Spécification et mise au point

Introduction

A quel point pouvons-nous faire confiance à un programme ? Cette question est très importante si il s'agit par exemple d'un programme de pilotage d'un avion. Un programme dit comment calculer un résultat, mais il ne dit rien de ce qu'il calcule, ni ne garantie le résultat calculé. Un programmeur ne peut donc se contenter d'écrire uniquement le code, il doit également expliquer ce que fait son programme et s'assurer qu'il le fait convenablement.

I Que fait ce programme ?

Prenons le programme Python ci-dessous que nous avions écrit lors du chapitre sur les tableaux.

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

En parcourant le code, on comprend ce que fait chaque instruction, mais pas nécessairement ce que calcule le programme. Le nom de la fonction suggère que l'on calcule le maximum d'un tableau, mais le programme détermine en faite *l'indice* maximum du tableau. On aurait alors rendu la fonction plus explicite en appelant la fonction **indice_maximum_tableau**, ou si la variable m s'était appelé **ind_max**. Même avec un nom plus explicite pour la fonction, 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 par à un tableau vide. Il serait alors intéressant, sans une nouvelle fois modifier le nom de la fonction, de préciser un message explicite lorsque la fonction échoue. Ce message préciserait que la fonction attend un tableau non vide.

II Documenter ses programmes

Python permet d'associer une *documentation* à toute fonction, sous 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 peut contenir 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.
```

La chaîne de documentation est affichée à l'écran. La première phrase décrit ce que la fonction renvoie (ici, l'indice du maximum du tableau). On appel 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*. Néanmoins,

le texte rajouté ne demeure qu'un commentaire, et rien n'empêche le programmeur de continuer à appliquer par accident cette fonction à un tableau vide.


Astuce

Du bon usage des commentaires. Un façon d'aider à la mise au point consiste à correctement documenter les programmes pendant leur construction, à l'aide de commentaires. Il faut cependant s'abstenir de commenter à outrance, au delà du nécessaire. Le plus osuvent, des noms de variables et de fonctions bien choisis remplacent avantageusement des commentaires. Ainsi, si deux variables représentent des coordonnées dans le plan, il est judicieux de les nommer **x** et **y** et il n'est alors pas nécessaire de préciser que la première est l'abscisse et la seconde l'ordonnée.

III 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 d'un tableau à l'entrée de la fonction.

```
def indice_maximum_tableau(t):
  """Renvoie l'indice du maximum du tableau t.
     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 meilleur solution consiste à utiliser l'instruction **assert** de Python, qui combine le test d'une condition à l'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 appe a eu lieux 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 le 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 si il est vide."""
  if len(t) == 0: return None
  ...
```

Comme on le voit, on a prit 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'absolue. 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 la 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 précautions.

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


Astuce

Renvoyer plutôt un indice non valide. On peut remarquer que le programme initial, qui renvoie 0 sur le tableau vide, renvoie un indice en dehors des bornes du tableau. Généralement, pour tout tableau qui manipule des indices d'un tableau **t**, on peut toujours renvoyer la valeur **len**(t) pour dénoter un indice invalide en cas d'échec. En python, on décourage cette pratique, préférant l'utilisation de la valeur **None** qui ne peut être confondue avec un indice.

IV 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** retourne effectivement 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 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 a 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 **ndice_maximum_tableau** mentionne le cas particulier du tableau vide, il est intéressant de tester la fonction sur ce cas.

```
>>> indice_maximum_tableau([2, 3, 1])
None
```

De nombreux autres situations méritent encore d'être testées, comme des cas particuliers où la valeur maximale est située au début ou à la fin du tableau, on 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 meilleur 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 cela 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 un fichier où est définie la fonction **indice_maximum_tableau**.


Astuce

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 textbfindice_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 comme ci-dessous.

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


Astuce

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 à 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 en 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 appartement 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.

Astuce

Tests et préconditions. Dans le cas d'une fonction dépendant d'une précondition, on 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 doit rien de ce que doit être le résultat. On ne saurait alors déterminer un "résultat attendu" auquel comparer le résultat obtenu.

Astuce

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.

V 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 par 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 (condensé d'histoires vraies)

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 -= 1
```

En application de nos critères informels de test, on chsoisit 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.

Astuce

Programme de test. Dans certains cas, plutôt que d'écrire chaque test sur une ligne individuelle, on peut écrire un programme effectuant une série de tests. Considérons par exemple une fonction **tableau_croissant** créant un tableau d'éléments aléatoires dont la taille **n** est donnée

en paramètre et dans lequel les éléments sont triés par ordre croissant. Vérifier que le résultat d'un test **tableau_croissant(5)** est correct revient essentiellement à vérifier que le tableau renvoyé contient ses éléments par ordre croissant. On peut donc d'une part définir une fonction chargée de vérifier si les éléments d'un tableau sont bien triés par ordre croissant, comme

```
def est_croissant(t):
  for i in range (len(t) - 1):
    if t[i] > t[i + 1]:
      return False
  return True
```

et d'autre par appliquer cette vérification à une série de tableaux générés par **tableau_croissant**. La boucle suivante par exemple teste dix tableaux de chacune des tailles 0 à 19.

```
for n in range(20):
  for _ in range(10):
    t = tableau_croissant(n)
    assert est_croissant(t)
```

Une telle boucle échouera au premier tableau rencontré qui ne soit pas croissant. On peut noter que, les tableaux de petite taille étant testés en premier, on favorise l'apparition des erreurs sur de petits exemples qui seront plus simples à analyser.

Astuce

Générer des tests aléatoires. Pour aller encore plus loin, on peut générer des tests aléatoirement. Considérons pour l'exemple une variante du problème des doublons vu dans les boucles avancées pour rechercher des doublons. On souhaite écrire une fonction doublon(t) prenant en paramètre un tableau t de taille **n** dont les éléments sont compris entre 1 et n-1 et renvoyant un élément qui apparaît au moins deux fois dans t. On peut utiliser Python pour réaliser un test de cette fonction en procédant en trois étapes. D'abord, *générer* un tableau **t** aléatoire vérifiant les conditions de taille et de valeur des éléments. Ensuite, *effectuer* le test en appelant la fonction sur **t** et en récupérant le résultat **r**. Enfin, énoncer un *verdict* sur le résultat, c'est-à-dire vérifier que **r** répond bien à la spécification de la fonction. Cette vérification confiée à une fonction auxiliaire que l'on appelle *oracle*, et qui e, l'occurrence vérifie que l'élément **r** apparaît bien plusieurs fois dans **t**.

```
def verifie_doublon(r, t):
  """Renvoie True si t contient au moins deux occurrences de r."""
  nb = 0
  for e in t:
    if e == r:
      nb +=1
  return nb > 1

for n in range(2, 20):
  for _ in range(10):
    t = tableau_aleatoire(n, 1, n-1)
    r = doublon(t)
    assert verifie_doublon(r, t)
```

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 faire mentalement ou sur papier à l'aide des méthodes de représentation de l'exécution. On peut ici se rendre compte que la variable i est initialisée avec la valeur 3 et que dès le premier tour de la 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, 2, 3, 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-ci 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 -=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 par la comparaison t[2] => t[3], autrement dit 2 <= 4, ce qui est **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 lors qu'elle trouve deux éléments consécutifs placés dans le bon ordre, sans vérifier que les autres le sont. 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 -=1
  return True
```

L'exécution du tableau [1, 3, 2, 4] affiche maintenant

```
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** sabs 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 retour de boucle effectué avec la valeur 0 pour la variable

i. Lors de ce tour, la comparaison t[i - 1] <= t[i] est donc précisément 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 -=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 ajouté 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 modifié.

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 différente ohbservée 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 être vérifiée,
- un tour de boucle manquant ou un en trop, boucle dont mes 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.

VI Invariant de boucle

Comme on l'a expliqué plus tôt, 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 (dessins, calculs, idées, 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 du programme. Les variables du programme sont-elles correctement initialisées avant la boucle ? Le nombre de tours de boucle est-il 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, 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 +=1
    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 :

$$a = q * b + r$$
$$0 <= r < b$$

ce qui est la différence d'une division euclidienne. Comme on le voit, le programme initialise **r** avec la valeur **a**, puis lui retranche **b** tant que **r >= b**. En particulier, on aura donc bien **x < b** une fois sorti de la boucle. Mais il faut également monter l'autre inégalité 0 <= r. Or, on est parti d'une valeur **a** supposé positive ou nulle, à laquelle on a ensuite retranché **b** uniquement lorsque elle était supérieur 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. Ici, le principe de l'algorithme est que l'on a *en permanence* la propriété

$$a = q * b + r$$

qui est vraie. Initialement, on a q = 0 et r = a et l'identité est donc vraie. Ensuite, chaque tour de boucle ajout la valeur 1 à **q** et retranche **b** à **r**, ce qui maintient l'identité grâce à un peu d'arithmétique élémentaire.

$$a = q * b + r$$
$$= (q + 1) * b + (r - b)$$

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

```
while r >= b:
  # invariant : 0 <= r
  # invariant : a = q * b + 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 la forme d'**assert** plutôt que de commentaires.

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

```python
def somme_premier_entiers(n):
  s = 0
  for i in range(1, n + 1):
    # invariant : s = 1 + 2 + ... + i-1
    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 incrémenté par la boucle **for**. A la sortie de la boucle, l'invariant est vérifié pour la valeur finale de **i**, c'est-à-dire n + 1, autrement dit

$$s = 1 + 2 + ... + (n + 1) - i$$

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 vrai *avant* d'exécuter s += i et d'incrémenter **i**. Le dernier tour de la boucle nous donne donc la propriété pour i = n + 1.

