Documentation et mise au point de programmes

Introduction

Lorsqu'on *utilise* une fonction, il n'est pas utile de connaître ni même de comprendre le code qui la définit mais on a absolument besoin de connaître son rôle, autrement dit de savoir ce que *fait* cette fonction. Cela signifie que le programmeur ne peut donc pas se contenter d'écrire uniquement le code de la fonction mais il doit s'assurer d'**expliquer ce que fait son programme** afin que d'autres puissent l'utiliser.

De plus, pour écrire sa fonction, le programmeur doit s'assurer que **son programme se comporte convenablement**, autrement dit que la fonction renvoie la (ou les) valeur(s) attendue(s), et ce quelles que soient les valeurs (admissibles) des paramètres. C'est une étape importante voire cruciale s'il s'agit par exemple d'un programme intervenant dans le pilotage d'un avion.

Vous aller découvrir dans ce chapitre les bonnes pratiques qui permettent au programmeur d'effectuer ces deux tâches.

Que « fait » un programme ?

Considérons la fonction Python suivante qui a été codée par l'un de vos camarades et que vous devez utiliser dans votre projet.

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

En la recevant telle quelle, il n'est pas évident que vous sachiez à quoi elle sert. Après l'avoir analysée un petit peu ou après avoir effectué quelques tests (en appelant la fonction pour quelques valeurs des paramètres `x` et `n`) il est possible de comprendre ce que fait cette fonction.

> Alors, elle fait quoi selon vous ? :-)

Vous conviendrez qu'il aurait été nettement plus simple pour vous, que votre camarade codeur vous explique ce que fait sa fonction et comment l'utiliser, d'autant plus que vous n'avez pas forcément le temps d'essayer de la comprendre vous-même.

Si le codeur avait nommé sa fonction `puissance(x, n)` à la place de `f(x, n)` cela aurait pu vous mettre

plus rapidement sur la piste. Cependant, même avec ce nommage plus explicite, vous savez a priori que cette fonction a un rapport avec une "puissance", mais c'est tout (renvoie-t-elle x^n ? ou n^x ? ou autre chose ?..) Pour savoir ce que fait et renvoie la fonction, vous êtes toujours obligé de lire et comprendre le code de la fonction. Un nommage explicite est donc une première (bonne) étape mais elle n'est pas suffisante.

Documenter son programme

La bonne pratique pour un codeur est de toujours expliquer son programme (ici sa fonction). Pour cela, une première méthode peut consister à commenter son code de la façon suivante.

```
# fonction qui renvoie la valeur de x^n
def puissance(x, n):
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Cela donne l'information à celui qui a accès au code mais pas aux autres : si par exemple cette fonction appartient à une bibliothèque que l'on importe, nous n'avons pas directement accès au code de la fonction.

Ecrire son propre texte d'aide dans la *chaîne de documentation*

Il existe une meilleure façon d'expliquer son code. Pour cela, on associera une *documentation* à notre fonction sous la forme d'une chaîne de caractères écrites entre triples guillemets.

```
def puissance(x, n):
    '''
    Renvoie la valeur de x^n.
    '''
    r = 1
    for i in range(n):
        r = r * x
    return r
```

On appelle cela la **chaîne de documentation** de la fonction (ou *docstring* en anglais). En procédant ainsi, le programmeur de la fonction permet à quiconque d'afficher cette chaîne de caractères en utilisant la fonction `help`.

```
help(puissance)
```

```
Help on function puissance in module __main__:

puissance(x, n)
    Renvoie la valeur de x^n
```

Vous pouvez essayer d'afficher la docstring de fonctions connues.

```
help(abs)  # abs est la fonction valeur absolue
```

```
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Etre précis dans sa documentation

L'objectif d'une *chaîne de documentation* est d'être courte mais aussi précise. Avec celle de la fonction `puissance(x, n)` donnée au-dessus, on est tenté de calculer certaines puissances pour vérifier son bon fonctionnement.

```
puissance(2, 3) # renvoie bien la valeur de 2^3
```

```
8
```

```
puissance(0.5, 2) # renvoie bien la valeur de (1/2)^2 = 1/4
```

```
0.25
```

```
puissance(-2, 5) # renvoie bien la valeur de (-2)^5 = -32
```

```
-32
```

