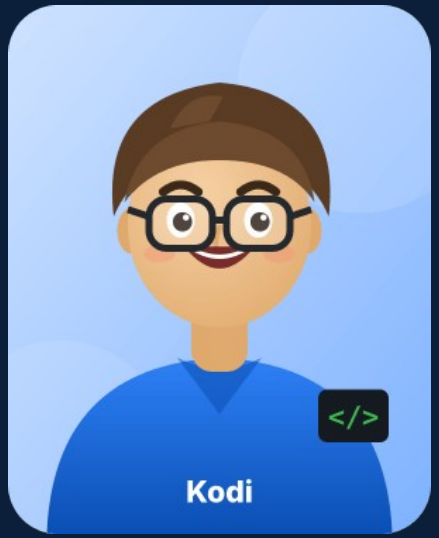
HEF-110 · MISSION 01 / 10

Pourquoi tester

Un test, c'est une preuve rejouable que ton code fait ce que tu crois. Sans preuve, tu n'as qu'une opinion.



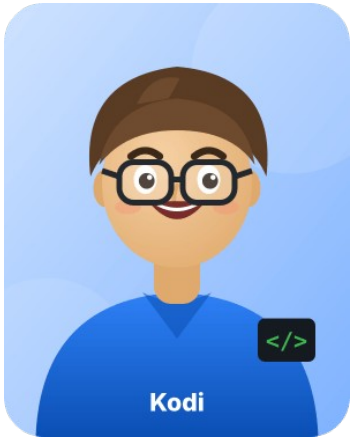


Badge : Filet de sécurité

80 min · intermédiaire

L'histoire

Vendredi 17 h. Kodi corrige un « tout petit » détail dans le calcul des remises de HashBox. Il relance l'application, clique sur deux pages, tout a l'air bon. Il déploie. Le lundi, Maya arrive : les clients qui ont deux cours dans leur panier paient le prix **plein**.



> *« Tu as vérifié à la main, dit Sam sans reproche. La question c'est : vérifié quoi ? et comment le refaire à l'identique dans six mois, ou pour la prochaine modification ? Un test, c'est la vérification que tu écris **une fois** et que l'ordinateur rejoue **mille fois**. »*

Dans cette mission, tu écris tes premiers tests, tu vois un test échouer pour de bonnes raisons, et tu comprends à quoi sert vraiment une suite de tests.

Ta mission

À la fin, je serai capable de…

1. expliquer à quoi servent les tests (confiance, régression, documentation, conception)
2. écrire un test automatisé avec `unittest` : préparer, agir, vérifier
3. lire un test rouge et en tirer la cause
4. distinguer les niveaux de tests (unitaire, intégration, E2E) et la pyramide
5. savoir qu'un test sans assertion, ou qui ne peut jamais échouer, est dangereux

Défi d'échauffement

Défi d'échauffement

Tu changes une pièce dans un moteur. Cite **trois façons** de vérifier que tout marche encore : essayer à la main, relire le schéma, ou lancer un banc d'essai qui mesure tout automatiquement. Laquelle peux-tu refaire à l'identique après chaque modification, sans oublier une vérification ? Laquelle devient impossible quand le moteur a 500 pièces ?

Le concept

À quoi servent les tests ?

Rôle	Explication
Détecter les bugs tôt	un bug trouvé par un test coûte quelques minutes ; trouvé en production, il coûte des heures, de l'argent, de la confiance
Protéger contre la régression	une *régression* : quelque chose qui marchait cesse de marcher après une modification
Permettre le changement	avec un filet de sécurité, on ose réorganiser le code (refactoring)
Documenter	un test lisible dit comment le code doit se comporter, avec des exemples exécutables
Guider la conception	du code difficile à tester est souvent du code mal découpé

Le concept

À quoi servent les tests ? (suite)

Un test **ne prouve pas** l'absence de bugs : il prouve que, pour les cas essayés, le comportement est celui attendu. Une suite de tests est un **filet** : plus les mailles sont serrées aux endroits risqués, plus il rattrape.

