Thème 1

Spécification et mise au point

À quel point pouvons-nous faire confiance à un programme ? Cette question est importante dans tous les cas, voire cruciale s'il s'agit par exemple d'un programme intervenant dans le pilotage d'un train ou d'un avion. Après tout, un programme dit comment calculer un résultat, mais il ne dit pas ce qu'il calcule et il n'apporte pas non plus de garantie sur le résultat calculé. Un programmeur ne peut pas se contenter d'écrire uniquement du code. Il doit également expliquer ce que fait son programme et s'assurer qu'il se comporte convenablement. Ce chapitre apporte des éléments de bonne pratique pour augmenter la confiance que l'on peut avoir dans les programmes.

A. Que fait ce programme ?

Considérons le programme Python suivant.

```
def maximum_tableau(t):
    m = 0
    for i in range(len(t)):
        if t[i] > t[m]:
            m = i
    return m
```

Mais que fait ce programme ? En relisant le code, on comprend ce que fait chaque instruction, mais pas nécessairement ce que calcule le programme. Le nom de la fonction suggère qu'on calcule le maximum d'un tableau mais, avec un peu d'intuition, de pratique ou quelques tests, on parvient à déterminer que ce programme calcule en fait l'indice du maximum du tableau. On aurait gagné du temps si cette fonction avait reçu un nom plus explicite, comme `indice_maximum_tableau` et non pas `maximum_tableau`, ou si la variable `m` avait porté le nom plus explicite `ind_max`. Cependant, même avec un nom plus explicite pour la fonction, cette description reste un peu grossière. En particulier, cette description ne rend pas compte du comportement sur un tableau vide.

```
>>> t = []
>>> i = maximum_tableau(t)
>>> t[i]
IndexError: list index out of range
```

On aurait préféré savoir à l'avance que la fonction ne s'applique pas à un tableau vide, par exemple avec un nom encore plus explicite, comme `indice_maximum_tableau_non_vide`. Une autre solution serait que la fonction échoue avec un message explicite, indiquant qu'elle attend un tableau non vide, ou encore qu'elle gère le cas d'un tableau vide autrement qu'en renvoyant 0.

B. Documenter ses programmes

Supposons qu'on choisisse la première solution, c'est-à-dire une fonction qui ne s'applique qu'à un tableau non vide. On peut utiliser un commentaire pour l'indiquer.

```
# renvoie l'indice du maximum du tableau t, supposé non vide
def indice_maximum_tableau(t) :
```

Cela ne donne l'information qu'à un programmeur ayant accès au code. Si par exemple la fonction fait partie d'une bibliothèque, comme la fonction color de Turtle, il faut trouver une autre façon de transmettre 1'information au programmeur. Pour cela, Python permet d'associer une documentation à toute fonction, sous la forme d'une chaîne de caractères donnée entre triple guillemets.

```
def indice_maximum_tableau(t) :
""" Renvoie l'indice du maximum du tableau t.
    Le tableau t est supposé non vide. """
```

Cette chaîne, optionnelle, est placée au début de la fonction et sur plusieurs lignes. On peut avoir accès à cette documentation avec la commande `help` de Python.

```
>>> help(indice_maximum_tableau)
indice_maximum_tableau(t)
    Renvoie l indice du maximum du tableau t.
    Le tableau t est supposé non vide.
```

Comme on le voit, la chaîne de documentation est affichée à l'écran. Ici, cette documentation contient deux éléments. La première phrase décrit ce que la fonction renvoie (ici, l'indice du maximum du tableau). On appelle cela une **postcondition**. La seconde phrase décrit les conditions d'utilisation de la fonction (ici, le tableau doit être non vide). On appelle cela une **précondition**. Bien qu'on ait ajouté une documentation décrivant une précondition à l'utilisation de cette fonction, il ne s'agit toujours que d'un commentaire. Rien n'empêche le programmeur de continuer à appliquer par accident cette fonction à un tableau vide, ce qui est susceptible de provoquer une erreur plus loin dans le programme lorsqu'on utilisera le résultat de cette fonction.

