# 10.2 Tester son programme



Lorsque l'on exécute un programme, il peut fonctionner comme prévu, mais il peut aussi "planter", ou bien boucler indéfiniment, ou bien ne pas produire le résultat escompté.
Et même s'il fonctionne comme prévu, rien ne garantit que c'est toujours le cas quelles que soient les données d'entrée qu'on lui fournit.

### Définition

**Tester** un programme (ou une fonction, une méthode, une classe, un module) consiste à vérifier qu'il fonctionne comme prévu dans un maximum de situations possibles. Un **jeu de test** est un programme ou un ensemble de programmes qui essaient de mettre en défaut la partie testée. Si un test échoue, il faut mettre au point le code, c'est-à-dire identifier puis corriger le problème, et effectuer à nouveau le test.

## 10.2.1 Les assertions

Une assertion permet de vérifier une condition . Si cette condition est vraie, alors l'assertion sera muette. Si elle est fausse, alors une erreur sera produite.
Voici un exemple :

```
def racine_carree(a):
        assert a >= 0, "Le nombre doit être positif"
        return a ** 0.5
```

```
>>> racine_carree(-4)
```

renvoie le message d'erreur

### Remarque

Même si la condition est évidente, il est bien d'utiliser les assertions. Au contraire, plus la condition est évidente, plus on a intérêt à l'utiliser ! Si votre programme plante et que vous n'avez pas placé d'assertions, vous risquez d'aller chercher l'erreur ailleurs pendant un certain temps.
Si l'assertion signale l'erreur, on le sait tout de suite :-)

**Exercice 10.5**
Ecrire une assertion qui vérifie qu'un paramètre `n` est un entier strictement positif

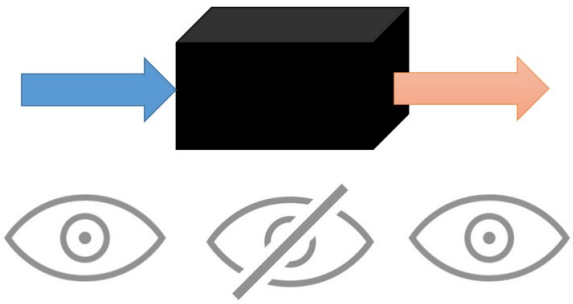
### 10.2.2 Plusieurs types de tests

On utilise plusieurs types de tests, à différentes étapes du développement. On commence par des **tests unitaires**, qui concernent une petite unité d'un programme, typiquement une fonction ou une méthode.
On effectue ensuite des **tests d'intégration**, qui consistent à vérifier que deux parties ou plus d'un programme fonctionnent correctement ensemble.
☛ Par exemple, on teste séparément (tests unitaire) une partie qui obtient des données d'un site Web, une autre qui réalise des calculs sur ces données et une troisième qui les affiche sous forme graphique. Puis on teste l'ensemble du programme (test d'intégration).

On peut être amené à effectuer des **tests de performance** pour vérifier que le logiciel se comporte correctement lorsqu'il est confronté à des données de grande taille ou, pour un serveur, à un grand nombre de connexions.
Enfin, les **tests d'utilisabilité** évaluent l'ergonomie du programme, c'est-à-dire la capacité des utilisateurs à effectuer les tâches souhaitées avec le programme de manière simple et efficace, grâce à une interface appropriée.

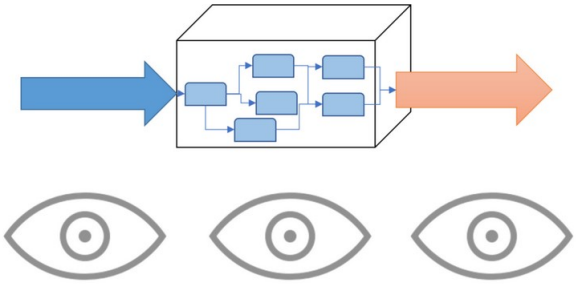
Les tests unitaires et d'intégration se divisent en deux catégories :

- en **tests fonctionnels**, qui vérifient qu'un programme ou une partie de programme (par exemple une fonction) est bien conforme à sa spécification, et tests structurels, qui vérifient le fonctionnement interne du programme. Les tests fonctionnels sont souvent appelés tests "boîte noire" car ils sont définis à partir de la spécification de l'entité testée, sans connaissance de son implémentation.



boîte noire

- À l'inverse, les **tests structurels** sont aussi appelés tests "boîte blanche" car ils sont écrits en fonction de l'implémentation du programme et cherchent à couvrir les différents chemins d'exécution : les branches d'une conditionnelle, les cas limites d'une boucle, etc.



boîte blanche

Par exemple, un test fonctionnel d'une pile vérifie que lorsque l'on empile un élément, le sommet de la pile est bien cet élément, tandis qu'un test structurel d'une pile implémentée par un tableau de taille fixe vérifie que l'on ne peut pas empiler un élément si la pile est pleine.

