

# Documenter et tester un programme

1<sup>ère</sup> NSI

# Introduction

# Introduction

**Utilisateur**  
(d'une fonction)



**Programmeur**  
(d'une fonction ou  
d'un programme)

# Introduction

## Utilisateur (d'une fonction)

- Pas besoin
  - de connaître le code
  - de comprendre le code
- Besoin de connaître le rôle de la fonction  
= ce que fait la fonction  
(pour pouvoir l'utiliser)



## Programmeur (d'une fonction ou d'un programme)

# Introduction

## Utilisateur (d'une fonction)

- Pas besoin
  - de connaître le code
  - de comprendre le code
- Besoin de connaître le rôle de la fonction  
= ce que fait la fonction  
(pour pouvoir l'utiliser)



## Programmeur (d'une fonction ou d'un programme)

- Doit expliquer ce que fait son programme  
→ création d'une **documentation**

# Introduction

## Utilisateur (d'une fonction)

- Pas besoin
  - de connaître le code
  - de comprendre le code
- Besoin de connaître le rôle de la fonction  
= ce que fait la fonction  
(pour pouvoir l'utiliser)



## Programmeur (d'une fonction ou d'un programme)

- Doit expliquer ce que fait son programme  
→ création d'une **documentation**
- Doit s'assurer que son programme est correct  
→ mise en place d'un **jeux de tests**

# Que « fait » un programme ?

# Que « fait » un programme ?

À quoi sert cette fonction ?

In [1]:

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



# Que « fait » un programme ?

## À quoi sert cette fonction ?

In [1]:

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

- En analysant un peu, on peut essayer de comprendre le rôle de la fonction...
- Mais :
  - Ni le temps ni l'envie
  - Surtout : ce n'est pas notre travail !

# Que « fait » un programme ?

À quoi sert cette fonction ?

In [1]:

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

- En analysant un peu, on peut essayer de comprendre le rôle de la fonction...
- Mais :
  - Ni le temps ni l'envie
  - Surtout : ce n'est pas notre travail !

Mieux :

In [2]:

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

# Que « fait » un programme ?

## À quoi sert cette fonction ?

In [1]:

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

- En analysant un peu, on peut essayer de comprendre le rôle de la fonction...
- Mais :
  - Ni le temps ni l'envie
  - Surtout : ce n'est pas notre travail !

## Mieux :

In [2]:

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

- Mais pas suffisant...
- Calcul d'une puissance...
  - $x^n$
  - $n^x$
  - Autre chose ?
  - Pour quelles valeurs de  $x$  ? De  $n$  ?

# Documenter un programme

# Documenter un programme

- Le programmeur doit toujours expliquer son programme (ici sa fonction)
- Première possibilité :

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
```

# Documenter un programme

- Le programmeur doit toujours expliquer son programme (ici sa fonction)

- Première possibilité :

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
```

- Beaucoup mieux : utiliser une **chaîne de documentation** (ou **docstring** en anglais)

- Au début du code de la fonction
- 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
```



# Documenter un programme

- Car permet à l'utilisateur de la consulter avec la fonction `help()` :

In [5]:

```
help(puissance)
```

```
Help on function puissance in module __main__:
```

```
puissance(x, n)
```

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

In [4]:

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

# Documenter un programme

- Car permet à l'utilisateur de la consulter avec la fonction `help()` :

```
In [5]: help(puissance)
Help on function puissance in module __main__:

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

In [4]:

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

```
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.
```



# Documenter un programme

- Maintenant que l'utilisateur connaît le rôle de cette fonction, il peut l'utiliser sereinement :

```
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
```

# Documenter un programme

- ... ou pas... :

```
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
```

# Documenter un programme

- Il faut être précis dans sa chaîne de documentation !
  - **Rôle** de la fonction
  - Les **paramètres** et les **valeurs acceptées** (= *préconditions*)
  - La ou les **valeurs renvoyée(s)** + éventuelles conditions sur ces valeurs (= *postconditions*)

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
```

# Documenter un programme

- Il faut être précis dans sa documentation !
  - Une version plus « professionnelle » :

```
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
```

# Documenter un programme

- Faire les exercices 1 et 2

# Programmation défensive

# Programmation défensive

- Le programmeur a fait correctement son travail si sa fonction est correcte et bien documentée :

```
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
```

- Et si l'utilisateur essaie d'appeler la fonction avec des paramètres d'entrée ne respectant pas les préconditions ?