C. Programmation défensive

Si on veut éviter qu'une utilisation de `indice_maximum_tableau` sur le tableau vide produise un résultat incohérent, qui risquerait de déclencher une erreur plus tard, on peut interrompre le programme dès lors que la fonction reçoit un tableau vide. On peut le faire très facilement en testant la taille du tableau à l'entrée de la fonction.

```
def indice_maximum_tableau(t):
""" Renvoie l'indice du maximum du tableau
    Le tableau t est supposé non vide. """
    if len(t) == 0:
        exit("indice_maximum_tableau : le tableau est vide")
```

La fonction `exit` de Python interrompt le programme, définitivement, après avoir affiché le message passé en argument. Pour savoir à quel endroit le programme a été interrompu, on a inclus le nom de la fonction dans le message. Une meilleure solution consiste à utiliser l'instruction `assert` de Python, qui combine le test d'une condition et 1'interruption du programme avec un message dans le cas où cette condition n'est pas vérifiée.

```
def indice_maximum_tableau(t):
""" Renvoie l'indice du maximum du tableau t.
    Le tableau t est supposé non vide. """
    assert len(t) > 0, "le tableau est vide"
```

Si la fonction est appelée sur un tableau vide, le test échoue et le programme est interrompu avec l'affichage du contexte dans lequel cet appel a eu lieu et du message qui accompagne le test.

```
>>> indice_maximum_tableau([])
Traceback (most recent call last):
  File "test.py", line 42, in <module>
  File "test.py", line 4, in indice_maximum_tableau
AssertionError: le tableau est vide
```

On appelle cela de la **programmation défensive**. Une autre façon d'être défensif consiste, plutôt que d'échouer, à renvoyer une valeur qui ne peut pas être confondue avec un résultat valide. Le bon choix pour cela consiste à utiliser la valeur `None`.

```
def indice_maximum_tableau(t):
""" Renvoie l'indice du maximum du tableau t,
    ou None s'il est vide """
    if len(t) == 0: return None
```

Comme on le voit, on a pris soin de modifier la chaîne de documentation de la fonction pour spécifier ce changement de comportement. Entre les deux stratégies défensives, utiliser `assert` d'une part et renvoyer `None` d'autre part, il n'y en a pas une qui soit meilleure que l'autre dans l'absolu. Selon le contexte d'utilisation, on pourra préférer l'une ou l'autre. Si on a choisi de renvoyer `None`, on peut appeler la fonction sans précaution particulière et tester ensuite la valeur renvoyée avant de l'utiliser.

```
r = indice_maximum_tableau(t)
if r != None:
    print("le maximum est", t[r])
```

Si on a choisi au contraire d'utiliser `assert`, il faut s'assurer que le tableau n'est pas vide avant d'appeler la fonction et on pourra ensuite utiliser le résultat sans précaution particulière.

```
if len(t) > 0:
    r = indice_maximum_tableau(t)
    print("le maximum est", t[r])
```

Enfin, si on est certain que le tableau n'est pas vide, ou si on accepte un échec du programme, alors les deux versions peuvent être utilisées et on peut toujours appeler la fonction sans précaution.

```
r = indice_maximum_tableau(t)
print("le maximum est", t[r])
```

D. Tester ses programmes

Même si on a correctement spécifié et documenté une fonction, il reste possible de faire une erreur en écrivant son code. Pour détecter ces éventuelles erreurs, on peut utiliser la fonction sur quelques cas concrets et vérifier qu'elle produit effectivement les résultats attendus. On appelle cela le test. Par exemple, on peut tester que la fonction `indice_maximum_tableau` de la section précédente renvoie bien 1 sur le tableau `[2, 3, 1]`. Pour cela, on peut appliquer la fonction à ce tableau et observer le résultat.

```
>>> indice_maximum_tableau([2, 3, 1])
1
```

Si le résultat effectivement renvoyé par la fonction n'est pas celui qui était attendu, alors le programme est manifestement erroné et il convient de trouver et corriger l'erreur. Si le résultat est bien celui attendu, comme ici, cela ne signifie pas nécessairement qu'il n'y pas d'erreur. Le programme peut contenir une erreur qui ne se révélerait que sur d'autres tableaux. Il faut donc effectuer d'autres tests. Par exemple, comme la spécification de `indice_maximum_tableau` mentionne le cas particulier du tableau vide, il est intéressant de tester la fonction sur ce cas.

```
>>> indice_maximum_tableau([])
None
```

De nombreuses autres situations méritent encore d'être testées, par exemple des cas particuliers où la valeur maximale est située au début ou à la fin du tableau, ou encore le cas d'un tableau ne contenant que des entiers négatifs.

