Industrialisation du Logiciel
TP1: Tests

Lorsque vous travaillez sur un projet, vous désirez savoir si le code que vous produisez est de bonne qualité. Même si tester manuellement son code n'est pas une mauvaise idée, il est très difficile de le faire exhaustivement et rapidement, d'autant plus pour les gros projets.

Dans ce TP, nous allons automatiser certains tests avec des exemples pratiques:

- La création de tests unitaires et d'intégration avec pytest
- La création d'un test fonctionnel avec Selenium.
- La création d'un test de non-régression
- La création de tests de performance avec cProfile et Locust
- L'utilisation de la fonction de coverage

Pour ces tests, vous trouverez 3 modules dans le dossier `src` contenant des fonctionnalités qu'il faudra tester. Notez que par défaut, tous les tests échouent. Il est donc normal d'avoir un nombre important d'erreurs lorsque vous lancerez vos tests au début. L'objectif de ce TP est de réussir tous les tests.

Structure du dossier TP1

```
├── instruction.pdf
├── src                          <- Application à tester
│   │
│   └── __init__.py
│   └── matrix_calculation.py    <- Module calcul matriciel
│   └── string_manipulation.py   <- Module manipulation de string
│   └── wallet.py                <- Module gestion d'un porte-monnaie
│
├── tests
│   └── fixtures
│   │   └── __init__.py
│   │   └── numpy_fixtures.py
│   │   └── wallet_fixtures.py
│   │
│   └── unit                                <- Tests unitaires
│   │   └── __init__.py
│   │   └── test_matrix_calculation.py      <- Pour le module matrix_calulcation.py
│   │   └── test_string_manipulation.py     <- Pour le module string_manipulation.py
│   │   └── test_wallet.py                  <- Pour le module wallet.py
│   │
│   └── regression
│   │   └── __init__.py
│   │   └── test_regression.py              <- Tests de non-regression
│   │
│   └── functional
│   │   └── __init__.py
│   │   └── test_selenium.py                <- Tests fonctionnels avec Selenium
│   │
│   └── integration
│   │   └── __init__.py
│   │   └── test_integration_wallet.py      <- Tests d'intégrations wallet.py
│   │
│   └── performance
│       └── __init__.py
│       └── performance.py                  <- Tests de performance avec cProfile
│       └── locustfile.py                   <- Tests de performance avec Locust
│
├── .coveragerc                             <- Fichier de configuration du coverage
├── .gitignore
├── .pre-commit-config.yaml                 <- Fichier de configuration de pre-commit
└── requirements.txt
```

Exercice 1: Pre-commit

Vous remarquerez la présence du package Python *pre-commit* dans le fichier `requirements.txt`. Ce package permet, grâce au fichier de configuration `.pre-commit-config.yaml`, d'appliquer une série de règles sur vos fichiers. La configuration actuelle est assez simple, elle contient 4 règles (hooks) :

- check-yaml -> Vérifie la syntaxe des fichiers .yaml.
- end-of-line-fixer -> S'assure que chaque fichier se termine par une nouvelle ligne.
- trailing-whitespace -> Supprimer les espaces inutiles, en fin de ligne par exemple.
- black -> Formatte le code suivant le formatter Python *Black*.

Beaucoup d'autres hooks existent, n'hésitez pas à consulter la documentation officielle de pre-commit.

Après avoir installé votre environnement virtuel ainsi que les packages Python, initialisez pre-commit avec la commande suivante:

```
pre-commit install
```

À présent, chaque nouveau commit sera accompagné d'une vérification par *pre-commit*. Si ce dernier détecte des erreurs, il le signalera dans la console et corrigera les fichiers concernés. Il faudra ensuite les réajouter avec la commande `git add` puis recréer un commit.

Exercice 2: Tests basiques

a)

Vous trouverez les fichiers `test_matrix_calculation.py`, `test_string_manipulation.py` et `test_wallet.py` dans le dossier `tests/unit`. Pour chacun de ces fichiers, vous devez créer des tests unitaires pertinents en vous servant de la librairie **pytest**. Les prototypes des méthodes de tests sont déjà écrits. À vous de les implémenter en prenant en compte les noms des méthodes.

Pour réaliser vos tests, exécutez la commande suivante à la racine du projet:

```
pytest
```

b)

Les tests d'intégration servent à savoir comment plusieurs fonctionnalités ou méthodes interagissent. Un scénario de transfert d'argent et de paiement différé vous est proposé dans le fichier `test_integration_wallet.py`. Vous devez implémenter ce scénario dans la méthode `test_deferred_payment`, et vous assurer que le test réussit.

Exercice 3: Tests de non-régression

Les tests de non-régression permettent de s'assurer qu'une nouvelle fonctionnalité ne provoque pas un dysfonctionnement de ce qui fonctionnait avant, ou que la correction d'un bug résoud effectivement le problème en retestant le programme après modification. Dans le dossier `tests/regression`, vous trouverez le fichier `test_regression.py`.

a)

Vous devez implémenter le test `test_regression_student` qui s'assure que la méthode `get_student` retourne toujours le même objet. Pour ce faire, utilisez la fixture *data_regression* de la librairie **pytest-regressions**. Lancez ensuite les tests, une erreur devrait apparaître.

```
FAILED tests/regression/test_regression.py::test_regression_student
 - Failed: File not found in data directory, created:
```

Relancez la commande, cette fois-ci le test devrait passer.

