Introduction à la programmation, cours 3

I. Boucle While

Objectif : répéter une instruction tant qu'une condition est vérifiée.
Syntaxe :

```
while <condition> :
        Instructions
suite_du_programme
```

Les *Instructions* sont répétées tant que *condition* est vérifiée. Lorsque *condition* n'est plus vérifiée, le programme exécute *suite_du_programme*.
Exemple : un programme qui demande un entier *A*, puis un entier *B* jusqu'à ce que celui-ci soit non-nul, puis qui calcul *A/B*.

```
A = input("Entrez le nombre A : ")
B = input("Entrez le nombre B : ")
while B == 0 :
        B = input("B est nul ! Recommencez :"
print "Le quotient vaut : ", A/B
```

NB : ATTENTION AUX BOUCLES INFINIES !
Si la condition de la boucle **while ne devient jamais fausse**, le programme boucle indéfiniment, voici deux exemples (pour forcer l'arrêt du programme et sortir de la boucle infinie faites **ctrl + c**):

```
n = 5
while n < 10 :
        print "n vaut : ", n
print "Fin."
```

```
while True :
        print "Je boucle…"
print "Fin."
```

Compteurs de boucle

Le compteur de boucle sert à compter combien de fois on entre dans la boucle (ou un nombre dépendant de cela).

```
var = intput("Donnez un entier non-nul : ")
n = 0     # n est le compteur de boucle, ici initialisé à 0
while var == 0 :
        n = n + 1
        var = input("Incorrect, recommencez : ")
print "Merci, il vous a fallu ", n, " essais supplémentaires"
```

IMPORTANT : toujours penser à initialiser le compteur.

On peut se servir du compteur dans la condition :

```
i = 0     #variable compteur
while i < 10 :     #compteur utiliser dans la condition du while
        print 2 ** i        #affiche les puissances de 2
        i = i + 1           #incrémentation du compteur, sinon boucle infinie
print "Fin"
```

Le ***pas*** est l'augmentation du compteur à chaque étape, dans l'exemple ci-dessus le pas est de 1, mais on peut

choisir le pas qui nous plaît :

```
i = 0      #variable compteur
while i < 20 :
        print i
        i = i + 2          #ici le pas est de 2
print "Fin"
```

```
i = 10    # variable compteur, ici initialiser à 10
while i > 0:
        print i
        i = i - 1          #ici le pas est de -1
print "Fin"
```

Exercice_1 : Ecrire un programme qui demande un entier *n* à l'utilisateur puis affiche la somme des entiers de 1 à n (utilisez une boucle while) (par exemple si n=3, ça affiche 6),

Les mots clés « break » et « continue »

Le mot clé **break** permet de sortir immédiatement d'une boucle while :

```
i = 1
while i < 100 :
        if i % 2 == 0      #si i est pair
                print "* "
                break
        i = i+1
        print "Incrementation de i"
print "Fin"
```

Le mot clé **continue** permet de remonter immédiatement au début d'une boucle while (l'exemple suivant contient une boucle infinie, comprenez-vous pourquoi ?):

```
i = 1
while i < 100 :
        if i % 2 == 0
                print "* "
                continue
        i = i+1
        print "Incrementation de i"
print "Fin"
```

Inconvénient de *break* et *continue* :

- Rend le code plus difficile à lire/analyser si il y a plusieurs niveaux d'imbrications et/ou de longues instructions dans le while
- N'a pas toujours d'équivalent dans les autres langage de programmation.

On essaiera donc de se passer de *break* et *continue* autant que faire se peut.

II. Les fonctions

Créer une fonction

Voilà comment créer une fonction :

```
def nom_de_la_fonction ( parametre1, parametre2, parametre3, parametreN ) :
        Bloc d'instructions
```

Décortiquons la ligne de définition de la fonction, on trouve dans l'ordre :

- **def**, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable. N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante.
- Les deux points qui clôturent la ligne (comme pour les boucles et les structures conditionnelles).