```
In [11]: puissance(2, -3)
```

```
Out[11]: 1
```

# Programmation défensive

- Le programmeur a fait correctement son travail si sa fonction est correcte et bien documentée :

```
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
```

- Et si l'utilisateur essaie d'appeler la fonction avec des paramètres d'entrée ne respectant pas les préconditions ?

```
In [11]: puissance(2, -3)
```

→ C'est de la faute de l'utilisateur !

```
Out[11]: 1
```



# Programmation défensive

- Néanmoins, le programmeur a la possibilité de faire en sorte que sa fonction soit bien utilisée : grâce à la construction `assert`

```
In [15]: # Cas d'un test valide
a = -2
assert a < 0
print(a) # est exécuté car l'assertion précédente est vraie
```

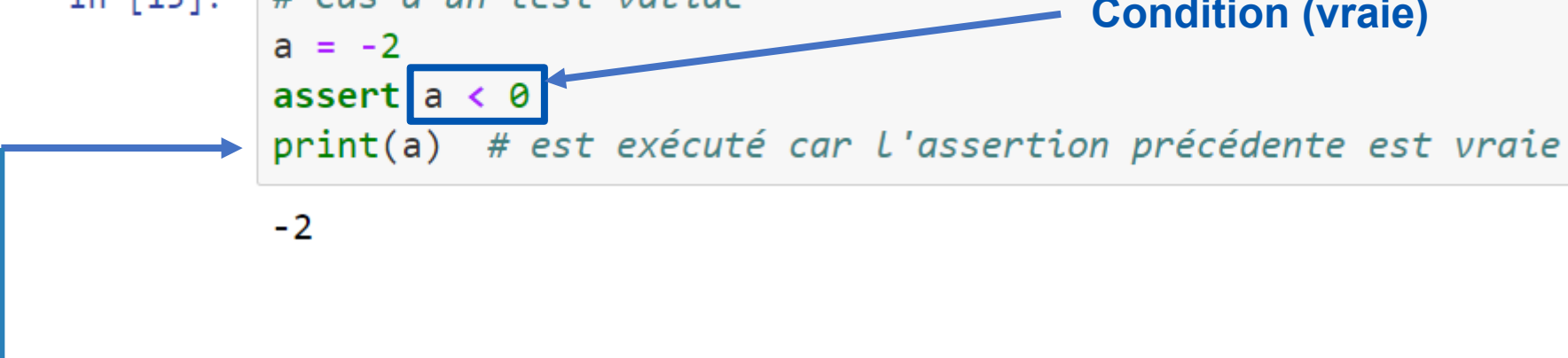
-2

**Condition (vraie)**

# Programmation défensive

- Néanmoins, le programmeur a la possibilité de faire en sorte que sa fonction soit bien utilisée : grâce à la construction `assert`

```
In [15]: # Cas d'un test valide
a = -2
assert a < 0
print(a) # est exécuté car l'assertion précédente est vraie
-2
```



Condition (vraie)

Le reste du code est exécuté car la condition est vraie

# Programmation défensive

- Néanmoins, le programmeur a la possibilité de faire en sorte que sa fonction soit bien utilisée : grâce à la construction `assert`

In [16]:

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

Condition (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:**

# Programmation défensive

- Néanmoins, le programmeur a la possibilité de faire en sorte que sa fonction soit bien utilisée : grâce à la construction `assert`

In [16]:

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

Condition (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:

Une erreur de type `AssertionError` (erreur d'assertion) est levée

# Programmation défensive

- Néanmoins, le programmeur a la possibilité de faire en sorte que sa fonction soit bien utilisée : grâce à la construction `assert`