Cependant, en faisant d'autres tests on se rend vite compte que l'on ne peut pas calculer certaines puissances avec la fonction.

```
puissance(2, -3)  # ne renvoie pas la valeur de 2^(-3) = 1/8
```

```
1
```

```
puissance(4, 1/2) # ne renvoie pas la valeur de 4^(1/2) = 2 mais lève une erreur
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-11-de97611f6342> in <module>
----> 1 puissance(4, 1/2) # ne renvoie pas la valeur de 4^(1/2) = 2 mais lève une erreur


<ipython-input-3-4b5a05ad5df6> in puissance(x, n)
      2     '''Renvoie la valeur de x^n'''
      3     r = 1
----> 4     for i in range(n):
      5         r = r * x
      6     return r


TypeError: 'float' object cannot be interpreted as an integer
```

On voit sur ces deux tests que la fonction ne permet pas de calculer correctement une puissance si l'exposant est négatif ou si l'exposant n'est pas entier. La chaîne de documentation de la fonction n'était donc pas assez précise.

De manière générale, la chaîne de documentation d'une fonction doit contenir sa **spécification**, c'est-à-dire :

- son **rôle**
- la (les) valeur(s) qu'elle renvoie (= **postcondition**)
- les valeurs acceptées des paramètres (= **précondition**)

Ainsi, il aurait fallu remplacer

```
'''Renvoie la valeur de x^n'''
```

par

```
'''Renvoie la valeur de x^n, où n est un entier positif ou nul'''
```

pour savoir quelles sont les conditions (sur les paramètres) d'utilisation de la fonction.

> **À FAIRE :**
> Exercices 1 et 2

Programmation défensive

Si une fonction est bien documentée et qu'elle est correcte, on peut considérer que le travail du programmeur (de la fonction) a été fait correctement. Cependant, rien n'empêche d'utiliser cette fonction avec des paramètres d'entrée ne respectant pas les préconditions définies dans la docstring : mais l'erreur

incombe alors à l'utilisateur qui, soit n'a pas lu correctement la documentation, soit a volontairement testé la fonction avec des valeurs non admises.

Pour parer à cela, le programmeur a la possibilité d'utiliser la construction `assert` composée d'une condition à tester et d'un message à afficher si la condition est fausse.

Par exemple, le test invalide suivant produit l'affichage d'un message d'erreur.

```
# Cas d'un test invalide
a = -2
assert a >= 0, "le nombre n'est pas positif"
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-12-df727cdc315f> in <module>
      1 # Cas d'un test invalide
      2 a = -2
----> 3 assert a >= 0, "le nombre n'est pas positif"


AssertionError: le nombre n'est pas positif
```

Tandis que le test qui suit ne produit aucun affichage puisque la condition testée est valide.

```
# Cas d'un test valide
b = 3
assert b >= 0, "le nombre n'est pas positif"
```

On pourrait donc utiliser ce mécanisme d'assertion dans notre fonction `puissance(x, n)` de la façon suivante.

```
def puissance(x, n):
    '''
    Renvoie la valeur de x^n, où n est un entier positif ou nul
    '''
    assert type(n) == int and n >= 0, "n n'est pas un entier positif ou nul"
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Avant l'écriture du corps de la fonction, on a ajouté une assertion (ligne 5) qui va au préalable tester si le paramètre `n` est bien un entier et que c'est bien un entier positif. Si ce n'est pas le cas, la fonction va lever une erreur le message d'erreur sera affiché.

```
puissance(2, -3)
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-15-7153c55a6eb4> in <module>
----> 1 puissance(2, -3)


<ipython-input-14-600cf4b80289> in puissance(x, n)
      3     Renvoie la valeur de x^n, où n est un entier positif ou nul
      4     '''
----> 5     assert type(n) == int and n >= 0, "n n'est pas un entier positif ou nul"
      6     r = 1
      7     for i in range(n):


