Tutoriel

MCours.com

python

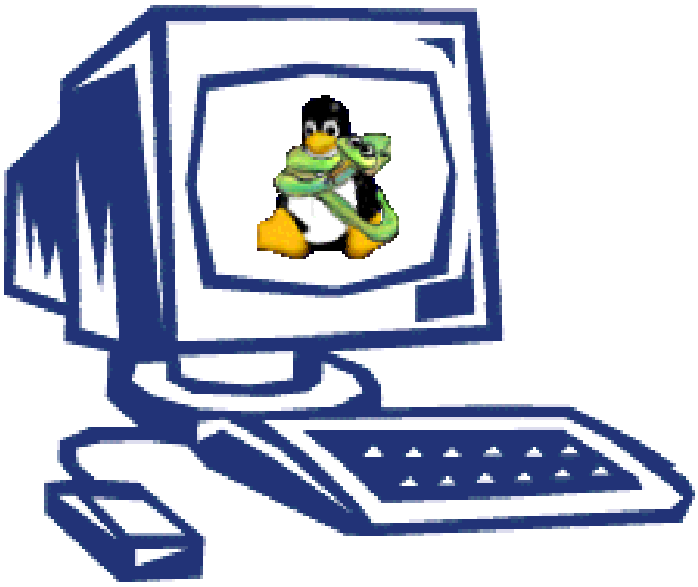


Table des matières :

- Introduction Chapitre 1 page 3
- Les types intégrés Chapitre 2 page 9
- Les variables Chapitre 3 page 36
- Les instructions Chapitre 4 page 38
- Les fonctions Chapitre 5 page 48
- Les modules Chapitre 6 page 58
- Les classes Chapitre 7 page 65
- Les exceptions Chapitre 8 page 75
- Les instructions __builtin__ Chapitre 9 page 85
- En conclusion Chapitre 10 page 91
- Index page 96

1 Introduction.

Table des matières :

1 Introduction. 1
1.1 A propos de ce Tutoriel. 4
1.1.1.1 Remerciements. 4
1.2 Le langage Python. 5
1.2.1 Portable ? 5
1.2.2 Gratuit ? 5
1.2.3 Extensible ? 5
1.2.4 Les Caractéristiques de Python 5
1.3 Comparaison des langages ! 6
1.4 Exécuter des programmes Python 7
1.4.1 Structure d'un programme en Python. 8

1.1 A propos de ce Tutoriel.

J'ai décidé de faire mon travail de diplôme sur le langage Python car l'étude d'un langage de haut niveau et qui introduit la programmation orientée objet m'a permis de compléter une base en programmation faite sur des langages de plus bas niveau (Assembleur et langage C).

Ce tutoriel est destiné à toutes les personnes qui ont déjà une connaissance suffisante en programmation fonctionnelle pour comprendre les mécanismes simples de sélection et d'itération.
Il s'adresse aussi aux personnes qui veulent apprendre un langage de programmation de haut niveau, qui permet d'introduire relativement simplement et sans obligation la programmation orientée objet, et qui par la richesse de ses librairies donne la possibilité de développer rapidement des utilitaires.

Ce tutoriel à été conçu pour donner un avant goût du langage de programmation Python. Il traite les éléments tels que les types intrinsèques, les instructions et les expressions du langage Python avec leurs utilisations de base. Il introduit aussi la base de la programmation orientée objet et le traitement des exceptions. Chaque partie traitant d'un objet, expression ou instruction est suivi d'un exemple simple qui montre les propriétés de l'objet en question.

Les exemples en code Python sont écrits en italiques, et lorsque un exemple à été créé avec l'invité interactif, il est précédé de >>> où bien … !

1.1.1.1 Remerciements.

Mes remerciements pour une aide sans faille à Madame Catherine Rochat et à Messieurs Jacques Finger, Georges Zünd, Fabien Verdan, Cédric Schöpfer, Julien Pernet….!!

1.2 Le langage Python.

Python est un langage de programmation à typage dynamique qui à été développé en 1989 par Guido Van Rossum et de nombreux bénévoles.
Les points forts de Python, sont qu'il est portable, dynamique, extensible et gratuit (General Public License), et qu'il permet une approche modulaire orientée objet, sans pour autant que cela soit obligatoire.

1.2.1 Portable ?

Python est non seulement portable entre les différentes variantes d'Unix, mais aussi sur les OS propriétaires comme MacOS, BeOS, MS-DOS et les différentes variantes de Windows.

1.2.2 Gratuit ?

L'interpréteur Python est gratuit pour tous les OS. En effet Python est placé sous GPL.

1.2.3 Extensible ?

Python a déjà une multitude de librairies et de modules par défaut. Certains sont multi-plateformes et d'autres pas. De plus, il est toujours possible d'en développer pour ses propres besoins.

1.2.4 Les Caractéristiques de Python.

La syntaxe de Python est très simple et, combinée avec des types de donnée évolués (listes, dico,...), conduit à des programmes à la fois compacts et lisibles.

Il gère par lui-même ses ressources (mémoires, descripteurs de fichier,...).

Il intègre comme Java un système d'exceptions, qui permet de simplifier la gestion des erreurs.

Il est orienté objet (sans que cela soit obligatoire). Il supporte l'héritage multiple et la surcharge des opérateurs.

Il est extensible. On peut facilement l'interfacer avec des librairies C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.

La librairie standard de Python, et les paquetages inclus, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standard (fichiers, pipes, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistances et bases de données, interfaces graphiques (Tcl/Tk).

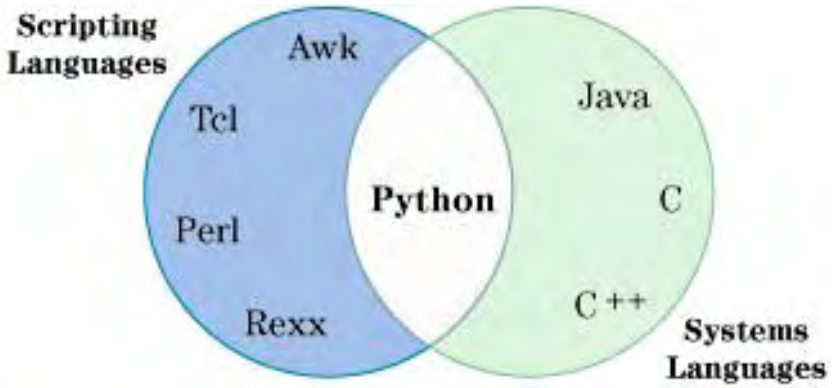
Python est un langage qui continue à évoluer, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre.

1.3 Comparaison des langages !

Python fait partie des langages de script tel que Perl, Tcl, Rexx, alors que Java, C++ et C sont des langages qui nécessitent une compilation.

Les langages de script sont plus rapides au développement que les autres. Les programmes comportent moins de lignes (environ 50 % de moins), par contre leur vitesse d'exécution est plus lente.
La place mémoire prise par les langages de script et Java lors de l'exécution d'un programme est plus grande qu'en C / C++.

Python se situe dans les langages de script rapide, qui utilisent peu de place mémoire (environ la même chose que le C).
La durée de travail pour écrire un programme en Python se situe dans les plus bas (si ce n'est pas le plus bas), de plus la fiabilité du langage Python est proche du 100 % !!



1.4 Exécuter des programmes Python.

Il y a plusieurs façons d'exécuter un programme Python :

- Lancer l'invité interactif. Il est dès lors possible de taper le code au fur et à mesure. Une ligne de l'invité interactif commence toujours par les caractères : >>> où bien … selon l'indentation.

- Créer un fichier comprenant du langage Python puis l'invité interactif de lancer la commande "import nom_du_fichier"(sans le .py) ce qui exécutera le code compris dans le fichier.

- De plus sur certain OS (Unix et Linux) il est aussi possible de créer un script avec sur la première ligne du fichier la commande "#! /usr/bin/python" (par exemple), cette méthode permet de lancer le programme python (il faut que le fichier soit exécutable).

- Python peut aussi se lancer depuis d'autres applications, mais je ne traite pas ce cas dans ce tutoriel.

- Sur la plupart des OS dits "user friendly" les programmes python se lancent en double-cliquant sur le fichier. Dans Windows les extensions .py sont automatiquement ouvertes avec Python, sur MacOS il faut d'abord faire un import et ensuite un .pyc est créé, ce programme en bytecode est alors automatiquement exécutable en cliquant dessus.

Lors de l'installation, Python règle les répertoires où il va chercher les modules et librairies. Il est cependant possible d'en rajouter, mais comme chaque OS a ses spécialité, il est préférable de lire la documentation propre à la version de Python. Nous verrons plus loin une fonction permettant à un programme de rechercher des modules dans de nouveaux répertoires.

1.4.1 Structure d'un programme en Python.

La structure et la syntaxe en Python ne sont pas très compliquées, cependant il est nécessaire de toujours suivre les règles de base !

Python ne nécessite pas de déclaration de type ; une variable est directement affectable par l'opérateur "=".

Un bloc de code en Python, par exemple une fonction, ne s'ouvre et ne se ferme pas avec un caractère (comme {} en C). En Python seul l'indentation de l'instruction indique à quel niveau elle se trouve. Il est très important de toujours garder la même indentation pour un bloc (en général 1 TAB).

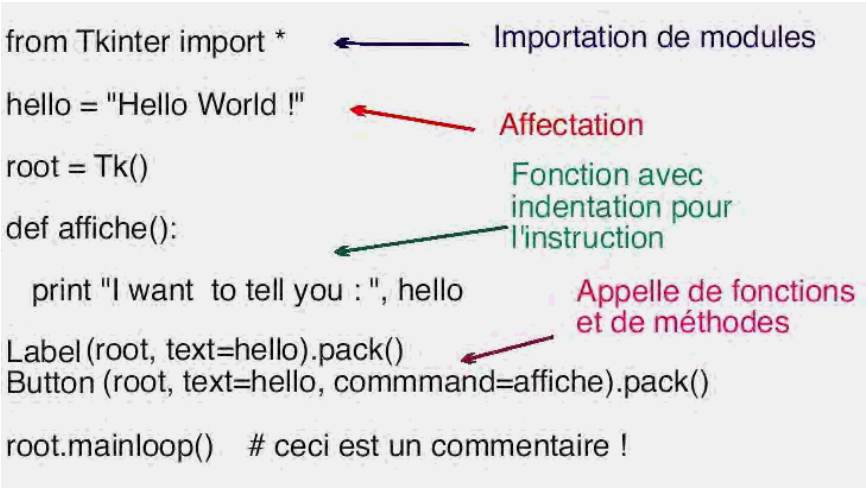
En général les commentaires sont marqués par le caractère #, depuis ce caractère Python prend le reste de la ligne comme un commentaire. Il est aussi possible dans un fichier de marquer un commentaire avec des guillemets,(ceci permet un commentaire sur plusieurs lignes """...."""").

Comme Python est un langage interprété, la lecture d'un fichier se fait de haut en bas. Il faut que les déclarations de fonctions, de variables et de classes se fassent avant leur première utilisation.

Il est possible de mettre plusieurs instructions sur la même ligne, si elles sont séparées par un point-virgule.

Un bloc doit forcément contenir une instruction, un « else »: sans rien générera une erreur.

Les instructions peuvent s'étendre sur plusieurs lignes si elles se trouvent dans une paire syntaxique, par exemple une paire de crochets. Une instruction peut aussi s'étendre sur plusieurs lignes si elle finit par un antislash.



2 Les types intégrés.

Table des matières :

2 Les types intégrés....9
2.1 Les types numériques :....10
2.1.1 Les entiers....10
2.1.2 Les réels....10
2.1.3 Les complexes....10
2.1.4 Les opérateurs principaux pour les types numériques sont :....11
2.1.5 Les tests de vérité....12
2.1.5.1 or et and....12
2.1.6 Exercice numérique No 1....13
2.1.7 Exercice numérique No 2....14
2.2 Les chaînes....15
2.2.1 Indiçage et extraction....15
2.2.2 Formatage des chaînes de caractères....16
2.2.3 L'opérateur surchargé %....16
2.2.4 Variation sur les chaînes....16
2.2.5 Exercice sur les chaînes No 1....17
2.2.6 Exercice sur les chaînes No 2....18
2.2.7 Exercice sur les chaînes No 3....19
2.3 Les listes....20
2.3.1 Exercice sur les listes No 1....21
2.3.2 Exercice sur les listes No 2....23
2.4 Les Dictionnaires....25
2.4.1 Exercice sur les dictionnaires No 1....26
2.5 Les tuples....28
2.5.1 Pourquoi les tuples....28
2.5.2 Exercice sur les tuples No 1....29
2.5.3 Exercice sur les tuples No 2....30
2.6 Les fichiers....32
2.6.1 Exercice sur les fichiers No 1....33
2.6.2 Exercice sur les fichiers No 2....35

2.1 Les types numériques :

Python permet l'utilisation de plusieurs sortes de constantes numériques. Ce sont les entiers, les entiers longs, les nombres à virgules flottantes et les complexes.

Constantes	Interprétation
314 / -2 / 0	Entiers normaux
314314314L	Entiers longs (taille illimitée)
1.23 / 3.14e-10 / 4E210	Virgules flottantes
0177 / 0x9ff	Constantes Octales et hexadécimales
3+4j / 3.0-4.0J	Constantes complexes

2.1.1 Les entiers

Les entiers peuvent être courts (32 bits) ou longs par défaut les types entiers sont courts et en base dix. Les types entiers longs sont de taille illimitée. Pour déclarer un entier long, il suffit de placer un 'L' après sa valeur (999999999L).
Il est aussi possible de créer des entiers avec des valeurs octales et hexadécimales. Pour une valeur octale il suffit de faire précéder la valeur par un '0' (077) et hexadécimale par '0x' (0x3f).

2.1.2 Les réels

Les réels se déclarent sous la forme "x = 1.0" et sont en base dix; pour ce qui est de leur taille, celle-ci est de type double par rapport au C.
Les réels permettent aussi des exposants sous la forme de "4E2". L'indice de l'exposant peux être positif ou négatif et le marqueur 'E' majuscule ou minuscule ! Les valeurs même entières avec des exposants, sont automatiquement transformées en réels.

2.1.3 Les complexes

Les constantes complexes en Python sont écrites "réelle+imaginaire", en terminant avec un 'J' ou 'j' (4+4J). De manière interne, elles sont implémentées par une paire de nombres à virgule flottante, mais les opérations numériques sur les nombres complexes utilisent des fonctions dédiées aux complexes.

2.1.4 Les opérateurs principaux pour les types numériques sont :

Opérateur	Déscription
x or y	ou logique
x and y	et logique
not x	négation logique
<, <=, >, >=, ==, <>, !=	opérateurs de comparaison
is, is not	Test d'identité
in, not in	Appartenance à une séquence
x | y	ou bits-à-bits
x ^ y	ou exclusif bits-à-bits
x & y	et bits-à-bits
x << y, x >> y	Décalage de x par y bits
x + y, x - y	addition ou concaténation / soustarction
x * y	multiplication ou répetition
x / y, x % y	division / reste de la div. (modulo)
-x	négation unaire

La priorité des opérateurs est similaire aux mathématiques donc avec X*Y+Z, X*Y est prioritaire par rapport à +Z. Les parenthèses permettent d'avoir des sous-expressions, dans X*(Y+Z), Y+Z est prioritaire par rapport à X*.

Dans le cas d'un mélange de types numériques lors d'une opération, on peut se demander quel est le type du résultat. La réponse est que Python évalue (comme d'autres langages) la complexité des opérandes et transforme toutes les autres en ce même type. La réponse sera du type le plus complexe de toutes les opérandes. Les entiers courts sont plus simples que les entiers longs, eux-mêmes plus simples que les nombres à virgule flottante qui sont eux plus simples que les complexes !

2.1.5 Les tests de vérité.

En Python « vrai » signifie n'importe quel nombre différent de zéro ou d'objets non vides.
« Faux » signifie « non Vrai », donc zéro, un objet vide , et None.
Les tests d'égalité en Python renvoient 1 ou 0 (Vrai, Faux), et les opérateurs booléens « and » et « or » retournent un objet opérande vrai ou faux.

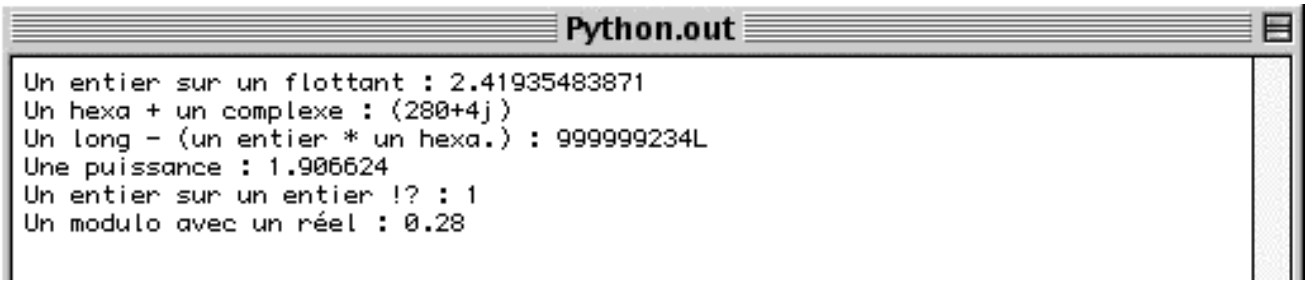
2.1.5.1 or et and.

