERMANN.
- Cours de Python : *Introduction à la programmation Python pour la biologie* de Patrick Fuchs et Pierre Poulain : chapitres 8 - Modules, 14 - Création de modules et 15 - Bonnes pratiques en programmation Python
- Page Web : http://python.lycee.free.fr/modules_utiles.html

Sébastien POINT & Germain BECKER, Lycée Mounier, ANGERS