

# CM 2 : Spécification de fonction

Info1.Algo1

2022-2023 Semestre Impair

- 1 Introduction
  - Problème posé
  - Spécification de fonction
  - Tests de propriété
- 2 Application : maximum d'une liste
- 3 Bilan

## Cadre du problème

Le cadre d'étude proposé est celui des **fonctions**. Il est cependant **généralisable** à tout code présentant une ou plusieurs entrées, et une ou plusieurs sorties (programme principal, portion de code).

On exclut les situations suivantes :

- **Interactions** avec l'utilisateur : les entrées sont données simultanément, les sorties sont recueillies simultanément.
- **Effets de bords** non-testables (affichage, ...)

# Problème posé

## Exemple

On considère la fonction suivante :

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

1) a) Déterminer les valeurs de retours de cette fonction dans les cas ci-contre.

b) Cette fonction effectue-t-elle bien la division euclidienne de a par b?

a	b
10	3
17	5
3	10
14	7
6	0

## Objectif du cours

Étant donnée une fonction, on souhaite répondre à la question suivante :

**La fonction effectue-t-elle bien ce qu'on attend qu'elle fasse?**

Pour répondre à cette question, on propose de procéder en deux étapes :

- 1 Décrire précisément ce que l'on attend d'une fonction : le **contrat de fonction** ou **spécification**.
- 2 Fournir un outil pour vérifier le comportement de la fonction : les **tests de propriétés**.

# Spécification de fonction

Afin de caractériser précisément ce que l'on attend d'une fonction, il est nécessaire de décrire les **exigences** que l'on se donne sur les paramètres et des valeurs de retour de la fonction étudiée.

## Définitions

- On appelle **pré-condition** la condition devant être vérifiée par les paramètres de la fonction afin que le problème puisse être résolu.
- On appelle **post-condition** la condition devant être vérifiée par la(les) valeur(s) de retour de la fonction. Cette vérification fait intervenir les paramètres de la fonction.

# Spécification de fonction

client

input  
satisfies **precondition**  $\Rightarrow$  output  
satisfies **postcondition**

*specification*

implementer

computation

## Définition

On appelle **spécification** d'une fonction la donnée :

- du **type** des paramètres et de la valeur de retour
- de la **pré-condition** et de la **post-condition**

de cette fonction.

La **spécification** de la fonction constitue le **contrat** qui lie le codeur de la fonction avec son utilisateur. Le codeur de la fonction s'engage, dès lors l'utilisateur respecte la pré-condition, à respecter la post-condition. On parle encore de **contrat de fonction**.



## Définition

On appelle **implémentation** d'une spécification donnée toute fonction qui respecte cette spécification.

- La spécification précise le **quoi**, tandis que l'implémentation précise le **comment**.
- La spécification est la seule chose à savoir pour pouvoir utiliser la fonction.

## Exemple (suite)

2) Dans cette question on s'intéresse à **division euclidienne de deux entiers positifs**.

a) Écrire la **spécification** de la fonction `division_euclidienne` dans ce cas.

b) Écrire une **implémentation** de cette fonction.

3) Comment sont modifiés la pré-condition et la post-condition lorsque l'on souhaite traiter le cas des entiers relatifs?

## Principe et instrumentation

- Le principe des **tests de propriété** est de vérifier **lors de l'exécution** certaines **propriétés** requises.
- La vérification est effectuée par des **assertions** (instruction `assert`).

# Tests de propriété

## Application : vérification d'une spécification

Les assertions destinées à vérifier la **pré-condition** et la **post-condition** sont placées respectivement en tout **début** et en toute **fin** de la fonction testée.

```
1 def ...(..., ...):  
2     assert ..., 'Pre-condition'  
3     ... # CODE DE LA FONCTION  
4     assert ..., 'Post-condition'  
5     return ...
```

## Exemple (suite et fin)

4) Compléter le code écrit à la question **2b** avec les assertions de pré- et post-condition.

# Tests de propriété

## Paramètres constants

Les paramètres de la fonction ne doivent pas être modifiés, sinon la post-condition ne peut pas être vérifiée correctement.

## Exemple

```
1 def division_euclidienne(a,b):  
2     assert a>=0 and b>0, 'Pre-condition'  
3     a,q,r = 0,0,0  
4     assert a==q*b+r and 0<=r<b, 'Post-condition'  
5     return q,r
```

La dernière assertion est bien vérifiée, mais n'est pas pertinente!

Exception : les tableaux (vu plus tard)

## Précisions :

- **Vérification statique des types** (non effectuée).
  - Annotations de type (Python).
  - Compilation (langages compilés).
- Si aucune pré-condition :

```
1 # Pas de pre-condition
```

ou bien :

```
1 assert True, 'Pre-condition'
```

## Rappels sur les instructions `assert`:

- Peuvent être **coûteuses en temps et/ou en mémoire**.
- Sont **exclusivement réservées au débogage**.
- Sont **désactivées** avec le mode optimisé de Python :

```
1 python3 -O fichier.py
```

- Doivent donc utiliser des expressions booléennes **sans effets de bord**.
- Ne doivent donc être utilisées :
  - **ni pour valider des entrées utilisateurs.**
  - **ni pour valider le contenu de fichiers.**

# Plan

- 1 Introduction
  - Problème posé
  - Spécification de fonction
  - Tests de propriété
- 2 Application : maximum d'une liste
- 3 Bilan



# Application : maximum d'une liste

## Énoncé

On souhaite écrire et valider avec des tests de propriété une fonction `maximum` qui accepte en entrée une liste d'entiers et retourne le maximum `m` de cette liste.