"or" et "and" retourne toujours des objets, exemple :

```
>>> 2 or 3, 3 or 2 # Si les deux son « vrai » retourne celui de gauche.
(2, 3)
>>> [] or 3, 3 or [] # Si seulement un est « vrai » retourne celui-ci.
(3, 3)
>>> [] or 3
3
>>> [] or {} # Si les deux sont « faux » retourne celui de droite.
{}

>>> 2 and 3, 3 and 2  # Si le résultat est vrai retourne celui de droite.
(3, 2)
>>> [] and 3, 3 and [] # Si seulement un est « vrai » retourne le « faux ».
([], [])
>>> [] and 3
[]
>>> [] and {}# Si les deux sont faux retourne celui de gauche.
[]
```

2.1.6 Exercice numérique No 1.

```
# fichier : exo_num_1.py by J.Tschanz
# fonction : quelques opérations sur les nombres.

nbr_1 = 3 # entier normal
nbr_2 = 999999999L # entier long
nbr_3 = 1.24 # virgule flottante
nbr_4 = 0xff # nombre hexa.
nbr_5 =  25 + 4.0j # complexe
nbr_6 = 4

# simple calculs !!

print "Un entier sur un flottant : %s " % (nbr_1/nbr_3)
print "Un hexa + un complexe : %s " % (nbr_4+nbr_5)
print "Un long - (un entier * un hexa.) : %s " % (nbr_2 - nbr_1*nbr_4)
print "Une puissance : %s " % (nbr_3**nbr_1) # puissance
print "Un entier sur un entier !? : %s" % (nbr_6/nbr_1) # attention entier divisé par un entier
print "Un modulo avec un réel : %s " % (nbr_6%nbr_3) # modulo avec un réel


# fin du fichier exo_num_1.py
```



2.1.7 Exercice numérique No 2.

```
# fichier : exo_num_2.py by J.Tschanz
# fonction : quelques opérations booléennes.

x = y = 1

# simple calculs !!

print "Décalage à gauche : %s " % (y<<2) # décalage à gauche : 0100 (4)
print "OU bits-à-bits : %s " % (x | 2)          #  0011 (3)
print "ET bits-à-bits : %s " % (x & 3) # 0001 (1)
print "XOR bits-à-bits : %s " % (x^y)          # 0000 (0)

# fin du fichier exo_num_2.py
```

2.2 Les chaînes

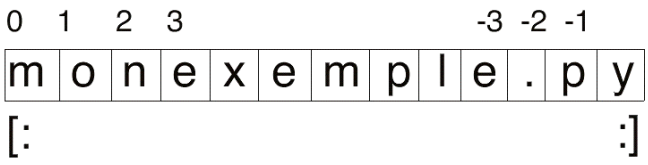
La chaîne est en fait une chaîne de caractères qui est utilisée pour stocker et représenter de l'information textuelle. D'un point de vue fonctionnel, les chaînes peuvent représenter tout ce qui peut être encodé comme du texte.
Les caractères en Python n'existent pas comme dans certains langages (C). Un caractère en Python est une chaîne d'un caractère. Les chaînes de caractères sont des séquences non modifiables : elle répondent aux opérations habituelles mais elle ne peuvent pas être modifiées sur place.

Opération	Intérprétation
s1=''	chaîne vide
s2="l'œuf"	double guillemets
bloc="""..."""	bloc à triple guillemet
s1+s2, s2*3	concaténation, répétition
s2[i], s2[i:j], len(s2)	indice, extraction, longueur
"Hello %s" % 'World'	formatage de chaîne
for x in s2, 'o' in s2	itération, appartenance

Les chaînes peuvent êtres déclarées avec un simple guillemet ou un double. Les deux sont possibles, mais le double guillemet permet d'avoir une apostrophe sans avoir recours au backslash. Le triple double guillemet permet d'entrer une chaîne de caractères sur plusieurs lignes, y compris les caractères de retour de ligne.

2.2.1 Indiçage et extraction.

L'indiçage et l'extraction sont importants pour la compréhension des types chaînes de caractères, listes, tuples, les deux s'effectuent avec des valeurs comprises entre crochets ([]).
L'indiçage est le fait de sélectionner un élément par rapport à sa position dans l'objet. L'indiçage peut s'effectuer en comptant depuis la gauche ou la droite, un indice positif indique que l'on commence à compter par la gauche (x[3]) et que le premier objet a comme indice 0, un indice négatif indique que l'on compte depuis la droite (x[-3]).
L'extraction est le fait de sélectionner une partie plus ou moins grande d'un objet, une tranche. Pour extraire une partie il faut insérer entre les crochets l'indice du premier élément suivi de deux points et de la valeur de l'élément suivant (x[2:5]). Attention la valeur placée sous le deuxième indice ne sera pas extraite !
Une valeur laissée vierge prend la valeur de 0 à gauche et du nombre du dernier élément de l'objet pour la droite (x[:5] => x[0:5] et x[3:] => x[3:fin]). Comme pour l'indiçage les valeurs négatives permettent de compter depuis la droite. Une extraction impossible donnera un objet vide !



2.2.2 Formatage des chaînes de caractères.

Il est impossible de changer sur place une chaîne, par indiçage par exemple. Pour modifier les informations textuelles d'une chaîne nous avons besoin d'en construire une nouvelle.

>>> s = 'dalut'
>>> s[0] = 's'
>>> erreur !!!
mais
>>> s = 's'+s[1:]
>>> s
>>> 'salut'

2.2.3 L'opérateur surchargé %.

L'opérateur % sur les nombres effectue un modulo (reste de division). Il fonctionne avec un format de chaîne sur la gauche (%s).
Avec l'opérateur, il est possible d'insérer dans une phrase une ou des chaînes quelles qu'elles soient.

>>>str = 'mon'
>>> print "c'est %s bateau" % str
>>> c'est mon bateau
>>> print "c'est %s bateau no %d" %(str, 27)
>>> c'est mon bateau no 27

%s	chaîne (ou autre objet)	%X	entier héxa. Majuscules
%c	caractère	%e	format de réels no 1
%d	entier décimal	%E	format de réels no 2
%u	entier non-signé	%f	format de réels no 3
%o	entier octal	%g	format de réels no 4
%x	entier hexadécimal	%%	% littéral no 5

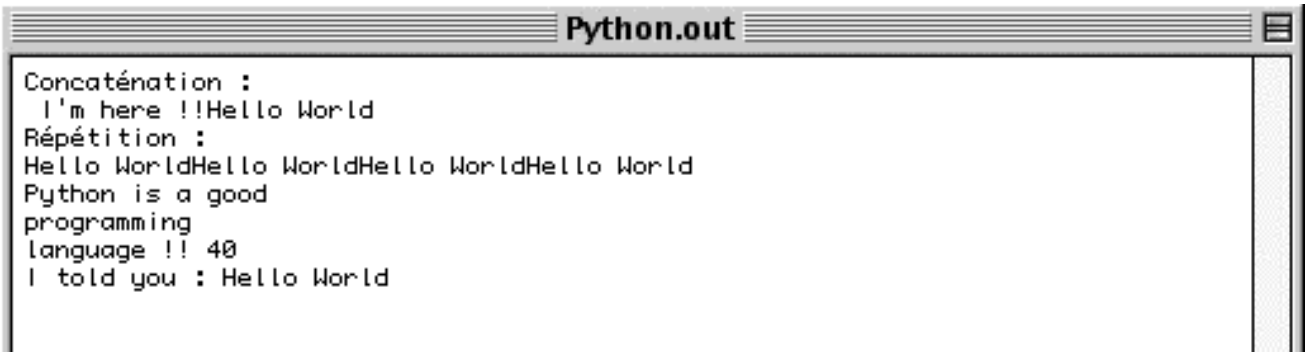
2.2.4 Variation sur les chaînes.

L'insertion du caractère antislash peut servir notamment à insérer des guillemets ou des retours de ligne, mais une multitude d'autres paramètres peuvent être insérés après l'antislash.

\[entrée]	ingnoré	\n	fin de ligne
\\	antislash (garde le 2eme)	\v	tabulation verticale
\'	guillemet simple	\t	tabulation horizontale
\"	guillemet double	\r	retour chariot
\a	sonnerie	\f	formfeed (saut de page)
\b	espace arrière	\0XX	valeur octale XX
\e	échappement	\xXX	valeur Hexa XX
\000	nulle (termine pas la chaîne)	\autre	autre caractère (retenu)

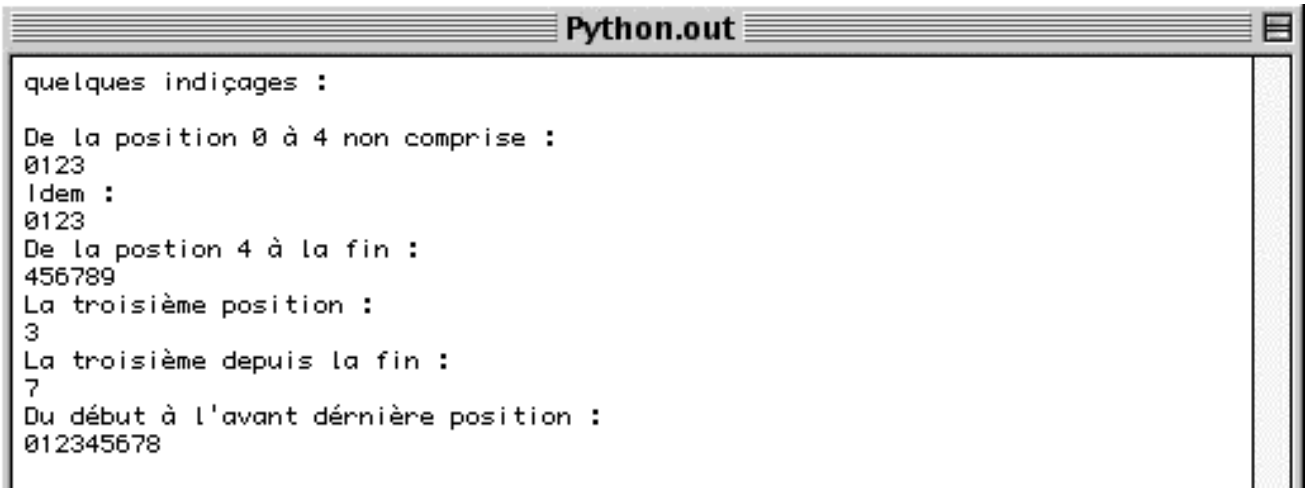
2.2.5 Exercice sur les chaînes No 1.

```
# fichier : exo_chaine_1.py by J.Tschanz
# fonction : quelques opérations sur les chaînes.

# déclarations
s1 = 'Hello World'
s2 = " I'm here !!"
s3 = """Python is a good
programming
language !!""" # sur plusieurs lignes !!

# opérations

print "Concaténation : "
print s2+s1

print "Répétition : "
print s1*4

"Longueur d'une chaîne : "
print s3, len(s3)

"Insertion : "
print "I told you : %s" % s1

# fin du fichier exo_chaine_1.py
```



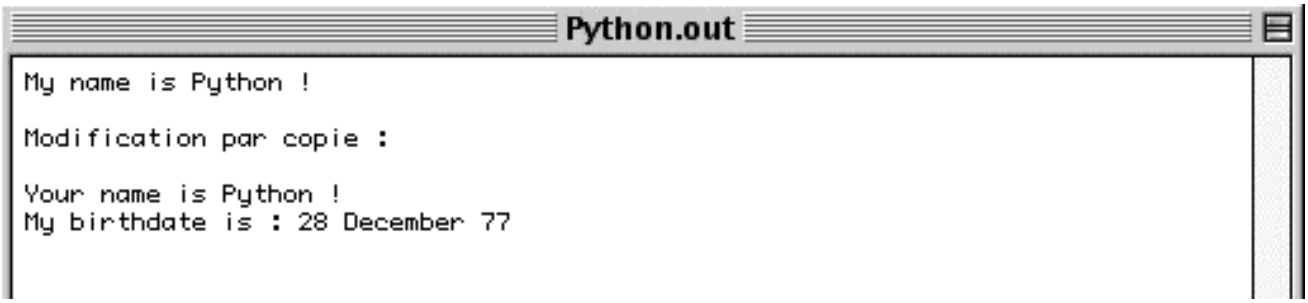
2.2.6 Exercice sur les chaînes No 2.

```
# fichier : exo_chaine_2.py by J.Tschanz
# fonction : quelques indiçages sur les chaînes.

# déclarations
s1 = "0123456789"

# indiçages
print "quelques indiçages : "
print ""
print "De la position 0 à 4 non comprise : "
print s1[0:4]

print "Idem : "
print s1[:4]

print "De la positon 4 à la fin : "
print s1[4:] #  (4eme comprise)

print "La troisième position : "
print s1[3]

print "La troisième depuis la fin : "
print s1[-3]

print Du début à l'avant dernière position : "
print s1[:-1]

# fin du fichier exo_chaine_2.py
```



2.2.7 Exercice sur les chaînes No 3.

```
# fichier : exo_chaine_3.py by J.Tschanz
# fonction : quelques changements de formatage.

# déclarations
s1 = "My name is Python !"

# modification par "copie"
print s1
print ""
print "Modification par copie : "
print ""
s1 = 'Your' + s1[2:]
print s1

s2 = 'December'
print "My birthdate is : %s %s %s" % (28,s2,77)

# fin du fichier exo_chaine_3.py
```



2.3 Les listes.

La liste est l'objet de collection ordonné le plus souple. Il permet de contenir toutes sortes d'objets : nombres, chaînes, et même d'autres listes. La déclaration d'une liste se fait par la mise entre crochets des objets de celle-ci.

>>> x = [0, 1, 'hello']

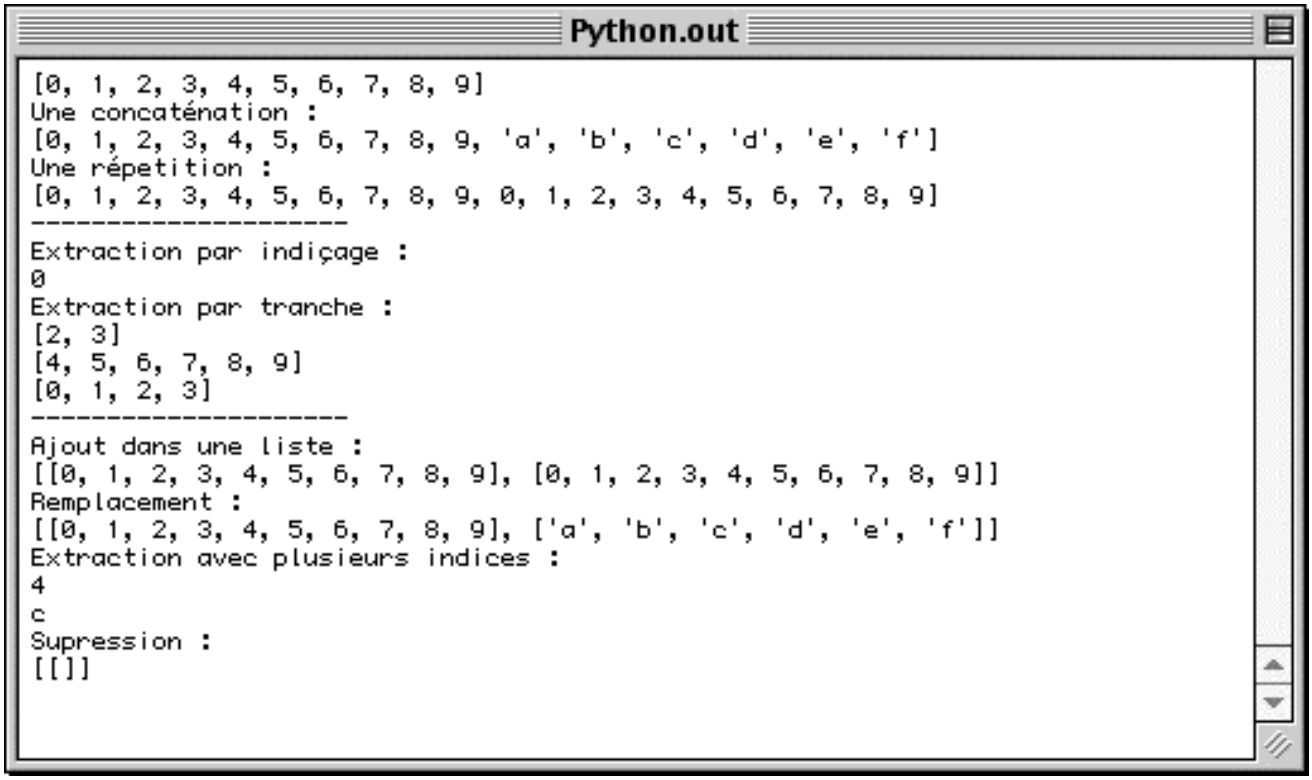
Du point de vue de l'utilisation, les listes permettent de collecter de nouveaux objets afin de les traiter tous ensembles.
Comme les chaînes, les objets d'une liste sont sélectionnables par indiçage. Les listes sont ordonnées et autorisent donc l'extraction de tranches et la concaténation.
Les listes permettent une modification de leur contenu : remplacement, destruction,... d'objets sans pour autant devoir en faire une copie (comme les chaînes). On dit quelles peuvent êtres changées sur place.

Opération	Interprétation
L1=[]	liste vide
L2=[0, 1, 2, 3]	4 élément indicé de 0 à 3
L3=['abc', ['def', 'ghi']]	liste incluses
L2[i], L3[i][j]	indice
L2[i:j]	tranche
len(L2)	longueur
L1+L2	concaténation
L1*3	répétition
for x in L2	parcours
3 in L2	appartenance
L2.append(4)	méthodes : agrandissement
L2.sort()	tri
L2.index()	recherche
L2.reverse()	inversion
del L2[k], L2[i:j]=[]	effacement
L2[i]=1	affectation par indice
L2[i:j]=[4, 5, 6]	affectation par tranche
range(4), xrange(0,4)	céation de listes / tuples d'entiers

Les dernières entrées de la table sont nouvelles. Les listes répondent aussi bien aux fonctions d'addition de répétition, qu'à l'appel de méthodes. Ces méthodes permettent l'ajout, la suppression, le tri et l'inversion.