Anatomie d'un test : préparer, agir, vérifier

```
Arrange (préparer) → Act (agir) → Assert (vérifier)
```

1. **Préparer** : créer les données et l'état de départ.
2. **Agir** : appeler **une** chose (la fonction testée).
3. **Vérifier** : comparer le résultat à ce qui est attendu (*assertion*).

Un bon test est **rapide**, **indépendant** des autres, **déterministe** (même résultat à chaque fois), **lisible**, et il échoue pour **une** raison claire.

Le concept

unittest : la boîte à outils standard de Python

Élément	Rôle
`unittest.TestCase`	classe regroupant des tests ; chaque méthode `test_…` est un test
`self.assertEqual(a, b)`	vérifie `a == b`
`self.assertTrue / assertFalse`	vérifie une condition
`self.assertRaises(Erreur)`	vérifie qu'une exception est levée
`self.assertAlmostEqual(a, b)`	égalité de flottants avec tolérance
`unittest.main()`	lance tous les tests

Le concept

Rouge, vert

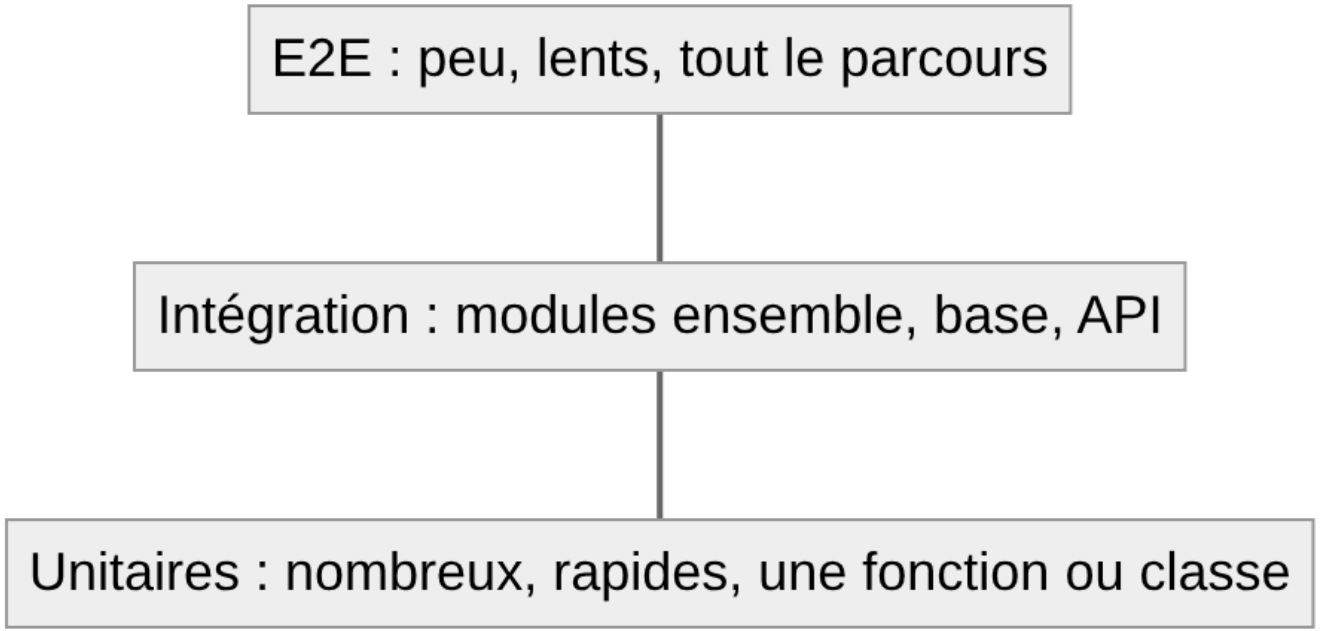
- **Rouge** : un test échoue. Il dit **ce qui est faux** (valeur attendue, valeur obtenue).
- **Vert** : tous les tests passent.

Un test qui n'a **jamais été rouge** n'a jamais prouvé qu'il savait détecter un bug : écris-le d'abord, ou casse volontairement le code pour le voir échouer.