Exemple, une fonction qui affiche la table de multiplication par 7 (la plus dur) :

```
def table_par_7():          #Cette fonction ne prend pas de paramètres, mais il faut quand même mettre les
                            #parenthèses
    nb = 7
    i = 0                   #le compteur de la boucle while
    while i < 10:           # Tant que i est strictement inférieure à 10,
        print i + 1, "*", nb, "=", (i + 1) * nb
        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Quand vous exécutez ce code à l'écran, il ne se passe rien, il faut **appelez la fonction**, ce qui se fait en ajoutant la ligne suivante :

```
table_par_7()    #appel de la fonction table_par_7()
```

Tout ça est très bien, mais pour bien voir la puissance des fonctions, ajoutons un argument à table_par_7 pour la transformer en une fonction qui affiche la table de multiplication de n'importe quel nombre :

```
def table(nb):    #on ajoute un paramètre à la fonction
    i = 0
    while i < 10:           # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1              # On incrémente i de 1 à chaque tour de boucle.

table(7)
table(12)
```

Exercice : Ajoutez un argument *max* à la fonction *table* ci-dessus pour qu'elle affiche nb*1, nb*2, …, nb*max.

L'instruction « return »

Il existe des fonctions comme *print* qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que *input()* ou *type()* qui renvoient une valeur.
Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument :

```
def carre (nb) :
        rep = nb * nb
return rep
```

L'instruction **return** signifie qu'on va **renvoyer** la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le *return* ne s'exécutera pas. Ajouter les deux lignes de codes suivantes :

```
var = carre (5)
print var
```

ou simplement `print carre(5)`

Exercice :

1) Ecrire une fonction **ajoute_prefixe** qui prend en argument un mot et un préfixe et qui renvoie la chaîne de caractère obtenue en concaténant le préfixe suivi du mot. Par exemple, *ajoute_prefixe("mentir", de")* vaut *"dementir"*.
2) Ecrire une fonction **repete** qui prend en argument un mot et un entier n et qui renvoie la chaîne de caractères obtenue en répétant le mot n fois à la suite (sans séparation). Par exemple, *repete("bla", 3)* vaut *"blablabla"*.
3) Ecrire une fonction **ajoute_longueur** qui prend comme argument un mot et qui ajoute sa longueur au début et à la fin. Par exemple, *ajoute_longueur("toto")* vaut *"4toto4"*.
   Note: On rappelle que la longueur d'un mot est obtenue avec la fonction **len()**, par exemple *len("toto")* vaut *4*. De plus, pour convertir un entier ou flottant en chaînes de caractères, il faut utiliser la fonction **str(...)**. Par exemple, *str(4)* vaut *"4"*.
4) Ecrire une fonction **que_des_nombres** qui prend en argument un mot et qui renvoie la chaîne de caractères obtenue à partir du mot en remplaçant chaque lettre par sa position dans le mot. Par exemple, *que_des_nombres("toto")* vaut *"1234"*, et *que_des_nombres("bonjour")* vaut *"1234567"*.

III. Boucle for

Les boucles for permettent de parcourir des structures tel que les éléments d'une listes, les lettres d'un mot etc etc. Voyons un premièr exemple d'une boucle *for* parcourant les éléments d'une liste. Notez l'utilisation du *mots clé* **in** :

```
liste = [2,4,6,8,10, "toto"]
for elt in liste :              # elt prends successivement les valeurs de la liste parcourue
        print elt
```

Regardez bien la coloration syntaxique sur votre éditeur, ça aide à comprendre.
Notez que le programme suivant fait exactement la même chose :

```
liste = [2,4,6,8,10, "toto"]
for x in liste :    #ici j'ai mis x à la place de elt . Ça change rien, j'aurais tout aussi bien pu mettre toto
        print x
```

Une boucle for peut aussi parcourir les lettres d'une chaîne de caractères:

```
var_chaine = "Je commence Python"
for x in var_chaine :
        print x
```

Le mot clé **in** peut aussi être utilisé pour tester si un symbole appartient à une chaîne de caractère :

```
var_chaine = "Je commence Python"
voyelle = "AEIOUYaeiouy"
for x in var_chaine :
        if x in voyelle :  #teste si x est une voyelle
                print x
        else :
                print "*"
```

Méditer bien sur le code ci-dessus, il contient un **if** dans une boucle for. Ainsi, les 4 dernières lignes sont toutes dans le bloc de for.

En passant, on peut faire la même chose avec une liste à la place de la chaîne de caractère :

