

# Cours tests & bugs

@crédits du cours :

**Antoine MAROT** ([antoine.marot@ensemblescolaire-niort.com](mailto:antoine.marot@ensemblescolaire-niort.com))

**David SALLÉ** ([david.salle@ensemblescolaire-niort.com](mailto:david.salle@ensemblescolaire-niort.com))

**Julien SIMONNEAU** ([julien.simonneau@ensemblescolaire-niort.com](mailto:julien.simonneau@ensemblescolaire-niort.com))

Ce document est mis à disposition selon les termes de la licence  
Creative Commons BY-NC-SA 4.0



Version du document v0.2  
Date 15/10/2020

## Table des matières

<a href="#">1 - Introduction</a>	3
<a href="#">1.1 - Plan</a>	3
<a href="#">1.2 - Un peu d'histoire</a>	4
<a href="#">1.3 - Vue d'ensemble</a>	5
<a href="#">1.4 - Conseils</a>	5
<a href="#">1.5 - Erreur avec l'interpréteur Python</a>	6
<a href="#">2 - Les exceptions</a>	7
<a href="#">2.1 - Utiliser try/except</a>	8
<a href="#">2.2 - Différenciation</a>	8
<a href="#">2.3 - Et quoi qu'il arrive...</a>	9
<a href="#">2.4 - Lever soi même une Exception</a>	9
<a href="#">3 - Bogues et débogueur</a>	10
<a href="#">3.1 - Outil de débogage</a>	10
<a href="#">3.2 - Source d'information</a>	11
<a href="#">4 - Tests unitaires</a>	12
<a href="#">4.1 - Introduction</a>	12
<a href="#">4.1.1 - Avantages/inconvénients</a>	12
<a href="#">4.1.2 - Outils</a>	12
<a href="#">4.2 - Mise en œuvre</a>	13
<a href="#">4.2.1 - Fonctionnement</a>	13
<a href="#">4.2.2 - Module calculs.py</a>	13
<a href="#">4.2.3 - Tests unitaires avec les assertions</a>	14
<a href="#">4.2.4 - Tests unitaires avec la librairie unittest</a>	15
<a href="#">5 - Optimisation des performances</a>	18
<a href="#">5.1 - Principe</a>	18
<a href="#">5.2 - Chronométrage simple</a>	18
<a href="#">5.3 - Profilage</a>	18

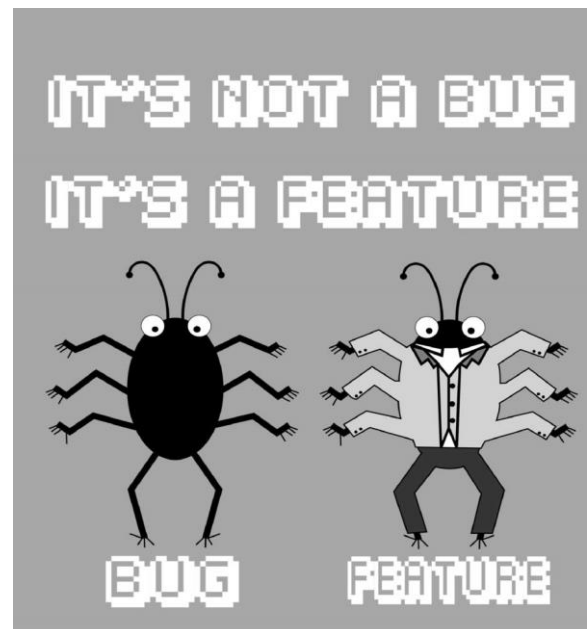
# 1 - Introduction

## 1.1 - Plan

Ce document présente quelques notions autour de la **mise au point** des programmes :

- gestion des erreurs/exceptions
- gestion des bugs
- tests unitaires
- analyse des performances

Débutons par un slogan en informatique !




## 1.2 - Un peu d'histoire

L'histoire de l'informatique est truffée d'inventions et de grandes réalisations, mais aussi parfois de désastres industriels. Les conséquences peuvent aller de simples erreurs au plantage complet du système en passant par les failles de sécurité.