Les niveaux de tests : la pyramide

Le concept

Les niveaux de tests : la pyramide (suite)



Le concept

Les niveaux de tests : la pyramide (suite)

Description du schéma : une pyramide à trois étages. En bas, de nombreux tests unitaires rapides ; au milieu, des tests d'intégration plus lents ; au sommet, peu de tests de bout en bout (E2E) qui parcourent toute l'application.

Niveau	Teste	Vitesse	Fiabilité du diagnostic
Unitaire	une fonction, une classe isolée	millisecondes	très précise
Intégration	plusieurs composants réels ensemble	secondes	moyenne
E2E	le système complet comme un utilisateur	secondes à minutes	large, fragile

On vise **beaucoup de tests rapides en bas, peu de tests lourds en haut**.

Le concept

Les faux amis

- **Test sans assertion** : il s'exécute mais ne vérifie rien (« il n'a pas planté »).
- **Test qui répète le code** : on recalcule la réponse avec la même formule que le code : si la formule est fausse, le test l'est aussi.
- **Test couplé à l'implémentation** : il casse à chaque réorganisation interne alors que le comportement n'a pas changé.
- **Test qui dépend de l'heure, du hasard, du réseau** : il passe ou échoue sans raison (mission 9).

🧠 ***Astuce*** *: lis un test comme une phrase : « étant donné un panier avec deux cours, quand on calcule le total, alors la remise de 10 % s'applique ».*

Exemple pas à pas

Problème : tester une fonction de moyenne, voir un test rouge, corriger, voir le vert, et vérifier que le test sait détecter un bug.

```
import io, unittest

def lancer(*classes):
    suite = unittest.TestSuite(unittest.defaultTestLoader.loadTestsFromTestCase(c)
    for c in classes)
    return unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)

# --- version 1 : naïve ---
def moyenne(notes):
    return sum(notes) / len(notes)

class TestMoyenne(unittest.TestCase):
    def test_cas_simple(self):
        self.assertEqual(moyenne([10, 20]), 15)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def test_une_seule_note(self):
        self.assertEqual(moyenne([7]), 7)

    def test_liste_vide_refusee_clairement(self):
        with self.assertRaises(ValueError):
            moyenne([])

resultat = lancer(TestMoyenne)
print("exécutés :", resultat.testsRun, "| échecs :", len(resultat.failures), "| er
   reurs :", len(resultat.errors))
assert resultat.testsRun == 3 and len(resultat.errors) == 1       # le test de la
   liste vide est ROUGE : ZeroDivisionError

# --- version 2 : corrigée ---
def moyenne(notes):
    if not notes:
        raise ValueError("impossible de faire la moyenne d'une liste vide")
    return sum(notes) / len(notes)
```

Exemple pas à pas (suite)

```
resultat = lancer(TestMoyenne)
print("après correction :", "VERT" if resultat.wasSuccessful() else "ROUGE")
assert resultat.wasSuccessful()

# --- un test sait-il détecter un bug ? On casse le code volontairement ---
def moyenne(notes):                        # bug : on divise par le mauvais nombre
    if not notes:
        raise ValueError("liste vide")
    return sum(notes) / (len(notes) + 1)

resultat = lancer(TestMoyenne)
print("avec un bug injecté :", len(resultat.failures), "test(s) en échec")
assert len(resultat.failures) == 2                                    # les deux tes
   ts de valeurs détectent le bug
```

Lecture :

Exemple pas à pas (suite)

1. Avec la version naïve, `test_liste_vide_refusee_clairement` est **rouge** : le code lève `ZeroDivisionError` au lieu d'une erreur claire. Le test a trouvé un défaut de conception.
2. Après correction, tout est **vert**.
3. En **injectant un bug**, deux tests deviennent rouges : c'est la preuve que la suite sait détecter cette sorte d'erreur. Cette technique (casser volontairement le code pour vérifier les tests) s'appelle le **test de mutation**, mission 9.

Chaque test suit *préparer, agir, vérifier* ; ici la préparation est triviale.

À toi, avec aide

Écris la fonction `est_bissextile(annee)` (règle : divisible par 4, sauf les siècles, sauf ceux divisibles par 400), **mais écris les tests d'abord** :

1. 1996 et 2024 sont bissextiles ;
2. 1900 et 2100 ne le sont pas ;
3. 2000 l'est ;
4. 2023 ne l'est pas ;
5. une année négative ou zéro lève `ValueError`.