Inclusion des tests

Plutôt que d'effectuer les tests dans la boucle interactive de Python, une meilleure solution consiste à les inclure dans le même fichier que le programme. En particulier, plutôt que de vérifier visuellement que chaque résultat obtenu est bien celui qui était attendu, on peut faire vérifier ceci par l'interprète Python, par exemple avec la construction `assert` et un test d'égalité.

```
assert indice_maximum_tableau([2, 3, 1]) == 1
assert indice_maximum_tableau([]) == None
assert indice_maximum_tableau([3, 1, 3, 7]) == 3
assert indice_maximum_tableau([8, 3, 1, 3, 7]) == 0
assert indice_maximum_tableau([-3, -1, -3, -7]) == 1
```

Si l'un de ces tests échoue, il faut rectifier le programme. Une fois l'erreur corrigée, il convient de relancer tous les tests, y compris ceux qui avaient déjà été effectués avec succès. En effet, en corrigeant une erreur, on peut en introduire une autre (c'est même assez courant). Il est donc intéressant d'avoir écrit tous les tests dans le fichier où est définie la fonction `indice_maximum_tableau`.

Remarques

- **Code de sortie d'un programme**
  Lorsqu'un programme termine, il signale au système d'exploitation s'il a terminé son exécution normalement, ou bien au contraire s'il s'est arrêté sur une erreur, à l'aide d'un code de sortie, qui est un entier. Lorsqu'un programme Python parvient sans erreur au terme de la séquence de ses instructions, il termine avec le code 0, qui désigne une exécution qui s'est correctement déroulée. De même, un appel à `exit()` termine le programme avec le code 0.
  Tout code de sortie non nul signale un problème et la valeur de l'entier peut être utilisée pour distinguer plusieurs raisons différentes pour l'échec du programme. Un appel à `exit("message")` ou une instruction `assert` qui évalue sa condition à `False` termine le programme avec le code de sortie 1. On peut utiliser `exit` avec un entier en argument, comme par exemple `exit(42)`, pour terminer le programme avec un code de sortie de son choix.
- **Spécification non déterministe**
  Pour un test donné, c'est la spécification qui indique le résultat attendu. On a une situation un peu plus délicate lorsque plusieurs réponses sont correctes. En particulier, que devrait renvoyer la fonction `indice_maximum_tableau` sur le tableau `[3, 1, 3]` où l'élément maximum apparaît deux fois ? La spécification n'indique pas que l'une des deux occurrences doit être préférée à l'autre. Ainsi, à moins d'être certain que le programme donnera toujours la préférence à la première occurrence, on ne peut écrire le test sous la forme

  ```
  assert indice_maximum_tableau([3, 1, 3]) == 0
  ```

  puisque la réponse 2 serait également acceptable. Dans ce cas nous ne pouvons que vérifier que le résultat obtenu est bien l'un des résultats admissibles. On peut le faire de la manière suivante.

  ```
  m = indice_maximum_tableau([3,1,3])
  assert m == 0 or m == 2
  ```

- **Quand définir ses tests ?**
  On peut définir de nombreux tests pour un programme donné sur la seule base de sa spécification. Il suffit donc d'avoir décidé ce que devait exactement faire une fonction pour commencer à écrire les tests associés. En particulier, il est tout à fait imaginable, et c'est même une pratique courante, d'écrire un certain nombre de tests pour une fonction avant même d'avoir écrit le code de cette fonction. Dans le cas d'un travail en équipe et une fois la spécification décidée en commun, on peut même confier la définition des tests et l'écriture du programme à deux personnes différentes. Cette pratique vise à éviter que le même oubli se glisse à la fois dans le programme et dans les tests et passe ainsi inaperçu.

