Cours 4 — Spécification & Validation

Bonne pratiques de spécification & Précisions sur les questions de validation

« *Testing shows the presence, not the absence of bugs.* »

Attribué à E. W. Dijkstra

Ce cours est orienté en direction du **contrôle-qualité** (*Quality process*) de la production d'un algorithme :

1. Contrôle qualité en matière de **spécifications** d'une procédure (fonction) : **signature**, précision et vérifications des **pré-conditions** et **post-conditions**, **commentaires**, **jeu de tests**.
2. Contrôle qualité en matière de **validation** d'une procédure : **preuve de terminaison** et **preuve de correction**.

Le troisième volet en matière de contrôle-qualité consiste à étudier et quantifier la **complexité** d'une procédure. Cela a été introduit dans le cours n°1.

Plan du cours

1 Spécifications d'une procédure

Une fois une procédure (ou fonction en Python) conçue, testée et analysée, on souhaite livrer un produit fini (**livrable**) possédant, en soi, toutes les informations permettant un usage valide de la procédure. Ceci passe par des **spécifications**.

L'analogue de cela en mathématique consiste à spécifier l'ensemble de définition et l'espace d'arrivée lorsque l'on définit une fonction au lien d'écrire uniquement $f(x) = blabla(x)$.

1.1 Signature

La première spécifications importante est la **signature** d'une fonction :

Definition 1 — *La **signature d'une fonction** est la donnée :*

1. *du nom de la fonction ;*
2. *du nom de ses arguments et de leur type ;*
3. *du type de la ou des valeur(s) renvoyées par la fonction.*

Le plus simple pour préciser cette signature dans le corps de la fonction est d'utiliser l'**annotation** suivante :

```
def ma_fonction(var1:type1,var2:type2) -> type de sortie:
```

Ainsi par exemple, pour l'algorithme de recherche dichotomique dans un tableau trié, nous écririons :

```
def bin_search(L:list,x:int) -> bool:
    g = 0
    d = len(L)-1
    while g <= d:
        m=(g+d)//2 # milieu ou partie entière du milieu
        if L[m] == x:
            return True
        elif L[m]<x:
            g=m+1
        else:
            d=m-1
    return False
```

A noter, que, si la fonction `bin_search` est par exemple définie de la sorte, alors : `bin_search.__annotations__` permet d'afficher la signature de la fonction `bin_search`.

On notera que cette annotation n'est pas évaluée par Python lors d'une exécution de la fonction, mais est considérée comme un commentaire. Notamment, lors d'un appel de la fonction aucun test ne sera effectué pour vérifié l'appartenance des types des arguments aux types spécifiés — il s'agit vraiment juste d'une « annotation ».

Enfin, une fonction qui ne sert par exemple qu'à afficher un graphe ou à afficher à l'aide de `print` aura un type de sortie qualifié de `none`.

1.2 Docstring

La deuxième spécification importante consiste à apporter un entête à la fonction visant à :

1. expliquer en une ou deux lignes la tâche effectuée par la fonction ;
2. expliquer les contraintes éventuelles pesant sur les arguments (les **pré-conditions**) ou sur valeurs retournées par la fonction (les **post-conditions**).

Pour ce faire, il est préférable d'utiliser le format standardisé du **Docstring**. Ainsi, pour reprendre notre exemple de la fonction `bin_search`, on écrira :

```
def bin_search(L:list,x:int) -> bool:
    '''
    Recherche dichotomique d'un élément dans un tableau trié
    Entrée : L une liste triée par ordre croissant contenant des flottants
             x un flottant
    Sortie : un booléen Vrai si x appartient à la liste et Faux sinon
    '''
    g = 0
    d = len(L)-1
    while g <= d:
        m=(g+d)//2 # milieu ou partie entière du milieu
        if L[m] == x:
            return True
        elif L[m]<x:
            g=m+1
        else:
            d=m-1
    return False
```

Plusieurs remarques :

1. Ainsi écrit, cet entête est un commentaire et donc n'est pas interprété par Python lors d'une exécution de la fonction. Cependant, ce format standard est reconnu par Python comme un **Docstring** et la commande `help(bin_search)` renverra le doctring en question. On notera à ce sujet que cette commande `help` affiche les docstring de toutes les fonctions Python que nous utilisons (on tapera pour s'en convaincre `help(divmod)` par exemple).
2. Ce Docstring doit être extrêmement synthétique mais comporter d'autres informations comme un invariant de boucle ou une indication sur la complexité, etc.
3. Le Docstring est utile pour produire un livrable de qualité comportant toutes les spécifications utiles ; pour autant ce n'est pas ce par quoi il faut commencer quand on conçoit un algorithme.

