Mini-projet-python-debugging-collaboratif- Groupe 01

📣 Enoncé de projet

🔄 Workflow Git

👩‍💻 Équipe

- **Sofiane** - Module 1: Moteur d'exécution
- **Ilyes** - Module 2: Debugging
- **Faiçal** - Module 3: Collaboration
- **Abderrahmane** - Module 4: Documentation & GitHub

📊 Progression du projet

Module	Progression	Dernière mise à jour
Moteur d'exécution	██████████ 100% ✅	28/01/2026
Module debugging	██████████ 100% ✅	07/02/2026
Collaboration	██████████ 100% ✅	07/02/2026
Documentation	██████████ 100% ✅	07/02/2026

- ✅ Moteur d'exécution sécurisé complet
- ✅ Gestion des exceptions et timeout
- ✅ Monitoring mémoire et temps d'exécution
- ✅ Historique et statistiques
- ✅ Tests unitaires (8 tests, 100% couverture)
- ✅ Documentation complète
- ✅ Exemples d'utilisation

🗂️ Structure du projet

📚 Documentation

- **Module 1 - Moteur d'Exécution**
- **Module 2 - Debugger**
- **Module 3 - Collaboration**
- **Démonstration du Projet Python - Debugging Collaboratif**

📝 Changelog

[Version 1.0.0] - 07/02/2026

Dernière mise à jour: 07/02/2026 par Abderrahmane

Mini-projet-python-debugging-collaboratif- Groupe 01

Enoncé de projet

📌 Contexte : Mini-projet collaboratif réalisé dans le cadre du Doctorat – Troisième Cycle, visant à approfondir les techniques de debugging Python et le travail collaboratif à l'aide des outils DevOps.

🎯 Objectifs pédagogiques :

- Maîtriser le debugging avancé en Python
- Identifier, analyser et corriger des erreurs complexes
- Travailler efficacement en équipe via Git/GitHub
- Utiliser des outils collaboratifs (cloud, Live Share)

🧩 Description du projet : Développer une plateforme permettant aux utilisateurs de soumettre du code Python, de détecter automatiquement les erreurs, de visualiser les logs et de collaborer à la correction du code.

⚙️ Fonctionnalités attendues :

- Exécution sécurisée de scripts Python
- Capture et analyse des exceptions
- Génération de logs détaillés
- Correction collaborative du code
- Historique des erreurs et corrections

🛠️ Technologies à utiliser : - Python - Git / GitHub - Google Colab - Visual Studio Live Share

👥 Répartition du travail (suggestion) :

- Membre 1 : moteur d'exécution
- Membre 2 : module de debugging
- Membre 3 : collaboration temps réel
- Membre 4 : documentation & gestion GitHub

📄 Livrables attendus :

- Dépôt GitHub structuré (Code source versionné)
- Wiki Documentation projet
- Rapport technique PDF
- Journal de commits

Workflow Git

Pour commencer à travailler

```
# 1. Créer une branche pour votre fonctionnalité
git checkout -b feature/nom-fonctionnalite

# 2. Faire vos modifications
# ... coder ...

# 3. Ajouter les fichiers modifiés
git add .

# 4. Commit avec un message descriptif
git commit -m "feat: ajout de la fonctionnalité X"

# 5. Pousser vers GitHub
git push origin feature/nom-fonctionnalite

# 6. Créer une Pull Request sur GitHub
```

Convention de nommage des commits

- `feat:` Nouvelle fonctionnalité
- `fix:` Correction de bug
- `docs:` Documentation
- `test:` Ajout/modification de tests
- `refactor:` Refactorisation du code
- `style:` Formatage, indentation

Tests et qualité du code

```
# Lancer tous les tests
pytest tests/ -v

# Tests avec couverture
pytest tests/ --cov=src --cov-report=term-missing

# Générer un rapport HTML de couverture
pytest tests/ --cov=src --cov-report=html
# Puis ouvrir: htmlcov/index.html

# Vérifier le style de code (PEP 8)
flake8 src/ tests/
```