```
In [16]: # Cas d'un test invalide
a = -2
assert a >= 0
print(a) # n'est pas exécuté car l'assertion précédente est fausse
```

Condition (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:

Une erreur de type `AssertionError` (erreur d'assertion) est levée

Le reste du code n'est pas exécuté car la condition est fausse

# Programmation défensive

- On peut personnaliser le message d'erreur :

```
In [17]: # ajout du message à afficher en cas d'erreur
a = -2
assert a >= 0, "le nombre a n'est pas positif"
```

Personnalisation du message d'erreur à afficher

```
-----
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
```

# Programmation défensive

- On peut personnaliser le message d'erreur :

In [17]:

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

Personnalisation du message d'erreur à afficher

```
-----  
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
```

# Programmation défensive

- Utilisation sur notre exemple :

```
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
```



# Programmation défensive

- Utilisation sur notre exemple :

```
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
```

- Pas de problème en cas d'assertion vraie :

```
In [20]: puissance(2, 3) # précondition vérifiée
```

```
Out[20]: 8
```

# Programmation défensive

- Et interruption de la fonction en cas d'assertion fausse :

```
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
```

# Programmation défensive

- Il est aussi possible d'écrire plusieurs assertions si on veut s'assurer que plusieurs conditions soient vraies

```
assert type(n) == int and n >= 0, "n n'est pas un entier positif ou nul"
```

équivalent à :

```
assert type(n) == int, "n n'est pas un entier"  
assert n >= 0, "n n'est pas positif"
```

# Documenter un programme

- Faire les exercices 3 et 4

# Tester ses programmes pour (se) convaincre

# Tester ses programmes pour (se) convaincre

- Le programmeur peut spécifier et documenter convenablement sa fonction sans que celle-ci soit correcte (il n'est pas rare de se tromper dans le code).
- Pour repérer les erreurs éventuelles il doit tester son programme
- Au lieu de faire les tests un à un, il est plus pertinent d'écrire des **jeux de tests**
- Cela peut se faire directement dans le programme, en utilisant `assert`

# Jeux de tests

```
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 des tests échoue, un message indique le premier test qui échoue et la fonction doit être corrigée
- Une fois la correction apportée, il faut relancer **tous** les tests

# Jeux de tests

- Une pratique intéressant est d'écrire les tests avant même d'écrire le code de la fonction !

```
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
```

- Après avoir écrit le code de la fonction, on exécute le programme :

- Si un test échoue : il faut corriger notre fonction



- Si tous les tests passent avec succès : **rien ne dit que notre fonction est correcte !**



# Jeux de tests

- Ici, tous les tests suivants passent avec succès :

```
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
```

# Jeux de tests

- Ici, tous les tests suivants passent avec succès :

```
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
```

- Mais ...

```
In [23]: appartient(2, [0, 4, 2, 5])
```

```
Out[23]: False
```

```
In [24]: appartient(2, []) # ne renvoie rien !
```

# Jeux de tests

- Qu'est-ce qu'un **bon** jeu de tests ? Difficile !
  - Impossible (la plupart du temps) de couvrir l'ensemble des cas possibles
  - Mais on peut suivre quelques règles :
    - 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.

# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

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

# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

```
# jeu de tests
```

```
assert appartient(2, [2, 4, 7, 5]) == True
```

```
assert appartient(2, [0, 4, 7, 5]) == False
```

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

Un cas où la fonction renvoie True

Un cas où la fonction renvoie False

Cas particulier d'un tableau réduit à un seul élément

# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

```
# jeu de tests
```

```
assert appartient(2, [2, 4, 7, 5]) == True
```

```
assert appartient(2, [0, 4, 7, 5]) == False
```

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

→ Un cas où la fonction renvoie True

→ Un cas où la fonction renvoie False

→ Cas particulier d'un tableau réduit à un seul élément

- On peut partir des tests qui ont échoué :

# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

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

Un cas où la fonction renvoie True  
Un cas où la fonction renvoie False  
Cas particulier d'un tableau réduit à un seul élément

- On peut partir des tests qui ont échoué :

```
In [23]: appartient(2, [0, 4, 2, 5])
```

```
Out[23]: False
```

# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

```
# jeu de tests
```

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

→ Un cas où la fonction renvoie True

→ Un cas où la fonction renvoie False

→ Cas particulier d'un tableau réduit à un seul élément

- On peut partir des tests qui ont échoué :

```
In [23]: appartient(2, [0, 4, 2, 5])
```

```
Out[23]: False
```

→ Il manque un test où la valeur cherchée est « au milieu » du tableau

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



# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

```
# jeu de tests
```

```
assert appartient(2, [2, 4, 7, 5]) == True
```

```
assert appartient(2, [0, 4, 7, 5]) == False
```

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

Un cas où la fonction renvoie True

Un cas où la fonction renvoie False

Cas particulier d'un tableau réduit à un seul élément

- On peut partir des tests qui ont échoué :

```
In [23]: appartient(2, [0, 4, 2, 5])
```

```
Out[23]: False
```

Il manque un test où la valeur cherché est « au milieu » du tableau

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

```
In [24]: appartient(2, []) # ne renvoie rien !
```

# Jeux de tests

- Essayons d'améliorer le jeu de tests pour notre fonction appartient

```
# jeu de tests
```

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

Un cas où la fonction renvoie True

Un cas où la fonction renvoie False

Cas particulier d'un tableau réduit à un seul élément

- On peut partir des tests qui ont échoué :

```
In [23]: appartient(2, [0, 4, 2, 5])
```

Il manque un test où la valeur cherché est « au milieu » du tableau

```
Out[23]: False
```

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

```
In [24]: appartient(2, []) # ne renvoie rien !
```

Il manque un test pour le cas particulier du tableau vide

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

# Jeux de tests

- En ajoutant un test où la valeur cherchée est en dernière position :

```
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
```

Des tests  
qui  
échouent  
(pour le  
moment)

# Corriger sa fonction

# Corriger sa fonction

- Comment trouver les erreurs ? → en partant d'un test qui échoue en suivant l'état des variables
  - Mentalement ou sur papier
  - Ou avec des outils numériques :
    - 1ère possibilité : afficher les valeurs de certaines variables
    - 2ème possibilité : exécuter un programme ligne par ligne (possible dans la plupart des IDE, ou sur Python Tutor)
    - 3ème possibilité : utiliser un débogueur (ou debugger) intégré à un IDE

# Corriger sa fonction

- On affiche la valeur de certaines variables à des endroits stratégiques :

In [25]:

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

Pour suivre la variable *i* à chaque tour de boucle

Pour voir si on passe « dans le if » ou « dans le else »

# Corriger sa fonction

- On affiche la valeur de certaines variables à des endroits stratégiques :

In [25]:

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

Pour suivre la variable *i* à chaque tour de boucle

Pour voir si on passe « dans le if » ou « dans le else »

In [26]:

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

```
i = 1  
là
```

Out[26]: False

# Corriger sa fonction

- On affiche la valeur de certaines variables à des endroits stratégiques :

In [25]:

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

Pour suivre la variable *i* à chaque tour de boucle

Pour voir si on passe « dans le if » ou « dans le else »

## Constatations :

- la variable *i* ne prend que la valeur 1  
→ il n'y a qu'un seul tour de boucle
- au cours de ce 1<sup>er</sup> tour : on passe dans le `else` et donc notre fonction renvoie `False` (dès le premier tour)

In [26]:

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

```
i = 1  
là
```

Out[26]: `False`



# Corriger sa fonction

- On corrige :

```
In [ ]: def appartient(v, t):  
        """  
        Renvoie True si l'entier v  
        """  
        for i in range(len(t)):  
            if t[i] == v:  
                return True  
            else:  
                return False
```