2.3.1 Exercice sur les listes No 1.

```
# fichier : exo_file_1.py by J.Tschanz
# fonction : montre les fonctions principales sur les listes
# définition des listes
liste_une = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
liste_deux = ['a', 'b', 'c', 'd', 'e', 'f']
liste_trois = []
# affichage simple
print liste_une
# concaténation
print "Une concaténation : "
print liste_une + liste_deux
print "Une répetition : "
# répetiton
print liste_une*2
print "----------------------"
# affichage avec un indice
print "Extraction par indiçage : "
print liste_une[0]
# avec une tranche (entre 2 et 4)
print "Extraction par tranche : "
print liste_une[2:4]

# tranche de 4 à la fin
print liste_une[4:]
# tranche du début à la 4
print liste_une[:4]
print "----------------------"
print "Ajout dans une liste : "
liste_trois.append(liste_une)
liste_trois.append(liste_une)
print liste_trois
print "Remplacement : "
liste_trois[1] = liste_deux
print liste_trois
print "Extraction avec plusieurs indices : "
print liste_trois[0][4]
print liste_trois[1][2]
print "Supression : "
liste_trois[0] = []
del liste_trois[1]
print liste_trois

# fin du fichier exo_liste_1.py
```

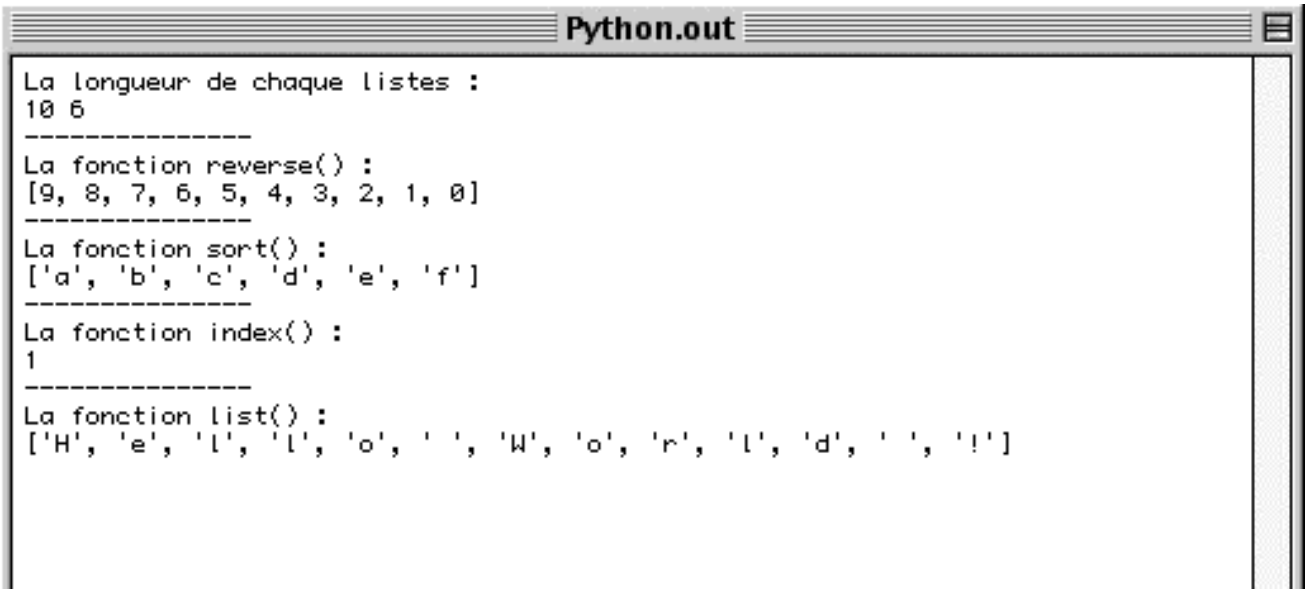

Python.out
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Une concaténation :
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c', 'd', 'e', 'f']
Une répetition :
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Extraction par indiçage :
0
Extraction par tranche :
[2, 3]
[4, 5, 6, 7, 8, 9]
[0, 1, 2, 3]

Ajout dans une liste :
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
Remplacement :
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], ['a', 'b', 'c', 'd', 'e', 'f']]
Extraction avec plusieurs indices :
4
c
Supression :
[[]]

2.3.2 Exercice sur les listes No 2.

```
# fichier : exo_file_2.py by J.Tschanz
# fonction : montre les fonctions principales sur les listes

# définition des listes
liste_une = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
liste_deux = ['f', 'e', 'd', 'c', 'b', 'a']

# afficher la longueur de chaque liste
print "La longueur de chaque listes : "
print len(liste_une), len(liste_deux)

print "----------"

# invertion de la liste
print "La fonction reverse() : "
liste_une.reverse()
print liste_une

print "----------"

# remise dans l'ordre de la liste_deux
print "La fonction sort() : "
liste_deux.sort()
print liste_deux

print "----------"

# recherche de l'indice depuis la valeur
print "La fonction index() : "
print liste_deux.index('b')

print "----------"

# création d'une list depuis une chaîne de caractères
print "La fonction list() : "
chaine = "Hello World !"
print list(chaine)

# fin du fichier exo_liste_2.py
```


Python.out
La longueur de chaque listes :
10 6

La fonction reverse() :
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

La fonction sort() :
['a', 'b', 'c', 'd', 'e', 'f']

La fonction index() :
1

La fonction list() :
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', ' ', '!']

2.4 Les Dictionnaires.

Le dictionnaire est un système très souple pour intégrer des données. Si on pense aux listes comme une collection d'objets ordonnés et portant un indice par rapport à la position, un dictionnaire est une liste comportant à la place de ces indices, un mot servant de clé pour retrouver l'objet voulu. Un dictionnaire est affecté par une paire d'accolades ({}).

>>> dico = {'japon' :'japan', 'chine' :'china'}

Les clés peuvent être de toutes les sortes d'objet non modifiables. Par contre la valeur stockée peut être de n'importe quel type.
Un dictionnaire n'est pas ordonné comme une liste, mais c'est une représentation plus symbolique de classement par clés. La taille d'un dictionnaire peut varier, et le dictionnaire est un type modifiable sur place.

Opération	Interprétation
d1 = {}	dictionnaire vide
d2={'one' : 1, 'two' : 2}	dictionnaire à duex éléments
d3={'count': {'one': 1, 'two': 2}}	inclusion
d2['one'], d3['count']['one']	indiçage par clé
d2.has_keys('one')	methodes : test d'appartenance
d2.keys()	liste des clés
d2.values()	liste des valeurs
len(d1)	longueur (nombre d'entrée)
d2[cle] = [nouveau]	ajout / modification
del d2[cle]	destruction

2.4.1 Exercice sur les dictionnaires No 1.

```
# file : exo_dico_1.py by J.Tschanz
# présentation des fonctions possibles avec les dictionnaires

dico_1 = {'one' : {'un' : 1}, 'two' : {'deux' : 2}}
dico_2 = {'japan' : 'japon', 'switzerland' : 'suisse'}
dico_3 = {}

# affichage simple
print "Nos deux dictionnaires : "
print dico_1
print dico_2
print "---------"

# affichage par indice de clé
print "Une recherche par clef : "
print dico_2['japan']
print dico_1['one']
print dico_1['one']['un']
print "---------"

# liste des clés d'un dico.
"La liste des clefs du dictionnaire 2 "
print dico_2.keys()
print "---------"

# liste des valeurs du dico.
print "La liste des valeurs du dictionnaire 2 "
print dico_2.values()
print "---------"

# la longueur du dico.
print "La longueur du dictionnaire 1 "
print len(dico_1)
print "---------"

# ajout d'une valeur ou modification
print "Ajout et modification dans le dictionnaire 2 "
dico_2['japan'] = ['nihon']
dico_2['germany'] = ['allemagne']
print dico_2
print "---------"
```

```
# suppression d'une valeur
print "Suppression dans le dictionnaire 2 "
del dico_2['japan']
print dico_2
print "---------"

# méthode de recherche d'appartenance
print "Méthode de recherche d'appartenance "
if (dico_1.has_key('one') == 1):
        print "dico_1 --> 'one' ! "
else:
        print "dico_1--> ?? "
if (dico_2.has_key('germany') == 1):
        print "dico_2 --> 'germany' ! "
else:
        print "dico_2--> ?? "
if (dico_1.has_key('japan') == 1):
        print "dico_1 --> 'japan' ! "
else:
        print "dico_1--> ?? "
# end of file : exo_dico_1.py
```

2.5 Les tuples.

Un des types de Python est le tuple. Le tuple est comme la liste, une collection ordonnée d'objets. Un tuple peut contenir n'importe quelle sorte d'objet, mais la grande différence avec la liste est que le tuple n'est pas modifiable sur place. Un tuple se déclare avec des valeurs entre parenthèses et non entre crochets comme la liste.

>>> tuple = (0, 1.4, 'world')

Les tuples peuvent être indicés pour la recherche d'un élément, une extraction est aussi possible. Ils supportent toutes sortes d'objets et même un tuple dans un tuple. Mais les tuples ne sont pas modifiables, donc comme les chaînes, ils ne peuvent pas grandir ou diminuer de taille.

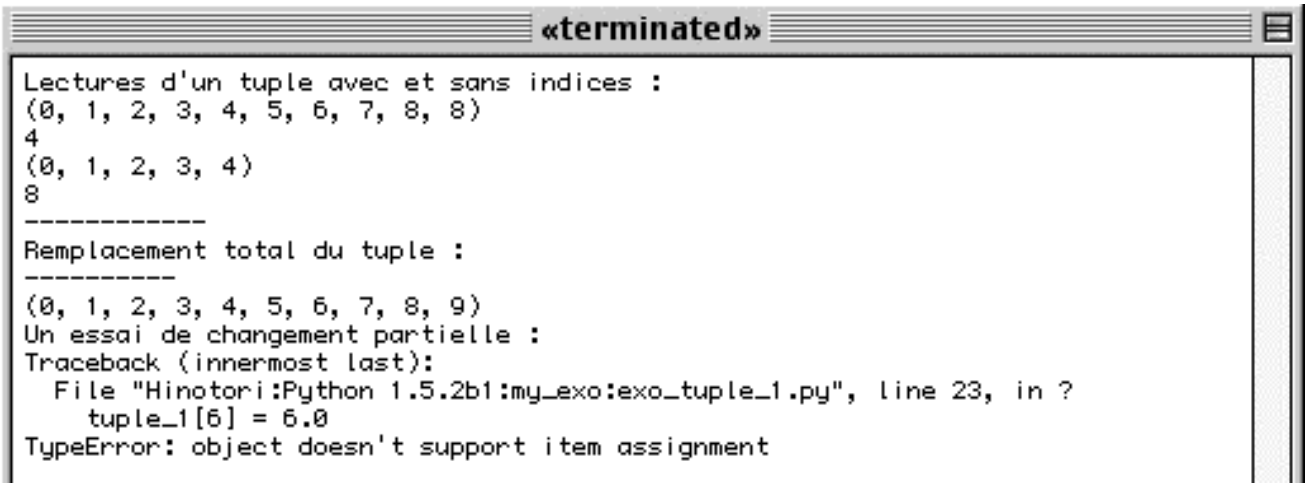
Opération	Interprétation
()	un tuple vide
n1 = (0,)	un tuple à un élément (et non une expression)
n2 = (0,1,2,3)	un tuple à quatre éléments
n2 = 0,1,2,3	un autre tuple à quatre éléments
n3 = ('abc', ('def', 'ghi'))	tuple avec inclusion
t[i], n3[i][j]	indiçage
n1[i:j]	tranche
len(n1)	longueur
n1+n2	concaténation
n2 * 3	répétition
for x in n2	itération
3 in s2	test d'appartenance

2.5.1 Pourquoi les tuples.

Une des réponses possible à cette question est que la non-possibilité de modifier les tuples assure une certaine intégrité. Vous pouvez être sûr qu'un tuple ne sera pas modifié à travers une référence ailleurs dans le programme.

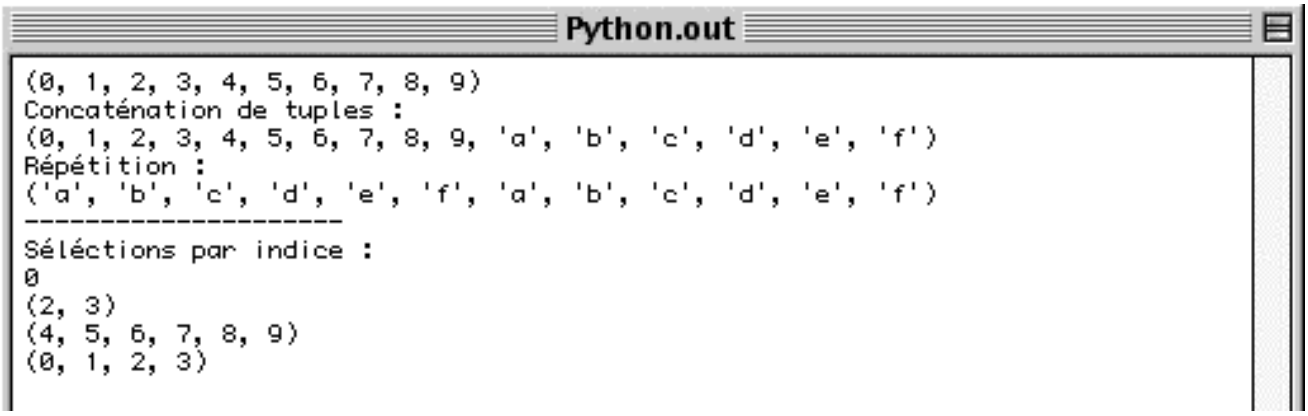
2.5.2 Exercice sur les tuples No 1.

```
# fichier : exo_tuple_1.py by J.Tschanz
# fonction : montre l'erreur lors de l'affectation d'un valeur.

# définition du tuple
tuple_1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 8,)

# lecture du tuple (idem à la liste)
print "Lectures d'un tuple avec et sans indices : "
print tuple_1
print tuple_1[4]
print tuple_1[:5]
print tuple_1[-2]

print "------------"
print "Remplacement total du tuple : "
tuple_1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
print "----------"
print tuple_1

# essai de changement de 6 par 6.0 (par assignement) !!

print "Un essai de changement partielle : "
tuple_1[6] = 6.0

# fin du fichier exo_tuple_1.py
```



2.5.3 Exercice sur les tuples No 2.

```
# fichier : exo_tuple_2.py by J.Tschanz
# fonction : montre les fonctions principales sur les tuples

# définition des listes
tuple_1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
tuple_2 = ('a', 'b', 'c', 'd', 'e', 'f')
tuple_3 = ()

# affichage simple
print tuple_1
print "Concaténation de tuples : "
print tuple_1 + tuple_2
print "Répétition : "
print tuple_2 * 2
print "----------------------"
print "Séléctions par indice : "
# affichage avec un indice
print tuple_1[0]

# avec une tranche (entre 2 et 4)
print tuple_1[2:4]
# tranche de 4 à la fin
print tuple_1[4:]
# tranche du début à la 4
print tuple_1[:4]
print "----------------------"

# insertion de tuples dans un tuple
tuple_3 = (tuple_1, tuple_2)
print tuple_3

# affichage avec plusieurs indices
print tuple_3[0][4]
print tuple_3[1][2]

#supression
tuple_3 = ()
print tuple_3

# afficher la longueur de chaque tuples
print len(tuple_1), len(tuple_2), len(tuple_3)

# fin du fichier exo_tuple_2.py
```


Python.out
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
Concaténation de tuples :
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c', 'd', 'e', 'f')
Répétition :
('a', 'b', 'c', 'd', 'e', 'f', 'a', 'b', 'c', 'd', 'e', 'f')

Séléctions par indice :
0
(2, 3)
(4, 5, 6, 7, 8, 9)
(0, 1, 2, 3)

2.6 Les fichiers.

Ce type intégré dans python se différencie des autres, car ce ne sont ni des nombres, ni des séquences, ni des maps. En fait la commande open permet à l'intérieur d'un programme python d'accéder à un fichier.

>>> mon_fichier = open('fichier', 'w')

Les commandes associées au traitement des fichiers ne se résument que par des méthodes.
Le tableau suivant montre les principales méthodes en vigueur sur les fichiers, il est important de dire que, lors de l'ouverture de l'objet-fichier celui-ci prend la forme de chaîne dans le programme Python.