Comment expliquez-vous cette différence de comportement ? Est-ce que quelque chose a changé entre les 2 exécutions ?

Concrètement, quel test est opéré par *data_regression*?

b)

Vous allez maintenant simuler un changement. Modifiez la fonction `get_student` et rajoutez une nouvelle note dans le tableau *grades*. Relancez ensuite les tests.

Pourquoi le test ne passe-t-il pas?

Quelle commande permet de dire à pytest que la nouvelle implémentation est en faite correcte ?

Exercice 4: Tests fonctionnels

Les tests fonctionnels servent à tester le comportement de votre application au moment de son utilisation. La particularité de ces tests est qu'il n'est pas forcément nécessaire de connaître le code pour les implémenter. Un outil très utile pour créer ce genre de tests est Selenium. Il permet de simuler un navigateur web et de le manipuler avec une API.

a)

Afin de tester les fonctionnalités que propose Selenium, vous allez implémenter la méthode `test_get_teachers` qui se trouve dans le fichier `test_selenium_example.py`. Certaines étapes ont déjà été faites, il vous suffit de les compléter en suivant les étapes inscrites en commentaire du fichier.

Utilisez la documentation de Selenium pour les détails de son utilisation.

Pour voir le résultat, lancez simplement la commande de tests. Notez l'utilisation de l'option -s qui permet d'afficher les `print()` de vos tests dans la console.

```
pytest -s
```

Attention: Il peut être difficile de sélectionner les bons noeuds d'une page web. N'hésitez pas à consulter des ressources externes et à tester vos sélecteurs, notamment de type XPATH, dans la console de votre navigateur web (F12).

Exercice 5: Tests de performance

a)

La librairie **cProfile** permet de mesurer la performance d'un programme. En l'associant à la librairie **pstats**, on peut notamment générer des données très utiles pour comprendre quelle partie du code prend le plus de temps à s'exécuter.

Ouvrez le fichier `performance.py` et implémentez la méthode `main()`. Cette dernière doit:

- Appeler les 2 fonctions `array_filling_1` et `array_filling_2` fournies dans le fichier.
- Utiliser cProfile pour profiler l'exécution des fonctions.
- Utiliser pstats pour réarranger la sortie du profiler et trier les données suivant le nombre d'appels (colonne `ncalls`).
- Afficher le résultat sur la console (voir méthode `print_stats`)

Que constatez-vous ? Y a-t-il une différence entre les temps d'exécutions des 2 fonctions ? Si oui, laquelle est la plus rapide, et pourquoi?

b)

cProfile est très utile pour tester la rapidité d'une application locale. Mais comment faire pour une application distante, comme un site web par exemple ? C'est ici qu'intervient **Locust**.

Cette librairie permet entre autres de tester les temps de réponse des routes d'un site web donné, et d'en générer des indicateurs et graphes pertinents. Pour faire des stress tests, c'est l'outil idéal.

Vous allez tester le site web des CFF avec Locust. Ouvrez le fichier `locustfile.py` et implémentez 3 nouveaux tests en dessous de celui déjà présent. On veut que chaque test s'exécute 2 fois plus que le précédent. Une fois fait, lancez le test en utilisant la commande suivante:

```
locust -f .\tests\performance\locustfile.py
```

Locust se lance par défaut sur http://localhost:8089.

Lancez le test sur l'URL `https://www.sbb.ch` avec 4 utilisateurs et un spawn rate de 1. Lancez le test pendant quelques secondes, observez les résultats puis répondez aux questions ci-dessous.

Important: Envoyer trop de requêtes en un temps trop court pourrait pousser le site web à vous blacklister, ce n'est pas le but ici. Pensez à bien respecter les paramètres spécifiés ci-dessus et ne laissez tourner vos tests Locust que 10 secondes au maximum.

Combien de routes différentes sont testées?

Que sont les RPS et pourquoi sont-elles différentes pour chaque route?

Quelle route possède la meilleure performance?

Exercice 6: Coverage

Lorsque vous testez un code, il peut être difficile de savoir quel pourcentage de votre code est effectivement testé. Heureusement, la fonction de coverage de pytest permet de s'en donner une bonne idée.

a)

Lancez l'analyse du coverage avec la commande suivante:

```
pytest --cov=.
```

Cette commande va demander à pytest de lancer tous les tests et de noter quelle partie du code a été testée ou non. A la fin, pytest affiche un tableau qui résume la couverture (coverage en anglais), c'est à dire le pourcentage du code qui a été testé. Notez bien que pytest possède ses propres définitions d'une ligne de code testée ou non testée, donc le coverage n'est qu'un indicateur. Dans un projet réel, il n'est pas nécessaire d'atteindre 100% de coverage, et à l'inverse 100% ne vous garantit pas que votre code a été entièrement et correctement testé.

Que représentent les pourcentages affichés?

Comment pytest sait-il quels fichiers analyser et quels fichiers ne pas analyser?

b)

Le coverage semble être incomplet. Cela indique que pytest a detecté des cas que vous n'avez pas testé. Regardez quels fichiers sont concernés et ajoutez des tests unitaires en plus pour faire monter le pourcentage total à 100%.

Indice: certaines fonctionnalités peuvent nécessiter plusieurs tests. pytest possède une commande pour vous aider à trouver quelles lignes ne sont pas bien testées.