Chapitre 2 :
Méthodes de programmation

I Introduction aux algorithmes

Un algorithme est une suite finie d'instructions qui peuvent être exécutées les unes à la suite des autres pour résoudre un problème.

Un algorithme ne dépend ni du langage de programmation, ni de la machine que laquelle il est exécuté.

1.1 Branchement conditionnel

Une instruction conditionnelle ou test permet d'exécuter un bloc d'instructions si une certaine condition est vérifiée. Il s'agit ainsi d'une instruction du type : Si condition alors instruction1 sinon instruction2.

La syntaxe est la suivante :

```
if condition1 :
    bloc d'instructions 1
elif condition2 :
    bloc d'instructions 2
elif condition3 :
    bloc d'instructions 3
    ⋮
else :
    bloc d'instructions
```

- La condition doit être une expression renvoyant un booléen. Si la condition est vérifiée (expression booléenne = True), alors le bloc d'instructions 1 est exécuté sinon (expression booléenne = False), les instructions du bloc 2 sont exécutées.
- Le début d'un bloc d'instructions est défini par un double point : le corps du bloc est alors indenté d'un nombre d'espaces fixes (quatre par défaut), et la fin de l'indentation marque la fin du bloc.
- Le mot clé elif est une contraction de else if.
- Les différentes conditions ne sont pas nécessairement disjointes, l'ordre des tests est donc important. Les conditions sont vérifiées successivement (i.e les unes après les autres). Dés qu'une condition est vraie, le programme exécute le bloc d'instructions associé puis le programme s'arrête. Il ne se préoccupe pas des conditions suivantes.
- La partie else est optionnelle.

➪ **Exemple 1 :** Les programmes suivants n'affichent pas la même chose :

```
1 def fonction1(x) :
2     if x>2 :
3         return x+1
4     elif x>0 :
5         return x+3
6     else :
7         return x
```

```
>>> fonction1(3)


.....
```

```
1 def fonction2(x) :
2     if x>0 :
3         return x+3
4     elif x>2 :
5         return x+1
6     else :
7         return x
```

```
>>> fonction2(3)


.....
```

➪ **Exemple 2 :** Les programmes suivants n'affichent pas la même chose :

```
1 def fonction3(a,b,c) :
2     x=a
3     if a<=b :
4         x=b
5     if x<=c :
6         x=c
7     return x
```

```
1 def fonction4(a,b,c) :
2     x=a
3     if a<=b :
4         x=b
5     elif x<=c :
6         x=c
7     return x
```

```
>>> fonction3(2,4,6)


.....

>>> fonction3(5,3,1)


.....
```

```
>>> fonction4(2,4,6)


.....

>>> fonction4(5,3,1)


.....
```

fonction3(a,b,c) renvoie .....

fonction4(a,b,c) renvoie .....

1.2 Boucles inconditionnelles

Une boucle inconditionnelle (ou boucle for) permet de répéter un bloc d'instructions un nombre déterminé de fois.
La syntaxe d'une boucle for est la suivante :

```
for compteur in iterable :
    instruction_1
        ⋮
    instruction_n
```

- Python exécutera les instructions 1 à n pour chaque valeur du compteur. Le compteur avance automatiquement à chaque passage dans la boucle.
- L'indentation est de nouveau essentielle pour que Python sache où le bloc d'instructions s'arrête.
- Un itérable est un objet dont on peut parcourir les valeurs.

Pour l'instant, nous utiliserons l'itérable range :
La fonction range permet de générer dynamiquement une suite sur laquelle on va pouvoir itérer. La fonction range prend entre 1 et 3 arguments : range((Nbe_de_depart),Nbe_final_exclu,(pas)).
Dans cette fonction, l'argument Nbe_de_depart est optionnel, sa valeur par défaut vaut 0 et l'argument pas est aussi optionnel, sa valeur par défaut est 1.
Ainsi range(i,j) crée la séquence des entiers de i à $j-1$ et range(j) crée la séquence des entiers de 0 à $j-1$.

