Programmer en Python

Le résumé du résumé

Henri Garreta
Université d'Aix-Marseille. Faculté des Sciences. Département d'Informatique.

Table des matières

Ce document a été produit le 5 mars 2014.
A tout moment, la version la plus récente peut être téléchargée à l'adresse `http://h.garreta.free.fr`

1 Mise en place

1.1 Obtenir Python

Le site officiel du langage Python est : `http://www.python.org/`. On peut y télécharger librement la dernière version du logiciel [1] (l'interpréteur et les bibliothèques) pour la plupart des systèmes d'exploitation. La documentation officielle, très copieuse, peut également être parcourue ou téléchargée à partir de ce site.

L'excellent livre d'initiation *Apprendre à programmer avec Python*, de Gérard Swinnen, a été publié chez O'Reilly [2]. Il peut aussi être librement téléchargé à partir de l'adresse `http://inforef.be/swi/python.htm`. Vous pouvez également en acheter un tirage sur papier chez *Planète Image* dans le hall de la faculté.

D'autres liens utiles concernant Python et sa documentation sont donnés dans mes pages `http://henri.garreta.perso.luminy.univmed.fr/PythonLicPro` et `http://henri.garreta.perso.luminy.univmed.fr/PythonBBSG`. On y trouve notamment la traduction en français du très utile *Tutoriel Python*.

Bien que ce ne soit pas strictement indispensable, pour programmer en Python il est recommandé de se munir d'un environnement de développement perfectionné. Nous recommandons *Eclipse* (`http://www.eclipse.org/downloads/`, contentez-vous de télécharger le produit nommé *Eclipse IDE for Java Developers*) étendu par le « plugin » *Pydev* (`http://pydev.org/`).

1. En septembre 2013, les versions officielles de Python sont les versions 3.3.2 et 2.7.5. Attention, il s'agit de deux branches incompatibles : la version 3 est certainement « meilleure », mais certaines bibliothèques spécialisées ne fonctionnent qu'avec la version 2.

2. Hélas, les éditions *O'Reilly France* ont été fermées définitivement (cf. `http://www.oreilly.fr/`).

1.2 Utiliser Python

Que ce soit sur Windows ou sur UNIX et Linux, la mise en place de Python ne pose aucun problème. Après une installation réussie, vous avez surtout deux manières d'utiliser l'interpréteur :

1°) *Interactivement*, en saisissant des commandes Python l'une après l'autre. Par exemple, voici une session interactive dans une console Linux pour effectuer quelques divisions – dont certaines peuvent surprendre ("`bash_$`" est l'invite de mon système Linux, "`>>>`" celle de l'interpréteur Python ; les textes tapés par l'utilisateur ont été reproduits en caractères penchés) :

```
bash_$ python
Python 2.3.4 (#1, Dec 11 2007, 05:27:57)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 100 / 3
33
>>> 100 / 3.0
33.333333333333336
>>> 100 / -3
-34
>>> 100 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> (ici on a tapé Ctrl-D)
bash_$
```

Notez que sur Windows, le programme *IDLE*, qui s'installe en même temps que Python, ouvre une console particulièrement bien adaptée à la pratique de Python.

2°) En exécutant un *programme* (on dit volontiers un *script*) préalablement saisi. Par exemple, si on a écrit un script, dans un fichier nommé `factorielle.py`, pour évaluer la factorielle $n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$ d'un nombre n lu au clavier, voici comment en obtenir l'exécution :

```
bash_$ python factorielle.py
nombre ? 40
40 ! = 815915283247897734345611269596115894272000000000
bash_$
```

Si vous lisez déjà le Python, le texte du fichier `factorielle.py` est visible à la section 5.4, page 20.

1.3 Se faire aider par *eclipse*

Une fois installés *eclipse* (il suffit de décompresser le fichier `eclipse.zip` téléchargé) et *Pydev* (là c'est plus compliqué ; suivez les instructions de la rubrique *Getting started* dans `http://pydev.org/manual.html` ou bien lisez ma page `http://henri.garreta.perso.luminy.univmed.fr/PythonLicPro/eclipse_python.html`), pour développer en Python il faut

1°) Activer *eclipse* en lançant le fichier nommé `eclipse.exe` sur Windows, `eclipse` ailleurs. Au bout d'un moment on obtient l'interface graphique représentée sur la figure 1.

2°) Lui faire adopter la perspective *Pydev* (menu *Window* ▷ *Open Perspective* ▷ *Other...* ▷ *Pydev* ou bien la petite icône en haut à droite de la fenêtre principale).

3°) La première fois, créer un nouveau projet (menu *File* ▷ *New* ▷ *Pydev Project*). Dans l'exemple de la figure 1, le projet a été nommé *Atelier Python*.

4°) Créer un fichier source (menu *File* ▷ *New* ▷ *Pydev Module*). Dans l'exemple de la figure 1, le module s'appelle `factorielle` (le fichier source s'appelle donc `factorielle.py`).

Vous n'avez plus qu'à saisir le code du script en question. Constatez comment les erreurs sont signalées dès leur frappe et comment dans beaucoup d'occasions eclipse vous suggère les mots corrects qu'il vous est permis d'écrire.

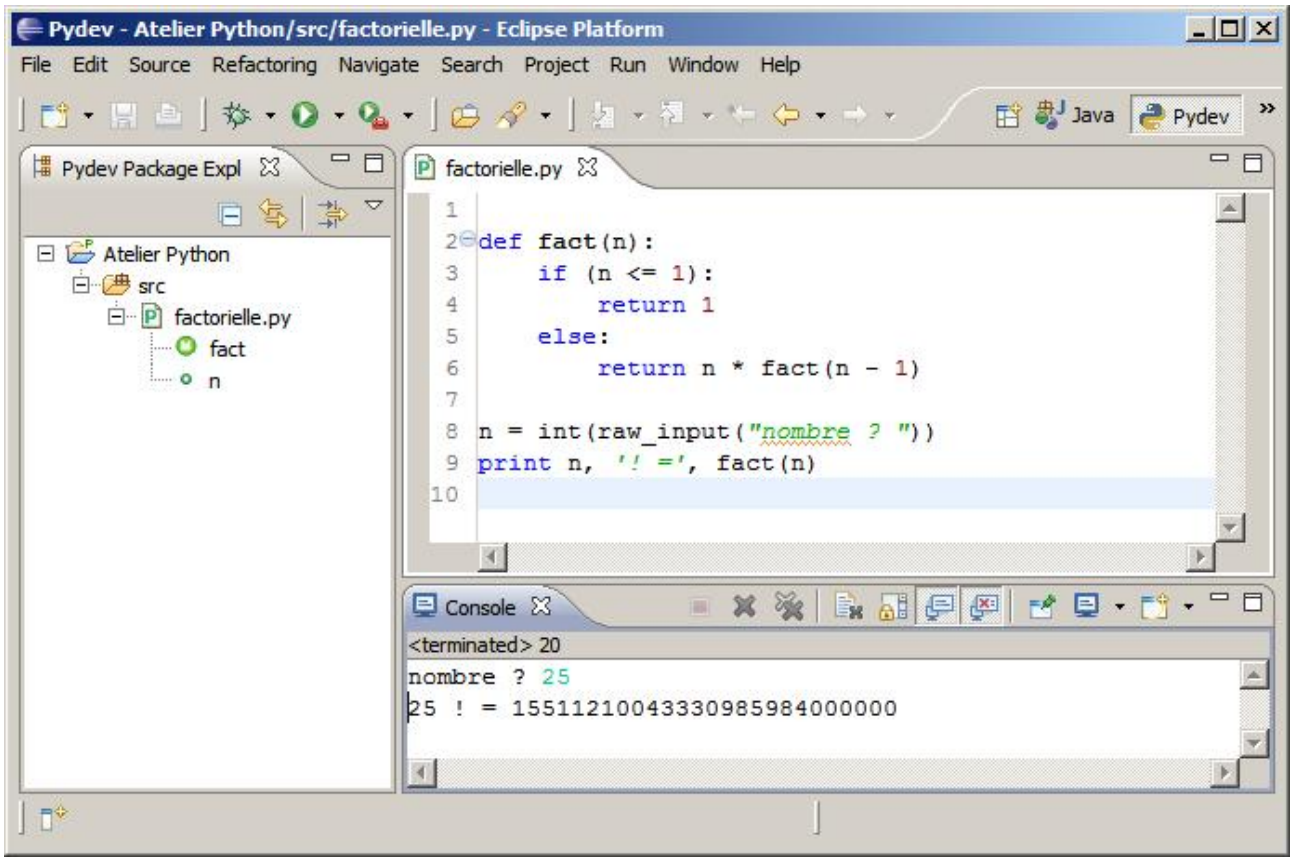

FIGURE 1 – Utilisation d'eclipse pour programmer en Python

5°) Pour essayer le programme écrit, assurez-vous que la fenêtre contenant son texte source est active, puis actionnez la commande du menu *Run* ▷ *Run As* ▷ *Python Run* ou bien, si le script en question a déjà été exécuté une fois, actionnez le bouton vert avec un triangle blanc en haut de la fenêtre.

2 Expressions

2.1 Constantes numériques

Une *constante littérale* est l'expression écrite d'une valeur connue ; exemples : `12`, `-5`, `3.5`, `0`, etc.