Quelques exemples de bugs ou bogues tristement connus :

Quand	Quoi	Conséquence
1947	Un insecte (bug in english) est venu créer <b>un faux contact</b> entre 2 relais électromécaniques de l'ordinateur Harvard Mark II. 	Une simple erreur
1991	Une <b>erreur d'arrondi</b> pour passer d'un entier à un réel sur 24 bits a engendré une erreur de 600m dans les calculs de trajectoire du système anti-missile Patriot. 	Mort de 28 soldats américains
1996	Un <b>dépassement d'entier</b> dans le programme d'Ariane5 qui avait été en partie copié/collé d'Ariane4. Mais les variables utilisées ne convenaient plus en taille pour la puissance d'Ariane 5 	Explosion de la fusée
1997	Quelques jours après son arrivée sur Mars, le robot Sojourner de la mission Pathfinder se bloque à cause d'une <b>inversion de priorité</b> dans ses tâches. Le problème sera corrigé à distance depuis la Terre 	Retard dans la mission
2014	Le célèbre clip Gangnam Style provoque un	Rien de particulier

	<b>débordement d'entier</b> dans le compteur de vue de la plateforme Youtube PSY - GANGNAM STYLE (강남스타일) M/V 	
2038	Un <b>dépassement d'entier</b> dans la gestion de la date. Y2038 en écho à Y2000 (bug de l'an 2000) apparaîtra peut-être dans les systèmes 32 bits qui compte les secondes à partir du 1er janvier 1970 pour gérer la date du système. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre> Binary   : 100000000 000000000 000000000 000000000 Decimal  : -2147483648 Date     : 1901-12-13 20:45:52 (UTC) Date     : 2038-01-19 03:14:08 (UTC) </pre> </div>	L'année deviendra 1901 au lieu de 2038

D'autres exemples ici : [https://en.wikipedia.org/wiki/List\\_of\\_software\\_bugs](https://en.wikipedia.org/wiki/List_of_software_bugs)

### 1.3 - Vue d'ensemble

Lorsqu'on programme en Python (et dans les autres langages également), on est rapidement confronté à 3 types d'erreurs.

<b>Type d'erreur</b>	<b>Description</b>	<b>Difficulté à résoudre</b>
Syntaxe liée au langage	Le code source comporte des erreurs et l'interpréteur n'arrive pas à l'exécuter. Il vous indique en général <i>Solution : se référer au message de l'interpréteur Python et vérifier la syntaxe</i>	Facile
Valeur ou action inappropriée	Le programme se lance bien, mais dans certaines conditions "plante". <i>Solution : se référer au message d'erreur de l'interpréteur Python et vérifier les valeurs notamment à l'aide d'un débogueur</i>	Difficile
Conception de l'algorithme	Le programme se lance bien, ne plante pas, mais ne produit pas le résultat escompté <i>Solution : revoir l'algorithme utilisé et les étapes de l'exécution du programme avec un débogueur</i>	Très difficile

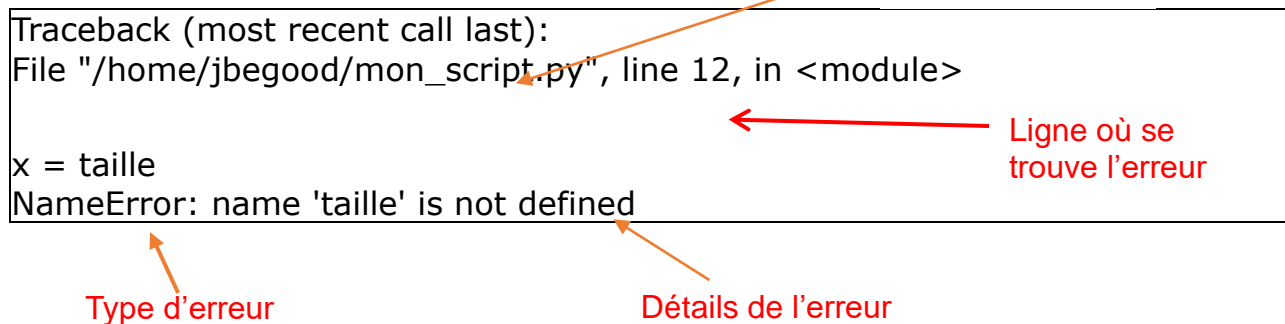
### 1.4 - Conseils

Conseils lorsqu'on crée un script Python :

