



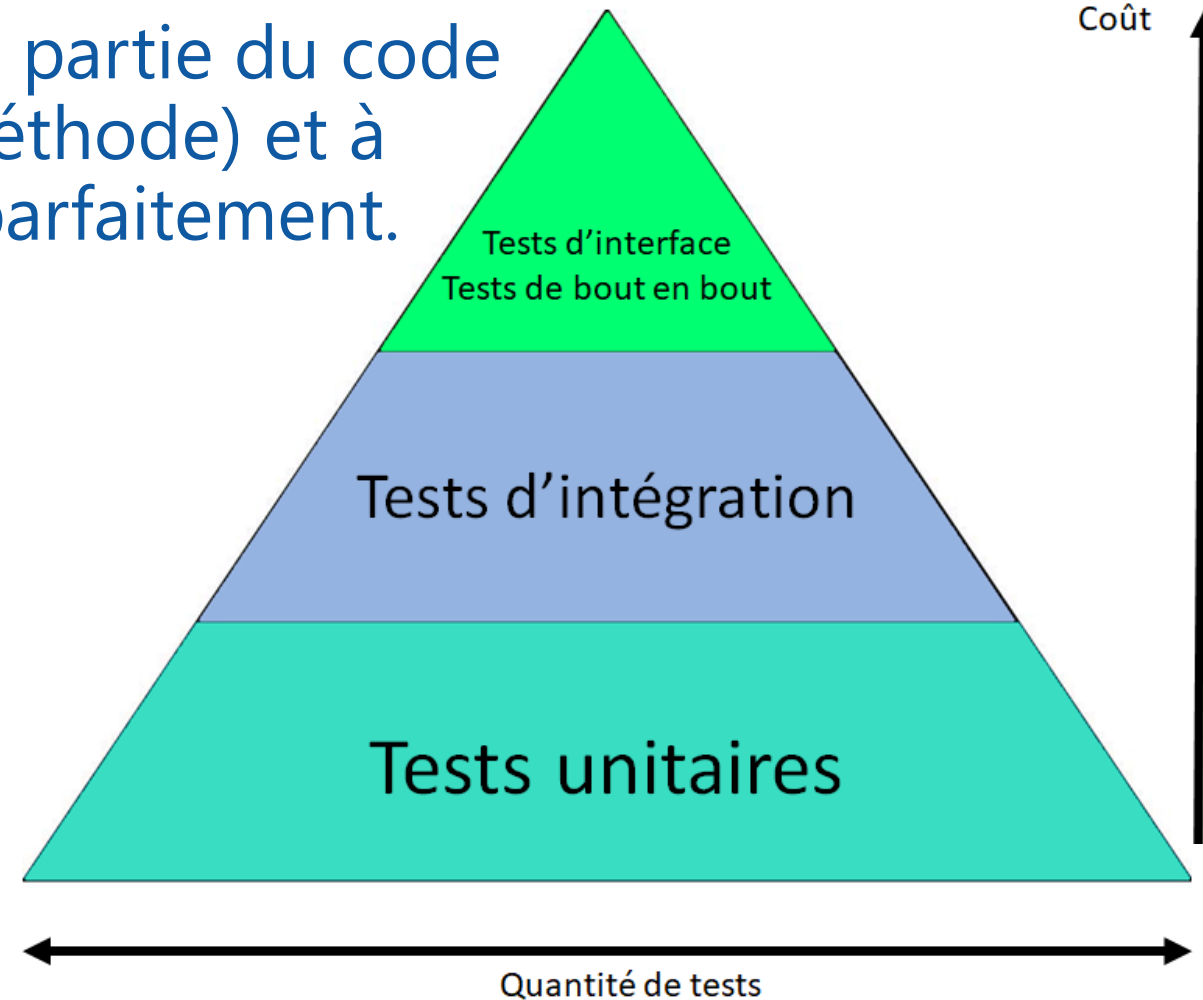
avec python™

Tests Unitaires

Les tests unitaires



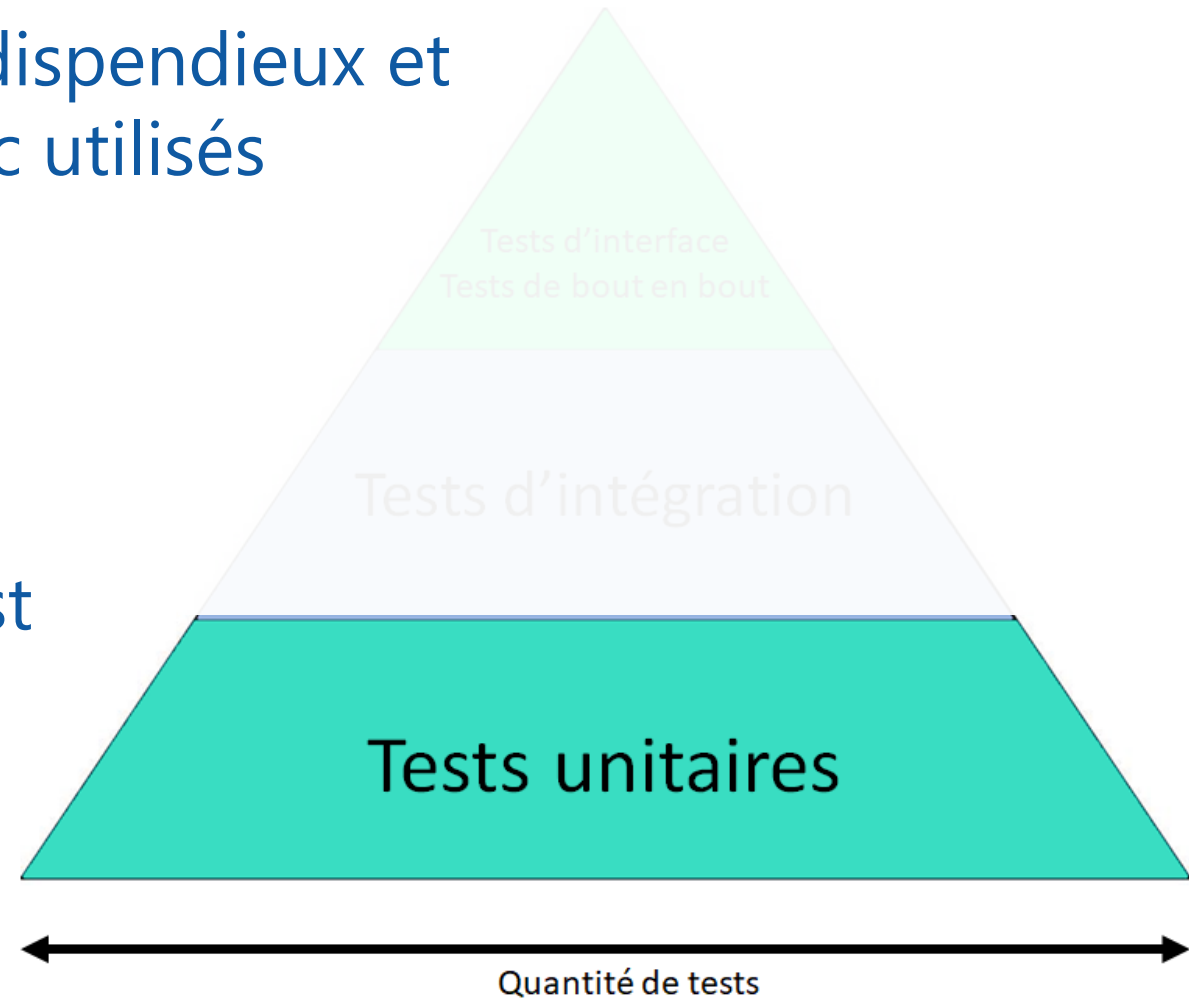
- Tests consistant à isoler une petite partie du code (généralement une fonction ou méthode) et à s'assurer que le code fonctionne parfaitement.
- UN test unitaire examine UNE seule chose à la fois
- Très utilisé lors du développement de logiciels



