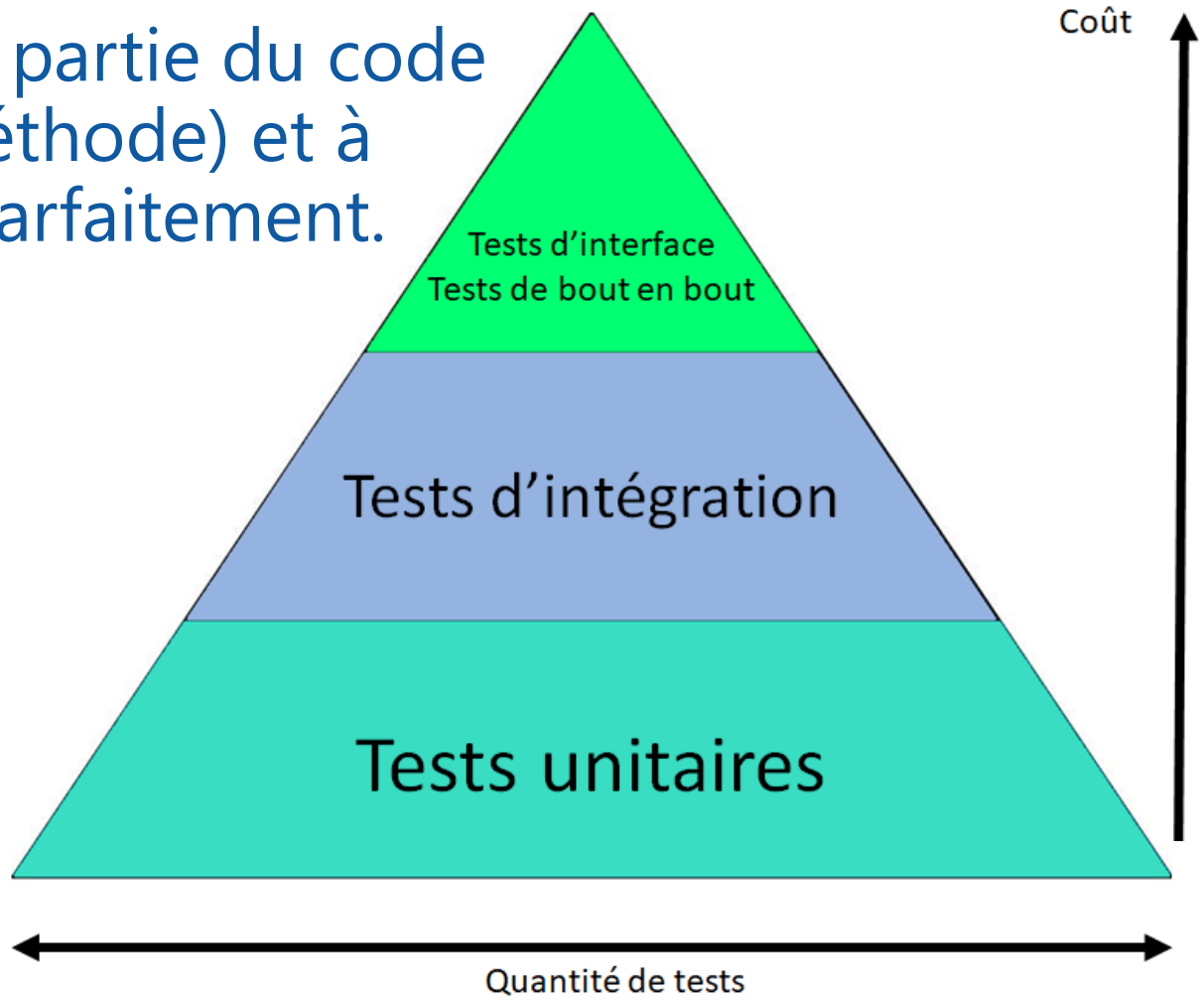
2N6 Programmation 2

avec python™

Tests Unitaires

Les tests unitaires

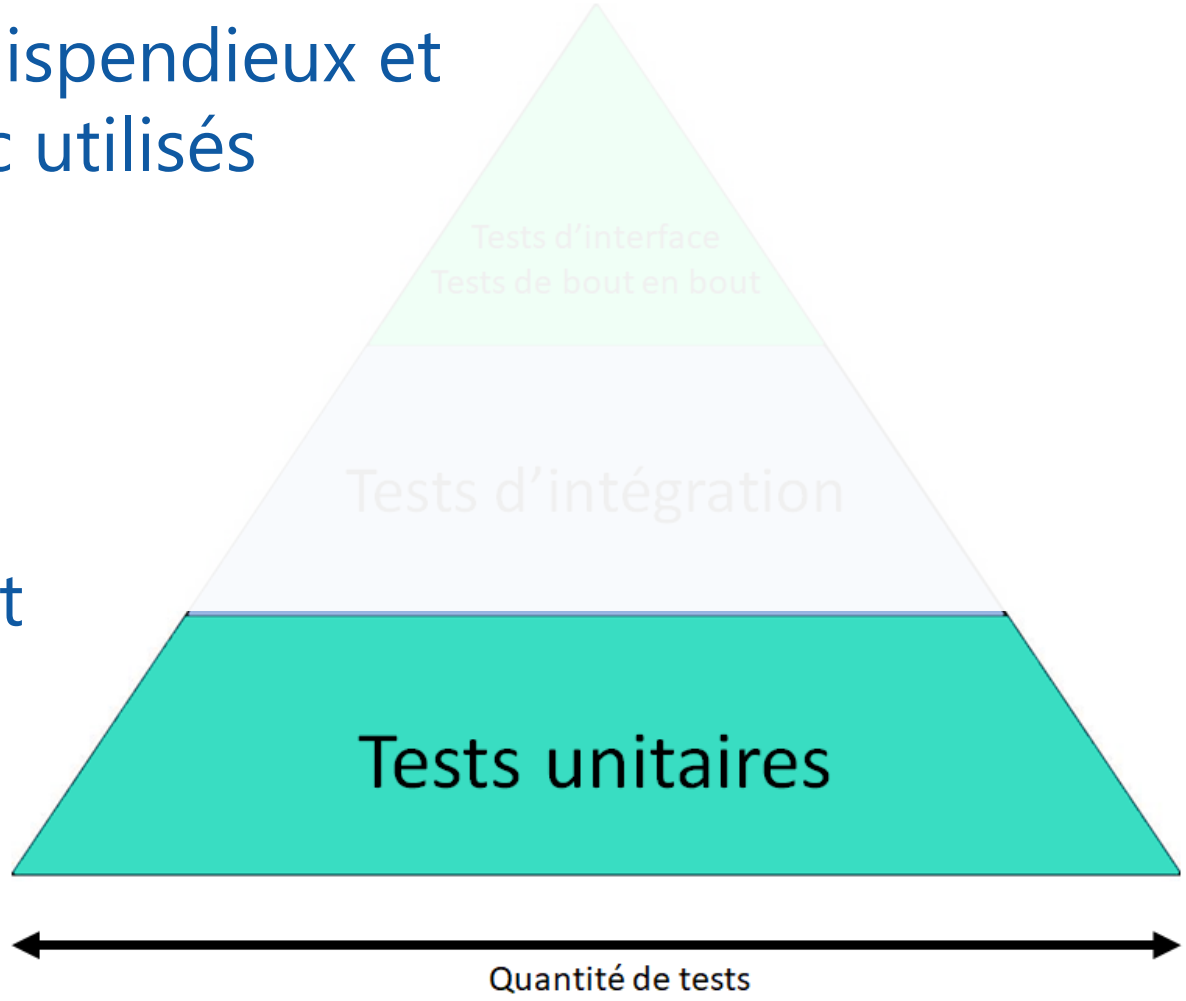
- Tests consistant à isoler une petite partie du code (généralement une fonction ou méthode) et à s'assurer que le code fonctionne parfaitement.
- UN test unitaire examine UNE seule chose à la fois
- Très utilisé lors du développement de logiciels



Les tests unitaires

- Les tests unitaires sont les moins dispendieux et les plus faciles à faire. Ils sont donc utilisés abondamment.

- Il y a 3 étapes à un test unitaire :
  - Préparer les données pour le test
  - Déclencher l'action à tester
  - Effectuer un **assert** pour vérifier le résultat de notre action.



