Chapitre 6
Modularité

Ce chapitre est le sixième chapitre de l'année de Terminale NSI. Il est consacré à la **modularité** : l'art de découper un programme en briques logicielles réutilisables. Nous verrons ce qu'est une **API** (Application Programming Interface) et comment l'utiliser pour récupérer des données, puis comment **créer nos propres modules** en Python, les **documenter** correctement et les **importer** proprement dans nos programmes.

I. Les API (Application Programming Interface)

1. Qu'est-ce qu'une API ?

Définition 6.1.
Une **API** (Application Programming Interface) permet aux développeurs d'intégrer une application à une autre. Cela peut permettre par exemple de récupérer des données structurées depuis un site web pour les exploiter de manière automatisée dans un programme.

■ **Exemple 6.1.**
Une application météo sur un téléphone n'invente pas elle-même les températures : elle utilise l'API d'un site météorologique pour récupérer les données (température, humidité, prévisions) et les afficher dans une interface agréable. Les données sont récupérées de manière automatisée, sans qu'un humain ne les recopie à la main. ■

R Les API sont partout dans le monde numérique : cartes et itinéraires, réseaux sociaux, paiement en ligne, envoi de messages... Chaque service qui expose des données à d'autres programmes le fait au travers d'une API.

2. Exemple : l'API OpenWeatherMap

Nous cherchons à récupérer dans un programme Python la température d'une ville et à l'afficher. Pour cela, nous allons utiliser une API existante : les services d'**OpenWeatherMap** (faites une recherche internet pour accéder au site).

Si vous vous rendez sur le site, vous voyez bien les informations qui nous intéressent, mais sous cette forme, nous ne pouvons pas les exploiter facilement. Si vous cliquez

sur l'onglet API, vous verrez tout ce que le site propose comme API. Certaines sont gratuites d'accès, d'autres payantes. Dans tous les cas, pour en bénéficier, il faudra créer un compte sur le site. Pour découvrir le principe de l'utilisation de ces API, nous n'aurons pas besoin de compte : une **API de démonstration** suffit pour tester notre programme. Libre à vous, si vous souhaitez travailler sur des données réelles et actualisées, de créer un compte.

Nous utiliserons dans cet exemple l'API : `https://openweathermap.org/current`.

On accède à une API via une **URL** fournie par le site. Un exemple d'une telle URL est :

```
https://samples.openweathermap.org/data/2.5/weather?zip=14200,fr&appid=439
    d4b804bc8187953eb36d2a8c26a02
```

Exemple 6.2.

Cette URL contient plusieurs paramètres :

- `zip` : code postal et pays (ici 14200 en France) ;
- `appid` : clé fournie par le site quand vous avez créé un compte. Celle qui est proposée ici donne accès à des informations de démonstration. Elles ont le bon format pour nous permettre de développer et tester notre programme.

■

R Si vous ouvrez cette URL avec un navigateur (par exemple Firefox), celui-ci va présenter les données de manière lisible. En réalité, le fichier qui est envoyé via cette URL est un fichier au format **JSON**, qui ressemble à une structure de dictionnaire avec des associations clé-valeur.

3. Le format JSON

Définition 6.2.

Le **JSON** (JavaScript Object Notation) est un format de données textuel, lisible par une machine comme par un humain. Il est organisé en **couples clé-valeur**, comme un dictionnaire Python. Un fichier JSON peut contenir des valeurs de types variés : nombres, chaînes de caractères, listes, et d'autres dictionnaires (on parle alors de structure **imbriquée**).

