Parcours autonome de reconstruction des bases Python

Commence au premier exercice qui te résiste et avance à vitesse libre. Pour chaque micro-exercice : **je lis → je complète → je programme → je vérifie**. Lance les `assert` avant de passer au suivant.

🟢 réussi sans correction 🟠 réussi avec aide 🔴 encore bloqué

Règle d’or : n’ouvre qu’un indice à la fois. La solution complète sert à contrôler un vrai essai, jamais à le remplacer.

Socle A — Je comprends Python

expressions, variables, types, fonctions, paramètres, return · 8 micro-exercices

A1 JE LIS

⌂ Menu général

Priorités de calcul

> **À retenir** — Python applique les priorités usuelles. // donne le quotient entier, % le reste.

Je lis

Sans exécuter, donner la valeur et le type de chaque expression.

```
3 + 2 * 4
17 // 5
17 % 5
7 / 2
```

Je vérifie

```
Réponses attendues : 11 (int), 3 (int), 2 (int), 3.5 (float).
```

▼ Indice 1

> Commence par la multiplication.

▼ Indice 2

> Une division avec / produit un float.

▼ Solution complète

```
11 ; 3 ; 2 ; 3.5. Types : int, int, int, float.
```

A2 JE LIS

⌂ Menu général

Variables et affectation

> **À retenir** — Une affectation calcule la partie droite puis range le résultat dans le nom de gauche.

Je lis

Suivre les valeurs de a et b, puis prévoir le résultat.

```
a = 4
b = a + 3
a = 10
resultat = a + b
```

Je vérifie

```
assert resultat == 17
```

▼ Indice 1

> b garde la valeur 7.

▼ Indice 2

> Changer a ensuite ne recalcule pas b.

▼ Solution complète

```
a vaut 10, b vaut 7, resultat vaut 17.
```

A3 JE COMPLÈTE

⌂ Menu général

Compléter un calcul

> **À retenir** — Un nom de variable doit décrire la valeur qu'il contient.

Je complète

Compléter pour calculer le périmètre d'un rectangle.

```
largeur = 7
longueur = 12
perimetre = ...
```

Je vérifie

```
assert perimetre == 38
```

▼ Indice 1

> Le périmètre contient deux largeurs et deux longueurs.

▼ Indice 2

> Écris 2 * (...).

▼ Solution complète

```
perimetre = 2 * (largeur + longueur)
```

A4 JE LIS

⌂ Menu général

Types simples

> **À retenir** — int : entier ; float : décimal ; bool : vrai/faux ; str : texte.

Je lis

Associer chaque valeur à son type.

```
42
3.0
True
"42"
```

Je vérifie

```
assert type(42) is int
assert type(3.0) is float
assert type(True) is bool
assert type("42") is str
```

▼ Indice 1

> Les guillemets signalent du texte.

▼ Indice 2

> True est un booléen, pas une chaîne.

▼ Solution complète

```
int ; float ; bool ; str.
```

A5 JE COMPLÈTE

⌂ Menu général

Une fonction très courte

À retenir — def crée une fonction. Le paramètre reçoit la valeur donnée lors de l'appel.

Je complète

Compléter la fonction double.

```
def double(x):
    return ...
```

Je vérifie

```
assert double(4) == 8
assert double(-3) == -6
```

▼ Indice 1

Le résultat dépend de x.

▼ Indice 2

Multiplier x par 2.

▼ Solution complète

```
def double(x):
    return 2 * x
```

A6 JE COMPLÈTE

⌂ Menu général

Deux paramètres

À retenir — Les paramètres sont des variables locales créées à l'appel.

Je complète

Compléter somme.

```
def somme(a, b):
    ...
```

Je vérifie

```
assert somme(3, 8) == 11
assert somme(-2, 5) == 3
```

▼ Indice 1

La fonction doit rendre une valeur.

▼ Indice 2

La ligne commence par return.

▼ Solution complète

```
def somme(a, b):
    return a + b
```

A7 JE LIS

⌂ Menu général

return ou print ?

> **À retenir** — print affiche ; return fournit un résultat réutilisable et testable.

Je lis

Expliquer pourquoi le premier assert échoue, puis corriger la fonction.

```
def carre(x):
    print(x * x)

assert carre(5) == 25
```

Je vérifie

```
assert carre(0) == 0
```

▼ **Indice 1**

> Une fonction sans return renvoie None.