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

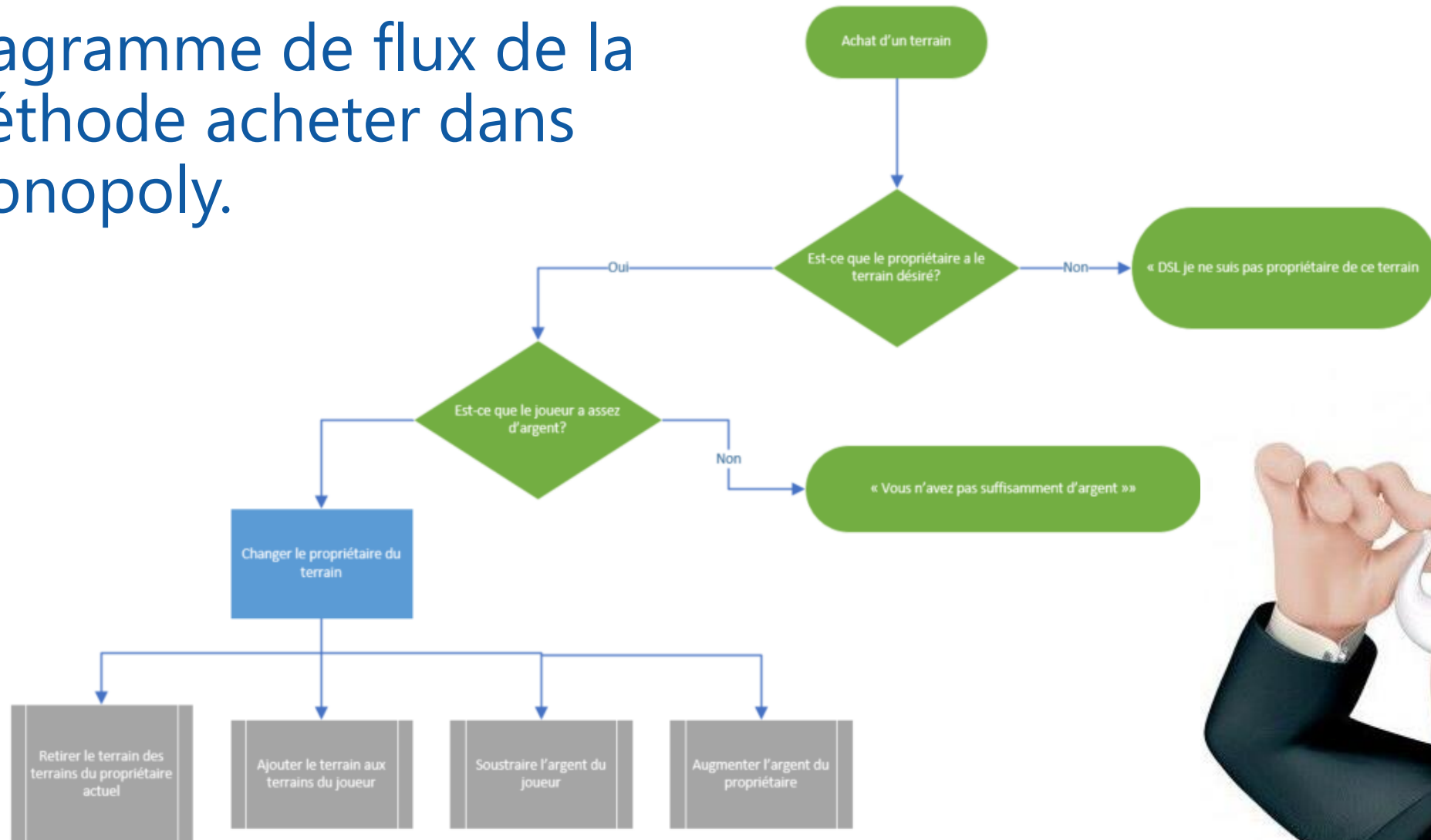
- Revenons à notre exemple d'un jeu de Monopoly.
- La classe joueur a une méthode qui lui permet d'acheter un terrain.