Opération	Interprétation
sortie = open('/tmp/spam', 'w')	crée un fichier de sortie ('w' => écriture)
entre = open('donnee', 'r')	ouvre un fichier en entrée ('r' => lecture)
s = entre.read()	lit le fichier entier dans une chaîne
s = entre.read(N)	lit N octets (1 ou plus)
s = entre.readline()	lit la ligne suivante
L = entre.readlines()	lit la fichier dans une liste de lignes
sortie.write(s)	écrit s dans le fichier
sortie.writelines(L)	écrit toutes les lignes contenues pas L
sortie.close()	fermeture manuelle

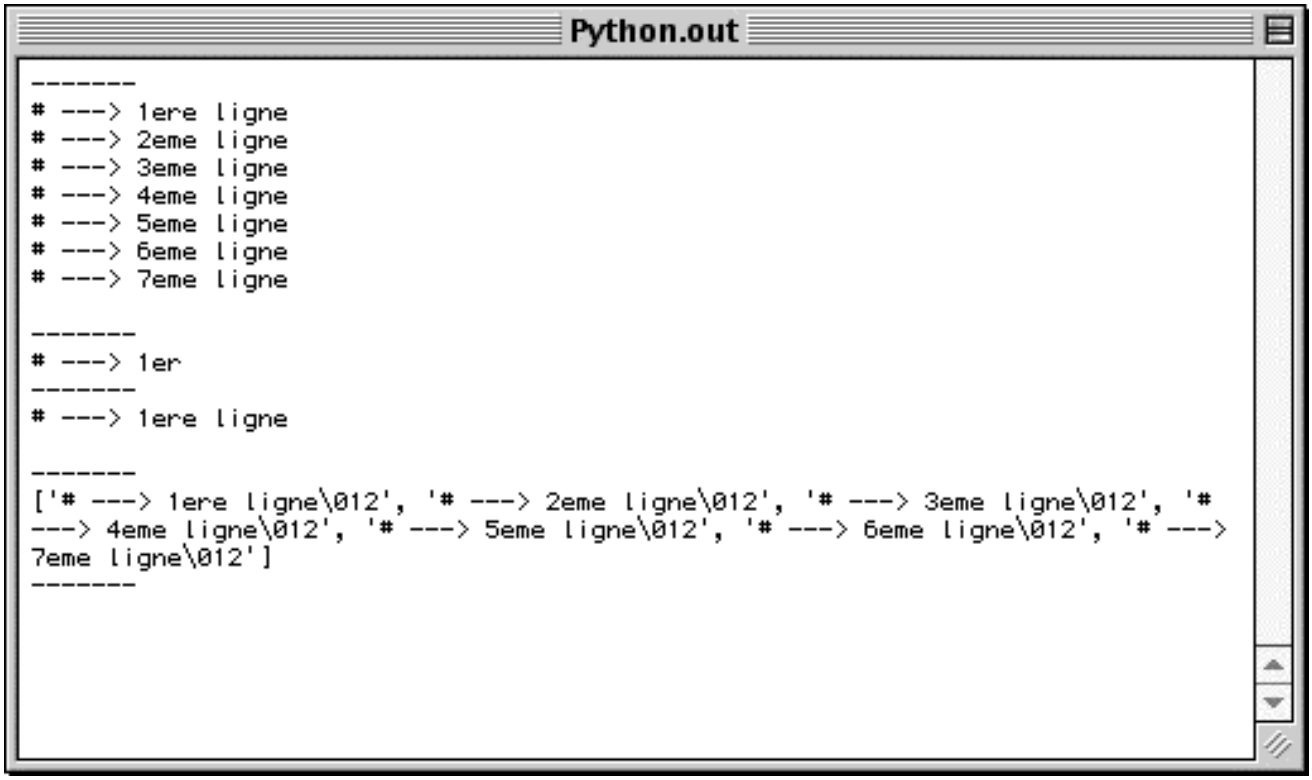
La fonction « close » ferme l'objet fichier en cours. Python, pour gérer l'espace mémoire, ferme dans certains cas le fichier lui-même, lorsqu'il n'est plus référencé nulle part. Cependant fermer un fichier manuellement permet une certaine clarté, ce qui est important dans une grande application.

2.6.1 Exercice sur les fichiers No 1.

```
# fichier : exo_read_file.py by j.tschanz
# cet exemple permet de voir comment sont mémorisées
# les lignes du ficher source.
# ouvre le fichier source en lecture et le referme après
# chaque lecture, afin de le reinitialiser au début.

source = open('read.py', 'r')
L1 = source.read()
source.close()

source = open('read.py', 'r')
L2 = source.read(10)
source.close()

source = open('read.py', 'r')
L3 = source.readline()
source.close()

source = open('read.py', 'r')
L4 = source.readlines()
source.close()
# affichage des valeurs
print "-------"
print L1
print "-------"
print L2
print "-------"
print L3
print "-------"
print L4
print "-------"
# fin du fichier : exo_read_file.py
```

Le fichier read.py :

```
# ---> 1ere ligne
# ---> 2eme ligne
# ---> 3eme ligne
# ---> 4eme ligne
# ---> 5eme ligne
# ---> 6eme ligne
# ---> 7eme ligne
```


Python.out

---> 1ere ligne
---> 2eme ligne
---> 3eme ligne
---> 4eme ligne
---> 5eme ligne
---> 6eme ligne
---> 7eme ligne

---> 1er

---> 1ere ligne

['# ---> 1ere ligne\012', '# ---> 2eme ligne\012', '# ---> 3eme ligne\012', '#
---> 4eme ligne\012', '# ---> 5eme ligne\012', '# ---> 6eme ligne\012', '# --->
7eme ligne\012']

2.6.2 Exercice sur les fichiers No 2.


```
# fichier : exo_file.py by j.tschanz
# filtre un document en enlevant tous les lignes dont le premier
# caractère est '#' (commentaire).

# ouvre le fichier source en lecture
source = open('exo_file.py', 'r')

# crée un fichier de sortie
copie = open('copie_exo.py', 'w')

# line prend la valeur d'une ligne du fichier source
for line in source.readlines():

# si le premier caractere est '#'
        if line[0] == '#':
# on retourne au départ
                continue
        else:
# écrit la ligne
                copie.write(line)
# ferme les deux fichiers
source.close()
copie.close()

# fin du fichier : exo_file.py
```

Fichier de sortie copie : exo.py :

```
source = open('exo_file.py', 'r')

copie = open('copie_exo.py', 'w')

for line in source.readlines():

        if line[0] == '#':
                continue

        else:
                copie.write(line)

source.close()
copie.close()
```

3 Les Variables.

Table des matières :

3 Les Variables.....36
3.1.1 L'affectation.....37
3.1.2 Règles sur les noms de variables.....37
3.1.3 La liaison. 37
3.1.4 Variable global. 37

3.1.1 L'affectation.

L'affectation permet de lier des objets à des noms de variables. Les affectations sont en général simples, mais il y a quand même quelques règles à suivre.
Python stocke des références sur des objets dans des noms ou des composantes de structure de données. Cela crée toujours des références à des objets et non des copies.
Les noms sont créés à la première affectation, il n'est pas nécessaire de prédéclarer les noms.
Les noms doivent être créés avant d'être référencés, en effet c'est une erreur d'utiliser un nom pas encore associé à une valeur. Python lève alors une exception.
Python comporte aussi des affectations implicites. Une affectation implicite est en fait toutes les affectations possibles qui ne comportent pas de =, par exemple lors d'importation de modules, ou d'instruction "for".
Il est aussi possible d'affecter des variables par paquets de tuples et de listes. Cette façon de faire permet des affectations par position, l'élément le plus à gauche du côté gauche de "=" prendra la valeur la plus à gauche du côté droit de "=".
L'affectation par cible multiples indiques que deux variables pointent sur le même objet, par exemple x = y = 2, ou x = 2 et y = x.

3.1.2 Règles sur les noms de variables.

La syntaxe pour les noms de variables doit commencer par une lettre ou un souligné, et peu se poursuivre avec un nombre quelconque de lettres, chiffres et soulignés.
Spam, spam et _spam sont corrects, mais 1_spam, spam: ou :spam ne le sont pas !
Python est sensible à la casse du nom de la variable, ainsi spam n'est pas égal à SPAM ou Spam.
Les mots réservés sont à part, Python lève une erreur si l'on emploie comme nom de variable un mot réservé (voir le chapitre 10).

3.1.3 La liaison.

Les variables sont modifiables et n'ont pas de type, c'est une simple référence sur un objet qui lui comporte un type. Il est donc possible d'affecter successivement plusieurs types à une même variable.

3.1.4 Variable global.

Par défaut, tout nom affecté dans une fonction local (propre) à celle-ci est créé uniquement quand la fonction est exécutée, et effacée en suite. L'instruction "global" permet d'affecter un nom au module englobant, Il sera donc visible depuis l'extérieur. Nous reverrons cela dans le chapitre dédié aux fonctions.

```
... global <nom>
... <instruction>
```

4 Les Instructions

Table des matières :

4 Les Instructions....38
4.1 Les instructions....39
4.1.1 Les expressions....39
4.2 Test If....39
4.2.1 Format général....39
4.2.2 Exercice de testes if No 1....40
4.2.3 Exercice de testes if No 2....41
4.3 Boucles while....42
4.3.1 break, continue, pass et le else de boucle !....42
4.3.2 Exercice de boucle while No 1....43
4.4 Boucle for....45
4.5 Boucles à compteurs et intervalles....45
4.5.1 range()....45
4.5.2 xrange()....46
4.5.3 Exercice sur la boucle for No 1....47

4.1 Les instructions.

Dans cette partie nous allons voir les instructions de base. Si le but de programmer est de faire des "choses" avec des "trucs" afin de sortir des "machins", alors les instructions sont les "choses" qui décrivent la procédure à effectuer pour sortir les "machins".

4.1.1 Les expressions.

En Python, il est aussi possible d'utiliser des expressions comme instructions. Comme le résultat ne sera pas sauvé, cela n'a donc de sens que si l'expression a des effets de bord utiles. Les expressions regroupent entre autre les appels de fonctions et de méthodes.

4.2 Test If.

Le test par "if" est le premier outil de sélection en Python. Le test "if" avec ses autres blocs permettent de choisir une partie du code à effectuer parmi d'autres. Le choix se fait par une opération logique.

4.2.1 Format général.

L'instruction "if" en Python peut comporter plusieurs tests conditionnels "elif", et se termine par un bloc optionnel "else". Le bloc qui sera exécuté sera le premier dont la condition est vraie, sinon le else, qui ne comporte aucune condition, sera exécuté. Le code suivant les instructions if, elif, else doit être indenté, et doit être présent !

forme de base.

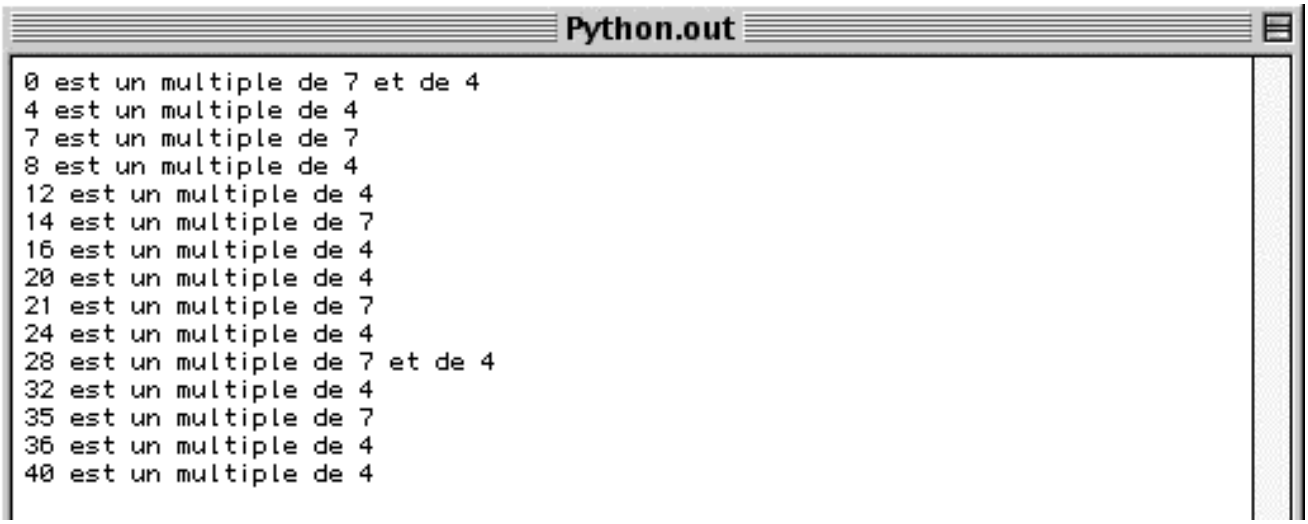
if <test1>:
    <instructions>
elif <test2>:
    <instructions>
elif <test3>:
    <instructions>
.....
else:
    <instructions>

4.2.2 Exercice de testes if No 1.

```
# fichier : exo_if_1.py by j.tschanz
# l'instruction simple avec le if.
# bien que plusieurs blocs sont sous une condition Vraie
# seulement le premier sera exécuté.

x = 0

if x == 0:
        print "x == 0"

elif x < 2:
        print "x < 2"

else:
        print " x = ?"

# fin du fichier : exo_if_1.py
```

4.2.3 Exercice de testes if No 2.

```
# fichier : exo_if_2.py by j.tschanz
# l'instruction simple avec le if.

x = range(41)

for i in x:
        if (i % 7) == 0:        # si modulo par 7 donne 0
                if (i % 4) != 0:# si modulo par 4 donne différent de 0
                        print "%d est un multiple de 7" % i
                else:
                        print "%d est un multiple de 7 et de 4" % i

        elif (i % 4) == 0:      # si modulo par 4 0
                print "%d est un multiple de 4" % i

        else:
                pass # ne fait rien

# fin du fichier : exo_if_2.py
```



4.3 Boucles while.

L'instruction Python "while" est la construction d'itération la plus générale. "while" exécute de manière répétitive le bloc indenté tant que le test de condition est réalisé. Si la condition est d'emblée fausse le bloc ne sera jamais exécuté.

"while" consiste en une ligne d'en-tête avec une expression de test, suivie d'un bloc de plusieurs instructions. Il peut y avoir une partie "else" optionnelle, qui est exécutée si le contrôle sort de la boucle sans utilisation de l'instruction break.

while <test> :
    <instructions>

else :
    <instructions>

4.3.1 break, continue, pass et le else de boucle !

Il est important de voir ces instructions en même temps que les boucles. Ces trois instructions, "pass", "break", et "continue", servent à gérer la continuité d'une boucle suivant les actions qui se passent à l'intérieur.

- L'instruction "break" a pour but de sortir de la boucle instantanément et de passer à la suite.

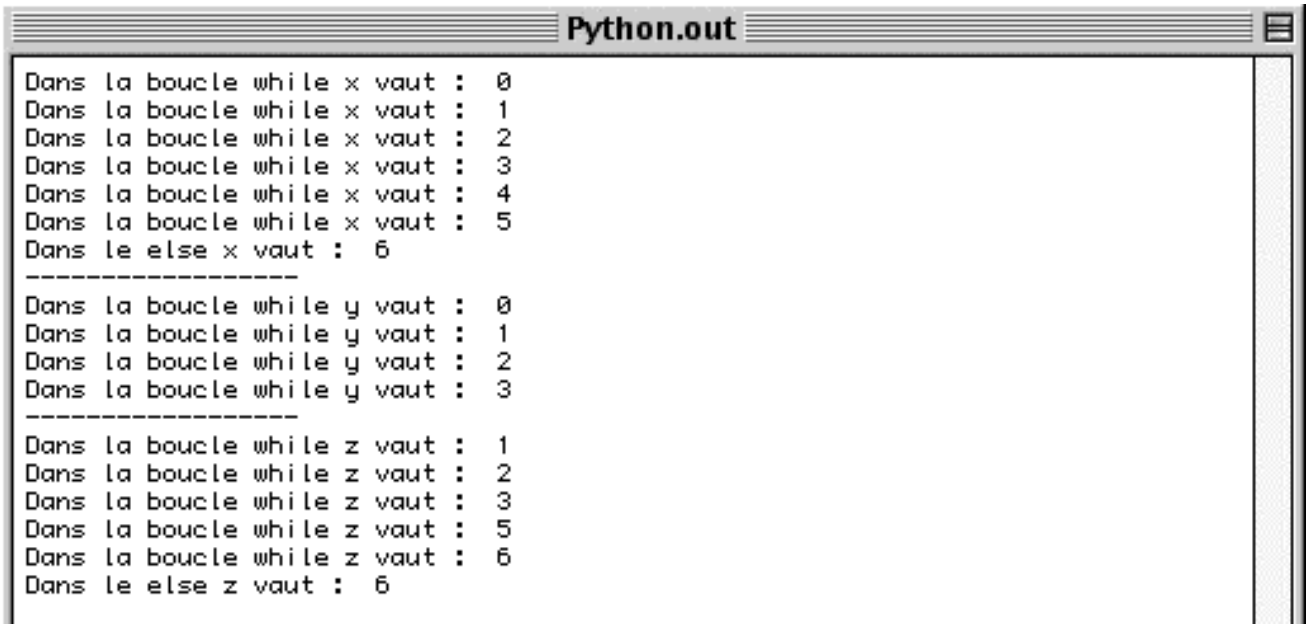
- "continue" saute au début de la boucle la plus imbriquée.

- "pass" ne fait rien du tout, mais comme on ne peut avoir une expression qui n'est pas suivie, "pass" peut servir à combler ce vide.

La boucle "else" exécute le bloc si la boucle se termine normalement. L'instruction "break" annule le passage dans la boucle "else".

4.3.2 Exercice de boucle while No 1.

```
# fichier : exo_while.py by j.tschanz
# l'expression while.

x = 0
y = 0
z = 0

while x < 6:
        print "Dans la boucle while x vaut : ", x
        x = x + 1
else:
        print "Dans le else x vaut : ", x

print "--------------------"

while y < 6:
        if y == 4:
                break   # Quand y == 4 on sort du while sans exécuter le bloc else.
        else:
                print "Dans la boucle while y vaut : ", y
                y = y + 1
else:
        print "Dans le else y vaut : ", y

print "--------------------"

while z < 6:
        z = z + 1
        if z == 4:
                continue        # On revient au départ et on continue, le bloc else sera exécuté.
        else:
                print "Dans la boucle while z vaut : ", z
else:
        print "Dans le else z vaut : ", z

# fin du fichier : exo_while.py
```


Python.out
Dans la boucle while x vaut : 0
Dans la boucle while x vaut : 1
Dans la boucle while x vaut : 2
Dans la boucle while x vaut : 3
Dans la boucle while x vaut : 4
Dans la boucle while x vaut : 5
Dans le else x vaut : 6

Dans la boucle while y vaut : 0
Dans la boucle while y vaut : 1
Dans la boucle while y vaut : 2
Dans la boucle while y vaut : 3

Dans la boucle while z vaut : 1
Dans la boucle while z vaut : 2
Dans la boucle while z vaut : 3
Dans la boucle while z vaut : 5
Dans la boucle while z vaut : 6
Dans le else z vaut : 6

4.4 Boucle for.

La boucle "for" est la séquence d'itération en Python, elle permet de traverser les éléments de tout objets qui répond aux opérations d'indiçage de séquence.
"for" fonctionne sur les chaînes, les listes, les tuples et d'autres objets issus de classes.

