1 Décomposition modulaire

Constat :
Dès qu'un programme est un peu long, il devient difficile de le modifier, de le comprendre, de le corriger. De plus, il existe souvent des parties de code que l'on souhaite utiliser à plusieurs endroits dans un programme.

Solution : la Modularité !

> L'approche modulaire d'un projet consiste à décomposer un programme en plusieurs composants réutilisables plus simples.

Ces composants pourront être codés séparément par différents programmeurs.
Ces composants pourront à leur tour être décomposés en sous-composants.

Avantages : Le code final est davantage lisible et compréhensible modifiable, facile à corriger.

Contraintes : Il est nécessaire de **documenter** et **spécifier** les liens entre les différentes parties pour rendre le projet maintenable.

2 Bibliothèques

La modularité amène également à utiliser des **bibliothèques** de fonctions (inutile de coder quelque chose qui a déjà été fait et qui est disponible).

> Une bibliothèque est un ensemble de fonctions, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.
>
> — évite de réécrire plusieurs fois les mêmes lignes de code.
> — évite de refaire ce qui a déjà été fait par d'autres.
> — simplifie l'écriture d'un programme complexe.
> — facilite les mises à jour internes.

En Python, il existe une bibliothèque standard qui contient de nombreux outils :

`https://docs.python.org/fr/3/library/index.html`

Il existe de nombreuses autres bibliothèques disponibles en langage Python que l'on peut importer à souhait.

Un point sur le vocabulaire utilisé :
- Un **module** est un fichier qui contient des variables, fonctions, objets, méthodes...
- Un **package** est un ensemble de dossiers (et sous dossiers). En général un package contient plusieurs modules.
- Une **bibliothèque** est constituée de plusieurs packages. On utile également le mot *library* pour désigner une bibliothèque.

3 Création de module (Python)

3.1 Documentation / Spécification

Il est bien entendu **possible de créer son propre module : c'est un simple fichier Python avec l'extension .py**.
Lorsqu'on créé un module, il faut le documenter de manière précise pour permettre son utilisation. Il faut également organiser des tests pour s'assurer de son bon fonctionnement.

Les commentaires dans un programme son indispensables pour vous, si vous y revenez plus tard, mais surtout pour les autres qui auront à utiliser votre programme.
On ne commente pas chaque ligne, mais plutôt un bloc d'instructions.
Au passage, commenter force à comprendre comment le code fonctionne : on ne peut pas commenter ce que l'on ne comprend pas.

Pour commenter une seule ligne, il suffit de la commencer par # :

```
# Ce programme permet de ...
```

Pour commenter sur plusieurs lignes ou mettre du code en attente, on encadre par des triples guillemets """ :

```
""" Un commentaire très long,
qui ne tient pas
sur une seule ligne.
"""
```

La **spécification** est la description précise d'une fonction permettant de l'utiliser sans savoir comment elle a été écrite :
- description de la tâche effectuée.
- description de ce qui est renvoyé.
- description des paramètres.
- les pré-conditions sur les paramètres.
- les post-conditions sur les résultats.

En Python, la spécification est inscrite dans la **docstring** en début du corps de la fonction (entre """ ... """).

On y accède avec la fonction **help** dans l'interpréteur.

Remarque : l'utilisation d'**assertions** peut permettre de contrôler les préconditions. Cette fonctionnalité est plus utile pour le programmeur en phase de développement que pour l'utilisateur.

Exemple d'utilisation de l'instruction `assert` :

```
>>>assert isinstance(n, int) and n > 0, "n est un entier strictement positif !"
```

3.2 Importer un module

Il existe plusieurs façon d'importer un module pour utiliser ses ressources dans un autre programme.

Pour illustrer ces méthodes, nous supposons que le module `mon_super_module` contient entre autres la fonction `ma_super_fonction(n)`, et la classe `Ma_super_classe`.

```
#### méthode 1 : import brut ####
# import du module :
import mon_super_module
# utilisation :
une_instance = mon_super_module.Ma_super_classe()
retour = mon_super_module.ma_super_fonction(5)

#### méthode 2 : import avec renommage ####
# import du module :
import mon_super_module as msm
# utilisation :
une_instance = msm.Ma_super_classe()
retour = msm.ma_super_fonction(5)

#### méthode 3 : import d'une seule fonction (+ renommage) ####
# import de la fonction spécifique :
from mon_super_module import ma_super_fonction as msf
# utilisation :
retour = msf(5)
""" remarque : impossible d'utiliser la classe
puisqu'on ne l'a pas importée !
"""

#### méthode 4 : import * : à éviter au maximum !!!! ####
# import du module dans l'espace de noms du programme en cours :
from  mon_super_module import *
# utilisation :
une_instance = Ma_super_classe()
retour = ma_super_fonction(5)
""" remarque : le risque est qu'une fonction ou une classe du module
porte le même nom qu'une fonction ou classe de votre programme
et on ne sait plus à qui on fait appel.
Ce problème a d'autant plus de risques de se produire
qu'on importe plusieurs modules.
"""
```

3.3 Analyser un module

Après avoir importé un module, on peut connaître les « ressources » (fonctions, classes, constantes) qu'il fournit avec la fonction **dir**.
On pourra remarquer dans l'exemple à suivre que Python crée automatiquement des attributs spéciaux (encadrés par des doubles underscores) dont l'utilité et le fonctionnement sortent du cadre du programme de Term NSI.

