

Spécification et tests

RÉSUMÉ

Écrire une fonction qui « marche » sur un exemple ne suffit pas. Il faut d'abord dire précisément ce qu'elle doit faire, c'est sa **spécification**, puis vérifier qu'elle le fait dans tous les cas prévus, ce sont ses **tests**. La spécification est un contrat entre celui qui écrit la fonction et celui qui l'utilise : elle fixe ce que l'appelant doit garantir (les préconditions) et ce que la fonction promet en retour (les postconditions). Les tests ne prouvent jamais qu'un programme est correct, mais ils révèlent les erreurs ; un bon jeu de tests vise les cas limites, là où les bugs se cachent.

Prérequis : Les fonctions, Structures conditionnelles.

① MÉMO

Spécifier une fonction

Prototype Le nom de la fonction, ses paramètres (avec le type attendu) et le type de la valeur renvoyée.

Docstring Chaîne placée juste sous la ligne de `f`, entre triples guillemets : rôle de la fonction, paramètres, valeur renvoyée, exemple. Elle s'affiche avec `help(f)`.

Préconditions Ce que l'appelant doit garantir avant l'appel (type, domaine : liste non vide, entier positif...). Si elles ne sont pas respectées, la fonction n'a rien à promettre.

Postconditions Ce que la fonction garantit après l'appel, lorsque les préconditions étaient respectées (une propriété du résultat).

L'instruction `assert`

`assert` condition, *"message affiché si la condition est fausse"*

- Si la condition est vraie, rien ne se passe. Si elle est fausse, Python arrête le programme avec une erreur `AssertionError` et affiche le message.
- Premier usage : vérifier une précondition au début d'une fonction.
- Second usage : écrire un test, en comparant le résultat obtenu au résultat attendu.
- Le message est facultatif mais précieux : il dit *ce qui* a échoué.

Construire un jeu de tests

Cas normaux Les situations courantes, avec un résultat calculé à la main.

Cas limites Les bords du domaine : liste vide ou à un seul élément, zéro, nombres négatifs, chaîne vide.

Cas particuliers Doublons, éléments tous égaux, liste déjà triée, majuscules et minuscules.

Règle Un test qui échoue prouve qu'il y a un bug. Un test qui réussit ne prouve pas que la fonction est correcte : il augmente seulement la confiance.

Organisation Regrouper les `assert` dans une fonction `test_nom()`, appelée après chaque modification du code.

Pièges fréquents

- Écrire `assert(cond, "msg")` avec des parenthèses : le couple est un tuple non vide, donc toujours vrai, et l'assertion ne détecte plus rien.
- Ne tester que les cas faciles et oublier la liste vide, le zéro ou les négatifs.
- Comparer des flottants avec `==` : `0.1 + 0.2 == 0.3` vaut `False`.
- Confondre `print` et `return` : une fonction qui affiche le bon résultat mais renvoie `None` échoue à tous les tests.
- Placer la docstring après le code de la fonction au lieu de la première ligne du corps.

Erreurs classiques

Code erroné	Code correct	Explication
<pre>assert(x > 0, "x positif attendu")</pre>	<pre>assert x > 0, "x positif attendu"</pre>	Avec les parenthèses, Python évalue le tuple <code>(x > 0, " ... ")</code> , qui est toujours vrai : l'assertion ne se déclenche jamais. Python signale d'ailleurs un <code>SyntaxWarning</code> .
<pre>def moyenne(L): return sum(L) / len(L) sans précondition</pre>	<pre>assert len(L) > 0, "liste vide" en première ligne du corps</pre>	Sur une liste vide, la division par zéro provoque un <code>ZeroDivisionError</code> peu explicite ; la précondition donne un message clair et documente le contrat.
<pre>assert aire(1) == 3.141592653589793</pre>	<pre>assert abs(aire(1) - 3.14159) < 1e-4</pre>	Les flottants sont des valeurs approchées : on compare à une tolérance, jamais avec l'égalité stricte.
<pre>def double(x): return 2 * x """Double x."""</pre>	Docstring en première ligne du corps, avant le code	Placée après le <code>return</code> , la chaîne n'est jamais atteinte et <code>help(double)</code> ne l'affiche pas.

