Test et Débogage en Python

1. Pourquoi tester un programme ?

Tester permet de :

- S'assurer que le programme fait ce qu'on attend (conformité fonctionnelle)
- Détecter les erreurs (failles de logique, fautes de frappe, mauvaise compréhension du langage)
- Gagner en confiance lors de la modification du code (tests de régression)
- Améliorer la qualité logicielle globale

2. Les types d'erreurs possibles

Type	Description
Erreur de conception	Algorithme incorrect, spécifications mal interprétées
Erreur de réalisation	Faute de frappe, mauvaise syntaxe ou mauvaise utilisation du langage
Erreur d'exécution	Division par zéro, dépassement de mémoire, boucle infinie…

Une **erreur non traitée devient un bug** (bogue).

3. Test de conformité

Exemple en Python :

```
import math

def racine(x):
    assert x >= 0, "x doit être positif"
    return math.sqrt(x)
```

Un **test de conformité** vise à vérifier qu'une fonction ou un programme se comporte comme prévu, en respectant ses spécifications. Dans l'exemple donné, on teste la fonction racine(x) qui calcule la racine carrée de x.

- **Spécifications :**
  - La fonction prend un nombre x en entrée.
  - Elle vérifie que x est positif (sinon, elle lève une erreur avec `assert`).
  - Elle retourne la racine carrée de `x (math.sqrt(x))`.

Objectif du test :
On vérifie que la fonction donne les résultats attendus pour certaines entrées :

Vérifier que `racine(4)` retourne bien `2`, etc.

```
assert racine(4) == 2   # Vérifie que √4 = 2
assert racine(0) == 0   # Vérifie que √0 = 0
```

- `assert` condition, message (optionnel) :
  - Si la condition est vraie, le programme continue.
  - Si elle est fausse, Python lève une exception `AssertionError` (ici, avec un message personnalisé).

Objectif des tests

- racine(4) == 2 :
  On s'assure que la fonction retourne bien 2 pour x = 4 (car √4 = 2).
- racine(0) == 0 :
  On vérifie le comportement pour x = 0 (cas limite).

Pourquoi utiliser `assert` ?

- **En développement** :
  Les `assert` aident à détecter rapidement des erreurs de logique (ex : un résultat inattendu).
- **En production** :
  Il est préférable d'utiliser des exceptions classiques (`if ... raise ValueError`) car les `assert` peuvent être désactivés (option -O en Python).

Limites

- Ces tests sont basiques. En pratique, on utilise des `frameworks` comme `unittest` ou `pytest` pour des tests plus complets.
- Ils ne couvrent pas tous les cas (ex : x négatif, nombres décimaux, etc.).

Exemple complet amélioré

```
import math

def racine(x):
    if x < 0:
        raise ValueError("x doit être positif")
    return math.sqrt(x)

# Tests
assert racine(4) == 2
assert racine(0) == 0
assert racine(9) == 3
```

- **Test de conformité** = Vérifier que le code respecte les spécifications.
- `assert` = Outil simple pour des vérifications rapides (à éviter en production).
- **Cas d'usage** = Tester des valeurs normales (4 → 2) et limites (0 → 0).

4. Tests unitaires : principe

Un test unitaire vérifie qu'une **fonction** (ou un petit module) fonctionne correctement **isolément** du reste de l'application.

Exemple :

```
import unittest
from math import sqrt

def racine(x):
    if x < 0:
        raise ValueError("x doit être positif")
    return sqrt(x)

class TestRacine(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(racine(9), 3)
        self.assertAlmostEqual(racine(2), sqrt(2))

    def test_zero(self):
        self.assertEqual(racine(0), 0)

    def test_negative(self):
        with self.assertRaises(ValueError):
            racine(-1)

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

5. Stratégies de test

Boîte noire

- On **ignore l'implémentation**.
- On teste les **entrées/sorties** selon les **spécifications**.
- Exemple : `racine(4)` doit renvoyer `2`, `racine(-1)` doit lever une erreur.

Boîte blanche

- On **connaît le code**.
- On teste **toutes les branches**, **tous les cas limites**.
- Objectif : maximiser la **couverture du code**.

6. Types de tests

Type	Objectif
✓ Test unitaire	Vérifier une **fonction individuelle**
🔗 Test d'intégration	Vérifier l'interaction entre plusieurs modules
💻 Test système	Vérifier l'ensemble de l'application en conditions réelles

✓ Test d'acceptation	Valider le produit selon les exigences du client
🔁 Test de régression	Vérifier que les modifications ne cassent pas l'existant

7. Sélection des cas de test

- **Cas nominal** : les valeurs classiques (ex. `racine(4)`)
- **Cas limite** : les bornes (ex. `racine(0)`)
- **Cas erroné** : les valeurs invalides (ex. `racine(-1)`)
- **Cas aléatoire** : valeurs diverses pour détecter des comportements inattendus

Tous les tests passés ≠ absence de bogue. Mais des tests bien choisis augmentent la confiance.

8. Débogage

Outils :

- **print()** : simple mais utile pour tracer l'exécution
- **assert** : pour détecter les violations de conditions attendues
- **IDE avec debugger** : pas à pas, variables, points d'arrêt
- **pdb** : debugger en ligne de commande (`import pdb; pdb.set_trace()`)

Exemple d'utilisation de `assert` :