La donnée représentée par une constante a toujours un *type* qui détermine ses propriétés formelles (comme : quelles opérations la donnée peut-elle subir ?) et matérielles (comme : combien d'octets la donnée occupe-t-elle dans la mémoire de l'ordinateur ?). La manière la plus simple *et la plus fiable* de connaître le type d'une expression consiste à poser la question à Python. Exemple d'une session de telles questions et réponses :

```
>>> type(0)
<type 'int'>
>>> type(-5)
<type 'int'>
>>> type(2000000000)
<type 'int'>
>>> type(3000000000)
<type 'long'>
>>> type(-5.0)
<type 'float'>
>>> type(602.21417e+021)
<type 'float'>
>>>
```

En ignorant la présentation `<type 'un type'>`, délibérément biscornue, le dialogue précédent nous apprend, ou du moins nous suggère, que

- sans surprise, des constantes comme 0, -5 ou 2000000000 représentent des nombres entiers (type `int`),
- lorsqu'un nombre entier est trop grand pour être représenté comme un entier ordinaire, Python le code automatiquement comme un *entier long* (type `long`) ; c'est très pratique, mais il faut savoir que les opérations arithmétiques sur de tels nombres sont moins rapides que celles sur les entiers ordinaires,
- dès qu'une constante numérique comporte un point, qui joue le rôle de virgule décimale, Python comprend qu'il s'agit d'un nombre *décimal*, on dit plutôt *flottant*, et le représente en machine comme tel (type `float`),
- lorsque les nombres décimaux sont très grands ou très petits on peut employer la « notation scientifique » bien connue ; par exemple, la constante `602.21417e+021` représente le nombre $6,0221417 \times 10^{23}$ ou encore 602214170000000000000000,
- le caractère flottant d'un nombre est « contagieux » : si une opération arithmétique a un opérande entier et un opérande flottant, alors pour effectuer l'opération l'entier est d'abord converti en flottant et l'opération est faite selon les règles des nombres flottants ; exemple : le résultat de la multiplication `1.0 * 5` est le nombre flottant `5.0`.

2.2 Variables

Un *identificateur* est une suite de lettres et chiffres qui commence par une lettre et n'est pas un mot réservé (comme `if`, `else`, `def`, `return`, etc.). Le caractère `_` est considéré comme une lettre. Exemples : `prix`, `x`, `x2`, `nombre_de_pieces`, `vitesseDuVent`, etc. Majuscules et minuscules n'y sont pas équivalentes : `prix`, `PRIX` et `Prix` sont trois identificateurs distincts.

Une *variable* est constituée par l'association d'un identificateur à une valeur. Cette association est créée lors d'une *affectation*, qui prend la forme

variable `=` *valeur*

A la suite d'une telle affectation, chaque apparition de la variable ailleurs que dans la partie gauche d'une autre affectation représente la valeur en question... jusqu'à ce qu'une nouvelle affectation associe une autre valeur à la variable. On confond parfois le nom de la variable (c.-à-d. l'identificateur) et la variable elle-même (l'*association* du nom à une valeur) ; c'est sans gravité.

Si un identificateur n'a pas été affecté (en toute rigueur il n'est donc pas un nom de variable) son emploi ailleurs qu'au membre gauche d'une affectation est illégale et provoque une erreur. Session Python de démonstration :

```
>>> nombre
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'nombre' is not defined
>>> nombre = 10
>>> nombre
10
>>> nombre = 20.5
>>> nombre
20.5
>>>
```

Comme on le voit sur cet exemple, en Python :

1. *Les variables n'ont pas besoin d'être déclarées* (c'est-à-dire préalablement annoncées), la première affectation leur tient lieu de déclaration.

2. *Les variables ne sont pas liées à un type* (mais les valeurs auxquelles elles sont associées le sont forcément) : la même variable `nombre` a été associée à des valeurs de types différents (un `int`, puis un `float`).

2.3 Chaînes de caractères

Une donnée de type *chaîne de caractères* est une suite de caractères quelconques. Une *constante chaîne de caractères* s'indique en écrivant les caractères en question soit entre apostrophes, soit entre guillemets : `'Bonjour'` et `"Bonjour"` sont deux écritures correctes de la même chaîne.
Si la chaîne doit contenir un des caractères `'` ou `"` cela fournit un critère pour choisir l'une ou l'autre manière de l'écrire : `'"Oui" dit-il'` ou `"L'un ou l'autre"`. Une autre manière d'éviter les problèmes avec le caractère d'encadrement consiste à l'inhiber par l'emploi de `\`, comme dans la chaîne `'Il n\'a pas dit "oui"'`.

L'encadrement par de triples guillemets ou de triples apostrophes permet d'indiquer des chaînes qui s'étendent sur plusieurs lignes :

```
>>> s = """Ceci est une chaîne
comportant plusieurs lignes. Notez que
    les blancs en tête de ligne
sont conservés"""
>>> s
'Ceci est une chaîne\ncomportant plusieurs lignes. Notez
que\n    les blancs en tête de ligne\nsont conservés'
>>>
```

L'affichage basique d'une telle chaîne est décevant (voir ci-dessus) mais montre bien que les fins de ligne, représentés par le signe `\n`, ont été conservés. L'affichage à l'aide de la fonction `print` est plus esthétique :

```
>>> print s
Ceci est une chaîne
comportant plusieurs lignes. Notez que
    les blancs en tête de ligne
sont conservés
>>>
```

Concaténation. L'opérateur `+` appliqué à deux chaînes de caractères produit une nouvelle chaîne qui est la *concaténation* (c'est-à-dire la mise bout-à-bout) des deux premières :

```
>>> 'Nabucho' + 'donosor'
'Nabuchodonosor'
>>>
```

Accès à un caractère individuel. On accède aux caractères d'une chaîne en considérant celle-ci comme une séquence indexée par les entiers : le premier caractère a l'indice 0, le second l'indice 1, le dernier l'indice $n-1$, n étant le nombre de caractères. Exemples :

```
>>> s = 'Bonjour'
>>> s[0]
'B'
>>> s[6]
'r'
>>> s[-1]
'r'
>>>
```

Le nombre de caractères d'une chaîne s est donné par l'expression `len(`s`)`. Pour accéder aux caractères à la fin d'une chaîne il est commode d'employer des indices négatifs : si i est positif, s`[-`i`]` est la même chose que s`[len(`s`) - `i`]`.

Tranches. On peut désigner des *tranches* dans les chaînes ; la notation est *chaîne*`[`*début*`:`*fin*`]` où *début* est l'indice du premier caractère dans la tranche et *fin* celui du premier caractère en dehors de la tranche. L'un et l'autre de ces deux indices peuvent être omis, et même les deux. De plus, ce mécanisme est tolérant envers les indices trop grands, trop petits ou mal placés :


```
>>> s = 'Bonjour'
>>> s[3:5]
'jo'
>>> s[3:]
'jour'
>>> s[:5]
'Bonjo'
>>> s[:]
'Bonjour'
>>> s[6:3]
''
>>> s[-100:5]
'Bonjo'
>>> s[3:100]
'jour'
>>>
```

Les chaînes de caractères sont immuables : une fois créées il ne peut rien leur arriver. Pour modifier des caractères d'une chaîne on doit construire une *nouvelle* chaîne qui, si cela est utile, peut remplacer la précédente. Par exemple, proposons-nous de changer en 'J' le 'j' (d'indice 3) de la chaîne précédente :

```
>>> s
'Bonjour'
>>> s[3] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> s = s[:3] + 'J' + s[4:]
>>> s
'BonJour'
>>>
```

2.4 Opérateurs

Opérateurs arithmétiques.

`+`, `-`, `*` : addition, soustraction, multiplication. Ces opérateurs obéissent à la règle dite « du plus fort » : si un des opérandes est flottant, l'autre opérande est converti si nécessaire en flottant et l'opération et le résultat sont flottants. Si, au contraire, les deux opérandes sont entiers, l'opération et le résultat sont entiers.

`/` : division. Obéit à la même règle que les trois précédents, mais on prendra garde au fait que cela peut être surprenant : `3/2` vaut `1` et `2/3` vaut `0`.
Pour obtenir un résultat flottant il faut se débrouiller pour qu'au moins un des opérandes soit flottant : `3.0/2` ou `float(3)/2` valent bien `1.5`.
Attention : `float(3/2)` vaut `1.0`, car la conversion en `float` intervient après la perte des décimales.

`%` : reste de la division euclidienne, celle qu'on apprend à l'école élémentaire [3] : si a et b sont entiers alors a/b est le quotient et $a\%b$ le reste de la division de a par b.
En toute rigueur cette opération ne devrait être définie que pour deux opérandes entiers, mais Python se débrouille pour la faire marcher même avec des opérandes flottants.

`**` : puissance : `a**b` vaut a^b, c'est-à-dire $a \times a \times \cdots \times a$ si b est entier, $e^{b \times log\ a}$ sinon.

Priorité des opérateurs. Les opérateurs *multiplicatifs* `*`, `/` et `%` ont une *priorité supérieure* à celle des opérateurs *additifs* `+` et `-`. Cela signifie, par exemple, que l'expression $a * b + c$ exprime une addition (dont le premier terme est le produit $a * b$). L'emploi de parenthèses permet de contourner la priorité des opérateurs. Par exemple, l'expression $a * (b + c)$ représente bien une multiplication (dont le second facteur est l'addition $b + c$).

