Documenter et tester un programme

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 : pour cela, le programmeur doit créer une **documentation**.

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. Pour cela, le programmeur doit écrire un ou plusieurs **jeux de tests** et s'assurer que les tests passent avec succès. 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.

In [1]:

```
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. En l'analysant un peu il est possible de comprendre ce que fait cette fonction mais on n'en a pas toujours ni le temps ni l'envie (surtout si ce n'est pas notre travail !)

> 😕 Alors, elle fait quoi selon vous ?

Ainsi, le version ci-dessous aurait déjà été plus parlante :

In [2]:

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

Un nommage explicite est donc une première (bonne) étape mais elle n'est pas suffisante. En effet, on se doute désormais que cette fonction calcule une puissance mais c'est à peu près tout :

- est-ce x^n ? ou n^x ? ou autre chose ? ...
- est-ce que cela marche pour n'importe quelles valeurs de `x` et de `n` ? ...

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.

In [3]:

```
# 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 ni à ce commentaire et donc nous ne serions pas beaucoup plus avancé...

Écrire 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.

In [4]:

```
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`.

In [5]:

```
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.

In [6]:

```
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.
```

In [7]:

```
help(len)
```

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

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

Maintenant que l'on sait mieux ce que fait cette fonction on peut facilement l'utiliser pour calculer par exemple 2^3, $\left(\frac{1}{2}\right)^2$, $(-2)^5$:

In [8]:

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

Out[8]:

```
8
```

In [9]:

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

Out[9]:

```
0.25
```

In [10]:

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

Out[10]:

```
-32
```

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

In [11]:

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

Out[11]:

```
1
```

In [12]:

```
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-12-de97611f6342> in <module>
----> 1 puissance(4, 1/2) # ne renvoie pas la valeur de 4^(1/2) = 2 mais lève une erreur

<ipython-input-4-81cdac8eae66> in puissance(x, n)
      4     """
      5     r = 1
----> 6     for i in range(n):
      7         r = r * x
      8     return r

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

Être précis dans sa documentation

L'objectif d'une **chaîne de documentation** est d'être courte mais aussi *précise*.

On voit sur les deux derniers appels 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**
- les paramètres ainsi que les valeurs acceptées de ces paramètres (= **précondition**)
- la (les) valeur(s) qu'elle renvoie ainsi que d'éventuelles conditions sur ces valeurs (= **postcondition**)

Ainsi, la version suivante de notre chaine de documentation est bien meilleure car elle indique quelles sont les conditions (sur les paramètres) d'utilisation de la fonction.

In [13]:

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

Ou encore, de manière plus "professionnelle", on peut écrire :

In [14]:

```
def puissance(x, n):
    """
    Renvoie la valeur de x^n.

    Paramètres
    ----------
        x : float
        n : int
            n doit être positif ou nul
    """
    r = 1
    for i in range(n):
        r = r * x
    return r
```

✍ **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` suivie d'une condition à tester.

Si cette condition est vraie, alors il ne se passe rien et le programme poursuit son exécution :

In [15]:

```
# Cas d'un test valide
a = -2
assert a < 0
print(a)  # est exécuté car l'assertion précédente est vraie
```

```
-2
```

En revanche, si la condition est fausse, alors une erreur (de type `AssetionError`, soit *erreur d'assertion*) est détectée et stoppe l'exécution du reste du programme.

Par exemple, le test invalide suivant produit l'affichage d'un message d'erreur et stoppe le programme (ici la dernière ligne n'est pas exécutée)

In [16]:

```
# Cas d'un test invalide
a = -2
assert a >= 0
print(a)  # n'est pas exécuté car l'assertion précédente est fausse
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-16-2b60d98e5060> in <module>
      1 # Cas d'un test invalide
      2 a = -2
----> 3 assert a >= 0
      4 print(a)  # n'est pas exécuté car l'assertion précédente est fausse

AssertionError:
```

On peut faire suivre la condition à tester d'un message pour expliquer l'erreur

In [17]:

```
# ajout du message à afficher en cas d'erreur
a = -2
assert a >= 0, "le nombre a n'est pas positif"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-17-c989dabdadf5> in <module>
      1 # ajout du message à afficher en cas d'erreur
      2 a = -2
----> 3 assert a >= 0, "le nombre a n'est pas positif"

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

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

In [18]:

```
def puissance(x, n):
    """
    Renvoie la valeur de x^n, où x est un flottant et 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 est interrompue (à la ligne 5) et le message d'erreur sera affiché (sinon, le programme se poursuit).

In [19]:

```
puissance(2, -3)  # précondition non vérifiée
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-19-04dc12ea1cf6> in <module>
----> 1 puissance(2, -3)  # précondition non vérifiée

<ipython-input-18-36877ddaacf4> in puissance(x, n)
      3     Renvoie la valeur de x^n, où x est un flottant et 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
```

In [20]:

```
puissance(2, 3)  # précondition vérifiée
```

Out[20]:

```
8
```

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*.

Utiliser des jeux 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.

In [21]:

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

# jeu de tests :
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).

Écrire 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 `appartient(v, T)` dont la spécification, écrite dans sa docstring, est la suivante :

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    # CODE A ECRIRE
```

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

In [ ]:

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    # CODE DE LA FONCTION À ÉCRIRE


# jeu de tests
assert appartient(2, [2, 4, 7, 5]) == True
assert appartient(2, [0, 4, 7, 5]) == False
assert appartient(2, [2]) == True
```

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.

L'importance de la qualité du jeu de tests

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` aucun des tests proposés n'échoue.

In [22]:

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    for i in range(len(t)):
        if t[i] == v:
            return True
        else:
            return False


# jeu de tests
assert appartient(2, [2, 4, 7, 5]) == True
assert appartient(2, [0, 4, 7, 5]) == False
assert appartient(2, [2]) == True
```

Tous les tests de notre jeu de tests passent avec succès, super notre fonction est correcte ! 😎💪

... Et pourtant, avec l'appel suivant c'est le drame ... 😮😱

In [23]:

```
appartient(2, [0, 4, 2, 5])
```

Out[23]:

```
False
```

Cet appel met en évidence que **le code de notre fonction est incorrect** puisque 2 appartient bien au tableau `[0, 4, 2, 5]`.

Pire, l'appel ci-dessous ne renvoie rien alors qu'il devrait renvoyer `False` 😮😱...

In [24]:

```
appartient(2, [])
```

Notre code est donc (au moins) doublement incorrect !! Et pourtant tous les tests de notre jeu de tests sont passés avec succès... Cela montre que notre jeu de tests n'était pas bon et que la qualité du jeu de tests est primordiale !

Mais qu'est-ce qu'un *bon* jeu de tests ?

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.

Essayons d'améliorer notre jeu de tests pour la fonction `appartient` en suivant ces préconisations !

Améliorer le jeu de tests

Notre jeu de tests est pour le moment le suivant :

```
# jeu de tests
assert appartient(2, [2, 4, 7, 5]) == True
assert appartient(2, [0, 4, 7, 5]) == False
assert appartient(2, [2]) == True
```

On a bien écrit un test pour lequel la fonction renvoie `True` et un autre pour lequel la fonction renvoie `False`. On a pensé à faire un test pour un tableau particuler : celui réduit à un seul élément (le troisième test). Par ailleurs, nos deux tests pour lesquels la fonction renvoie `True` sont particuliers car à chaque fois la valeur `v` cherchée se trouve en première position (d'indice 0) dans le tableau.

Améliorons cela en partant des deux appels `appartient(2, [0, 4, 2, 5])` et `appartient(2, [])` effectués précédemment et qui ont montré que notre fonction était incorrecte. Si on a analyse ces deux exemples, on se rend compte que :

- notre fonction ne renvoie pas la bonne valeur dans le cas d'un tableau vide. Il faudra donc intégrer un test pour le cas très particulier du tableau vide :

```
assert appartient(2, []) == False
```

- notre fonction ne renvoie pas la bonne valeur dans le cas où la valeur cherchée se trouve en "milieu" de tableau. On va intégrer un tel test à notre jeu de tests :

```
assert appartient(2, [0, 4, 2, 5]) == True
```

Tant qu'à faire, on peut ajouter un dernier test où la valeur `v` se trouve en dernière position du tableau, ce qui est aussi un cas particulier à considérer. Voici donc le jeu tests amélioré proposé (qui ne passera pas avec succès puisque nous n'avons toujours pas modifié le code de notre fonction `appartient`) :

In [ ]:

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    for i in range(len(t)):
        if t[i] == v:
            return True
        else:
            return False

