

# Fonction `main`

---

En Python, ce bloc de code :

```
if __name__ == "__main__":  
    main()
```

sert à **déterminer comment un fichier Python est utilisé** :

- **exécuté directement** ou **importé comme module**.

`__name__`

`__name__` est une variable spéciale définie par Python.

- Si le fichier est **exécuté directement**, alors :

```
__name__ == "__main__"
```

- Si le fichier est **importé depuis un autre fichier**, alors :

```
__name__ == "nom_du_fichier"
```

`if __name__ == "__main__":`

Il permet de dire :

Ce code ne doit s'exécuter **que si ce fichier est lancé directement**, pas s'il est importé.

## Exemple

fichier `calcul.py`

```
def main():  
    print("Programme principal")  
  
def addition(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    main()
```

## Exécution directe

```
python calcul.py
```

Résultat :

```
Programme principal
```

fichier `test.py`

```
import calcul  
  
print(calcul.addition(2, 3))
```

Résultat :

```
5
```

`main()` n'est PAS exécuté, car `calcul.py` est importé, pas lancé directement.

## Utilité

- Sépare le **code réutilisable** (fonctions, classes) du **code d'exécution** (tests, interface, logique principale)
- Rend le fichier importable sans effets de bord
- Très utilisé dans les projets et les bibliothèques Python

## Résumé

- `main()` contient le point d'entrée du programme
- `if __name__ == "__main__":` empêche son exécution lors d'un import
- Indispensable pour écrire du **code propre et réutilisable**

---

Exemple AVEC `if __name__ == "__main__"`

`outil.py`

```
def dire_bonjour(nom):  
    print(f"Bonjour {nom}")  
  
def main():
```

```
    dire_bonjour("Carnus")

if __name__ == "__main__":
    main()
```

### Exécution directe

```
python outil.py
```

#### Sortie :

```
Bonjour Carnus
```

### autre.py

```
import outil

outil.dire_bonjour("Rodez")
```

#### Sortie :

```
Bonjour Rodes
```

- `main()` ne s'exécute pas à l'import
- On contrôle exactement ce qui se passe

### Exemple SANS `if __name__ == "__main__"`

#### outil\_mauvais.py

```
def dire_bonjour(nom):
    print(f"Bonjour {nom}")

# Code exécuté automatiquement
dire_bonjour("Carnus")
```

### Exécution directe

```
python outil_mauvais.py
```

**Sortie :**

Bonjour Carnus

**autre.py**

```
import outil_mauvais  
  
outil_mauvais.dire_bonjour("Rodez")
```

**Sortie :**

Bonjour Carnus  
Bonjour Rodez

**Problème :**

- Le message "**Bonjour Carnus**" s'affiche **sans qu'on le veuille**
- L'import déclenche du code non désiré

## Exemple

### Mauvaise pratique

```
print("Connexion à la base de données...")
```

Ce code s'exécute à **chaque import**, même pour un test ou un script secondaire.

### Bonne pratique

```
def main():  
    print("Connexion à la base de données...")  
  
if __name__ == "__main__":  
    main()
```

La connexion ne se fait **que si le fichier est lancé directement**

## Conclusion

Imaginer un fichier Python comme un **outil** :

- `if __name__ == "__main__"` = bouton **ON**
- Importer = ouvrir la boîte sans appuyer sur ON
- **Tout ce qui doit s'exécuter automatiquement va dans `main()`**
- **Tout ce qui doit être réutilisé reste hors du `main()`**

**Sans** `if __name__ == "__main__"` → Importer le module = exécuter le code situé au niveau *global*

**Avec** `if __name__ == "__main__"` → Importer le module ≠ exécuter le code principal → C'est l'utilisateur (ou le script appelant) qui décide quoi lancer

- **Tout le code au niveau global** est exécuté à l'import
- Le `if __name__ == "__main__"` **protège** le code d'exécution
- Les **fonctions et classes** restent toujours accessibles

---

## Exemple

```
# fichier a.py
print("chargement du module")

def f():
    print("fonction f")

if __name__ == "__main__":
    print("exécution principale")
```

```
import a
```

**Affiche :**

```
chargement du module
```

```
python a.py
```

**Affiche :**

```
chargement du module
exécution principale
```

## Exercice 1

module.py

```
print("Module chargé")

def f():
    print("fonction f")

if __name__ == "__main__":
    print("programme principal")
```

Question :

Que s'affiche lors de l'exécution de :

```
import module
```

## Exercice 2

module.py

```
x = 0
x = x + 1
```

test.py

```
import module
import module
print(module.x)
```

Que s'affiche lors de l'exécution de test.py ?