3. Étant donnés deux nombres entiers a et b, avec $b > 0$, faire la division euclidienne de a par b consiste à déterminer deux nombres entiers q et r vérifiant $a = b \times q + r$ et $0 \leq r < b$. On dit alors que q est le quotient et r le reste de la division de a par b. En Python on a donc `a/b` $= q$ et `a%b` $= r$.

Heureusement, ce qu'il faut savoir en programmation à propos de la priorité des opérateurs coïncide avec la pratique constante en algèbre (rappelez-vous « *une expression est une somme de termes, un terme est un produit de facteurs* ») ; c'est pourquoi nous ne développerons pas davantage cette question ici.

Opérateurs de comparaison.

`==`, `!=` : égal, non égal.
`<`, `<=`, `>`, `>=` : inférieur, inférieur ou égal, supérieur, supérieur ou égal.

La règle du plus fort vaut aussi pour les comparaisons : les deux opérandes sont amenés à un type commun avant l'évaluation de la comparaison.

Conformément à l'habitude répandue, la priorité des opérateurs de comparaison est *inférieure* à celle des opérateurs arithmétiques. Ainsi, l'expression `a == b + c` signifie bien `a == (b + c)`.

Booléens et opérateurs booléens. Le résultat d'une comparaison comme *a* `==` *b* est de type `booléen` ; ce type ne possède que deux valeurs, `False` (faux) et `True` (vrai).

A certains endroits, comme les conditions des instructions `if` ou `while`, le langage prescrit la présence d'expressions dont le résultat doit être de type booléen. Par une tolérance ancienne, maintenant découragée, on peut mettre à la place une expression quelconque ; elle fonctionnera comme un booléen à l'aide de la convention suivante :

– seront tenus pour *faux* : `False`, `None`, le zéro de n'importe quel type numérique (`0`, `0L`, `0.0`), la chaîne de caractères vide et toute structure de données vide (liste, ensemble, table associative, etc.),
– toute autre valeur sera tenue pour *vraie*.

Les connecteurs logiques sont

– `not` (négation) : `not` *a* est vraie si et seulement si *a* est fausse,
– `and` (conjonction) : *a* `and` *b* est vraie si et seulement si *a* et *b* sont toutes deux vraies,
– `or` (disjonction) : *a* `or` *b* est vraie si et seulement si au moins une deux expressions *a*, *b* est vraie.

Notez que les opérateurs `and` et `or` sont « optimisés » de la manière suivante : le premier opérande est évalué d'abord ; si sa valeur permet de prédire le résultat de l'opération (c.-à-d. si *a* est faux dans *a* `and` *b* ou vrai dans *a* `or` *b*) alors le second opérande n'est même pas évalué. Cela permet d'écrire des expressions dont le premier terme « protège » le second, comme (rappelez-vous que diviser par zéro est une erreur qui plante le programme) :

```
if (y != 0) and (x / y > M) ...
```

3 Structures de contrôle

Python et l'indentation. L'*indentation* (c'est-à-dire la marge blanche laissée à gauche de chaque ligne) joue un rôle syntaxique et permet d'économiser les accolades {...} et les *begin...end* d'autres langages. En contrepartie, elle obéit à des règles strictes :

1. Une *séquence* d'instructions (instructions consécutives, sans subordination entre elles) est faite d'instructions écrites avec la *même indentation*.

2. Dans une structure de contrôle (instruction `if`, `for`, `def`) une séquence d'instructions *subordonnées* doit avoir une *indentation supérieure* à celle de la première ligne de la structure de contrôle. Toute ligne écrite avec la même indentation que cette première ligne marque la fin de la séquence subordonnée.

3.1 Instruction conditionnelle

Syntaxe :

```
if condition_1 :
    instruction_1
    ...
    instruction_p
elif condition_2 :
    instruction_{p+1}
    ...
    instruction_q
else :
    instruction_{q+1}
    ...
    instruction_r
```

L'organigramme de la figure 2 explique l'exécution de l'instruction conditionnelle.

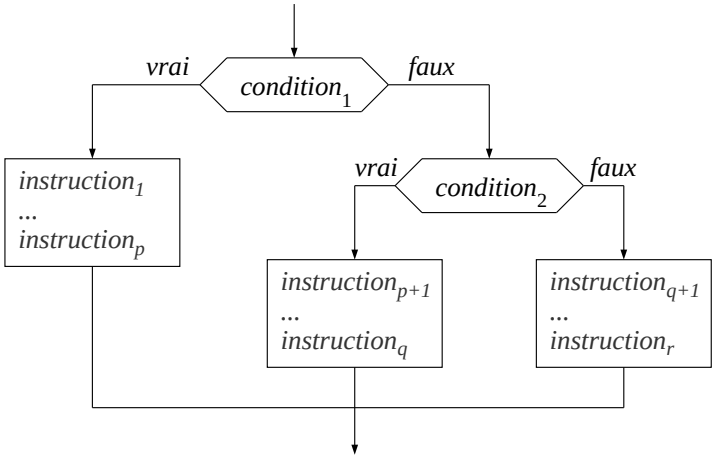

FIGURE 2 – Organigramme de l'instruction conditionnelle

Le groupe de lignes de « `elif` *condition*$_2$ `:` » à « *instruction*$_q$ » peut apparaître un nombre quelconque de fois ou être absent. Le groupe de lignes qui va de « `else :` » à « *instruction*$_r$ » peut être absent.

• **Exemple de script 1**. Résolution d'une équation du second degré : étant donnés trois nombres a, b et c, avec $a \neq 0$, trouver les valeurs de x vérifiant $ax^2 + bx + c = 0$. Algorithme bien connu : examiner le signe de $\Delta = b^2 - 4ac$. Si $\Delta > 0$ il y a deux solutions réelles $x_1 = \frac{-b+\sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b-\sqrt{\Delta}}{2a}$. Si $\Delta = 0$ il y a une seule solution, $x = \frac{-b}{2a}$. Si $\Delta < 0$ il n'y a pas de solution (ou, du moins, pas de solution réelle). Programme :

```
from math import *

a = float(raw_input("a : "))
b = float(raw_input("b : "))
c = float(raw_input("c : "))

delta = b * b - 4 * a * c
if delta > 0:
    x1 = (-b + sqrt(delta)) / (2 * a)
    x2 = (-b - sqrt(delta)) / (2 * a)
    print 'deux solutions:', x1, x2
```

```
elif delta == 0:
    x = -b / (2 * a)
    print 'une solution:', x
else:
    print 'pas de solution reelle'
```

3.2 Boucle *tant que*

```
while condition_1 :
    instruction_1
    ...
    instruction_p
else :
    instruction_{p+1}
    ...
    instruction_q
```

L'organigramme de la figure 3 explique l'exécution de cette instruction.

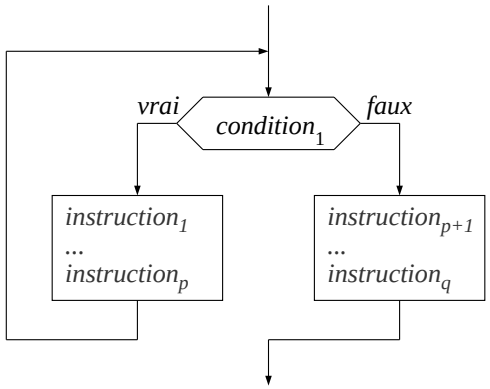

FIGURE 3 – Organigramme de la boucle *while*

La partie qui va de « `else :` » à « $instruction_q$ » peut être absente. Pratiquement, elle n'a d'intérêt que si une instruction `break` est présente dans la boucle.

• **Exemple de script 2**. Étant donné un nombre positif a, trouver le plus petit entier k vérifiant $a \leq 2^k$. Algorithme naïf (il y a plus efficient) : construire successivement les termes de la suite 2^k ($2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$...) jusqu'à ce qu'ils égalent ou dépassent la valeur de a. Programme :

```
a = input("a : ")
k = 0
p = 1           # on veille à avoir constamment p = 2**k
while p < a:
    p = p * 2
    k = k + 1
print 'k:', k, ' 2^k:', p
```

3.3 Boucle *pour*

Syntaxe :

```
for variable in séquence :
    instruction_1
    ...
    instruction_p
else :
    instruction_{p+1}
    ...
    instruction_q
```

La partie qui va de « `else :` » à « $instruction_q$ » peut être absente. Pratiquement, elle n'a d'intérêt que si une instruction `break` est présente dans la boucle.