▼ **Indice 2**

> Remplace print par return.

▼ **Solution complète**

```
def carre(x):
    return x * x
```

A8 JE PROGRAMME

⌂ Menu général

Moyenne autonome

> **À retenir** — Une fonction peut combiner ses paramètres dans l'expression renvoyée.

Je programme

Écrire moyenne(a, b), sans regarder Héron.

```
def moyenne(a, b):
    # à toi
```

Je vérifie

```
assert moyenne(12, 16) == 14
assert moyenne(0, 8) == 4
```

▼ **Indice 1**

> Additionne d'abord les deux valeurs.

▼ **Indice 2**

> Puis divise la somme par 2.

▼ **Solution complète**

```
def moyenne(a, b):
    return (a + b) / 2
```

Socle B — Je prends une décision

booléens, comparaisons, if, modulo, conditions · 8 micro-exercices

B1 JE LIS

⌂ Menu général

Lire des booléens

À retenir — Une comparaison produit True ou False. == teste l'égalité ; = affecte.

Je lis

Prévoir les quatre résultats.

```
7 > 3
4 == 5
6 != 2
3 <= 3
```

Je vérifie

```
True ; False ; True ; True
```

▼ Indice 1

Lis chaque symbole comme une phrase.

▼ Indice 2

<= signifie inférieur ou égal.

▼ Solution complète

```
True ; False ; True ; True.
```

B2 JE COMPLÈTE

⌂ Menu général

Décider avec if

> **À retenir** — Le bloc indenté après if ne s'exécute que si la condition est vraie.

Je complète

Compléter pour renvoyer "positif" ou "negatif" (0 est positif ici).

```
def signe(x):
    if ...:
        return "positif"
    else:
        return ...
```

Je vérifie

```
assert signe(-4) == "negatif"
assert signe(0) == "positif"
```

▼ **Indice 1**

> Compare x à 0.

▼ **Indice 2**

> Dans else, renvoie la chaîne manquante.

▼ **Solution complète**

```
def signe(x):
    if x >= 0:
        return "positif"
    else:
        return "negatif"
```

B3 JE COMPLÈTE

⌂ Menu général

Maximum de deux valeurs

> **À retenir** — Un if/else choisit exactement une des deux branches.

Je complète

Compléter maximum.

```
def maximum(a, b):
    if a >= b:
        return ...
    else:
        return ...
```

Je vérifie

```
assert maximum(8, 3) == 8
assert maximum(2, 9) == 9
assert maximum(5, 5) == 5
```

▼ Indice 1

> Dans la première branche, a est le plus grand.

▼ Indice 2

> Chaque branche renvoie son candidat.

▼ Solution complète

```
def maximum(a, b):
    if a >= b:
        return a
    return b
```

B4 JE PROGRAMME

⌂ Menu général

Minimum autonome

> **À retenir** — Les tests décrivent précisément le comportement attendu.

Je programme

Écrire minimum(a, b).

```
def minimum(a, b):
    # à toi
```

Je vérifie

```
assert minimum(8, 3) == 3
assert minimum(-2, 4) == -2
assert minimum(5, 5) == 5
```

▼ Indice 1

> Adapte la structure de maximum.

▼ Indice 2

> Teste a <= b.

▼ Solution complète

```
def minimum(a, b):
    if a <= b:
        return a
    return b
```

B5 JE LIS

⌂ Menu général

Le reste modulo

> **À retenir** — n % k est le reste de la division entière de n par k.

Je lis

Compléter mentalement les tests.

```
14 % 2
17 % 5
21 % 3
```

Je vérifie

```
assert 14 % 2 == 0
assert 17 % 5 == 2
assert 21 % 3 == 0
```

▼ Indice 1

> Cherche le plus grand multiple qui ne dépasse pas le nombre.

▼ Indice 2

> Un multiple exact donne un reste nul.

▼ Solution complète

```
0 ; 2 ; 0.
```

B6 JE COMPLÈTE

⌂ Menu général

Tester la parité

> **À retenir** — Un entier est pair si son reste modulo 2 vaut 0.

Je complète

Compléter est_pair pour renvoyer directement un booléen.

```
def est_pair(n):
    return ...
```

Je vérifie

```
assert est_pair(8)
assert not est_pair(7)
```

▼ Indice 1

> Utilise %.

