11. Écriture de modules

Au premier semestre, vous avez appris à importer et utiliser des *modules* prédéfinis. Dans ce chapitre, vous allez maintenant apprendre à écrire vos propres modules. L'intérêt est de pouvoir garder dans un même fichier des constantes et fonctions intéressantes que vous auriez déjà définies, afin de les réinvestir ensuite facilement dans les prochains scripts que vous écrirez.

11.1. Écriture d'un module simple

11.1.1. Principe

D'un point de vue technique, un module Python n'est rien d'autre qu'un fichier d'extension `.py` contenant des déclarations de variables et de fonctions, comme tous les scripts que vous avez écrits jusque là. Le nom du module sera le nom que vous aurez donné à votre fichier. Si par exemple vous avez nommé votre fichier `foo.py`, alors le nom de votre module est `foo` et vous l'importerez avec l'instruction :

```
import foo
```

(ou l'une de ses déclinaisons introduites dans le chapitre 5).

Remarque. Attention, il faut que votre module se trouve dans le répertoire de travail courant du script qui l'importe. (Il existe des méthodes pour importer un module situé dans un dossier différent, mais nous ne les traiterons pas dans ce cours.)

De plus, Python va d'abord chercher parmi les modules installés sur la machine avant de regarder dans le répertoire de travail courant, donc n'utilisez pas le nom d'un module déjà existant (`math`, `numpy`, `random`, ...)

À tester. Dans un fichier nommé **`geometry2d.py`**, enregistrez les instructions ci-dessous.

```
ORIGIN = (0, 0)

def distance(A, B):
    return ((B[0] - A[0]) ** 2 + (B[1] - A[1]) ** 2) ** (1 / 2)
```

Ensuite, dans **un autre fichier python situé dans le même répertoire**, saisissez puis exécutez les instructions suivantes :

```
import geometry2d as geo

print(geo.ORIGIN)

A = (2, 3)
B = (6, 6)
print(geo.distance(A, B))
```

Que se passe-t-il si vous retirez le préfixe `geo.` devant les identifiants `ORIGIN` ou `distance` ?

On rappelle le principe de *localité* des identifiants créés au sein d'un module. Lorsque, depuis un script, on importe un module avec l'instruction **`import`**, tous les identifiants de variables et fonctions introduits dans le module seront préfixés par le nom du module (ou son alias si on utilise `as`). Ceci est mis en place

afin d'éviter des « collisions » dans le cas où un identifiant serait ré-utilisé par d'autres modules importés ou par le script lui-même.

À tester. Dans un même répertoire, créez les trois scripts suivants :

— `moduleA.py`

```
foo = 3
```

— `moduleB.py`

```
foo = 8
```

— `test.py`

```
import moduleA as A
import moduleB as B
foo = 5

print(A.foo)
print(B.foo)
print(foo)
```

Attention. Ceci n'est évidemment plus vrai si on importe seulement une partie du module via une instruction du type `from nom import id1, id2, ...` (voir chapitre 5).

Puisqu'un module est avant tout un script Python, rien n'empêche d'y importer d'autres modules si besoin. Le module `geometry2d.py` pourrait donc être amélioré de la manière suivante[1] :

```
ORIGIN = (0, 0)

def distance(A, B):
    return sqrt((B[0] - A[0]) ** 2 + (B[1] - A[1]) ** 2)
```

Exercice 11.1. Complétez le module `geometry2d` en lui ajoutant les fonctions suivantes.

1. Une fonction `translate(A, v)` qui renvoie le point obtenu en translatant le point `A` d'un vecteur `v` (on continuera à représenter points et vecteurs du plan par des tuples).
2. Une fonction `rotate(A, C, theta)` qui renvoie le point obtenu en appliquant à `A` une rotation d'angle `theta` (en radians) autour du point `C`.

   Le point B obtenu en appliquant à A une rotation d'angle θ autour de C a pour coordonnées :

$$\begin{cases} x_B = x_C + \cos(\theta)(x_A - x_C) - \sin(\theta)(y_A - y_C), \\ y_B = y_C + \sin(\theta)(x_A - x_C) + \cos(\theta)(y_A - y_C). \end{cases}$$

Écrivez ensuite un script Python qui utilise ces nouvelles fonctions après avoir importé `geometry2d` .

1. C'est une amélioration car pour des raisons techniques, le calcul de `x ** (1/2)` donne des résultats bien moins précis que ce qu'on obtiendrait avec la fonction `sqrt` du module `math` ou `numpy`.

11.1.2. Le mot-clé __name__

Attention : au moment où on importe un module, celui-ci est aussitôt interprété dans sa totalité. Cela signifie notamment que si le module contient des instructions affichant du contenu dans la console (ou dans une fenêtre externe), alors le simple fait de l'importer va déclencher cet affichage. C'est une pratique à éviter ! Un utilisateur qui va importer votre script pour utiliser vos fonctions n'a peut-être pas envie d'afficher quoi que ce soit dans la console.

À tester. Modifiez le module `geometry2d` en rajoutant la ligne 8 comme suit :

```
1 from numpy import sqrt
2
3 ORIGIN = (0, 0)
4
5 def distance(A, B):
6     return sqrt((B[0] - A[0]) ** 2 + (B[1] - A[1]) ** 2)
7
8 print("Module exécuté !")
```

Que se passe-t-il désormais lorsque vous exécutez le fichier `geometry2d.py` ? Et lorsque vous exécutez un autre script qui importe `geometry2d` ?