Pour connaître une description plus précise de ses ressources, on utilise la fonction **help** (utile si le module a été bien documenté !).

En poursuivant l'exemple précédent, voici ce qu'on pourrait obtenir :

```
>>> import mon_super_module as msm
>>> dir(msm)
['Ma_super_classe', '__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__', '__package__', '__spec__',
 'ma_super_fonction']
>>> help(msm)
Help on module mon_super_module:
NAME
    mon_super_module
DESCRIPTION
    Module de test : illustre le principe de la modularité.
CLASSES
    builtins.object
        Ma_super_classe

    class Ma_super_classe(builtins.object)
     |  Classe représentant un entier
     |
     |  Methods defined here:
     |  __init__(self)
     |      Instanciation de la classe
     |      *** crée un attribut x (type entier) de valeur 0
     |  ----------------------------------------------------------
     |  Data descriptors defined here:
     |  def
     |  __dict__
     |      dictionary for instance variables (if defined)
     |  __weakref__
     |      list of weak references to the object (if defined)
FUNCTIONS
    ma_super_fonction(n)
        Renvoie le double de n.
        *** entrée : n : entier
        *** sortie : 2*n : entier
```

On peut accéder à l'aide d'une seule fonction avec la syntaxe `help(msm.ma_super_fonction)`

3.4 Code du module d'exemple

Pour finir, voici le code source du fichier `mon_super_module.py` qui a servi d'exemple :

```
""" Module de test : illustre le principe de la modularité. """

def ma_super_fonction(n):
    """ Renvoie le double de n.
    *** entrée : n : entier
    *** sortie : 2*n : entier
    """
    assert isinstance(n, int), "n doit être un entier."
    return 2 * n

class Ma_super_classe:
    """ Classe représentant un entier """
    def __init__(self):
        """ Instanciation de la classe
        *** crée un attribut x (type entier) de valeur 0
        """
        self.x = 0

if __name__ == "__main__":
    retour = ma_super_fonction(5)
    print(retour) # affichage de 10
    a = Ma_super_classe()
    print(a.x) # affichage de 0
    retour = ma_super_fonction(2.6) # erreur : 2.6 n'est pas un entier
```

Explication de la dernière partie du fichier introduite par l'instruction :
`if __name__ == "__main__":`

Un module Python porte automatiquement un attribut spécial appelé `__name__`.
Cet attribut vaut le nom du fichier du module lorsqu'il est importé dans un autre programme (ex : `"mon_super_module"`). Mais si le module est exécuté en tant que programme indépendant, cet attribut vaut `"__main__"`.

Autrement dit, les instructions qui apparaissent après `if __name__ == "__main__":` ne sont prises en compte que si le module est exécuté de façon indépendante.
Cela permet donc en particulier de tester les fonctionnalités du module pendant sa conception.

4 API (Application Programming Interface)

4.1 Définition

Voici un extrait de la définition introductive de Wikipédia :

« Une **interface de programmation d'application**, ou API pour Application Programming Interface, est un **ensemble normalisé de classes, de méthodes, de fonctions et de constantes** qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels. Elle est offerte par une **bibliothèque logicielle** ou un **service we**b, le plus souvent accompagnée d'une **description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur**.

Dans l'industrie contemporaine du logiciel, les applications informatiques se servent de nombreuses interfaces de programmation, car la programmation se fait en réutilisant des briques de fonctionnalités fournies par des logiciels tiers. **Cette construction par assemblage nécessite pour le programmeur de connaître la manière d'interagir avec les autres logiciels, qui dépend de leur interface de programmation. Le programmeur n'a pas besoin de connaître les détails de la logique interne du logiciel tiers**, et celle-ci n'est pas nécessairement documentée par le fournisseur. Seule l'API est réellement nécessaire pour utiliser le système tiers en question.

Des logiciels tels que les systèmes d'exploitation, les systèmes de gestion de base de données, les langages de programmation, ou les serveurs d'applications comportent une ou plusieurs interface(s) de programmation. »

4.2 Quelques exemples (API Python ou web)

Les quelques API citées peuvent être utilement consultées dans le cadre de projets NSI.

1. **Microbit** : `https://microbit-micropython.readthedocs.io/en/v1.0.1/microbit_micropython_api.html`
   Décrit les fonctionnalités accessibles pour programmer un microcontrôleur microbit en Python.

2. **Tkinter** : `https://docs.python.org/fr/3/library/tk.html`
   API pour créer des interfaces graphiques en Python.

3. **Turtle** : `https://docs.python.org/fr/3/library/turtle.html`
   Expose une API pour faire des dessins dans un Canvas Tkinter.

4. **sqlite3** : `https://docs.python.org/fr/3/library/sqlite3.html`
   API pour interagir avec les bases de données SQLite en Python.

5. **API Web** : Il existe de nombreuses API orientées vers les applications Web. On peut citer par exemple des API météo qui permettent à tout un chacun de diffuser une info météo sur son site. `https://api.meteo-concept.com/`