Règles de base

1. **Créer une branche** pour chaque nouvelle fonctionnalité
2. **Écrire des tests** pour le nouveau code
3. **Documenter** les fonctions et modules
4. **Faire des commits atomiques** avec des messages clairs
5. **Créer une Pull Request** pour review

🗁 Structure du projet

```
mini-projet-python-debugging-collaboratif-g01/
│
├── src/                          # Code source
│   ├── __init__.py
│   ├── execution_engine.py       # ✅ Module 1 - Moteur d'exécution (Sofiane)
│   ├── debugger.py               # ⏳ Module 2 - Debugging (Membre 2)
│   ├── collaboration.py          # ⏳ Module 3 - Collaboration (Membre 3)
│   └── utils.py                  # Utilitaires communs
│
├── tests/                        # Tests unitaires
│   ├── __init__.py
│   ├── test_execution_engine.py  # ✅ Tests module 1
│   ├── test_debugger.py          # ⏳ Tests module 2
│   └── test_collaboration.py     # ⏳ Tests module 3
│
├── docs/                         # Documentation
│   ├── execution_engine_doc.md   # ✅ Doc module 1
│   ├── debugger_doc.md           # ⏳ Doc module 2
│   └── collaboration_doc.md      # ⏳ Doc module 3
│
├── examples/                     # Exemples d'utilisation
│   └── example_usage.py          # ✅ Démonstrations
│
├── logs/                         # Fichiers de logs
│   └── .gitkeep
│
├── .gitignore                    # Fichiers à ignorer
├── requirements.txt              # Dépendances Python
├── README.md                     # Ce fichier
└── CONTRIBUTING.md               # Guide de contribution
```

Étapes d'installation

```
# 1. Cloner le repository
git clone https://github.com/[superviseur]/mini-projet-python-debugging-
collaboratif-g01.git
cd mini-projet-python-debugging-collaboratif-g01

# 2. Créer un environnement virtuel (recommandé)
python -m venv venv

# 3. Activer l'environnement virtuel
# Sur Windows:
venv\Scripts\activate
# Sur Linux/Mac:
source venv/bin/activate
```

```
# 4. Installer les dépendances
pip install -r requirements.txt

# 5. Vérifier l'installation
python -c "from src.execution_engine import ExecutionEngine; print('✓
Installation réussie!')"
```

💻 Utilisation

Exemple rapide - Module 1 (Moteur d'exécution)

```
from src.execution_engine import ExecutionEngine

# Créer une instance
engine = ExecutionEngine(timeout=10, max_memory_mb=100)

# Exécuter du code
code = """
print("Hello World!")
x = 10 + 20
print(f"Résultat: {x}")
"""

result = engine.execute_code(code)

# Afficher le résultat
if result['success']:
    print("✓ Exécution réussie!")
    print(result['output'])
else:
    print("✗ Erreur:", result['error'])
```

Lancer les tests

```
# Tous les tests
pytest tests/ -v

# Tests avec couverture de code
pytest tests/ --cov=src --cov-report=html

# Tests d'un module spécifique
pytest tests/test_execution_engine.py -v
```

Lancer les exemples

```
# Démonstration du moteur d'exécution
python examples/example_usage.py

# Démonstration basique
python src/execution_engine.py
```

📄 `docs/collaboration_doc.md` (Module 3)

📚 Documentation - Collaboration Temps Réel

Auteur: Faiçal Hima

Module: `collaboration.py`

📌 Vue d'ensemble

Le module de collaboration gère l'aspect social et la traçabilité du projet. Il permet de suivre les membres actifs d'une session et d'historiser chaque correction apportée, créant ainsi un journal d'audit pour le travail d'équipe.

