Structures itératives

12 septembre 2021

I La boucle «Tant que»

1) Un exemple introductif : décollage de la fusée

On désire écrire un script qui affiche successivement en passant à la ligne (après chaque nombre) $10, 8, 6, 4, 2$, décollage. On pourrait bien sûr écrire

```
print(10)
print(8)
...
print(2)
print("décollage")
```

Mais si je remplace 10 par 100, vous ne serez plus d'accord. On va demander à l'ordinateur de répéter une tâche tant qu'une condition n'est pas réalisée.

```
temps ← 10
Tant que temps > 0
   afficher temps
   temps ← temps - 2
FinTantque
afficher "décollage"
```

Pour mieux comprendre, on peut «tracer» cet algorithme, c'est-à-dire regarder l'évolution des variables au cours de chaque itération.

temps	10	8	6	4	2	0
temps > 0	V	V	V	V	V	F

Au départ, `temps` vaut 10, donc la condition `temps >0` est vraie, on exécute donc les instructions de la boucle tant que. À chaque itération, la valeur de `temps` est diminuée de 2, donc inexorablement à un moment donné, `temps` vaudra 0 et donc la condition `temps >0` sera fausse et l'algorithme se terminera.

Ceci illustre les deux points fondamentaux d'une boucle «Tant que» :

- il faut pouvoir entrer dans la boucle
- il faut sortir de la boucle

2) La syntaxe Python

On retiendra le mot clé `while`, le symbole : et l'indentation.

```
instructions_avant_boucle
while condition_realise:
   instructions_dans_boucle
instructions_apres_boucle
```

Cela donne pour notre exemple précédent :

```
temps = 10
while temps > 0:
   print(temps)
   temps = temps -2
print("décollage")
```

3) Exemples modèles

Exercice 1 Proposer un script qui détermine le plus petit entier n tel que $2^n > 1000$. Déterminer le nombre de multiplications effectuées.

Corrigé: On peut répondre de tête : en effet, on sait que $2^{10} = 1024$ (valeur référence), donc $2^9 = 512$ ainsi l'entier n recherché est 10. On peut aussi utiliser le logarithme... Ceci étant dit, présentons plusieurs méthodes algorithmiques.

script 1 :

```
cible = 1000
n = 0
while 2**n <= cible:
    n = n + 1
print(n)
```

script 2 :

```
cible = 1000
n = 0
puissance_deux = 1
while puissance_deux <= cible:
    puissance_deux = 2*puissance_deux
    n = n + 1
print(n)
```

Les deux scripts sont corrects et renvoient 10, mais ils n'ont pas la même efficacité. «Traçons» le premier script :

n	0	1	2	$\cdots$	9	10
2^n	1	2	4	$\cdots$	512	1024
$2^n \leqslant 1000$	V	V	V	$\cdots$	V	F

Ainsi $n = 10$. A chaque itération, il calcule 2^n et donc effectue n multiplications. Puisqu'on calcule $2^1, 2^2, 2^9, \ldots, 2^{10}$ donc on effectue $1 + 2 + 3 + \ldots + 10 = \frac{10\times 11}{2} = 55$ multiplications. On peut faire beaucoup mieux, car si l'on a dèjà calculé 2^{n-1}, il suffit de multiplier par 2 pour obtenir 2^n. C'est ce que met en forme le deuxième script, qui n'effectue donc que 10 multiplications.

Remarque : le script ci-dessous qui ressemble beaucoup au premier, renvoie aussi la réponse correcte 10, mais il est faux !

script 3 :

```
cible = 1000
n = 0
while 2**n < cible:
    n = n + 1
print(n)
```

En effet, si je prend pour cible 1024 à la place de 1000, on obtient la trace suivante :

n	0	1	2	$\cdots$	9	10
2^n	1	2	4	$\cdots$	512	1024
$2^n < 1024$	V	V	V	$\cdots$	V	F

Il renvoie alors 10 comme réponse, au lieu de 11 !
Il faut ainsi être très vigilant avec les conditions d'arrêt d'une boucle «tant que».

Exercice 2 (Temps d'attente du double six) Écrire un script qui simule le lancer de deux dés et affiche le nombre de lancers nécessaires avant l'obtention du double six. On pourra importer la fonction `randint` du module `random` pour tirer un entier aléatoire, en exécutant l'instruction `from random import randint`.