1.3 Assert

To assert en anglais signifie « affirmer, faire valoir, énoncer (comme énoncer un fait vrai) ».

Une fois la signature de la fonction posée et ses spécifications précisées dans un docstring, il peut être pertinent de spécifier une ou des contraintes particulières à vérifier sur les arguments de la fonctions, avant l'exécution de la fonction, permettant de stopper l'exécution de la fonction en renvoyant un message dans le cas où les contraintes ne sont pas satisfaites.

C'est le sens de la commande `assert` :

```
assert test booléen
```

Cette fonction ne renvoie rien si l'évaluation du test booléen renvoie `True` et renvoie le message d'erreur `AssertionError` si l'évaluation du test booléen renvoie `False`.

Prenons par exemple le cas de la division euclidienne de deux entiers. La division euclidienne de a par b (deux entiers positifs) existe si, et seulement si, $b \neq 0$. Ainsi

```
def my_divmod(a:int,b:int) -> tuple:
'''
Calcul itératif de la division euclidienne de deux entiers
Entrée : a et b deux entiers positifs avec b != 0
Sortie : deux entiers positifs q et r tels que a=bq+r
'''
assert b!=0
q,r=0,a
while r >= b:
    r = r-b
    q = q+1
return q,r
```

Listing 1 – Division euclidienne

Il peut être parfois important de tester sous la forme d'un `assert` le type d'un argument, auquel cas, on écrira `assert isinstance(argument,type)`. Par exemple : `assert isinstance(2.,list)` va renvoyer une `AssertionError` alors que `assert isinstance(2.,float)` va valider l'évaluation de l'assertion.

Toute fonction ne nécessite pas de préciser des `assert`, il faut les préciser pour mettre en avant des cas limites de la fonctions pour lesquels l'exécution ne fonctionnerait pas ou n'aurait pas de sens ou renverrait un résultat erroné, etc. Il s'agit typiquement d'éviter des divisions par zéro, des évaluation en des indices hors de la taille d'une liste, des arguments qui ne seraient pas du type souhaité.

1.4 Commentaires

Enfin, les commentaires en cours de programme sont essentiel pour la qualité du code fournit : ils doivent être succincts, efficace et doivent pointer uniquement les points algorithmiquement pertinents.

1.5 Jeu de tests

Construire un jeu de tests, consiste à sélectionner un ensemble de valeurs pour les arguments de la fonctions et à calculer la/les valeurs retournées par la fonction en ces valeurs.

Le choix des valeurs testées pour un jeu de test répond à une **stratégie**. Il s'agit de mettre en lumière un fait particulier. Cela peut être :

- tester quelques cas simples pour montrer que la fonction retourne les valeurs souhaitées dans les cas souhaités ;
- tester des valeurs produisant des erreurs ou des sorties incorrectes de la fonction.
- tester des valeurs limites ou des valeurs interdites afin d'exhiber des comportements résiduels.
- tester un nombre important de données (qui pourraient être construite de manière aléatoire par exemple) afin de mettre en lumière un comportement de nature statistique (une moyenne par exemple).
- tester des cas qui pourraient nécessiter un temps d'exécution long afin de mettre en lumière un point de complexité de la fonction.
- etc.

Une fois un jeu de tests construit, on l'écrira le plus souvent sous la forme d'`assert` :

```
assert mafonction(var1,var2) == monresultat1
assert mafonction(var3,var4) == monresultat2
...
```

2 Validation d'une fonction

Maintenant que notre fonction a été spécifiée correctement, nous voulons **prouver** son bon fonctionnement. Il s'agit de la **validation** d'une fonction.

A cette fin, deux points vont être spécifiquement étudiés :

1. dans les cas d'une fonction itérative de type `while` ou d'une fonction récursive, nous voulons **prouver la terminaison** de l'itération ou de la récursion pour toute valeur autorisée des arguments (autorisée dans les spécifications).
2. dans le cas d'une fonction itérative ou récursive, nous voulons **prouver que le résultat obtenu est correct**, à savoir qu'il est bien le résultat annoncé dans les spécifications.

2.1 Terminaison

Étant donné une fonction construite à l'aide d'une itération de type `while`, on souhaite montrer que l'exécution de la fonction produit un résultat en un nombre fini d'itérations quelles que soient les valeurs des arguments fournies dans la plage de valeurs spécifiées.

Pour ce faire, nous allons utiliser un **variant de boucle** :

Définition 2 — Étant donné une fonction itérative de type `while`. Un variant de boucle est une fonction des variables de la k-ième itération qui est strictement croissante ou strictement décroissante.

Nous montrerons que ce variant v_k possède une propriété **à partir d'un certain rang** qui implique la propriété de sortie de boucle.