Si vous voulez malgré tout inclure des instructions d'affichage dans votre module pour effectuer des tests, vous pouvez utiliser la variable `__name__`. Cette variable est créée automatiquement à chaque fois qu'un fichier Python est interprété, et sa valeur est définie comme suit :

— si le fichier en question est le *point d'entrée* du programme, c'est-à-dire que c'est le fichier qui a été donné directement à l'interpréteur Python, alors la variable `__name__` contient la chaîne de caractères `"__main__"`,

— sinon, si le fichier a été appelé en tant que module par un autre script via l'instruction `import`, alors la variable `__name__` contient le nom du module.

À tester. Modifiez maintenant la ligne 8 du module `geometry2d` comme suit :

```
1 from numpy import sqrt
2
3 ORIGIN = (0, 0)
4
5 def distance(A, B):
6     return sqrt((B[0] - A[0]) ** 2 + (B[1] - A[1]) ** 2)
7
8 print(__name__)
```

Comparez les différences obtenues :

— en exécutant directement le fichier `geometry2d.py`,

— en exécutant un autre script qui importe `geometry2d`.

Ainsi, on pourra par exemple rajouter à la fin du fichier `geometry2d.py` le bloc suivant :

```
if __name__ == "__main__":
    # Quelques tests pour vérifier le fonctionnement du module.
    # Les lignes suivantes seront traitées seulement si on
    # exécute geometry2d.py directement, pas si geometry2d est
    # importé depuis un autre script.
    print("Test de la fonction distance :", distance((2, 3), (6, 6)))
```

11.2. Documentation d'un module

Si vous comptez distribuer votre module (ou même vous en resservir personnellement dans un futur plus ou moins proche), il est fortement recommandé de *documenter* celui-ci. Cela se fait en ajoutant des phrases pour présenter votre module ou expliquer l'utilité des fonctions qu'il contient. Actuellement, le module `geometry2d` qui a servi d'exemple au fil de la section précédente n'est pas documenté. La fonction `help` de Python ne fournit alors que des informations minimales sur celui-ci.

À tester. Importez votre module avec l'instruction `import geometry2d`. Observez ensuite le résultat des instructions suivantes :

1. `help(geometry2d)`
2. `help(geometry2d.distance)`

La documentation d'un module se fait en rajoutant dans le fichier des *doc-strings*, c'est-à-dire des chaînes de caractères placées à des endroits clés du programme. Plus précisément :

— une chaîne de caractères placée au tout début d'un fichier servira de description de celui-ci lorsqu'il sera importé comme module,

— une chaîne de caractères placée au tout début d'une déclaration de fonction servira de message d'aide pour la fonction en elle-même.

Le module `geometry2d` pourra ainsi être documenté comme dans la figure 11.1.

À tester. Réessayez les instructions `help(geometry2d)` et `help(geometry2d.distance)` une fois le module `geometry2d` documenté.

Exercice 11.2. Documentez correctement les fonctions `translate` et `rotate` écrites dans l'exercice précédent.

Exercices supplémentaires

Exercice 11.3. Rajoutez au module `geometry2d` les fonctions de votre choix et documentez-les. Quelques exemples pour vous inspirer :

— `point_symmetric(A, C)` : renvoie le symétrique du point A par rapport au point C,

— `regular_polygon(n, C, r)` : renvoie une liste de points formant un polygone régulier à n côtés, de centre le point C, et tel que la distance entre C et ses sommets soit égale à r,

— `dot_product(u, v)` : renvoie le produit scalaire entre deux vecteurs u et v.

```
"""
A module dedicated to planar geometry.
"""
from numpy import sqrt

ORIGIN = (0, 0)

def distance(A, B):
    """
    Compute the Euclidean distance between two points in the plane.

    Parameters
    ----------
    A : tuple
        First point.
    B : tuple
        Second point.

    Returns
    -------
    float
        The Euclidean distance between A and B.

    """
    return sqrt((B[0] - A[0]) ** 2 + (B[1] - A[1]) ** 2)
```

FIGURE 11.1. – Le module `geometry2d` complètement documenté.

Exercice 11.4. Créez un module `library` qui permet de manipuler des listes de livres au format CSV (telles que le fichier `books.csv` fourni sur Moodle). Votre module comportera les fonctions suivantes, correctement documentées :

- `add_entry(path, title, author, year, genre)` : ajoute à la fin du fichier une nouvelle ligne contenant les informations passées en paramètres,
- `get_library_size(path)` : renvoie le nombre d'entrées listées dans le fichier,
- `get_from_author(path, author)` : renvoie une liste contenant tous les titres de l'auteur passé en paramètre,
- `get_from_genre(path, genre)` : renvoie une liste contenant tous les couples (auteur, titre) correspondant au genre littéraire passé en paramètre,
- `get_from_year(path, year)` : renvoie une liste contenant tous les couples (auteur, titre) correspondant à une œuvre parue l'année passée en paramètre.

L'argument `path` *correspond à chaque fois au chemin du fichier CSV à traiter.*

À retenir

- Savoir écrire un module et l'importer depuis un autre script.
- Comprendre la valeur stockée dans la variable `__name__`.
- Savoir documenter un module et ses fonctions.
- Savoir accéder à la documentation d'un module ou de ses fonctions à l'aide de la fonction `help`.