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

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

Tests unitaires - exemple

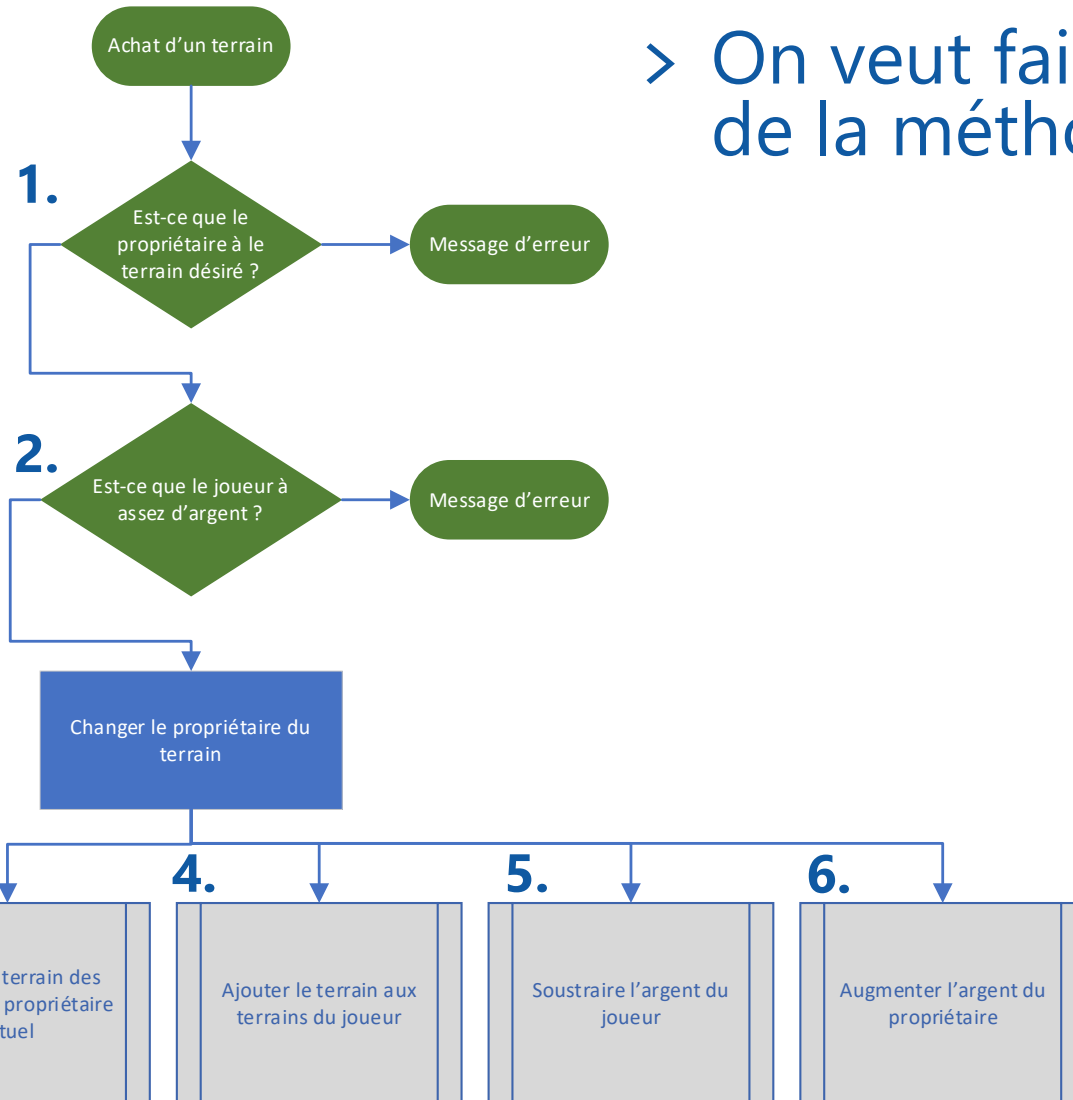
Diagramme de flux de la méthode acheter dans Monopoly.