Fonctionnement : si la *séquence* indiquée est composée des éléments $valeur_1$, $valeur_2$, ... $valeur_k$ alors l'exécution de la structure précédente revient à l'exécution de la séquence :

```
variable = valeur_1
instruction_1
...
instruction_p
variable = valeur_2
instruction_1
...
instruction_p
...

variable = valeur_k
instruction_1
...
instruction_p
instruction_{p+1}
...
instruction_q
```

Ainsi, la boucle *for* de Python prend toujours la forme du parcours d'une séquence préexistante. Pour obtenir la forme plus classique « pour i valant successivement a, $a + p$, $a + 2 \times p$, ... b » (dont un cas particulier archi-fréquent est « pour i valant successivement 0, 1, ... k ») on emploie la fonction `range` qui construit une séquence sur demande :

```
for variable in range(a, b, p):
    instruction_1
    ...
    instruction_p
```

Pour cerner la fonction `range` on peut s'en faire montrer le fonctionnement par Python. Session :

```
>>> range(10, 20, 3)
[10, 13, 16, 19]
>>> range(10, 20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>>
```

VARIANTE. La fonction `range` construit effectivement en mémoire la séquence demandée ; on peut regretter l'encombrement qui en résulte, alors qu'il nous suffirait de disposer *successivement* de chaque élément de la séquence. La fonction `xrange` fait cela : vis-à-vis de la boucle *for* elle se comporte de la même manière que `range`, mais elle ne construit pas la séquence :

```
>>> range(-35, +35, 5)
[-35, -30, -25, -20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30]
>>> xrange(-35, +35, 5)
xrange(-35, 35, 5)
>>> for i in xrange(-35, +35, 5): print i,
-35 -30 -25 -20 -15 -10 -5 0 5 10 15 20 25 30
>>>
```

3.4 *Break* et *else* dans les boucles

L'instruction `break` placée à l'intérieur d'une boucle *while* ou *for* produit l'abandon immédiat de la boucle et la continuation de l'exécution par la première instruction qui se trouve *après* la boucle. La clause `else`, si elle est présente, n'est alors pas exécutée.

• **Exemple de script 3**. Chercher l'indice de la première occurrence d'un caractère c dans une chaîne s[4] ; renvoyer la valeur de cet indice ou -1 si c n'est pas présent dans s :

```
acquisition de la chaîne s et du caractère c
for i in range(len(s)):
    if s[i] == c:
        break
else:
    i = -1
ici, la valeur de i répond exactement à la question
```

Attention, piège ! Notez que `else` est indenté comme `for`. L'aligner avec `if` aurait complètement changé le sens de ce bout de code et l'aurait rendu faux.

4 Structures de données

4.1 Tuples

Un *tuple* est une *séquence immuable* : on ne peut pas modifier ses éléments ni lui en ajouter ou lui en enlever. En contrepartie de cette rigidité les tuples sont très compacts (i.e. ils occupent peu de mémoire) et l'accès à leurs éléments est très rapide.

On crée un tuple en écrivant ses éléments, séparés par des virgules et encadrés par des parenthèses. Si cela ne crée aucune ambiguïté, les parenthèses peuvent être omises. Un tuple constitué d'un seul élément a doit être écrit « `a,` » ou « `(a,)` ». Le tuple sans éléments se note « `()` ».

Dans les exemples suivants, la valeur de la variable `t` est un tuple de cinq éléments :

```
>>> t = 2, 'deux', 2.0, True, (1, 2, 3, 4)
>>> t
(2, 'deux', 2.0, True, (1, 2, 3, 4))
>>> t, type(t), len(t)
((2, 'deux', 2.0, True, (1, 2, 3, 4)), <type 'tuple'>, 5)
>>> t[1]
'deux'
>>> t[-1]
(1, 2, 3, 4)
>>>
```

Comme on le voit, un élément d'un tuple peut être un tuple à son tour. C'est tellement facile et pratique qu'on

4. Cet exemple est important, car il est l'archétype du problème *recherche linéaire d'une valeur dans une séquence*. Cependant, il faut savoir que s'agissant de la recherche d'un caractère dans une chaîne il n'est pas nécessaire d'écrire un programme, la fonction `s.find(c)` de la bibliothèque Python répond parfaitement à la question.

écrit des tuples sans s'en rendre compte : l'expression « `t, type(t), len(t)` » est elle-même un tuple de trois éléments.

Un tuple *de variables* peut figurer comme membre gauche d'une affectation, il faut alors que le membre droit soit un tuple de même taille :

```
>>> point = 3, 4
>>> point
(3, 4)
>>> x, y = point
>>> x
3
>>> y
4
>>>
```

4.2 Listes

Une liste est une *séquence modifiable* : on peut changer ses éléments, lui en ajouter et lui en enlever. Cela rend les listes un peu moins compactes et efficaces que les tuples, mais considérablement plus utiles pour représenter des structures de données qui évoluent au cours du temps.

On construit une liste en écrivant ses éléments, séparés par des virgules, encadrés par les crochets. La liste vide se note `[ ]`. Voici quelques opérations fondamentales sur les listes (voir aussi § 7.1) :

liste`[`*indice*`]`	désignation de l'élément de rang *indice* de la *liste*,
liste`[`*indice*`] =` *expression*	remplacement de l'élément de *liste* ayant l'*indice* indiqué,
$liste_1$ `+` $liste_2$	concaténation (mise bout-à-bout) de deux listes,
liste`.append(`*élément*`)`	ajout d'un *élément* à la fin d'une *liste*,
liste`.insert(`*indice*`,` *élément*`)`	insertion d'un *élément* à l'emplacement de *liste* indiqué par l'*indice* donné,
élément `in` *liste*	test d'appartenance : *élément* $\in$ *liste* ?
`for` *élément* `in` *liste*	parcours séquentiel (voir § 3.3).

N.B. Il y a donc deux manière de commander l'opération « ajouter un élément à la fin d'une liste ». On prendra garde à cette différence entre elles :

liste `+ [` *élément* `]`	ne modifie pas *liste*, mais renvoie une nouvelle liste comme résultat,
liste`.append(`*élément*`)`	modifie *liste* et ne renvoie pas de résultat.

• **Exemple de script 4**. Calcul de la liste des nombres premiers inférieurs ou égaux à une certaine borne. Algorithme : passer en revue chaque nombre impair (les nombres pairs ne sont pas premiers), depuis 3 jusqu'à la borne donnée, en recherchant s'il est divisible par un des nombres premiers déjà trouvés, ce qui signifierait qu'il n'est pas premier. Programme :

```
borne = 100           # pour l'exemple
premiers = [ ]
for candidat in xrange(3, borne + 1, 2):

    estPremier = True
    for premier in premiers:
        if candidat % premier == 0:
            estPremier = False
            break

    if EstPremier:
        premiers.append(candidat)
print premiers
```

4.3 Ensembles

Les ensembles sont des structures de données avec les caractéristiques suivantes :
– il n'y a pas d'accès indexé aux éléments, ni d'ordre entre ces derniers,
– il n'y a pas de répétition des éléments : un élément appartient ou non à un ensemble, mais cela n'a pas de sens de se demander s'il s'y trouve plusieurs fois,
– en contrepartie, le test d'appartenance est optimisé : le nécessaire est fait pour que le test *élément* ∈ *ensemble* ? (en Python : *élément* `in` *ensemble*) puisse être évalué de manière extrêmement efficace.

On construit un ensemble par une expression de la forme

`set(` *séquence* `)`

où *séquence* est une donnée parcourable (liste, tuple, chaîne de caractères, etc.). Exemple :

```
>>> s = set("abracadabra")
>>> s
set(['a', 'r', 'b', 'c', 'd'])
>>>
```

Parmi les principales opérations des ensembles :

élément `in` *ensemble*	*élément* appartient-il à *ensemble* ?
ensemble`.add(`*élément*`)`	ajout de l'*élément* indiqué à l'*ensemble* indiqué
ensemble`.remove(`*élément*`)`	suppression de l'*élément* indiqué de l'*ensemble* indiqué
$ensemble_1$`.issubset(`$ensemble_2$`)`	tous les éléments de $ensemble_1$ appartiennent-ils à $ensemble_2$?
$ensemble_1$`.union(`$ensemble_2$`)`	ensemble des éléments appartenant à $ensemble_1$ ou à $ensemble_2$
$ensemble_1$`.intersection(`$ensemble_2$`)`	éléments de $ensemble_1$ qui sont aussi éléments de $ensemble_2$
$ensemble_1$`.difference(`$ensemble_2$`)`	éléments de $ensemble_1$ qui ne sont pas dans $ensemble_2$

Notez que les opérations *a*`.issubset(`*b*`)`, *a*`.union(`*b*`)`, *a*`.intersection(`*b*`)` et *a*`.difference(`*b*`)` peuvent aussi se noter, respectivement, *a* `<=` *b*, *a* `|` *b*, *a* `&` *b* et *a* `-` *b*.

• **Exemple de script 5**. Déterminer les caractères qui apparaissent exactement une fois dans un texte donné. Algorithme : déterminer l'ensemble des caractères qui apparaissent au moins une fois et l'ensemble de ceux qui apparaissent plus d'une fois ; la différence entre ces deux ensembles est la réponse à la question.

```
texte = """Maitre Corbeau, sur un arbre perche
Tenait en son bec un fromage
Maitre Renard, par l'odeur alleche
lui tint a peu pres ce langage"""

auMoinsUneFois = set()
auMoinsDeuxFois = set()
for caract in texte:
    if caract in auMoinsUneFois:
        auMoinsDeuxFois.add(caract)
    else:
        auMoinsUneFois.add(caract)

print auMoinsUneFois.difference(auMoinsDeuxFois)
```

4.4 Dictionnaires

Un *dictionnaire*, ou *table associative*, est une collection de couples (*clé*, *valeur*) telle que
– il n'y a pas deux couples ayant la même clé,
– la structure est implémentée de manière que la recherche d'une valeur à partir de la clé correspondante soit extrêmement efficace.