🎯 Fonctionnalités principales

1. **Gestion de Session**

- Enregistrement des collaborateurs par nom
- Calcul de la durée de la session collaborative
- Monitoring du nombre de participants actifs

2. **Suivi des Corrections (Audit Trail)**

- Journalisation de "Qui a corrigé Quoi"
- Horodatage automatique des interventions
- Description textuelle des solutions appliquées

3. **Reporting d'Équipe**

- Résumés statistiques des sessions
- Export de rapports d'activité formatés pour le Wiki/Rapport final

---

🔧 Utilisation

Exemple basique de gestion de session

```
from src.collaboration import CollaborationManager

# Créer une session
collab = CollaborationManager()

# Enregistrer des membres
collab.register_collaborator("Sofiane")
collab.register_collaborator("Ilyes")
```

```
# Enregistrer une action de correction
collab.log_correction(
    collaborator="Membre 2",
    error_type="SyntaxError",
    fix_description="Ajout des deux-points manquants ligne 5"
)

# Afficher le rapport d'activité
print(collab.format_collab_report())
```

📊 Structure du résultat

La méthode `get_session_summary()` retourne un dictionnaire avec les clés suivantes:

Clé	Type	Description
`duration`	str	Temps écoulé depuis le début de session
`total_collaborators`	int	Nombre de membres uniques enregistrés
`total_fixes`	int	Nombre total de corrections logguées
`fixes`	list	Liste détaillée des dictionnaires de correction

⚙️ Configuration

- **Stockage**: Les données sont maintenues en mémoire vive pour la session actuelle et persistées via les logs système dans `logs/debugger.log`.
- **Mode Collaborative**: Compatible avec l'utilisation de VS Code Live Share.

🧪 Tests

Exécuter les tests unitaires:

```
# Lancer les tests de collaboration
pytest tests/test_collaboration.py -v

# Vérifier la couverture
pytest tests/test_collaboration.py --cov=src.collaboration
```

🔄 Intégration avec les autres modules

Module de Debugging (Module 2)

Le `CollaborationManager` utilise les `error_type` identifiés par le Debugger pour documenter les corrections effectuées par les membres de l'équipe.

Changelog

- Première version du gestionnaire de collaboration
- [x] Système d'enregistrement des membres
- [x] Journal d'audit des corrections
- [x] Générateur de rapports de session

📄 `docs/debugger_doc.md` (Module 2)

📚 Documentation - Moteur de Debugging Avancé

Auteur: Tarek Ilies Embarki

Module: `debugger.py`

📌 Vue d'ensemble

Le moteur de debugging avancé intervient après l'exécution d'un script pour intercepter les exceptions. Il analyse les tracebacks bruts, extrait les informations critiques (type d'erreur, ligne) et fournit des suggestions de correction basées sur une base de connaissances intégrée.

🎯 Fonctionnalités principales

1. **Analyse d'Exceptions**

- Extraction du type d'erreur (SyntaxError, NameError, etc.)
- Identification précise de la ligne fautive via Regex
- Nettoyage des messages d'erreur système pour l'utilisateur

2. **Système de Suggestions**

- Base de connaissances (Knowledge Base) pour les erreurs courantes
- Conseils pédagogiques pour la résolution
- Gestion des erreurs inconnues avec lien vers la documentation officielle

3. **Classification et Logging**

- Évaluation de la sévérité (High/Medium)
- Journalisation persistante via `loguru` dans `logs/debugger.log`
- Historisation des erreurs pour analyse collaborative

🔧 Utilisation

Installation des dépendances

```
pip install -r requirements.txt
```

Exemple basique

```

```

```python
from src.debugger import Debugger

# Initialiser le debugger
debugger = Debugger()

# Simuler un résultat d'exécution erroné
result_errone = {
    'success': False,
    'error': 'ZeroDivisionError: division by zero (line 4)',
    'output': ''
}

# Analyser l'erreur
analysis = debugger.analyze(result_errone)

# Afficher le rapport formaté
print(debugger.format_report(analysis))
```

📊 Structure du résultat

La méthode `analyze()` retourne un dictionnaire avec les clés suivantes:

Clé	Type	Description
`status`	str	"SUCCESS" ou "FAILED"
`error_type`	str	Classe de l'exception (ex: NameError)
`line_number`	int/str	Ligne détectée ou "Unknown"
`message`	str	Message d'erreur détaillé
`suggestion`	str	Conseil de correction proposé
`severity`	str	Niveau de criticité (High/Medium)

🛡️ Sécurité & Fiabilité

1. **Regex Robustes**: Extraction sécurisée des numéros de ligne même sur des formats de traceback variés.
2. **Fallback**: En cas d'erreur non reconnue, le système bascule sur une suggestion générique sans faire planter l'application.
3. **Isolation des Logs**: Les fichiers de logs sont limités en taille (rotation) pour éviter la saturation disque.

🧪 Tests

Exécuter les tests unitaires:

```
# Tous les tests du module 2
pytest tests/test_debugger.py -v

# Avec couverture de code
pytest tests/test_debugger.py --cov=src.debugger
```

🔄 Intégration avec les autres modules

Module d'Exécution (Module 1)

Le debugger reçoit directement le dictionnaire de sortie de `ExecutionEngine`.

📝 Changelog

- ✨ Implémentation de l'analyseur Regex
- ✅ Base de connaissances initiale (7 types d'erreurs)
- ✅ Intégration de `loguru` pour la traçabilité
- ✅ Tests unitaires avec 100% de couverture

📊 Démonstration Interactive du Projet

Visualiser le Notebook

Notre démonstration complète est disponible sous forme de Jupyter Notebook :

👉 **Ouvrir la démonstration interactive**

Contenu du Notebook

1. Module d'Exécution

- Initialisation du moteur
- Exécution de code sécurisé
- Capture des erreurs
- Statistiques d'exécution

2. Module de Debugging

- Analyse des erreurs
- Suggestions de correction
- Rapports formatés

3. Module de Collaboration

- Gestion des sessions
- Journal des corrections
- Rapports d'équipe

Exécution locale

Pour exécuter le notebook localement :

```
# Cloner le repository
git clone https://github.com/VOTRE_USERNAME/mini-projet-python-debugging-
collaboratif-g01.git

# Installer les dépendances
pip install jupyter psutil loguru

# Lancer Jupyter
jupyter notebook project_demo.ipynb
```

📚 Documentation - Moteur d'Exécution Sécurisé

Auteur: Sofiane

Module: `execution_engine.py`

📌 Vue d'ensemble

Le moteur d'exécution sécurisé permet d'exécuter du code Python fourni par les utilisateurs de manière isolée et contrôlée, avec capture des erreurs, timeout et monitoring des ressources.

🎯 Fonctionnalités principales

1. **Exécution sécurisée**

- Isolation du code utilisateur
- Protection contre les boucles infinies (timeout)
- Limitation de la consommation mémoire
- Capture des outputs (stdout/stderr)

2. **Gestion des erreurs**

- Détection des erreurs de syntaxe
- Capture des exceptions runtime
- Stack trace détaillée
- Classification des types d'erreurs

3. **Monitoring**

- Temps d'exécution précis
- Consommation mémoire
- Historique des exécutions
- Statistiques globales

🔧 Utilisation

Installation des dépendances

```
pip install -r requirements.txt
```

Exemple basique

```

```

```
from src.execution_engine import ExecutionEngine

# Créer une instance du moteur
engine = ExecutionEngine(timeout=10, max_memory_mb=100)

# Exécuter du code
code = """
print("Hello World")
x = 10 + 20
print(f"Résultat: {x}")
"""

result = engine.execute_code(code)

# Vérifier le résultat
if result['success']:
    print("✓ Exécution réussie!")
    print(f"Output: {result['output']}")
    print(f"Temps: {result['execution_time']:.4f}s")
else:
    print("✗ Erreur détectée!")
    print(f"Erreur: {result['error']}")
    print(f"Traceback: {result['traceback']}")
```

