Test Driven Development (TDD)

Les outils de test

Semifir

Un langage = un outil

- Les outils utilisés dépendent du langage utilisé
- Pour un même langage, il peut exister plusieurs frameworks/outils de tests
- Ici, nous nous concentrerons sur les outils Python.

Python : 2 outils phares

- Pytest
- Unittest

Pytest

Semifir

Présentation

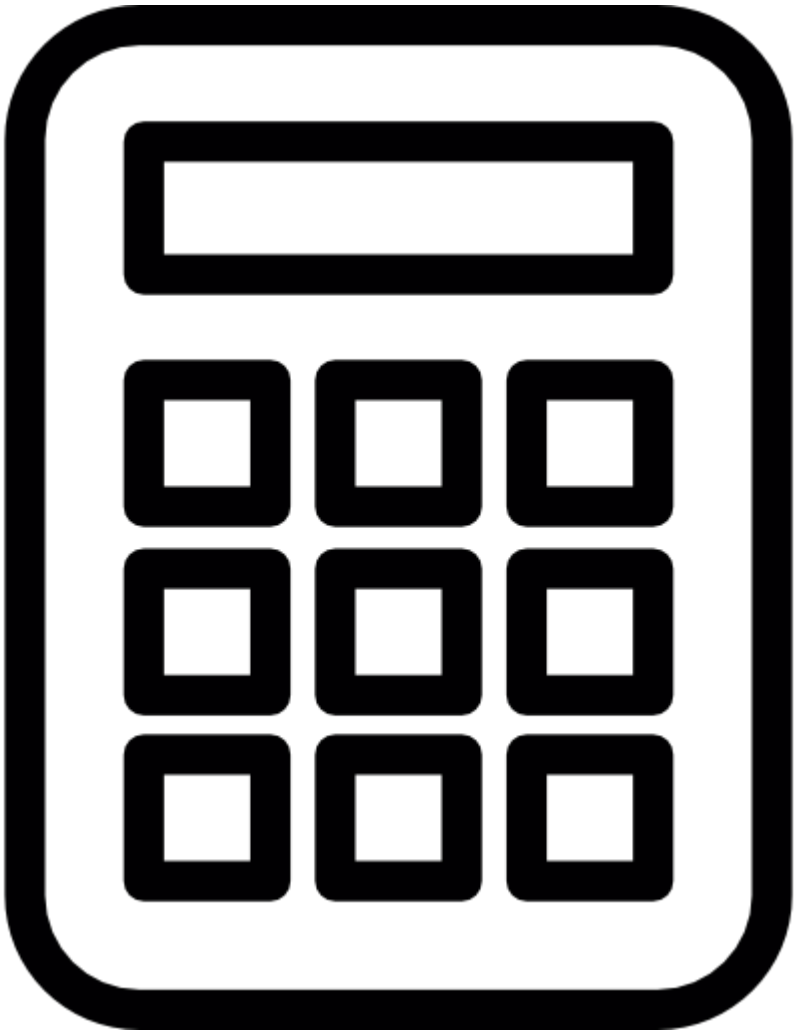
- Framework de tests pour Python.
- Permet de créer et d'exécuter des tests unitaires de manière simple et efficace.

Avantages

- Facile à installer et à utiliser.
- Compatible avec tous les interpréteurs Python (2.7, 3.5, PyPy).
- S'intègre facilement avec d'autres outils tels que Jenkins, Travis CI ou encore Gitlab CI.

- Capacité à exécuter des tests à partir de n'importe quel fichier Python.
- Fournit de nombreux outils pour la gestion des erreurs et des exceptions (la création de rapports détaillés sur les résultats de tests et la gestion de la couverture de code.)

Démonstration



Testons une calculatrice avec Pytest !