▼ Indice 2

> Compare n % 2 à 0.

▼ Solution complète

```
def est_pair(n):
    return n % 2 == 0
```

Tester un multiple

> **À retenir** — Le même schéma permet de tester un multiple quelconque.

Je programme

Écrire est_multiple(n, k).

```
def est_multiple(n, k):
    # à toi
```

Je vérifie

```
assert est_multiple(15, 5)
assert not est_multiple(14, 5)
```

▼ Indice 1

> Le diviseur est maintenant k.

▼ Indice 2

> Le reste doit être égal à 0.

▼ Solution complète

```
def est_multiple(n, k):
    return n % k == 0
```

B8 JE COMPLÈTE

⌂ Menu général

Conditions composées

> **À retenir** — and exige deux conditions vraies ; or en exige au moins une ; not inverse.

Je complète

Renvoyer True si n est compris entre 10 et 99 inclus.

```
def deux_chiffres(n):
    return ... <= n ... n <= ...
```

Je vérifie

```
assert deux_chiffres(10)
assert deux_chiffres(99)
assert not deux_chiffres(100)
```

▼ **Indice 1**

> Deux bornes doivent être respectées.

▼ **Indice 2**

> Python accepte 10 <= n <= 99.

▼ **Solution complète**

```
def deux_chiffres(n):
    return 10 <= n <= 99
```

Socle C — Je répète

range, for, compteur, accumulateur, while · 9 micro-exercices

C1 JE LIS

⌂ Menu général

Lire range

> **À retenir** — range(fin) va de 0 à fin exclue. range(debut, fin, pas) exclut aussi fin.

Je lis

Écrire les valeurs produites.

```
list(range(5))
list(range(2, 7))
list(range(10, 3, -2))
```

Je vérifie

```
[0,1,2,3,4] ; [2,3,4,5,6] ; [10,8,6,4]
```

▼ Indice 1

> La borne finale n’est jamais incluse.

▼ Indice 2

> Avec un pas négatif, les valeurs diminuent.

▼ Solution complète

```
[0, 1, 2, 3, 4] ; [2, 3, 4, 5, 6] ; [10, 8, 6, 4].
```

C2 JE COMPLÈTE

⌂ Menu général

Boucle for

> **À retenir** — for répète le bloc une fois par valeur parcourue.

Je complète

Compléter pour afficher 1, 2, 3, 4, 5.

```
for i in range(..., ...):
    print(i)
```

Je vérifie

```
Affichage de 1 à 5 inclus.
```

▼ Indice 1

> Début 1.

▼ Indice 2

> La borne finale doit être 6.

▼ Solution complète

```
for i in range(1, 6):
    print(i)
```

C3 JE LIS

⌂ Menu général

Accumulateur

> **À retenir** — Un accumulateur part d'une valeur neutre puis est mis à jour à chaque tour.

Je lis

Suivre total après chaque tour et donner la valeur finale.

```
total = 0
for n in range(1, 5):
    total = total + n
```

Je vérifie

```
assert total == 10
```

▼ Indice 1

> Les valeurs sont 1, 2, 3, 4.

▼ Indice 2

> Totaux successifs : 1, 3, 6, 10.

▼ Solution complète

```
total vaut successivement 1, 3, 6 puis 10.
```

C4 JE COMPLÈTE

⌂ Menu général

Somme de 1 à n

> **À retenir** — Pour additionner, initialiser à 0 et ajouter chaque valeur.

Je complète

Compléter somme_n(n).

```
def somme_n(n):
    total = ...
    for i in range(1, ...):
        ...
    return total
```

Je vérifie

```
assert somme_n(1) == 1
assert somme_n(5) == 15
```

▼ Indice 1

> Pour inclure n, la borne finale est n + 1.

▼ Indice 2

> Mets total à jour avec i.

▼ Solution complète

```
def somme_n(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
```

C5 JE COMPLÈTE

⌂ Menu général

Compteur

> **À retenir** — Un compteur part de 0 et augmente de 1 quand une condition est vraie.

Je complète

Compter les multiples de 3 entre 1 et n inclus.

```
def compte_multiples_3(n):
    compteur = ...
    for i in range(...):
        if ...:
            ...
    return compteur
```

Je vérifie

```
assert compte_multiples_3(10) == 3
assert compte_multiples_3(2) == 0
```

▼ Indice 1

> Parcours 1 à n inclus.