- corriger les erreurs de syntaxe
- anticiper les erreurs de saisie de l'utilisateur, imaginer le pire
- tester unitairement (entrées => sorties, cas limite avec 0, "", tous les cas de figure if/else)
- vérifier les boucles while et leurs conditions de sortie (invariant)
- contrôler les erreurs avec try/except
- se méfier des calculs et comparaisons avec les nombres flottants

## 1.5 - Erreur avec l'interpréteur Python

Anatomie d'un message d'erreur de l'interpréteur Python :

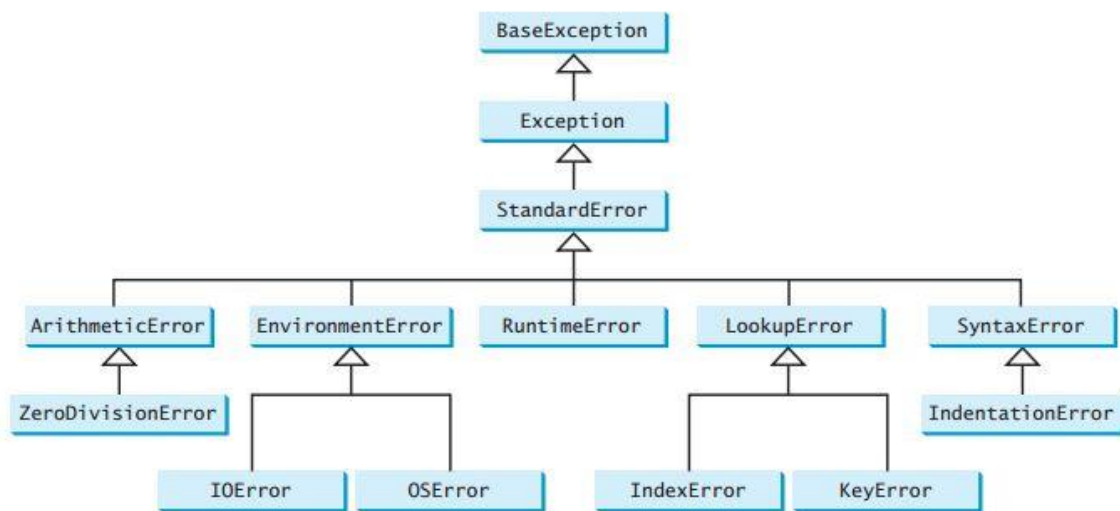


A noter que quand on utilise des fonctions qui appellent d'autres fonctions, le message d'erreur retrace tout le parcours et il faut commencer à lire par la fin. Quelques erreurs courantes liées au langage Python :

Type	Description
<b>SyntaxError</b>	Erreur de syntaxe : <ul style="list-style-type: none"><li>· confusion entre <code>=</code> et <code>==</code></li><li>· oubli d'une parenthèse <code>(</code> ou <code>)</code></li><li>· oubli d'un délimiteur de string <code>"</code> ou <code>'</code> ou <code>"""</code></li><li>· oubli du <code>:</code> annonçant un bloc</li></ul>
<b>NameError</b>	Erreur sur le nom d'une variable ou fonction <ul style="list-style-type: none"><li>· mauvais nom</li><li>· déclaration au mauvais endroit</li><li>· oubli d'import d'un module</li></ul>
<b>IndentationError</b>	Erreur d'indentation <ul style="list-style-type: none"><li>· mauvais alignement</li><li>· mélange d'espaces et tabulations (Python3)</li></ul>
<b>TypeError</b>	Erreur sur les types <ul style="list-style-type: none"><li>· addition impossible entre <code>int</code> et <code>string</code></li><li>· arrondi d'un <code>string</code></li></ul>
<b>AssertionError</b>	Erreur issue d'un test unitaire
<b>ZeroDivisionError</b>	Erreur de division par 0
<b>KeyError</b>	Erreur sur un dictionnaire
<b>IOError</b>	Erreur lors d'une entrée/sortie (fichier...)