Tests unitaires - exemple

- Revenons à notre exemple d'un jeu de Monopoly.

- La classe joueur a une méthode qui lui permet d'acheter un terrain.

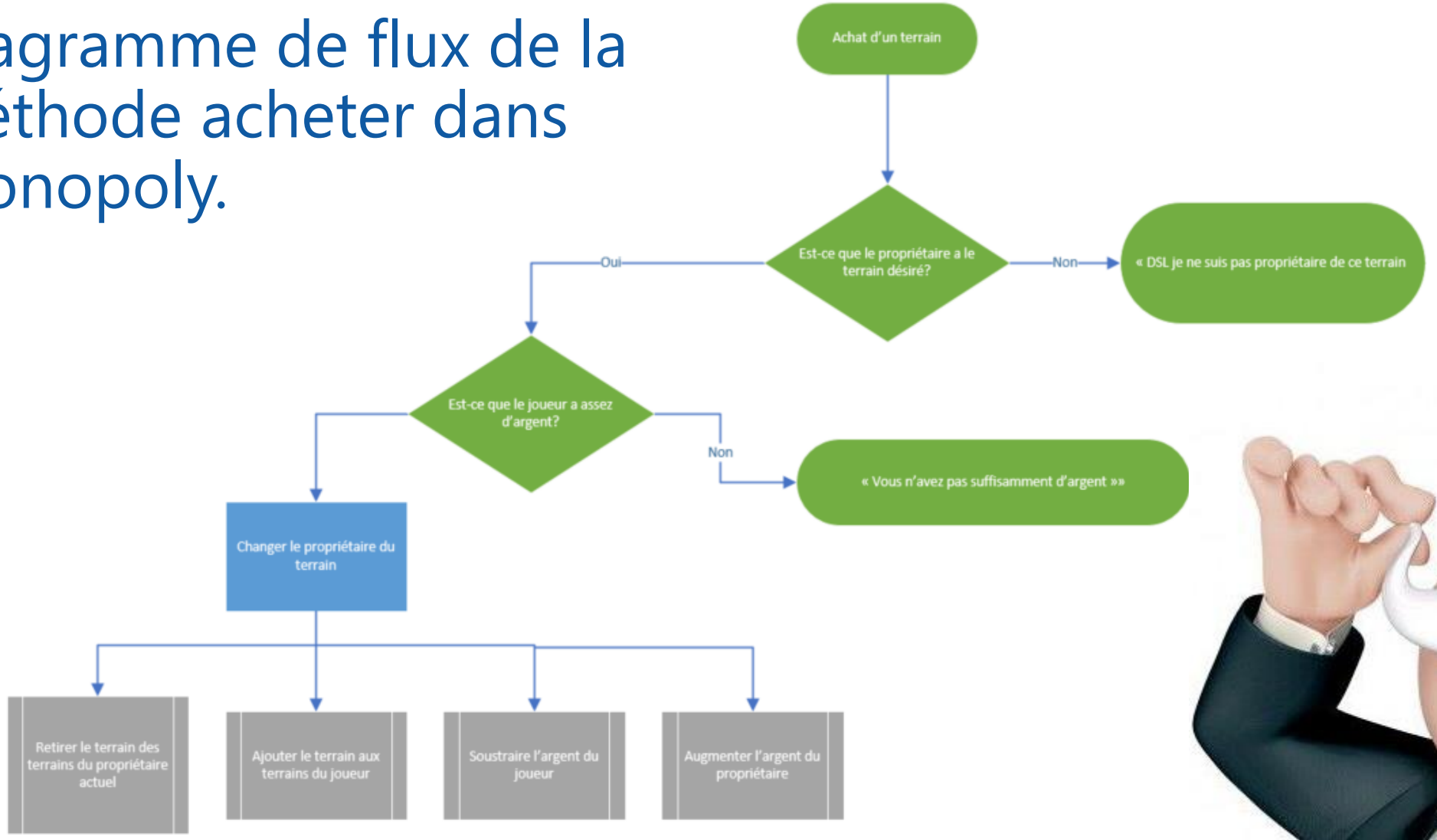
```
class Joueur:
    def __init__(self, montant_cash, ls_terrains):
        pass

    def acheter(self, proprietaire_actuel, terrain):
        pass
```