Savoir-faire

A retenir

Il convient de correctement **documenter** ses programmes, notamment en utilisant des **commentaires**, des **docstrings** et des **invariants de boucle**. Chaque fonction du programme doit avoir une **spécification**, qui peut être renseignée dans la **chaîne de documentation** de la fonction. Si cette spécification contient une **précondition**, c'est-à-dire une condition qui doit être vérifiée avant d'appeler la fonction, on peut adopter une **programmation défensive** où la condition est explicitement testée par la fonction ou par celui qui l'appelle. On peut notamment utiliser la construction **assert** pour cela. Pour mettre au point ses programmes, il faut les **tester** correctement. Lorsqu'une erreur est détectée, il faut l'identifier, rectifier le programme et relancer les tests.

Exercices de fin de chapitre

Exercice

Exercice 1

Donner un invariant de boucle pour la fonction suivante qui calcul x à la puissance n.

```python
def puissance(x, n):
  r = 1
  for i in range(n):
    r = r * x
  return r
```

Exercice

Exercice 2

Écrire des tests pour la fonction **puissance** de l'exercice précédent

Exercice

Exercice 3

Pour la fonction suivante, donner les meilleurs nom, une docstrings, un invariant de boucle et des tests.

```python
def f(t):
  s = 0
  for i in range(len(t)):
    s += t[i]
  return s
```

Exercice 4

On prétend que la fonction suivante teste l'appartenance de la valeur v au tableau t.

```python
def appartient(v, t):
  i = 0
  while i < len(t) - 1 and t[i] != v:
    i = i + 1
  return i < len(t)
```

Donner des tests pour cette fonction, et en particulier un test montrant qu'elle est incorrecte

Exercice 5

On prétend que la fonction suivante teste l'appartenance de la valeur v au tableau t.

```python
def appartient(v, t):
  for i in range(len(t)):
    if t[i] == v:
      trouvee = True
    else:
      trouvee = False
  return trouvee
```

Donner des tests pour cette fonction, et en particulier un test montrant plusieurs raisons pour laquelle cette fonction est incorrecte.

Correction des exercices

Correction de l'exercice 1

Avant d'exécuter le tour de boucle **i**, on a $r = x^i$. On vérifie en particulier que c'est bien vrai pour $i = 0$, car $r = 1 = x^0$. On peut donc écrire

```
for i in range(n)
  # invariant : r = x à la puissance i
```

On pourrait aussi utiliser une notation mathématique, comme $r = x^i$. A la toute dernière itération, pour $i = n - 1$, on a donc $r = x^{n-1}$ avant d'effectuer une dernière multiplication par x. On aura donc bien calculé x^n au final.

Correction de l'exercice 2

Comme cela a été expliqué, une bonne idée consiste à tester la fonction sur des valeurs "aux limites". Ici, on peut par exemple tester la fonction sur les valeurs 0 et 1 pour les variables x et n

```
assert puissance(21, 0) == 0
assert puissance(89, 1) == 89
assert puissance(1, 55) == 1
assert puissance(0, 34) == 0
```

Bien entendu, il faut également tester des valeurs plus grandes. On peut utiliser par exemple des puissances particulières dont on connaît bien le résultat.

```
assert puissance(2, 10) == 1024
assert puissance(10, 3) == 1000
```

Enfin, il faut tester des valeurs négatives pour x.

```
assert puissance(-1, 0) == 0
assert puissance(-1, 1) == -1
assert puissance(-2, 9) == -512
assert puissance(-2, 10) == 1024
```

Correction de l'exercice 3

Ctte fonction calcule clairement la somme des éléments du tableau t. L'invariant de boucle indique que s contient la somme des éléments jusqu'à t[i] exclu.

```
def somme(t):
  """Renvoie la somme des éléments du tableau t"""
  s = 0
  for i in range(len(t)):
    # invariant : s = t[0] + ... + t[i-1]
    s += t[i]
  return s
```

En ce qui concerne les tests, on peut faire des cas particuliers avec le tableau vide, un tableau à un élément, ainsi qu'avec des valeurs négatives.

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

Correction de l'exercice 4

La fonction renverra **False** à tort si **v** apparaît uniquement dans la dernière case de **t**. Voici une série de tests, dont le troisième met l'erreur en évidence.

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

Correction de l'exercice 5

Cette fonction renvoie **None** si elle est appliquée au tableau vide. De plus, le booléen qu'elle renvoie est systématiquement celui correspondant à la dernière comparaison faite : la fonction teste en réalité si **v** est le dernier élément du tableau. Voici une série de tests (dont les deux premiers détecteront un problème).

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