Les clés ne doivent pas être des structures modifiables mais, à part cela, elles peuvent être de n'importe quel type ; les valeurs sont quelconques. On construit explicitement un dictionnaire par une expression de la forme

`{` *clé*$_1$ `:` *valeur*$_1$ `,` *clé*$_2$ `:` *valeur*$_2$ `,` ... *clé*$_k$ `:` *valeur*$_k$ `}`

Parmi les principales opérations des dictionnaires :

dict`[`*clé*`] =` *valeur*	ajoute au dictionnaire *dict* une paire `(`*clé*`,` *valeur*`)` ou, si une telle paire existait déjà, modifie sa partie *valeur*,
dict`[`*clé*`]`	renvoie la *valeur* correspondant à la *clé* donnée ; une erreur est déclenchée si une telle *clé* n'existe pas dans le dictionnaire,
dict`.get(`*clé* [`,`*valsinon*]`)`	renvoie la *valeur* correspondant à la *clé* donnée ; si une telle *clé* est absente, renvoie *valsinon* ou (si *valsinon* est omise) `None`,
dict`.has_key(`*clé*`)`	vrai si et seulement si la *clé* indiquée existe dans le dictionnaire *dict*,
`del` *dict*`[`*clé*`]`	supprime du dictionnaire *dict* la paire `(`*clé*`,` *valeur*`)`,
dict`.keys()`	renvoie une copie de la liste des clés du dictionnaire *dict*,
dict`.values()`	renvoie une copie de la liste des valeurs du dictionnaire *dict*,
dict`.items()`	renvoie une copie de la liste des associations constituant le dictionnaire *dict*.

• **Exemple de script 6**. Compter le nombre d'apparitions de chaque caractère d'un texte donné. Algorithme : construire un dictionnaire dont les clés sont les caractères du texte et les valeurs le nombre d'occurrences de chacun. Programme :

```
texte = """Maitre Corbeau, sur un arbre perche
Tenait en son bec un fromage
Maitre Renard, par l'odeur alleche
lui tint a peu pres ce langage"""

dico = { }
for car in texte:
    if dico.has_key(car):
        dico[car] = dico[car] + 1
    else:
        dico[car] = 1

for car in dico.keys():
    print car, ':', dico[car], 'occurrence(s)'
```

4.5 Tableaux

Les *tableaux* les plus basiques [5], comme ceux des langages C ou Java, existent en Python mais ne font pas partie du cœur du langage. Définis dans le module `array` de la bibliothèque standard, pour pourvoir les utiliser dans un programme on doit écrire au début de celui-ci la directive

```
from array import array
```

Il s'agit de tableaux *simples* et *homogènes*. Cela veut dire que leurs éléments doivent avoir un type commun, et que ce doit être un type primitif de la machine sous-jacente (ou plutôt du langage C dans lequel la machine Python est écrite).

Lors de la création d'un tableau on doit indiquer le type de ses éléments ; cela se fait par une lettre : `'i'` pour un tableau d'entiers, `'f'` pour un tableau de flottants, etc. Exemple, création d'un tableau d'entiers (il n'est pas nécessaire à ce niveau de donner la taille – nombre maximum d'éléments – du tableau) :

```
tableau = array('i')
```

5. Quel que soit le langage de programmation utilisé, un *array* (en français on dit *tableau*) est un arrangement d'éléments *contigus* et *de même type*. C'est une structure compacte (l'encombrement du tableau est celui de ses éléments, aucune mémoire additionnelle n'est requise) et très efficace (l'accès à un élément à partir de son indice se fait en un temps constant).

Voici les principales opérations qu'un tel tableau pourra subir ensuite :

tableau`[`*indice*`]`	obtention de la valeur de l'élément ayant l'*indice* indiqué,
tableau`[`*indice*`] =` *valeur*	affectation d'une nouvelle valeur à l'élément ayant l'*indice* indiqué ; cet élément doit exister, sinon une erreur est déclenchée,
tableau`.append(`*valeur*`)`	ajout de la *valeur* indiquée à la fin du *tableau* indiqué ; cette opération augmente le nombre d'éléments du tableau,
tableau`.extend(`*séquence*`)`	allongement du tableau indiqué par les éléments de la séquence indiquée ; cela permet notamment de transformer une liste, un ensemble, etc., en un tableau, ou – à l'aide des fonctions `range` ou `xrange` – de donner à un tableau une taille quelconque,

• **Exemple de script 7**. On reprend le problème de l'exemple de la section 4.4, compter le nombre d'occurrences de chaque caractère d'un texte, qu'on résout ici en utilisant un tableau de 256 compteurs, chacun destiné à compter le nombre d'apparitions d'un caractère (on suppose que le texte est écrit avec les 256 caractères usuels « iso-8859-1 »). Programme :

```
texte = """Maitre Corbeau, sur un arbre perche
Tenait en son bec un fromage
Maitre Renard, par l'odeur alleche
lui tint a peu pres ce langage"""

from array import array

compteurs = array('i')
compteurs.extend( [ 0 ] * 256 )

for c in texte:
    compteurs[ord(c)] = compteurs[ord(c)] + 1

for i in range(256):
    if compteurs[i] != 0:
        print chr(i), ':', compteurs[i], 'occurrence(s)'
```

On remarquera dans le programme précédent comment le tableau est initialisé avec une liste de 256 zéros (concaténation de 256 exemplaires de la liste `[ 0 ]`).

5 Fonctions

5.1 Notions et syntaxe de base

Définition de fonction :

`def` *nomFonction*`(`*argument*$_1$`,` *argument*$_2$`,` ... *argument*$_k$`):`
    instruction$_1$
    instruction$_2$
    ...
    instruction$_p$

La ligne « `def` *nomFonction*`(`*argument*$_1$`,` *argument*$_2$`,` ... *argument*$_k$`)` » est appelée l'*en-tête* de la fonction ; la séquence « *instruction*$_1$, *instruction*$_2$, ... *instruction*$_p$ » le *corps* de cette dernière. La fin du corps, et donc de la définition de la fonction, est indiquée par l'apparition de lignes ayant la même indentation que l'en-tête.

Appel de fonction. L'effet d'une définition de fonction n'est pas d'exécuter les instructions qui en composent le corps, mais uniquement de mémoriser ces instructions en vue d'une (hypothétique) exécution ultérieure, provoquée par une expression, appelée *appel* de la fonction, qui prend la forme

nomFonction`(`*expression*$_1$`,` *expression*$_2$`,` ... *expression*$_k$`)`

Les expressions $expression_1$, $expression_2$, ... $expression_k$ sont appelées les *arguments effectifs* de l'appel. Leurs valeurs sont affectées à $argument_1$, $argument_2$, ... $argument_k$, les *arguments formels* de la fonction, juste avant l'exécution de son corps, comme si cette exécution commençait par une série d'affectations :

$argument_1 = expression_1$
$argument_2 = expression_2$
...
$argument_k = expression_k$

Très souvent, le corps d'une fonction contient une ou plusieurs instructions de la forme

`return` *expression*

dans ce cas, l'appel de la fonction, à l'endroit où il est écrit, représente cette valeur renvoyée ; cet appel prend alors plutôt la forme

resultat `=` *nomFonction*`(`$expression_1$`,` $expression_2$`,` ... $expression_k$`)`

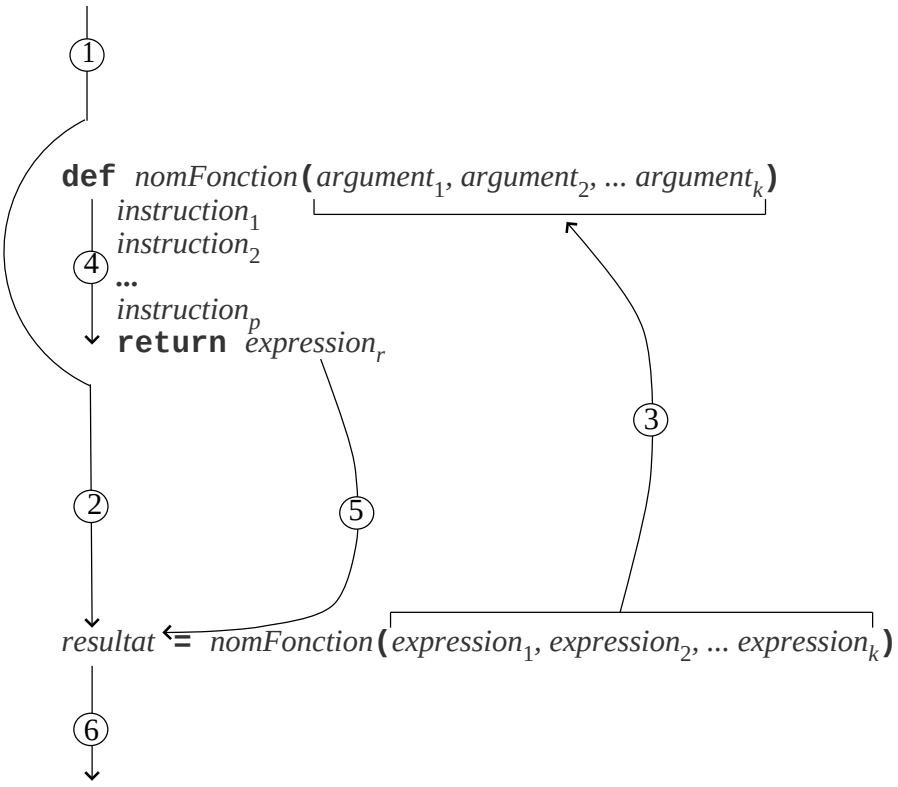

FIGURE 4 – Chronologie de l'*exécution* d'un script comportant une fonction

La figure 4 montre l'ordre chronologique d'*exécution* des instructions d'un programme comportant la définition et l'appel d'une fonction.