AssertionError: n n'est pas un entier positif ou nul
```

En écrivant cette assertion, le programmeur teste au préalable si les *préconditions sont valides* avant d'exécuter le reste du programme. On parle alors de *programmation défensive*.

> **Remarque :** il est possible de combiner plusieurs assertions. Par exemple, l'assertion précédente aurait pu être "séparée" en les deux assertions suivantes :
>
> ```
> assert type(n) == int, "n n'est pas un entier"
> assert n >= 0, "n n'est pas positif"
> ```

> **À FAIRE :** Exercices 3 et 4

Tester ses programmes pour (se) convaincre

On peut tout à fait avoir bien spécifié et documenté sa fonction sans que celle-ci ne fonctionne comme prévu. En effet, il n'est pas rare de se tromper dans le code. Pour repérer les éventuelles erreurs, le programmeur peut utiliser sa fonction sur des cas concrets et vérifier que celle-ci renvoie la (ou les) bonne(s) valeur(s). On appelle cela le *test*.

Inclusion de tests

Plutôt que de faire les tests manuellement un par un, il est possible d'utiliser la construction `assert` directement dans le fichier contenant le programme.

```python
def puissance(x, n):
    '''Renvoie la valeur de x^n, où n est un entier positif ou nul'''
    r = 1
    for i in range(n):
        r = r * x
    return r

assert puissance(2, 3) == 8
assert puissance(0, 2) == 0
assert puissance(5, 0) == 1
assert puissance(-2, 5) == -32
assert puissance(0.5, 2) == 0.25
```

Si l'un de ces tests échoue, un message indique le premier échec et la fonction doit être corrigée. Une fois que la correction a été faite, il faut relancer *tous* les tests. En effet, en corrigeant la fonction il est possible d'introduire une autre erreur (et donc qu'un des tests qui passait avec succès, échoue avec la correction apportée).

Ecrire ses tests avant le code de la fonction

Une pratique courante consiste à écrire des tests avant même d'écrire le code de la fonction. En effet, si la spécification de la fonction est claire, on sait quel doit être le comportement de celle-ci. Par exemple, supposons que l'on veuille écrire une fonction `indice_max_tab(T)` dont la spécification, écrite dans sa docstring, est la suivante :

```python
def indice_maxi_tab(T):
    '''
    Renvoie l'indice de la première occurence de la valeur maximale du tableau T. T est suppos
    '''
    # CODE A ECRIRE
```

On connaît son comportement et on peut tout suite écrire les tests suivants.

```python
def indice_maxi_tab(T):
    '''
    Renvoie l'indice de la première occurence de la valeur
    maximale du tableau T. T est supposé non vide.
    '''
    # CODE A ECRIRE

assert indice_maxi_tab([4, 3, 2, 1]) == 0
assert indice_maxi_tab([4, 5, 3, 2]) == 1
assert indice_maxi_tab([1, 2, 3, 6]) == 3
assert indice_maxi_tab([3, 2, 5, 2]) == 2
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-33-9d366810793a> in <module>
      5     # CODE A ECRIRE
      6
----> 7 assert indice_maxi_tab([4, 3, 2, 1]) == 0
      8 assert indice_maxi_tab([4, 5, 3, 2]) == 1
      9 assert indice_maxi_tab([1, 2, 3, 6]) == 3


AssertionError:
```

Ensuite, on peut écrire le code de la fonction et exécuter le programme. Si l'un des tests échoue on est certain d'avoir fait une erreur et il faut la corriger. Cependant, si tous nos tests *passent*, nous ne sommes pas sûr que notre fonction est bien écrite pour autant.

Un bon jeu de test

Il est souvent impossible d'écrire de manière exhaustive tous les tests possibles car il y en a bien souvent une infinité. Pour se convaincre que notre fonction est bien écrite, l'enjeu consiste donc à trouver un ensemble de tests qui couvrent les différents comportements du programme.

Par exemple, avec le code suivant pour la fonction `appartient(v, T)` aucun des tests proposés n'échoue.

```
def indice_maxi_tab(T) :
    '''
    Renvoie l'indice de la première occurence de la valeur
    maximale du tableau T. T est supposé non vide.
    '''
    valeur_max = T[0]
    indice = 0
    for i in range(len(T)) :
         if T[i] > valeur_max :
            indice = i
    return indice

assert indice_maxi_tab([4, 3, 2, 1]) == 0
assert indice_maxi_tab([4, 5, 3, 2]) == 1
assert indice_maxi_tab([1, 2, 3, 6]) == 3
assert indice_maxi_tab([3, 2, 5, 2]) == 2
assert indice_maxi_tab([1, 1, 1, 1]) == 0
assert indice_maxi_tab([3, 5, 5, -1]) == 2
```

On peut alors penser que notre fonction est correcte, d'autant plus que les tests choisis couvraient toutes les positions possibles du maximum. Cependant, voici un nouveau test qui échoue.

