Mise au point et débogage des programmes

Langages et programmation

1 Lisibilité du code

Un code simple, clair et lisible sera beaucoup plus facile à déboguer :

— choisir des **noms de variables significatifs** (respecter les conventions `snake_case` en Python) ;

— **spécifier et documenter** les fonctions (docstrings) ;

— privilégier un **découpage en fonctions** simples plutôt que de gros blocs de code.

1.1 Annotations de types

Python permet d'**annoter** les paramètres et la valeur de retour d'une fonction avec le type attendu :

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

L'interpréteur n'en tient pas compte (aucune vérification à l'exécution), mais ces annotations documentent la fonction, sont lues par l'éditeur (complétion, signalement d'un appel avec un mauvais type) et par des outils de vérification comme `mypy`. Elles complètent la docstring et les assertions : la docstring dit ce que fait la fonction, l'annotation sur quels types, l'assertion vérifie une propriété à l'exécution.

2 Outils de mise au point

2.1 Messages d'erreur

Le message d'erreur de Python indique le **type d'erreur** et la **ligne** où l'erreur est apparue. Attention : ce n'est pas forcément la ligne fautive (une parenthèse manquante à la ligne précédente, par exemple).

Quelques messages courants :

— `SyntaxError: invalid syntax` : syntaxe incorrecte (mot mal orthographié, symbole manquant) ;

— `IndexError: list index out of range` : débordement dans une liste ;

— `TypeError: 'int' object is not callable` : parenthèses après un entier ;

— `TypeError: 'float' object is not subscriptable` : crochets après un flottant.

2.2 Affichage avec `print`

La méthode la plus classique de débogage consiste à insérer des instructions `print` provisoires pour afficher les valeurs des variables à des points critiques :

```python
def recherche(L, x):
    for i in range(len(L)):
        print(f"i={i}, L[i]={L[i]}")  # affichage de débogage
        if L[i] == x:
            return i
    return -1
```

2.3 Assertions

Une assertion interrompt l'exécution si la condition n'est pas vérifiée. On les utilise pour vérifier des **préconditions** (avant le traitement) et des **postconditions** (avant le `return`) :

```python
def moyenne(L):
    assert len(L) > 0, "La liste ne doit pas être vide"
    return sum(L) / len(L)
```

2.4 Le débogueur Python

L'instruction `breakpoint()` (Python 3.7+) met l'exécution en pause et ouvre le débogueur interactif `pdb`. Commandes principales :

— `n(ext)` : exécute la ligne en cours ;

— `s(tep)` : entre à l'intérieur d'un appel de fonction ;

— `p variable` : affiche la valeur d'une variable ;

— `c(ontinue)` : reprend l'exécution jusqu'au prochain point d'arrêt ;

— `r(eturn)` : continue jusqu'au `return` de la fonction en cours.

3 Gestion des exceptions

3.1 Le mécanisme `try/except`

Une exception est une erreur détectée à l'exécution. Plutôt que de laisser le programme s'arrêter brutalement, on peut **capturer** l'exception :

```python
try:
    x = int(input("Saisir x : "))
    print(f"1/x = {1 / x}")
except ValueError:
    print("Erreur de saisie : ce n'est pas un entier")
except ZeroDivisionError:
    print("Impossible de diviser par zéro")
finally:
    print("Fin du calcul")
```

Le bloc `finally` est exécuté dans tous les cas (erreur ou non).

3.2 Lever une exception avec `raise`

On peut lever soi-même une exception :

```python
def factorielle(n):
    if n < 0:
        raise ValueError("n doit être un entier positif ou nul")
    return 1 if n == 0 else n * factorielle(n - 1)
```

3.3 Exceptions personnalisées

On peut définir ses propres exceptions en héritant de la classe `Exception` :

```python
class StockInsuffisantError(Exception):
    pass

def commander(stock, quantite):
    if quantite > stock:
        raise StockInsuffisantError(f"Stock={stock}, demandé={quantite}")
```

④ Causes typiques de bugs

4.1 Problèmes liés au typage

En Python, `[1, 2, 3] * 2` donne `[1, 2, 3, 1, 2, 3]` (concaténation) et non `[2, 4, 6]`. En C, l'affectation d'un `float` dans un `int` tronque la partie décimale ; un dépassement de capacité lors d'une telle conversion a causé l'explosion d'Ariane 5 en 1996.

4.2 Effets de bord non désirés

Une fonction qui modifie un état extérieur à son environnement local (variable globale, liste passée en paramètre) crée un **effet de bord**. Ces modifications cachées sont source de bugs difficiles à repérer.

4.3 Débordements dans les tableaux

Toujours vérifier qu'on ne sort pas du tableau, et traiter le cas du tableau vide. Attention à l'ordre des conditions :

```python
# Incorrect : accède à lst[i] avant de vérifier i < len(lst)
while lst[i] < 100 and i < len(lst):

# Correct : vérifie d'abord que l'indice est valide
while i < len(lst) and lst[i] < 100:
```

4.4 Comparaisons entre flottants

Une boucle `while x != y` risque de ne jamais se terminer si `x` et `y` sont des flottants (erreurs d'arrondi). On utilise plutôt :

```python
while abs(x - y) > 1e-10:
    # ...
```

5 Tests

5.1 Principes

On teste son code régulièrement, morceau par morceau. Un bon **jeu de tests** explore les situations critiques : valeurs minimales, maximales, cas limites (liste vide, division par zéro...).

Le **TDD** (*Test Driven Development*) préconise d'écrire les tests *avant* le code.

5.2 Module `doctest`

Les tests sont écrits directement dans la docstring de la fonction :

```
def fact(n):
    """
    Renvoie n! pour n entier positif ou nul.

    >>> fact(5)
    120
    >>> [fact(n) for n in range(7)]
    [1, 1, 2, 6, 24, 120, 720]
    """
    assert n >= 0, "n doit être positif ou nul"
    return 1 if n == 0 else n * fact(n - 1)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

5.3 Module `unittest`

Les tests sont organisés dans une classe héritant de `unittest.TestCase`, dans un fichier qui importe la fonction `fact` depuis le fichier `mon_fichier.py` où elle est définie :

```
import unittest
from mon_fichier import fact

class TestFact(unittest.TestCase):
    def test_fact_5(self):
        self.assertEqual(fact(5), 120)

    def test_fact_0(self):
        self.assertEqual(fact(0), 1)

    def test_fact_negatif(self):
        with self.assertRaises(AssertionError):
            fact(-2)

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

Méthodes d'assertion principales : `assertEqual(a, b)`, `assertTrue(a)`, `assertFalse(a)`, `assertRaises(erreur)`.

5.4 Module `pytest`

`pytest` offre une syntaxe plus légère, avec de simples fonctions et le mot-clef `assert` :

```python
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)
```

On exécute les tests avec la commande `pytest test_fact.py`.

6 Optimisation des performances

6.1 Chronométrage

Le module `time` permet de chronométrer une portion de code :

```python
import time

debut = time.perf_counter()
resultat = ma_fonction(1_000_000)
fin = time.perf_counter()

print(f"Durée : {fin - debut:.4f} s")
```

6.2 Profilage

Le profilage avec `cProfile` détaille, pour chaque fonction, le nombre d'appels et le temps passé :

```python
# En ligne de commande :
# $ python3 -m cProfile mon_fichier.py
```

Le profilage identifie précisément les **goulots d'étranglement** : les fonctions sur lesquelles concentrer les efforts d'optimisation.