Chapitre 2 : Ecriture et analyse d'un programme

1 Instructions

1.1 Expression et affectation

En Python expression et affectation sont deux instructions particulières.

- Une expression est utilisée pour calculer une valeur ou appeler une fonction qui va effectuer une tâche sans rien renvoyer (comme print par exemple). Une expression est composée de noms (ou identifiants), de nombres, de chaines de caractères, d'éléments séparés par des signes de ponctuation, entourés de parenthèses, de crochets, d'accolades et d'opérateurs. Une expression a une valeur.
- Une affectation est utilisée pour lier un nom à une valeur ou pour modifier un élément d'un objets mutable comme une liste. L'instruction affecter est représentée par le signe =. La synthaxe est nom=expression ou nom[i]=expression.

1.2 Instruction

Une instruction est un morceau de code minimal qui produit un effet. Elle est exécutée par une machine. Une instruction simple peut s'écrire sur une seule ligne. On peut mettre plusieurs instructions par ligne, séparées par des ;

```
a=2;a+=2;print(a);
```

Une instruction composée est une instruction sur une ligne terminée par : suivie par une ou plusieurs instructions indentées (par exemple if, else, while, for, def...)

A part les expressions, les instruction n'ont pas de valeur, c'ets un choix de conception du langage Python qui a pour conséquence la levée de certains bugs peu visibles lors de la lecture du code :

```
if x=1:
    x=2
```

Ici if doit être suivi d'une expression et donc d'une valeur booléenne (True ou False), je me suis trompé et j'ai écrit une instruction d'affectation au lieu d'un test avec ==. Python refusera d'exécuter ce programme,

En revanche si on avait décidé que l'affectation renvoyait une valeur (par exemple en plus d'affecter x à 1 je renvoie True pour signifier que tout s'est bien passé et False si l'affectation est un échec) alors l'instruction x=2 serait exécutée par l'ordinateur sans que la condition soit vérifiée.

En effet, avec Python, if doit être suivi d'un booléen mais va convertir n'importe quelle valeur en booléen. Cela permet de gagner un peu de temps comme avec le code suivant :

```
if x%2:
    print("x est impair")
```

1.3 Effet de bord

On dit qu'une fonction a un effet de bord si elle modifie une valeur utilisée ailleurs que dans la fonction (comme un de ses paramètres ou une variable globale).

```
Point_de_Vie=3

def touche_héros(k):
    Point_de_Vie-=k;

def touche_héros2(k)
    return Point_de_vie-k;
```

Ici la fonction touche modifie la variable Point_de_vie définie avant : elle a donc un effet de bord. La deuxième fonction elle renvoie les nouveaux points de vie de notre héros mais sans les modifier, libre alors à l'utilisateur de remplacer l'ancienne valeur par la nouvelle.

Remarque : il parait qu'une fonction sans effet de bord est plus simple à tester. L'important est de garder à l'esprit ce que l'on fait la première fonction calcule le nouveau total de point de vie et l'affecte, elle ne renvoie rien. La seconde calcule le total de point de vie et le renvoie, il faudra ensuite l'affecter mais on est sur qu'elle ne fait rien d'autre que le calcul demandé.

Exercice :

La fonction suivante a-t-elle un effet de bord ?

```
def f(liste):
    liste=2*liste+m
    n=len(liste)
    liste[n-1][0]=4
    return liste
```

Exercice :

La fonction suivante a-t-elle un effet de bord ?

```
dico={0:0,1:1}

def fibo(n):
    for i in range(2,n+1):
        dico[i]=dico[i-2]+dico[i-1]
    return dico[n]
```

2 Spécification et annotation

2.1 Spécification d'une fonction

Une spécification permet d'informer un utilisateur de ce que va faire la fonction. On peut préciser les contraintes sur les paramètres ou les limites du résultat (précision, taux d'erreur...). On la met au début du corps de la fonction entre des triple guillemets.

```
def carre(x)
    """Calcule le carré d'un nombre, ne pas utiliser avec une liste"""
    return x**2
```

On peut avoir accès à la spécification d'une fonction (je rappelle que si la fonction est dans un module, vous n'avez à priori pas son code sous les yeux) via la fonction help

```
>>> help(carre)
Help on function carre in module __main__:

carre(x)
    Calcule le carré d'un nombre, ne pas
 utiliser avec une liste
```

La spécification d'une fonction ne change rien à l'exécution du programme par Python, tout comme les commentaires (voir plus tard), elle est destinée à l'utilisateur. En plus de la spécification, des choix pertinents pour les noms de vos fonctions et de vos variables sont indispensable pour que le code puisse être relu par un programmeur tiers ou meme par vous après quelques mois.