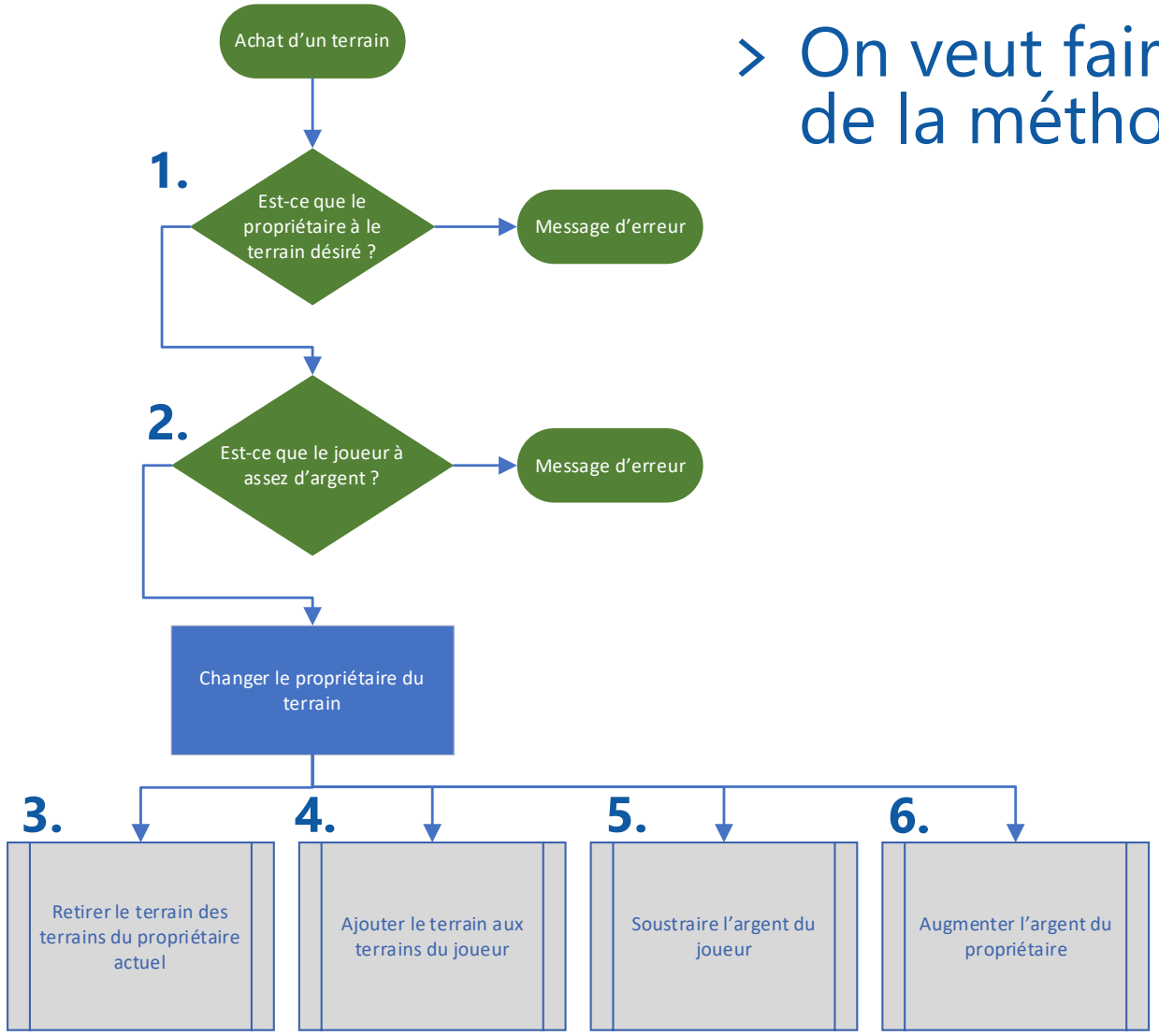
Tests unitaires - exemple

retour

Diagramme de flux de la méthode acheter dans Monopoly.



