Chapitre 2 : Boucles et conditions

I) Les boucles For (Pour)

1) Principe

Def : Itération : action répétée un certain nombre de fois

Pour réaliser ces itérations, on utilise un itérateur (souvent il s'agira de i; j ou k par habitude) que l'on fait varier d'une valeur initiale (*ini*) à une valeur finale (*fin*).

La structure d'une boucle est donc la suivante en langage naturel :

```
Pour k allant de ini à fin
    Faire instruction
Fin pour
```

L'itérateur est une variable qui va changer de valeur à chaque itération. Cela permet donc de savoir quand s'arrêter.

2) Opérateur range()

Def : Pour créer la liste des valeurs de l'itérateur sur laquelle la boucle va être effectuée, on utilise souvent la fonction range(ini,fin+1,pas). Cette fonction génère une liste de valeurs espacées de « pas » entre « ini » et « fin ».

Exemple : range(1,10,1) correspond aux entiers compris entre 1 et 9

 *Python ne « compte » pas jusqu'à la dernière valeur du range, mais jusqu'à la précédente …*

Rq : - lorsque le pas vaut 1, on peut simplement écrire range(ini,fin+1).

- pour faire varier l'itérateur de 0 à n-1, on peut écrire simplement range(n).

La syntaxe d'une boucle « Pour » en Python est donc :

```
For k in range(ini,fin+1,pas):
    Instructions
```

 La ligne comportant for doit se finir par « : » et il faut s'assurer que les instructions de la boucle soient *indentées*. Une indentation est un décalage vers la droite, ce qui est fait automatiquement sous Spyder pour certains mots-clés qui le nécessitent. On peut aussi les réaliser avec la touche TAB ⇤⇥.

Rq : lorsqu'on écrit un code sur papier, il est demandé de repérer les indentations avec un trait vertical à gauche « | ».

Exemples :

1) Donner les lignes de code affichant dans la console les entiers compris entre 5 et 15.

```
for k in range(5,16):
    print(k)
```

2) Donner les lignes de code affichant dans la console les carrés des entiers compris entre 0 et 10 par pas de 3.

```
for k in range(0,11,3):
    print(k**2)
```

3) Calcul itératif

La boucle for est bien adaptée pour le calcul des différents termes d'une suite récursive (c'est-à-dire dont le terme u_n dépend des précédents).

Exemple :

1. Pour le code suivant, réaliser un tableau avec les valeurs successives de k et de u.

```
1 n=5
2 u=1
3 q=-2
4 for k in range(n):
5     u=q*u
6 print(u)
```

k					
u					

2. De quel type de suite s'agit-il ? Quel est le résultat affiché dans la console ?

3. Que cela change-t-il si on indente la ligne 6 ?

Prop : pour calculer la somme des termes d'une liste ou celle des termes d'une suite, on peut initialiser une variable notée souvent « S » ou « Somme » à 0 avant la boucle, puis calculer les valeurs successives de « S » (ou « Somme ») dans la boucle.

Exemple : Que réalise le code suivant ? Quelle sera la valeur affichée ?

```
S=0
for k in range(0,11):
    S=S+k
    print(S)

```

Rq : On peut remplacer l'instruction $S = S + k$ par $S+= k$ si on veut « alléger » la syntaxe.

Prop : De la même manière, on peut initialiser une variable « P » ou « Produit » à 1 avant la boucle, et utiliser l'opération Produit *= dans la boucle.

Exemple : Ecrire des lignes de code permettant d'afficher la valeur de n ! = 1 x 2 x … x n (factorielle n).

```
n=int(input('n'))
p=1
for k in range (1,n+1):
    p=p*k
print("n! = ",p)
```

4) Parcours d'une liste

Il existe 2 façons principale de parcourir une liste :
- Une boucle « for » allant de 0 à $len(L)$
- On peut aussi utiliser l'instruction for x in L

Exemple : Soit $L = [1,2,3,4,5,6,7,8,9]$ écrire un programme qui calcule la somme des termes de L.

Avec l'instruction $len(L)$	Avec l'instruction for x in L
$S = 0$ For k in range$(0, len(L))$: $S = S + L[k]$	$S = 0$ For x in L: $S = S + x$

Def : On peut également créer une liste M à partir d'une liste L en ne gardant que les éléments de L satisfaisant une certaine condition. Le principe de la syntaxe suivante est appelée « création d'une liste en compréhension ». Cette méthode permet de se passer d'une boucle pour créer la liste M.

