Tutoriel Python

Release 2.4.1

Guido van Rossum
Fred L. Drake, Jr., editor

24 septembre 2005
Traduction française dirigée par Olivier Berger
Mise à jour par Henri Garreta

Python Software Foundation
E-mail: docs@python.org

Résumé

Python est un langage de programmation facile à utiliser et puissant. Il offre des structures de données puissantes de haut niveau et une approche simple mais réelle de la programmation orientée-objet. La syntaxe élégante de python et le typage dynamique, ajoutés à sa nature interprétée, en font un langage idéal pour écrire des scripts et pour le développement rapide d'applications dans de nombreux domaines et sur la plupart des plates-formes.

L'interpréteur python et la vaste librairie standard sont librement disponible pour toutes les plates-formes principales sous forme de sources ou de binaires à partir du site Web de Python, http ://www.python.org/, et peuvent être distribués librement. Le même site contient aussi des distributions et des pointeurs vers de nombreux modules Python provenant d'autres fournisseurs, des programmes et des outils, et de la documentation supplémentaire.

L'interpréteur Python est facilement extensible par de nouvelles fonctions et de nouveaux types de données implémentés en C ou en C++ (ou d'autres langages appelables depuis le C). Python convient également comme langage d'extension pour des logiciels configurables.

Ce tutoriel introduit le lecteur de façon informelle aux concepts et caractéristiques de base du langage et du système Python. Il est utile d'avoir un interpréteur Python disponible pour expérimenter directement, mais tous les exemples sont auto-porteurs, donc le tutoriel peut également être lu sans interpréteur sous la main.

Pour une description des objets et modules standards, voir le document *Python Library Reference*. Le *Python Reference Manual* donne une définition plus formelle du langage. Pour écrire des extensions en C ou C++, lire les manuels *Extending and Embedding* et *Python/C API Reference*. Il existe aussi plusieurs livres décrivant Python en profondeur.

Ce tutoriel n'essaye pas d'être complet et de traiter chaque possibilité, ou même toutes les caractéristiques utilisées couramment. A la place, il présente bon nombre des caractéristiques les plus remarquables de Python, et vous donnera une bonne idée de la "couleur" et du style du langage. Après l'avoir lu, vous serez capable de lire et d'écrire des programmes ou des modules en Python, et vous serez prêts à en apprendre plus sur les différents modules de bibliothèques Python décrits dans le *Python Library Reference*.

TABLE DES MATIÈRES

CHAPITRE

UN

Pour vous mettre en appétit

Si vous avez jamais écrit un long script shell, vous connaissez probablement ce sentiment : vous aimeriez ajouter encore une autre fonctionnalité, mais il est déjà tellement lent, et tellement gros, et si compliqué ; ou bien la fonctionnalité requiert un appel système ou une autre fonction qui est accessible seulement en C… Habituellement, le problème à résoudre n'est pas suffisamment grave pour justifier une réécriture du script en C ; peut-être que le problème requiert des chaînes de caractères de longueur variable ou d'autres types de données (comme des listes triées de noms de fichiers) qui sont faciles à faire en shell mais nécessitent beaucoup d'effort d'implémentation en C, ou peut-être n'êtes-vous pas suffisamment familier avec le C.

Une autre situation : peut-être devez-vous travailler avec plusieurs bibliothèques C, et le cycle habituel en C écrire, compiler, tester, re-compiler est trop lent. Vous avez besoin de développer du logiciel plus rapidement. Ou alors, peut-être avez-vous écrit un programme qui pourrait utiliser un langage d'extension, et vous ne voulez pas définir un langage, écrire et mettre au point un interpréteur pour lui, puis le lier à votre application.

Dans toutes ces situations, Python pourrait être le langage qu'il vous faut. Python est simple d'emploi, mais c'est un vrai langage de programmation, qui offre bien plus de structure et de de possibilités que le shell pour des programmes volumineux. D'autre part, il offre également beaucoup plus de vérification d'erreurs que le C, et, étant un *langage de très haut niveau*, il contient des types de données de haut niveau intégrés, comme des tableaux redimensionnables et des dictionnaires, qui vous demanderaient des jours à implémenter efficacement en C. Grâce à ses types de données plus généraux, Python est applicable à un domaine de problèmes beaucoup plus large que *Awk* ou même *Perl*, et de nombreuses choses sont au moins aussi faciles en Python que dans ces langages.