Tests unitaires - exemple

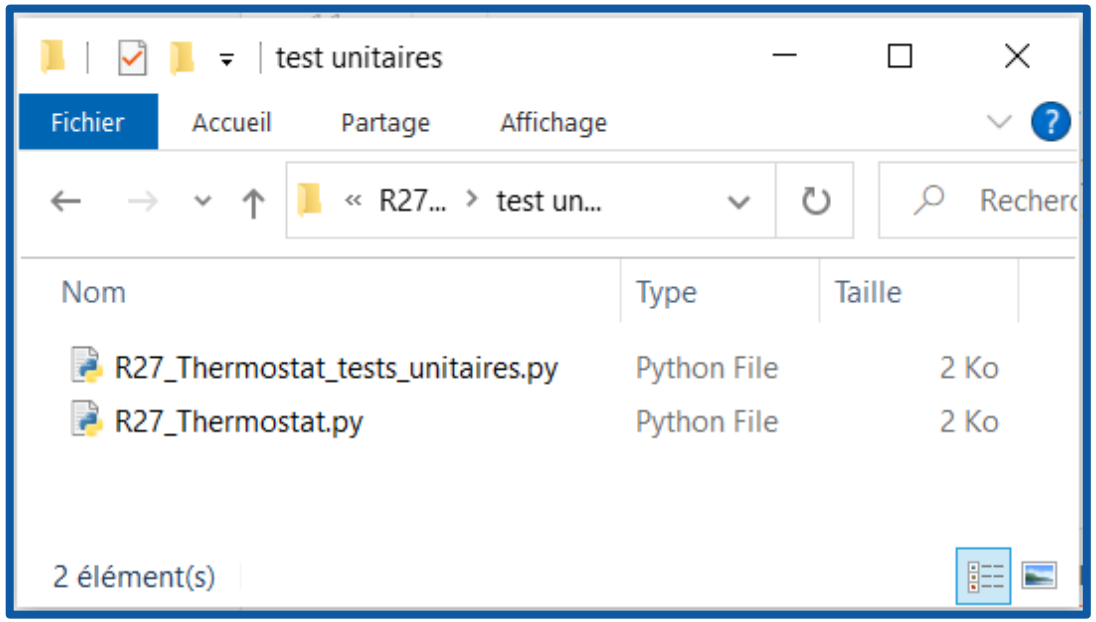


- On veut faire un test pour chacune des portions de la méthode.

1. Est-ce qu'on vérifie bien que le propriétaire initial a le terrain ?
2. Est-ce qu'on vérifie bien que le joueur a assez d'argent ?
3. Est-ce qu'on retire bien le terrain du propriétaire actuel ?
4. Est-ce qu'on ajoute bien le terrain aux terrains du joueur qui achète ?
5. Est-ce qu'on soustrait bien l'argent de ce joueur ?
6. Est-ce qu'on donne bien cet argent au propriétaire initial ?

Module unittest

- Le module **unittest** fait partie de la librairie standard.
- Permets de créer et d'exécuter facilement des tests unitaires.
- Un nouveau fichier .py est créé pour les tests.
- Ce fichier va importer le module **unittest** ainsi que le script que nous voulons tester.



https://docs.python.org/3/library/unittest.html

Structure de tests unitaires

```python
import unittest

class test_methodes_string(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('FOO', 'foo'.upper())

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello',
        'world'])
        # Vérifie que le split échoue lorsqu'on
        ne sépare pas avec un charactère
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

- Importation du module unittest
- Création d'une classe pour faire une série de tests unitaires.
- Création de méthodes, chacune testant UNE fonctionnalité.
- Appel de la fonction .main() de unittest.
  - Exécute tous les tests et fournis un rapport

https://docs.python.org/3/library/unittest.html

Exécution des tests unitaires

```
24
25  if __name__ == '__main__':
26      unittest.main(verbosity=2)
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL .NET INTE

```
test_isupper (__main__.test_methodes_string) ... ok
test_split (__main__.test_methodes_string) ... ok
test_upper (__main__.test_methodes_string) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

- Nombre de tests effectués
- Temps d'exécution
- Résumé (ok indiquant que tous les tests ont réussi)

- Après avoir déclaré nos classes et méthodes test, on appelle la fonction unittest.main()
- La fonction unittest.main() va exécuter chacune des fonctions de chaque classe et nous fournir un rapport.
- Un test est réussi si tous les **asserts** contenus réussissent.

Exécution des tests unitaires