Corrigé: Voici un premier code possible :

```
de1 = 0
de2 = 0
nbre_lancers = 0

while de1 + de2 != 12:
   de1 = randint(1,6)
   de2 = randint(1,6)
   nbre_lancers += 1  # ou  nbre_lancers = nbre_lancers + 1
print(nbre_lancers)
```

Attention : certains seront peut-être tentés de remplacer la condition du `while` par celle-ci : `while (de1 != 6) and (de2 !=6)`. C'est une erreur de logique, la négation de (de1 $=$ 6 et de2 $=$ 6) est (de1 $\neq$ 6 ou de2 $\neq$ 6). On pouvait donc remplacer par `while (de1 != 6) or (de2 !=6)`

Voici un deuxième code possible utilisant une variable booléenne, ce qui donne un code assez élégant et tout aussi performant que le précédent :

```
nbre_lancers = 0
pas_double_six = True # variable booléenne
while pas_double_six:
   de1 = randint(1,6)
   de2 = randint(1,6)
   if de1 == 6 and de2 == 6:
       pas_double_six = False
   nbre_lancers += 1
print(nbre_lancers)
```

Remarque : comme il y a une chance sur 36 d'obtenir le double 6, le temps d'attente moyen est de 36.

Exercice 3 (Le juste prix) On veut écrire un script qui demande à l'utilisateur de deviner un nombre entre 1 et 100 qui a été choisi aléatoirement. Si la réponse de l'utilisateur est fausse, le programme indique si sa réponse est trop grande ou trop petite et l'utilisateur peut à nouveau faire une proposition. Le programme ne doit se terminer que si la bonne réponse est donnée.

1. Proposer un script
2. Modifier le script précédent de sorte que le programme affiche à la fin le nombre de réponses nécessaires pour trouver le juste prix.
3. Comment modifier le programme pour qu'il s'arrête à partir de dix mauvaises réponses ?

Corrigé:

```
import random

x = random.randint(1,100)  # x est le prix à deviner
reponse = 0
nbre_reponses = 0

while reponse != x:
    reponse = int( input("Donner un nombre entier entre 1 et 100: "))
    nbre_reponses += 1
    if reponse > x :
        print("C'est moins")
    elif reponse < x :
        print("C'est plus")
    else:
        print("C'est la bonne réponse")
print("Vous avez réussi avec " + str(nbre_reponses) + " réponses")
```

Voici quelques prolongements possibles :

- Si l'utilisateur est malin et utilise une technique de dichotomie, au pire combien de réponses devra-t-il fournir pour deviner un nombre entre 1 et 100000000 ?
- On inverse les rôles. C'est l'utilisateur qui choisit un entier. On doit écrire un programme qui trouve la bonne réponse de manière performante.

Exercice 4 (Algorithme d'Euclide) L'algorithme d'Euclide permet de calculer le pgcd de deux entiers. Il repose sur les deux propriétés suivantes :

- $\text{pgcd}(a, 0) = a$.
- si r est le reste de la division euclidienne de a par b, alors $\text{pgcd}(a, b) = \text{pgcd}(b, r)$.

Écrire une fonction `euclide` qui prend en argument deux entiers et renvoie leur pgcd.

Corrigé: Voici une version itérative, nous verrons plus tard une version récursive.

```
def euclide(a,b):
    """Données: a et b deux entiers naturels non nuls
       Résultat: le pgcd de a et b, calculé par l'algorithme d'Euclide"""
    while b != 0:
        r = a % b
        a = b
        b = r
    return a
```

4) La nécessité d'une preuve de terminaison

Deux situations opposées :

```
a = 5
while a > 5:
   a = a + 1
   print(a)
```

```
a = 10
while a > 5:
   a = a + 1
   print(a)
```

Le premier script ne fait rien car on rentre pas dans la boucle. Le deuxième script lui au contraire boucle indéfiniment. Il faudra donc toujours s'assurer que notre algorithme se termine, ce qui parfois ne sera pas immédiat et nécessitera des preuves mathématiques. Pour votre culture, il existe des algorithmes dont on ne sait pas prouver la terminaison. Par exemple la conjecture de la suite de Syracuse, que nous verrons en exercice.

II Vers la boucle «pour»

1) Correspondance entre les boucles «Tant que» et «Pour»

Dans l'exemple du juste prix, on ne sait pas à l'avance combien d'itérations on doit faire. Mais parfois, on le sait comme dans l'exemple de la fusée. La plupart des langages de programmation permettent alors d'utiliser l'instruction «pour», qui permet une écriture plus concise. Par exemple, pour afficher le carré des entiers de 1 à 25, voici deux scripts qui conviendraient l'un avec une boucle «pour», l'autre avec une boucle «tant que» :

Pour k de 1 à 25 :
  afficher k^2 ;
Fin du pour

$k \leftarrow 1$;
Tant que $k \leqslant 25$, faire :
  afficher k^2
  $k \leftarrow k + 1$
Fin du tant que

2) La syntaxe Python

Traduisons l'exemple précédent :

```
for k in range(1,26):
    print(k**2)
```

Les mots clefs sont **in** et **range**. La fonction `range` en Python, constitue un exemple d'**itérateur**. On prendra garde au fait que l'élément 26 n'est pas créé par `range(1,26)`. On s'arrête à 25.