Une spécification est un genre de contrat entr ele programmeur et l'utilisateur : on garentit le bon fonctionnement de la fonction sous réserve que l'utilisateur lui passe des arguments du bon type, de la bonne valeur etc...

```
def somme(x,y):
    """calcule la somme de 2 entiers"""
    return x+y
```

Ce programme va fonctionner avec deux chaines de caractères mais ne renvoie pas des résultats prévus par l'auteur du programme :

```
>>> somme("2","3")
'23'
```

2.2 Annotations et commentaires

Pour que le code puisse être relu par l'auteur ou par une personne extérieure, il est très utile de commenter certaines lignes, pour expliquer ce que l'on fait à l'intérieur d'une fonction, ou pour expliquer ce que fait un bloc d'instructions. Pour cela on utilise # suivi du commentaire.

Tout ce qui suit # sur la ligne n'est donc pas pris en compte par Python et visible uniquement par les gens qui ont accès au code source.

```
def multiplieliste(l,m):
    return [i*m for i in l] # ici on forme une liste à partir des éléments de l
#que l'on multiplie par i, on utilise une méthode par compréhension  pour avoir un code court
```

Il est indispensable de mettre des commentaires si vous travaillez sur un projet qui dure plus de deux semaines ou si vous êtes plusieurs à travailler sur un même projet. Cela représente quasi 100% des situations professionnelles...

3 Assertion

Une assertion est une instruction qui permet d'arrêter le programme en cas de mauvaise utilisation.

On affirme qu'une propriété est vraie en écrivant le mot clef assert suivi d'une expression booléenne étant vraie ou fausse. Si elle est vraie le programme ne fait rien mais si elle est fausse, il s'arrête et le message d'erreur AssertError.

```
def inverse(x):
    assert(x!=0)
    return(1/x)
```

```
>>> inverse(2)
0.5
>>> inverse(0)
Traceback (most recent call last):
  File "<string>", line 449, in runcode
  File "<interactive input>", line 1, in
<module>
  File "<module1>", line 25, in inverse
AssertionError
>>>
```

Les assertions, en plus d'être un outil de contrôle pour être certain qu'une fonction est utilisée comme on le souhaite permettent de debugger efficacement un programme, parfois une fonction est appelée par une autre fonction, elle même appelée par une autre fonction et on peut rapidement perdre la trâce du problème.

Exercice :

La suite de Syracuse est définie comme ceci : le premier terme est un entier, si un terme est pair le suivant est sa moitié, s'il est impair, le suivant est le triple plus 1. Mathématiquement : $u_0 = n \in \mathbb{N}$ et $u_{k+1} = \begin{cases} \frac{u_k}{2} & \text{si } u_k \text{ pair} \\ 3u_k + 1 & \text{sinon} \end{cases}$. On code la fonction suivante :

```python
def syracuse(n):
    liste=[n]
    while n!=1:
        if n%2:
            n=3*n+1
        else:
            n=n//2
        liste.append(n)
    return liste
```

1. Ajouter une assertion pour vérifier que n est strictement positif.
2. Si n est un flottant non entier, il peut y avoir un problème. Proposez une manière d'éviter ce problème via une assertion ou en modifiant une unique ligne du programme.

Exercice :

Ecrire une fonction prenant en entrée deux listes l_1 et l_2 de même longueur et qui renvoie la liste formée en faisant les additions terme à terme des listes l_1 et l_2. On vérifiera la longueur des listes avec une assertion.

4 Terminaison et correction d'un algorithme

Dans cette section, on se demande si un algorithme termine (qu'il ne continue pas de calculer à l'infini) et lorsqu'il termine s'il renvoie le bon résultat. Comme en math ou on démontre qu'un théorème est vrai avant de l'utiliser à l'infini, on montre qu'un algorithme est correct avant de pouvoir l'utiliser à l'infini.

4.1 Terminaison

Pour montrer qu'un algorithme termine , il faut montrer que chaque bloc élémentaire termine ! Or, les boucles for et les instructions conditionnelles terminent forcément. Le seul souci pourrait venir d'une boucle while ou d'uun algorithme récursif qui s'appelle à l'infini.

Etudions le code suivant :

```python
def fonction(n):
    while n!=0:
        n=n-1
```

- Si on entre un entier naturel la fonction termine
- Si on rentre un entier négatif où un nombre à virgule, il continue à l'infini.

