Chapitre n° 3 : modularité

1. Importations dans Python

1.1. Présentation et définitions

En général, dans l'écriture d'un programme, il n'est pas question de réécrire ce qui existe déjà : ce serait beaucoup trop long et improductif. C'est pour cela que de nombreux programmes commencent par importer des fonctionnalités supplémentaires. Un des intérêts du langage Python est justement la richesse de ses nombreuses bibliothèques et de ses modules dans des domaines très variés.

Définition : *un module est un fichier contenant des variables et des fonctions.*

Définition : *un package est une collection de modules sous un espace de nom commun. Il contient un fichier* __init__.py *pour indiquer à Python qu'il ne s'agit pas d'un simple répertoire, mais d'un package.*

Définition : *une bibliothèque est constituée d'un ou plusieurs* packages *(ensemble de dossiers et de sous-dossiers) contenant plusieurs modules.*

Remarque : bibliothèque se traduit en anglais par *library*, à ne pas confondre avec une librairie qui se traduit en anglais par *bookshop* ou *bookstore*.

Une bibliothèque standard est distribuée avec Python. Il est possible d'importer des bibliothèques supplémentaires selon les besoins.

Définition : *une API (*Application Programming Interface*) est une interface permettant à des programmes d'interagir avec une bibliothèque sans en avoir un accès direct.*

C'est la spécification d'un protocole, en quelque sorte, qui permet de communiquer avec la bibliothèque.

1.2. Syntaxe des importations

La syntaxe minimale pour importer un module utilise la fonction import.

Exemple : import de toutes les fonctions de la bibliothèque *numpy*.

```
1 import numpy
```

Remarque : avec cette syntaxe, pour utiliser une des fonctions du module, il faut faire précéder le nom de la fonction par le nom du module.

Exemple : erreur car la fonction *cos* n'est pas définie par défaut dans Python.

```
1 import numpy
2 print(cos(0))
```

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 import numpy
2 print(numpy.cos(0))
```

Pour éviter de devoir saisir le nom complet du module, il est possible d'utiliser un alias avec la synthaxe import ... as ....

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 import numpy as np
2 print(np.cos(0))
```

La synthaxe from ... import ... *permet de n'importer qu'une ou plusieurs fonctions et évite de devoir saisir le nom du module.*

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 from numpy import cos
2 print(cos(0))
```

La syntaxe précédente peut permettre d'importer toutes les fonctions tout en évitant de devoir saisir le nom du module : from ... import $*$. *Cette possibilité est cependant vivement déconseillée.*

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 from numpy import *
2 print(cos(0))
```

1.3. Bibliothèques classiques

Pour un usage pédagogique, nous avons utilisé la bibliothèque *turtle*. Il permet de découvrir de façon ludique la programmation, dont la programmation orientée objet et la programmation événementielle.

Les bibliothèques *numpy* et *scipy* sont utilisées pour le calcul scientifique, *matplotlib* pour tout ce qui concerne les graphiques, *pil* pour la manipulation et le traitement d'images, *tkinter* pour obtenir des outils pour la création d'interfaces graphiques, *pygame* pour la création de jeux en 2D, etc.

Remarque : dans le cas où l'utilisateur créer un module qu'il souhaite ensuite importer, il faut préciser à l'interpréteur Python le chemin d'accès au fichier. La syntaxe sera du type :

```
1 from os import chdir
2 chdir(''C:/users/Toto/docs/programmes'')
```

Une alternative, plus simple, est de placer dans le même dossier le fichier du module et le fichier du programme réalisant l'importation.

2. Modules, interfaces et encapsulation

2.1. Découpage et réutilisation de code

Une des clefs du développement à grande échelle consiste à circonscrire et séparer proprement les différentes parties du programme. Typiquement, l'interface graphique est séparée du cœur de l'application. *Les grands programmes doivent être fractionnés en plusieurs parties (briques), c'est le principe de la modularité.*

Chaque partie (brique) est un module. Les fonctionnalités définies dans un des modules peuvent être utilisées dans un autre module. On a donc tout intérêt à ce que chaque module soit dédié à la résolution d'une tâche, ou d'un ensemble de tâches apparentées, clairement identifiable.

Les avantages sont que le programme est :

- lisible ;
- portable ;
- réutilisable (une brique peut servir à plusieurs programmes) ;
- facile à maintenir et à modifier.

2.2. Interfaces

Un module possède une partie publique appelée interface et une partie privée appelée implémentation (ou réalisation).

L'interface énumère les fonctions définies dans le module et qui sont destinées à être utilisées dans la réalisation d'autres modules, appelés clients. *L'interface est donc liée à sa documentation : pour chaque fonction il faut connaître son rôle, son nom, la liste de ses paramètres et sa spécification*, c'est à dire les conditions auxquelles la fonction peut être appliquée et les résultats à attendre. Éventuellement, des informations concernant le temps d'exécution ou l'espace mémoire peuvent être utiles.

L'objectif de l'interface est de permettre à un utilisateur de faire appel aux fonctionnalités du module sans avoir besoin de consulter son code.

L'implémentation peut être plus ou moins complexe : l'auteur du module est libre de coder à sa manière. Il doit juste respecter le cahier des charges des fonctions décrites dans l'interface.

Remarque : en Python, une variable *__name__* est crée automatiquement à l'exécution d'un programme. Elle contient le nom du programme courant quand on importe un programme, mais le nom du programme principal est toujours *__main__*. Par conséquent, les instructions placées après la condition suivante ne sont exécutées que si le module est le programme principal :

```
if __name__ = "__main__":
    bloc d'instructions
```

C'est utile, par exemple, pour placer des tests importants pour l'implémentation du module, mais qui ne font pas partie de son interface.

2.3. Encapsulation

L'auteur d'un module, pour créer son implémentation, est libre d'utiliser les structures de données qu'il souhaite, de définir des fonctions, des objets, etc., qui ne sont pas inclus dans l'interface. Ces éléments internes, souvent qualifiés de « détails d'implémentations », peuvent constituer une large part du code du module.

Tous ces éléments, hors de l'interface, sont qualifiés de privés et ne doivent pas être utilisés par les modules clients. On parle à leur propos d'encapsulation, pour signifier qu'ils sont comme enfermés dans une boîte hermétique que l'utilisateur ne doit pas ouvrir.

L'intérêt de l'encapsulation est de diminuer le couplage entre les modules. Ainsi, l'auteur d'un module peut faire évoluer son code, sans que les codes des modules clients n'aient besoin d'être adaptés.

Remarque : de nombreux langages de programmation, adaptés aux projets à grande échelle, permettent un contrôle strict de l'encapsulation, rendant l'accès aux éléments privés très compliqué voire impossible.

En Python, une telle fonctionnalité n'existe pas. En revanche, l'auteur d'un module peut indiquer que certains éléments (variables globales et fonctions) sont privés en faisant commencer le nom par « _ ». Par convention, tous les autres éléments sont publics et doivent être compris comme appartenant à l'interface.

Capacités exigibles

- Utiliser des API (*Application Programming Interface*) ou des bibliothèques.
- Exploiter leur documentation.
- Créer des modules simples et les documenter.