▼ Indice 2

> Réutilise est_multiple ou i % 3 == 0.

▼ Solution complète

```
def compte_multiples_3(n):
    compteur = 0
    for i in range(1, n + 1):
        if i % 3 == 0:
            compteur += 1
    return compteur
```

C6 JE PROGRAMME

⌂ Menu général

Produit répété

> **À retenir** — Pour multiplier, la valeur neutre est 1.

Je programme

Écrire puissance(x, n) sans ** ni pow, pour n entier positif ou nul.

```
def puissance(x, n):
    # à toi
```

Je vérifie

```
assert puissance(2, 8) == 256
assert puissance(3, 0) == 1
```

▼ Indice 1

> Initialise le résultat à 1.

▼ Indice 2

> Répète n fois : résultat = résultat * x.

▼ Solution complète

```
def puissance(x, n):
    resultat = 1
    for _ in range(n):
        resultat *= x
    return resultat
```

C7 JE LIS

⌂ Menu général

Comprendre while

> **À retenir** — while répète tant que sa condition est vraie. La variable liée à la condition doit évoluer.

Je lis

Donner les valeurs affichées et la valeur finale de n.

```
n = 1
while n < 10:
    print(n)
    n = n * 2
```

Je vérifie

```
Affiche 1, 2, 4, 8 ; puis n vaut 16.
```

▼ **Indice 1**

> Double n à chaque tour.

▼ **Indice 2**

> Le tour suivant s'arrête quand n atteint ou dépasse 10.

▼ **Solution complète**

```
Affichage : 1, 2, 4, 8. Valeur finale : 16.
```

C8 JE COMPLÈTE

⌂ Menu général

Atteindre un seuil

> **À retenir** — Avant un while, initialiser l'état ; dans la boucle, garantir un progrès vers l'arrêt.

Je complète

Trouver le plus petit entier k tel que 2**k >= seuil.

```
def premier_exposant(seuil):
    valeur = 1
    k = 0
    while ...:
        valeur = ...
        k = ...
    return k
```

Je vérifie

```
assert premier_exposant(1) == 0
assert premier_exposant(9) == 4
```

▼ Indice 1

> Continue tant que valeur est trop petite.

▼ Indice 2

> Double valeur et augmente k de 1.

▼ Solution complète

```
def premier_exposant(seuil):
    valeur, k = 1, 0
    while valeur < seuil:
        valeur *= 2
        k += 1
    return k
```

C9 JE VÉRIFIE

⌂ Menu général

Boucle sûre

À retenir — Un invariant simple et un variant qui diminue aident à éviter les boucles infinies.

Je vérifie

Réparer la boucle pour qu'elle termine et renvoie 0.

```
def descendre(n):
    while n > 0:
        n = n + 1
    return n
```

Je vérifie

```
assert descendre(5) == 0
```

▼ Indice 1

n part positif : doit-il augmenter ?

▼ Indice 2

Remplace + 1 par - 1.

▼ Solution complète

```
def descendre(n):
    while n > 0:
        n -= 1
    return n
```

Socle D — Je manipule plusieurs données

listes, indices, parcours, append, recherche et comptage · 8 micro-exercices

D1 JE LIS

⌂ Menu général

Indices et longueur

> **À retenir** — Dans une liste de longueur n, les indices vont de 0 à n-1.

Je lis

Compléter les résultats.

```
notes = [12, 15, 9, 18]
notes[0]
notes[-1]
len(notes)
```

Je vérifie

```
12 ; 18 ; 4
```

▼ Indice 1

> Le premier indice est 0.

▼ Indice 2

> -1 désigne le dernier élément.

▼ Solution complète

```
12 ; 18 ; 4.
```

D2 JE COMPLÈTE

⌂ Menu général

Modifier une case

> **À retenir** — Une affectation avec un indice remplace un élément, sans changer la longueur.

Je complète

Remplacer 9 par 11.

```
notes = [12, 15, 9, 18]
notes[...] = ...
```

Je vérifie

```
assert notes == [12, 15, 11, 18]
```

▼ Indice 1

> 9 est en troisième position.

▼ Indice 2

> La troisième position a l'indice 2.

▼ Solution complète

```
notes[2] = 11
```

D3 JE COMPLÈTE

⌂ Menu général

Parcours par valeurs

> **À retenir** — for valeur in liste donne directement chaque élément.