## 2 - Les exceptions

Toutes les erreurs du tableau précédent sont en fait des exceptions qui **héritent** de la même classe mère **Exception**.



Lorsque Python lève une exception telle que celles présentes dans le tableau, le programme est stoppé immédiatement et un message d'erreur est affiché pour prévenir l'utilisateur de la cause du problème.

Prenons l'exemple de ce programme qui saisit un nombre et calcule son inverse :

```
# Version normale
x = int(input("Saisir x : "))
print("1/x =", 1/x)
```

Selon ce que va saisir l'utilisateur, le résultat sera très différent. En cas de mauvaise saisie, le programme stoppe.

Saisie d'un entier x=4	Saisie d'un string x="2"	Saisie d'un entier x = 0
Saisir x : 4 1/x = 0.25	Traceback (...): File "test.py", line 1... ValueError: invalid literal for int() with base 10: "2"	Traceback (...): File "/test.py", line 2... ZeroDivisionError: division by zero

## 2.1 - Utiliser try/except

Cependant parfois, il est plus intéressant de "capturer" cette exception de manière à informer l'utilisateur du problème et lui proposer une alternative, plutôt que de stopper brutalement le programme.

Pour cela on utilisera les mot-clefs try et except :

- **try** va débiter un bloc d'instructions à risque, susceptible de générer une erreur
- **except** va intercepter l'erreur pour éviter l'arrêt brutal du programme et proposer un traitement du problème, souvent un affichage

Le modèle est celui-ci :

```
try:
    # Code à risque
except Exception as erreur:
    # Traitement de l'erreur (affichage...)
```

Pour le programme qui calcule l'inverse cela donnerait :

```
# Version avec capture simple et affichage
try:
    x = int(input("Saisir x : "))
    print("1/x =", 1 / x)
except Exception as erreur:
    print("Erreur :", erreur)
```

## 2.2 - Différenciation

Si l'on souhaite différencier le traitement des erreurs, il faudra ajouter plusieurs blocs **except**, typiquement un par type d'erreur

```
# Version avec capture et traitement différenciés
try:
    x = int(input("Saisir x : "))
    print("1/x =", 1 / x)
except ValueError as erreur:
    print("Erreur de saisie :", erreur)
except ZeroDivisionError as erreur:
    print("Impossible de diviser par zéro :", erreur)
```

## 2.3 - Et quoi qu'il arrive...

On peut également ajouter un bloc d'instructions qui sera exécuté quoiqu'il arrive : erreur ou comportement normal. Pour cela on ajoutera à la fin le mot clef

**finally :**

```
# Version avec capture et traitement différenciés et
# quoiqu'il arrive un message de fin
try:
    x = int(input("Saisir x : "))
    print("1/x =", 1 / x)
except ValueError as erreur:
    print("Erreur de saisie :", erreur)
except ZeroDivisionError as erreur:
    print("Impossible de diviser par zéro :", erreur)
finally:
    print("Merci d'avoir utiliser ce calculateur")
```

## 2.4 - Lever soi-même une Exception

Quand on écrit son propre module Python avec ses fonctions, ses classes, ses méthodes, on a souvent besoin de lever nous-même des exceptions sans attendre l'interpréteur Python.

A l'aide du mot-clef **raise**, on peut lever une exception à tout moment.

```
# Levée d'exception
x = int(input("Saisir x : "))
if x == 13:
    raise ValueError("Désolé, je suis superstitieux")
print("1/x =", 1/x)
```

Celle-ci sera interceptée par un bloc try/except ou pas. Il est également possible de définir ses propres exceptions personnalisées en la rattachant à la classe Exception via l'héritage

```
# Définition d'une Exception personnalisée
class SuperstitionError(Exception):
    """Classe représentation l'erreur liée au nombre 13"""
    pass

# Levée d'une exception personnalisée
x = int(input("Saisir x : "))
if x == 13:
    raise SuperstitionError("Désolé, je suis superstitieux")
print("1/x =", 1/x)
```

