Cours 2 : Rappels et compléments sur les fonctions

Objectifs

— bonnes pratiques de codage,
— preuves de programme,
— récursivité.

Introduction Dans l'industrie ou la recherche, un logiciel peut représenter des codes de millions de lignes avec une responsabilité humaine vitale comme dans les transports ou l'énergie. Il est donc essentiel de produire des programmes sûrs qui sont conformes aux exigences du cahier des charges.

La communauté informatique a mis en place des propositions d'amélioration du codage en Python, les PEP pour *Python Enhancement Proposal*. En voici quelques unes :

— *Beautiful is better than ugly*
— *Explicit is better than implicit*
— *Simple is better than complex*
— *Complex is better than complicated*
— *Flat is better than nested*
— *Sparse is better than dense*
— *Readability counts...*

Dans le cadre de cet enseignement, seule la programmation fonctionnelle est abordée.

1 Règles de conception des programmes

1. *L'analyse descendante* : parce qu'un algorithme simple diminue le risque d'erreurs, on procède par raffinements successifs.
2. *La modularité* : l'analyse descendante précédente aboutit à la décomposition du programme en plusieurs modules qui contiennent chacun une série de fonctions. Quand plusieurs modules sont regroupés ensemble, on parle de bibliothèque (*library* en anglais). Chacune de ces fonctions puis de ces modules peut être testée séparément. Une conséquence de cette décomposition en modules est la transmission des données sous forme d'*arguments* pour les fonctions. On privilégiera donc les *variables locales* aux variables globales.
3. *Les spécifications et les preuves de programme* : ils précisent le cahier des charges des éléments du programme et prouve que le programme se termine et qu'il aboutit au résultat recherché.
4. *La portabilité* : pour améliorer la portabilité de ses programmes aux différents systèmes d'exploitation, machines ou versions de logiciels, il est prudent de ne pas abuser des particularités d'un langage.
5. *La lisibilité* : insérer des commentaires au sein de son code et documenter systématiquement ses fonctions.

2 Spécifications et preuves de programme

L'objectif de cette section est de répondre à deux questions lors de la mise au point d'un algorithme : la *terminaison* et la *correction*. Et pour atteindre ces objectifs, il est également nécessaire de spécifier les conditions d'exécution de l'algorithme.

Remarque 1. *Pour la terminaison, le problème ne se pose que pour les boucles conditionnelles de type* `while`.

2.1 Spécifications

Définition 1. *Une* condition *pour un fragment de programme est une propriété portant sur les variables intervenant dans le fragment de programme.*

Définition 2. *Une* précondition *est une condition supposée vérifiée au moment où débute l'exécution du fragment de programme. Elle répond à la question : « Á quelles conditions entre-t-on dans l'algorithme ? ». Elle concerne donc les* arguments *de la fonction.*

Définition 3. *Une* postcondition *est une condition qui doit être vérifiée à la fin de l'exécution du fragment de programme. Elle répond à la question : « Á quelles conditions sort-on de l'algorithme ? ».*

Exercice 1 :
Spécifier le programme suivant.

Exemple 1.
```
def division_euclidienne(a, b) :
    '''renvoie le quotient et le reste de la division euclidienne de a par b'''
    q = 0
    r = a
    while r >= b :
        q = q + 1
        r = r - b
    return (q, r)
```

2.2 Terminaison d'une boucle

Pour démontrer que la boucle conditionnelle de cette fonction se termine, on va trouver un *variant de boucle*.

Définition 4. *Un* variant de boucle *est une quantité c qui vérifie les deux conditions suivantes :*
- $c \in \mathbb{N}$,
- *c décroit strictement à chaque itération. Comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations.*

Exercice 2 :
Montrer que les boucles conditionnelles de l'exemple 1 et des deux exemples ci-dessous se terminent en déterminant pour chacun de ces exemples un *variant de boucle*.

Exemple 2.
```
def PGCD(a, b) :
    '''Recherche du PGCD de deux entiers a et b'''
    u = a
    v = b
    while u != v :
        if u > v :
            u = u - v
        else :
            v = v - u
    return (u)
```

Exemple 3.
```
# Un programme a tester
c = 0
while p > 0 :
    if c == 0 :
        p = p - 2
        c = 1
    else :
        p = p + 1
        c = 0
```

Exercice 3 :
En mathématiques, on appelle *suite de Syracuse* (du nom d'une université américaine où le problème a été posé pour la première fois), ou *suite de Collatz* (Lothar Collatz, mathématicien allemand du XXe siècle qui énonça la conjecture associée) une suite d'entiers naturels définie de la manière suivante.

On part d'un nombre entier plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur. On s'arrête quand on arrive au chiffre 1.

Si on arrive à la valeur 1, la suite se répète indéfiniment selon un cycle donné appelé *cycle trivial*.
- Écrire une fonction affichant les termes de la suite de Syracuse d'un entier N donné.
- De même que pour les exemples précédents, montrer que votre programme se termine.