• **Exemple de script 8**. Voici une fonction qui calcule la moyenne et l'écart-type des éléments d'une liste de nombres. Programme :

```
from math import sqrt

def moyenneEcartType(listeVal):
    nombre = 0
    sommeValeurs = 0.0
    sommeCarres = 0.0
    for x in listeVal:
        nombre += 1
        sommeValeurs += x
        sommeCarres += x * x
    moyenne = sommeValeurs / nombre
    ecartt = sqrt(sommeCarres / nombre - moyenne * moyenne)
    return (moyenne, ecartt)

# un appel de la fonction, pour la tester:

valeurs = [ 1, 9, 4, 6, 8, 2, 5, 3, 7 ]
resultat = moyenneEcartType(valeurs)
print resultat
```

(L'exécution de ce programme produit l'affichage de `(5.0, 2.5819888974716116)`)

L'instruction *return* ne fait pas qu'indiquer le résultat renvoyé par la fonction ; quel que soit l'endroit où elle est écrite, elle sert aussi à provoquer l'abandon de la fonction et le retour à l'endroit où celle-ci a été appelée :

• **Exemple de script 9**. Recherche de l'indice de la première occurrence d'une valeur dans une liste, écrite sous forme de fonction (reprise de l'exemple donné à la section 3.4). Dans le programme suivant, notez comment la fonction est abandonnée dès la rencontre de la valeur cherchée :

```
def position(valeur, liste):
    for i in range(len(liste)):
        if liste[i] == valeur:
            return i
    return -1

# un appel de la fonction, pour la tester:

uneValeur = 64
uneListe = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

p = position(uneValeur, uneListe)
if p >= 0:
    print "la valeur est a la position", p
else:
    print "la valeur n'est pas presente"
```

(L'exécution de ce script produit l'affichage du texte « `la valeur est a la position 6` ».)

5.2 Variables locales et globales

A défaut d'une déclaration *global* (voir ci-après), toute variable apparaissant dans une fonction comme membre gauche d'une affectation est *locale* à cette fonction. Cela signifie qu'elle a une portée réduite à la fonction et, surtout, qu'elle est créée chaque fois que la fonction est appelée et détruite chaque fois que la fonction se termine. A l'opposé de cela, les variables globales sont les variables créées à l'extérieur de toute fonction ; elles existent depuis le moment de leur création jusqu'à la fin de l'exécution du programme.
Une variable globale peut, sans précaution particulière, être utilisée à l'intérieur d'une fonction si elle n'est pas le membre gauche d'une affectation et si elle n'est pas masquée par une variable locale de même nom.

Pour que dans une fonction une variable globale puisse être le membre gauche d'une affectation, la variable en question doit faire l'objet d'une déclaration

`global` *nomDeVariable*

Par exemple, imaginons une fonction dont on voudrait connaître le nombre de fois qu'elle est appelée pendant le déroulement d'un programme. Voici comment :

```
nbrAppels = 0  # pour compter les appels de uneFonction

def uneFonction():
    global nbrAppels      # sans cela nbrAppels serait une (nouvelle) variable locale
    nbrAppels = nbrAppels + 1
    le reste de la fonction
...
et, vers la fin du script, on ajoutera
print nbrAppels, 'appels de uneFonction'
```

5.3 Plus sur les paramètres formels

Paramètres avec valeur par défaut. Lors de la définition d'une fonction il est possible de spécifier des valeurs par défaut pour un ou plusieurs des arguments formels. Cela permet d'appeler la fonction avec moins d'arguments effectifs que d'arguments formels, les valeurs par défaut remplaçant les arguments manquants.

• **Exemple de script 10**. Le script suivant pose une question fermée (dont la réponse est *oui* ou *non*) et permet de recommencer un certain nombre de fois, tant que la réponse n'est pas dans une liste de bonnes réponses :

```
def demanderAccord(question, essais = 1000, alerte = "oui ou non, s'il vous plait!"):
    while True:
        reponse = raw_input(question).lower().strip()
        if reponse in ('o', 'oui'):
            return True
        if reponse in ('n', 'non'):
            return False
        essais -= 1
        if essais < 0:
            raise IOError, "l'utilisateur semble incapable de répondre"
        print alerte

# divers appels pour tester la fonction
print demanderAccord("On s'en va?", 5, "En français, patate!")
print demanderAccord("On s'en va?", 3)
print demanderAccord("On s'en va?")
```

Paramètres à mot clé. Les paramètres effectifs d'un appel de fonction peuvent être donnés dans un ordre différent de celui des arguments formels : il suffit pour cela d'écrire les arguments déplacés sous la forme de couples *nomArgumentFormel* = *expression*.

Par exemple, voici une fonction (censée afficher un texte dans une fenêtre graphique) avec de nombreux arguments ayant des valeurs par défaut :

```
def afficher(texte, x = 0, y = 0, police = 'Courrier', taille = 12,
        style = 'normal', avantPlan = 0x000000, arrierePlan = 0xFFFFFF):
    corps de la fonction
```

Et voici plusieurs appels légitimes utilisant les arguments à mot-clés :

```
afficher('Salut')
...
afficher('Salut', police = "Times", avantPlan = 0xFF0000)
...
afficher(x = 50, y = 200, texte = 'Salut', style = "italic")
```

ARGUMENTS EN NOMBRE VARIABLE. Si une fonction comporte un argument formel précédé d'une étoile `*` alors lors d'un appel on pourra mettre à cet endroit un *nombre quelconque d'arguments effectifs* ; le paramètre formel sera affecté par le tuple formé par ces arguments.

Par exemple, la fonction (purement démonstrative) suivante :

```
def uneFonction(a, b, *c):
    print 'a:', a, '- b:', b, '- c:', c
```

peut être appelée avec un nombre quelconque de paramètres effectifs, à la condition qu'il y en ait au moins deux. Les deux premiers seront affectés à `a` et `b` respectivement, tandis que `c` sera affecté par le tuple formée de tous les autres arguments. Essais : les trois appels suivants

```
uneFonction(11, 22, 33, 44, 55, 66)
uneFonction(11, 22, 33)
uneFonction(11, 22)
```

affichent respectivement

```
a: 11 - b: 22 - c: (33, 44, 55, 66)
a: 11 - b: 22 - c: (33,)
a: 11 - b: 22 - c: ()
```

5.4 Fonctions récursives

Une fonction est dite récursive lorsque son corps contient un ou plusieurs appels d'elle-même. Par exemple, la factorielle d'un nombre n se définit « itérativement » par $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$. Il est évident qu'une définition récursive de la même quantité est : $n! = \begin{cases} \text{si } n \leq 1 \text{ alors } 1 \\ \text{sinon } n \times (n-1)! \end{cases}$, ce qui donne le code suivant :

• **Exemple de script 11**. Calcul récursif de $n!$. Fichier *factorielle.py* :

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)

# essai:
n = int(raw_input("nombre ? "))
print n, '! =', fact(n)
```

5.5 Forme lambda et *list comprehension*

FORME LAMBDA. Lorsqu'une fonction se réduit à l'évaluation d'une *expression* elle peut être définie de manière anonyme par une construction de la forme :

`lambda` arg_1, arg_2, ... arg_k `:` *expression*

Cela définit une fonction, qui n'a pas de nom, qui a arg_1, arg_2, ... arg_k pour arguments formels et dont le corps se réduit à « `return` *expression* ».

Le lien avec l'autre manière de définir une fonction est facile à comprendre : l'effet de la définition


```
def unNom(arg1, arg2, ... argk):
    return expression
```

est le même que celui de l'affectation

```
unNom = lambda arg1, arg2, ... argk : expression
```

List comprehension. La syntaxe d'une *list comprehension* est la suivante :

```
[ expression for var1, var2, ... vark in séquence ]
```

Cette expression construit la liste formée par les résultats des évaluations de *expression* obtenus en donnant au tuple de variables (var_1, var_2, ... var_k) successivement les valeurs de la *séquence* indiquée. La partie « `for` var_1, var_2, ... var_k `in` *séquence* » peut se répéter. Exemples pour comprendre :

```
>>> t = (1, 2, 3, 4, 5)
>>> [x for x in t]
[1, 2, 3, 4, 5]
>>> [x * x for x in t]
[1, 4, 9, 16, 25]
>>> [ x * y for x in t for y in t]
[1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10, 15, 20, 25]
```

5.6 Chaîne de documentation

La chaîne de documentation d'une fonction est une chaîne de caractères écrite entre l'en-tête et le corps. Sa présence n'est pas une obligation mais est fortement recommandée du point de vue méthodologique, car elle permet d'expliciter avec concision et précision l'algorithme implémenté par la fonction, ses arguments, son résultat, les contraintes que doivent satisfaire les uns et les autres, etc.

Outre d'éventuels lecteurs (humains) de votre programme, la chaîne de documentation s'adresse à l'interpréteur Python, qui est capable de l'exploiter, notamment lors de la construction automatique de la documentation. Pour cette raison, la structure de cette chaîne est imposée ; elle se compose
– d'une première ligne, commençant par une majuscule et finissant par un point,
– suivie par une ligne blanche,
– suivie par un texte de plusieurs lignes dans lequel on a un peu plus de liberté.

• **Exemple de script 12**. Voici une nouvelle rédaction de l'exemple de la section 5.1 :