## 3 - Bogues et débogueur

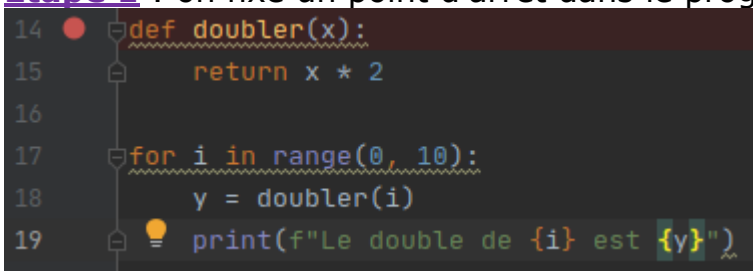
### 3.1 - Outil de débogage

Le débogueur est un outil intégré dans la majorité des IDE. Il permet de fixer des points d'arrêt (**breakpoints**) et de consulter l'état des variables (**watches**). On peut ensuite avancer pas à pas en mode :

- **Step over** (F8) : saute par dessus les fonctions
- **Step into** (F7) : entre dans les fonctions

Exemple d'utilisation dans PyCharm.

**Etape 1** : on fixe un point d'arrêt dans le programme dans la zone à déboguer

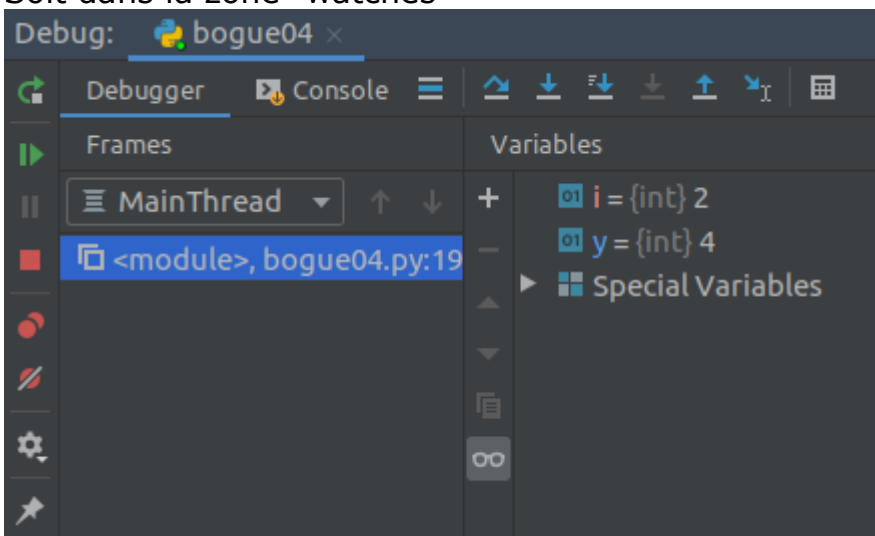


**Etape 2** : on lance l'exécution du programme en mode debug (icône insecte)



**Etape 3** : on visualise l'état des variables

Soit dans la zone "watches"



Soit directement dans le code

```

14 ● def doubler(x):
15     return x * 2
16
17     for i in range(0, 10):    i: 2
18         y = doubler(i)    y: 4
19     print(f"Le double de {i} est {y}")

```

**Etape 4** : on avance pas à pas avec la touche F8 ou F7

**Etape 5** : on reboucle à l'étape 4 jusqu'à trouver l'origine du bogue

### 3.2 - Source d'information

Où trouver de l'aide quand on a un bogue

- <https://stackoverflow.com/>
- <https://github.com/>

Les types de vulnérabilités les plus courantes liées à des bogues :

- <https://www.cvedetails.com/vulnerabilities-by-types.php>
- <https://owasp.org/www-project-top-ten/>