Voici le contenu du fichier JSON renvoyé par l'URL précédente :

```
{
    "coord": {"lon": -122.09, "lat": 37.39},
    "weather": [{"id": 500, "main": "Rain",
                 "description": "light rain", "icon": "10d"}],
    "base": "stations",
    "main": {"temp": 280.44, "pressure": 1017, "humidity": 61,
             "temp_min": 279.15, "temp_max": 281.15},
    "visibility": 12874,
    "wind": {"speed": 8.2, "deg": 340, "gust": 11.3},
    "clouds": {"all": 1},
    "dt": 1519061700,
    "sys": {"type": 1, "id": 392, "message": 0.0027, "country": "US",
            "sunrise": 1519051894, "sunset": 1519091585},
    "id": 0,
    "name": "Mountain View",
    "cod": 200
}
```

▪ Exemple 6.3.
Ce JSON se lit comme un dictionnaire Python : la clé `"coord"` est associée au dictionnaire `{"lon": -122.09, "lat": 37.39}`, la clé `"main"` au dictionnaire `{"temp": 280.44, "pressure": 1017, ...}`, la clé `"weather"` à une *liste* contenant un dictionnaire, etc. Les données sont organisées en dictionnaires imbriqués. ▪

R Cette structure reprend exactement la notion de **dictionnaire** vue en Première : un ensemble d'associations clé-valeur, où une valeur peut elle-même être un dictionnaire ou une liste. Le JSON n'est qu'une représentation textuelle de cette structure, ce qui permet de l'échanger entre machines.

4. Le module requests

Grâce au module `requests`, Python peut récupérer les données au format JSON et les intégrer dans un dictionnaire.

▪ Exemple 6.4.
Voici un programme basique permettant de récupérer et d'afficher quelques informations. La fonction `get_weather` construit l'URL avec une chaîne formatée (f-string), envoie la requête avec `requests.get` et renvoie directement le dictionnaire obtenu avec `r.json()` : ▪

```
import requests

API_KEY = "439d4b804bc8187953eb36d2a8c26a02"

def get_weather(api_key, location):
```

```
    url = f"https://samples.openweathermap.org/data/2.5/weather?zip={
        location}&appid={api_key}"
    r = requests.get(url)
    return r.json()
```

La fonction `get_weather` va renvoyer directement un dictionnaire :

```
>>> weather = get_weather(API_KEY, "14000, fr")
>>> print(weather)
{'coord': {'lon': -122.09, 'lat': 37.39},
 'weather': [{'id': 500,
              'main': 'Rain',
              'description': 'light rain',
              'icon': '10d'}],
 'base': 'stations',
 'main': {'temp': 280.44,
          'pressure': 1017,
          'humidity': 61,
          'temp_min': 279.15,
          'temp_max': 281.15},
 'visibility': 12874,
 'wind': {'speed': 8.2, 'deg': 340, 'gust': 11.3},
 'clouds': {'all': 1},
 'dt': 1519061700,
 'sys': {'type': 1,
         'id': 392,
         'message': 0.0027,
         'country': 'US',
         'sunrise': 1519051894,
         'sunset': 1519091585},
 'id': 0,
 'name': 'Mountain View',
 'cod': 200}
```

Récupérer la température se fait alors via :

```
>>> print(weather["main"]["temp"])
```

■ **Exemple 6.5.**
En effet, notre dictionnaire sous la clé `"main"` renvoie un nouveau dictionnaire : ■

```
{'temp': 280.44,
 'pressure': 1017,
 'humidity': 61,
 'temp_min': 279.15,
 'temp_max': 281.15}
```

qui à son tour renvoie la température via sa clé `"temp"`.

R Le module `requests` n'appartient pas à la bibliothèque standard de Python : il doit être installé une seule fois sur la machine (avec la commande `pip install requests`) avant de pouvoir être importé. La bibliothèque standard, elle, est livrée avec Python : c'est le cas par exemple de `random` et de `math`.

II. Créer un module en Python

1. Principe général

Vous l'avez vu, le travail sur les API a été grandement facilité par l'utilisation du module `requests`, qui fait en réalité tout le travail pour nous. Grâce à ce module, notre programme est très concis et lisible. De plus, nous pouvons exploiter ce même module `requests` dans plusieurs programmes sans avoir à le redévelopper.

Nous allons voir maintenant comment créer nos propres modules en Python, afin d'enrichir le langage de nouvelles fonctions et d'objets réutilisables facilement.

Définition 6.3.
Un **module** est un ensemble de fonctions suffisamment générales pour être réutilisables dans plusieurs projets. Pour plus de clarté, on ne regroupera dans un même module que les fonctions relatives à une même fonctionnalité.