Par exemple :

- ce variant v_k est à valeur entières positives strictement décroissantes : donc devient négatif à partir d'un certain rang.
- ce variant v_k est l'écart $l - u_k$ d'une suite croissante qui tend vers l donc $l - u_k$ est inférieur à ε à partir d'un certain rang.
- ce variant v_k est l'écart $\beta_k - \alpha_k$ entre deux suites adjacentes $\alpha_k \leqslant \beta_k$ donc v_k est inférieur à ε à partir d'un certain rang.

Si un variant de boucle simple est clairement identifiable, il peut être pertinent de l'écrire dans le Docstring de la fonction.

Exemples :

1. Nous avons déjà étudié un variant pour l'algorithme de recherche dichtomique d'un élément dans une liste triée.
2. Il est intéressant d'identifier un variant pour plusieurs algorithmes que nous connaissons déjà :
   — calcul de la division euclidienne de deux entiers ;
   — l'algorithme d'Euclide pour le calcul du pgcd
   — résolution de $f(x) = 0$; par la méthode de dichotomie ;
   — recherche d'un maximum dans une liste ;
   — calcul d'une somme.
3. Dorénavant, dès que nous établirons un algorithme itératif important, nous identifierons et prouverons la terminaison à l'aide d'un variant (ce sera notamment le cas, pour l'étude des algorithmes de tri).

2.2 Correction

Étant donné une fonction construite à l'aide d'une itération, on souhaite montrer que l'exécution de la fonction produit un résultat conforme à la spécification, plus précisément, nous dirons qu'une fonction est :

— **partiellement correcte** si la sortie qu'il renvoie est conforme à sa spécification indépendamment de savoir s'il termine. Autrement dit, si, quand il termine, renvoie une sortie conforme à sa spécification.
— **totalement correcte** si, quel que soit son argument conforme à sa spécification, l'algorithme termine et la sortie qu'il renvoie est conforme à sa spécification.

Pour ce faire, nous allons utiliser un **invariant de boucle** :

Définition 3 — Étant donné une fonction itérative. Un **invariant de boucle** est une propriété logique (ou prédicat) fonction des variables de la k-ième itération telle que :

— **Initialisation :** avant l'entrée dans la boucle (itération 0) cette propriété soit vraie.
— **Récurrence :** si la propriété est vraie lors de la k-ième itération, alors cette propriété est vrai lors de la $k+1$-ième.
— **Terminaison :** si la condition de sortie de la boucle s'applique, la propriété est toujours vrai une fois sorti de la boucle.

Si un invariant de boucle simple est clairement identifiable, il peut être pertinent de l'écrire dans le Docstring de la fonction.

Remarque : **invariant de boucle et traçage de valeurs.**

— Il peut être parfois difficile d'identifier un invariant de boucle. Pour ce faire, il peut être intéressant de tracer les valeurs au fur et à mesure de l'itération. Pour ce faire, ou bien on les affichera directement à l'aide de `print` ou bien on les stockera dans une liste que l'on renverra en fin de fonction.
— A contrario, une fois que l'on a démontré un invariant de boucle, il peut être intéressant de tester cet invariant au fur et à mesure de l'itération : cela peut se faire par un `assert` ou tout simplement par un `print` de l'expression booléenne associée ou encore par un simple `print` des valeurs.

Exemples :

1. Nous avons déjà étudié un invariant pour l'algorithme de recherche dichotomique d'un élément dans une liste triée.
2. Il est intéressant d'identifier un invariant pour plusieurs algorithmes que nous connaissons déjà :
   — calcul de la division euclidienne de deux entiers ;
   — l'algorithme d'Euclide pour le calcul du pgcd
   — résolution de $f(x) = 0$; par la méthode de dichotomie ;
   — recherche d'un maximum dans une liste ;

— calcul d'une somme.

3. Dorénavant, dès que nous établirons un algorithme itératif important, nous identifierons et prouverons la terminaison à l'aide d'un variant (ce sera notamment le cas, pour l'étude des algorithmes de tri).

2.3 Étude d'un exemple spécifique

On considère la fonction `mystere` suivante :

```
def mystere(a:int,b:int) -> int:
    '''
    Entree : a et b deux entiers strictement positifs
    Sortie : a*b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
          total = total + y
        x = x // 2
        y = y * 2
    return total
```

Cela ne semble pas évident au premier coup d'œil que cette fonction renvoie bien le produit qu'elle annonce dans ses spécifications. Pour ce faire, on peut se donner une idée via un traçage de le la boucle pour une valeur spécifique :

x	**y**	**total**
18	3	0
9	6	0
4	12	6
2	24	6
1	48	6
0	96	54
fin		