Bons ensembles de test

En général on ne peut pas écrire un ensemble de tests exhaustif, qui suffirait à exclure toute erreur. Les entiers de Python, par exemple, sont en nombre infini, de même que les tableaux d'entiers, les chaînes, etc. Il est donc exclu de tous les tester individuellement. Le mieux qu'on puisse faire a priori est de trouver un « bon » ensemble de tests, qui soit suffisant pour donner une bonne confiance dans le programme testé.
Il est délicat de déterminer à quel moment un ensemble de tests est suffisant. L'objectif est que les tests réunis couvrent tous les « comportements » possibles du programme, ce qui est un concept assez vague. Voici une liste de points d'usage général à garder en tête et à compléter au cas par cas.

- Si la spécification du programme mentionne plusieurs cas, chacun de ces cas doit faire l'objet d'un test.
- Si une fonction renvoie une valeur booléenne, essayer d'avoir des tests impliquant chacun des deux résultats possibles.
- Si le programme s'applique à un tableau, il faut inclure un test couvrant le cas du tableau vide.
- Si le programme s'applique à un nombre, il peut être utile de tester pour ce nombre des valeurs positives, des valeurs négatives, ainsi que zéro.
- Si le programme fait intervenir un nombre appartenant à un certain intervalle, il peut être utile d'inclure des tests dans lesquels le nombre est aux limites de cet intervalle. En particulier, si le programme fait intervenir un indice d'un tableau, on peut inclure des tests dans lesquels cet indice sera 0 ou l'indice maximal.

Remarques

- **Tests et préconditions**
  Dans le cas d'une fonction dépendant d'une précondition, on n'effectue que des tests en accord avec cette précondition. En effet, lorsque les conditions d'utilisation d'une fonction ne sont pas remplies, la spécification ne dit rien de ce que doit être le résultat. On ne saurait alors déterminer un « résultat attendu » auquel comparer le résultat obtenu.
- **Critères de couverture**
  En plus des critères informels évoqués ici, on utilise également dans l'industrie quelques critères quantitatifs appelés critères de couverture, mesurant par exemple le pourcentage des instructions du programme qui sont effectivement exécutées lors d'au moins un des tests.

E. Corriger les erreurs

L'échec d'un test, c'est-à-dire l'observation d'une différence entre le résultat attendu et le résultat effectif d'un programme, voire une interruption inopinée du programme, atteste qu'une erreur est présente mais n'indique pas directement ce qu'est cette erreur ni à quel endroit du programme elle se trouve. Commence alors un travail d'enquête pour localiser, identifier et corriger l'erreur.

Cas pratique

Considérons pour l'exemple la fonction suivante, dont l'objectif est de tester si les éléments d'un tableau sont rangés par ordre croissant. Cette fonction analyse toutes les paires d'éléments consécutifs en commençant par la fin du tableau, et teste à chaque fois si le premier est bien inférieur ou égal au second.

```
def est_croissant(t):
    i = len(t) - 1
    while i >= 0:
        if t[i] <= t[i + 1]:
            return True
        else:
            return False
        i = i - 1
```

En application de nos critères informels de test, on choisit un test pour lequel le résultat devra être `True`, par exemple le tableau `[1, 2, 3, 4]`, un test pour lequel le résultat devra être `False`, par exemple le tableau `[1, 3, 2, 4]`, et on ajoute un test pour le tableau vide, pour lequel le résultat attendu est `True`. On observe dès le début des tests que `est_croissant([1, 2, 3, 4])` produit une erreur interrompant le programme avec l'affichage suivant.

```
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    assert est_croissant([1, 2, 3, 4]) == True
  File "test.py", line 4, in est_croissant if t [i] <= t [i + 1]:
IndexError: list index out of range
```