➪ **Exemple 3 :**

```
1 def fonction5(n) :
2     s=0
3     for i in range(n+1) :
4         s+=i
5     return s
```

fonction5(n) renvoie :

$$\sum_{i=0}^{n} i.$$

1.3 Boucles conditionnelles

Une boucle conditionnelle (boucle while) est utilisée pour répéter un bloc d'instructions tant qu'une condition est vérifiée.
La syntaxe générale d'une boucle while est :

```
while condition :
    bloc d'instructions
```

- En arrivant devant cette boucle, Python évalue si la condition est vraie. Si elle l'est, il exécute le bloc d'instructions. Il réévalue ensuite si la condition est vraie ou non, et continue d'exécuter le bloc d'instructions tant que cette condition est vraie. Là encore, l'indentation est essentielle pour que Python comprenne exactement où s'arrête le bloc d'instructions à exécuter.
- Si la condition n'est jamais vérifiée, il n'y a pas d'erreur, le bloc d'instructions n'est jamais parcouru.
- Si la condition est toujours vérifiée, la boucle sera infinie et le programme ne s'arrêtera jamais. Il faut donc vérifier que la condition a une chance de devenir fausse. Pour interrompre l'exécution du script, on peut utiliser CTRL+I ou l'icône interrompre (éclair jaune au dessus de la console).
- Les conditions peuvent être reliées par les opérateurs logiques or, and.
- Si la condition fait apparaître un compteur, il ne faut pas oublier de faire avancer le compteur.
- Si on connaît le nombre d'itération à effectuer, on utilise plutôt une boucle for. Sinon, on utilise une boucle while.
- Il est toujours possible de remplacer une boucle for par une boucle while mais ce n'est pas toujours pertinent.

➪ **Exemple 4 :**

```
1 def fonction6(n) :
2     s,i=0,1
3     while s<n :
4         s+=i
5         i+=1
6         print(s,i)
7     return i-1
```

La ligne 6 est un balisage qui permet de voir le comportement de la fonction. Cette ligne sera supprimée lorsque la fonction sera validée.

On teste la fonction :

```
>>> fonction6(16)
1   2
3   3
6   4
10   5
15   6
21   7
6
```

La fonction renvoie le plus petit entier k tel que :

$$\sum_{i=1}^{k} i \geq n.$$

➪ **Exemple 5 :**

```
1 def fonction7(n) :
2     i=-1
3     while i<0 or i>n :
4         i=eval(input("Entrez un nombre entre 0 et n :"))
5     print(i)

>>> fonction7(20)
Entrez un nombre entre 0 et n : 25
Entrez un nombre entre 0 et n : -9
Entrez un nombre entre 0 et n : 2
2
```

1.4 Exemple classique

On considère la suite définie par :

$$u_0 = 1 \text{ et : } \forall n \in \mathbb{N}, u_{n+1} = 2u_n + 1.$$

La fonction suivante prend comme argument n et renvoie u_n.

```
1 def suite(n) :
2     u=1
3     for i in range(1,n+1) :
4         u=2*u+1
5     return u
```

On remarquera que les instructions de la boucle ne dépendent pas de i. Ainsi, la valeur de i n'a pas d'importance mais c'est le nombre de passages dans la boucle qui est important.

La fonction suivante prend comme argument m et renvoie le plus petit n tel que $u_n \geq m$.

```
1 def seuil(m) :
2     u=1
3     n=0
4     while u<m :
5         u=2*u+1
6         n+=1
7     return n
```

Contrairement à la fonction précédente, il faut introduire un compteur n pour connaître la valeur de l'indice. On peut montrer que $\lim u_n = +\infty$, ainsi il va bien exister un indice n tel que $u_n \geq m$ donc la boucle while n'est pas infinie.