## Résumé

- L'import **exécute** le code global
- `__main__` protège seulement ce qui est dedans
- Un module n'est chargé **qu'une fois**

## Exercice 3

Fichier module.py

```
print("A")

x = 1

def f():
    print("B")

if __name__ == "__main__":
    print("C")
    f()
```

### Question 1

Que s'affiche-t-il lorsque l'on exécute directement :

```
python module.py
```

### Question 2

Que s'affiche-t-il lorsque l'on exécute le programme suivant ?

**test.py**

```
import module
module.f()
```

### Question 3

Que s'affiche-t-il si l'on modifie **test.py** ainsi ?

```
import module
import module
module.f()
```

## Corrigé (Exercie 3)

### Question 1 — Exécution directe

- Le fichier est exécuté directement
- `__name__ == "__main__"` est **vrai**

Ordre d'exécution :

1. `print("A")`

2. `x = 1`
3. `print("C")`
4. appel de `f()` → `print("B")`

**Affichage :**

```
A
C
B
```

### Question 2 — Import simple

- Le module est **importé**
- `__name__ == "__main__"` est **faux**

Ordre d'exécution :

1. `print("A")` (code global exécuté)
2. `x = 1`
3. le bloc `if` est **ignoré**
4. appel explicite de `module.f()` → `print("B")`

**Affichage :**

```
A
B
```

### Question 3 — Double import

- Python n'exécute un module **qu'une seule fois**
- Le second `import module` ne fait rien

Ordre :

1. `print("A")` (une seule fois)
2. `module.f()` → `print("B")`

**Affichage :**

```
A
B
```

## Exercice 4

Fichier `a.py`

```
print("A1")

def f():
    print("A2")

if __name__ == "__main__":
    print("A3")
```

Fichier **b.py**

```
print("B1")

import a

def g():
    print("B2")
    a.f()

if __name__ == "__main__":
    print("B3")
    g()
```

### Question 1

Que s'affiche-t-il lorsque l'on exécute :

```
python a.py
```

### Question 2

Que s'affiche-t-il lorsque l'on exécute :

```
python b.py
```

### Question 3

Que s'affiche-t-il lorsque l'on exécute dans l'interpréteur Python :

```
import b
b.g()
```

## Corrigé (Exercice 4)

### Question 1 — `python a.py`

- `a.py` est exécuté directement
- `__name__ == "__main__"` est vrai

**Affichage :**

```
A1
A3
```

### Question 2 — `python b.py`

Ordre réel d'exécution :

1. `print("B1")`
2. `import a`
  - `print("A1")`
  - le `if __name__ == "__main__"` de `a.py` est ignoré
3. définition de `g()`
4. `__name__ == "__main__"` est vrai pour `b.py`
5. `print("B3")`
6. appel de `g()` :
  - `print("B2")`
  - appel de `a.f()` → `print("A2")`

**Affichage :**

```
B1
A1
B3
B2
A2
```

### Question 3 — `import b` puis `b.g()`

#### Étape 1 : `import b`

- `b.py` est importé (pas exécuté directement)

Ordre :

1. `print("B1")`
2. `import a`
  - `print("A1")`
3. le `if __name__ == "__main__"` de `b.py` est ignoré

### Étape 2 : `b.g()`

- `print("B2")`
- `a.f()` → `print("A2")`

### Affichage total :

```
B1
A1
B2
A2
```

---

---

## Lien avec les classes

### Modules

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

« Le module regroupe des fonctions qui vont ensemble »

### Classes

#### Sans attributs

```
# calculatrice.py
class Calculatrice:
    def addition(self, a, b):
        return a + b
```

#### Avec attributs

```
class Calculatrice:
    def __init__(self):
        self.resultat = 0
```

```
def addition(self, a, b):  
    self.resultat = a + b  
    return self.resultat
```

« La classe regroupe des fonctions **et** des données »

- Même logique, **nouvel outil**.

« Une classe, c'est comme un module plus intelligent. »

---