Donc cet algorithme peut ne pas terminer

Etudions l'algorithme d'exponentiation rapide suivant (version non récursive) :

```python
def expo_rapide(x,n):
    y=x
    z=1
    m=n
    while m>0:
        q,r=m//2,m%2 # quotient et reste dans la division euclidienne de m par 2.
        if r==1:
            z*=y # on multiplie z par y, le résultat est affecté à z.
        y*=y # on met y au carré, le résultat est affecté à y.
        m=q
        #Inv
        #Inv
    return z
```

Si on regarde m, à chaque tour de boucle il est divisé par 2, c'est donc une suite d'entier naturels strictement décroissante. Elle atteint forcément 0 en un temps fini et l'algorithme termine.

Exercice :

```
x=1
while x/2>0:
    x=x/2
```

Prouver la terminaison et donner la valeur finale de x.

4.2 Variant de boucle

Dans l'exemple précédent, on montre que l'algorithme se termine via une quantité qui décroit jusque 0. C'est un variant de boucle.

Définition : Un variant de boucle est une quantité positive, à valeurs dans $\mathbb{N}$, dépendant des variables de la boucle, qui décroît strictement à chaque passage dans la boucle.

Le variant de boucle est en général assez évident à trouver

Exemple :

```
s=0
while L!=[]: #Tant que L est non vide
    s+=L[0]
    L=L[1:] #L[1:] est la liste constituée de tous les éléments de L, sauf le premier.
```

Quand est-ce que la boucle s'arrête ?

Quel sera le variant de boucle ?

Est-ce que le programme termine ?

4.3 Correction

Pour montrer qu'un algorithme est correct, il s'agit de montrer que quels que soient les paramètres vérifiant sa spécification, l'action de l'algorithme correspond à ce qui est attendu. Analyser les boucles for et while est essentiel, car l'action de ces boucles n'est pas forcément évidente en première lecture. La notion essentielle pour montrer la correction des boucles est celle d'invariant de boucle.

Définition : Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.

Exemple : Dans l'algorithmme suivant (pas très utile certes)

```
a=0
b=7
while b>0:
    b=b-1
    a=a+1
```

$a+b=7$ durant toute l'exécution. Lorsque $b=0$ alors $a=7$ et j'ai donc transféré la valeur de b vers a (encore une fois le but d'un tel algo n'est pas le sujet ici).

Correction d'une boucle while :

Reprenons l'algorithme d'exponentiation rapide :

Après une (plus ou moins longue) réflexion, on remarque que $y^m \times z = x^n$

Vérifions que c'est vrai à chaque tour de boucle.

Ainsi lorsque $m = 0$ alors $z = x^n$ et c'est gagné.

Exercice :

```
def somme(L,x):
    s=0
    for i in range(len(L)):
        s+=L[i]
    return s
```

1. Que calcule le programme suivant ?
2. Trouver un invariant de boucle.

Exercice :

```
def f(a,b):
    s,k=a,b
    while k>0:
        s=s+1
        k=k-1
    return s
```

Vrai ou faux ?

1. $s + k = a + b$ est un invariant de boucle
2. la valeur finale de k est 1
3. $k \geq 0$ est un invariant de boucle
4. le résultat renvoyé est la somme $a + b$

5 Tests d'un programme

5.1 Bugs

Si une erreur est commise dans l'écriture d'un programme, elle peut avoir des conséquences désastreuses sur l'exécution du programme : le programme ne finit pas, renvoie une erreur, fait planter l'ordinateur, etc...

Parfois c'est une erreur humaine :

```
def exemple_erreur(l):
    n=len(l)
    for k in range(n+1):
        l[k]=3
    return l
```

Où est l'erreur ?

Parfois l'erreur vient de l'ordinateur et des nombres qu'il est capable de traiter :

```
def prodracine(x,y):
    return (x*y)**0.5

def prodracine2(x,y):
    return (x**0.5)*(y**0.5)
```

```
>>> prodracine(10,10)/prodracine2(10,10)
0.9999999999999998
>>> prodracine(10,10)
10.0
>>> prodracine2(10,10)
10.000000000000002
>>> prodracine(1e155,1e155)/prodracine2(1e155,1e155)
inf
```

Il est donc fondamental de tester notre programme avec des valeurs pour lesquelles on connait le résultat en sortie, ou bien une série de valeurs aléatoires, ou bien les valeurs limites...

5.2 Jeu de test

Un programme ne fonctionne pas toujours du premier coup et il est indispensable de le tester avec des valeurs simples, voir même de tester des morceaux du programme à part ou en cours d'écriture pour vérifier qu'ils fonctionnent comme convenu.