II Programmation

2.1 Assertion

Lorsqu'on écrit un programme, celui-ci doit fonctionner pour des données vérifiant des critères particuliers. Par exemple, si on écrit une fonction qui vérifie si un entier est pair ou non, il faut que l'utilisateur entre un entier afin que la fonction renvoie un résultat valable.

Il est possible de faire un test dès le début de l'exécution afin de vérifier si un critère est vérifié. Pour cela, on utilise la commande assert. Si le test n'est pas vérifié, une erreur d'assertion sera renvoyée.

➪ **Exemple 6 :**

```
def pair(n) :
    assert type(n)==int
    if n%2==0 :
        return True
    else :
        return False


>>> pair(2.5)
Traceback (most recent call last) :
...
assert type(n)==int
AssertionError
```

Dans la majorité des énoncés, on introduit une spécification, par exemple : "Ecrire une fonction qui prend comme argument un nombre entier et qui renvoie ...". Dans ce cas, il est inutile de faire une assertion, il suffit d'écrire une fonction qui a le bon comportement si l'utilisateur rentre un argument entier.

2.2 Instruction break

L'instruction break permet de terminer l'exécution d'une boucle (while ou for). Les instructions qui suivent la boucle sont alors exécutées.

➪ **Exemple 7 :**

```
for i in range(1000) :
    if i>2 :
        break
    print(i)

0
1
2
```

.

➪ **Exemple 8 :**

```
x=10
while x<=100 :
    if x==30 :
        break
    print(x)
    x+=10

10
20
```

2.3 Balisage

Afin de comprendre ou de corriger les erreurs dans un programme, on peut faire un balisage, c'est-à-dire afficher un message à l'intérieur d'une boucle afin de voir l'action de chaque itération.

➪ **Exemple 9 :**

```
def balise(n) :
    s=0
    for i in range(n) :
        if i**2<=n :
            s+=i**3
        print('i vaut',i,'et s vaut',s)
    return s


>>> balise(5)
i vaut 0 et s vaut 0
i vaut 1 et s vaut 1
i vaut 2 et s vaut 9
i vaut 3 et s vaut 9
i vaut 4 et s vaut 9
9
```

2.4 Tests

Pour vérifier si un programme fonctionne, il faut le tester. On testera les programmes sur plusieurs valeurs qui correspondent aux différents cas possibles.

➪ **Exemple 10 :** Dans cet exemple, on cherche à écrire une fonction qui renvoie le maximum de trois nombres.

```
def FauxMax(a,b,c) :
    x=a
    if a<=b :
        x=b
    elif x<=c :
        x=c
    return x


>>> FauxMax(5,2,1)
5
>>> FauxMax(1,2,5)
2
```

Le test avec (5,2,1) ne suffit par pour voir que la fonction est fausse car on a pris le cas particulier d'une liste triée de façon décroissante. Le test avec (1,2,5) montre que la fonction ne renvoie pas le résultat voulu.

III Coût d'un algorithme

Le coût d'un algorithme correspond au nombre d'opérations élémentaires à réaliser. Il s'agit d'une notion théorique, appelée complexité, qui sera étudiée au second semestre. Afin d'introduire cette problématique, nous allons mesurer le temps d'exécution des fonctions. Pour cela, on utilisera la fonction time du module time. Cette fonction donne le temps écoulé depuis le 1er janvier 1970. Attention, le résultat renvoyé dépend de l'ordinateur utilisé. Ainsi, l'interprétation des résultats doit se faire par comparaison de résultats sur le même ordinateur.

3.1 Branchement conditionnel

On étudie la fonction qui donne la parité d'un entier.

```
def pair(n) :
    tps=time.time()
    assert type(n)==int
    if n%2==0 :
        print("temps d'execution :",time.time()-tps)
        return True
    else :
        print("temps d'execution :",time.time()-tps)
        return False


>>> pair(10)
temps d'execution : 0.0
True
>>> pair(10**9)
temps d'execution : 0.0
True
```

On voit que le temps d'exécution ne dépend pas de la taille de la variable, on parle de coût constant.

