# Chapitre 9 - Modularité

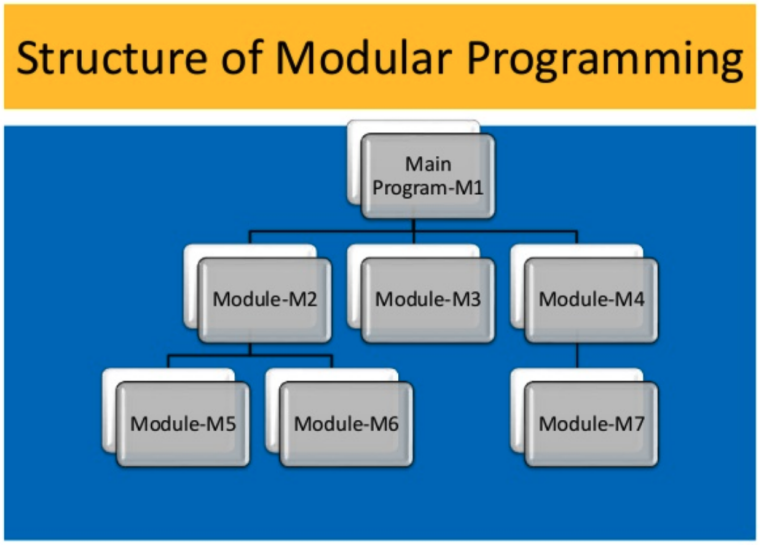
**Objectifs :**

- ▷ Savoir utiliser des bibliothèques
- ▷ Exploiter leur documentation
- ▷ Créer des modules simples et les documenter.

## 1 Introduction

La **programmation modulaire** consiste à décomposer une grosse application en modules que l'on peut ensuite améliorer et utiliser indépendamment dans d'autres applications.

Cette méthode réalise une **encapsulation** comparable à celle de la programmation orientée objet.




Un module doit être conçu pour pouvoir être utilisé par d'autres utilisateurs comme une interface qui fournit des données (`math.pi`, `math.e`...), des objets (structures de données comme les piles, les arbres, les graphes, ..., *etc.*) et des traitements (fonctions, méthodes), sans rentrer dans le code du module.

Il faut donc prévoir une **documentation claire et complète**.

## 2 Exemples

Nous avons déjà créé des modules en implémentant des listes, des piles, des files, des arbres et des graphes.

Python dispose d'une vaste bibliothèque de modules, certains sont fréquemennt utilisés comme `math`, `random`, `numpy`, `matplotlib`, `tkinter`,..., *etc.*.

### 2.1 Module simple

Un module simple est juste un fichier écrit en Python.

**Exemple 1 :** *Turtle*

- ▷ Importer le module `turtle` avec `import turtle`
- ▷ Repérer son emplacement sur le disque dur avec `turtle.__file__`
- ▷ Ouvrir le fichier `turtle.py` dans un éditeur et observer le texte en début de fichier
- ▷ Afficher la doc avec `print(turtle.__doc__)`

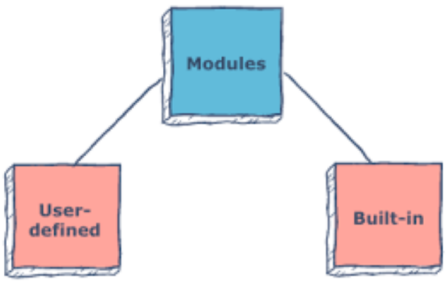
Remarquez que `turtle` fait appel à d'autres modules. D'après vous, quel module gère l'interface graphique de `turtle` ?

### 2.2 Module complexe

Les modules plus complexes sont des répertoires contenant de nombreux fichiers, parfois codés en C.

**Exemple 2 :** *Numpy*

- ▷ Importer le module `numpy` avec `import numpy`
- ▷ Repérer son emplacement sur le disque dur avec `numpy.__file__`
- ▷ Ouvrir le fichier `numpy/__init__.py`
- ▷ Regarder la documentation et l'afficher dans votre interpréteur Python
- ▷ Remarquez que le fichier `__init__.py`, qui est appelé lorsqu'on importe le module `numpy` n'est qu'un des nombreux fichiers de ce module.




Certains modules sont incorporés (`builtins`) dans l'interpréteur Python.