Lance les tests **avant** d'écrire la fonction (ils doivent échouer), puis implémente jusqu'au vert.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Écris une fonction vide qui renvoie `None` pour voir les tests rouges.

Indice 2

`annee % 4 == 0 and (annee % 100 != 0 or annee % 400 == 0)`.

Indice 3

Vérifie l'année avant le calcul.

À toi, avec aide · réponse

Réponse

```
import io, unittest

def lancer(*classes):
    suite = unittest.TestSuite(unittest.defaultTestLoader.loadTestsFromTestCase(c)
    for c in classes)
    return unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)

class TestBissextile(unittest.TestCase):
    def test_multiples_de_4(self):
        for annee in (1996, 2024):
            self.assertTrue(est_bissextile(annee), annee)

    def test_siecles_non_bissextiles(self):
        for annee in (1900, 2100):
            self.assertFalse(est_bissextile(annee), annee)

    def test_siecle_divisible_par_400(self):
        self.assertTrue(est_bissextile(2000))
```

À toi, avec aide · réponse

Réponse (suite)

```
    def test_annee_ordinaire(self):
        self.assertFalse(est_bissextile(2023))

    def test_annee_invalide(self):
        for annee in (0, -4):
            with self.assertRaises(ValueError):
                est_bissextile(annee)

def est_bissextile(annee):          # version vide : tout doit être rouge
    return None

rouge = lancer(TestBissextile)
assert not rouge.wasSuccessful()

def est_bissextile(annee):
    if annee <= 0:
        raise ValueError("l'année doit être positive")
    return annee % 4 == 0 and (annee % 100 != 0 or annee % 400 == 0)
```

Réponse (suite)

```
vert = lancer(TestBissextile)
assert vert.wasSuccessful(), vert.failures + vert.errors
print("rouge puis vert :", rouge.testsRun, "tests")
```

Les tests écrits d'abord **documentent la règle** : les cas 1900, 2000 et 2100 sont ceux que la mémoire oublie.

À toi, en solo

Construis ton propre mini-framework de tests pour comprendre ce que fait `unittest` en coulisses :

1. un décorateur `@test` qui enregistre une fonction dans un registre ;
2. une fonction `verifier(condition, message="")` qui lève `EchecAssertion` ;
3. un lanceur `executer()` qui exécute chaque test, capture `EchecAssertion` (échec) et toute autre exception (erreur), mesure la durée, et renvoie un rapport (`réussis`, `échecs`, `erreurs`, détails) ;
4. teste ton framework avec 5 tests dont un échec et une erreur volontaires.

À toi, en solo · réponse

Pistes de correction

```
import time

REGISTRE = []

class EchecAssertion(Exception):
    pass

def test(fonction):
    REGISTRE.append(fonction)
    return fonction

def verifier(condition, message=""):
    if not condition:
        raise EchecAssertion(message or "condition fausse")

def executer(registre):
    rapport = {"reussis": [], "echecs": [], "erreurs": []}
    for fonction in registre:
        debut = time.perf_counter()
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        try:
            fonction()
            rapport["reussis"].append(fonction.__name__)
        except EchecAssertion as e:
            rapport["echecs"].append((fonction.__name__, str(e)))
        except Exception as e:
            rapport["erreurs"].append((fonction.__name__, f"{type(e).__name__}: {e
   }"))
    return rapport

@test
def addition_simple():
    verifier(1 + 1 == 2)

@test
def liste_vide_est_fausse():
    verifier(not [], "une liste vide est fausse")

@test
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def echec_volontaire():
    verifier(2 + 2 == 5, "2 + 2 devrait faire 4")

@test
def erreur_volontaire():
    {}["absent"]

@test
def chaine():
    verifier("ab".upper() == "AB")

rapport = executer(REGISTRE)
print(len(rapport["reussis"]), "réussis,", len(rapport["echecs"]), "échec,", len(r
   apport["erreurs"]), "erreur")
assert len(rapport["reussis"]) == 3 and rapport["echecs"][0][0] == "echec_volontai
   re" and rapport["erreurs"][0][1].startswith("KeyError")
```