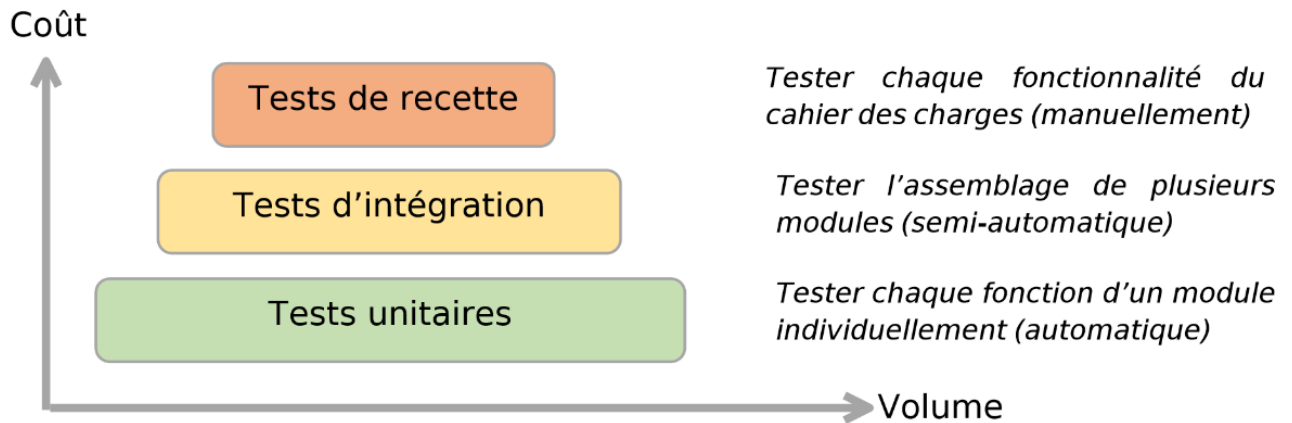


## 4 - Tests unitaires

### 4.1 - Introduction

Les **tests unitaires** consistent à vérifier le bon fonctionnement d'une portion de code informatique, typiquement une fonction.

Ils s'inscrivent dans une démarche plus générale de tests qui consiste à valider qu'un produit répond bien au besoin exprimé en amont. Pyramide des tests :



Certaines méthodes de développement comme **TDD** (Test Driven Development) propose même d'écrire les tests avant le code source.

#### 4.1.1 - Avantages/inconvénients

Avantages :

- s'assurer du bon fonctionnement
- trouver des bugs le plus tôt possible
- garantir la non régression du code après la correction d'un bug
- servir de documentation et d'exemple en complément de l'API

Inconvénients :

- cela peut être long et fastidieux à réaliser
- 

#### 4.1.2 - Outils

Plusieurs outils sont disponibles dont certains seront présentés en détails dans la suite du document :

- **assert** : mot clef Python
- **unittest** : librairie standard de tests unitaires
- **doctest** : librairie standard utilisant les tests unitaires dans les docstring
- **pytest** : librairie de tests unitaires
- 

## 4.2 - Mise en œuvre

### 4.2.1 - Fonctionnement

Imaginons que vous avez écrit un module Python nommé **calculs.py** contenant différentes fonctions.

Pour réaliser des tests unitaires sur votre module, vous allez devoir créer un nouveau fichier **test\_calculs.py** dont le rôle sera de valider le bon fonctionnement de votre module rectangle.py.

Cela se fera en 3 étapes :

1. **initialisation** (setUp) : préparation d'un environnement pour exécuter le test (fixture)
2. **vérification** (test\_xxx) : comparaison du résultat obtenu avec le résultat

escompté

3. **désactivation** (tearDown) : désinstallation de l'environnement de test afin de ne pas polluer les tests suivants

Les tests doivent être indépendants et reproductibles, c'est ce qui explique les étapes 1 et 3.

#### 4.2.2 - Module calculs.py

Soit le module **calculs.py** ci-dessous à tester.

```
def additionner(a, b):  
    """Calcule et retourne la somme de a et b"""  
    return a * b  
  
def etre_pair(n):  
    """Test la parité d'un nombre"""  
    if n % 2 == 0:  
        return True  
    else:  
        return False
```

Vous aurez noté que le développeur a fait une faute de frappe dans la fonction additionner() en remplaçant le signe + par \*.

Tout l'intérêt des tests unitaires va être de repérer cette erreur suffisamment tôt, idéalement avant l'intégration du module calcul.py dans un autre projet. Car une fois intégré, l'erreur peut s'avérer beaucoup plus longue et complexe à trouver et corriger.

#### 4.2.3 - Tests unitaires avec les assertions

Afin de tester ce module **calculs.py**, il va falloir créer un fichier **test\_calculs.py** contenant nos tests unitaires.