② EXEMPLES

Programme 2 · Une fonction spécifiée et testée

```
1 def moyenne(L):
2     """Renvoie la moyenne des nombres de la liste L.
3
4     Précondition : L est une liste non vide de nombres.
5     Postcondition : le résultat est compris entre min(L) et max(L).
6     Exemple : moyenne([10, 14, 12]) renvoie 12.0
7     """
8     assert len(L) > 0, "moyenne : la liste ne doit pas être vide"
9     total = 0
10    for x in L:
11        total += x
12    return total / len(L)
13
14 print(moyenne([10, 14, 12]))    # 12.0
```

Programme 3 · Une précondition non respectée

```
1 moyenne([])
```

Python s'arrête et affiche la dernière ligne du message d'erreur :

```
AssertionError: moyenne : la liste ne doit pas être vide
```

Programme 5 · Un jeu de tests regroupé dans une fonction

```
1 def est_bissextile(annee):
2     """Renvoie True si annee est bissextile (calendrier grégorien)."""
3     return (annee % 4 == 0 and annee % 100 != 0) or annee % 400 == 0
4
5 def test_est_bissextile():
6     assert est_bissextile(2024) == True      # divisible par 4
7     assert est_bissextile(2023) == False     # cas normal
8     assert est_bissextile(1900) == False     # divisible par 100, pas par 400
9     assert est_bissextile(2000) == True      # divisible par 400
10    print("test_est_bissextile : tous les tests passent")
11
12 test_est_bissextile()
```

Un jeu de tests pour `maximum(L)` :

Famille	Entrée	Résultat attendu
cas normal	[3, 7, 2]	7
un seul élément	[5]	5
tous négatifs	[-3, -7, -2]	-2
doublons	[4, 4, 4]	4
maximum en première position	[9, 1, 2]	9
maximum en dernière position	[1, 2, 9]	9

Programme 6 · Un test qui révèle un bug

```
1 def maximum(L):
2     maxi = 0                # erreur : initialisation à 0
3     for x in L:
4         if x > maxi:
5             maxi = x
6     return maxi
7
8 assert maximum([3, 7, 2]) == 7      # passe : le bug reste invisible
9 assert maximum([-3, -7, -2]) == -2 # échoue : AssertionError
```

Le second test échoue, car `maximum([-3, -7, -2])` renvoie 0. Le cas limite « tous négatifs » révèle l'erreur d'initialisation; la correction est `maxi = L[0]`.

Programme 7 · Postcondition vérifiée dans la fonction

```
1 def racine_entiere(n):
2     """Renvoie le plus grand entier k tel que k * k <= n.
3
4     Précondition : n est un entier positif ou nul.
5     """
6     assert isinstance(n, int) and n >= 0, "n doit être un entier positif ou nul"
7     k = 0
```

```

8   while (k + 1) * (k + 1) <= n:
9       k += 1
10  assert k * k <= n < (k + 1) * (k + 1)    # postcondition
11  return k
12
13  print(racine_entiere(15), racine_entiere(16), racine_entiere(2026))    # 3 4 45

```

③ EXERCICES

Exercice 1 *Spécifier puis coder* On veut une fonction `division_euclidienne(a, b)` qui renvoie le couple (q, r) du quotient et du reste de la division de a par b , sans utiliser les opérateurs `//` et `%`.

1. Écrire la docstring : rôle, préconditions sur a et b , postcondition reliant a , b , q et r .
2. Coder la fonction par soustractions successives, avec une assertion pour la précondition et une pour la postcondition.
3. Écrire `test_division_euclidienne()` avec quatre tests, dont les cas $a = 0$, $a < b$ et a multiple de b .

Exercice 2 *Un jeu de tests qui révèle un bug* La fonction suivante doit compter les éléments **strictement** positifs d'une liste.

```

def nb_positifs(L):
    """Renvoie le nombre d'éléments strictement positifs de L."""
    n = 0
    for x in L:
        if x >= 0:
            n += 1
    return n

```

1. Proposer trois tests sous forme d'assertions, dont au moins un qui échoue.
2. Quel cas révèle le bug ? Pourquoi les autres tests ne le voient-ils pas ?
3. Corriger la fonction.

Exercice 3 *Tester une fonction sur les chaînes*

```

def nb_voyelles(mot):
    """Renvoie le nombre de voyelles (a, e, i, o, u, y) de mot, majuscules comprises."""
    compteur = 0
    for lettre in mot:
        if lettre.lower() in "aeiouy":
            compteur += 1
    return compteur

```

1. Écrire `test_nb_voyelles()` avec au moins quatre tests : un mot sans voyelle, la chaîne vide, un mot en majuscules, un mot formé uniquement de voyelles.
2. Un camarade a oublié `.lower()`. Lequel de tes tests échoue avec sa version ? Quelle valeur renvoie-t-elle ?