Un module s'importe par son **nom**, qui est le nom du fichier Python privé de l'extension `.py`.

■ **Exemple 6.6.**
Le module `random` regroupe des fonctions en lien avec l'aléatoire : `randint`, `choice`, `shuffle`, ...Ces fonctions sont stockées dans un fichier `random.py` quelque part sur le système de fichiers de votre ordinateur, et c'est ce fichier que Python charge lorsque vous écrivez `import random`. ■

Un module étant une **boîte noire** — on n'a en général pas le code des fonctions sous les yeux —, celui-ci doit exposer de manière claire au développeur son **interface**, c'est-à-dire :

— la liste des fonctions et leur rôle ;
— ce que chaque fonction prend en entrée ;
— ce que chaque fonction renvoie en résultat.

Définition 6.4.
L'**interface** d'un module est ce que le développeur doit connaître pour pouvoir l'utiliser : les noms des fonctions, leurs paramètres et ce qu'elles renvoient. L'**implémentation** est le code interne des fonctions, qui reste caché à l'utilisateur du module.

Pour décrire précisément ces informations, un formalisme particulier existe sous Python : les **docstrings**. Ainsi, un développeur pourra accéder à l'interface du module pour savoir comment utiliser chacune des fonctions.

2. Documenter un module : les docstrings

■ **Exemple 6.7.**
Tapez ce code dans une console Python : ■

```
>>> from random import randint
>>> help(randint)
```

Vous constaterez l'affichage de ce texte explicatif :

```
Help on method randint in module random:
randint(a, b) method of random.Random instance
Return random integer in range [a, b], including both end
points.
```

Cela est rendu possible par l'utilisation de **docstrings** à l'intérieur de la fonction `randint`.

■ **Exemple 6.8.**
Si vous tapez `help(random)`, vous aurez une description générale des fonctions disponibles dans le module. ■

R Il est intéressant de regarder les sources d'un module officiel de Python pour voir comment sont implémentées toutes ces docstrings : https://github.com/python/cpython/blob/master/

On retiendra :
— une **docstring** est une chaîne de caractères délimitée par des **doubles quotes** (en pratique, trois guillemets doubles ouvrants et trois fermants) ;
— un **module** commence par une docstring de présentation générale ;
— chaque **fonction** commence par une docstring présentant le rôle et le fonctionnement de celle-ci.

■ **Exemple 6.9.**
Voici l'implémentation de la fonction `randint` dans le module `random`, qui montre d'où provient le texte d'aide affiché par la fonction `help()` : ■

```
def randint(self, a, b):
    """Return random integer in range [a, b], including
    both end points.
    """
    return self.randrange(a, b + 1)
```

R Pour en savoir plus sur les docstrings, vous pouvez vous référer au guide de préconisations Python **PEP-257** : https://www.python.org/dev/peps/pep-0257/

3. Importer un module

Vous êtes confrontés à l'import de modules depuis la première ligne de Python que vous avez rencontrée pratiquement. En effet, si vous voulez utiliser des fonctions

aléatoires ou mathématiques de base, vous êtes obligé d'importer le module `random` ou `math`.

> **R ATTENTION :** n'importez **JAMAIS** un module par la commande
>
> ```
> from module import *
> ```
>
> et pourtant, c'est la première ligne que l'on trouve lorsque l'on programme en Python sur calculatrice. C'est une très mauvaise pratique menant à des bugs difficiles à déceler. En effet, cela importe toutes les fonctions du module sans distinction, avec le risque d'écraser d'autres fonctions de même nom dans d'autres modules.
>
> Vous trouverez beaucoup de code Python sur internet important les modules ainsi. C'est mal, ne le faites pas !

Il y a deux moyens d'importer proprement des modules :

■ **Exemple 6.10.**

Import ciblé : `from module import fonction_1, fonction_2, ...`

Avec cette syntaxe, on indique explicitement les fonctions que l'on souhaite utiliser. C'est très clair pour le lecteur et le développeur. Il n'y a pas de risques de confusion ou de conflits de noms, car on voit les fonctions que l'on importe. Pour utiliser les fonctions, il suffit de les invoquer simplement par leur nom : `fonction1(arguments)`.

