# Chapitre 5 :la Modularité et la gestion des erreur en Python

Par Robert DIASSÉ

## 1. Introduction à la Modularité en Python

La modularité est un principe fondamental en programmation qui consiste à diviser un programme en plusieurs fichiers et fonctions réutilisables. Cela améliore la lisibilité, la maintenabilité et la réutilisation du code.

En Python, la modularité repose sur trois concepts principaux :

- L'utilisation des **modules** et **packages** pour structurer le code.
- La définition de **fonctions** pour éviter la redondance et améliorer l'organisation.
- La gestion des **erreurs et exceptions** pour un code robuste et fiable.

Ce cours couvre ces aspects en détail, avec des exemples pratiques et des exercices pour renforcer la compréhension.

---

## 2. Les Modules en Python

Un **module** en Python est un fichier contenant du code réutilisable (fonctions, classes, variables). Il permet de structurer un projet en séparant les fonctionnalités.

### 2.1 Création et importation d'un module

Un module est simplement un fichier `.py` contenant des définitions que l'on peut réutiliser dans d'autres scripts.

**Exemple : Création d'un module `math_util.py`**

```python
# math_util.py
def addition(a, b):
    return a + b

def soustraction(a, b):
    return a - b
```

**Exemple : Utilisation du module dans un autre fichier `main.py`**

```python
# main.py
import math_util

resultat = math_util.addition(5, 3)
print(f"Le résultat de l'addition est : {resultat}")
```

Dans cet exemple, le fichier `math_util.py` définit deux fonctions et le fichier `main.py` les importe et les utilise.

### 2.2 Importation avancée

Il est possible d'importer uniquement certaines fonctions d'un module.

```python
from math_util import addition

resultat = addition(10, 4)
print(f"Résultat : {resultat}")
```

On peut aussi renommer un module lors de son importation pour simplifier l'écriture.

```python
import math_util as mu

resultat = mu.soustraction(8, 3)
print(f"Résultat : {resultat}")
```

Enfin, pour importer toutes les fonctions d'un module :

```python
from math_util import *
```

Cette pratique est cependant déconseillée car elle peut provoquer des conflits de noms.

### 2.3 Vérification d'exécution d'un module

Python permet d'exécuter un module à la fois comme script principal et comme module importé. On utilise :

```python
if __name__ == "__main__":
    print("Ce module est exécuté directement")
```

Ainsi, le code sous cette condition ne s'exécutera que si le fichier est exécuté directement.

---

## 3. Les Fonctions en Python

Les **fonctions** permettent d'encapsuler du code réutilisable, améliorant la clarté et la modularité d'un programme.

### 3.1 Appel d'une fonction

```
def saluer(nom):
    """Affiche un message de salutation."""
    print(f"Bonjour, {nom} !")

saluer("Alice")
```

Explication :

- `def saluer(nom):` définit une fonction prenant un argument `nom`.
- `"""Affiche un message de salutation."""` est une **docstring**, utile pour documenter la fonction.
- `saluer("Alice")` appelle la fonction avec l'argument `"Alice"`.

### 3.2 Retourner une valeur

```
def carre(x):
    """Retourne le carré d'un nombre."""
    return x * x

resultat = carre(5)
print(f"Le carré de 5 est {resultat}")
```

Ici, `return x * x` permet de renvoyer une valeur qui peut être utilisée ailleurs dans le programme.

### 3.3 Les paramètres par défaut

Une fonction peut avoir des valeurs par défaut pour ses paramètres.

```
def multiplier(a, b=2):
    return a * b

print(multiplier(5))  # Utilise la valeur par défaut de `b`
print(multiplier(5, 3))  # Remplace `b` par 3
```

Si un argument n'est pas fourni lors de l'appel, la valeur par défaut est utilisée.

### 3.4 Les fonctions lambda

Les **fonctions anonymes** (ou **lambda**) permettent d'écrire des fonctions courtes sans les nommer.

```
carre = lambda x: x * x
print(carre(6))
```

Elles sont souvent utilisées avec `map()`, `filter()` ou `sorted()`.

---

# 4. Les Fonctions Avancées

Python propose des concepts avancés pour structurer son code de manière efficace.

## 4.1 Les fonctions comme arguments

```
def appliquer_operation(operation, a, b):
    return operation(a, b)

addition = lambda x, y: x + y
print(appliquer_operation(addition, 3, 4))  # Affiche 7
```

Une fonction peut être passée en paramètre à une autre fonction.

## 4.2 Les closures (fermetures)

Une **closure** est une fonction interne qui capture les variables locales de la fonction englobante.

```
def creer_multiplier(facteur):
    def multiplier(nombre):
        return nombre * facteur
    return multiplier

double = creer_multiplier(2)
print(double(5))  # Affiche 10
```

## 4.3 Les décorateurs

Les **décorateurs** modifient dynamiquement le comportement d'une fonction.

```
def decorateur(fonction):
    def wrapper():
        print("Avant l'exécution")
        fonction()
        print("Après l'exécution")
    return wrapper

@decorateur
def afficher():
    print("Bonjour !")
```

```
afficher()
```

Ici, `afficher()` est entourée du comportement défini par `decorateur`.

---

# 5. Gestion des Erreurs en Python

La gestion des erreurs permet de capturer et traiter les exceptions pour éviter que le programme ne plante.

## 5.1 Utilisation de `try ... except`

```
try:
    x = int(input("Entrez un nombre : "))
    print(f"Le double est {x * 2}")
except ValueError:
    print("Erreur : Vous devez entrer un nombre valide.")
```

Si l'utilisateur entre un texte au lieu d'un nombre, le programme affiche un message au lieu de planter.

## 5.2 Capture d'exceptions spécifiques

```
try:
    resultat = 10 / 0
except ZeroDivisionError:
    print("Erreur : Division par zéro interdite.")
except Exception as e:
    print(f"Une erreur est survenue : {e}")
```

On peut capturer des erreurs précises ou bien toutes les erreurs avec `Exception`.

## 5.3 Utilisation de `finally`

Le bloc `finally` s'exécute toujours, même en cas d'erreur.

```
try:
    fichier = open("test.txt", "r")
except FileNotFoundError:
    print("Fichier introuvable.")
finally:
    print("Opération terminée.")
```

---

# 6. Exercices Pratiques

1. **Créer un module `calcul.py`** avec les fonctions `addition()`, `soustraction()`, `multiplication()` et `division()`.
2. **Créer un décorateur `log_execution()`** qui affiche "Début d'exécution" et "Fin d'exécution" autour d'une fonction.
3. **Écrire une fonction `division_secure(a, b)`** qui capture `ZeroDivisionError` et retourne `None` au lieu d'une erreur.

Ce cours couvre les bases et aspects avancés de la modularité en Python, permettant une meilleure organisation et robustesse du code.