La boucle "for" commence par une ligne d'en-tête qui spécifie une cible d'affectation, ainsi qu'un objet qui sera itérer.

for <cible> in <objet> :
<instructions>

else :
<instructions>

La boucle "for" en Python fonctionne différemment qu'en C, elle affecte les objets de l'élément séquence à la cible un par un. A chaque affectation elle exécute le corps de la boucle. "for" quitte la boucle une fois que tous les éléments de l'élément séquence ont été parcourus, techniquement qu'une exception déclenchée par un indice hors-limite soit levée. "for" génère automatiquement l'indiçage de séquence.
Comme "while", "for" comprend, les instructions "pass", "break", "continue" et le bloc optionnel "else", qui sera exécuté si la boucle est quittée normalement (tous les éléments visités).

4.5 Boucles à compteurs et intervalles.

La boucle "for" englobe la plupart des boucles à compteurs et c'est donc le premier outil pour traverser une séquence. Avec les boucles à compteurs d'intervalles il est possible de créer des séquences afin de pouvoir spécialiser l'indiçage de "for" !

4.5.1 range().

L'instruction range retourne une liste d'entiers croissants successifs qui peuvent être utilisés comme index.

"Range" peut avoir de un à trois arguments.

Si il n'y a qu'un argument il représente le nombre d'éléments de la liste partant depuis zéro et indicé de un. Si "range" est appelé avec deux arguments, il s'agit de la borne de départ de la liste et celle d'arrivée. Chaque élément de la liste sera une valeur incrémentée de un, entre la borne de départ et celle d'arrivée (non comprise). Quand "range" comportent trois arguments il s'agit de la borne de départ de la liste, celle d'arrivée et le pas d'incrémentation.

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

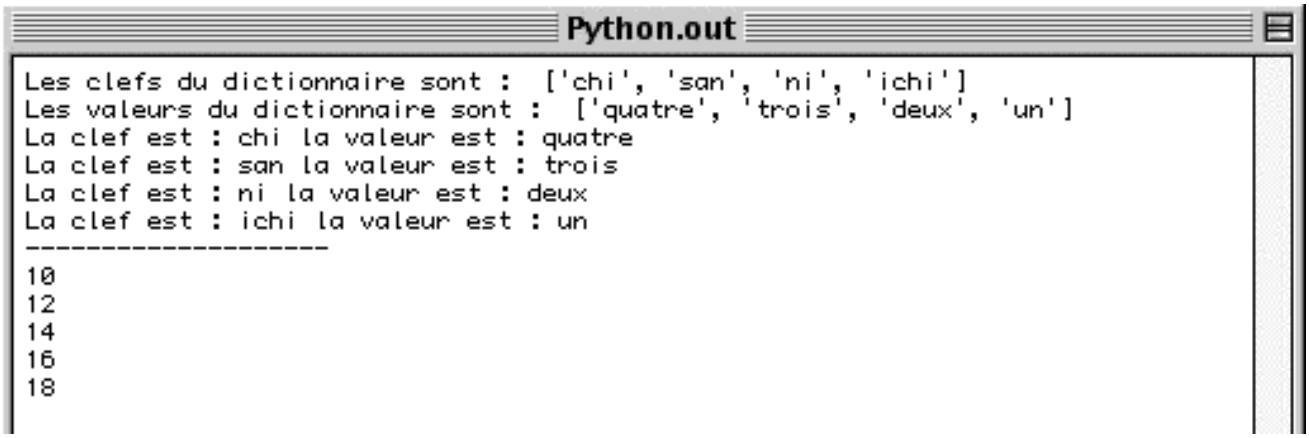
4.5.2 xrange().

"xrange" fonctionne comme "range", mais il permet de calculer au fur et à mesure. Ce qui à pour but de prendre moins de place mémoire lors de la création d'une grande séquence ! En plus les valeurs sont stockées dans un tuple et non dans une liste.

```
>>> x = xrange(10)
>>> x
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> y = xrange(10,20)
>>> y
(10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
>>> z = xrange(10,20,3)
>>> z
(10, 13, 16, 19)
```

4.5.3 Exercice sur la boucle for No 1.

```
# fichier : exo_for_1.py by j.tschanz
# l'instruction "for".

x = {"ichi" : "un", "ni" : "deux", "san" : "trois", "chi" : "quatre"}

print "Les clefs du dictionnaire sont : ", x.keys()
print "Les valeurs du dictionnaire sont : ", x.values()

for i in x.keys():
        print "La clef est : %s la valeur est : %s "% (i, x[i])

print "--------------------"

for i in range(10,20,2):
        print i

# fin du fichier : exo_for_1.py
```



5 Les Fonctions

Table des matières :

5 Les Fonctions 48
5.1 Les fonctions....49
5.1.1 Pourquoi les fonctions ?....49
5.1.2 Bases des fonctions.....49
5.1.2.1 def.....49
5.1.2.2 return.....49
5.1.3 Arguments d'une fonction : 50
5.1.3.1 par dictionnaire : 50
5.1.3.2 par tuple:....50
5.1.4 Les variables global....50
5.2 L'expression "Lambda". 51
5.2.1 Exercice avec une fonction No 1....52
5.2.2 Exercice avec une fonction No 2....53
5.2.3 Exercice avec une fonction No 3....54
5.2.4 Exercice avec une fonction No 4....55
5.2.5 Exercice avec une fonction No 5....56
5.2.6 Exercice avec une fonction No 6....57

5.1 Les fonctions.

Les fonctions sont une possibilité au programmeur d'encapsuler des instructions afin de pouvoir traiter des objets différents et plusieurs fois par exécution d'un programme.
Les fonctions permettent ainsi de spécifier des paramètres qui peuvent être différents à chaque fois que leurs codes sont exécutés.

5.1.1 Pourquoi les fonctions ?

Les fonctions permettent une réutilisation d'un bout de code. Une fonction permet aussi de grouper du code qui peut être exécuté ultérieurement, alors qu'avant le code était exécuté immédiatement.
Les fonctions ont aussi le grand avantage de découper le programme en blocs aux spécifications bien précises. De plus cela rend le code bien plus lisible ce qui n'est pas négligeable lors de grosses applications !

5.1.2 Bases des fonctions.

Nous avons déjà utilisé des fonctions précédemment, en effet les fonction len(), open(), font parties d'un groupe de fonctions de base (__builtin__, voir chapitre 1 et 10).
Ici nous allons voir les expressions utiles pour la création de fonctions personnelles :

5.1.2.1 def.

L'instruction "def" définis une nouvelle fonction, elle est suivie du nom de la fonction et se termine par ":". Le code se trouvant sous une instruction "def" devra être indenté d'une fois au moins ! En bref "def" est une instruction exécutable, elle génère un nouvel objet portant le nom fourni.

```
>>> def ma_fonc():
...             <instruction>
```

5.1.2.2 return.

"return" quitte la fonction et retourne la valeur <valeur>, en affectant ainsi la variable appelante.

"return" peut retourner un tuple.

```
...     <instruction>
...     return <valeur>
```

5.1.3 Arguments d'une fonction :

Il est possible de passer des arguments sous plusieurs formes, la méthode la plus simple consiste à envoyer les valeurs voulues par la fonction en respectant leurs nombres et leurs types. La forme générale pour passer des arguments est "fonction(arg_un, arg_deux, ..., argn)", et la réception est "def fonction(arg_un, arg_deux,..., narg):". Le nom envoyé et celui de réception peuvent être totalement différents !
Par cette méthode il est aussi possible de définir des valeurs par défaut, par exemple au cas ou l'utilisateur ne donnerait pas une information à une question traitée ensuite. La mise par défaut se fait dans la définition de la fonction :

```
>>> def fonction(arg_un=0, arg_deux=2, ..., narg='Hello World'):
```

Ici les valeurs par défaut sont de type différent, cela n'a aucune importance. Si il y a une valeur envoyée, alors la valeur par défaut est ignorée !
Lorsque le nombre d'arguments est arbitraire il est possible d'envoyer ceux-ci sous deux formes différentes. En effet en plaçant "*arg" dans la définition de la fonction les arguments seront reçus sous forme d'un tuple.
Par contre avec la forme "**arg" ils seront sous la forme d'un dictionnaire. Pour passer des arguments sous la forme d'un dictionnaire il faut leurs attribuer un nom, car les dictionnaires ne sont pas classés par indice (comme les tuples), mais par clés. Ceci se fait lors de l'appel de la fonction :

5.1.3.1 par dictionnaire :

```
>>> def ma_fonction(**arg):
>>> ...                  pass
>>> ma_fonction(arg_un=un_arg, arg_deux=deux_arg)
```

5.1.3.2 par tuple:

```
>>> def ma_fonction(*arg):
>>> ...                  pass
>>> ma_fonction(arg_un, arg_deux)
```

5.1.4 Les variables global.

Lors de la création d'une variable dans une fonction celle-ci n'est pas forcément visible depuis le module de base. Que veut dire global en Python ? global est une sorte de "visible au plus haut niveau du module", une "déclaration" global doit se faire dans une fonction pour que la variable puisse être utilisée depuis un autre bloc !

```
>>>def ma_fonc():
>>>    global x #Valeur global
>>>    y = 12 #Valeur local
```

5.2 L'expression "Lambda".

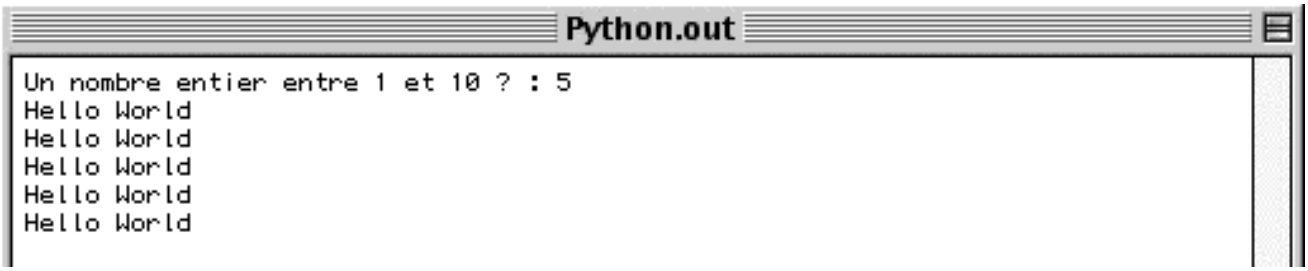
Outre l'instruction "def", Python offre une autre forme pour créer des objets fonctions, il s'agit de l'expression "lambda" (similaire à sa forme sous LISP). La forme générale est le mot-clé lambda suivi d'un ou plusieurs arguments puis d'une expression placée après deux-points. Lambda est un expression, et non une fonction. Par conséquent elle peut apparaître là ou "def" ne peut pas prendre place (à l'intérieur d'une constante de liste, par exemple !).
Le corps de lambda est similaire à ce que l'on place dans l'instruction return du bloc def.

```
>>> def fonc(x,y,z): return x+y+z
...
>>> f(2,3,4)
9

>>> f=lambda x,y,z: x+y+z
>>> f(2,3,4)
9
```

Il est à noter que les valeurs par défaut fonctionnent aussi avec l'expression lambda !

5.2.1 Exercice avec une fonction No 1.

```
# fichier : exo_func_1.py by j.tschanz
# cet exemple permet de voir une fonction simple.

def mon_affichage():
        print "Hello World"

x = input("Un nombre entier entre 1 et 10 ? : ")

for i in range(x):
        mon_affichage()

# fin du fichier : exo_func_1.py
```

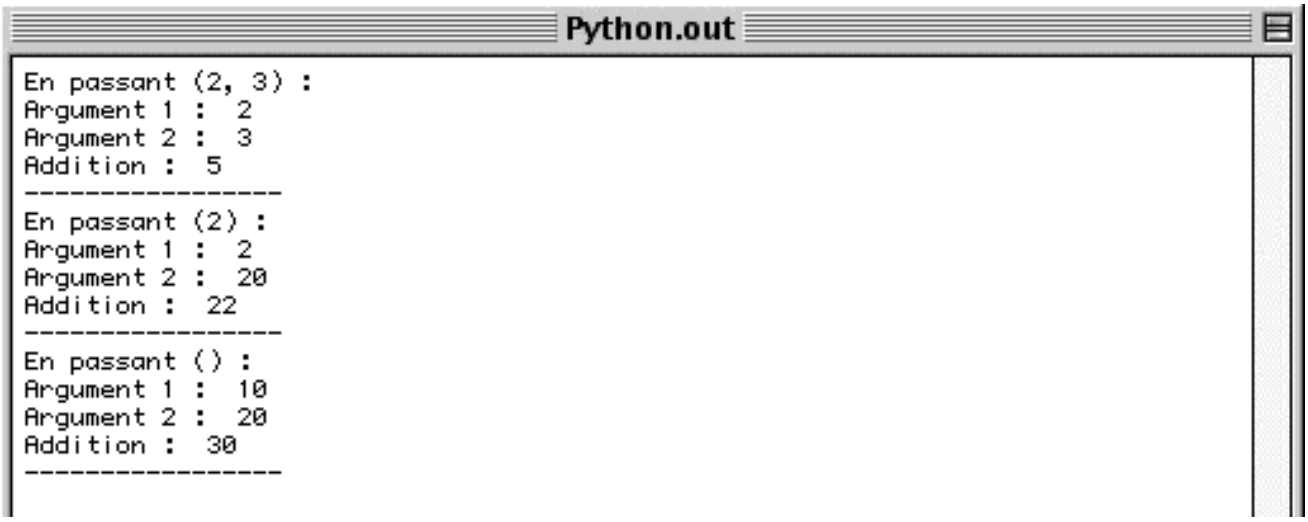


5.2.2 Exercice avec une fonction No 2.

```
# fichier : exo_func_2.py by j.tschanz
# cet exemple permet de voir une fonction simple.

def mon_affichage(x, y):
        print "Argument 1 : ",  x
        print "Argument 2 : ", y
        print "Addition : ",  (x+y)

arg_1 = raw_input("Entrez le premier argument : ")
arg_2 = raw_input("Entrez le deuxième argument : ")
mon_affichage(arg_1, arg_2)

# fin du fichier : exo_func_2.py
```

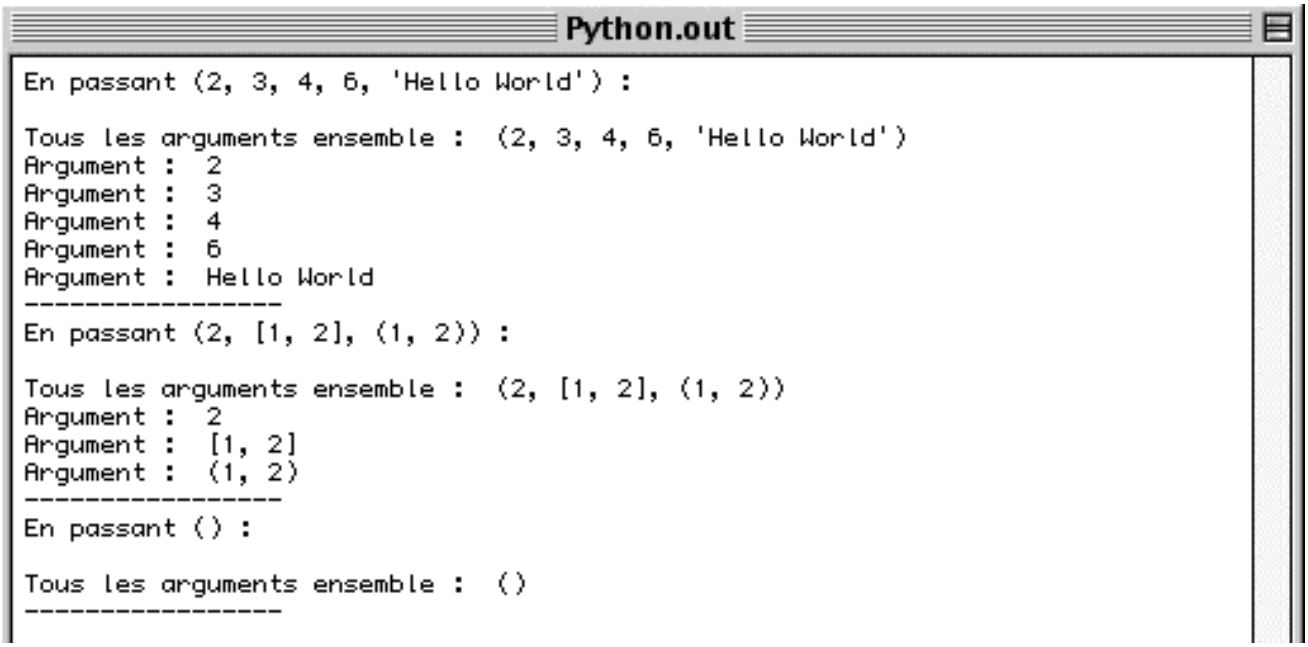
5.2.3 Exercice avec une fonction No 3.

```
# fichier : exo_func_3.py by j.tschanz
# valeur par défaut pour les arguments.

def mon_affichage(x=10, y=20):
        print "Argument 1 : ",  x
        print "Argument 2 : ", y
        print "Addition : ",  (x+y)

print "En passant (2, 3) : "
mon_affichage(2, 3)
print "-----------------"
print "En passant (2) : "
mon_affichage(2)
print "-----------------"
print "En passant () : "
mon_affichage()
print "-----------------"

# fin du fichier : exo_func_3.py
```