Exemple : $L = [1,2,3,4,5,6,7,8,9]$ et on veut créer la liste M contenant uniquement les entiers pairs de L. (Notons que l'instruction : if $a\%b == 0$: permet de savoir si un entier a est divisible par un entier b)

$M = [x$ for x in L if $x\%2 == 0]$

Exemple :

Écrire les lignes de code créant une chaîne de caractère « motInverse » à partir d'une chaîne de caractère « Mot », et composée des mêmes caractères mais en ordre inverse.

```
mot=input("mot : ")
rep=""
for x in mot:
    rep=x+rep
print(rep)
```

Rq : Les boucles for sont souvent utiles pour créer une liste.

Exemple : on souhaite créer une liste L avec les entiers de 1 à 100 :

- solution 1 : taper la liste à la main : L = [1,2, … , 100]… un peu long non ???

- solution 2 : créer la liste par *concaténation* avec une boucle for :

```
L=[]
for k in range(1,101):
    L=L+[k]
print(L)
```

- solution 3 : créer la liste en *compréhension* avec for :

```
L=[k for k in range(1,101)]
print(L)
```

- solution 4 : créer la liste avec la fonction append et une boucle for :

```
L=[]
for k in range(1,101):
    L.append(k)
print(L)
```

Def : l'instruction "append" permet donc d'ajouter un élément à la fin d'une liste.
Plus précisément, l'instruction L.append(k) ajoute la valeur k à la fin de la liste L. Cela revient à concaténer la liste L avec la liste $[k]$.

5) Boucles imbriquées

Il est parfois utile de créer des codes composés de plusieurs boucles for imbriquées. La difficulté est alors de bien identifier chaque itérateur et de respecter les indentations.

Exemple : L'algorithme suivant permet d'afficher les tables de multiplication (jusqu'à 10×10).

```
for i in range (1,11):
    for j in range(1,11):
        print(i,'*',j,'=',i*j,';',end=' ')
    print()
    print()

```

Rq : l'instruction « end=' ' » permet de ne pas aller à la ligne après l'affichage. De même l'instruction « print() » permet d'aller à la ligne. La double instruction « print() » permet donc de sauter une ligne entre les différentes tables de multiplication.

II) Les conditions (if ; else ; elif)

1) Variables booléennes

Def : Les booléens sont un type de variable en informatique. Une variable booléenne est souvent le résultat d'un test logique binaire (test d'égalité, d'inégalité, etc.), dont le résultat ne peut être vrai ou faux.

En langage Python, une variable booléenne (type **bool**) peut prendre les valeurs **True** ou **False**.

Prop : Un booléen peut être le résultat d'un test simple, dont les opérateurs sont les suivants en langage Python :

Test	Opérateur
Inégalité stricte	**< ou >**
Inégalité large	**<= ou >=**
Égalité	**==**
Différence	**!=**

Def : les connecteurs lient plusieurs tests ou booléens ensemble pour obtenir un résultat global, selon les règles de la logique.

Connecteur	Écriture en Python
« ET » logique	**and**
« OU » logique	**or**
« NON » logique	**not**

Rq : En python, l'écriture d'une inégalité, ou d'une « non égalité » se voit attribuée la valeur booléenne correspondante à sa véracité.

Exemple :

```
1  x=1>2
2  print(x) #false
3  y=4!=8
4  print(y) #true
5  z=1<2 and 2<3
6  print(z) #true
```

2) Les structures if ; else ; elif

Structure simple

De très nombreux algorithmes vont nécessiter l'utilisation d'instructions conditionnelles.

La plus simple s'écrirait en pseudo-code : « si … alors … »

En Python, une instruction conditionnelle simple s'écrit de la manière suivante :

if condition :
    instruction
suite du code


La structure conditionnelle présente une *indentation* pour délimiter les instructions du reste du code et un « **:** » à la fin de la condition. Si la condition est vraie (**True**), alors l'instruction est exécutée. Sinon, rien ne se passe et Python sort de la structure conditionnelle.