```
24
25  if __name__ == '__main__':
26      unittest.main(verbosity=2)
```

```
PROBLÈMES   SORTIE   CONSOLE DE DÉBOGAGE   TERMINAL   .NET INTERACTIVE   JUPYT

======================================================================
FAIL: test_split (__main__.test_methodes_string)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpetit\CMT-420_I
2 (Pilote réseau)\R26-27 - tests unitaires - tp 3\demo\test_unitaire_d
    with self.assertRaises(TypeError):
AssertionError: TypeError not raised

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

- Si un test échoue, nous obtenons un message d'erreur.
- Les autres tests qui suivent sont quand même exécutés
- Une phrase résume l'**assert** qui a échoué.

Types de vérifications (assert)

```python
import unittest

class test_methodes_string(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('FOO', 'foo'.upper())

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello',
        'world'])
        # Vérifie que le split échoue lorsqu'on
        ne sépare pas avec un charactère
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

- **assertTrue** : Vérifie si le paramètre qu'on lui passe est résolu à une valeur **True**
- **assertFalse** : Vérifie que le paramètre passé est résolu à une valeur **False**
- **assertEqual** : Compare deux valeurs et vérifie si elles sont égales.
  - Par standard : On place la valeur attendue en premier, puis la valeur évaluée.
  - Ex: **assertEqual**(5, 2+2+1)
- with **assertRaises** : Vérifie que le code contenu dans cette section soulève bien l'erreur ou l'exception attendue. (le test réussi uniquement si l'erreur est soulevée)

https://docs.python.org/3/library/unittest.html

Les différentes méthodes **assert**

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

- Il y a plusieurs autres types de **assert** selon ce qu'on désire évaluer.
- Dans les exercices d'aujourd'hui, on se limitera aux plus fréquents (true, false, equals, raises

https://docs.python.org/3/library/unittest.html#assert-methods

Sauté des tests

- S'il y a des tests qu'on ne veut pas effectuer, on peut les sauter à l'aide de décorateurs **@unittest**
  - **.skip** va sauter le test automatiquement
  - **.skipif** va sauter le test uniquement si une condition est évaluée à vrai
  - **.skipunless** va sauter le test **à moins** qu'une condition soit évaluée à vrai.

- La méthode **.skipTest()** va permettre de sauter un test après qu'il est été commencé.

```python
import unittest
import sys

version_python = sys.version

class MyTestCase(unittest.TestCase):

    @unittest.skip("skip à l'aide de décorateurs")
    def test_nothing(self):
        self.fail("ne devrait pas se rendre ici")

    @unittest.skipIf(version_python < "3.10",)
    def test_format(self):
        pass

    @unittest.skipUnless(sys.platform.startswith("win"))
    def test_windows_support(self):
        pass

    def test_maybe_skipped(self):
        if "situation plus complexe que prévu":
            self.skipTest("Hors du cadre de ce test")
        pass
```

https://docs.python.org/3/library/unittest.html

Les constantes dans python

- Les constantes ne font pas partie de python contrairement à d'autres langages de programmation.
- Les constantes fonctionnent selon un standard.
  De façon similaire aux attributs privés, il s'agit d'une entente entre programmeurs quant à leur utilisation.
- Les constantes sont placées dans un fichier constant.py
- Leurs noms sont entièrement en majuscule, ex : MIN_TEMPERATURE
- On ne modifie jamais la valeur d'une constante.

Automatisation des tests

- Les tests unitaires permettent entre autres l'automatisation des tests.
- Fréquent suivant chaque commit ou avant de merge dans une branche.
- Peut-être effectué à des intervalles réguliers si on interagit beaucoup avec d'autres services (services web, api, etc.)
- Vous en verrez un peu plus dans les prochains cours, yay !

Le Développement Dirigé par Tests

- Le TDD (Test Driven Development) consiste à faire les tests en premier.
- Essentiellement :
  1. Identifier un besoin.
  2. Créer un test juste assez grand pour qu'il échoue.
  3. Créer juste assez de code/méthodes pour que le test soit un succès.
  4. Répéter jusqu'à ce que le projet soit terminé ou que weekend arrive.

- D'autres types de tests sont très utilisés dans l'industrie. Telle que les tests fonctionnels et les tests d'intégrations... vous en verrez quelques-uns dans les prochains cours.