```
def moyenneEcartType(listeVal):
    """Moyenne et écart-type d'une liste de nombres.

    listeVal doit être une liste de nombres non vide
    renvoie les valeurs de la moyenne et l'écart-type
    sous forme d'un tuple de deux nombres flottants
    """

    nombre = 0
    sommeValeurs = 0.0
    sommeCarres = 0.0
    for x in listeVal:
        nombre += 1
        sommeValeurs += x
        sommeCarres += x * x
    moyenne = sommeValeurs / nombre
    ecartt = sqrt(sommeCarres / nombre - moyenne * moyenne)
    return (moyenne, ecartt)
```

6 Entrées-sorties

6.1 Acquisition de données au clavier

Une seule fonction, simple et pratique, pour les saisies au clavier :

`raw_input(`*invite*`)`

Cette fonction affiche l'*invite* indiquée puis engrange tout ce que l'utilisateur tape au clavier jusqu'à la frappe d'une fin de ligne (la touche *Entrée*). La fonction renvoie les caractères ainsi acquis sous forme d'une chaîne de caractères. Essai :

```
>>> s = raw_input("A toi: ")
A toi: Bonjour. Ca va?
>>> s
'Bonjour. Ca va?'
>>>
```

Cette fonction est souvent employée pour acquérir des données qui ne sont pas des chaînes (des nombres, etc.). Il faut alors la composer avec une *fonction de conversion* qui, en principe, a le même nom que le type visé (`int`, `float`, etc.). Exemple :

```
>>> x = raw_input("donne x: ")
donne x: 123
>>> print x, type(x)
123 <type 'str'>
>>> x = float(raw_input("donne x: "))
donne x: 123

>>> print x, type(x)
123.0 <type 'float'>
>>>
```

6.2 Affichage de données mises en forme

Fonctions d'affichage. Trois manières d'obtenir l'affichage d'une valeur à l'écran :
- l'affichage automatique du résultat que fait l'interpréteur lorsqu'on lui donne comme travail l'évaluation d'une expression ; cet affichage est très utile quand on débute en Python mais cesse de l'être quand on réalise des programmes importants, car ceux-ci se terminent généralement sans renvoyer de valeur ou ne sont même pas destinés à se terminer,
- la fonction `print`, certainement le moyen le plus pratique,
- l'application à l'unité de sortie standard (l'écran) de fonctions de traitement de fichiers plus savantes, comme `write`, expliquées à la section 6.3.

Une différence notable entre l'affichage automatique du résultat d'une évaluation et l'affichage que produit la fonction `print` : le premier affiche des données *réinjectables*, c'est-à-dire possédant la syntaxe qu'elles doivent avoir lorsqu'elles sont écrites dans les scripts, tandis que le second écrit des données *nettoyées*, plus lisibles par un humain. Démonstration :

```
>>> s = "J'ai dit \"Bonjour\"\net je n'ai rien entendu!"
>>> s
'J\'ai dit "Bonjour"\net je n\'ai rien entendu!'
>>> print s
J'ai dit "Bonjour"
et je n'ai rien entendu!
>>>
```

La fonction `print` s'emploie de la manière suivante :

`print` *expression*$_1$ `,` *expression*$_2$ `,` ... *expression*$_k$ [`,`]

Les *expressions* sont évaluées et les valeurs obtenues sont affichées, chacune séparée de la suivante par une espace. La dernière virgule est facultative : si elle est présente, la dernière donnée sera suivie d'une espace (et pas d'une fin de ligne) ; si elle est absente, la dernière donnée sera suivie d'une fin de ligne de sorte que les éventuels affichages ultérieurs se feront sur les lignes en-dessous.

Pour une mise en forme plus savante il faut employer cette fonction sous la forme

`print` *chaîne_de_caractères*

où *chaîne_de_caractères* est le résultat de la mise en forme des données à l'aide de l'opérateur `%`, selon les explications ci-dessous.

Réapparition de l'opérateur `%`. Si s est une chaîne de caractères, l'opération s `%` t a une interprétation tout à fait particulière :
- il est supposé que s est une chaîne exprimant un *format*, lire l'explication ci-après,
- il est nécessaire que t soit un tuple ayant autant d'éléments que le format l'implique,
- dans ces conditions, la valeur de l'expression s `%` t est le résultat de la mise en forme des données qui composent t selon les indications que donne s.

Le format s obéit exactement aux mêmes règles que dans les appels de la fonction `printf(`s`, ...)` du langage C. Une explication détaillée de ces règles sortirait du cadre de ce petit polycopié. Pour débuter, il suffit de savoir que
- `%d` indique un nombre entier, `%`n`d` un nombre entier cadré à droite dans une zone de n caractères,
- `%f` indique un nombre flottant, `%`n`f` un flottant cadré à droite dans une zone de n caractères, `%`n`.`m`f` un flottant cadré à droite dans une zone de n caractères, ayant m chiffres après le point décimal,
- `%c` indique un caractère, `%`n`c` un caractère cadré à gauche dans une zone de n caractères,
- `%s` indique une chaîne de caractères, `%`n`s` une chaîne cadrée à gauche dans une zone de n caractères.

Session pour comprendre :

```
>>> x = 12.345678
>>> 'resultat: %f' % x
'resultat: 12.345678'
>>> 'resultat: %12f' % x
'resultat:    12.345678'
>>> 'resultat: %12.3f' % x
'resultat:       12.346'
>>> '%d/%02d/%4d' % (5, 8, 2008)
'5/08/2008'
>>> 'x[%d] = %.2f' % (i, x[i])
'x[8] = 123.45'
>>> 'chr(%d) = %c' % (65, 65)
'chr(65) = A'
>>>
```

6.3 Fonctions pour les fichiers

Quand on aborde les fichiers on s'intéresse pour la première fois à des données qui existent *à l'extérieur* de notre programme. Du point de vue du programmeur, un fichier ouvert « en lecture » doit être vu comme un tube par lequel arrivent des données extérieures chaque fois que le programme les demande, de même qu'il faut voir un fichier ouvert « en écriture » comme un tube par lequel s'en vont à l'extérieur les données que le programme y dépose.

On notera que cette approche permet de voir comme des fichiers le clavier et l'écran du poste utilisé. Cependant, les fichiers auxquels on pense d'abord sont ceux enregistrés sur des supports non volatils (disques durs, clés USB, cédéroms, etc.). Ils représentent des ensembles de données extérieures au programme, qui existent déjà quand l'exécution de celui-ci démarre et qui ne disparaissent pas quand celle-ci se termine.

Pour les fonctions de la bibliothèque Python tout fichier est considéré comme *séquentiel*, c'est-à-dire constitué par une succession d'informations lues ou écrites les unes à la suite des autres. Une fois qu'un fichier a été ouvert

par un programme, celui-ci maintient constamment une *position courante*, représentée par un « pointeur » (fictif) qui indique constamment le prochain octet qui sera lu ou l'endroit où sera déposé le prochain octet écrit. Chaque opération de lecture ou d'écriture fait avancer ce pointeur.

Voici les principales fonctions pour le traitement des fichiers :

fichier `= open(`*nom_du_fichier*`, `*mode*`)`

Ouverture du fichier. Cette fonction crée et renvoie une valeur qui représente dans le programme l'extrémité d'une sorte de tube dont l'autre extrémité est le fichier en question (entité extérieure au programme, qui vit sa propre vie dans le système environnant).

nom_du_fichier est une chaîne de caractères qui identifie le fichier selon la syntaxe propre du système d'exploitation,

mode est une des chaînes suivantes :

`"r"` : ouverture en *lecture* ; le fichier doit exister, le pointeur est positionné au début du fichier. Sur ce fichier seules les opérations de lecture sont permises.

`"w"` : ouverture en *écriture* ; le fichier existe ou non ; s'il existe il sera écrasé (entièrement vidé). Le pointeur est positionné à la fin du fichier, qui est aussi son début. Seules les opérations d'écriture sont permises.

`"a"` : ouverture en *allongement* ; le fichier existe ou non ; s'il existe, il ne sera pas remis à zéro. Le pointeur est positionné à la fin du fichier. Seules les opérations d'écriture sont permises.

Exemple (avec le nom d'un fichier de UNIX) :

```
fichier = open("/users/henri/aout_2008/resultats.txt", "r")
```

fichier`.read()`

Lit tout le contenu du *fichier* indiqué et le renvoie sous forme de chaîne de caractères.

Attention, cette fonction ne convient qu'aux fichiers de taille raisonnable, dont le contenu tient dans la mémoire de l'ordinateur.

fichier`.readline()`

Lit une ligne du *fichier* indiqué (lors du premier appel la première ligne, au second appel la deuxième, etc.).

Attention. Dans les fichiers de texte, la fin de chaque ligne est indiquée par un caractère spécial, qu'on peut noter `'\n'` dans les programmes. En toute rigueur cette marque ne fait pas partie de la ligne, cependant les fonctions `readline` et *readlines* l'ajoutent à la fin de la ligne lue et le renvoient dans le résultat : ne pas oublier de l'enlever si nécessaire (voyez les exemples de la section suivante).

fichier`.readlines()`

Lit tout le contenu du *fichier* indiqué et le renvoie sous la forme d'une liste de chaînes de caractères (une chaîne par ligne).

fichier`.write(`*chaîne*`)`

Écrit la *chaîne* indiquée dans le *fichier* en question.

fichier`.writelines(`*séquence*`)`

Écrit les chaînes composant la *séquence* (liste, tuple, ensemble, etc.) de chaînes donnée dans le *fichier* indiqué.

fichier`.close()`

Ferme le *fichier* indiqué.

Prendre garde au fait que tant qu'un fichier nouvellement créé (c'est-à-dire un fichier ouvert en écriture) n'a pas été fermé son statut dans le système n'est pas bien défini : le fichier peut ne pas apparaître sur le disque dur, ou ne pas être complet, etc. Pour un fichier en écriture, l'appel final de `close` est donc obligatoire.

6.4 Exemples de traitement de fichiers

• **Exemple de script 13**. Vérifier le succès de l'ouverture du fichier et lire les lignes d'un fichier de texte.