Exemple avec validation préalable

```
# Valider le code avant exécution
code = "print('test')"
is_valid, message = engine.validate_code(code)

if is_valid:
    result = engine.execute_code(code)
else:
    print(f"Code invalide: {message}")
```

Consulter l'historique

```
# Récupérer les 5 dernières exécutions
history = engine.get_history(limit=5)

for i, entry in enumerate(history, 1):
    print(f"\n--- Exécution #{i} ---")
    print(f"Code: {entry['code']}")
    print(f"Succès: {entry['result']['success']}")
```

Obtenir des statistiques

```
stats = engine.get_stats()

print(f"Total exécutions: {stats['total_executions']}")
print(f"Taux de succès: {stats['success_rate']:.2f}%")
print(f"Temps moyen: {stats['avg_execution_time']:.4f}s")
print(f"Mémoire moyenne: {stats['avg_memory_used']:.2f}MB")
```

📊 Structure du résultat

La méthode `execute_code()` retourne un dictionnaire avec les clés suivantes:

Clé	Type	Description
`success`	bool	True si l'exécution a réussi
`output`	str	Sortie standard du programme
`error`	str	Message d'erreur (si échec)
`execution_time`	float	Temps d'exécution en secondes
`memory_used`	float	Mémoire utilisée en MB
`traceback`	str	Stack trace complète
`timestamp`	str	Date et heure de l'exécution

⚙️ Configuration

Paramètres du constructeur

```
ExecutionEngine(timeout=10, max_memory_mb=100)
```

- **timeout** (int): Temps maximum d'exécution en secondes (défaut: 10s)
- **max_memory_mb** (int): Mémoire maximale autorisée en MB (défaut: 100MB)

🛡️ Sécurité

Mesures de protection

1. **Timeout**: Arrêt automatique après le délai défini
2. **Limite mémoire**: Protection contre la surconsommation
3. **Isolation**: Environnement d'exécution séparé
4. **Pas d'accès fichiers**: Le code ne peut pas lire/écrire de fichiers (par défaut)

Limitations connues

⚠ **Attention**: Ce moteur ne protège pas contre:

- Les opérations réseau non contrôlées
- L'import de modules système dangereux
- Les attaques par déni de service sophistiquées

Pour une utilisation en production, considérer l'ajout de:

- `RestrictedPython` pour limiter les imports
- Conteneurisation (Docker) pour isolation complète
- Rate limiting au niveau applicatif

Tests

Exécuter les tests unitaires:

```
# Tous les tests
pytest tests/test_execution_engine.py -v

# Avec couverture de code
pytest tests/test_execution_engine.py --cov=src/execution_engine
```

Intégration avec les autres modules

Module de debugging (Membre 2)

```
# Le moteur peut passer ses résultats au debugger
from src.debugger import Debugger

result = engine.execute_code(code)
if not result['success']:
    debugger = Debugger()
    analysis = debugger.analyze_error(result)
```

Module de collaboration (Membre 3)

```
# Partager les résultats d'exécution
from src.collaboration import ShareSession

result = engine.execute_code(code)
session = ShareSession()
session.broadcast_execution_result(result)
```

☑ Évolutions futures

- ☐ Support des entrées utilisateur multiples
- ☐ Sauvegarde de l'historique en base de données
- ☐ Export des logs au format JSON
- ☐ Interface web pour visualisation
- ☐ Support des notebooks Jupyter
- ☐ Sandboxing renforcé avec Docker

🤝 Contribution

Pour contribuer à ce module:

1. Créer une branche: `git checkout -b feature/nom-feature`
2. Commiter les changements: `git commit -m "Description"`
3. Pousser: `git push origin feature/nom-feature`
4. Créer une Pull Request

📝 Changelog

Version 1.0.0 (28/01/2026)

- ✨ Première version du moteur d'exécution
- ✅ Exécution sécurisée avec timeout
- ✅ Capture des exceptions
- ✅ Monitoring mémoire et temps
- ✅ Historique et statistiques
- ✅ Tests unitaires complets