■

```
from random import randint
a = randint(1, 100)
```

■ **Exemple 6.11.**

Import du module entier : `import module`

Cette syntaxe concise à l'import est aussi sans ambiguïté. Par contre, elle va imposer que chaque appel de fonction soit précédé du module en préfixe. Pour utiliser les fonctions, on les invoquera en faisant : `module.fonction1(arguments)`. ■

```
import random
a = random.randint(1, 100)
```

■ **Exemple 6.12.**

Import avec alias : il est possible, en cas de nom de module trop compliqué, de lui donner un alias plus court par commodité. ■

```
import random as rd
a = rd.randint(1, 100)
```

Instruction	Appel des fonctions	Quand l'utiliser
`import module`	`module.fonction(...)`	Import global, préfixe systématique
`from module import f1, f2`	`f1(...)`, `f2(...)`	Import ciblé, sans préfixe
`import module as m`	`m.fonction(...)`	Alias pour raccourcir le nom

R Quelle que soit la syntaxe choisie, importer un module **exécute** son code de haut en bas (c'est ainsi que les définitions de fonctions sont créées). D'où la bonne pratique : ne mettre au niveau global d'un module que des définitions, et non des instructions qui s'exécuteraient à l'import.

III. Boîte noire, interface et implémentation

Utiliser un module, c'est l'utiliser comme une **boîte noire** : on connaît ce que font les fonctions proposées (leur **interface**), mais pas nécessairement comment elles le font (leur **implémentation**). Cette séparation est un des intérêts majeurs de la modularité.

Exemple 6.13.
La fonction `math.sqrt` calcule une racine carrée. Que les développeurs de Python réécrivent son code interne (par exemple pour le rendre plus rapide), tant que le nom, les paramètres et le comportement restent identiques, aucun programme utilisant `math.sqrt` n'a besoin d'être modifié : l'interface n'a pas changé. ■

Exemple 6.14.
Un module personnel bien conçu se réutilise dans plusieurs programmes sans copier-coller. Si une fonction du module contient un bug, on corrige le bug une seule fois dans le module, et tous les programmes qui l'importent bénéficient de la correction.
■

R La distinction entre interface et implémentation est étudiée en détail dans le chapitre 3 de cette année. On y voit notamment que plusieurs implémentations différentes peuvent respecter la même interface.

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre : API, format JSON, création de modules, docstrings, bonnes pratiques d'import.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : syntaxe des imports, vocabulaire de la modularité, lecture de code court, prédiction d'affichage, rappels de Première sur les dictionnaires. Chaque question doit pouvoir être traitée en moins de trente secondes.

Exercice 6.1.

Dire si les affirmations suivantes sont vraies ou fausses, en justifiant brièvement.

1. Un module est un ensemble de fonctions réutilisables dans plusieurs programmes.
2. Le nom d'un module est le nom du fichier Python auquel on ajoute l'extension `.py`.
3. On importe un module avec le mot-clé `import`.
4. Le module `random` contient des fonctions liées à l'aléatoire.
5. Le nom du module `math` est le nom du fichier `math.py` privé de l'extension.

■

Exercice 6.2.

Quelle instruction importe le module `math` en entier ?

1. `import math`
2. `from math import *`
3. `import all math`
4. `include math`

■

Exercice 6.3.

Après `import math`, comment appelle-t-on la fonction `sqrt` ?

1. `math.sqrt(16)`
2. `sqrt(16)`
3. `math::sqrt(16)`
4. `math->sqrt(16)`

■

■ **Exercice 6.4.**
Après `from math import sqrt`, comment appelle-t-on la fonction `sqrt` ?

1. `math.sqrt(16)`
2. `sqrt(16)`
3. `from.sqrt(16)`
4. `import.sqrt(16)`

■

■ **Exercice 6.5.**
On exécute le programme suivant :

```
import random as rd
a = rd.randint(1, 6)
```

1. Comment appelle-t-on la fonction `randint` ?
2. Que se passerait-il si l'on écrivait `random.randint(1, 6)` à la place ? Nommer l'erreur.

