Chapitre I

Révisions 1 : Rappels élémentaires de programmation

Sommaire

1 Variables informatiques : manipulation de base

Une variable informatique est l'élément de base de stockage des informations dans un programme informatique. Elle est généralement désigné par une lettre (au minimum) **a,b,c,x,y etc...**, ou un assemblage de plusieurs lettres **ab,bf0, xy, var6, essai etc....**

Lorsque l'on stocke une valeur dans une variable, cette dernière désigne en fait une adresse dans la mémoire dans laquelle est stockée la valeur. On appelle cela un pointeur.
Pour connaître la correspondance entre le nom choisi de la variable et l'adresse mémoire correspondant à la valeur stockée, Python exploite une table d'allocation mémoire.

En Python, l'affectation d'une valeur à un nom de variable se fait par le signe " =", et l'interrogation de la valeur pour affichage par simple rappel du nom de la variable :

```
>>> x = 9
>>> x
9
```

Affectation simultanée :

```
>>> x,a = 9,3
>>> x
9
>>> a
3
```

Echange de deux variables :

```
>>> x,a = a,x
>>> x
3
>>> a
9
```

QUESTION : Comment fait-on la même chose dans un script, c'est à dire un programme Python ?

RÉPONSE : Seul l'affichage se fait par une nouvelle commande : *print*

Listing I.1 –

```
1 a,b = 9,3
2 print "La variable a contient:",a,"et la variable b contient:",b
3 a,b=b,a
4 print "La variable a contient:",a,"et la variable b contient:",b
```

qui donne la sortie suivante :

```
La variable a contient : 9 et la variable b contient : 3
La variable a contient : 3 et la variable b contient : 9
```

EXERCICE N°1: *Jouons un peu avec l'addition ! Interpréter les scripts suivants "à la main" et expliquer leur rôle :*

Listing I.2 –

```
a,b = 9,3
print "La variable a contient:",a," et la variable b contient:",b
c=a
a=b
b=c
print "La variable a contient:",a," et la variable b contient:",b
```

Listing I.3 –

```
a,b = 9,3
print "La variable a contient:",a," et la variable b contient:",b
a=a+b
b=a-b
a=a-b
print "La variable a contient:",a," et la variable b contient:",b
```

INTÉRÊT DU SECOND SCRIPT : économie de mémoire car occupation de celle-ci par 2 variables et non 3 seulement !

2 Structures conditionnelles : *if, elif, else, elif*

Les structures conditionnelles sont des *séquences d'instructions* que Python réalise uniquement lorsqu'une condition est vérifiée. Ces mots clé sont *if, else, elif*.

La condition à vérifier est souvent formulée avec un opérateur de comparaison renvoyant le résultat booléen False ou True :

égal à	$\Leftrightarrow$	==
différent de	$\Leftrightarrow$	!=
plus grand que	$\Leftrightarrow$	>
plus petit que	$\Leftrightarrow$	<
supérieur ou égal à	$\Leftrightarrow$	>=
inférieur ou égal à	$\Leftrightarrow$	<=

Listing I.4 –

```
# Donne le signe d'un entier relatif
print "Entrer un nombre entier relatif:"
n=input()
if type(n)!=int: #vérifie si n est entier avec l'opérateur booléen "!=" différent de
    print"n n'est pas un entier"
elif n==0: #sinon vérifie si n est nul avec l'opérateur booléen "==" égal à
    print "n est l'entier nul"
elif n>0:
    print "n est entier positif"
else: # sinon
    print "n est négatif"
print"Fin d'execution"
```

Ce qui donne par exemple pour n=2 :

```
Entrer un nombre entier relatif :
2
n est un entier positif
Fin d'execution
```

et pour n=-2 :

```
Entrer un nombre entier relatif :
-2
n est un entier négatif
Fin d'execution
```

NB : les conditions après un *elif* sont testées uniquement si les conditions antérieures n'ont pas été vérifiées.

3 Boucle conditionnelle *while*

La commande *while* permet de réaliser une série d'instructions de manière itérative dans une boucle dite **conditionnelle**, c'est à dire tant qu'une condition est vérifiée. L'exemple ci-dessous, qui réinvestit un test *if* calcule la factorielle de n à l'aide d'une boucle *while* :

Listing I.5 –

```
print"Entrer la valeur d'un entier positif"
n=input()
if type(n)!=int: # Teste si n n'est pas un entier ("vrai" si le type de n n'est pas 'int'
    print"vous n'avez pas entré un entier"
else: #
    fact=1
    while n>0: #teste la condition sur n et stoppe la boucle dès que n=0
        fact=fact*n
        n=n-1
    print(fact)
```

Un autre exemple plus amusant qui donne l'heure courante tant que la demande de sortie du programme n'est pas ordonnée par l'utilisateur :

Listing I.6 – L'horloge non parlante!!!

```
import time      # importation du module time
quitter = 'n'    # initialisation
while quitter != 'o':
    # ce bloc est exécuté tant que la condition est vraie
    # strftime() est une fonction du module time
    print('Heure courante ',time.strftime('%H:%M:%S'))
    quitter = input("Voulez-vous quitter le programme (o/n) ? ")
print("A bientôt")
```

4 Boucle inconditionnelle *for*

4.1 Préliminaire : commande *range* et les listes

La commande *range* permet de générer une **liste** de nombres. Elle possède plusieurs syntaxes possibles. Par exemple, pour générer une liste composée des dix premiers entiers naturels, on peut écrire :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On remarque que dans la seconde syntaxe proposée, le second argument (10) est encore exclu de la liste. On peut également compter à rebourd, **et nous avons toujours ce souci avec le dernier nombre de la liste** :

```
>>> range(10,1,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
```