```
var_chaine = "Je commence Python"
```

```
voyelle = ["A", "E", "I", "O", "U", "Y", "a" ,"e", "i", "o" ,"u" ,"y"]
for x in var_chaine :
        if x in voyelle :  #teste si x est une voyelle
                print x
        else :
                print "*"
```

Maintenant une autre utilisation de for extrêmement pratique. Cette fois, on utilise deux mots clés : **range** et **in**. On commence par un exemple :

```
for x in range (1, 10, 3) :
        print x
```

Regardez bien ce que le code ci-dessus affiche, et essayez de comprendre ce qu'il se passe.
Vous avez compris ? Plus ou moins ? Bon, je vous explique.
range (deb, fin, pas) :

- c'est une fonction qui prend des arguments entiers (c'est à dire qu'il faut remplacer *deb*, *fin* et *pas* par des entiers)
- ça génère une séquence d'entier entre *deb* et *fin-1* avec le pas choisi.

Voici deux autres exemples. Hésitez surtout pas à en faire d'autres vous-même. Et surtout prenez le temps de réfléchir à chaque résultat, pour bien comprendre ce qu'il se passe.

```
for x in range (12, 24, 5) :
        print x
```

```
for x in range (20, 10, -2) :
        print x
```

NB_1 : en cas d'incohérence, la boucle est ignorée et le programme passe directement à l'instruction suivante :