Exemple : Voici un algorithme qui affiche un message si le nombre entré est un entier.

```
1  from math import floor
2  x=float(input("entrer un réel : "))
3  if x==floor(x):
4      print("c est un entier")
5
```

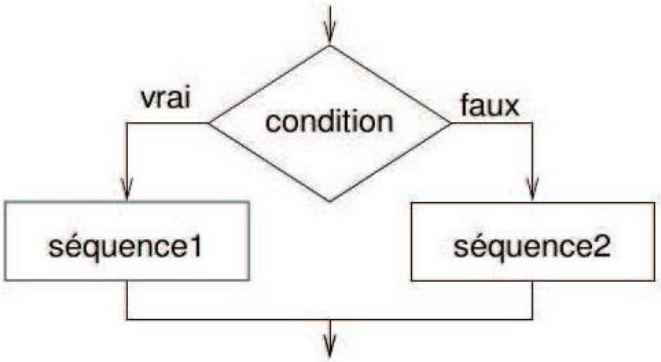
rq : - Attention à bien mettre une double égalité pour effectuer la comparaison
- Si le réel n'est pas un entier, l'algorithme n'affichera rien.
- l'instruction floor (à charger dans la bibliothèque math) correspond à la partie entière d'un réel.

Structure avec alternative

En pseudo-code, une telle structure s'écrirait : « si … alors … sinon ... ».

En Python, cela s'écrit sous la forme :

if condition :
instruction1
else :
instruction2
suite du code



 Il faut veiller aux indentations après le « if » et le « else », ainsi qu'aux « : » pour que le code ne renvoie pas de messages d'erreur.

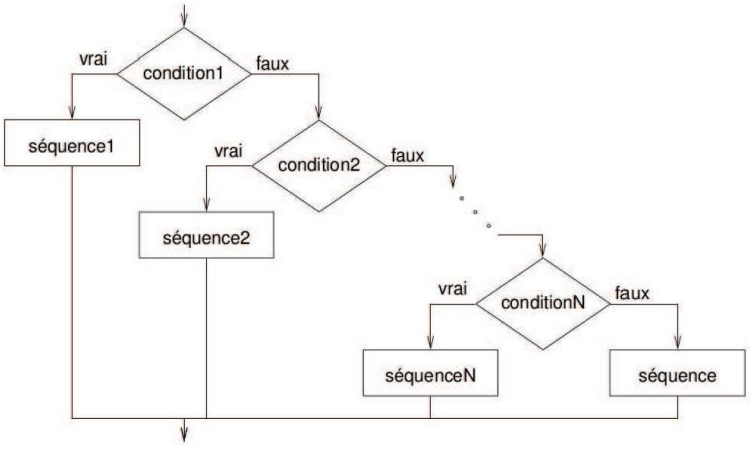
Exemple : test de majorité :

```
1 age=float(input('quel est votre âge ?'))
2 if age>=18:
3     print('vous êtes majeur')
4 else :
5     print('vous êtes mineur')
```

Structure avec plusieurs alternatives

On rencontre parfois des problèmes qui nécessitent plusieurs tests successifs. En pseudo-code, cela s'écrirait : « si … alors … sinon si … alors … sinon si … alors … (etc.) … sinon ... ».

La représentation d'une telle structure est la présentée ci-dessous :



En langage Python, on utilise autant de « **elif** » (contraction de *else if*) que nécessaire entre le « **if** » initial et le « **else** » final (on peut également terminer par un « **elif** ») :

if condition1 :
    instruction 1
elif condition2 :
    instruction 2
elif
  ⋮
else :
    instruction n
suite du code

L'exécution d'une telle structure se déroule de la manière suivante :

- si la condition 1 est vérifiée, l'instruction 1 est réalisée et Python sort de la structure,

- sinon, si la condition 2 est vérifiée, l'instruction 2 est réalisée et Python sort de la structure,

- etc.

- enfin, si aucune des conditions précédentes n'a été vérifiée, la dernière instruction (celle du **else** final, qui n'est pas suivie par un test booléen) est exécutée.

Exemple : Le code suivant détermine le nombre de solution de l'équation de degré 2 : $ax^2 + bx + c = 0$

```
a=float(input("a= "))
b=float(input("b= "))
c=float(input("c= "))
d=b**2-4*a*c
if d<0:
    print("pas de solution")
elif d==0:
    print("1 seule solution")
else :
    print("2 solutions")
```

Rq : le « else » pourrait être remplacé par « elif d > 0 » si on préfère.

