CHAPITRE 1

Manipuler les modules

1. **Création des modules**
2. Importation des modules

Création de modules

- Un module est **un fichier « .py »** contenant un ensemble de **variables, fonctions et classes** que l'on peut importer et utiliser dans le programme principal (ou dans d'autres modules).
- Pour créer un module, il suffit de programmer les variables/fonctions et classes qui le constituent dans un fichier portant le nom du module, suivi du suffixe « .py ». Depuis un (autre) programme en Python, il suffit alors d'utiliser la primitive import pour pouvoir utiliser ces variables/fonctions/classes.
- **Les modules :**
  - Permettent la **séparation** du code et donc une **meilleure organisation du code**
  - Maximisent la **réutilisation**
  - **Facilitent le partage du code**

Exemple :

```
"""
exemple de module, aide associée
"""

exemple_variable = 3

def exemple_fonction():
    """exemple de fonction"""
    return 0
class exemple_classe:
    """exemple de classe"""

    def __str__(self):
        return "exemple_classe"
```

- L'exemple ci-dessus montre la définition d'un module contenant une variable, une fonction et une classe. Ces trois derniers peut être exploités par n'importe quel fichier important ce module.
- Le nom du module représente le nom du fichier sans son extension.

CHAPITRE 1

Manipuler les modules

1. Création des modules
2. **Importation des modules**

- l'instruction **import nom_module** permet l'importation d'un module. L'importation doit se faire avant l'exploitation des composantes du module.
- En règle générale, toutes les importations sont faites au début du programme. Cependant, elles peuvent être faite dans n'importe quelle partie du programme.

```
import module_exemple  #importation du module module_exemple


c = module_exemple.exemple_classe ()  #appel de la classe exemple_classe
print(c)
print(module_exemple.exemple_fonction()) #appel de la fonction  exemple_fonction
```

Exemple :

- L'instruction **as** suivi de **alias** permet l'assignation d'un identificateur à un module. Cet identificateur peut être différent du nom du fichier dans lequel est définit le module.
- Cette méthode représente une autre manière de faire afin d'importer des modules.

```
import module_exemple as alias
c = alias.exemple_classe()
print(c)
print(alias.exemple_fonction())
```

- La syntaxe suivante n'est pas recommandée car elle masque le module d'où provient une fonction en plus de tout importer.

- **import *** vous offre la possibilité d'importer toutes les composantes d'un module (classes, attributs et fonctions). Cependant, il est possible d'importer qu'une classe de ce module en écrivant **from module_exemple import exemple_class**

- Lorsqu'on importe un module, l'interpréteur Python le recherche dans différents répertoires selon l'ordre suivant :
  1. Le répertoire courant ;
  2. Si le module est introuvable, Python recherche ensuite chaque répertoire listé dans la variable shell PYTHONPATH ;
  3. Si tout échoue, Python vérifie le chemin par défaut (exemple pour windows \Python\Python39\Lib)

```
from module_exemple import *  # importer toutes les classes, attributs, fonctions..
from module_exemple import exemple_classe, exemple_fonction

c = exemple_classe()
print(c)
print(exemple_fonction())
```

Arborescence de modules

- Si vous avez plusieurs modules, il est préférable de répartir leurs fichiers dans des répertoires. Ces derniers doivent apparaitre dans la liste **sys.path .**
- Python permet la définition de paquetage. Un répertoire est considéré comme étant un package qui contient tous les fichiers Python.
- Avant Python 3.3, un package contenant des modules Python doit contenir un fichier **__init__.py**

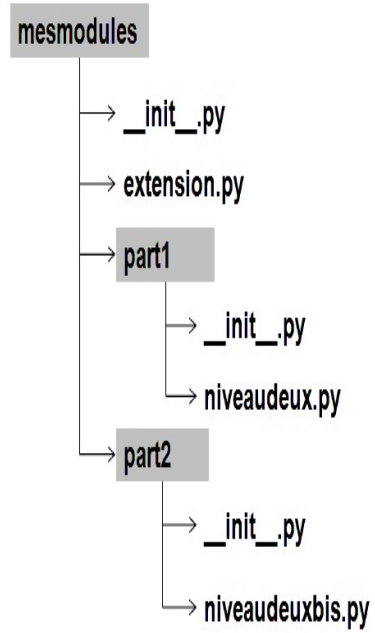
Importation absolue

- Une importation absolue spécifie la ressource à importer à l'aide de son chemin d'accès complet à partir du répertoire racine.

Exemple :

```
import mesmodules.extension
import mesmodules.part1.niveaudeux
import mesmodules.part2.niveaudeuxbis
```

- En exécutant l'instruction **import mesmodules.extension, Python** exécute le fichier extension.py mais également tout le contenu du fichier **__init__.py**



PARTIE 4

Importation relative

- Une importation relative spécifie la ressource à importer par rapport à l'emplacement actuel, c'est-à-dire l'emplacement où se trouve l'instruction import.
  - **Le symbole .** permet d'importer un module dans le même répertoire.
  - **Le symbole ..** permet d'importer un module dans le répertoire parent.

Exemple :

- **La fonction A** peut utiliser **la fonction B ou C** en les important de la façon suivante :

```
from .subpackage1 import B

from .subpackage1.moduleX import C
```

- **La fonction E** peut utiliser **la fonction F ou A ou C** en les important de la façon suivante :

```
from ..moduleA import F
from .. import A
from ..subpackage1.moduleX import
```

```
package/
    __init__.py         # fonction A
    subpackage1/
        __init__.py     # fonction B
        moduleX.py      # fonction C
    subpackage2/
        __init__.py     # fonction D
        moduleY.py      # fonction E
    moduleA.py          # fonction F
```

PARTIE 4

PYTHONPATH

- Pour ajouter un dossier au PYTHONPATH en Python, il faut indiquer directement dans le code les lignes suivantes :

```
import sys
sys.path.insert(0 ,"E:/exempleImport")
```

Chemin du module à emporter

- La fonction path du module sys permet de vérifier la variable PYTHONPATH

```
import sys
print(sys.path)
```

- **Remarque :** insert ne permet pas d'ajouter le dossier en question dans PYTHONPATH de façon permanente

```
['', 'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python39\\Lib\\idlelib', 'E:\\exempleImport', 'C:\\Users\\DELL\\AppData\\Local\\Pr
ograms\\Python\\Python39\\python39.zip', 'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python39\\DLLs', 'C:\\Users\\DELL\\AppData\\Lo
cal\\Programs\\Python\\Python39\\lib', 'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python39', 'C:\\Users\\DELL\\AppData\\Local\\Pro
grams\\Python\\Python39\\lib\\site-packages']
```