Exercice 4 *Le piège de l'assertion*

1. Expliquer pourquoi `assert(x >= 0, "x doit être positif")` ne signale jamais d'erreur, même pour `x = -5`. Corriger.
2. Écrire `chiffres(n)` qui renvoie la liste des chiffres de l'entier `n` (précondition : `n` entier positif ou nul), par exemple `chiffres(2026)` renvoie `[2, 0, 2, 6]`. Vérifier la precondition par une assertion, puis, avant le `return`, vérifier par une assertion la postcondition : les chiffres renvoyés recomposent bien `n`.
3. Tester avec 0, 7 et 2026.

SOLUTIONS DES EXERCICES

Corrigé de l'exercice 1.

```
1 def division_euclidienne(a, b):
2     """Renvoie (q, r), quotient et reste de la division euclidienne de a par b.
3
4     Préconditions : a et b sont des entiers, a >= 0 et b > 0.
5     Postcondition : a == b * q + r et 0 <= r < b.
6     """
7     assert isinstance(a, int) and isinstance(b, int), "a et b entiers attendus"
8     assert a >= 0 and b > 0, "il faut a >= 0 et b > 0"
9     q = 0
10    r = a
11    while r >= b:
12        r = r - b
13        q = q + 1
14    assert a == b * q + r and 0 <= r < b
15    return (q, r)
16
17 def test_division_euclidienne():
18     assert division_euclidienne(7, 3) == (2, 1)    # cas normal
19     assert division_euclidienne(0, 5) == (0, 0)    # a = 0
20     assert division_euclidienne(3, 5) == (0, 3)    # a < b
21     assert division_euclidienne(12, 4) == (3, 0)   # a multiple de b
22     print("test_division_euclidienne : tous les tests passent")
23
24 test_division_euclidienne()
```

Corrigé de l'exercice 2.

```
1 assert nb_positifs([1, 2, 3]) == 3    # passe
2 assert nb_positifs([-1, -2]) == 0     # passe
3 assert nb_positifs([0, 1, -3]) == 1   # échoue : renvoie 2
```

Cas révélateur : la présence de 0. Avec >=, le zéro est compté alors qu'il n'est pas strictement positif. Les deux premiers tests ne contiennent pas de zéro, donc > et >= donnent le même résultat : le bug est invisible.

Correction : remplacer `if x >= 0:` par `if x > 0:`. Le troisième test passe alors : `nb_positifs([0, 1, -3])` renvoie 1.

Corrigé de l'exercice 3.

```
1 def test_nb_voyelles():
2     assert nb_voyelles("rythm") == 1    # une seule voyelle, le y
3     assert nb_voyelles("xkcd") == 0    # sans voyelle
4     assert nb_voyelles("") == 0        # chaîne vide
5     assert nb_voyelles("PYTHON") == 2  # majuscules : Y et O
6     assert nb_voyelles("aeiouy") == 6  # que des voyelles
7     print("test_nb_voyelles : tous les tests passent")
8
9 test_nb_voyelles()
```

Sans `.lower()`, seul le test `nb_voyelles("PYTHON") == 2` échoue : la version fautive compare "P", "Y", ... à des lettres minuscules et renvoie 0. C'est le cas « majuscules » qui révèle l'oubli ; tous les autres tests passent.

Corrigé de l'exercice 4.

1. Les parenthèses créent le tuple ($x \geq 0$, "x doit être positif"). Un tuple non vide est toujours considéré comme vrai, donc l'assertion ne se déclenche jamais. Correction : `assert x >= 0, "x doit être positif"`, sans parenthèses.
- 2.

```
1  def chiffres(n):
2      """Renvoie la liste des chiffres de n, de gauche à droite.
3
4      Précondition : n est un entier positif ou nul.
5      Postcondition : les chiffres renvoyés recomposent n.
6      """
7      assert isinstance(n, int) and n >= 0, "n doit être un entier positif ou nul"
8      if n == 0:
9          return [0]
10     L = []
11     reste = n
12     while reste > 0:
13         L.append(reste % 10)
14         reste = reste // 10
15     L.reverse()
16     valeur = 0
17     for c in L:
18         valeur = valeur * 10 + c
19     assert valeur == n, "les chiffres ne recomposent pas n"
20     return L
```

3. `chiffres(0)` renvoie `[0]`, `chiffres(7)` renvoie `[7]`, `chiffres(2026)` renvoie `[2, 0, 2, 6]`. Le cas $n = 0$ est traité à part : sans lui, la boucle ne s'exécute pas et la fonction renverrait la liste vide.