Il faut donc construire un jeu de tests pertinent :

- des cas simples,
- des valeurs extrêmes ou cas interdits,
- un grand nombre de données,
- des cas qui pourraient prendre beaucoup de temps,

Exemple :

```
def pgcd(a,b):
    while b!=0:
        q,r=a//b,a%b
        a=b
        b=r
    return a
```

Je propose le jeu de tests suivant :

```
>>> pgcd(3,9)
3
>>> pgcd(9,3)
3
>>>
>>> pgcd(20,30)
10
>>> pgcd(30,20)
10
>>> pgcd(30,0)
30
>>> pgcd(0,10)
10
>>> pgcd(0,0)
0
>>> pgcd(1.5,5.7)
8.881784197001252e-16
```

En fonction du programme, on réfléchit à des tests pertinents ici pour pgcd(m, n) on peux faire $m \geq n, n \geq m, n = m, n = 0, m = 0, n = m = 0$ etc

Dans le jeu de test effectué, avec des nombres à virgules on remarque un résultat complètement incohérent mais en même temps le pgcd n'existe pas donc il est sage de mettre une assertion pour vérifier le type des entrées...

Exercice :

1. Ecrire une fonction permute qui échange le premier et le dernier élément d'une liste. (on ne cherchera pas à traiter des cas particulier).
2. Ecrire un jeu de test pour cette fonction. (liste de quelques éléments, 1 élément, vide)
3. Corriger votre programme pourqu'il passe tous les tests.

6 Complexité d'un algorithme

6.1 Recherche du meilleur algorithme

On cherche à minimiser le nombre d'opérations effectuées par la machine. Ce nombre dépend de la taille de l'entrée de l'algortithme.

Exemple :

Voici les règles pour calculer le coût d'un algorithme.

- Les opérations (addition, multiplication, évaluation, affectation) sont peu coûteuses, elles comptent pour 1.
- Si p et q sont des instructions, le coût de p et q est la somme des coûts de p et de q.
- Le coût de (if b : p else : q) est le max du coût de p et du coût de q auquel on rajoute le coût de l'évaluation du booléen b.
- Le coût de (for k in range(n) : p) est $n\text{cout}(p)$ si le coût de p ne dépend pas de k, sinon c'est $\sum \text{cout}(p(k))$
- While b : p est plus compliqué car on ne maitrise aps forcémment le nombre d'itérations de la boucle. On peut parler du cas le meilleur et du cas le pire

Exercice :

6.2 Notation O

On étudie pas le nombre exact d'opérations faites dans un algorithme mais un ordre de grandeur, ce qui simplifiera pas mal les calculs. Par exemple si on fait $n^2 - 3n$ opérations pour une entrée de taille n alors on dit que la complexité est de l'ordre de n^2. Pourquoi est-ce plus simple ?

Exemple :

Définition : On dit qu'un algorithme a une complexité en $O(f(n))$ si son coût est compris (pour n grand) entre $C_1 f(n)$ et $C_2 f(n)$ avec C_1 et C_2 deux constantes.

Exemple :

Pour votre culture générale, voici e temps d'exécutions d'algorithmes en fonction de leur complexité pour un processeur de 1Ghz et une entrée de taille un million (10^6).

Complexité	Nom	Temps d'exécution	Remarque
$O(1)$	constant	1ns	Dépend pas de n
$O(\log(n))$	logarithmique	10ns	idéal
$O(n)$	linéaire	1ms	
$O(n^2)$	quadratique	15min	horrible si n est grand
$O(n^3)$	cubique	30 ans	
$O(2^n)$	exponentiel	10^{300000} ans	inutilisable

Exercice : 7

6.3 Nuances de complexité

On l'a vu en fonction de certains tests (if, boucle while) certains calculs compliqués peuvent ou pas être effectués. On va donc étudier le pire des cas.

- Cas le pire /cas le meilleur

Exemple :

- Complexité en espace :

Il s'agit de l'espace mémoire nécessaire pour faire fonctionner l'algorithme. Si on reprend l'exemple de l'algo qui renvoie la liste des diviseurs la complexité en espace est au pire $2\sqrt{n}$ le nombre maximal de diviseurs.

Exercice : 8

Exercice : 8

6.4 Exemple de la recherche dans un tableau

6.4.1 Cas de base

On écrit un algorithme qui prend en entrée une liste l et un nombre n et qui renvoie True si $n \in l$ et False sinon.

6.4.2 Si le tableau est trié

Exemple :