Ce fichier va utiliser le mot clef Python **assert** pour vérifier que le résultat produit par une fonction est bien conforme aux attentes. Si ce n'est pas le cas une erreur **AssertionError** stoppe le programme.

```
# Assertion vraie, ne produisant pas d'erreur  
assert 1 == 1, "1 doit être égal à 1"  
  
# Assertion fausse, produisant une AssertionError  
assert 1 == 0, "1 = 0 devrait lever une erreur"  
  
Traceback (most recent call last):  
  File "tests_assert.py", line 4, in <module>  
    assert 1 == 0, "1 = 0 devrait lever une erreur"  
AssertionError
```

Pour les tests unitaires du module calculs.py, on pourrait écrire les assertions suivantes :

```
# Importation du module à tester et ses fonctions  
from calculs import additionner, etre_pair  
  
# Tests unitaires de la fonction additionner()  
assert additionner(2, 2) == 4, "2 + 2 = 4"  
assert additionner(0, 0) == 0, "0 + 0 = 0"  
assert additionner(11, 5) == 16, "11 + 5 = 16"
```

```
# Tests unitaires de la fonction etre_pair()
assert etre_pair(5) == False, "5 n'est pas pair"
assert etre_pair(4) == True, "4 est pair"
assert etre_pair(0) == True, "0 est pair"
```

Le lancement de **test\_calculs.py** produira alors l'erreur suivante :

```
Traceback (most recent call last):
  File "tests_calculs.py", line 7, in <module>
    assert additionner(11, 5) == 16, "11 + 5 = 16"
AssertionError
```

Notre script de tests unitaires a bien repéré une erreur sur la fonction `additionner()` qu'il devrait être aisé de corriger.

A noter que les 2 premiers tests sont passés malgré l'erreur de codage, d'où l'importance d'être le plus exhaustif possible dans l'écriture des tests.

Cependant on pourra remarquer quelques faiblesses avec cette méthode `assert` :

- le script s'est arrêté à la première erreur, donc on ne sait pas si la fonction `etre_pair()` est correcte
- quand tous les tests passeront avec succès, le script ne produira rien
- certains tests sont plus difficiles à réaliser avec `assert` (chaînes de caractères, sortie affichage, plusieurs fonctions...)
- si chaque test demandait une initialisation avant de s'exécuter (ouverture fichier, base de données, réseaux...) il faudrait le faire avant chaque `assert` => très lourd car de nombreux copier/coller

#### 4.2.4 - Tests unitaires avec la librairie `unittest`

Avec la librairie **unittest** les mêmes tests qu'avec **assert** pourraient ressembler à :

```
# Librairies utilisées
import unittest
from calculs import additionner, etre_pair

class TestCalculs(unittest.TestCase):
    """Classe gérant les tests unitaires du module calculs"""

    def test_additionner(self):
        """Tests unitaires de la fonction additionner()"""
        self.assertEqual(additionner(2, 2), 4, "2 + 2 => 4")
        self.assertEqual(additionner(0, 0), 0, "0 + 0 => 0")
        self.assertEqual(additionner(11, 5), 16, "11 + 5 => 16")

    def test_etre_pair(self):
        """Tests unitaires de la fonction etre_pair()"""
        self.assertEqual(etre_pair(5), False, "5 est impair")
        self.assertEqual(etre_pair(4), True, "4 est pair")
        self.assertEqual(etre_pair(0), True, "0 est pair")

# Lancement des tests
if __name__ == '__main__':
    unittest.main()
```

On retrouve dans ce code :

<b>Code python</b>	<b>Description</b>
<code>import unittest</code>	Importation de la librairie unittest qui fait partie de la librairie standard Python, donc pas d'installation
<code>class TestCalculs(unittest.TestCase):</code>	Création d'une classe regroupant les tests unitaires et héritant de TestCase
<code>def test_additionner(self):</code>	Fonction débutant par test_xxx signifiant qu'elle embarque des tests unitaires
<code>self.assertEqual(     additionner(2, 2),     4,     "2 + 2 =&gt; 4")</code>	Assertion vérifiant le résultat retourné par une fonction accompagnée d'un message
<code>if __name__ == '__main__':     unittest.main()</code>	Lancement des tests, la librairie va rechercher seul les tests à effectuer (méthodes débutant par test_xxx et les exécuter)