1) Pour les deux **fonctions auxiliaires** suivantes (qui servent justement aux tests de propriétés) aucune pré-condition / post-condition n'est demandée.

a) Écrire la fonction auxiliaire `est_membre` qui accepte en paramètre une liste d'entiers `liste` et un entier `m`, et retourne le booléen indiquant si `m` est un élément de `liste`.

b) Écrire la fonction auxiliaire `est_majorant` qui accepte en paramètre une liste d'entiers `liste` et un entier `m`, et retourne le booléen indiquant si `m` est supérieur ou égal à tous les éléments de `liste`.

# Application : maximum d'une liste

## Énoncé (suite)

2) a) Écrire la spécification de la fonction `maximum` (types des entrées / sorties, pré-condition et post-condition). On se servira bien-entendu des fonctions auxiliaires écrites à la question 1.

b) Écrire une implémentation de cette spécification.

3) Dans cette question, on souhaite analyser pourquoi chaque partie de la post-condition est nécessaire :

a) Si l'on suppose que l'on ne teste pas la propriété `est_membre(liste,m)`, donner un contenu de `liste` et une valeur de `m` qui vérifie le reste des propriétés sans que `m` soit le maximum de `liste`.

b) Même question si l'on suppose que l'on ne teste pas la propriété `est_majorant(liste,m)`.

# Plan

- 1 Introduction
  - Problème posé
  - Spécification de fonction
  - Tests de propriété
- 2 Application : maximum d'une liste
- 3 Bilan

# Comparaison avec d'autres approches

## Comparaison avec les tests unitaires

- **Tests de propriété :**
  - Vérification sur des **paramètres quelconques** non décidés à l'avance.
  - Tests effectués **en contexte** (lorsque la fonction est appelée par d'autres fonctions dans le cadre d'un code volumineux).
- **Tests unitaires :**
  - Vérification sur des **paramètres choisis** dans les fonctions de test.
  - Peuvent **cibler spécifiquement** certaines situations (cas extrêmes, particuliers, etc.). Exemple : 366e jour d'une année bissextile.

En anglais : *Property Based Testing* vs. *Example Based Testing*

## Comparaison avec la vérification formelle

- **Tests de propriété :**
  - Vérification **dynamique** lors de l'appel et de l'exécution de la fonction.
  - Le code de la fonction est une **boîte noire** : seul compte le résultat obtenu.
  - **Nombre fini** de combinaisons testées.
- **Vérification formelle** (vue en UE Info2.Algo2):
  - **Analyse statique du code de la fonction.**
  - Fournit une **preuve** que pour **toutes les valeurs possibles des paramètres**, le résultat retourné est correct.

Dijkstra : «*Program testing can be used to show the presence of bugs, but never to show their absence!*»

# Comparaison avec d'autres approches

## Comparaison avec la documentation de code

- **Tests de propriété :**
  - **Mise en œuvre de la description** : test effectué à l'exécution, sur des valeurs réelles.
  - Le code et la vérification sont cohérents (on ne peut pas changer l'un sans changer l'autre).
- **Documentation de code (docstring) :**
  - **Description de la fonction (du module, de la classe) ajoutée en première ligne du bloc.**
  - Fournit la documentation du code à destination du développeur.
  - Les exemples peuvent être utilisés comme tests unitaires au moyen du module doctest
  - La documentation est gérée au même endroit que le code qui l'implémente.

```

"""
Ceci est un module avec une fonction
    plus_un().
>>> plus_un(1)
2
"""
def plus_un(n):
    """Retourne n + 1.

    >>> plus_un(42)
    43

    >>> [plus_un(n) for n in range(6)]
    [1, 2, 3, 4, 5, 6]

    >>> plus_un('A')
    Traceback (most recent call last):
      ...
    ValueError: n doit etre un int
    """

    if not type(n) == int:
        raise ValueError("n doit etre un
                           int")
    if n == 42:
        return 666 #OUPS
    return n + 1

if __name__ == "__main__":
    help(plus_un) # affichage de l'aide.
    import doctest
    doctest.testmod() # declenche les tests

```

```

Help on function plus_un in module
    __main__:

plus_un(n)
    Retourne n + 1.

>>> plus_un(42)
43

>>> [plus_un(n) for n in range(6)]
[1, 2, 3, 4, 5, 6]

>>> plus_un('A')
Traceback (most recent call last):
  ...
ValueError: n doit etre un int

*****
File "main.py", line 9, in
    __main__.plus_un
Failed example:
    plus_un(42)
Expected:
    43
Got:
    666
*****
1 items had failures:
  1 of 3 in __main__.plus_un
***Test Failed*** 1 failures.

```

# Un double objectif

- **Une méthode de débogage** : permet d'identifier quelle portion de code a failli. S'utilise en complément des **tests unitaires**.
- **Un pas vers la vérification formelle** : Les fonctions utilisées ponctuellement pour vérifier que la fonction a correctement répondu à la question reprennent pour la plus grande partie les caractérisations logiques (formules, etc.) qui sont utilisées lors de la vérification formelle.