Je complète

Calculer la somme des éléments.

```
def somme_liste(tab):
    total = 0
    for valeur in ...:
        ...
    return total
```

Je vérifie

```
assert somme_liste([3, 5, 2]) == 10
assert somme_liste([]) == 0
```

▼ Indice 1

> Parcours tab.

▼ Indice 2

> Ajoute valeur à total.

▼ Solution complète

```
def somme_liste(tab):
    total = 0
    for valeur in tab:
        total += valeur
    return total
```

D4 JE LIS

⌂ Menu général

Parcours par indices

> **À retenir** — range(len(tab)) produit tous les indices valides.

Je lis

Donner les couples affichés.

```
tab = ["a", "b", "c"]
for i in range(len(tab)):
    print(i, tab[i])
```

Je vérifie

```
(0,a), (1,b), (2,c)
```

▼ **Indice 1**

> len(tab) vaut 3.

▼ **Indice 2**

> range(3) produit 0, 1, 2.

▼ **Solution complète**

```
0 a ; 1 b ; 2 c.
```

D5 JE COMPLÈTE

⌂ Menu général

Construire avec append

> **À retenir** — append ajoute une valeur à la fin de la liste.

Je complète

Construire la liste des carrés de 0 à n inclus.

```
def carres(n):
    resultat = []
    for i in range(...):
        resultat.append(...)
    return resultat
```

Je vérifie

```
assert carres(3) == [0, 1, 4, 9]
```

▼ Indice 1

> Il faut inclure n.

▼ Indice 2

> Ajoute i * i.

▼ Solution complète

```
def carres(n):
    resultat = []
    for i in range(n + 1):
        resultat.append(i * i)
    return resultat
```

D6 JE COMPLÈTE

⌂ Menu général

Rechercher une valeur

> **À retenir** — On peut renvoyer dès qu'une valeur cherchée est trouvée.

Je complète

Écrire contient(tab, cible) sans utiliser in.

```
def contient(tab, cible):
    for valeur in tab:
        if ...:
            return ...
    return ...
```

Je vérifie

```
assert contient([4, 8, 2], 8)
assert not contient([4, 8, 2], 7)
```

▼ Indice 1

> Compare valeur et cible.

▼ Indice 2

> Après toute la boucle, rien n'a été trouvé.

▼ Solution complète

```
def contient(tab, cible):
    for valeur in tab:
        if valeur == cible:
            return True
    return False
```

D7 JE PROGRAMME

⌂ Menu général

Compter dans une liste

> **À retenir** — Recherche et comptage diffèrent : le comptage doit parcourir toute la liste.

Je programme

Écrire occurrences(tab, cible).

```
def occurrences(tab, cible):
    # à toi
```

Je vérifie

```
assert occurrences([2, 3, 2, 2], 2) == 3
assert occurrences([], 5) == 0
```

▼ Indice 1

> Initialise un compteur.

▼ Indice 2

> Augmente-le quand valeur == cible.

▼ Solution complète

```
def occurrences(tab, cible):
    compteur = 0
    for valeur in tab:
        if valeur == cible:
            compteur += 1
    return compteur
```

D8 JE PROGRAMME

⌂ Menu général

Filtrer une liste

> **À retenir** — Construire une nouvelle liste évite de modifier celle que l'on parcourt.

Je programme

Renvoyer uniquement les nombres pairs de tab.

```
def pairs(tab):
    # à toi
```

Je vérifie

```
assert pairs([1, 4, 6, 3]) == [4, 6]
assert pairs([]) == []
```

▼ Indice 1

> Crée resultat = [].

▼ Indice 2

> Pour chaque valeur paire, append.

▼ Solution complète

```
def pairs(tab):
    resultat = []
    for valeur in tab:
        if valeur % 2 == 0:
            resultat.append(valeur)
    return resultat
```

Socle E — Je construis un programme

fonctions auxiliaires, tests, composition et assemblage · 9 micro-exercices

E1 JE LIS

⌂ Menu général

Réutiliser une fonction

> **À retenir** — Une fonction testée peut devenir une brique fiable d’une autre fonction.

Je lis

Expliquer le résultat puis compléter le test.

```
def est_pair(n):
    return n % 2 == 0

def signe_pair(n):
    return n > 0 and est_pair(n)
```

Je vérifie

```
assert signe_pair(8)
assert not signe_pair(-8)
assert not signe_pair(7)
```

▼ Indice 1

> Les deux conditions reliées par and doivent être vraies.