2.3 Conformité ou correction d'une boucle

Pour démontrer qu'un algorithme est correct, c'est à dire qu'il calcule bien ce que l'on attend, on établit un *invariant de boucle.*

Définition 5. *Un* invariant de boucle *est une propriété qui :*
- *est vérifiée avant d'entrer dans la boucle ;*
- *si elle est vérifiée avant une itération, alors elle est vérifiée après celle-ci ;*
- *permet d'en déduire que le programme est correct.*

Exercice 4 :
Montrer que les algorithmes des deux premiers exemples sont corrects.

2.4 Conclusion

En pratique, exhiber les conditions, un variant de boucle et un invariant sera suffisant pour prouver la terminaison et la correction d'une boucle. Et pour vos propres algorithmes, n'oubliez pas que *Explicit is better than implicit.*

Enfin, ce problème de terminaison peut devenir un problème mathématique complexe. L'objectif pour nous est de s'assurer que nos propres algorithmes se terminent et aboutissent au résultat recherché. Ces spécifications et preuves de programme doivent touver naturellement leurs places dans l'écriture de vos programmes, en utilisant la documentation de fonction et les commentaires.

Exercice 5 :
Écrire une fonction qui recherche l'indice de la première occurrence d'un élément dans une liste non triée et qui renvoie `None` sinon. Spécifier et documenter rigoureusement votre fonction.

3 Récursivité

Définition 6 (Une mauvaise définition de ce type de fonction)**.** *Une* fonction récursive *est une fonction récursive.*

Définition 7 (Une bonne définition de ce type de fonction)**.** *On appelle* fonction récursive *une fonction qui comporte un appel à elle-même.*

Plus précisément, [1] page 33, une fonction récursive doit respecter ce qu'on pourrait appeler les « trois lois de la récursivité » :

1. une fonction récursive contient un cas de base ;
2. une fonction récursive doit modifier son état pour se ramener au cas de base ;
3. une fonction récursive doit s'apppeler elle-même.

Une fonction mathématique définie par une relation de récurrence (et une condition initiale, bien entendu), peut de façon naturelle être programmée de manière récursive.

Exemple 4. *Pour un réel x fixé, x^n peut se définir, comme fonction de n, par récurrence à partir des relations :*

$$x_0 = 1 \text{ et } x^n = x \cdot x^{n-1} \text{ si } n > 1$$

Le programme en Python s'écrit :

```
def puissance(x, n) :
    '''renvoie x^n'''
    if n > 0 :
        return ( x * puissance(x, n - 1) )
    else :
        return (1)
```

Modifions légèrement ce programme, pour bien en comprendre l'exécution :

```
def puissance(x, n) :
    '''renvoie x^n'''
    if n == 0 :
        return (1)
    else :
        print( '--' * n + '> appel de puissance ({}, {})'.format(x, n - 1) )
        y = x * puissance(x, n - 1)
        print( '--' * n + '> sortie de puissance ({}, {})'.format(x, n - 1) )
        return ( y )

puissance(2, 5)
```

```
----------> appel de puissance (2, 4)\\
--------> appel de puissance (2, 3)\\
------> appel de puissance (2, 2)\\
----> appel de puissance (2, 1)\\
--> appel de puissance (2, 0)\\
--> sortie de puissance (2, 0)\\
----> sortie de puissance (2, 1)\\
------> sortie de puissance (2, 2)\\
--------> sortie de puissance (2, 3)\\
----------> sortie de puissance (2, 4)
```

La machine applique la règle : `puissance(x, n) = x * puissance(x, n - 1)` *tant que l'exposant est différent de 0, ce qui introduit des calculs intermédiaires jusqu'à aboutir au cas de base :* `puissance(x, 0) = 1`*. Les calculs en suspens sont alors achevés dans l'ordre inverse jusqu'à obtenir le résultat final.*

Exercice 6 (Calcul des termes d'une suite définie par récurrence, [2] page 130) **:**
Programmer un algorithme qui permette de calculer les termes de la suite (u_n) définie ci-dessous.

$$\begin{cases} u_0 = 2 \\ u_n = \frac{1}{2} \cdot \left(u_{n-1} + \frac{3}{u_{n-1}} \right) \end{cases}$$

Exercice 7 (Conversion entier → binaire) **:**
Écrire une fonction réalisant cette conversion en utilisant une fonction itérative, puis en utilisant une fonction récursive. La fonction renverra le résultat sous la forme d'une chaîne de caractères.

Références

[1] Alexandre Casamayou-Boucau et Pascal Chauvin et Guillaume Connan. *Programmation en Python pour les Mathématiques.* Dunod, collection Sciences Sup, 2012.

[2] Benjamin Wack et al. *Informatique pour tous en classes préparatoires aux grandes écoles.* Eyrolles, 2013.