Une fonction extraite d'un module connu sera en principe plus performante qu'une fonction programmée soi-même. Elle a été optimisée et testée, et sera beaucoup plus rapide si elle est écrite en C (ce qui est le cas des modules `math` et `numpy` par exemple).

## 3 Accès à la documentation d'un module

Tester toutes les commandes suivantes avec des modules et fonctions que vous utilisez régulièrement. Par exemple avec la fonction `randint` du module `random`.

- ▷ `dir(nom_du_module)` renvoie tous les attributs du module.
- ▷ `nom_du_module.__doc__` renvoie la description du module (chaîne de caractères entrée en premières lignes de code).
- ▷ `nom_du_module.nom_de_la_fonction.__doc__` renvoie la description (ou la documentation ) d'une fonction (chaîne de caractères entrée en premières lignes).
- ▷ `help(nom_du_module)` génère et affiche une documentation constituée des descriptions du module et de ses différentes classes et fonctions.
- ▷ Fonctionnement similaire avec les classes : la méthode `__doc__` appliquée à une classe d'un module renvoie la description de la classe et la fonction `help()` affiche la description de la classe et de toutes ses méthodes.
- ▷ Après avoir fait `import sys`, tester `sys.builtin_module_names` qui permet d'obtenir la liste des modules incorporés.
- ▷ Tester avec quelques modules que vous avez importés (`random`, `turtle`, `tkinter`,`math` ,. . ., *etc.*) : `random.__doc__`, `help(tkinter)`, `turtle.goto.__doc__`, `help(math.tan)`, . . .

# 4 Documenter un module

La documentation d'un module est une chaîne de caractères située au début du module.

Toutes les fonctions du module sont également documentées par une chaîne de caractères située au début de la fonction. Il n'y a pas de règle universelle pour écrire la documentation d'un code, mais on précise généralement les préconditions et postconditions des fonctions.

```
NAME
    sklearn

DESCRIPTION
    Machine learning module for Python
    ==================================

    sklearn is a Python module integrating classical machine
    learning algorithms in the tightly-knit world of scientific Python
    packages (numpy, scipy, matplotlib).

    It aims to provide simple and efficient solutions to learning problems
    that are accessible to everybody and reusable in various contexts:
    machine-learning as a versatile tool for science and engineering.

    See http://scikit-learn.org for complete documentation.
```

Voici un exemple de fonction bien documentée :

```python
def pgcd(a,b) :
    """ Calcule le pgcd de a et b
        Parametres
        ------------
            a : int (entier quelconque)
            b : int (entier quelconque)
        Renvoie
        ---------
            int : pgcd de a et de b """

    while b != 0 :
        a, b = b, a%b

    return a
```

# 5 A propos de l'importation

En Python, `import module` va charger et exécuter le fichier désigné, en général `module.py` ou dans le cas d'un module complexe, un fichier `__init__.py`.

## 5.1 Remarques

- `module.__file__` renvoie l'emplacement de ce fichier
- import module définit alors un espace de noms, comme en POO
- Les données, classes, fonctions du fichier importé sont accessibles avec `module.donnees` ou `module.fonction`,...
- `import module as md` va importer le module en le renommant `md`, afin de raccourcir les noms

## 5.2 Exemples

Quelques exemples de renommages bien connus :

- `import numpy as np`
- `import matplotlib.pyplot as plt`
- `import tkinter as tk`

## 5.3 `import *`

Attention, la ligne de code ci-dessous :

```
from module import *
```

va importer toutes les fonctions du module sans créer d'espace de noms.

Ce n'est généralement pas recommandé car si deux entités (classes, fonctions...) portent le même nom dans le module et dans notre code, l'une des deux va être écrasée.

## 5.4 `main`

On peut aussi utiliser la ligne de code ci-dessous :

```
if __name\_\_ == '__main__' :
```

Lors d'un import, le fichier importé est exécuté, ce qui permet de charger ses variables, classes et fonctions en mémoire. Dans le cas d'un module importé, la variable `__name__` contient le nom du module.

Ce nom est égal à `__main__` lorsque le module est le fichier principal, et pas lorsqu'il est importé.

Ainsi l'instruction conditionnelle précédente permet d'écrire des instructions **qui ne seront exécutées que lorsqu'on lance le script en fichier principal** et ne seront pas exécutées lors d'un import.

C'était le cas dans les modules créés dans le `TP Immeubles` !