À toi, en solo · réponse

Pistes de correction (suite)

Distinguer **échec** (une assertion fausse : le code se comporte autrement que prévu) et **erreur** (le test ou le code a planté de façon inattendue) accélère le diagnostic.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Tester « à la main » seulement	Vérification non répétable, oublis	Un test automatique pour chaque comportement important
Test sans assertion	Il passe toujours : fausse confiance	Chaque test vérifie un résultat observable
Recopier la formule du code dans le test	Le test partage la même erreur	Utiliser des exemples calculés indépendamment, à la main
Considérer « tout vert » comme une preuve	Les cas non testés restent inconnus	Penser aux limites, erreurs et cas rares

Mini-boss

Mini-boss

Filet de sécurité pour la remise de HashBox. Voici la règle : 1 cours = prix plein ; 2 cours = −10 % sur le total ; 3 cours ou plus = −20 % ; un code promo valide ajoute −5 € (jamais en dessous de 0). Le prix de chaque cours est en centimes (entiers).

1. Écris `total_panier(prix_cours, code_promo=None)` **sans regarder la solution**, mais écris d'abord au moins 10 tests, y compris : panier vide, 1/2/3/10 cours, promo valide et invalide, promo plus grande que le total ;
2. lance les tests, corrige jusqu'au vert ;
3. **injecte 4 bugs différents** (mauvaise borne `>` au lieu de `>=`, oubli du plancher à 0, remise appliquée deux fois, arrondi) et vérifie que ta suite en détecte chacun ;
4. écris en 5 lignes ce que ta suite ne garantit **pas**.

Mini-boss · réponse

Pistes de correction

Un bon travail : une table de cas (panier, promo, attendu) calculés à la main — par exemple `([10000, 5000], None) → 13500`, `([10000, 5000, 3000], None) → 14400`, `([1000], "PROMO5") → 500`, `([300], "PROMO5") → 0` — et des tests nommés par comportement. Les 4 bugs injectés doivent chacun rendre **au moins un** test rouge ; si un bug survit, c'est qu'il manque un cas (typiquement la borne exacte à 3 cours ou le plancher à 0). Ce que la suite ne garantit pas : plusieurs codes promo cumulés, prix négatifs, très gros paniers, concurrence, format d'affichage.

Quiz éclair

Quiz éclair

1. Un test automatisé sert surtout à :

A. rejouer une vérification à l'identique après chaque modification
B. rendre le code plus rapide
C. remplacer la documentation du projet

2. Un test qui n'a jamais été rouge :

A. est forcément parfait
B. n'a pas prouvé qu'il sait détecter un bug
C. est inutile à exécuter

3. Dans la pyramide des tests, il y a le plus de :

A. tests E2E, complets

Quiz éclair

Quiz éclair (suite)

B. tests manuels

C. tests unitaires, rapides et précis

4. 🔁 *Révision HEF-100, mission 01* : Un énoncé du module HEF-100 : le principe du moindre privilège signifie :

A. donner uniquement les droits nécessaires

B. donner tous les droits aux administrateurs

C. supprimer les comptes

5. 🔁 *Révision HEF-100, mission 07* : Une injection est évitée en :

A. doublant la longueur du mot de passe

B. séparant la donnée de l'instruction

Quiz éclair

Quiz éclair (suite)

C. désactivant les journaux

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

1. Quel bug que tu as déjà vécu aurait été détecté par un test de trois lignes ?
2. Pourquoi vérifier « à la main » devient-il impossible quand le logiciel grossit ?

À retenir

À retenir

1. Test = preuve rejouable : préparer, agir, vérifier.
2. Rouge puis vert ; un test jamais rouge n'est pas fiable.
3. `unittest` : TestCase, assertions, `assertRaises`.
4. Pyramide : beaucoup d'unitaires, quelques intégrations, peu d'E2E.
5. Faux amis : sans assertion, copie de la formule, couplage, dépendance au hasard.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Filet de sécurité

Prochaine mission : Spécifications et comportements

