

Mise au point et débogage des programmes

Fiche de synthèse

① PRÉVENIR LES ERREURS

Un code lisible se débogue plus facilement : noms de variables significatifs, fonctions courtes, spécifiées et documentées par une docstring.

Définition 1 (Annotation de type) Une **annotation** indique le type attendu d'un paramètre et de la valeur renvoyée. Python ne la vérifie pas à l'exécution : elle documente la fonction et sert à l'éditeur et aux outils comme `mypy`.

```
def moyenne(notes: list[float]) -> float:
    return sum(notes) / len(notes)
```

② LIRE UN MESSAGE D'ERREUR

Le message donne le **type** de l'erreur et la **ligne** où elle est apparue, qui n'est pas forcément la ligne fautive : une parenthèse oubliée à la ligne précédente, par exemple, signale l'erreur plus loin.

Erreur	Cause
<code>SyntaxError</code>	mot mal écrit, symbole manquant
<code>IndexError</code>	indice hors de la liste
<code>NameError</code>	nom de variable inconnu
<code>TypeError</code>	opération sur un type qui ne la permet pas, par exemple des parenthèses après un entier
<code>ZeroDivisionError</code>	division par zéro

③ TECHNIQUES DE DÉBOGAGE

- **Affichages** : des `print` provisoires montrent la valeur des variables aux endroits critiques.
- **Assertions** : `assert condition, "message"` arrête le programme si la condition est fausse. On vérifie ainsi des **préconditions**, au début d'une fonction, et des **post-conditions**, juste avant le `return`.
- **Débogueur** : l'instruction `breakpoint()` met l'exécution en pause. La commande `p n` affiche la variable `n`, `n` exécute la ligne suivante, `s` entre dans la fonction appelée, `c` continue jusqu'au point d'arrêt suivant.

④ EXCEPTIONS

Définition 2 (Exception) Une **exception** est une erreur détectée pendant l'exécution. Si rien ne l'intercepte, le programme s'arrête et affiche son message.

```
try:
    x = int(input("Saisir x : "))
    print(f"1/x = {1 / x}")
except ValueError:
    print("Ce n'est pas un entier.")
except ZeroDivisionError:
    print("Division par zéro impossible.")
finally:
    print("Fin du calcul.")
```

- Python exécute le bloc `try` ; à la première erreur, il exécute le bloc `except` du type correspondant au lieu d'arrêter le programme.
- Le bloc `finally` s'exécute dans tous les cas, erreur ou non.
- `raise ValueError("message")` lève soi-même une exception.
- On crée une exception personnalisée par héritage : `class SuperstitionError(Exception): pass`.

⑤ CAUSES FRÉQUENTES D'ERREURS

- **Débordement de tableau** : accès à un indice qui n'existe pas. Penser à la liste vide, et tester l'indice *avant* l'accès : `i < len(lst) and lst[i] < 100`, car `and` n'évalue pas sa seconde condition si la première est fausse.
- **Typage** : `[1, 2] * 2` vaut `[1, 2, 1, 2]` et non `[2, 4]`. En C, un entier `int` peut dépasser sa capacité et devenir négatif.
- **Effet de bord** : une fonction qui modifie une variable extérieure, comme une liste globale, rend le programme difficile à suivre.
- **Flottants** : `while x != y` : peut ne jamais s'arrêter à cause des arrondis ; on écrit `while abs(x - y) > 1e-10`.
- **Conditions** : oublier un cas, ou se tromper entre inégalité stricte et large, entre `and` et `or`. Par exemple, `not a and not b` équivaut à `not (a or b)`.

⑥ TESTS

Définition 3 (Jeu de tests) Un **jeu de tests** est l'ensemble des valeurs utilisées pour tester un programme. Il comprend les cas courants et les **cas limites** : liste vide, valeurs minimale et maximale, valeur interdite.

Avec `doctest`, les exemples de la docstring servent de tests :

```
def fact(n):
    """Renvoie n! pour n entier naturel.

    >>> [fact(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    """
    assert n >= 0, "n doit être positif"
    if n == 0:
        return 1
    return n * fact(n - 1)

import doctest
doctest.testmod() # muet si tout passe
```

Avec `pytest`, chaque fonction dont le nom commence par `test` est un test, lancé par la commande `pytest test_fact.py` :

```
import pytest
from mon_fichier import fact

def test_fact_5():
    assert fact(5) == 120

def test_fact_negatif():
    with pytest.raises(AssertionError):
        fact(-2)
```

Avec `unittest`, les tests sont les méthodes d'une classe qui hérite de `unittest.TestCase` et emploient `self.assertEqual(a, b)`, `self.assertTrue(a)` ou `self.assertRaises(Erreur)`.

Le **développement piloté par les tests** consiste à écrire les tests d'abord, puis le code qui les fait passer.

⑦ PERFORMANCES

Pour mesurer la durée d'un appel à une fonction additionner :

```
import time

debut = time.perf_counter()
resultat = additionner(1_000_000)
fin = time.perf_counter()
print(f"Durée : {fin - debut} s")
```

Le **profilage**, lancé par `python3 -m cProfile fichier.py`, donne pour chaque fonction le nombre d'appels et le temps passé : on optimise d'abord la fonction qui consomme le plus de temps.

À retenir.

- Le message d'erreur donne le type de l'erreur et la ligne où elle est apparue, pas toujours celle qui est fautive.
- Une assertion vérifie une condition pendant le développement ; `try/except` traite une erreur sans arrêter le programme.
- Un jeu de tests couvre les cas courants et les cas limites ; `doctest`, `pytest` et `unittest` les automatisent.
- On ne compare jamais deux flottants avec `==` ou `!=`.
- Le profilage désigne la fonction à optimiser en priorité.