**Il ne faut pas renvoyer  
False si  $T[i] \neq v$  mais  
seulement lorsque tout le  
tableau a été parcouru**

# Corriger sa fonction

- On corrige :

```
In [ ]: def appartient(v, t):  
        """  
        Renvoie True si l'entier v  
        """  
        for i in range(len(t)):  
            if t[i] == v:  
                return True  
            else:  
                return False
```

**Il ne faut pas renvoyer False si  $T[i] \neq v$  mais seulement lorsque tout le tableau a été parcouru**

```
In [29]: def appartient(v, t):  
        """  
        Renvoie True si l'entier v appartient à t  
        """  
        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 #  
assert appartient(2, [0, 4, 5, 2]) == True #  
assert appartient(2, []) == False # cas du t
```

**Tous les tests passent avec succès**

# Corriger sa fonction

- On corrige :

```
In [ ]: def appartient(v, t):
        """
        Renvoie True si l'entier v
        """
        for i in range(len(t)):
            if t[i] == v:
                return True
            else:
                return False
```

**Il ne faut pas renvoyer False si  $T[i] \neq v$  mais seulement lorsque tout le tableau a été parcouru**

```
In [29]: def appartient(v, t):
        """
        Renvoie True si l'entier v appartient à t
        """
        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 #
assert appartient(2, [0, 4, 5, 2]) == True #
assert appartient(2, []) == False # cas du t
```

**Tous les tests passent avec succès**

→ **notre fonction est a priori correcte**

# Documenter un programme

- Faire les exercices 5, 6, 7, 8

# Intégration des tests à la chaîne de documentation

- Choisir et vérifier des tests pertinents = travail précieux → important d'en garder trace
- Des tests bien choisis peuvent constituer une explication très efficace de l'effet d'une fonction
  - On peut les **inclure directement dans la chaîne de documentation**
  - Utiliser des **modules spécifiques pour extraire les tests et les vérifier** (par exemple doctest, mais il en existe d'autres)

# Intégration des tests à la chaîne de documentation

- Syntaxe à respecter avec les trois chevrons (>>>)

```
In [30]: def puissance(x, n):  
    '''  
    Renvoie la valeur de x^r  
  
    >>> 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
```

# Intégration des tests à la chaîne de documentation

- Syntaxe à respecter avec les trois chevrons (>>>)
- Vérifier les tests avec le module doctest :

```
In [31]: import doctest
```

```
In [32]: doctest.testmod()
```

```
Out[32]: TestResults(failed=0, attempted=5)
```

```
In [30]: def puissance(x, n):  
        '''  
        Renvoie la valeur de x^r  
  
        >>> 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
```

# Intégration des tests à la chaîne de documentation

- Syntaxe à respecter avec les trois chevrons (>>>)
- Vérifier les tests avec le module doctest :

```
In [31]: import doctest
```

```
In [32]: doctest.testmod()
```

```
Out[32]: TestResults(failed=0, attempted=5)
```

Si on veut plus de détails

```
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
Trving:
```

```
In [30]: def puissance(x, n):
        '''
        Renvoie la valeur de x^n

        >>> 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
```



# Documenter un programme

- Faire l'exercice 9

# Bilan

- Le programmeur peut/doit documenter sa fonction en spécifiant son comportement de manière concise et précise dans sa chaîne de documentation (ou docstring)  
→ permet aux autres utilisateurs de comprendre le rôle de la fonction et son domaine d'utilisation
- Le programmeur a la possibilité de se prémunir d'une mauvaise utilisation de sa fonction (programmation défensive) grâce à la construction `assert`
- La construction `assert` permet aussi d'effectuer une série de tests pour se convaincre que notre programme est correct ou mettre en évidence des erreurs
- Un bon jeu de tests est difficile à élaborer : il doit contenir tous les tests importants, en particulier il faut penser aux cas particuliers
- Le succès d'un jeu de tests ne garantit pas qu'un programme est correct

# Sources / 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 manuel
- Ressource Eduscol : Mise au point de programmes testés