Python vous permet de séparer vos programmes en modules qui peuvent être réutilisés dans d'autres programmes en Python. Il est fourni avec une vaste collection de modules standard que vous pouvez utiliser comme base pour vos programmes — ou comme exemples pour s'initier à la programmation en Python. Il y a aussi des modules intégrés qui fournissent des fonctionnalités comme les entrées/sorties vers les fichiers, les appels systèmes, les sockets, et même des interfaces avec les toolkits d'Interface Homme Machine comme Tk.

Python est un langage interprété, ce qui peut vous faire gagner un temps considérable pendant la réalisation de programmes car aucune compilation ou édition de liens n'est nécessaire. L'interpréteur peut être utilisé de façon interactive, ce qui facilite l'expérimentation avec les possibilités du langage, l'écriture de programmes jetables, ou le test de fonctions pendant le développement ascendant de vos logiciels. C'est aussi une calculatrice de bureau assez pratique.

Python permet d'écrire des programmes très compacts et lisibles. Les programmes écrits en Python sont typiquement beaucoup plus courts que leurs équivalents en C, pour plusieurs raisons :

- les types de données de haut niveau vous permettent de réaliser des opérations complexes en une seule instruction ;
- le regroupement des instructions se fait par indentation, sans accolades de début/fin ;
- il n'est pas nécessaire de déclarer les variables ou les arguments.

Python est *extensible* : si vous savez programmer en C, il est facile d'ajouter une nouvelle fonction intégrée ou un module dans l'interpréteur, soit pour réaliser les opérations critiques à vitesse maximum, soit pour linker les programmes en Python à des bibliothèques qui ne sont disponibles que sous forme binaire (comme une bibliothèque graphique propriétaire). Une fois que vous serez accro, vous pourrez linker l'interpréteur Python dans votre application écrite en C et l'utiliser comme langage d'extension ou de commande pour cette application.

Au passage, le langage est nommé d'après l'émission de la BBC "Monty Python's Flying Circus" et n'a rien à voir avec de vilains reptiles. Faire référence à des dialogues des Monty Python dans la documentation n'est pas seulement autorisé, c'est encouragé !

Maintenant que vous êtes tout excité par Python, vous voudrez l'examiner plus en détail. Puisque la meilleure façon d'apprendre un langage est de l'utiliser, vous êtes invité à le faire dès maintenant.

Dans le chapitre suivant, les éléments nécessaires à l'utilisation de l'interpréteur sont expliqués. Ce sont des informations plutôt rébarbatives, mais essentielles pour pouvoir tester les exemples donnés plus loin.

Le reste de ce tutoriel introduit les différentes caractéristiques du langage et du système Python à travers des exemples, en commençant par les expressions, instructions et types de données simples, puis les fonctions et modules, et enfin en survolant des concepts avancés comme les exceptions et les classes définies par l'utilisateur.

CHAPITRE

DEUX

Utilisation de l'interpréteur Python

2.1 Lancement de l'interpréteur

L'interpréteur Python est habituellement installé à l'emplacement '/usr/local/bin/python' sur les machines Unix sur lesquelles il est disponible ; placer '/usr/local/bin' dans le path de votre shell UNIX permet de le lancer en tapant la commande

```
python
```

sous le shell. Puisque le choix du répertoire dans lequel est installé l'interpréteur est une option d'installation, d'autres endroits sont possibles ; vérifiez avec votre gourou Python local ou votre administrateur système. (Par exemple, '/usr/local/python' est un autre emplacement populaire.)

Tapez le caractère de fin-de-fichier (`Control-D` sur UNIX, `Control-Z` sur DOS ou Windows) à l'invite (prompt) principale pour quitter l'interpréteur avec un code de retour de zéro. Si ça ne marche pas, vous pouvez quitter l'interpréteur en tapant les commandes suivantes : '`import sys ; sys.exit()`'.