On peut modifier le pas de la fonction range, par exemple les éléments parcourus par `range(1,60,2)` sont les nombres impairs de 1 à 59.

3) Exemples modèles

Exercice 5 Voici une procédure qui affiche les diviseurs d'un entier n.

```
def diviseurs(n):
    for k in range(1, n+1): # pour k de 1 à n
        if n % k == 0:
            print(k)
```

Exercice 6 Pour les trois scripts suivants, dire ce qui est affiché et combien de fois.

```
for i in range(3):
    print('bidule')
print('truc')
```

```
for i in range(3):
    print('bidule')
for j in range(4):
    print('truc')
```

```
for i in range(3):
    print('bidule')
    for j in range(4):
        print('truc')
```

Corrigé:

1. Bidule est affiché 3 fois et truc 1 fois.
2. Bidule est affiché 3 fois et truc 4 fois.
3. Bidule est affiché 3 fois et truc 12 fois.

Exercice 7 (Un peu de dessin) Écrire trois procédures qui permettraient de réaliser les trois affichages suivants, avec comme paramètres le nombre de lignes et ou de colonnes, ainsi que le choix du caractère.

```
****
****
****
```

```
*
**
***
****
```

```
   *
  **
 ***
****
```

On rappelle que le caractère de fin de ligne est `'\n'` et que le caractère «espace» est `' '`. Ainsi les instructions

```
chaine = 'love' +  '\n' + 'rock'
print(chaine)
```

afficheront

```
love
rock
```

Corrigé: Voici une possibilité.

```
def rectangle(nbre_ligne , nbre_colonne, caractere ):
    chaine = '' # chaine vide
    for i in range(nbre_ligne):
        for j in range(nbre_colonne):
            chaine = chaine + caractere
        chaine = chaine + '\n'  # on ajoute le caractere de fin de ligne
    print(chaine)


def triangle_inferieur_gauche(nbre_ligne, caractere ):
    chaine = '' # chaine vide
    for i in range(1, nbre_ligne + 1):
        for j in range(1,i+1):
            chaine = chaine + caractere
        chaine = chaine + '\n'
    print(chaine)


def triangle_inferieur_droit(nbre_ligne, caractere ):
    chaine = '' # chaine vide
    for i in range(1,nbre_ligne + 1):
        for j in range(1, nbre_ligne - i +1):
            chaine = chaine + ' '
        for j in range(nbre_ligne -i +1 , nbre_ligne + 1):
             chaine = chaine + caractere
        chaine = chaine + '\n'
    print(chaine)


rectangle(4, 3, '*')
triangle_inferieur_gauche(4, '*')
triangle_inferieur_droit(4,'*')
```

On retiendra que dès que l'on veut traverser un tableau à deux dimensions (lignes et colonnes), on a besoin de deux boucles imbriquées!! On pouvait proposer d'autres solutions grâce aux raccourcis de concaténation permis par Python, comme `5 * 'X'` qui crée la chaîne `XXXXX`.

```
n= 4 # nbre de lignes
m = 3 # nbre de colonnes
caractere = '*'

for i in range(n):
    print( caractere * m)
```

```
for i in range(n):
    print( caractere * (i+1))

for i in range(n):
    ligne = (n -1 - i) * ' ' + (i +1) * caractere
    print(ligne)
```

Leur code est plus court mais l'opérateur `*` de multi-concaténation cache le fait important qu'il y a deux boucles imbriquées.

4) Notion d'itérateur en Python

Il y a d'autres objets Python autres que `range` que l'on peut itérer : les listes, les tuples ou les chaînes de caractères. Par exemple, observons les scripts suivants :

```
mot = "love"
for lettre in mot:
    print(lettre)
```

Cela affiche les lettres du mot :

```
l
o
v
e
```

```
liste = [3, 5, 1, 8]
s = 0
for x in liste:
    s = s + x
print(s)
```

Cela affiche les sommes intermédiaires des éléments de la liste :

```
3
8
9
17
```

Voici pour finir une fonction `verlan(mot)` qui prend une chaîne de caractère `mot` et la renvoie écrite à l'envers. Par exemple `verlan('vache')` renverra `'ehcav'`.

```
def verlan(mot):
    nouveau_mot = ''
    for lettre in mot:
        nouveau_mot = lettre + nouveau_mot
    return nouveau_mot
```