Note: ./Demo/00_Pytest_Calculette.md

Exemple (1/2)

```
# import de la classe Calculatrice depuis le module Calculatrice
from src.Calculatrice import Calculatrice

# définition de la fonction test_additionner_deux_nombres()
def test_additionner_deux_nombres():
    a = 1   # on définit la valeur de a
    b = 2   # on définit la valeur de b
    calculatrice = Calculatrice()   # on crée une instance de la classe Calculatrice

    resultat = calculatrice.additionner(a, b)   # on appelle la méthode additionner() de la
calculatrice avec les valeurs de a et b

    assert resultat == 3, "L'addition de 1 et 2 doit être égal à 3"   # on vérifie si le
résultat est égal à 3

# définition de la fonction test_soustraire_deux_nombres()
def test_soustraire_deux_nombres():
    a = 1   # on définit la valeur de a
```

```
    b = 2   # on définit la valeur de b
    calculatrice = Calculatrice()   # on crée une instance de la classe Calculatrice

    resultat = calculatrice.soustraire(a, b)   # on appelle la méthode soustraire() de la
calculatrice avec les valeurs de a et b

    assert resultat == -1, "La soustraction de 1 et 2 doit être égal à -1"   # on vérifie si le
résultat est égal à -1

# définition de la fonction test_multiplier_deux_nombres()
def test_multiplier_deux_nombres():
    a = 1   # on définit la valeur de a
    b = 2   # on définit la valeur de b
    calculatrice = Calculatrice()   # on crée une instance de la classe Calculatrice

    resultat = calculatrice.multiplier(a, b)   # on appelle la méthode multiplier() de la
calculatrice avec les valeurs de a et b

    assert resultat == 2, "La multiplication de 1 et 2 doit être égal à 2"   # on vérifie si le
résultat est égal à 2

# définition de la fonction test_diviser_deux_nombres()
def test_diviser_deux_nombres():
    a = 1   # on définit la valeur de a
    b = 2   # on définit la valeur de b
    calculatrice = Calculatrice()   # on crée une instance de la classe Calculatrice

    resultat = calculatrice.diviser(a, b)   # on appelle la méthode diviser() de la calculatrice
avec les valeurs de a et b

    assert resultat == 0.5, "La division de 1 et 2 doit être égal à 0,5"   # on vérifie si le
résultat est égal à 0.5
```

Exemple (2/2)

```
class Calculatrice:
    def additionner(self, a: float, b: float) -> float:
        """
        Additionne deux nombres

        :param a: float
        :param b: float
        :return: float
        """
        return a + b

    def soustraire(self, a: float, b: float) -> float:
        """
        Soustrait deux nombres
```

```
        :param a: float
        :param b: float
        :return: float
        """
        return a - b

    def multiplier(self, a: float, b: float) -> float:
        """
        Multiplie deux nombres

        :param a: float
        :param b: float
        :return: float
        """
        return a * b

    def diviser(self, a: float, b: float) -> float:
        """
        Divise deux nombres

        :param a: float
        :param b: float
        :return: float
        """
        return a / b
```

Questions

Si nous reprenons notre exemple de calculatrice :

- QUID de la division par 0 ?
- QUID des erreurs ?

Division par 0.

- Aucune US n'est définie pour gérer ce cas
- Si nous prenons la décision de gérer ce cas, nous risquons de "travailler pour rien"
- En effet, si l'US n'est pas définie, nous ne pouvons pas la tester

> Rien ne nous dit que ce comportement doit être refusé ou autorisé !

Les bugs

- L'objectif des tests est de vérifier que l'application fonctionne comme attendue.
- Ce n'est pas de vérifier que l'application ne plante pas.

- Si un comportement inattendu est détecté, il faut le corriger : alors nous créerons un test, basé sur une US.

> Les bugs seront remontés en temps voulu !

Pourquoi ne pas tester les bugs ?

- Rien ne dit que le comportement est inattendu
- Essayer d'anticiper les bugs est une perte de temps : on peut se tromper
- Il n'est pas utile de vouloir à tout prix tester 100% des cas possible

---

La notion de coverage

- Un coverage est une mesure de la couverture des tests
- Globalement, il nous indique quel pourcentage de notre code est testé

Question

> Quel coverage devons-nous avoir ?