Visiblement l'un des accès `t[i]` ou `t[i + 1]` au tableau `t` est fait en dehors de ses indices valides. Pour en savoir plus on peut chercher à connaître la valeur de la variable `i` au moment où ce problème apparaît.
Une première approche consiste à reproduire le déroulement pas à pas de l'exécution du programme, en suivant les valeurs des variables à chaque étape. On peut le faire mentalement, sur papier, à l'aide d'outils offerts par certains IDE ou utiliser le module `pdb`. On peut ici rapidement se rendre compte que la variable `i` est initialisée avec la valeur 3 et que dès le premier tour de boucle on effectue la comparaison `t[3] <= t[4]`. L'indice 4 étant hors des limites du tableau l'erreur apparaît. Corrigeons donc le programme en remplaçant `t[i] <= t[i + 1]` par `t[i - 1] <= t[i]` et reprenons les tests. Notre fonction s'exécute maintenant sans erreur sur le tableau `[1, 2, 3, 4]` et renvoie `True` comme attendu. En revanche, elle répond également `True` si on l'applique au tableau `[1, 3, 2, 4]` : la fonction contient donc une autre erreur.
Pour trouver cette nouvelle erreur on peut reprendre le déroulement pas à pas de l'exécution de notre fonction, cette fois sur le tableau `[1, 3, 2, 4]`. Mais plutôt que de tout faire à la main, on peut s'aider de Python en ajoutant dans le programme que l'on cherche à corriger quelques instructions d'affichage qui nous permettront de suivre les valeurs prises par les variables à certains points cruciaux. Par exemple, on peut inclure au début de la boucle un affichage de la valeur de `i`.

```
def est_croissant(t):
    i = len(t) - 1
    while i >= 0:
        print("nouveau tour avec i =", i)
        if t[i - 1] <= t[i]:
            return True
        else:
            return False
        i = i - 1
```

Cette fonction appliquée à notre exemple `[1, 3, 2, 4]` affiche une unique ligne.

```
nouveau tour avec i = 3
```

Autrement dit, un seul tour de boucle a été effectué. Regardons le détail de ce tour de boucle : il commence avec la comparaison `t[2] <= t[3]`, autrement dit `2 <= 4` qui s'évalue à `True`. La première branche est alors sélectionnée, qui termine l'exécution de la fonction en renvoyant `True`. L'erreur est la présence de ce return : notre fonction s'arrête en renvoyant `True` dès qu'elle trouve deux éléments consécutifs placés dans le bon ordre, sans vérifier que les autres le sont également. Il faut donc modifier la fonction pour ne renvoyer `True` qu'après le parcours intégral du tableau.

```
def est_croissant(t):
    i = len (t) - 1
    while i >= 0:
        print("nouveau tour avec i =", i)
        if t[i - 1] > t[i]:
            return False
        i = i - 1
    return True
```

L'exécution sur le tableau `[1, 3, 2, 4]` affiche donc cette fois

```
nouveau tour avec i = 3
nouveau tour avec i = 2
```

avant de renvoyer `False` comme attendu. On peut constater de même que le test du tableau vide réussit, en renvoyant `True` sans effectuer aucun tour de boucle.
Cependant cette fois, le test de `est_croissant` appliqué au tableau `[1, 2, 3, 4]` échoue en renvoyant `False` au lieu de `True`. Si l'on a laissé l'instruction `print` dont on s'aide ici pour localiser les erreurs l'affichage est le suivant.

```
nouveau tour avec i = 3
nouveau tour avec i = 2
nouveau tour avec i = 1
nouveau tour avec i = 0
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    assert est_croissant([1, 2, 3, 4]) == True
AssertionError
```

On en déduit que la fonction renvoie `False` lors d'un tour de boucle effectué avec la valeur 0 pour la variable `i`. Lors de ce tour, la comparaison `t[i - 1] <= t[i]` est donc `t[-1] <= t [0]`, c'est-à-dire `4 <= 1` qui effectivement vaut `False` et implique l'exécution de la branche return `False`. L'erreur est donc ici un accès à l'indice -1 du tableau `t`. Dans la plupart des langages de programmation, cet accès aurait été une erreur interrompant l'exécution du programme. En Python, cependant, `t[-1]` désigne la dernière valeur du tableau et l'exécution du programme se poursuit donc comme si de rien n'était pour aboutir à un résultat erroné. On corrige donc encore le programme de la manière suivante pour empêcher ce dernier tour de boucle.

```
def est_croissant(t):
    """Renvoie True si les éléments de t sont rangés par
    ordre croissant."""
    i = len(t) - 1
    while i > 0:
        if t[i - 1] > t[i]:
            return False
        i = i - 1
    return True

assert est_croissant([1, 2. 3, 4]) == True
assert est_croissant([1, 3, 2, 4]) == False
assert est_croissant([]) == True
```

Dans cette version définitive on a retiré l'instruction d'affichage qui n'avait été ajoutée qu'à titre d'assistance à la localisation des erreurs. On laisse en revanche l'ensemble des tests, qui d'une part documentent l'effort de vérification qui a été fait, et de plus pourront encore jouer un rôle si la fonction `est_croissant` doit à nouveau être modifiée.

Recherche d'erreurs

Comme illustré dans le cas pratique, la recherche des erreurs commence souvent par une reproduction de l'exécution du programme sur un test qui a échoué. Une première idée sur la localisation peut être obtenue grâce à l'inclusion temporaire d'instructions d'affichage, qui peuvent jouer deux rôles :

- signaler quels blocs de code sont exécutés et combien de fois,
- donner les valeurs de certaines variables jugées importantes.

On peut ensuite affiner la recherche en suivant pas à pas l'exécution du programme (mentalement, sur papier, ou avec un outil d'assistance intégré à l'environnement de programmation) à proximité des moments où le comportement du programme n'est plus ce qu'il devrait être. La différence observée peut être par exemple :

- une mauvaise valeur pour une variable, dont il faut donc revoir le calcul,
- la sélection du mauvais bloc dans une instruction de branchement, dont la condition doit en ce cas être vérifiée,
- un tour de boucle manquant ou en trop, boucle dont les bornes ou la condition d'arrêt méritent alors réflexion.

S'ajoutent à ces problèmes les possibles interruptions du programme avant même la production d'un résultat, qui peuvent être analysées exactement de la même manière.

Une liste non exhaustive des « Erreurs » est donnée ci-dessous pour information.

Exception	Cause	Résolution
`IndentationError` ou `TabError`	Le fichier mélange des tabs et des espaces ou n'utilise pas le même nombre de tabs ou d'espaces partout.	Activez l'affichage des tabs et espaces dans votre éditeur de texte, et assurez-vous d'utiliser 4 espaces partout comme valeur d'indentation.
`AssertionError`	Une expression `assert` est fausse.	
`NameError`	Vous tentez d'utiliser un nom (de variable, fonction, classe, etc) qui n'existe pas.	Vérifiez que ce nom ne contient pas de faute (Python est sensible à la casse). Assurez-vous que ce que vous nommez a bien été créé avant cette ligne.
`IndexError`	Vous tentez d'accéder à une partie d'une indexable (souvent une liste ou un tuple) qui n'existe pas.	
`TypeError`	Vous tentez une opération incompatible avec ce type.	Si l'erreur a lieu au niveau d'une fonction que vous appelez, assurez-vous de passer des paramètres du type attendu par la fonction. Vous pouvez vérifier le type d'un objet avec `type()`. Vérifiez également que vous n'utilisez pas un opérateur incompatible avec un type ou entre deux types incompatibles.
`ValueError`	Vous passez une valeur à une fonction qui n'a aucun sens dans ce contexte ou dont le sens est ambiguë.	Assurez-vous de ne pas passer une valeur aberrante et que le résultat attendu soit évident. Par exemple, si vous essayez de faire `int("é")`, la conversion de la lettre « é » en entier n'a pas de résultat évident.
`ZeroDivisionError`	Vous faites une division par zéro.	
`IOError`	Erreur d'entrée / sortie	Vérifiez que vous pouvez lire / écrire depuis et vers la ressource que vous utilisez. Parmi les problèmes récurrents : disque dur plein, système corrompu, absence de permissions, fichier inexistant, etc.