▼ Indice 2

> -8 est pair mais pas positif.

▼ Solution complète

```
Les trois assertions passent.
```

E2 JE COMPLÈTE

⌂ Menu général

Fonction auxiliaire

À retenir — Décomposer évite de répéter un calcul et rend chaque morceau testable.

Je complète

Utiliser carre dans somme_carres.

```
def carre(x):
    return x * x

def somme_carres(a, b):
    return ...
```

Je vérifie

```
assert carre(4) == 16
assert somme_carres(3, 4) == 25
```

▼ Indice 1

Appelle carre deux fois.

▼ Indice 2

Additionne carre(a) et carre(b).

▼ Solution complète

```
def somme_carres(a, b):
    return carre(a) + carre(b)
```

E3 JE PROGRAMME

⌂ Menu général

Valeur absolue

> **À retenir** — Traiter séparément les cas aide à écrire une petite fonction fiable.

Je programme

Écrire valeur_absolue sans abs.

```
def valeur_absolue(x):
    # à toi
```

Je vérifie

```
assert valeur_absolue(3) == 3
assert valeur_absolue(-3) == 3
assert valeur_absolue(0) == 0
```

▼ Indice 1

> Si x est positif ou nul, renvoie x.

▼ Indice 2

> Sinon renvoie -x.

▼ Solution complète

```
def valeur_absolue(x):
    if x >= 0:
        return x
    return -x
```

E4 JE COMPLÈTE

⌂ Menu général

Comparer avec une tolérance

> **À retenir** — Pour des flottants, on teste souvent si une différence absolue est sous un seuil.

Je complète

Compléter proches(a, b, seuil) en réutilisant valeur_absolue.

```
def proches(a, b, seuil):
    return ...
```

Je vérifie

```
assert proches(2.0, 2.0005, 0.001)
assert not proches(2.0, 2.01, 0.001)
```

▼ Indice 1

> Calcule la différence a - b.

▼ Indice 2

> Teste valeur_absolue(a - b) < seuil.

▼ Solution complète

```
def proches(a, b, seuil):
    return valeur_absolue(a - b) < seuil
```

E5 JE COMPLÈTE

⌂ Menu général

Transformer puis tester

> **À retenir** — La composition suit le raisonnement : calcul intermédiaire, puis décision.

Je complète

Renvoyer True si le carré de essai est proche de x.

```
def carre(x):
    return x * x

def essai_correct(essai, x, seuil):
    return proches(..., ..., ...)
```

Je vérifie

```
assert essai_correct(1.9999, 4, 0.001)
assert not essai_correct(1.9, 4, 0.001)
```

▼ Indice 1

> Compare carre(essai) à x.

▼ Indice 2

> Transmets seuil sans le modifier.

▼ Solution complète

```
def essai_correct(essai, x, seuil):
    return proches(carre(essai), x, seuil)
```

E6 JE PROGRAMME

⌂ Menu général

Une étape d’amélioration

> **À retenir** — Nommer l’étape élémentaire simplifie ensuite la boucle principale.

Je programme

Écrire moyenne puis amelioration_essai(essai, x) : moyenne de essai et x/essai.

```
def moyenne(a, b):
    # ...

def amelioration_essai(essai, x):
    # ...
```

Je vérifie

```
assert amelioration_essai(1, 2) == 1.5
assert amelioration_essai(2, 1) == 1.25
```

▼ Indice 1

> Commence par moyenne.

▼ Indice 2

> Appelle moyenne(essai, x / essai).

▼ Solution complète

```
def moyenne(a, b):
    return (a + b) / 2

def amelioration_essai(essai, x):
    return moyenne(essai, x / essai)
```

E7 JE COMPLÈTE

⌂ Menu général

Assembler une boucle

À retenir — La boucle principale répète l'amélioration tant que le test d'arrêt est faux.

Je complète

Compléter test : pas de calcul détaillé dans la boucle, seulement les briques.

```
def test(essai, x, seuil):
    while not ...:
        essai = ...
    return essai
```

Je vérifie

```
assert proches(test(1, 9, 0.001), 3, 0.001)
```

▼ Indice 1

La condition d'arrêt est essai_correct.

▼ Indice 2

La mise à jour appelle amelioration_essai.

