ALGORITHMIQUE · FICHE PRATIQUE

Cheatsheet : Tester et expliquer un petit programme

Cas limites, assertions, invariant, terminaison, coût et contrats des mini-projets.

Par Ali ZAINOUL · Cours gratuit

Vérifier un programme commence avant son exécution : annoncez ce qu'il accepte et calculez quelques résultats attendus à la main. Cette fiche montre des tests exécutables, un raisonnement d'arrêt et un petit calcul de commande complet.

Un résultat correct sur une entrée ne suffit pas à conclure pour toutes les données. Les tests, la justification du résultat, la terminaison et le comptage du travail répondent à des questions distinctes.

1. Construire des cas de test indépendants

Un cas de test associe une entrée précise à un résultat attendu. Une assertion vérifie une condition et signale un échec si elle est fausse. Les attentes doivent venir du contrat ou d'un calcul indépendant, pas être copiées du programme à vérifier.

Notre fonction reçoit une suite ordonnée de nombres réels finis. Elle compte les valeurs strictement supérieures à zéro, y compris chaque occurrence d'un doublon. Zéro ne compte pas ; une suite vide produit zéro. Ces règles définissent le problème avant son implémentation.

Attentes pour count_positive

Entrée	Résultat	Question couverte
[]	0	Aucun élément
[0]	0	Frontière de la positivité

Entrée	Résultat	Question couverte
[5]	1	Un élément retenu
[-2, 0, 3, 3]	2	Négatif, zéro et deux occurrences positives

2. Exécuter des vérifications explicites

Python fournit assert comme instruction ; MATLAB propose la fonction assert. Un échec interrompt ici le jeu de vérifications et signale que le résultat obtenu ne correspond pas à l'attente. Le message final ne s'affiche que si ces vérifications réussissent.

En MATLAB, la fonction s'enregistre dans count_positive.m et le script de tests dans check_count_positive.m, dans le même dossier. Les assertions Python peuvent être désactivées par certaines options d'exécution : elles ne doivent pas constituer l'unique validation de données externes.

Compter les valeurs positives et tester le résultat

Le parcours ajoute 1 au compteur seulement lorsqu'une valeur vérifie value > 0.

Pseudo-code

```
// example.pseudo
// count_positive
FUNCTION count_positive(values)
    count <- 0
    FOR EACH value IN values DO
        IF value > 0 THEN
            count <- count + 1
        END IF
    END FOR
    RETURN count
END FUNCTION
```

Python

```python
# count_positive.py
def count_positive(values):
    count = 0
    for value in values:
        if value > 0:
            count = count + 1
    return count

assert count_positive([]) == 0
assert count_positive([0]) == 0
assert count_positive([5]) == 1
assert count_positive([-2, 0, 3, 3]) == 2
print("All tests passed.")
```

`assert` compare le résultat à une attente. `return` fournit le compteur ; `print` observe la réussite des tests.

MATLAB

```
% count_positive.m
function count = count_positive(values)
    count = 0;
    for index = 1:numel(values)
        value = values(index);
        if value > 0
            count = count + 1;
        end
    end
end
```

`numel` fournit le nombre d'éléments. Pour une suite vide, le parcours 1:0 ne visite aucun élément.

MATLAB - appel

```
% check_count_positive.m
assert(count_positive([]) == 0);
assert(count_positive([0]) == 0);
assert(count_positive([5]) == 1);
assert(count_positive([-2, 0, 3, 3]) == 2);
disp('All tests passed.');
```

`assert` signale un échec de condition ; il ne corrige pas automatiquement la fonction.

> **Résultat expliqué.** La dernière attente vaut 2 : les deux occurrences de 3 comptent séparément. Le message All tests passed. confirme ces cas précis, pas toutes les entrées possibles.

3. Résultat, arrêt et coût : trois contrôles

Un invariant de boucle est une propriété vraie avant chaque nouveau test. Pour la somme de 1 à n, avant de traiter value, total contient la somme de 1 à value - 1. Initialement value vaut 1 et total vaut 0 : la somme sans terme est bien zéro. Ajouter value puis augmenter value de 1

conserve la propriété.