Réponse

- On considère qu'un coverage de 80% est acceptable

Pourquoi 80% ?

- 80% est un chiffre arbitraire
- C'est le juste milieu entre le temps passé à écrire les tests et leur utilité

> Plus on écrira de test, moins le coverage augmentera

Mais encore...

- Un souci d'efficacité et de logique
- Si atteindre des 80% de coverage arrivera vite, les dépasser prendra de plus en plus de temps
- Plus on avancera, plus ce sera spécifique, voire alambiqué

> Passer 4 h à augmenter le coverage de 5%, pour un problème qui arriverait dans 2% des cas n'est pas pertinent

---

Et en TDD ?

- En TDD sur le principe, le coverage est de 100%
- On parle de 80% de coverage au niveau de l'application

Les erreurs

- S'il n'est pas utile d'anticiper les bugs, rien n'interdit de tester les erreurs
- Pytest met à notre disposition des moyens de tester les exceptions

> Valable uniquement si nous avons une US définie pour gérer ce cas !

Le test

```
# /tests/test_Calculatrice.py
import pytest
# [...]
def test_diviser_par_zero():
    calculatrice = Calculatrice()

    # Given
    a = 1
    b = 0

    # When
    with pytest.raises(ValueError) as exception:
            calculatrice.diviser(a, b)

    # Then
    # En testant la classe :
    assert isinstance(exception.value, ValueError)

    # En testant le message :
    assert str(exception.value) == "Le dénominateur ne peut pas être égal à 0"


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

La classe

```
# /src/Calculatrice.py
class Calculatrice {
# [...]
def diviser(self, a: float, b: float) -> float:
        if b == 0:
            raise ValueError("Le dénominateur ne peut pas être égal à 0")
```

```
        else:
            """
            Divise deux nombres

            :param a: float
            :param b: float
            :return: float
            """
        return a / b
}
```

Execution des tests

- Il faut créer un fichier qui s'appelle `__init__.py` dans chaque dossier.
- Il a pour but d'être exécuté lorsque le répertoire est importé comme un module.

> Il est important de noter que le fichier `__init__.py` peut être vide, mais il doit être présent dans un répertoire pour que celui-ci soit considéré comme un package.python.

Execution des tests

```
cd tests
pytest .\test_nom_du_test
```

Unittest

Semifir

unittest

Présentation

- Framework de test fourni par défaut avec Python.
- Similaire à Pytest.

Avantages

- Facilité d'utilisation et syntaxe simple.
- Intégration facile avec d'autres outils de développement.
- Approche modulaire pour organiser les tests.
- Découverte automatique des tests grâce à une convention de nommage.

- Assertions prédéfinies pour vérifier facilement les résultats attendus.
- Rapports détaillés pour faciliter le débogage.
- Intégration avec des outils de couverture de code pour mesurer la couverture des tests.

Exemple (1/2)

- Créez un un projet appelé `test_unittest`
- Créez deux dossiers à la racine du projet : `src` et `tests`

Comme pour pytest il faut installer unittest

```
pip install unittest
```

- Dans le dossier `tests`, créez un fichier `test_calculatrice.py` et copiez-y le code suivant :

```
# Importation du module unittest
import unittest
# Importation de la classe Calculatrice du fichier Calculatrice.py
from src.Calculatrice import Calculatrice

# Définition de la classe de test qui hérite de unittest.TestCase
class TestCalculatrice(unittest.TestCase):
    # Définition de la méthode de test appelée test_somme
    def test_somme(self):
        a = 4
        b = (2+2)
```

```
        # Appel de la méthode valeurEgale de l'instance de la classe Calculatrice
        resultat = Calculatrice().valeurEgale(a, b)