▼ Solution complète

```
def test(essai, x, seuil):
    while not essai_correct(essai, x, seuil):
        essai = amelioration_essai(essai, x)
    return essai
```

E8 JE VÉRIFIE

⌂ Menu général

Diagnostiquer par les tests

> **À retenir** — Tester les briques avant l'ensemble localise immédiatement une erreur.

Je vérifie

Classer les tests du plus local au plus global, puis dire lequel lancer en premier après une modification de moyenne.

```
assert proches(test(1, 9, .001), 3, .001)
assert amelioration_essai(1, 2) == 1.5
assert moyenne(12, 16) == 14
```

Je vérifie

```
Ordre : moyenne, amelioration_essai, test.
```

▼ Indice 1

> Commence par la fonction modifiée.

▼ Indice 2

> Puis teste ses utilisatrices, de proche en proche.

▼ Solution complète

```
moyenne → amelioration_essai → test.
```

E9 JE PROGRAMME

⌂ Menu général

Écrire le contrat

> **À retenir** — Avant de coder, préciser entrées, sortie et exemples transforme un problème flou en contrat.

Je programme

Pour une fonction compte_pairs(tab), écrire son contrat, trois assert (dont liste vide), puis le code.

```
def compte_pairs(tab):
    """..."""
    # à toi
```

Je vérifie

```
assert compte_pairs([]) == 0
assert compte_pairs([1, 3]) == 0
assert compte_pairs([1, 2, 4]) == 2
```

▼ Indice 1

> La sortie est un entier.

▼ Indice 2

> Réutilise est_pair dans une boucle avec compteur.

▼ Solution complète

```
def compte_pairs(tab):
    """Renvoie le nombre d'entiers pairs de tab."""
    compteur = 0
    for n in tab:
        if est_pair(n):
            compteur += 1
    return compteur
```

BOSS ASSEMBLAGE FINAL

⌂ Menu général

Boss final — Refaire Héron avec ses propres briques

> **Mission** — Tu sais maintenant lire une expression, écrire et tester une fonction, décider, accumuler, boucler et assembler. Le défi est de reconstruire la méthode sans recopier la correction d'origine.

1. **1. Cartographier** — Sur une feuille, relier : moyenne, valeur absolue, puissance, amélioration d'un essai, qualité d'un essai, boucle complète. Écrire pour chaque brique ses entrées et sa sortie.
2. **2. Reconstruire les briques** — Écrire moyenne(a,b), valeur_absolue(x) et puissance(x,n) sans ** ni pow. Lancer au moins deux assert par fonction.
3. **3. Préparer Héron** — Écrire amelioration_essai(essai,x) avec moyenne ; écrire est_suffisamment_bon(essai,x) pour |essai²-x| < 0.001 en réutilisant les briques.
4. **4. Assembler** — Écrire test(essai,x) avec while : améliorer tant que l'essai n'est pas suffisamment bon. Ne mettre dans la boucle que le test et la mise à jour.
5. **5. Simplifier et étendre** — Écrire racine_carree(x) qui démarre à 1, puis une fonction qui renvoie les racines des nombres pairs de 2 à 100.
6. **6. Valider** — Tester 4, 9 et 16 avec une tolérance plutôt qu'une égalité exacte. Expliquer en une phrase pourquoi cette tolérance est préférable pour les flottants.

▼ Indice 1

> Commence par les trois briques déjà travaillées séparément.

▼ Indice 2

> La condition du while est la négation du test de qualité.

▼ Indice 3

> Pour valider un résultat y attendu : valeur_absolue(racine_carree(x) - y) < 0.001.

▼ Solution de contrôle — à ouvrir après un vrai essai

⌂ Menu général

```
def moyenne(a, b): return (a + b) / 2
def valeur_absolue(x): return x if x >= 0 else -x
def puissance(x, n):
    r = 1
    for _ in range(n): r *= x
    return r
def amelioration_essai(essai, x): return moyenne(essai, x / essai)
def est_suffisamment_bon(essai, x):
    return valeur_absolue(puissance(essai, 2) - x) < 0.001
def test(essai, x):
    while not est_suffisamment_bon(essai, x):
        essai = amelioration_essai(essai, x)
    return essai
def racine_carree(x): return test(1, x)
def racines_paires(): return [racine_carree(x) for x in range(2, 101, 2)]
```