Une quantité de progression explique l'arrêt. Tant que value <= n, il reste n - value + 1 entiers à traiter. Ce nombre entier diminue de 1 par tour jusqu'à zéro. Pour n = 4, la sortie se fait avec value = 5 et l'invariant donne la somme complète 10.

Un coût n'a de sens qu'avec une convention. Si l'on compte seulement les additions au total, il y en a n. On ne compte pas ainsi les comparaisons, les mises à jour des variables ni une durée en secondes. Une mesure de temps dépendrait aussi du langage et de la machine.

Même boucle, différentes observations

Pour n = 4	Valeur	Interprétation
Résultat	10	1 + 2 + 3 + 4
Tours du corps	4	Une valeur ajoutée par tour
Tests value <= n	5	Le dernier test est faux
Additions au total	4	Opération choisie pour le coût
Valeur après sortie	5	Première valeur qui dépasse n

4. Mini-projet : une commande aux règles vérifiables

Le prix unitaire est un entier non négatif en centimes et la quantité un entier non négatif. Une commande vide coûte zéro. Sinon, les frais valent 100 centimes si le sous-total est strictement inférieur à 1 000 centimes, et zéro à partir de 1 000 centimes. Aucun taux de taxe ou remise n'est supposé.

Le seuil de 1 000 centimes et les frais de 100 sont des constantes de la règle choisie. La branche de retour anticipé traite le cas vide avant les frais. Une règle différente peut être valide, mais elle demanderait d'autres résultats attendus.

Calculer le total puis vérifier trois frontières

En MATLAB, enregistrez order_total.m puis use_order_total.m dans le même dossier.

Pseudo-code

```
// example.pseudo
// order_total
FUNCTION order_total(price, quantity)
    IF quantity == 0 THEN
        RETURN 0
    END IF
    subtotal <- price * quantity
    delivery <- 0
    IF subtotal < 1000 THEN
        delivery <- 100
    END IF
    RETURN subtotal + delivery
END FUNCTION
```

Python

```
# order_total.py
def order_total(price, quantity):
    if quantity == 0:
        return 0
    subtotal = price * quantity
    delivery = 0
    if subtotal < 1000:
        delivery = 100
    return subtotal + delivery

print(order_total(250, 3))
assert order_total(250, 0) == 0
assert order_total(250, 3) == 850
assert order_total(250, 4) == 1000
```

Le premier `return` traite la commande vide. Le dernier `return` fournit le total des autres commandes.

MATLAB

```
% order_total.m
function total = order_total(price, quantity)
    if quantity == 0
        total = 0;
        return;
    end
    subtotal = price * quantity;
    delivery = 0;
    if subtotal < 1000
        delivery = 100;
    end
    total = subtotal + delivery;
end
```

`return` quitte l'appel après avoir affecté total. Les autres chemins atteignent `end` avec leur résultat.

MATLAB - appel

```
% use_order_total.m
disp(order_total(250, 3));
assert(order_total(250, 0) == 0);
assert(order_total(250, 3) == 850);
assert(order_total(250, 4) == 1000);
```

Résultat expliqué. Trois articles donnent un sous-total de 750 puis un total de 850 centimes. Quatre articles donnent 1 000 centimes sans frais.

Erreur à éviter. Une entrée hors du contrat et une erreur de calcul ne sont pas la même situation. Ces petits exemples n'effectuent pas une validation complète de tous les types.

▼ Une modification remplace value > 0 par value >= 0 dans count_positive. Quel test existant révèle immédiatement le défaut ?

Le cas [0] doit renvoyer 0. La version modifiée renverrait 1 et ferait échouer cette assertion. Le test de frontière est donc utile même si plusieurs autres exemples réussissent.

Télécharger cette fiche en PDF · **Toutes les fiches par thème**

Version web : https://www.clearcode.fr/courses/programming/algorithmic/cheatsheet/practice