```
for x in range (100, 110, -2) : #Ici, l'ensemble que parcourt x est vide, on n'entre donc pas dans la boucle for
        print x
for x in range (110, 100 ,-2) :
        print x
```

NB_2 : dans `for x in range (deb, fin, pas)`, les paramètres **deb** et **pas** sont **optionnelles**.

- **range(a) :** séquence des entiers dans [0, a[, c'est-à-dire dans [0, a-1] : ici **deb** et **pas** ne sont pas précisés et sont automatiquement mis à respectivement **0** et **1**.
- **range(b,c)** : séquence des valeurs [b, c[, c'est-à-dire dans [b, c-1] : ici *pas* n'est pas précisé, et est automatiquement mis à 1.

Faites des testes de boucle for avec des range contenant seulement un ou deux paramètres.

Exercices :

Exo_1 : Ecrire un programme qui demande à l'utilisateur de taper un nombre impair. Si l'utilisateur tape un nombre pair, le programme doit afficher "J'ai demandé un nombre impair! Re-essayez:" et ainsi de suite jusqu'à ce que l'utilisateur tape enfin un nombre impair. Lorsque c'est le cas, le programme affiche "Merci" et s'arrête.

Exo_2 :
Dans cet exercice, on vous demande d'écrire des **fonctions**. **A chaque fois** que vous écrivez une fonction, il est **impératif** de la tester sur des exemple pour s'assurer qu'elle renvoie bien le résultat attendue.
Dans cet exo, tous les nombres considérés seront des **entiers positifs ou nuls**.

1) Ecrire une fonction `somme_pairs` qui prend en argument une liste et qui renvoie la somme des nombres pairs contenus dans la liste.

   Exemple : `somme_pairs([4, 7, 12, 0, 21, 5])` vaut `16` (car 16=4+12+0).

2) Ecrire une fonction `nb_elem_pairs` qui prend en argument une liste et qui renvoie le nombre d'entiers pairs contenus dans la liste.

   Exemple : `nb_elem_pairs([4, 7, 12, 0, 21, 5])` vaut `3` (car 4, 12 et 0 sont pairs).

3) Ecrire une fonction `max_pair` qui prend en argument une liste et qui renvoie le plus grand entier pair contenu dans la liste. On supposera pour simplifier (dans cette question uniquement) que la liste contient toujours au moins un nombre pair.

   Exemple : `max_pairs([4, 7, 12, 0, 21, 5])` vaut `12`.

4) Ecrire une fonction `min_pair` qui prend en argument une liste et qui renvoie le plus petit entier pair contenu dans la liste. Si la liste ne contient aucun entier pair, la fonction renverra `None`.

   Exemple : `min_pairs([4, 7, 12, 0, 21, 5])` vaut `0` et `min_pairs([9, 3, 1])` vaut `None`.

5) Ecrire une fonction `indice_de` qui prend en argument un entier (supposé pair) et une liste, et qui renvoie l'indice auquel apparaît cet entier dans la liste. Si l'entier n'apparaît pas, la fonction renverra `None`. On suppose ici pour simplifier que l'entier cherché n'apparaît pas plusieurs fois dans la liste.
   Indice*:* si la boucle `for e in liste` ne vous convient pas, n'oubliez pas que vous pouvez faire un `for + range` ou un `while`...

   Exemple : `indice_de(12, [4, 7, 12, 0, 21, 5])` vaut `2` (car 12 est placé à l'indice 2), et `indice_de(6, [4, 7, 12, 0, 21, 5])` vaut `None`.

6) Ecrire une fonction `trouve_premier_pair` qui prend en argument une liste et qui renvoie l'entier pair qui apparaît en premier dans la liste. Si la liste ne contient pas d'entier pair, la fonction renverra `None`.

   Exemple : `trouve_premier_pair([1, 15, 4, 7, 12, 3])` vaut `4` et `trouve_premier_pair([1, 17, 7])` vaut `None`.

7) Ecrire une fonction `extrait_pairs` qui prend en argument une liste `l1` et qui renvoie la liste obtenue à partir de `l1` en ne gardant que les entiers pairs (et sans changer leur ordre). **Attention**, `l1` ne doit **pas être modifiée** par la fonction.

   Exemple : `extrait_pairs([4, 7, 12, 0, 3])` vaut `[4, 12, 0]` et `extrait_pairs([21, 17, 3])` vaut `[ ]`.

Exo_3 :Ecrire un programme qui demande à l'utilisateur plusieurs nombres successivement en affichant "Nombre ?" jusqu'à ce que l'utilisateur entre le nombre 0.
Le programme doit alors afficher "Tous -", ou "Tous +", ou "Seulement 0", ou "Ni tous +, ni tous -" selon que l'utilisateur n'a rentré que des nombres négatifs, ou bien que des nombres positifs, ou bien seulement zéro, ou bien aucun des cas précédents.
Dans ce dernier cas (Ni tous +, ni tous -), le programme doit afficher "Somme -", ou "Somme=0", ou "Somme +" selon que la somme totales des nombres donnés par l'utilisateur est strictement négative, nulle ou strictement positive.

Exo_4 :

1) Ecrire une fonction `decale` qui prend en argument une liste `l` et un nombre `d` et qui renvoie la liste obtenue à partir de `l` en ajoutant un décalage de `d` à chaque élément de `l`.

   Exemple : `decale([4, 17, 12], 3)` vaut `[7, 20, 15].`

2) Ecrire une fonction `intercale_zeros` qui prend en argument une liste `l` et qui renvoie la liste

obtenue à partir de `l` en intercalant un zéro après chaque élément de `l`.

Exemple : `intercale_zeros([4, 17, 12])` vaut `[4, 0, 17, 0, 12, 0]`.

3) Ecrire une fonction `supprime` qui prend en argument une liste `l` et un élément `elem` et qui renvoie la liste obtenue à partir de `l` en supprimant toutes les occurrences de `elem`.

   Exemple : `supprime([4, 7, 12, 4, 4, 0, 4, 5], 4)` vaut `[7, 12, 0, 5]`.

4) Ecrire une fonction `insere_milieu` qui prend en argument une liste `l` et un élément `elem` et qui renvoie la liste obtenue à partir de `l` en ajoutant `elem` "au milieu" de `l`, c'est-à-dire:
   1) Si la longueur de `l` est paire, l'élément est ajouté une fois, au milieu de la liste.
   2) Si la longueur de `l` est impaire: l'élément est ajouté de part à d'autre de l'élément central.

   Exemple : `insere_milieu([4, 7, 12, 3], 0)` vaut `[4, 7, 0, 12, 3]` et `insere_milieu([9, 3, 5, 6, 2], 1)` vaut `[9, 3, 1, 5, 1, 6, 2]`.

5) Ecrire une fonction `decoupe` qui prend en argument une liste de nombres `l` et un nombre `seuil` et qui renvoie deux listes: la première est obtenue à partir de `l` en ne gardant que les nombres inférieurs ou égaux à `seuil`; la deuxième est obtenue à partir de `l` en ne gardant que les nombres strictement supérieurs à `seuil`.

   Exemple : `decoupe([14, 27, 12, 0, 40, 34, 20, 11], 20) a pour valeur de retour [14, 12, 0, 20, 11] et [27, 40, 34].`