```
def lireLignes(nomFichier):
    try:
        fic = open(nomFichier, 'r')
    except IOError:
        print nomFichier + " : ce fichier n'existe pas"
        return None

    texte = fic.readlines()
    fic.close();
    return [ t[:-1] for t in texte ]

# un essai de cette fonction:

texte = lireLignes('MonTexte.txt')
if texte != None:
    for ligne in texte:
        print ligne
```

Un exposé sur les exceptions du langage Python sortirait du cadre de ce petit polycopié ; on peut cependant comprendre ce qui est fait dans cet exemple : l'instruction « `fic = open(nomFichier, "r")` » est enfermée dans un bloc *try...except* destiné à attraper une éventuelle erreur (on dit plutôt *exception*) de type *IOError* (erreur d'entrée-sortie). Si une telle erreur est effectivement déclenchée, un message est affiché et la fonction renvoie `None`. S'il n'y a pas d'erreur le bloc *try...except* devient sans effet.

En cas de succès de l'ouverture du fichier, la fonction lit toutes ses lignes par un unique appel de `readlines` puis ferme le fichier, devenu inutile.

L'expression « `return [ t[:-1] for t in texte ]` » finale mérite une explication. Il faut se souvenir que `readlines` laisse à la fin de chaque ligne une marque `'\n'` qu'il faut enlever si on veut faire les choses proprement. D'où notre expression : si `t` est une chaîne, `t[:-1]` produit la chaîne obtenue en enlevant le dernier caractère et `[ t[:-1] for t in texte ]` la séquence obtenue en faisant cela sur chaque ligne de `texte`.

• **Exemple de script 14**. Lire des lignes avec des conditions. Lorsque le nombre de lignes d'un fichier risque d'être très élevé, ou bien lorsque toutes les lignes ne sont pas acceptables, on ne peut pas utiliser la fonction `readlines`.

La fonction suivante lit dans un fichier les `nombreMax` premières lignes qui ne commencent pas par le caractère `#`, ou moins si le fichier n'a pas autant de telles lignes. On suppose ici que l'éventuel échec de l'ouverture du fichier n'a pas à être traitée spécifiquement (on le laisse afficher des messages en anglais et tuer le programme).

```
def lireLignes(nomFichier, nombreMax):
    fic = open(nomFichier, 'r')

    texte = []
    uneLigne = fic.readline()
    while len(texte) < nombreMax and uneLigne != '' :
        if uneLigne[0] != '#' :
            texte.append( uneLigne[:-1] )
        uneLigne = fic.readline()

    fic.close();
    return texte

# un essai de cette fonction:

texte = lireLignes('MonTexte.txt', 10)
for ligne in texte:
    print ligne
```

remarquer comme la fin du fichier est détectée : aussi longtemps que le fichier n'est pas fini, le résultat renvoyé par `fic.readline()` n'est pas la chaîne vide, puisqu'il y a au moins le caractère `\n` qui marque la fin de la ligne. Donc la production d'une ligne vide (pour laquelle la condition `uneLigne != ''` deviendra fausse) ne signifie que l'échec de la lecture, le fichier étant fini.

• **Exemple de script 15**. Lire un fichier de nombres. L'exercice n'a pas une grande difficulté mais nous en donnons ici une solution car il s'agit d'un problème extrêmement fréquent dans les applications.

Nous supposons que les lignes du fichier n'ont pas une structure fixe mais qu'elles sont « propres » : il n'y a que des nombres et des caractères blancs (espaces, tabulations et fins de ligne) qui les séparent, comme ceci :

```
2.5 3 5.25 8
   -0.5

   9   8.2   7.95   4.3   4.25   4.1
```
etc.

Script :

```
def lectureNombres(nomFichier):
    resultat = []
    fichier = open(nomFichier, "r")
    while True:
        ligne = fichier.readline()
        if ligne == '':
            break
        for mot in ligne.split():
            if mot.find('.') >= 0:
                resultat.append(float(mot))
            else:
                resultat.append(int(mot))
    fichier.close()
    return resultat

s = lectureNombres("nombres.txt")
print [ (t, type(t)) for t in s ]
```

• **Exemple de script 16**. Produire un fichier de nombres. Avec ce que nous savons déjà, aucune difficulté pour l'enregistrement dans un fichier d'une collection de nombres, même lorsqu'on nous impose une mise en forme compliquée.

La fonction suivante reçoit un nom de fichier et trois séquences X, Y, Z (supposées de même longueur) de nombres flottants et produit un fichier de texte où chaque ligne porte un entier i et un triplet $(X[i], Y[i], Z[i])$ présentés de la manière suivante :

```
0001 (     5.148,    12.000,    -8.100 )
0002 (    21.739,     4.640,     0.000 )
```
etc.

Script :


```
def ecrireResultats(nomFichier, X, Y, Z):
    fichier = open(nomFichier, "w")
    n = min(len(X), len(Y), len(Z))
    for i in range(0, n):
        s = "%04d (%10.3f,%10.3f,%10.3f)\n" % (i, X[i], Y[i], Z[i])
        fichier.write(s)
    fichier.close()
```

calcul des séquences s_1, s_2, s_3

```
ecrireResultats("resultats.txt", S1, S2, S3)
print "C'est fini"
```

7 Annexes

7.1 Opérations sur les séquences

Voici des tableaux récapitulatifs des opérations sur les séquences. Il y a six types de séquences (certains ignorés dans le cadre de ce cours) : les *chaînes de caractères*, les *chaînes de caractères Unicode*, les *listes*, les *tuples*, les *buffers* et les objets `xrange`. Tous ces types représentent des séquences immuables sauf les listes qui, elles, supportent les modifications.

Pour des précisions supplémentaires, voyez la section 3.6 du manuel *Python Library Reference* (cf. `http://docs.python.org/lib/typesseq.html`)

Opérations sur toutes les séquences

x `in` s	`True` si au moins un élément de s est égal à x, `False` autrement
x `not in` s	`False` si au moins un élément de s est égal à x, `True` autrement
s `+` t	concaténation (mise bout-à-bout) des séquences s et t
s `*` n ou n `*` s	concaténation de n copies « superficielles » de s
s`[`i`]`	$i^{\text{ème}}$ élément de s (le premier élément a l'indice 0)
s`[`i`:`j`]`	tranche de s depuis l'indice i (inclus) jusqu'à l'indice j (exclu), c'est-à-dire la séquence formée des éléments s_i, s_{i+1} ... s_{j-1}
s`[`i`:`j`:`k`]`	tranche de s depuis l'indice i jusqu'à l'indice j, par sauts de k
`len(`s`)`	nombre d'éléments de s
`min(`s`)`	plus petit élément de s
`max(`s`)`	plus grand élément de s

Opérations additionnelles sur les listes. En tant que séquences *modifiables*, les listes supportent en plus les opérations suivantes :

s`[`i`] =` x	remplacement de l'élément d'indice i de s par x
s`[`i`:`j`] =` t	remplacement des éléments de la tranche de s depuis l'indice i jusqu'à l'indice j par ceux de la séquence t
`del` s`[`i`:`j`]`	le même effet que s`[`i`:`j`] = []`
s`[`i`:`j`:`k`] =` t	remplacement des éléments de la tranche s`[`i`:`j`:`k`]` par ceux de la séquence t
s`.append(`x`)`	même chose que s`[len(`s`):len(`s`)] = [`x`]`
s`.extend(`t`)`	même chose que s`[len(`s`):len(`s`)] =` t
s`.count(`x`)`	renvoie le nombre de i pour lesquels on a s`[`i`] ==` x

s`.index(`*x*`[, `*i*`[, `*j*`]])`	plus petit k tel que $i \leq k < j$ et *s*`[k] ==` *x*
s`.insert(`*i*`, `*x*`)`	même chose que *s*`[`*i*`:`*i*`] = [`*x*`]`
s`.pop([`*i*`])`	même chose que « *x* `=` *s*`[`*i*`]; del` *s*`[`*i*`]; return` *x* » ; si *i* est absent la valeur -1 est prise par défaut, c'est-à-dire que c'est le dernier élément qui est enlevé de la séquence et renvoyé
s`.remove(`*x*`)`	même chose que `del` *s*`[`*s*`.index(`*x*`)]`
s`.reverse()`	renverse « sur place » les éléments de *s*
s`.sort([`*cmp*`[, `*key*`[, `*décroissant*`]])`	trie, sur place, les éléments de *s*

7.2 Les scripts donnés en exemple