# un meilleur jeu de tests :
assert appartient(2, [2, 4, 7, 5]) == True
assert appartient(2, [0, 4, 7, 5]) == False
assert appartient(2, [2]) == True
assert appartient(2, [0, 4, 2, 5]) == True  # cas où v est au "milieu du tableau"
assert appartient(2, [0, 4, 5, 2]) == True  # cas où v est en dernière position
assert appartient(2, []) == False  # cas du tableau vide
```

> On va terminer en corrigeant notre fonction pour tous les tests soient validés !

Corriger sa fonction

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'utiliser des outils numériques.

Première possibilité : afficher les valeurs de certaines variables

Pour un programmeur débutant, la première idée à mettre en oeuvre est souvent d'afficher les valeurs de certaines variables à des endroits stratégiques du programme. Par exemple, on peut afficher la valeur de la variable `i` à chaque tour de boucle, et afficher le message `"ici"` ou `"là"` selon que l'on passe dans le `if` ou dans le `else` :

In [25]:

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    for i in range(1, len(t)):
        print("i =", i)
        if t[i] == v:
            print("ici")
            return True
        else:
            print("là")
            return False
```

On utilise le premier appel qui posait problème :

In [26]:

```
appartient(2, [0, 4, 2, 5])
```

```
i = 1
là
```

Out[26]:

```
False
```

On se rend compte que la variable `i` ne prend que la valeur `0`, autrement dit que l'on ne fait qu'un seul tour de boucle : le premier (celui d'indice `0`). De plus, lors de ce premier tour de boucle, on voit que le message `"là"` est affiché, autrement dit que l'on passe dans le `else`. Or, si on arrive dans le `else` notre fonction renvoie `False`.

Or, il ne faut pas renvoyer `False` mais poursuivre la recherche si la valeur `v` n'est pas en première position (ni à une autre position d'ailleurs). On peut alors corriger notre programme en retirant le `else` et en renvoyant `False` **après** la boucle `for` !

In [27]:

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    for i in range(len(t)):
        print("i =", i)
        if t[i] == v:
            print("ici")
            return True
    print("là")
    return False
```

In [28]:

```
appartient(2, [0, 4, 2, 5])
```

```
i = 0
i = 1
i = 2
ici
```

Out[28]:

```
True
```

On peut essayer à nouveau notre jeu de tests (en enlevant les affichages) :

In [29]:

```
def appartient(v, t):
    """
    Renvoie True si l'entier v appartient à tableau d'entiers t, et False sinon.
    """
    for i in range(len(t)):
        if t[i] == v:
            return True
    return False

# un meilleur jeu de tests :
assert appartient(2, [2, 4, 7, 5]) == True
assert appartient(2, [0, 4, 7, 5]) == False
assert appartient(2, [2]) == True
assert appartient(2, [0, 4, 2, 5]) == True  # cas où v est au "milieu du tableau"
assert appartient(2, [0, 4, 5, 2]) == True  # cas où v est en dernière position
assert appartient(2, []) == False  # cas du tableau vide
```

Super, tous les tests passent avec succès, et comme notre jeu de tests couvre tous les cas particuliers, on peut considérer que notre fonction est *a priori* correcte 👍.

Deuxième possibilité : exécuter un programme ligne par ligne

Il existe des outils efficaces permettant d'exécuter ligne par ligne un programme pour y déceler une erreur. L'un d'entre eux se nomme *Python tutor*. Il n'est alors plus nécessaire d'utiliser des `print` pour faire des affichages.

Voici un lien vers Python tutor pour exécuter l'appel `appartient(2, [0, 4, 2, 5])` avec la version incorrecte de la fonction `appartient`. On peut alors suivre les exécutions ligne après ligne pour se rendre compte qu'il n'y a qu'un seul tour de boucle avec renvoi de la valeur `False` dès ce premier tour.

Troisième possibilité : utiliser des outils de débogage

Enfin, les IDE les plus récents proposent généralement des *débogueurs* (ou *debugger*) qui permettent d'exécuter les programmes ligne par ligne et d'insérer des *points d'arrêts* pour stopper l'exécution du programme à certains endroits. On peut ainsi suivre les valeurs des variables pas à pas pour trouver la ou les erreurs.

> ✍ **À FAIRE :** Exercices 5, 6, 7 et 8

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.

In [30]:

```
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).

In [31]:

```
import doctest
```

In [32]:

```
doctest.testmod()
```

Out[32]:

```
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.

In [33]:

```
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__.appartient
    __main__.f
1 items passed all tests:
   5 tests in __main__.puissance
5 tests in 4 items.
5 passed and 0 failed.
Test passed.
```

Out[33]:

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

✍ **À FAIRE :** Exercice 9.

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 ou, de manière plus professionnelle, utiliser des *debogueurs* ;
- 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* ;
- en particulier, le succès d'un jeu de tests ne garantit pas qu'un programme est correct ;
- qu'il existe une syntaxe permettant d'inclure les tests dans la chaîne de documentation et un outil (comme le module `doctest`) 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, Université de 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