```
assert indice_maxi_tab([1, 5, 4, 3]) == 1
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-20-7d7de65c0ad6> in <module>
----> 1 assert indice_maxi_tab([1, 5, 4, 3]) == 1


AssertionError:
```

Ce test met en évidence que notre fonction est incorrecte et montre bien que la qualité du jeu de tests est importante.

Il n'est pas simple de définir ce qu'est un **bon "jeu de tests"** mais de manière générale, voici quelques règles que l'on peut appliquer :

- si la spécification mentionne plusieurs cas, il faut s'assurer de tous les tester ;
- si la fonction renvoie un booléen, il faut s'assurer de tester les deux résultats possibles ;
- si la fonction s'applique à un tableau, il faut tester le cas où le tableau est vide ;
- si la fonction doit parcourir un tableau en entier, il faut s'assurer que celui-ci est parcouru entièrement ;
- si la fonction s'applique à un nombre, il faut s'assurer de tester des cas où le nombre est positif, où le nombre négatif et où le nombre vaut zéro ;
- si la fonction s'applique à un nombre appartenant à un intervalle, il faut s'assurer de tester les cas où le nombre est égal aux bornes de l'intervalle.

> **À FAIRE :** Exercices 5, 6 et 7

Corriger sa fonction : afficher les valeurs de certaines variables

Il n'est pas toujours évident de trouver pourquoi notre code ne fonctionne pas. Les tests qui échouent nous donne cependant de précieux éléments sur les erreurs dans le programme. En effet, c'est souvent en partant d'un test qui échoue que l'on trouve les erreurs en suivant l'état des variables. Ceci peut se faire mentalement ou sur papier mais il également possible d'afficher la valeur de certaines variables à des endroits stratégiques du programme. Par exemple, on peut ajouter les instructions suivantes pour afficher l'état des variables à chaque tour de boucle de notre fonction `indice_maxi_tab(T)`.

```
def indice_maxi_tab(T) :
    '''
    Renvoie l'indice de la première occurence de la valeur
    maximale du tableau T. T est supposé non vide.
    '''
    valeur_max = T[0]
    indice = 0
    for i in range(len(T)) :
        print("tour de boucle :", i)  # on affiche le numéro de chaque tour de boucle
        if T[i] > valeur_max :
            indice = i
            print("indice maxi :", indice)
            print("valeur maxi :", valeur_max)             # on affiche l'indice du maximum
    return indice
```

On sait que l'instruction `indice_maxi_tab([1, 5, 4, 3])` doit normalement renvoyer la valeur 1 mais que ce n'est pas le cas d'après le dernier test. On peut tester cet appel maintenant que l'on a ajouté l'affichage de certaines variables.

```
indice_maxi_tab([1, 5, 4, 3])
```

```
tour de boucle : 0
tour de boucle : 1
indice maxi : 1
valeur maxi : 1
tour de boucle : 2
indice maxi : 2
valeur maxi : 1
tour de boucle : 3
indice maxi : 3
valeur maxi : 1




3
```

On se rend compte que la variable `indice` change de valeur aux 2ème, 3ème et 4ème tour de boucle. Autrement dit, les conditions `T[1] > valeur_max`, `T[2] > valeur_max` et `T[3] > valeur_max`. On comprend alors notre erreur en remarquant que la variable `valeur_max` n'est pas mise à jour et vaut `T[0]` (ici 1) tout au long du programme, ce qui explique que les 3 conditions précédentes sont vraies et que la valeur renvoyée par notre fonction est erronée.

On peut alors corriger notre programme en ajoutant une ligne pour mettre à jour la valeur maximale.

```python
def indice_maxi_tab(T) :
    '''
    Renvoie l'indice de la première occurence de la valeur
    maximale du tableau T. T est supposé non vide.
    '''
    valeur_max = T[0]
    indice = 0
    for i in range(len(T)) :
        if T[i] > valeur_max :
            valeur_max = T[i]   # mise à jour de la valeur maximale
            indice = i
    return indice
```

```python
indice_maxi_tab([1, 5, 4, 3])
```

```
1
```

Après cette modification, il faudrait normalement encore vérifier que tous les autres tests *passent* avec succès.

Intégration des tests à la chaîne de documentation

Choisir et vérifier des tests pertinents est un travail précieux, dont il est important de garder une trace. De plus, des tests bien choisis peuvent constituer une explication très efficace de l'effet d'une fonction.

Une pratique fréquente est d'inclure une série de tests directement dans la chaîne d'aide d'une fonction.