Une analogie est souvent utilisée pour différencier ces techniques, en comparant le système testé à une voiture :

- En méthode « boîte noire », on vérifie que la voiture fonctionne en allumant les lumières, en klaxonnant et en tournant la clé pour que le moteur s'allume. Si tout se passe comme prévu, la voiture fonctionne.
- En méthode « boîte blanche », on emmène la voiture chez le garagiste, qui regarde le moteur ainsi que toutes les autres parties (mécaniques comme électriques) de la voiture. Si elle est en bon état, elle fonctionne.

| Critère | Boîte noire | Boîte blanche |
|---|---|---|
| Accès au code | Non | Oui |
| Ce qu'on observe | Entrées / Sorties | Fonctionnement interne |
| Objectif principal | Vérifier le résultat final | Analyser la logique interne |
| Type de test | Test fonctionnel | Test structurel |
| Exemple typique | Utilisation d'une API sans lire son code | Débogage d'une fonction ligne par |
| Complexité d'analyse | Simple | Plus approfondie |

**Exercice 10.6**

On donne la fonction suivante, qui calcule la somme des carrés des $n$ premiers entiers :

```
def somme_carres(n):
    s = 0
    for i in range(1, n+1):
        s += i**2
    return s
```

On rappelle que la formule mathématique permettant de calculer directement cette somme est :

$$S_n = \frac{n(n+1)(2n+1)}{6}$$

1. **Test en boîte noire** : Tester la fonction avec $n = 3$, $n = 4$, $n = 5$. Comparer les résultats obtenus avec ceux donnés par la formule ci-dessus. Conclure sur la validité de la fonction.
2. **Analyse en boîte blanche** : Analyser le code pour comprendre son fonctionnement et proposer une optimisation.

### 10.2.3 Construire des jeux de tests

Un jeu de tests est un ensemble de tests destinés à valider un aspect d'un programme. Par exemple, un jeu de tests unitaires d'une fonction inclut l'ensemble des tests "boîte noire" et "boîte blanche" de cette fonction. Un jeu de tests est constitué d'un ou plusieurs programmes qui importent ou incluent la partie à tester et contiennent les différents tests.

**Exercice 10.7**
Une année bissextile est une année divisible par 4, mais pas par 100, sauf si elle est divisible par 1000.

1. Écrire une fonction qui indique si une année est bissextile, avec une seule conditionnelle
2. Écrire une fonction qui indique si une année est bissextile, avec une conditionnelle séparée pour chacun des cas
3. Écrire un jeu de tests qui couvrent tous les cas possibles et tester les deux fonctions

**Exercice 10.8**
On considère la fonction suivante :

```
def mystere(x):
    y = 0
    for z in x:
        if z > y:
            y = z
    return y
```

1. Que fait cette fonction ? La réécrire en donnant des noms expressifs aux variables
2. Écrire un test qui produit un résultat faux
3. Corriger la fonction pour qu'elle passe le test

Pour faciliter l'exécution d'un jeu de tests en Python, on peut utiliser `pytest` qui permet d'exécuter un ensemble de fonctions de test contenues dans un fichier.
Une fonction de test est une fonction dont le nom commence par `test_` ou termine par `_test`. Elle doit contenir une assertion qui est vraie si le test est correct. Dans le cas particulier où la fonction de test doit vérifier qu'une assertion du programme a échoué, comme dépiler une pile vide, on utilise la deuxième forme de test ci-dessous.

```
def test_valeur_de_retour():
    assert f(10) == 100        # test OK si l'assertion est vraie
def test_assertion():
    with pytest.raises(AssertionError):
        dépiler(creer_pile()) # code qui doit faire échouer une
     assertion
```

La commande `pytest m.py` exécute l'ensemble des fonctions de test du fichier m.py et indique celles qui ont réussi et celles qui ont provoqué une erreur. Il est de bonne pratique de fournir avec chaque module un module de test pour pytest.