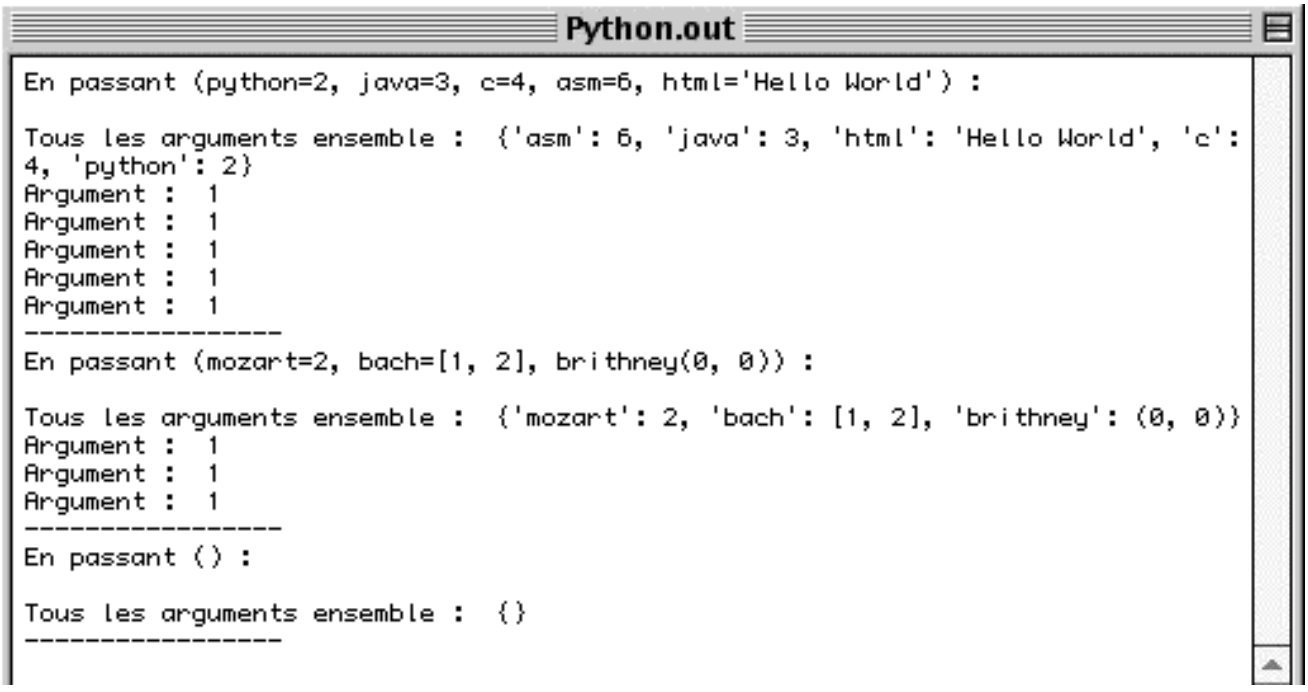
5.2.4 Exercice avec une fonction No 4.

```
# fichier : exo_func_4.py by j.tschanz
# passage d'arguments par tuple.

def mon_affichage(*arg):
        print "  "
        print "Tous les arguments ensemble : ", arg
        for i in arg:
                print "Argument : ", i


print "En passant (2, 3, 4, 6, 'Hello World') : "
mon_affichage(2, 3, 4, 6,  "Hello World")
print "-----------------"
print "En passant (2, [1, 2], (1, 2)) : "
mon_affichage(2, [1, 2], (1, 2))
print "-----------------"
print "En passant () : "
mon_affichage()
print "-----------------"

# fin du fichier : exo_func_4.py
```



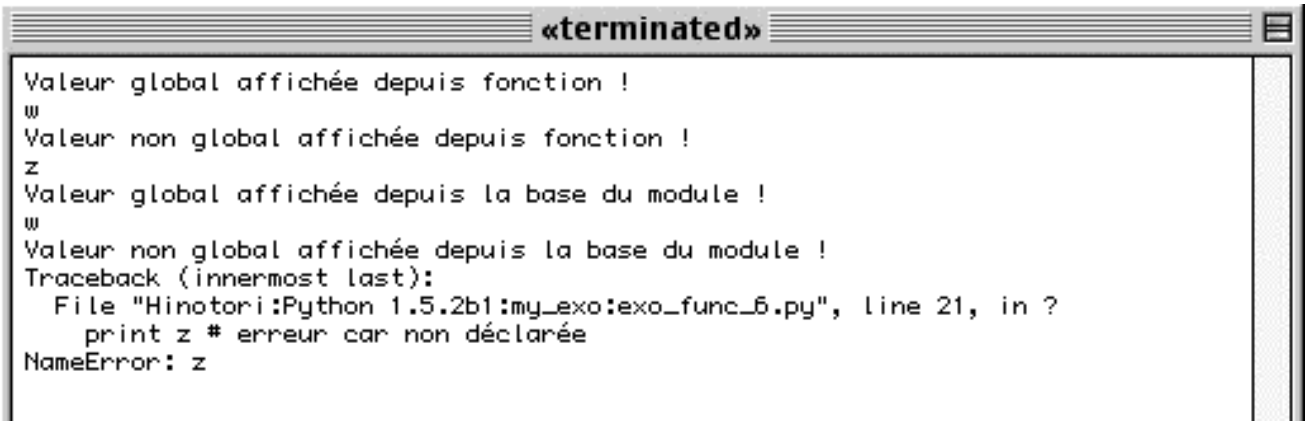
5.2.5 Exercice avec une fonction No 5.

```
# fichier : exo_func_5.py by j.tschanz
# passage d'arguments par dictionnaire.

def mon_affichage(**arg):
        print "  "
        print "Tous les arguments ensemble : ", arg
        for i in arg.keys():
                print "Argument : ", arg.has_key(i)


print "En passant (python=2, java=3, c=4, asm=6, html='Hello World') : "
mon_affichage(python=2, java=3, c=4, asm=6, html='Hello World')
print "-----------------"
print "En passant (mozart=2, bach=[1, 2], brithney(0, 0)) : "
mon_affichage(mozart=2, bach=[1, 2], brithney=(0, 0))
print "-----------------"
print "En passant () : "
mon_affichage()
print "-----------------"

# fin du fichier : exo_func_5.py
```



5.2.6 Exercice avec une fonction No 6.

```
# fichier : exo_func_6.py by j.tschanz
# la valeur global !

def fonc_2():
        global w
        w = 'w'
        print w

def fonc_1():
        z = 'z'
        print z

print "Valeur global affichée depuis fonction !"
fonc_2()
print "Valeur non global affichée depuis fonction !"
fonc_1()

print "Valeur global affichée depuis la base du module !"
print w
print "Valeur non global affichée depuis la base du module !"
print z  # erreur car non déclarée

# fin du fichier : exo_func_6.py
```



6 Les Modules.

Table des matières :

6 Les Modules. 58
6.1 Les modules. 59
6.1.1.1 Notion de base sur les modules 59
6.1.2 Utilisation. 59
6.1.3 L'instruction import. 59
6.1.4 L'instruction from 60
6.1.5 L'instruction reload 60
6.1.6 Modifier le chemin de recherche des modules ! 60
6.1.7 Exercice sur un module N0 1. 61
6.1.8 Exercice sur un module N0 2 et 3. 62
6.1.8.1 Le module : 62
6.1.8.2 Exercice 2. 63
6.1.8.3 Exercice 3. 64

6.1 Les modules.

Les modules en Python regroupent l'unité d'organisation de plus haut niveau pour les programmes. Les modules fournissent un moyen facile d'organiser des composants afin de former un système.
Les modules remplissent plusieurs rôles, dont celui de conserver de manière permanente du code dans un fichier. Contrairement au code tapé à l'invité interactif le code contenu dans un module est conservé, de plus il peut être rechargé et re-éxecuté autant de fois que nécessaire. Un module est aussi une partition de l'espace de noms du système, car dès qu'un module est chargé tous se qui se trouve dans se module "existe", le code que vous exécutez ou les objets créés sont toujours contenus de façon implicite dans un module. Du point de vue fonctionnel, les modules permettent, d'une façon pratique, de partager des ressources entre différents programmes.

6.1.1.1 Notion de base sur les modules.

Un module est très simple à créer car ce n'est que du code, Python, ou dérivé du C. Les trois instructions importantes concernant les modules sont "import", "from" et "reload". Pour qu'un module soit importable il faut que le répertoire soit visible par Python (PYTHONPATH).

Opération	interprétation
import mod	Récupère un module en entier
from mod import nom	Récupère un nom particulier depuis un module
from mod import *	Récupère tous les noms à la racine d'un module
reload(mod)	Force le rechargement du code d'un module chargé

6.1.2 Utilisation.

Pour voir quels répertoires sont visible par Python il faut importer le module "sys" :
>>> import sys
>>> sys.path
Ceci vous indiquera les répertoires accessibles.

6.1.3 L'instruction import.

L'instruction "import <module>" importe tout le module <module>, en exécutant le total du module. Le module est exécuté en premier avant de pouvoir travailler sur les objets.

>>> import <module>

6.1.4 L'instruction from.

L'instruction "from" s'utilise avec "import" de la manière "from <module> import <nom>". Là seulement l'objet <nom> du module <module> sera importé, le code restant du module sera aussi exécuté, mais les objets ne seront pas mémorisés et les éventuelles variables ne seront pas affectées. Si <nom> est remplacé par *, on obtient alors une copie de tous les noms définis a la racine du module.

>>> from <module> import <nom_1>, <nom_2>
*>>> from <module_2> import **

La grande différence entre "import <module>" et "from <module> import *" est que les noms ne sont pas mémorisés sur le même niveau. Le premier sera mémorisé sous le nom du module, tandis qu'avec "from" la valeur est directement visible dans la racine.

6.1.5 L'instruction reload.

La fonction "reload" est en fait un fonction se trouvant dans __builtin__ (comme print), cette fonction s'exécute par "reload(<nom>)" et a pour but de recharger le module <nom>. Le code du module sera de nouveau exécuté depuis le début et les variables qui auraient pu changer seront remises à leurs valeurs d'origine. Cette commande ne marche que pour les modules entièrement importés !

>>> reload(<module>)

6.1.6 Modifier le chemin de recherche des modules !

Il est possible au lancement d'un programme de modifier et/ou de rajouter des répertoires accessibles par Python, afin que celui-ci puisse trouver les modules à charger.

Pour changer les répertoires de recherche :
>>> import sys
>>> sys.path['rep_1\\rep_2'] # protégez l'antislash avec un antislash !!
>>> sys.path # pour voir le résultat !

Pour ajouter un ou des répertoires de recherche :
>>> import sys
>>> sys.path.append('rep_1\\rep_2') # protégez l'antislash avec un antislash !!
>>> sys.path # pour voir le résultat !

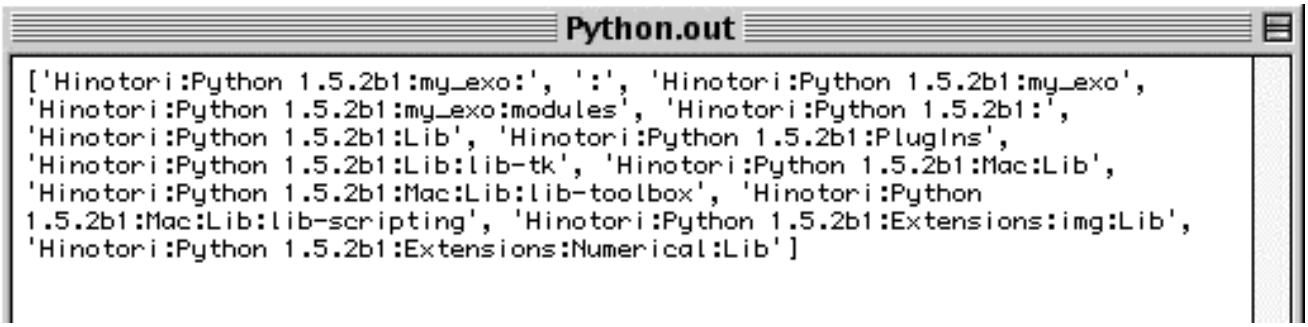
6.1.7 Exercice sur un module N0 1.

 # fichier : module_1.py by J.Tschanz
 # fonction : visualisation des répertoires visibles par python.

 # import du module système
import sys

 # montre les répertoires visible par Python
print sys.path

 # fin du fichier module_1.py



6.1.8 Exercice sur un module No 2 et 3.

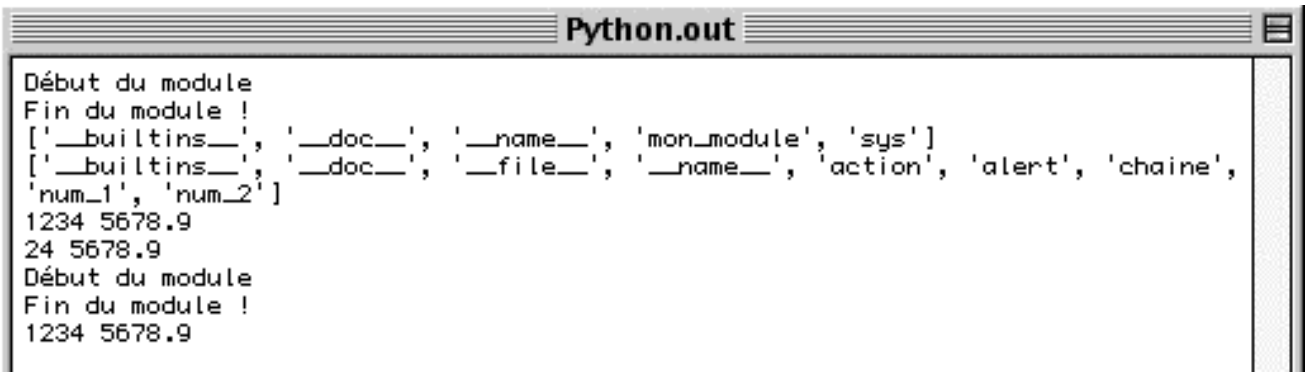
6.1.8.1 Le module :

Le module qui sera chargé par la suite :

```
# start of mon_module.py by j.tschanz

print "Début du module"

num_1, num_2 = 1234, 5678.9
chaine = "le module : mon_module.py"

def action():
        print "Action !!"

def alert():
        for i in range(4):
                print "\a"


print "Fin du module !"

# end of mon_module.py
```

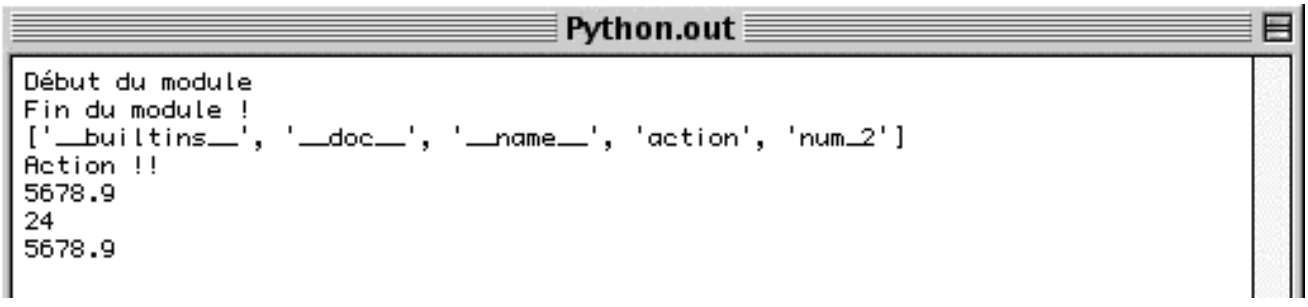
6.1.8.2 Exercice 2.

```
# fichier : module_2.py by J.Tschanz
# fonction : voir l'instruction import et la méthode reload.

# import du module système
import sys
# import de mon module
import mon_module

# affiche tous les modules chargés
print dir()

# affiche les valeurs chargées du module <mon_module>
print dir(mon_module)

# affichage des valeurs
print mon_module.num_1, mon_module.num_2

# changement d'une des valeurs
mon_module.num_1 = 24
print mon_module.num_1, mon_module.num_2

# affichage des valeurs après un reload
reload(mon_module)
print mon_module.num_1, mon_module.num_2

# fin du fichier module_2.py
```



6.1.8.3 Exercice 3.

```
# fichier : module_3.py by J.Tschanz
# fonction : l'expression from.

# import de mon module
from mon_module import num_2, action


# affiche tous les modules chargés et valeurs
print dir()

# la function
action()

print num_2

num_2 = 24

print num_2

# équivalent de la fonction reload.
from mon_module import num_2
print num_2


# fin du fichier module_3.py
```



7 Les classes

Table des matières :

7 Les classes 65
7.1 Les classes 66
7.1.1 Un peu de vocabulaire. 66
7.1.2 La surcharge des opérateurs : 66
7.2 Exercice sur les classes. 67
7.2.1 Définition d'une classe. 67
7.2.2 L'héritage des classes. 69
7.2.3 Les retours possible avec une classe. 71
7.2.4 La surcharge des opérateurs. 73

7.1 Les classes.

Jusqu'à maintenant nous avons seulement utilisé la programmation fonctionnelle. Mais Python offre la possibilité de programmer en orienté objet.
La programmation orientée objet permet un niveau d'abstraction plus fin. On considère qu'un langage est orienté objet lorsqu'il supporte au moins les trois mécanismes suivant :

L'encapsulation : ce mécanisme permet de mettre en arrière plan les données d'un objet, en mettant plutôt en en avant le comportement de cet objet.

L'Héritage : Cette relation permet de créer de nouveaux objets, et méthodes, en utilisant les caractéristiques de classes déjà existantes.

Polymorphisme : le polymorphisme est un mécanisme qui permet à une méthode d'agir différemment en fonction du type d'objet qui est la cible de l'opération.

7.1.1 Un peu de vocabulaire.

Classe : Objet ou instruction définissant des membres et/ou des méthodes ! Une classe se crée avec le mot réservé class.

Instance : Attention ce mot dans le contexte de la programmation orienté objet prend la définition du mot anglais qui signifie : cas particulier, ou exemple. Et non le mot français qui signifie : prière, demande urgente !
Bref une instance est un objet créé à partir d'une classe, donc elle hérite de ces membres et de ses méthodes.

Membre : Attribut d'une classe ou d'une de ses instances.

Méthode : Attribut d'une classe qui est associé à une fonction. Une méthode se crée avec le mot réservé def (semblable à une fonction)

Self : Nom donné conventionnellement à l'instance d'une classe dans la définition de ses méthodes.

Héritage : Mécanisme permettant à une instance de bénéficier des membres et des méthodes d'autres classes. La classe enfant (ou sous classe) est la classe qui hérite de membres et de méthodes d'une classe parent (ou super classe).

7.1.2 La surcharge des opérateurs :

À travers une méthode qui porte un nom spécifique (en général __xxx__) il est possible avec Python d'intercepter certaines fonctions. En effet quand une instance de classe utilise une opération associée Python appelle automatiquement la méthode appartenant à la classe de l'instance.

Voilà à quoi peut servir la surcharge des opérateurs :

La surcharge des opérateurs permet aux classes d'intercepter les opérations normales de Python.

Les classes peuvent surcharger tous les opérateurs des expressions de Python.

Les classes peuvent aussi surcharger les opérations associées aux objets.

La surcharge permet aux instances de classes d'agir de façon plus proche des types intégrés.

La surcharge des opérateurs est implémentée en fournissant des méthodes de classes nommées spécialement.

7.2 Exercice sur les classes.

Pour comprendre les classes nous allons étudier un exemple permettant de suivre les différentes possibilités de ces objets. Une classe est un modèle sur lequel des instances en deviennent des cas particuliers !

7.2.1 Définition d'une classe.

Pour notre exemple, nous allons prendre une classe qui prend en compte les membres d'une équipe de basket. Notre classe est le profil de base d'un joueur de basket avec comme paramètres, le numéro du maillot, le nombre de points marqués durant la saison, et le nombre de passes décisives (passe qui aboutit par un panier) durant la saison.
Notre classe "Joueur" devient :

```
>>> class Joueur:
...     def __init__(self, num="x", pts=0, passe=0):
...             self.num = num
...             self.pts = pts
...             self.passe = passe
```

Dans cette première définition nous avons une classe qui comporte une méthode. Cette méthode __init__ permet lors de l'affectation de prendre les arguments et d'en faire des membres de l'instance.
Nous allons rajouter dans notre class une méthode permettant d'évaluer la performance du joueur, ne retournant rien, mais affichant une note allant de 1 à 5 par rapport au nombre de points !:

```
# Fichier joueur.py by j.tschanz
# C'est une classe simple !

class Joueur:
        def __init__(self, num="x", pts=0, passe=0):
                self.num = num
                self.pts = pts
                self.passe = passe
# passge d'arguments à membres de l'instance.

        def note(self):
                if self.pts < 30:
                        return(1)
                elif self.pts < 60:
                        return(2)
                elif self.pts < 80:
                        return(3)
                elif self.pts < 100:
                        return(4)
                elif self.pts > 100:
                        return(5)
                else:
                        pass

# fin de la class Joueur et du fichier joueur.py
```

Maintenant nous allons évaluer les performances de nos joueurs en les déclarants instances (cas particulier !) de la class Joueur :

```
# Fichier equipe.py by j.tschanz
# Les joueurs
from joueur import *
jack = Joueur("23", 84, 6)     # Création de l'instance
bob = Joueur("11", 125, 2)
luc = Joueur("2", 34, 10)
mark = Joueur("45", 18, 34)
billy = Joueur("98", 10, 2)

print "Les performances"
print "Jack : %s" % (jack.note())      # Affichage de la note
print "Bob : %s" % (bob.note())
print "Luc : %s" % (luc.note())
print "Mark : %s" % (mark.note())
print "Billy : %s" % (billy.note())

# fin de equipe.py
```

La sortie de l'évaluation donne :

Maintenant les joueurs sont des instances de la classe Joueur, et les numéros de maillot, le nombre de points et le nombre de passes sont des membres de cette instance.
Pour afficher un membre d'une instance il suffit d'inscrire le nom de l'instance suivit par le nom du membre, les deux étant séparés par un point.

```
>>> jack.pts
84
>>> jack.num
'23'
>>> jack.passe
6
>>> jack
<joueur.Joueur instance at 668c100>
>>> dir(jack)
['num', 'passe', 'pts']
>>>
```

7.2.2 L'héritage des classes.

Jusque là nous avons vu un exemple simple d'encapsulation des classes, maintenant toujours en suivant notre exemple nous allons faire un héritage de classe.
En effet notre joueur Mark na pas de bonnes performances car il est passeur et par conséquent ne marque pas beaucoup de points. Nous allons créer une nouvelle classe qui sera une classe enfant de la classe Joueur (donc Joueur sera la classe parent !), notre fichier de classe devient :

```
# Fichier joueur.py by j.tschanz
# C'est une classe simple !
class Joueur:

    def __init__(self, num="x", pts=0, passe=0):
        self.num = num
        self.pts = pts
        self.passe = passe
```

```
# passge d'arguments à membres de l'instance.
        def note(self):
                if self.pts < 30:
                        return(1)
                elif self.pts < 60:
                        return(2)
                elif self.pts < 80:
                        return(3)
                elif self.pts < 100:
                        return(4)
                elif self.pts > 100:
                        return(5)
                else:
                        pass

# ********************************************

class Passeur(Joueur):
# classe Passeur avec comme parent Joueur.
        def note(self):
                if self.passe < 15:
                        return("faible")
                elif self.passe < 30:
                        return("moyen")
                elif self.pass > 30:
                        return("bien")
                else:
                        pass

# fin de la class Joueur et du fichier joueur.py
```

Il faut savoir qu'une classe enfant peut avoir plusieurs classes parents, comme vous l'avez compris les classes parent se trouvent dans la parenthèse de la ligne de définition de la classe. L'ordre de hiérarchie va de gauche à droite donc si :

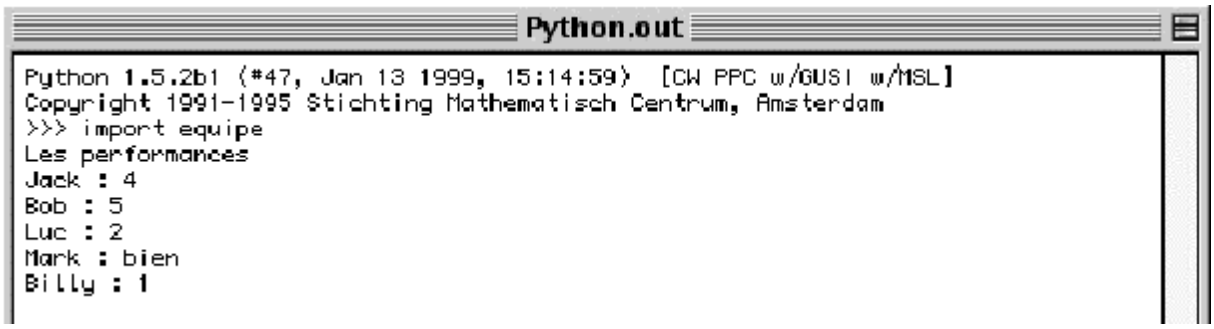
```
>>> class A(B, C)
```

La classe A est enfant de B avant d'être enfant de C, se qui veut dire que les méthodes seront cherchée d'abord dans A puis B et si elles ne s'y trouvent pas ensuite dans C. Les méthodes entre enfants et parents peuvent porter le même nom !

Depuis là notre joueur pourra avoir son propre barème d'evaluation, mais pour cela il faut encore changer son affectation dans le fichier de base :

```
# Fichier equipe.py by j.tschanz
# Les joueurs
from joueur import *
jack = Joueur("23", 84, 6)
bob = Joueur("11", 125, 2)
luc = Joueur("2", 34, 10)
mark = Passeur("45", 18, 34)                # Changement d'affectation.
billy = Joueur("98", 10, 2)

print "Les performances"
print "Jack : %s" % (jack.note())
print "Bob : %s" % (bob.note())
print "Luc : %s" % (luc.note())
print "Mark : %s" % (mark.note())
print "Billy : %s" % (billy.note())

# fin de equipe.py
```

La sortie est :



Et voilà notre classe Passeur a bien fonctionné, tous les arguments ont été traités avec __init__ de la classe Joueur, mais la méthode note à été celle de la classe Passeur.

7.2.3 Les retours possibles avec une classe.

Nous avons vu jusqu'à maintenant l'encapsulation des classes qui permettent de ne s'occuper que de l'objet et non de ces membres. Nous avons aussi vu la possibilité de faire des sous-classes reprenant les méthodes de leurs classes parents.

Maintenant nous allons créer dans la classe Joueur une méthode permettant d'additionner les points et le nombre de passes décisives.

```
def total(self, *other):
        x = self.pts
        y = self.passe

        for i in other:
                x = x+i.pts
                y = y+i.passe

        return("total", x, y)
```

Cette méthode retourne un tuple contenant un nom, le nombre de points et le nombres de passes décisives. Pour calculer le total il suffit après le lancement du programme équipe d'entrer la commande :

```
>>> d = jack.total(bob, luc, mark, billy)
>>> print d
('total', 271, 54)
```

"d" n' est qu'une variable, mais il est possible dans un retour d'affecter une classe à une variable. Par exemple si on change la méthode par :

```
def total(self, *other):
        x = self.pts
        y = self.passe

        for i in other:
                x = x+i.pts
                y = y+i.passe

        return Joueur("total", x, y)     # La variable devient une Instance.
```

La variable 'd' deviendra une instance de la classe joueur, il sera donc possible d'appeler la méthode note.

```
>>> d = jack.total(bob, luc, mark, billy)
>>> print d
<joueur.Joueur instance at 663bf00>
>>> d.note()
5
>>> print d.num, d.pts, d.passe
total 271 54
>>>
```

7.2.4 La surcharge des opérateurs.

Il nous reste à voir la surcharge des opérateurs, en effet il serait plus simple de marquer :
>>> d = jack+bob+billy+mark+luc

Les méthodes spéciales nous permettent de faire cela avec leurs propriété de court-circuit les opérations normalles de Python.

```
def __add__(self, *other):
        x = self.pts
        y = self.passe

        for i in other:
                x = x+i.pts
                y = y+i.passe

        return Joueur("total", x, y)

def __repr__(self):

        return "numéro : %s points : %d passe : %d" % (self.num, self.pts, self.passe)
```

Nous avons rajouté deux méthodes : __add__ qui nous permet d'additionner des classes Joueur juste avec l'opérateur '+', quand à __repr__, elle nous permet lors de l'appel de la fonction print de renvoyer un formatage bien précis.

```
>>> d = jack + bob + billy + mark + luc
>>> print d
numéro : total points : 271 passe : 54
>>>
```

Voici une liste partielle des méthodes dites spéciales :

Méthodes	Surcharge	Appelée pour
__init__	Constructeur	Création d'objet : Classe()
__del__	Destructeur	Suppression d'objet
__add__	Opérateur ‘+'	X+Y
__or__	Opérateur 'l' (ou bit-à-bit)	X1Y
__repr__	Affichage, conversions	print X, `X`
__call__	Appels de fonctions	X()
__getattr__	Qualification	X indéfini
__getitem__	Indiçage	X[cle], boucles for, tests in
__setitem__	Affectation par indice	X[cle] = valeur
__getslice__	Découpage	X[bas:haut]
__len__	Longueur	len(X), tests
__cmp__	Comparaison	X =Y,X<Y
__radd__	Opérateur '+' de côté droit	Non instance + X

La programmation orientée objet n’est pas une chose simple et beaucoup de livres lui sont consacrés, mais l’évolution du programme de l’équipe de basket montre une bonne partie des capacités de cette façon de programmer.

8 Les exceptions

Table des matières :

8 Les exceptions 75
8.1 Les exceptions 76
8.1.1 Description des mots réservés : 76
8.1.1.1 try. 76
8.1.1.2 except. 76
8.1.1.3 else. 77
8.1.1.4 finally. 77
8.1.2 L'instruction raise. 77
8.1.3 Exceptions sous forme de classe. 77
8.1.4 L'instruction assert. 78
8.1.5 Exercice sur les exceptions No 1. 79
8.1.6 Exercice sur les exceptions No 2 80
8.1.7 Exercice sur les exceptions No 3 81
8.1.8 Exercice sur les exceptions No 4 82
8.1.9 Exercice sur les exceptions No 5. 83
8.1.10 Exercice sur les exceptions No 6. 84

8.1 Les exceptions.

Les exceptions font partie d'un mécanisme de contrôle du déroulement du programme. Les messages d'erreurs aperçus lors de l'exécution d'un programme viennent de la table d'exceptions par défaut de l'interpréteur Python. En effet, les exceptions non interceptées par une instruction "try" atteignent le sommet du processus Python, et exécute la logique par défaut des traitements d'exceptions.

Le traitement d'une exception permet de passer par dessus le traitement d'un bout de programme. La structure de gestion des exceptions est faite par blocs de code. Il y a deux façons principales de gérer les exceptions, l'une d'elle comprend les instructions "try", "except", et "else" :

```
try:
        <instruction>       exécute des actions.
except <nom>:
        <instruction>       si <nom> est déclenchée pendant le bloc try.
except <nom2>:
        <instruction>       si <nom2> est déclenchée pendant le bloc try.
...
else:
        <instruction>       si aucune exception n'est déclenchée.
```

L'autre comprend les instructions "try", et "finally" :

```
try:
        <instruction>       exécute des actions.
finally:
        <instruction>       exécuté de toute façon.
```

La clause "finally" ne peut pas être utilisé dans la même instruction "try" que "except" et "else", donc il vaut mieux se les représenter comme deux instructions différentes.

8.1.1 Description des mots réservés :

8.1.1.1 try.

"try" indique que la suite du programme sera surveillée par une gestion d'exception, quelle soit laissée par défaut ou qu'une nouvelle gestion soit implémentée.

8.1.1.2 except.

"except" permet la gestion d'une ou plusieurs exceptions. Le mot réservé "except" peut-être suivit de plusieurs arguments :

except: prend toutes les exceptions quelles que soit leurs types.
except <nom>: prend en compte l'exception <nom>.
except (<nom1>, <nom2>): Intercepte n'importe laquelles des exceptions listées.
except <nom>, <valeur>: intercepte l'exception <nom> avec sa donnée <valeur>.

8.1.1.3 else.

"else" permet d'exécuter la suite du programme au cas ou aucune exception ne serait survenue.

8.1.1.4 finally.

"finally" applique toujours le bloc, quoi qu'il se passe.

8.1.2 L'instruction raise.

Il est aussi possible en Python de lever des exceptions, ceci se fait par le mot réservé "raise". La forme générale reste simple : le mot raise suivi par le nom de l'exception à déclencher, et si nécessaire un argument à envoyer.

raise <nom> # déclenche une exception manuellement.
raise <nom>, <données> # idem mais avec des données.

Que peut lever "raise" comme nom d'exception ? "raise" peut appeler une exception intégrée a Python (IndexError), ou le nom d'un objet de type chaîne que vous avez défini dans votre programme. Il peut aussi se référencer à une classe à une instance.

8.1.3 Exceptions sous forme de classe.

Python a récemment généralisé la notion d'exception. Elles peuvent maintenant être identifiées par des classes et des instances de classes.
Avec l'ajout de cette notion d'exception "raise" peut prendre maintenant cinq formes différentes.

- raise chaine # comme except avec le même objet chaîne
- raise chaine, donnee # avec un donnée supplémentaire

- raise calsse, instance # except avec cette classe, ou classe parent
- raise instance # comme raise instance.__class__, instance
- raise # re-raise de l'exception courante

L'utilité est que, lorsque l'on gère une grande quantité d'exceptions, il n'est pas nécessaire de mettre de nom après "except", car le nom de l'exception levée sera traité dans la classe. De plus il est possible de rajouter des sous-classes afin de compléter un domaine d'exception et ceci sans toucher au code existant dans la classe parente.

8.1.4 L'instruction assert.

L'instruction "assert" permet de rajouter des contrôles pour le débogage d'un programme.
On peut dire que "assert" est essentiellement un raccourci syntaxique pour un "raise" compris dans la fonction __debug__. Il s'écrit de la forme :

```
assert <test>, <données>  # la partie données n'est pas obligatoire.
                          # "assert" lève toujours l'exception AssertionError !
```

Et cela est équivalant à :

```
if __debug__ :
        if not <test> :
                raise AssertionError, <données>
```

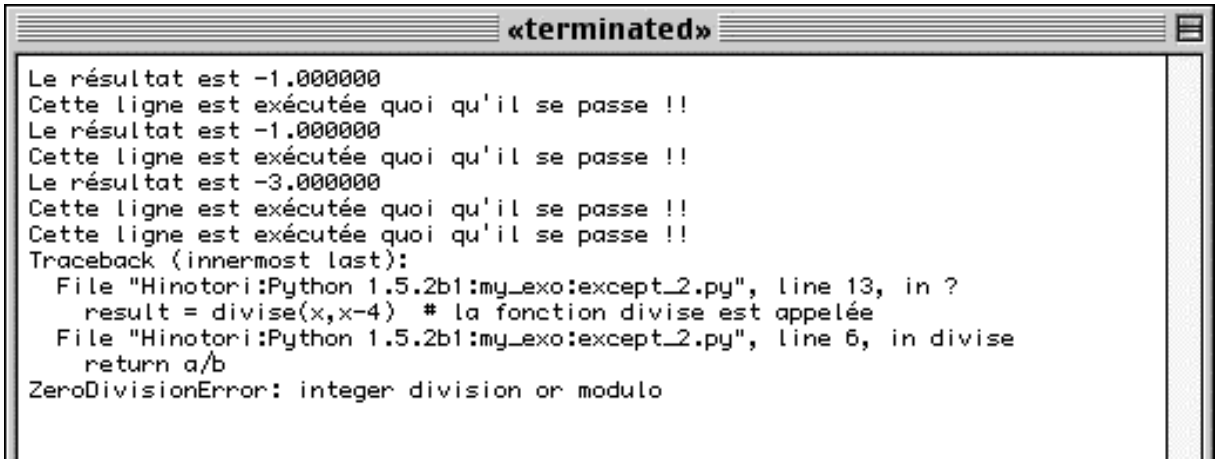
Les assertions peuvent être enlevées du byte code si l'option -O est appelée lors de la compilation. Les assertions sont principalement utilisées pendant le développement. Il est impératif d'éviter les effets de bord dans les assertions car, lors de leurs suppressions, l'effet est aussi enlevé.

8.1.5 Exercice sur les exceptions No 1.


```
# fichier : except_1.py by J.Tschanz
# fonction : montre une gestion simple d'exceptions

# notre fonction qui divise a par b
def divise (a, b):
        return a/b

for x in range(4.0):

# début du bloc de gestion des exceptions

        try:
                result = divise(x+2.0,x)        # la fonction divise est appelée
        except ZeroDivisionError:
                print "Quoi une division par zéro !!!" # si une division par zéro.
        except:
                print "Alors la je comprend pas !!" # si une autre erreur survient.
        else:
                print "Le résultat est %f " % result # si tout est OK.

# fin du fichier except_1.py
```

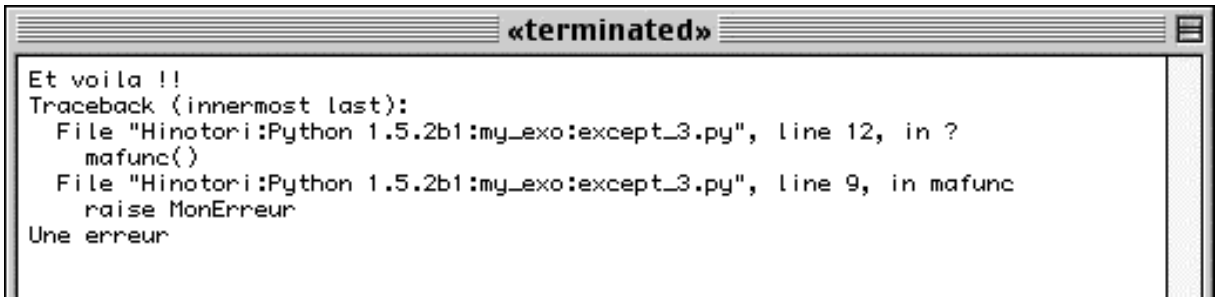
8.1.6 Exercice sur les exceptions No 2

```
# fichier : except_2.py by J.Tschanz
# fonction : montre une gestion simple d'exceptions

# notre fonction qui divise a par b
def divise (a, b):
        return a/b

for x in range(1.0, 5.0 ,1.0):

# début du bloc de gestion des exceptions

        try:
                result = divise(x,x-4)  # la fonction divise est appelée
                print "Le résultat est %f " % result # si tout est OK.
        finally:
                print "Cette ligne est exécutée quoi qu'il se passe !!"

# fin du fichier except_2.py
```



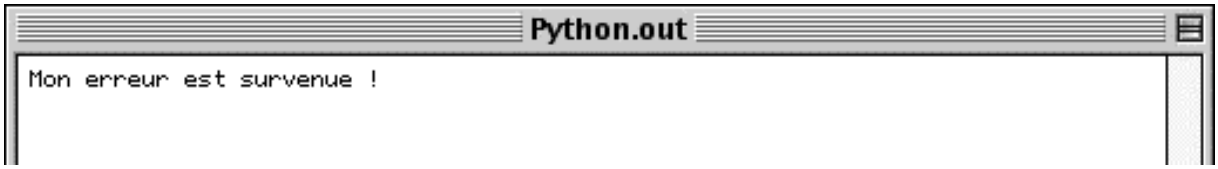
8.1.7 Exercice sur les exceptions No 3

```
# fichier : except_3.py by J.Tschanz
# fonction : montre une gestion simple d'exception utilisateur.

# définition de l'erreur utilisateur.
MonErreur = "Une erreur"

# function générant une exception.
def mafunc():
        raise MonErreur

try:
         mafunc()
finally:
        print "Et voila !!"

# fin du fichier except_3.py
```



8.1.8 Exercice sur les exceptions No 4

```
# fichier : except_4.py by J.Tschanz
# fonction : montre une gestion simple d'exception utilisateur.

# définition de l'erreur utilisateur.
MonErreur = "Une erreur"

# function générant une exception.
def mafunc():
        raise MonErreur, "survenue"

try:
 mafunc()
except MonErreur,  value:
        print "Mon erreur est %s !" % value
else:
        print "Et voila !!"

# fin du fichier except_4.py
```



8.1.9 Exercice sur les exceptions No 5.

```
# fichier : except_5.py by J.Tschanz
# fonction : montre un déclenchement d'exception.


# function générant une exception.
def mafunc():
        raise EOFError

try:
 mafunc()
except EOFError:
        print "Erreur d' I/O !!"
else:
        print "Et voila !!"

# fin du fichier except_5.py
```

8.1.10 Exercice sur les exceptions No 6.

```
# fichier : except_6.py by J.Tschanz
# fonction : montre les exceptions en sous-classe
import sys
# définition d'une erreur - chaîne de caractères
MonErreur = "MonErreur !"

# définition des class d'éxceptions
class General:
        pass

class Enfant(General):
        pass

# fonctions levant les d'exceptions
def except_1():
        raise MonErreur
def except_2():
        X = General()
        raise X
def except_3():
        X = Enfant()
        raise X
def except_4():
        raise Enfant()
def except_5():
        raise

for fonc in (except_1, except_2, except_3, except_4, except_5):
        try:
                fonc()
        except MonErreur:
                print "Intercepte : ", sys.exc_type
        except General:
                print "Intercepte : ", sys.exc_type

# fin du fichier except_6.py
```

9 __builtin__

Table des matières :

9 __builtin__....85
9.1 Les expressions et fonctions internes. 86
9.1.1 L'expression print. 86
9.1.2 L'expression input. 86
9.1.3 L'expression raw_input. 86
9.1.4 La fonction dir(x)....87
9.1.5 La fonction type(objet). 87
9.1.6 La fonction abs(nombre). 87
9.1.7 La fonction cmp(x, y)....87
9.1.8 Les fonction max(s), min(s)....88
9.1.9 La fonction apply(fonc, arg, …). 88
9.1.10 La fonction map(fonc, liste,…)....88
9.1.11 La fonction reduce(fonc, liste, …)....88
9.1.12 La fonction filter(fonc, liste, …). 89
9.1.13 La fonction eval(expression, …). 89
9.1.14 La fonction exec(instruction, …)....89
9.1.15 La fonction execfile(file, …). 90

9.1 Les expressions et fonctions internes.

Mis à part les types, les instructions de sélection et d'itération, Python comprend aussi plusieurs expressions et instructions permettant d'effectuer les opérations les plus courantes en un appel de méthode.

Il en existe beaucoup, et parfois elles sont très spécifiques. Nous allons en voir quelques unes afin de voir les possibilités de Python, et de mieux comprendre ce que nous avons déjà utilisé.

9.1.1 L'expression print.

Print est l'expression la plus courante, elle permet d'afficher à l'écran les informations qui la suivent.

```
print "Hello World" # Affichage d'une chaîne
print "Hello %s" % "world" # Affichage avec un argument
print len(liste) # Affichage du résultat d'une méthode
```

L'expression print suis la règle de formatage des chaînes. (voir chapitre 2)

9.1.2 L'expression input.

Input est une expression qui permet de poser une question à l'utilisateur, et d'en affecter la réponse à une variable.
Input ne supporte que les entiers et attend une entrée avant de continuer.

```
>>> var = input("un entier ? : ")
un entier ? : 4
>>> print var
4
```

9.1.3 L'expression raw_input.

Raw_input fonctionne comme input mais permet d'affecter n'importe quel type d'objet à la variable.

```
>>> var = raw_input("Yes/No ? : ")
Yes/No ? : Yes
>>> print var
Yes
```

9.1.4 La fonction dir(x).

La fonction dir permet d'énumérer les objets d'un module, les modules déjà chargés, les méthodes d'une classe, etc.

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'var']
>>> import sys
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys', 'var']
>>> dir(sys)
['__doc__', '__name__', '__stderr__', '__stdin__', '__stdout__', 'argv', 'builtin_module_names',
'copyright', 'exc_info', 'exc_type', 'exec_prefix', 'executable', 'exit', 'getrefcount', 'hexversion',
'maxint', 'modules', 'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile',
'settrace', 'stderr', 'stdin', 'stdout', 'version']
```

9.1.5 La fonction type(objet).

Type retourne le type de l'objet envoyé en argument.

```
>>> type(10)
<type 'int'>
>>> type("hello")
<type 'string'>
```

9.1.6 La fonction abs(nombre).

Abs retourne la valeur absolue du nombre envoyé en argument.

```
>>> abs(-8)
8
```

9.1.7 La fonction cmp(x, y).

Cmp compare deux arguments et retourne une valeur négative si le premier est plus petit que le deuxième, zéro s'ils sont égaux, et une valeur positive si le premier est plus grand.

```
>>> cmp(2,3)
-1
>>> cmp(2,2)
0
>>> cmp(3,2)
1
```

9.1.8 Les fonction max(s), min(s).

Ces deux fonction permettent de sortir l'objet de plus grande et plus petite valeur d'une séquence contenant au minimum un objet.

9.1.9 La fonction apply(fonc, arg, …).

Certains programmes ont besoin d'appeler des fonctions arbitrairement et ce, de manière générique, sans connaître leur nom ou argument. La forme "apply" appelle simplement la fonction passée en paramètre avec la liste d'arguments. La véritable utilité de "apply" est de pouvoir rendre l'appel de fonction différente à un même point du programme !

```
>>>if <test>:
>>>    action, argum = f_1, (1,)
>>>else:
>>>    action, argum = f_2, (2,3,4)
>>>
>>>apply(action, argum)
```

Dans cet exemple la fonction appelée change en fonction du test if !

9.1.10 La fonction map(fonc, liste,…).

Les programmes appliquent souvent une opération à chaque élément d'une liste. Python fournit une fonction interne qui effectue la plus grande partie du travail.
La fonction "map" applique une fonction passée en argument à chaque élément d'un objet séquence et retourne une liste contenant tous les résultats des appels.

```
>>> cp = [1,2,3,4]
>>> cp = map((lambda x, x+10), cp)
>>> cp
>>> [11,12,13,14]
```

9.1.11 La fonction reduce(fonc, liste, …).

La fonction "reduce" nécessite au minimum deux arguments car elle travaille de façon à avoir un résultat découlant d'un objet séquence.
La fonction reduce exécute la fonction une première fois avec les deux premiers arguments (d'une liste par exemple), ensuite elle prend le résultat comme premier argument et son deuxième argument sera l'élément suivant de la liste.
Il est possible de donner un paramètre de départ à la fonction reduce. Ce qui permettra de donner la valeur de départ pour une addition successive, par exemple.

```
>>> c = (0,1,2,3,4)
>>> y = reduce(lambda x,y:x+y, c)
>>> y
10
```

9.1.12 La fonction filter(fonc, liste, …).

"filter" permet de ressortir d'une liste, ou autre objet séquence, toutes les valeurs qui sont différentes de 0 après le passage dans une fonction. Il est donc, par exemple, possible de sortir les nombres impairs avec un modulo 2.

```
>>> c = [0,1,2,3,4]
>>> y = filter(lambda x: x%2, c)
>>> y
[1,3]
```

9.1.13 La fonction eval(expression, …).

"eval" permet d'exécuter une chaîne comme si'l s'agissait d'une expression. On peut évaluer une expression littérale, une simple expression mathématique, et même une expression "builtin".

```
>>> eval("'x'*4")
'xxxx'
>>> eval("len('world')")
5
```

9.1.14 La fonction exec(instruction, …).

"exec" ressemble à "eval", mais au lieu de prendre une chaîne comme une expression, exec l'exécute comme si'l s'agissait d'une instruction en python ! Ceci peut par exemple servir à faire des "import" dans une itération.

```
>>> s = ['sys', 'os', 'string', 'Tkinter']
>>> for i in s:
>>>    exec 'import'+' '+i
>>>
>>> dir()
['Tkinter', '__builtins__', '__doc__', '__name__', 'i', 'os', 's', 'string', 'sys']
```

9.1.15 La fonction execfile(file, …).

Execfile permet d'exécuter un programme contenu dans un fichier. C'est en un appel extérieur.

>>> execfile("hello.py")
"Hello World"

fichier hello.py :

print "Hello World"

10 Annexes.

Table des matières :

10 Annexes....91
10.1 Les mots réservé....92
10.2 Les outils de Python....93
10.3 Conclusion....94
10.4 Bibliographie....95
10.4.1 Livre....95
10.4.2 Internet....95

10.1 Les mots réservés.

and	**"et" logique**
assert	**insértion d'un test d'exception pour le débogage**
break	**instruction pour quitter une boucle**
class	**déclaration d'une classe**
continue	**instruction pour continuer une boucle**
def	**définition d'une fonction**
del	**supression d'un objet**
elif	**condition supplémentaire pour une sélection avec "if"**
else	**bloc optionnel d'exécution de code**
except	**définition d'une gestion d'exception**
exec	**execution d'une chaîne de caractères**
finally	**bloc final pour une gestion d'exceptions**
for	**commande d'itération**
from	**appel d'objet pour une importation**
global	**déclaration pour une variable visible de partout**
if	**commande de séléction**
import	**importation de module**
in	**recherche d'un objet cible dans un objet séquence**
is	**comparaison**
lambda	**expression de déclaration d'une fonction simple**
not	**"non" logique**
or	**"ou" logique**
pass	**expression équivalent à ne rien faire**
print	**commande d'affichage**
raise	**levée d'exception**
return	**expression pour le retour depuis une fonction**
try	**début de la gestion d'exception**
while	**commande d'itération**

10.2 Les outils de Python.

Python possède par défaut un grande quantité de librairies. Nous allons en énumérer quelques-unes :

cgi	**librairie pour le traitement de cgi**
ftplib	**librairie en rapport avec le service ftp**
gdbm	**création de base de données par dictionnaire**
math	**librairie mathématique**
os	**fonction et commande par rapport à l'OS**
poplib	**gestion de courriers éléctroniques**
random	**gestion de choix aléatoires**
re / regex	**librairie d'expressions régulières**
shelve	**gestion de base de données**
smtplib	**gestion de courrier éléctronique**
socket	**librairie concernant les sockets**
string	**gestion et transformation par rapport aux chaînes de caractères**
sys	**librairie contenant des fonctions Python en rapport avec le système**
telnet	**fonctions telnet pour Python**
time	**librairie concernant le temps, et la date**
Tkinter	**création d'un environnemet graphique**

10.3 Conclusion.

Le langage Python est en pleine évolution, et de plus en plus de grosses entreprises l'utilisent. Sa librairie très développée et la simplicité de sa syntaxe lui permettent d'être un langage très performant dans le domaine des applications ou la vitesse n'est pas la plus importante, l'apprentissage d'un langage orienté objet, et pour des applications devant passer d'un OS à l'autre.

Le futur de Python est déjà assuré par une nouvelle version 2.0 qui reprend exactement la même syntaxe, et les mêmes propriétés. Une version faite sur la base de Java (JPython) existe aussi, en plus d'être compatible avec Python, il est en plus possible avec cette version de charger des librairies Java.

Pour ma part, j'espère que ce tutorial aura permis de donner une approche positive de ce langage à typage dynamique. J'ai, en tous les cas eu beaucoup de plaisir à travailler avec ce langage et de l'essayer sur plusieurs OS (Linux, MacOS 9, Win 95, Win 98) différents afin de tester la portabilité de ce langage, et je n'ai jamais été déçus.

Ce projet m'a aussi permis de mieux comprendre la programmation orientée objet, et de développer de petits utilitaires très pratiques.

Je vais continuer à étudier ce langage dans le futur, et ceci du coté des possibilités d'utiliser Python en rapport avec l'OS. Car bien que Python soit portable et existe sur beaucoup d'OS il est très intéressant de l'étudier sur une plate-forme Linux ou Unix, car elles permettent bien plus de possibilités qu'une autre plate-forme.

Jérôme Tschanz
j.tschanz@romandie.com

2 octobre 2000

10.4 Bibliographie.

10.4.1 Livre.

- "Comparaison of C, C++, Java, Perl, Python,..." Lutz Prechelt (Internet).
- "Introduction à Python" Mark Lutz & David Ascher (O'Reilly).
- "Python and Tkinter" Fredrik Lundh (Internet).
- "The Standard Python Library" Fredrik Lundh (e-matter).
- "Linux France Magazine" numéros 17, 18, 19, 20
- "Python Reference Manual" Guido Van Rossum (Open Docs Library)
- "Python Pocket Reference" Mark Lutz (O'Reilly)
- "Python and Tkinter" John E. Grayson (Manning Publication)

10.4.2 Internet.

- www.python.org Tout sur Python
- www.linux-center.org/articles/9812/python.html Introduction à Python

Index

	Chap.
% l'opérateur	2
**arg	5
*arg	5
\ l'opérateur	2
__builtin__	9
abs	9
affectation	3
and	2
apply	9
arguments de fonction	5
assert	8
bloc de code	1
break	4
chaîne	2
class	7
clés (dico.)	2
cmp	9
commentaire	1
complexes	2
continue	4
def	5
dictionnaire	2
dir	9
elif	4
else (if)	4
else (for)	4
else (while)	4
else (exception)	8
entiers	2

eval	**9**
except	**8**
exception	**8**
exec	**9**
execfile	**9**
extraction	**2**
fichier	**2**
filter	**9**
fonction	**5**
for	**4**
formatage de chaîne	**2**
from	**6**
global	**3**
héritage (classe)	**7**
if	**4**
import	**6**
indentation	**1**
indiçage	**2**
input	**9**
instance	**7**
instruction	**4**
invité interactif	**1**
lambda	**5**
listes	**2**
map	**9**
max	**9**
membre (classe)	**7**
méthode	**7**
min	**9**
module	**6**
noms de variables	**3**
or	**2**
pass	**4**

print	**9**
raise	**8**
range	**4**
raw_input	**9**
reduce	**9**
réels	**2**
reload	**6**
return	**5**
return (classe)	**7**
self (classe)	**7**
surcharge des opérateurs	**7**
sys.path	**6**
try	**8**
tuple	**2**
type	**9**
variable	**3**
while	**4**
xrange	**4**