■

■ **Exercice 6.6.**
Que se passe-t-il à l'exécution du programme suivant ?

```
import math
print(sqrt(16))
```

1. Une erreur `NameError` est levée : `sqrt` seul n'est pas défini.
2. Le programme affiche `4.0`.
3. Le programme affiche `16`.
4. Le module `math` est importé une deuxième fois.

■

■ **Exercice 6.7.**
Rappel de Première : on dispose du dictionnaire suivant :

```
weather = {"main": {"temp": 280.44}, "wind": {"deg": 340}}
```

Quelle expression renvoie la température ?

1. `weather["main"]["temp"]`
2. `weather["temp"]`
3. `weather[0]["temp"]`

4. `weather["main", "temp"]`

■

■ **Exercice 6.8.**
Rappel de Première : que va afficher le programme suivant ?

```
def meteo():
    return {"ville": "Caen", "temp": 18}

print(meteo()["ville"])
```

■

■ **Exercice 6.9.**
Le fichier `outils.py` contient la fonction `calculer_moyenne`, qui calcule la moyenne d'une liste de nombres. Que va afficher ce programme ?

```
from outils import calculer_moyenne
print(calculer_moyenne([10, 20, 30]))
```

■

■ **Exercice 6.10.**
Pourquoi déconseille-t-on généralement d'écrire `from math import *` ?

1. Tous les noms du module arrivent d'un coup : on ne sait plus d'où vient chaque fonction, et des noms existants peuvent être écrasés.
2. Cette syntaxe n'existe pas en Python.
3. Elle n'importe que la première fonction du module.
4. Elle rend le programme beaucoup plus lent à l'exécution.

■

Exercices d'application

Les exercices d'application demandent d'écrire des programmes et des modules complets : exploitation d'une API, analyse de données JSON, création de modules documentés et de programmes de test.

■ **Exercice 6.11.**
On reprend la fonction `get_weather` du cours, qui prend en entrée une clé API et

un code postal (avec le pays), et renvoie le dictionnaire météo complet.

1. Écrire le code qui récupère la météo pour la ville de Caen (code postal 14000, France) et affiche la **direction du vent** (en degrés).
2. Dans l'exemple du cours (ville de Mountain View), quelle est la direction du vent ?

■

■ **Exercice 6.12.**

On dispose du dictionnaire suivant, qui reprend la structure d'une réponse d'API météo :

```
donnees = {
    "ville": "Paris",
    "main": {"temp": 18.5, "humidity": 72},
    "weather": [{"description": "ciel degage"}]
}
```

1. Afficher le nom de la ville.
2. Afficher la température.
3. Afficher l'humidité.
4. Afficher la description du temps. Attention : la valeur associée à la clé `"weather"` est une *liste* contenant un dictionnaire.

■

■ **Exercice 6.13.**

Le programme suivant est mal écrit. Le corriger en utilisant les bonnes pratiques du chapitre (import ciblé ou import avec préfixe, jamais d'import étoile).

```
from random import *
from math import *

print(randint(1, 6))
print(sqrt(16))
print(random.randint(1, 6))
```

■

■ **Exercice 6.14.**

Écrire un module `aires` permettant de calculer les aires de figures géométriques usuelles :

— triangle ;
— disque ;
— carré ;
— rectangle ;
— parallélogramme ;
— losange.

Vous veillerez à respecter les consignes suivantes :
— à chaque figure correspondra une fonction du même nom (les noms seront écrits sans accent : `carre`, `parallelogramme`, ...) ;
— vous déciderez des paramètres pertinents à communiquer à chaque fonction pour le calcul de l'aire (par exemple, la fonction `triangle` prendra la base et la hauteur) ;
— le module et chaque fonction seront correctement documentés (docstrings), afin qu'un développeur ne connaissant pas votre module puisse l'utiliser facilement grâce à la fonction `help`.

Vous écrirez ensuite un petit programme afin de tester le fonctionnement de ce module.

```
def triangle(base, hauteur):
    """Renvoie l'aire d'un triangle."""
    ...
```

■

■ **Exercice 6.15.**

Écrire un module `conversions` contenant des fonctions de conversion de températures :
— `celsius_vers_fahrenheit(c)` : renvoie la température en degrés Fahrenheit, sachant que $F = \frac{9}{5}C + 32$;
— `fahrenheit_vers_celsius(f)` : renvoie la température en degrés Celsius, sachant que $C = \frac{5}{9}(F - 32)$.

Chaque fonction sera documentée par une docstring. Écrire ensuite un programme de test qui convertit 25 °C en degrés Fahrenheit, puis le résultat obtenu en degrés Celsius (on doit retomber sur 25). ■

■ **Exercice 6.16.**

Écrire un module `statistiques` contenant :
— `moyenne(liste)` : renvoie la moyenne des nombres de la liste ;
— `variance(liste)` : renvoie la variance de la liste, définie par $V = \frac{1}{n}\sum(x_i - m)^2$ où m est la moyenne ;
— `mediane(liste)` : renvoie la médiane de la liste (on pourra trier la liste avec `sorted` et distinguer les cas pair et impair).

Le module commencera par une docstring de présentation générale, et chaque fonction sera documentée. Écrire un programme de test utilisant la liste `[10, 12, 14, 16]` (moyenne 13, variance 5) puis la liste `[10, 20, 30]` (médiane 20). ■

■ **Exercice 6.17.**
En utilisant **Hourly Forecast 4 days** (https://openweathermap.org/api/hourly-forecast), afficher les prévisions météo pour les 4 prochains jours.

1. Se rendre sur la page de l'API pour observer la structure des données renvoyées (notamment la clé qui contient la liste des prévisions horaires).
2. Écrire un programme qui récupère ces données et affiche, pour chaque jour, la température prévue (on pourra se limiter à une prévision par jour).

■

Exercices de réflexion

Les exercices de réflexion demandent d'analyser des situations et de concevoir des modules plus complets, en argumentant sur les choix de conception.

■ **Exercice 6.18.**
Un module propose une fonction `trier(liste)` qui trie une liste dans l'ordre croissant. La version 1 du module utilise un tri par insertion ; la version 2 utilise un tri fusion.

1. Qu'est-ce qui change pour l'utilisateur du module entre la version 1 et la version 2 ?
2. Dans quels cas la différence entre les deux versions est-elle visible ?
3. Le développeur du module renomme ensuite sa fonction `trier` en `tri`. Que se passe-t-il pour les programmes qui utilisaient `trier` ? Justifier.

■

■ **Exercice 6.19.**
Rédiger un paragraphe argumenté (5 à 8 lignes) expliquant pourquoi l'instruction `from module import *` est une mauvaise pratique, en donnant un exemple concret de bug qu'elle peut provoquer. ■

■ **Exercice 6.20.**
On dispose de l'interface (sans le code) de deux fonctions d'un module `outils`,

décrites par leurs docstrings :

```
def mystere_1(texte):
    """Renvoie la chaine texte sans ses espaces."""

def mystere_2(texte, mot):
    """Renvoie True si mot apparait dans texte, sinon False."""
```

1. Sans voir l'implémentation, expliquer ce que font ces deux fonctions, puis donner un exemple d'appel et le résultat attendu pour chacune.
2. Proposer un jeu de tests permettant de vérifier que l'implémentation respecte bien son interface.
3. Pourquoi est-il possible d'utiliser ces fonctions sans connaître leur code interne ?

■

■ **Exercice 6.21.**

On souhaite gérer les notes d'une classe : calculer des moyennes, trier les élèves, afficher un bulletin.

1. Lister les fonctions qu'un module `notes` devrait proposer pour répondre à ce besoin. Pour chacune, préciser les paramètres et la valeur renvoyée (c'est-à-dire définir l'interface).
2. Écrire le début du module : la docstring de présentation générale et deux des fonctions de votre choix, correctement documentées.
3. Écrire un petit programme de test pour ces deux fonctions.
4. Expliquer en quoi le découpage en module rend ce travail plus simple qu'un programme unique, notamment si plusieurs élèves de la classe travaillent sur des parties différentes du projet.

■