III) Les boucles while

1) La boucle while

Def : La boucle **while** (« tant que » en français) est une **boucle conditionnelle** : un corps de boucle contenant un bloc d'instructions est répété (ou **itéré**) **tant qu**'une condition est vérifiée.

 Les contraintes d'indentation sont les mêmes que pour une boucle **for**. La syntaxe en Python est la suivante :

```
while condition:
    instructions
```

Exemple : Considérons le code suivant :

```
1 c=3
2 p=1
3 while c>0:
4     p=p*2
5     c=c-1
```

Détaillons l'exécution de la boucle **while** :

	c>0	p	c
Avant la boucle			
Itération n°1			
Itération n°2			
Itération n°3			
Itération n°4			

Def : « c » est ici le **variant de boucle**, c'est-à-dire une variable qui évolue à chaque itération et qui assure que la boucle se termine bien.

 Lorsqu'on écrit une boucle **while**, il faut vérifier qu'elle se terminera forcément. On justifiera ainsi la **terminaison** de l'algorithme.

Rq : Pour programmer une boucle conditionnelle (while)

- identifier une condition de sortie de boucle,
- écrire le corps de boucle, en s'assurant que celui-ci peut modifier la valeur d'un **variant de boucle** opérant sur la condition du while,
- prévoir une initialisation des variables en amont de la boucle,
- revenir au niveau d'indentation du **while** pour le reste du programme,
- s'assurer que la condition de fermeture de boucle est certaine d'être réalisée.

2) Comparaison avec la boucle FOR

La boucle conditionnelle **while** présente deux avantages par rapport à la boucle inconditionnelle **for** :

- Dans certains cas, le nombre d'itérations n'est pas connu : la boucle **for** ne pourrait pas être utilisée,

- Elle permet d'économiser de la mémoire : la boucle **for** nécessite une liste d'éléments que parcourt l'itérateur de boucle (souvent au moyen de la fonction **range()**, voir chapitre 3).

Attention, une boucle while présente un gros inconvénient par rapport à une boucle for : Le programme peut tourner indéfiniment si la condition de sortie n'est jamais vérifiée. Alors qu'avec une boucle For, le programme se terminera toujours.

Exemple (algorithme seuil) :

Soit (u_n) une suite géométrique de premier terme $u_0 = 1$ et de raison 1.2.
Le programme suivant déterminer le plus petit rang à partir duquel $u_n \geq 1000$

```
1  u=1
2  n=0
3  q=1.2
4  while u<1000:
5      u=u*q
6      n=n+1
7  print(n)
```

Rq : On peut justifier la terminaison de cet algorithme car pour une suite géométrique de premier terme positif avec $q > 1$, on sait que sa limite vaut $+\infty$. Il existera donc forcément un rang n à partir duquel u_n sera supérieur à 1000

3) Interruption de boucle

Def : Il existe en Python une instruction qui permet de sortir d'une boucle **for** (ou d'une boucle **while**) : la fonction **break**.
Cette fonction sera surtout utile si le programme cherche une valeur, et qu'une fois trouvée, il n'est plus nécessaire de continuer.

Exemple (test de primalité) : L'algorithme suivant détermine si un entier n entré par l'utilisateur est premier ou non.

```
1  n=int(input("entrer un entier : "))
2  rep=True
3  for k in range(2,n):
4      if n%k==0:
5          rep=False
6          break
7  print(rep)
```

Rq : - un entier est premier s'il ne possède aucun autre diviseur que 1 et lui-même. S'il possède un autre diviseur, noté k alors c'est qu'en divisant n par k le reste est nul ($n\%k = 0$). Le nombre est alors non premier.

- Une fois qu'on a trouvé un diviseur non trivial de n, il est inutile d'en chercher d'autre, et on peut interrompre la boucle avec l'instruction « break ».

- 1 n'est pas premier par convention, il faudrait faire un test spécifique pour cette valeur puisque ce programme renverra « true » si $n = 1$ puisqu'on ne passe pas dans la boucle For.

Rq : Le danger de la boucle **while** est que le nombre d'itérations peut être très grand voire infini (ainsi, le programme tourne sans jamais s'arrêter).