        # Vérification que le résultat est bien égal à True
        self.assertEqual(resultat, True )

# Vérification si ce script est exécuté directement (pas en tant que module)
if __name__ == '__main__':
# Exécution des tests unitaires avec unittest.main()
    unittest.main()
```

Exemple (2/2)

- Dans le dossier `src`, créez un fichier `calculatrice.py` et copiez-y le code suivant :

```
class Calculatrice:
    def additionner(self, a: float, b: float) -> float:
        """
        Additionne deux nombres

        :param a: float
        :param b: float
        :return: float
        """
        return a + b

    def soustraire(self, a: float, b: float) -> float:
        """
        Soustrait deux nombres

        :param a: float
        :param b: float
        :return: float
        """
        return a - b

    def multiplier(self, a: float, b: float) -> float:
        """
        Multiplie deux nombres

        :param a: float
        :param b: float
        :return: float
        """
        return a * b

    def diviser(self, a: float, b: float) -> float:
        if b == 0:
            raise ValueError("Le dénominateur ne peut pas être égal à 0")
        else:
```

```
            """
            Divise deux nombres

            :param a: float
            :param b: float
            :return: float
            """
        return a / b

    def valeurEgale(self, a: float, b: float) -> bool:
        if a == b:
            return a == b
            """
            Compare deux nombres

            :param a: float
            :param b: float
            :return: bool
            """
        else :
            raise ValueError("Les deux nombres ne sont pas égaux")
```

Execution des tests

```
# A la racine du projet
py -m unittest discover
```

- En utilisant `discover`. Nous pouvons exécuter automatiquement les tests.

Création d'un mock avec Pytest

Etape 1

- Installation : Assurez-vous d'avoir `pytest-mock` installé.
- Vous pouvez l'installer en utilisant `pip` :

```
pip install pytest-mock
```

Etape 2

- Importation : Importez `pytest` et `pytest_mock` dans votre fichier de test :

```
from pytest_mock import mocker
```

Etape 3

- Création du mock : Utilisez l'objet `mocker` pour créer votre mock. Vous pouvez utiliser la méthode `mocker.Mock()` pour créer un mock simple :

```
def test_exemple(mocker):
    # Création du mock
    my_mock = mocker.Mock()
```

Etape 4

- Configuration du comportement du mock.
- Vous pouvez définir le comportement attendu du mock en utilisant les attributs et les méthodes du mock.
- Exemple : Vous pouvez spécifier la valeur de retour d'une méthode du mock

```
def test_exemple(mocker):
    # Création du mock avec une valeur de retour
    my_mock = mocker.Mock()
    my_mock.some_method.return_value = 42
```

- `some_method.return_value` est une méthode utilisée pour définir une valeur de retour d'un objet mock lorsqu'il est appelé.
- Nous avons notre objet mock `some_method` et à son appel il renvoie la valeur 42.

Etape 5

- Utilisation du mock dans le test : Utilisez le mock dans votre test pour vérifier son comportement.

```
def test_exemple(mocker):
    my_mock = mocker.Mock()
    my_mock.some_method.return_value = 42

    # Utilisation du mock
    resultat = my_mock.some_method()

    # Assertion sur le résultat
    assert resultat == 42
    my_mock.some_method.assert_called_once()
```

- `assert_called_once()` est une méthode utilisée pour vérifier si un objet mock nommé `some_method` est appelé **exactement une fois**.
- Si l'appel de l'objet mock se produit plusieurs fois ou pas du tout, une assertion sera déclenchée et le test échouera.

Paramétrage d'un mock

- Lorsque vous créez un mock avec Pytest et `pytest-mock`, vous pouvez le paramétrer en définissant le comportement attendu de ses attributs et méthodes.

- Voici quelques exemple de paramétrage courant d'un mock :

```
def test_exemple(mocker):
    my_mock = mocker.Mock()
    my_mock.some_method.return_value = 42
    # ...
```

- Lever une exception lors de l'appel d'une méthode :

```
def test_exemple(mocker):
    my_mock = mocker.Mock()
    my_mock.some_method.side_effect = ValueError("Quelque chose n'a pas fonctionné")
    # ...
```

- `some_method.side_effect` est une méthode utilisée pour définir un effet secondaire lors d'un appel d'un objet mock.
- Au lieu de renvoyer une valeur de retour, l'objet mock lèvera une exception.
- Dans cet exemple, l'objet mock `some_method` lèvera une exception de type `ValueError` avec le message `Quelque chose n'a pas fonctionné` lors de son appel.

- Vous pouvez également écrire `some_method.Exception('error')`.
- Cela déclenchera une exception de générique. Tandis que `ValueError` déclenchera une exception spécifique.

- Vérifier les arguments passés à une méthode :

```
def test_exemple(mocker):
    my_mock = mocker.Mock()
    # ...
```

```
    my_mock.some_method.assert_called_with(arg1, arg2)
```

- `some_method.assert_called_with(arg1, arg2)` est une méthode utilisée pour vérifier si un objet mock nommé `some_method` est appelé avec les arguments `arg1` et `arg2`.
- Si l'objet mock a été appelé avec d'autres arguments ou pas du tout une assertion sera déclenchée et le test échouera.

- Définir un comportement différent pour chaque appel à une méthode :

```
def test_exemple(mocker):
    my_mock = mocker.Mock()
    my_mock.some_method.side_effect = [1, 2, 3]
    # ...
```

Dans cet exemple, chaque appel à la méthode `some_method` du mock renverra successivement les valeurs 1,2 et 3.

- Compter le nombre d'appels à une méthode :

```
def test_exemple(mocker):
    my_mock = mocker.Mock()
    # ...
    assert my_mock.some_method.call_count == 3
```

- `some_method.call_count` est une méthode qui renvoie le nombre de fois où un objet mock a été appelé.
- Dans cet exemple, la méthode a été utilisé pour obtenir le nombre d'appels effectués sur l'objet mock et l'utiliser pour assertions pour vérifier le comportement attendu.

> Ces exemples illustrent quelques façons de paramétrer un mock avec `Pytest` et `pytest-mock`.

Démonstration !

Gestion d'un carnet d'adresses

Note: 01_Pytest_Mock.md

À vous de jouer !

Un peu de pratique de tests unitaires

- Réalisez les exercices demandés repository qui vous a été fourni.
- Vous réaliserez :
    - Les classes
    - Les tests unitaires

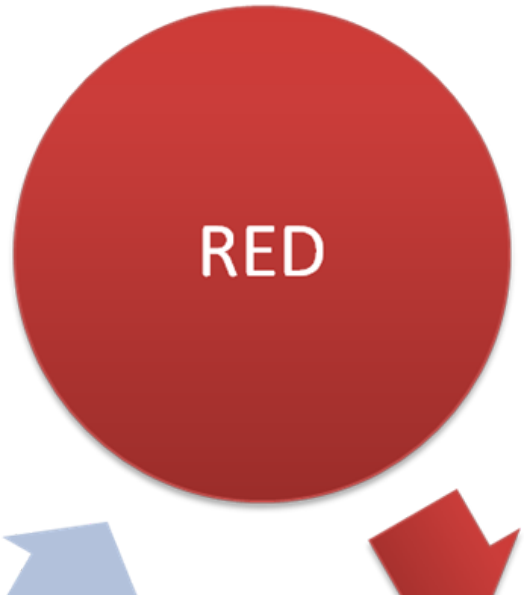
Note: Créez un clone à partir du template et partagez-le aux élèves : Exercices. Choisissez les exercices qui vous conviennent. En fonction du niveau des élèves. NB : N'incluez **PAS** les autres branches Propositions : FizzBuzz, Thermomètre

---

La suite !

- ☑ Introduction
- ☑ Outils
- ☑ Bonne Pratique

Semifir




REFACTOR
GREEN