F. Invariant de boucle

Comme on l'a expliqué précédemment, il convient de correctement documenter ses programmes. En particulier, il est important de ne pas perdre la trace des raisons pour lesquelles nos programmes fonctionnent correctement. S'il est une bonne pratique d'avoir toujours une feuille et un crayon à côté de son clavier quand on programme, il n'en reste pas moins vrai que ce qui est écrit sur cette feuille (petits dessins, calculs, idée principale, etc.) sera perdu s'il n'est pas joint au programme d'une façon ou d'une autre.
Quand les programmes contiennent des boucles, notamment, il est particulièrement important de se persuader de la logique sous-jacente du programme. Les variables du programme sont-elles correctement initialisées avant la boucle ? Le nombre de tours de boucle est-il le bon et, le cas échéant, l'indice de boucle est-il le bon ? Les valeurs obtenues au final sont-elles les bonnes ? Toutes ces questions peuvent être abordées avec la notion d'invariant de boucle. Il s'agit d'une propriété attachée à une boucle, qui est vraie initialement, avant de commencer à exécuter la boucle, et maintenue vraie par toute itération de la boucle, d'où son nom d'invariant. En particulier, elle sera donc vraie à la sortie de la boucle.
Le plus simple est de donner un exemple. Considérons la fonction suivante qui calcule le quotient et le reste de la division euclidienne de a par b par la méthode des soustractions successives.

```
def division_euclidienne(a, b):
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

On suppose $a > 0$ et $b > 0$ et on cherche à se persuader que cette fonction renvoie bien une paire d'entiers (q, r) telle que

$$a = b \times q + r$$
$$0 \leqslant r < b$$

ce qui est la définition d'une division euclidienne. Comme on le voit, le programme initialise r avec la valeur de a, puis lui retranche b tant que $r \geqslant b$. En particulier, on aura donc bien $r < b$ une fois sorti de la boucle. Mais il faut également montrer l'autre inégalité à savoir $0 \leqslant r$. Or, on est parti d'une valeur a supposée positive ou nulle, à laquelle on a ensuite retranché b uniquement lorsque qu'elle était supérieure ou égale à b. La valeur de r reste donc toujours positive ou nulle. On peut l'indiquer comme un invariant de boucle.

```
while r >= b:
    # invariant : 0 <= r
```

Il reste à établir l'autre propriété, à savoir $a = b \times q + r$. Ici, le principe de l'algorithme est que l'on a en permanence la propriété $a = b \times q + r$ qui est vraie. Initialement, on a $q = 0$ et $r = a$ et l'identité est donc trivialement vraie. Ensuite, chaque tour de boucle ajoute 1 à q et retranche b à r, ce qui maintient l'identité grâce à un peu d'arithmétique élémentaire.

$$\begin{aligned} a &= b \times q + r \\ &= b \times (q + 1) + (r - b) \end{aligned}$$

Au final, on peut avantageusement documenter notre programme avec ces deux invariants de boucle.

```
while r >= b:
    # invariant : 0 <= r
    # invariant : a = b * q + r
```

Si on a le moindre doute quant à ces invariants, ou si le programme contient une erreur que l'on est en train de chercher à identifier, on peut faire vérifier ces deux invariants de boucle par Python en les écrivant sous forme d'assertions plutôt que de commentaires.

```
while r >= b:
    assert 0 <= r
    assert a == b * q + r
```

Ainsi, ces deux invariants seront systématiquement vérifiés pendant les tests de notre fonction `division_euclidienne`. Une fois la mise au point effectuée, on peut revenir vers des commentaires uniquement pour rendre le programme plus efficace.
Un invariant de boucle peut également être attaché à une boucle `for`. Dans ce cas, la propriété invariante peut faire référence à l'indice de boucle. Prenons l'exemple très simple d'une fonction calculant la somme des entiers de 1 à n.

```
def somme_premiers_entiers(n):
    s = 0
    for i in range(1, n + 1):
        # invariant : s = 1 + 2 + ... + i-1
        s = s + i
    return s
```

Ici, l'invariant de boucle indique que la variable s contient la somme des entiers de 1 à $i-1$, où i est l'indice de boucle, qui prend toutes les valeurs de 1 à n. Initialement, on a $s = 0$, ce qui correspond bien à une somme ne contenant aucune valeur (car $i = 1$). Lorsqu'on effectue une itération de la boucle, quelle qu'elle soit, on ajoute i à la somme s, ce qui préserve bien l'invariant, car la variable i est alors incrémentée par la boucle `for`. À la sortie de la boucle, l'invariant est vérifié pour la valeur finale de i, c'est-à-dire $n+1$, autrement dit :

$$\begin{aligned} s &= 1 + 2 + \cdots + (n+1) - 1 \\ &= 1 + 2 + \cdots + n \end{aligned}$$

comme escompté. Il est important de noter que la toute dernière itération de la boucle s'est faite pour $i = n$. Mais l'invariant décrit une propriété vraie *avant* d'exécuter $s = s + i$ et d'incrémenter i. Le dernier tour de boucle nous donne donc la propriété pour $i = n + 1$.