```
def inverse(x):
    assert x != 0, "Division par zéro"
    return 1 / x
```

9. Utiliser `pytest` ou `unittest`

Avec `unittest` :

- Intégré à Python
- Structuration en classes
- Rapide à mettre en place

Avec `pytest` :

- Plus moderne
- Syntaxe simplifiée
- Meilleur support des fixtures et tests paramétrés
-

```
pip install pytest
pytest test_moncode.py
```

10. En résumé

- Le **test** est une étape essentielle du développement logiciel
- Il existe plusieurs **niveaux** et **types** de tests
- Des outils comme `unittest`, `pytest`, ou même `assert` permettent de les mettre en œuvre efficacement
- **Déboguer** consiste à identifier et corriger les erreurs
- Tester, ce n'est pas prouver l'absence de bugs, mais augmenter la confiance dans la robustesse du code

Finalisation du projet

Objectifs pédagogiques :

- Apprendre à optimiser le code et les requêtes SQL pour améliorer les performances de l'application.
- Comprendre l'importance de la gestion des erreurs dans une application réelle.
- Savoir rédiger une documentation utilisateur claire et efficace.

Thèmes abordés :

1. **Bonnes pratiques de finalisation d'un projet logiciel :**
   - Nettoyage du code : suppression des redondances, mise en place de conventions de nommage claires.
   - Optimisation du code : éviter les calculs ou requêtes redondantes, simplifier les structures conditionnelles.
   - Gestion des erreurs : utilisation des blocs `try/except`, création de messages d'erreurs clairs pour l'utilisateur.
   - Tests fonctionnels et vérification des cas limites.
2. **Préparation de la documentation utilisateur :**
   - Qu'est-ce qu'une bonne documentation ?
   - Inclure une description générale, des instructions d'installation et d'utilisation, une FAQ, et les contacts pour le support.
   - Format possible : PDF, site web, ou manuel intégré dans l'application.

Exemple concret :

Optimisation d'une application de gestion de bibliothèque :

- Revue des requêtes SQL : remplacement de requêtes `SELECT *` par des requêtes ciblées.
- Gestion des exceptions : ajouter des `try/except` pour gérer les erreurs de connexion à la base de données ou les entrées invalides de l'utilisateur.

Conseil de présentation en classe :

Projetez un extrait de code avant/après optimisation et commentez en direct les différences. Ensuite, montrez un exemple de documentation utilisateur bien structurée (réel ou simulé).

Travaux Pratiques

Exercice 1 : Optimisation des requêtes SQL

- Reprendre les requêtes existantes dans le projet et les analyser.
- Identifier les requêtes inefficaces (ex. : jointures inutiles, absence d'index).
- Réécrire les requêtes en version optimisée.

Exercice 2 : Rédaction d'un guide utilisateur

- Créer un document (PDF ou Markdown) expliquant :
  - Le but de l'application
  - Comment l'installer
  - Comment l'utiliser (avec captures d'écran si possible)
  - Les problèmes connus ou limitations

Présentation du projet

Objectifs pédagogiques :

- Savoir structurer un discours technique en vue d'une présentation orale.
- Préparer un support visuel clair et percutant.
- Être capable de présenter son projet devant un public et de répondre aux questions.

Thèmes abordés :

1. **Préparation de la présentation orale :**
   - Structure classique : Introduction → Besoin/problématique → Fonctionnement → Démonstration → Conclusion.
   - Gérer le temps : ne pas dépasser 10 minutes, garder une marge pour les questions.
   - Bonnes pratiques : parler lentement, regarder son public, éviter de lire ses slides.
2. **Conception d'un support visuel :**
   - Utiliser un outil comme PowerPoint, Google Slides ou Canva.
   - Illustrer avec des captures d'écran de l'application, des schémas, des graphiques.
   - Éviter de surcharger les slides de texte.

Exemple concret :

Présentation type d'un projet de gestion de bibliothèque :

- Diapo 1 : Titre + noms des étudiants
- Diapo 2 : Problématique
- Diapo 3 : Fonctionnalités principales
- Diapo 4 : Architecture (base de données, interface)
- Diapo 5 : Démonstration
- Diapo 6 : Perspectives et conclusion

Travaux Pratiques

Exercice 1 : Préparation d'une présentation

- Chaque binôme prépare sa présentation PowerPoint ou équivalent.
- Crée une démonstration fonctionnelle de son application.
- Répartir les rôles entre les membres du groupe.

Exercice 2 : Simulation de présentation

- Simuler la présentation devant un petit groupe (autres étudiants).
- Recueillir des **feedbacks** : clarté, posture, support, rythme, démonstration.
- Répéter la présentation en tenant compte des retours.