Le lancement de ce script de test produira le résultat suivant :

```
$ python3 -m unittest
```

```
=====
=====
FAIL: test_additionner (test_calculs_unittest.TestCalculs)
Tests unitaires de la fonction additionner()
-----
Traceback (most recent call last):
  File "test_calculs.py", line 21, in test_additionner
    self.assertEqual(additionner(11, 5), 16, "11 + 5 => 16")
AssertionError: 55 != 16 : 11 + 5 => 16

-----
Ran 5 tests in 0.001s

FAILED (failures=1, errors=1)
```

Par rapport à la méthode assert, on peut remarquer que :

- tous les tests ont été lancés, le script ne s'est pas arrêté à la première erreur
- d'autres méthodes de tests que assertEquals() existent :
  - o assertTrue() : teste si égal à True
  - o assertIsNone() : teste si vaut None
  - o assertAlmostEqual() : teste si égal avec une marge d'erreur
  - o assertRaises() : teste si une exception est levée
  - o assertRegex() : teste une chaîne de caractères
  - o assertLogs() : teste si un log a bien été produit

La librairie unittest permet également d'initialiser (setUp) l'environnement (fixture) avant l'exécution de chaque test et aussi de le désactiver (tearDown) pour le prochain. Dans cet exemple on ouvre et ferme un fichier de données :

```

class TestCalculs(unittest.TestCase):
    """Classe gérant les tests unitaires du module calculs"""

    def setUp(self):
        """Initialisation des tests"""
        # Ouverture du fichier contenant le jeu de test
        self.fichier = open("data.json", "r")

    def tearDown(self):
        """Désactivation des tests"""
        # Fermeture du fichier contenant le jeu de test
        self.fichier.close()

# ...suite de la classe tronquée

```

## 5 - Optimisation des performances

### 5.1 - Principe

Les gains d'optimisation sont souvent liés :

- à l'algorithme utilisé
- au langage lui même et sa machinerie interne

### 5.2 - Chronométrage simple

Le chronométrage d'une portion de code ou d'une fonction peut s'effectuer avec la fonction **perf\_counter()** du module **time** comme ci-dessous :

```

# Importation des outils pour mesurer le temps
import time

# Fonction qu'on souhaite chronométrer
def additionner(limite):
    somme = 0
    for i in range(0, limite+1):
        somme += i
    return somme

# Lancement et chronométrage de la fonction
debut = time.perf_counter()
resultat = additionner(1000000)
fin = time.perf_counter()

# Affichage du résultat et de la durée d'exécution
print(f"La somme jusqu'à 1000000 est {resultat}")
delai = fin - debut
print(f"Durée d'exécution = {delai} s")

```

### 5.3 - Profilage

Le profilage d'un code donne des résultats beaucoup plus complet avec pour chaque fonction :

- le temps total d'exécution
- le nombre d'appels
- la durée moyenne d'un appel
- ...

Le module **cProfile** présent dans la distribution standard de Python permet de réaliser facilement ce profilage. Il suffit de lancer l'exécution d'un script depuis un terminal comme ci-dessous :

```
jbegood@pc:$ python3 -m cProfile ./time04.py
1001004 function calls in 14.529 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1      0.000      0.000   14.529   14.529 time04.py:1(<module>)
      1      0.001      0.001   14.529   14.529 time04.py:1(faire_quelque_chose)
     1000      0.180      0.000   14.529      0.015 time04.py:5(faire_autre_chose)
1000000  14.349      0.000   14.349      0.000 time04.py:9(faire_sembant)
      1      0.000      0.000   14.529   14.529 {built-in method builtins.exec}
      1      0.000      0.000      0.000      0.000 {method 'disable' of '_...' objects}
```

On peut noter par exemple que la fonction **faire\_sembant()** a été appelée 1000000 de fois pour un temps total d'exécution de 14,349s soit presque l'intégralité du temps d'exécution du programme complet. C'est précisément sur cette fonction qu'il faudrait focaliser les optimisations.