Tests unitaires - exemple



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

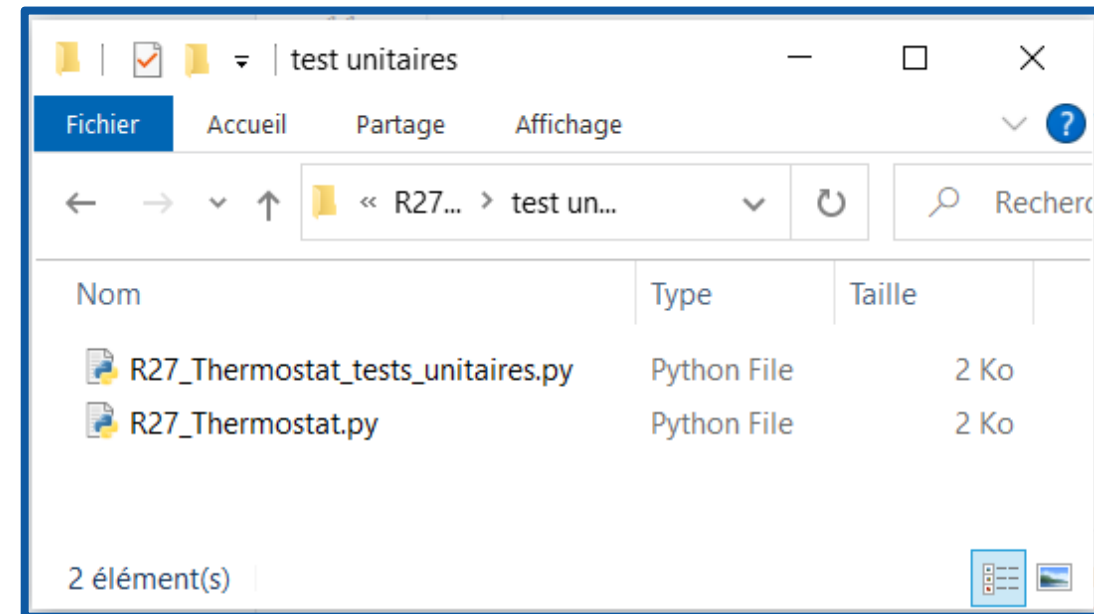


1. Est-ce qu'on vérifie bien que le propriétaire initial a le terrain ?
2. Est-ce qu'on vérifie bien que le joueur a assez d'argent ?
3. Est-ce qu'on retire bien le terrain du propriétaire actuel ?
4. Est-ce qu'on ajoute bien le terrain aux terrains du joueur qui achète ?
5. Est-ce qu'on soustrait bien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





Structure de tests unitaires

```
1 import unittest
2
3 class test_methodes_string(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('FOO', 'foo'.upper())
7
8     def test_isupper(self):
9         self.assertTrue('FOO'.isupper())
10        self.assertFalse('Foo'.isupper())
11
12    def test_split(self):
13        s = 'hello world'
14        self.assertEqual(s.split(), ['hello', 'world'])
15        # Vérifie que le split échoue lorsqu'on
16        # ne sépare pas avec un caractère
17        with self.assertRaises(TypeError):
18            s.split(2)
19
20 if __name__ == '__main__':
21     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

Exécution des tests unitaires



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

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL .NET INTER

```
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)

```
1 import unittest
2
3 class test_methodes_string(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('FOO', 'foo'.upper())
7
8     def test_isupper(self):
9         self.assertTrue('FOO'.isupper())
10        self.assertFalse('Foo'.isupper())
11
12    def test_split(self):
13        s = 'hello world'
14        self.assertEqual(s.split(), ['hello', 'world'])
15        # Vérifie que le split échoue lorsqu'on
16        # ne sépare pas avec un caractère
17        with self.assertRaises(TypeError):
18            s.split(2)
19
20 if __name__ == '__main__':
21     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)



Les différentes méthodes **assert**

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

- > 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



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.

```
1  import unittest
2  import sys
3
4  version_python = sys.version
5
6  class MyTestCase(unittest.TestCase):
7
8      @unittest.skip("skip à l'aide de décorateurs")
9      def test_nothing(self):
10         self.fail("ne devrait pas se rendre ici")
11
12         @unittest.skipIf(version_python < "3.10",)
13         def test_format(self):
14             pass
15
16         @unittest.skipUnless(sys.platform.startswith("win"))
17         def test_windows_support(self):
18             pass
19
20         def test_maybe_skipped(self):
21             if "situation plus complexe que prévu":
22                 self.skipTest("Hors du cadre de ce test")
23             pass
```



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.



- > 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.