```
def puissance(x, n):
    '''
    Renvoie la valeur de x^n, où n est un entier positif ou nul

    >>> puissance(2, 3)
    8

    >>> puissance(0, 2)
    0

    >>> puissance(5, 0)
    1

    >>> puissance(-2, 5)
    -32

    >>> puissance(0.5, 2)
    0.25

    etc.

    '''
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Par convention, les tests prennent la forme d'expressions Python précédées de la syntaxe `>>>`, qui représente l'invite de commande de l'interpréteur Python classique. Chaque expression est suivi de l'affichage (éventuellement vide) qui serait provoqué si elle était évaluée dans l'interpréteur.

Extraction et vérification automatique de tests : le module *doctest*

Un outil prédéfini, accessible par le biais du module `doctest`, permet d'extraire automatiquement et de vérifier chacun des tests présents dans les chaînes de documentation, dans toutes les fonctions d'un module par exemple (par défaut, les tests du module courant sont extraits et vérifiés).

```
import doctest
```

```
doctest.testmod()
```

```
TestResults(failed=0, attempted=5)
```

Ici, il n'y a qu'une fonction qui contient des tests inclus dans sa chaîne de documentation. On se rend compte que les 5 tests pour la fonction `puissance(x, n)` sont passés avec succès.

Il est possible d'ajouter un paramètre optionnel à la fonction `testmod` afin d'obtenir plus d'informations sur

les tests effectués, même en cas de succès.

```
doctest.testmod(verbose = True)
```

```
Trying:
    puissance(2, 3)
Expecting:
    8
ok
Trying:
    puissance(0, 2)
Expecting:
    0
ok
Trying:
    puissance(5, 0)
Expecting:
    1
ok
Trying:
    puissance(-2, 5)
Expecting:
    -32
ok
Trying:
    puissance(0.5, 2)
Expecting:
    0.25
ok
3 items had no tests:
    __main__
    __main__.f
    __main__.indice_maxi_tab
1 items passed all tests:
   5 tests in __main__.puissance
5 tests in 4 items.
5 passed and 0 failed.
Test passed.




TestResults(failed=0, attempted=5)
```

> **À FAIRE :** Exercices 8, 9 et 10

Conclusion

Nous avons vu :

- comment *documenter* une fonction en spécifiant son comportement de manière concise et précise dans sa docstring. Cette pratique permet aux autres utilisateurs de comprendre le rôle de la fonction et son domaine d'utilisation (préconditions sur les paramètres) ;
- que la construction `assert` était un bon moyen d'effectuer une série de *tests* pour se convaincre que notre programme est correct et, éventuellement, de mettre en évidence des erreurs ;
- qu'il est possible de rechercher les erreurs en *affichant les valeurs* de certaines variables à des endroits stratégiques ;
- qu'il n'y a pas de méthode systématique pour s'assurer qu'on a pensé à tous les tests importants : il faut donc être particulièrement vigilant pour élaborer un *jeu de tests de qualité* ;
- qu'il existe une syntaxe permettant d'inclure les tests dans la chaîne de documentation et un outil permettant de les extraire et de les vérifier automatiquement.

Pour aller plus loin

- Il existe bien d'autres outils de documentation (pydoc, Sphinx) et de test (pytest, unittest), pour des usages plus complexes.

- Il est possible, de manière optionnelle, d'indiquer dans l'en-tête d'une fonction le type de certains paramètres et / ou du résultat. Cela peut être utile par exemple pour alléger la chaîne de documentation. Il existe aussi des outils externes qui permettent de vérifier que ces annotations de types sont vérifiées.

Références :

- Documents ressources du DIU EIL Nantes, C. DECLERCQ.
- Numérique et Sciences Informatiques, 1re, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, éditions ELLIPSES : Site du livre
- Ressource Eduscol : Mise au point de programmes testés

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS