

# --Log3430 -- Méthodes de test et de validation du logiciel



Nouredine Kerzazi |  
[Nouredine Kerzazi@polymtl.ca](mailto:Nouredine.Kerzazi@polymtl.ca)

Bram Adam |  
[Adam.bram@polymtl.ca](mailto:Adam.bram@polymtl.ca)

# Test Unitaires

## Lab #1



**Objectifs :** Développer des compétences sur les tests boîte blanche

- mettre en pratique les connaissances théoriques acquises sur les tests unitaires.
- Se familiariser avec les tests unitaires impliquant un accès aux bases de données.
- Être en mesure d'analyser des graphes de flot de contrôle
- Comprendre les critères de couverture (de conditions, chemin, flot de données)

# Agenda

---

- **Présentation, Groupes, attentes et directives**
- **(What, Why, How ) to Test ?**
- **Rappel sur les tests unitaires avec Python**
  - Unittest, Pytest
- **Démo**
- **À vous !!!**
- **Consignes de remise.**

# Qu'est ce que les tests unitaires ?

---



- Le test unitaire est un moyen de vérifier qu'un extrait de code fonctionne correctement.
- Un test unitaire doit réellement porter sur une seule unité.

# Pourquoi des tests unitaires ?

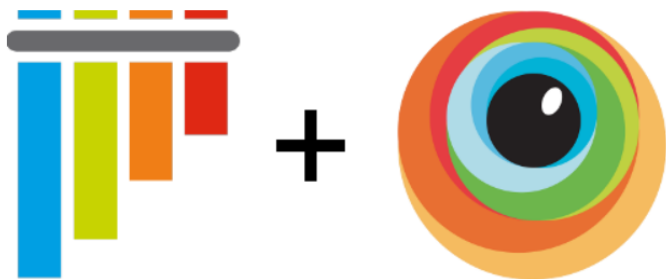
---



- **un test unitaire** permet de tester le bon fonctionnement d'une partie précise d'un programme.
- Il permet de s'assurer que le comportement d'une application est correct.

# Comment faire des tests unitaires avec Python ?

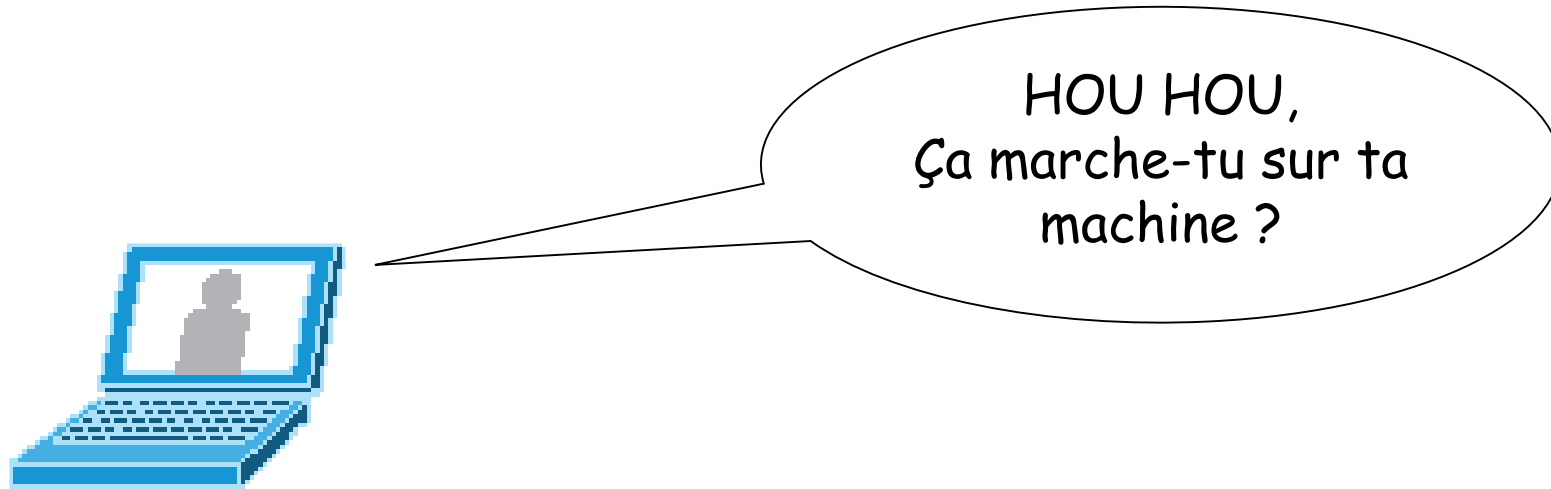
---



- Deux Frameworks les plus connus :
  - unittest
  - pytest
- Les tests unitaires couvriront en priorité les cas nominaux, les cas d'erreurs et les cas aux limites.

# Tester

---



- **Non, ca marche pas. C'est un peu Bizard !!!**



**Mobile Phone**



**Palm Top**



**Laptop**



**Windows PC**



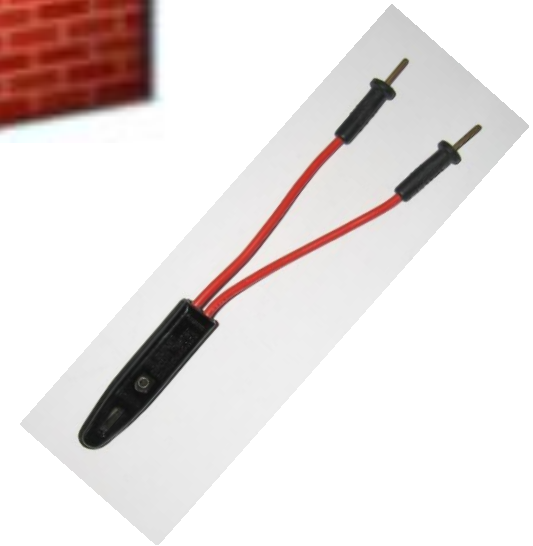
**PDA**



**Linux System**



**Firewall**





# Rappel sur les tests unitaires avec Python

---

```
def somme(x, y):  
    """Fonction de sommation"""  
    return x + y  
  
def soustraire(x, y):  
    """Fonction de soustraction"""  
    return x - y  
  
def multiplier(x, y):  
    """Fonction de multiplication"""  
    return x * y  
  
def diviser(x, y):  
    """Fonction de division"""  
    if y == 0:  
        raise ValueError('Pas de division par zero!!!')  
    return x / y
```

- Soit le fichier calcul.py
- On veut créer des tests unitaires pour les 4 fonctions de bases.

# unittest

```
import unittest
import calcul

# Voir les assertions possibles ici
# https://docs.python.org/3/library/unittest.html#unittest.TestCase.debug

class TestCalc(unittest.TestCase):
    def test_somme(self):
        resultat = calcul.somme(18, 2)
        self.assertEqual(calcul.somme(18, 2), 20)

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

```
C:\Users\nkerzazi\PycharmProjects\TestPoly>python -m unittest test_calc1.py
.
-----
Ran 1 test in 0.000s

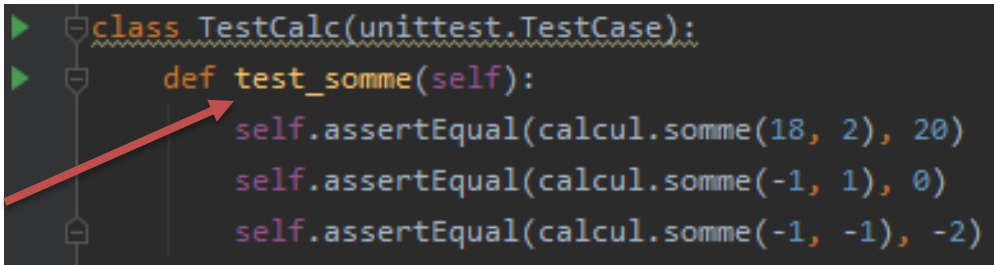
OK
```

```
=====
FAIL: test_somme (test_calc1.TestCalc)
-----
Traceback (most recent call last):
  File "C:\Users\nkerzazi\PycharmProjects\TestPoly\test_calc1.py", line 8, in test_somme
    self.assertEqual(calcul.somme(18, 2), 21)
AssertionError: 20 != 21
-----
Ran 1 test in 0.000s

FAILED (failures=1)
```

# Unittest

---



```
class TestCalc(unittest.TestCase):  
    def test_somme(self):  
        self.assertEqual(calcul.somme(18, 2), 20)  
        self.assertEqual(calcul.somme(-1, 1), 0)  
        self.assertEqual(calcul.somme(-1, -1), -2)
```

A screenshot of a code editor showing a Python class `TestCalc` that inherits from `unittest.TestCase`. Inside the class, there is a method `test_somme`. This method contains three `self.assertEqual` calls, each testing a different sum calculation. A red arrow points from the left margin to the `test_somme` method definition.

- Il s'agit d'un seul test. C'est bon avec plusieurs valeurs, mais c'est un seul test !!!
- Attention aux conventions de nommage. Toujours **test\_xxxx**.

# Plus de couverture

```
import unittest
import calcul
# Voir les assertions possibles ici
# https://docs.python.org/3/library/unittest.html#unittest.
class TestCalc(unittest.TestCase):
    def test_somme(self):
        self.assertEqual(calcul.somme(18, 2), 20)
        self.assertEqual(calcul.somme(-1, 1), 0)
        self.assertEqual(calcul.somme(-1, -1), -2)
    def test_soustraire(self):
        self.assertEqual(calcul.soustraire(18, 2), 16)
        self.assertEqual(calcul.soustraire(-1, 1), -2)
        self.assertEqual(calcul.soustraire(-1, -1), 0)
    def test_multiplier(self):
        self.assertEqual(calcul.multiplier(18, 2), 36)
        self.assertEqual(calcul.multiplier(-1, 1), -1)
        self.assertEqual(calcul.multiplier(-1, -1), 1)
    def test_diviser(self):
        self.assertEqual(calcul.diviser(18, 2), 9)
        self.assertEqual(calcul.diviser(-1, 1), -1)
        self.assertEqual(calcul.diviser(-1, -1), 1)
        self.assertEqual(calcul.diviser(18, 5), 3.6)

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

```
Terminal: Local × +
C:\Users\nkerzazi\PycharmProjects\TestPoly>python -m unittest test_calc1.py
....
-----
Ran 4 tests in 0.000s

OK
```

# Tester les exceptions soulevées – méthode 1

---

```
def diviser(x, y):  
    """Fonction de division"""  
    if y == 0:  
        raise ValueError('Pas de division par zero!!!')  
    return x / y
```

```
self.assertRaises(ValueError, calcul.diviser, 20, 0)
```

- Les paramètres de la fonction sont envoyés séparément

# Tester les exceptions soulevées – méthode 2

---

```
def diviser(x, y):  
    """Fonction de division"""  
    if y == 0:  
        raise ValueError('Pas de division par zero!!!')  
    return x / y
```

```
with self.assertRaises(ValueError):  
    calcul.diviser(20, 0)
```

- Utilisation du Context Manager

# Les critères de couverture

---

01 IConditions  
\*\*\*\*

03 Data Flow  
\*\*\*\*

02 Chemin  
\*\*\*\*

04 ???  
\*\*\*\*





À vous !!!



# Merci !

## Questions?

- [noureddine.kerzazi@polymtl.ca](mailto:noureddine.kerzazi@polymtl.ca)

**Remise .Zip bien commenté**  
**le jeudi 6 février 23h55 (B1)**