3.2 Boucles inconditionnelles

• La fonction suivante renvoie : $\sum_{i=1}^{n} i^2$.

```
def unfor(n) :
    tps=time.time()
    s=0
    for i in range(1,n+1) :
        s+=i**2
    print("temps d'execution :",time.time()-tps)
    return s


>>> unfor(10**5)
temps d'execution : 0.0305023193359375
333338333350000
>>> unfor(2*10**5)
temps d'execution : 0.061917781829833984
2666686666700000
```

On remarque que, lorsque n est multiplié par 2, le temps d'exécution est aussi multiplié approximativement par 2. On parle de coût linéaire. Cela correspond au passage dans une boucle for de longueur n.

• Les deux fonctions suivantes renvoient : $\sum_{i=1}^{n}\sum_{j=1}^{n} i.j^2 = \left(\sum_{i=1}^{n} i\right)\left(\sum_{j=1}^{n} j^2\right)$.

```
def deuxfor1(n) :
    tps=time.time()
    s=0
    for i in range(1,n+1) :
        for j in range(1,n+1) :
            s+=i*j**2
    print("temps d'execution :",time.time()-tps)
    return s

def deuxfor2(n) :
    tps=time.time()
    s=0
    t=0
    for i in range(1,n+1) :
        s+=i
    for j in range(1,n+1) :
        t+=j**2
    print("temps d'execution :",time.time()-tps)
    return s*t


>>> deuxfor1(10**3)
temps d'execution : 0.39001989364624023
167083666750000
>>> deuxfor1(10**4)
temps d'execution : 39.09942054748535
166708336666675000000
>>> deuxfor2(10**3)
temps d'execution : 0.0
167083666750000
>>> deuxfor2(10**4)
temps d'execution : 0.0
166708336666675000000
>>> deuxfor2(10**5)
temps d'execution : 0.04631972312927246
16667083336666675000000000
>>> deuxfor2(10**6)
temps d'execution : 0.4212532043457031
1666670833336666666750000000000
```

On remarque que :

- Pour la fonction deuxfor1 : en multipliant n par 10, le temps le temps d'exécution est multiplié approximativement par 100. On parle de coût quadratique. C'est le cas des boucles imbriquées.
- Pour la fonction deuxfor2 : en multipliant n par 10, le temps le temps d'exécution est multiplié approximativement par 10. On a encore un coût linéaire. C'est le cas des boucles successives.
- Le coût linéaire est bien meilleur que le coût quadratique. Si c'est possible, on préfèrera les boucles successives aux boucles imbriquées.

3.3 Boucles conditionnelles

La fonction suivante renvoie le plus petit n tel que $u_n \geq m$ avec : $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = u_n + 2$ et affiche le temps d'exécution.

```
def seuil(m) :
    tps=time.time()
    u=1
    n=0
    while u<m :
        u=u+2
        n+=1
    print("temps d'execution :",time.time()-tps)
    return n


>>> seuil(10**6)
temps d'execution : 0.03078150749206543
500000

>>> seuil(2*10**6)
temps d'execution : 0.08400869369506836
1000000
```

On remarque que le coût semble linéaire.

La fonction seuil2 renvoie la même chose que seuil mais en utilisant une fonction intermédiaire suite qui prend comme argument n et qui renvoie u_n.

```
def suite(n) :
    u=1
    for i in range(1,n+1) :
        u=u+2
    return u


def seuil2(m) :
    tps=time.time()
    n=0
    while suite(n)<m :
        n+=1
    print("temps d'execution :",time.time()-tps)
    return n


>>> seuil2(10**4)
temps d'execution : 0.5543525218963623
5000

>>> seuil2(2*10**4)
temps d'execution : 2.2181930541992188
10000
```

On remarque que le coût semble quadratique. En effet, à chaque passage dans la boucle while, u_n est recalculé à partir de u_0 et pas de u_{n-1} ce qui nécessite un coût linéaire.