4.2 Commande *for*

La commande *for* permet de réaliser une série d'instructions de manière itérative dans une boucle inconditionnelle un nombre de fois déterminé. Le compteur de boucle prend ces valeurs successives dans une liste indiquée comme argument selon la syntaxe suivante :

Listing I.7 – Calcul de n !

```
print "Entrer un nombre entier:"
n=input()
L=range(1,n+1) #On remarque la nécessité de mettre n+1 en second argument pour parcourir tous les entiers jusqu'à n et non pas n-1
fact=1
for i in L:
    fact*=i #Formulation "compacte" du calcul  et de l'affectation
print(fact)
```

Un autre exemple simple est l'algorithme de chiffrement élémentaire dit Code de César qui consiste à décaler de n rangs d'alphabet toutes les lettres d'un mot. n constitue ce que l'on appelle la clé. Cryptons par exemple le mot **"bonjour"**en choisissant $n = 3$:

Listing I.8 – Le code de César

```
def cesar(ch):
    alphabet="abcdefghijklmnopqrstuvwxyz"
    res=""
    for c in ch:
        for i in range(26):
            if c==alphabet[i]:
                res=res+alphabet[(i+3)%26] #permet une rotation circulaire sur les 26 lettres (ie décalage de 3 modulo 26)
    return res
print cesar("bonjour")
```

On peut également combiner structure conditionnelle et boucle inconditionnelle. L'exercice qui suit en propose un exemple :

EXERCICE N°2: *Extraction selon critère*

On donne une liste de nombres stockés dans la variable liste.
Ecrire un programme permettant :

- *d'extraire de* liste *les éléments positifs et les classer par ordre croissant dans une nouvelle liste* listep
- *d'afficher cette liste une fois réalisée et d'indiquer le nombre de ces éléments.*
- *d'indiquer si la liste est vide le cas échéant*

5 Fonctions

5.1 Principe de définition

On peut être amené dans un programme à faire appel plusieurs fois à la même suite d'instructions. Il est alors possible de définir une procédure désignée par un nom pour executer ce bloc d'instructions. Cet objet porte le nom

impropre de **fonction** même si l'appellation **procédure** lui conviendrait mieux dans notre langue française.

Définissons par exemple une fonction permettant de calculer la factorielle d'un entier entré au clavier :

Listing I.9 –

```
def factorielle(n):
    if type(n)!=int:
        print"n'est pas un entier!!!"
    else:
        f = 1
        for i in range(1, n+1):
            f *= i
    return(f)#renvoie le résultat de la fonction
for i in range(5):
    print (factorielle(i+1)) #+1 toujours en raison de ce décalage d'indice dans une
    liste générée par range
```

```
1
2
6
24
120
```

Nous pouvons également transmettre ses arguments à une fonction sans que le nombre de ces derniers soit nécessairement défini. On ajoute alors "*" devant le nom générique choisi pour les arguments.

La fonction qui suit calcule par exemple la somme alternée d'une série d'arguments numériques entiers transmis en nombre quelconque :

Listing I.10 –

```
def sommeAlt(*args):
    s=0
    sg=1
    badtest=0
    for i in args:
        if type(i)!=int:
            print"Attention: au moins un de vos arguments n'est pas un entier!"
            badtest=1
        else:
            s+=sg*i
            sg=-sg
    if badtest==0:
        return s
print sommeAlt(1,2,3),sommeAlt(4,5,6)
```

```
2 5
```

5.2 Portée ou visibilité des variables

a - Cas simple : variables locales et globales dans les fonctions

On appelle **portée des variables** le domaine de visibilité de ces dernières au sein de la définition d'une fonction. On distingue les variable locales à une fonction des variables globales.

Les exemples suivants illustre cela :

Listing I.11 –

```
x = 42
def f():
    return x #renvoie la valeur de x variable globale soit 42
def g():
    x = 3
    return x #renvoie la valeur de x locale car définie dans le programme soit 3
def h():
    global x
    x = 17
    return x #renvoie la valeur de x globale modifiée dans le programme
```

Ce qui donne la sortie suivante :

```
>>> f()
42
>>> g()
3
>>> x
42
>>> h()
17
>>> x
17
```

b - Cas plus "fin" : variables locales dans les sous-procédures ou sous-fonctions

Une sous-procédure ou sous-fonction consiste en une procédure ou fonction définie à l'intérieur même d'une procédure ou fonction.

A RETENIR :

PROPRIÉTÉ - (5.2) - 1:

- *une sous-fonction n'est pas appelable en dehors de la fonction dans laquelle elle est incluse.*
- *les variables locales de la sous-fonction ne sont pas visibles de l'extérieur, donc ni de la fonction dans laquelle elle est incluse, et fatalement pas non plus de l'extérieur des deux fonctions.*
- *comme toujours, les variables définies comme globales sont visibles de partout.*

Listing I.12 – Variable locale

```
### Définitions fonction et sous-fonction ###
def fonction1():
    x=1
    def fonction2():
        x=2
        print "au niveau 2, on a x=",x
        return None
    ### Appel à la sous-fonction ###
    fonction2()
    print "au niveau 1 (après appel à fonction2), on a x=",x
    return None
```

```
>>> fonction1()
au niveau 2, on a x=2
au niveau 1 (après appel à fonction2), on a x=1
```

Listing I.13 – Variable globale

```
### Définitions fonction et sous-fonction ###
def fonction1():
    global x
    x=1
    def fonction2():
        global x
        x=2
        print "au niveau 2, on a x=",x
        return None
    ### Appel à la sous-fonction ###
    fonction2()
    print "au niveau 1 (après appel à fonction2), on a x=",x
    return None
```

```
>>> fonction1()
au niveau 2, on a x=2
au niveau 1 (après appel à fonction2), on a x=2
```