On peut alors ajouter une condition pour que la boucle s'arrête si le nombre d'itérations dépasse un certain seuil : on utilise alors un second **variant de boucle**.

Typiquement, ce variant est un entier et on utilise un test sur sa valeur comme condition de sortie de boucle.

Exemple :

La limite de la suite définie par $u_{n+1} = \frac{u_n}{4} + 3$ avec $u_0 = 1$ est 4. On cherche à déterminer la valeur de n pour laquelle on s'approche à 10^{-14} de cette limite. Vu la précision recherchée, on veut s'assurer que le nombre d'itérations (et donc le temps de calcul) ne sera pas trop grand : on se limite ici à $n = 100$.

On peut ainsi écrire indifféremment ces deux versions :

```
1  u=1
2  erreur=1e-14
3  n=0
4  while abs(u-4)>erreur:
5      u=u/4+3
6      n=n+1
7      if n>100:
8          break
9  print(n, abs(u-4)<erreur)
```

```
1  u=1
2  erreur=1e-14
3  n=0
4  while(abs(u-4)>erreur and n<100):
5      u=u/4+3
6      n=n+1
7  print(n, abs(u-4)<erreur)
```

En sortie, le programme affiche la valeur de n recherchée.

Si la boucle ne respecte pas le nombre d'itérations maximal autorisé, n sera égal à 100.

On affiche également la valeur du test (qui sera un booléen ici), pour vérifier que le programme a bien fonctionné.

Exemple de détermination de la valeur approchée d'une racine carrée

Héron d'Alexandrie rapporte la méthode itérative Babylonienne pour déterminer la solution de $x^2 = a$, c'est-à-dire pour déterminer la racine carrée de a avec la meilleure précision possible.

Prenons un exemple : $a = 811$

La première étape est de donner une valeur très approximative v de la racine : Disons ici 30 (puisque $30^2 = 900$). Lors du codage, on pourra prendre $v = \frac{a}{2}$ ou $v = \frac{a}{3}$

Il faut ensuite calculer $\frac{811}{30} = 27.03$. Les mathématiciens antiques avaient compris que la racine carrée recherchée devait donc se situer entre 27.03 et30 : la nouvelle valeur de v est donc la moyenne de ces deux nombres, soit 28,52.

On répète ensuite le calcul : $\frac{811}{28,52} = 28,44$. La valeur de $\sqrt{811}$est donc située entre 28,44 et 28,52.

On répète alors l'itération avec $v = \frac{28.44+28.52}{2} = 28,48$. On s'approche ainsi très vite de la valeur de la racine carrée de 811. On répète l'opération tant que la différence entre deux valeurs successives de v n'est pas inférieure à un certain seuil de précision.

A la main, ces calculs deviennent vite fastidieux, mais avec Python, c'est très simple :

```
1  a=float(input("nombre dont on cehrche la racine carrée : "))
2  p=float(input("précision voulue : "))
3  v=a/3
4  n=0
5  while(abs(a/v-v)>p):
6      v=((v+a/v)/2)
7      n=n+1
8  print(v,n)
```

Pour $n = 811$ et $p = 10^{-6}$, on trouve $v = 28,478062$ et un nombre d'itérations seulement égal à 6 !!!

Exercices :

La ligne de code suivante : `1 from random import*` permet d'utiliser ensuite les instructions randint(min,max) qui choisit un entier au hasard entre min et max (min et max étant à donner), et random() qui choisit un réel au hasard entre 0 et 1.

1) Ecrire un programme qui génère une liste de 20 entiers tous compris aléatoirement entre 1 et 9

2) Ecrire un programme qui affiche sous une forme triangulaire une fois le chiffre 1 puis à la ligne du dessous deux fois le chiffre 2, puis à la ligne du dessous trois fois le chiffre 3 etc… jusqu'à dix fois le chiffre 10.

3) Le jeu du plus ou moins.

Ecrire un algorithme qui choisit une valeur au hasard entre 1 et 100 et qui demande à l'utilisateur de retrouver ce nombre. Le programme doit indiquer si le nombre à trouver est plus grand ou plus petit que le nombre entré à chaque tentative. On pourra aussi afficher le nombre de tentatives qui ont été nécessaires pour retrouver le nombre de départ.