Les fonctions d'édition de ligne de l'interpréteur ne sont habituellement pas très sophistiquées. Sur UNIX, celui qui a installé l'interpréteur peut avoir activé le support de la bibliothèque GNU Readline, qui ajoute une édition interactive plus élaborée et des fonctions d'historique. Peut-être la vérification la plus rapide pour savoir si l'édition de ligne de commande est supportée consiste à taper Control-P au premier prompt affiché par Python. Si ça fait un bip, vous disposez de l'édition de ligne de commande ; voir l'Annexe A pour une introduction aux touches. Si rien ne semble se passer, ou si `^P` est affiché, l'édition de ligne de commande n'est pas disponible ; vous pourrez seulement utiliser backspace pour enlever des caractères de la ligne courante.

L'interpréteur fonctionne en quelque sorte comme le shell UNIX : lorsqu'il est lancé avec l'entrée standard connectée à un périphérique tty, il lit et exécute les commandes interactivement ; lorsqu'il est lancé avec un nom de fichier en argument ou avec un fichier comme entrée standard, il lit et exécute un *script* depuis ce fichier.

Une deuxième façon de lancer l'interpréteur est '`python -c commande [arg] ...`', ce qui exécute les instructions dans `commande`, de façon analogue à l'option `-c` du shell. Puisque les instructions en Python contiennent souvent des espaces ou d'autres caractères qui sont spéciaux pour le shell, il est conseillé de mettre entièrement entre guillemets (doubles quotes) la `commande`.

Notez qu'il y a une différence entre '`python fichier`' et '`python <fichier`'. Dans le second cas, les requêtes en entrée du programme, comme les appels à `input()` et `raw_input()`, sont effectuées depuis *fichier*. Puisque ce fichier a déjà été lu jusqu'à la fin par l'interpréteur avant que le programme ne commence à s'exécuter, le programme rencontrera immédiatement une fin-de-fichier. Dans le premier cas (qui est habituellement ce qu'on souhaite) les requêtes sont satisfaites par l'éventuel fichier ou périphérique qui est connecté à l'entrée standard de l'interpréteur Python.

Lorsqu'un fichier script est utilisé, il est parfois utile de pouvoir lancer le script et de passer ensuite en mode interactif. Cela peut être fait en passant en paramètre `-i` avant le script. (Cela ne marche pas si le script est lu depuis l'entrée standard, pour la même raison expliquée dans le paragraphe précédent.)

2.1.1 Passage de paramètres

Lorsqu'ils sont connus de l'interpréteur, le nom du script et les paramètres supplémentaires sont passés au script dans la variable `sys.argv`, qui est une liste de chaînes. Sa longueur est d'au moins un ; quand aucun script ou arguments n'est donné, `sys.argv[0]` est une chaîne vide. Lorsque le nom du script donné est `'-'` (c'est à dire l'entrée standard), `sys.argv[0]` est positionné à `'-'`. Quand `-c commande` est utilisé, `sys.argv[0]` est positionné à `'-c'`. Les options trouvées après `-c commande` ne sont pas retirées par le traitement des options de l'interpréteur Python mais laissées dans `sys.argv` pour être traitées par la `commande`.

2.1.2 Mode interactif

Quand les commandes sont lues depuis un terminal, l'interpréteur fonctionne en *mode interactif*. Dans ce mode, il demande la commande suivante avec le *prompt principal*, habituellement trois signes supérieur `>>>` ; pour les lignes de continuation, il questionne avec le *prompt secondaire*, par défaut trois points (`... `). L'interpréteur imprime un message de bienvenue spécifiant son numéro de version et une notice de copyright avant d'afficher le prompt principal :

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Les lignes de continuation sont nécessaires lorsqu'on saisit une construction sur plusieurs lignes. Comme exemple, voici une instruction `if` :

```
>>> le_monde_est_plat = 1
>>> if le_monde_est_plat:
...     print "Gaffe à pas tomber par dessus bord!"
...
Gaffe à pas tomber par dessus bord!
```

2.2 L'interpréteur et son environnement

2.2.1 Gestion des erreurs

Quand une erreur survient, l'interpréteur imprime un message d'erreur et une trace de l'état de la pile. En mode interactif, il retourne alors au prompt principal ; lorsque l'entrée se fait depuis un fichier, il termine l'exécution avec un code de sortie différent de zéro après avoir imprimé la trace de pile. (Les exceptions gérées par une clause `except` dans une instruction `try` ne sont pas des erreurs dans ce contexte ; voir 8.2.) Certaines erreurs sont fatales dans tous les cas et causent une sortie avec un code différent de zéro ; cela se produit pour les aberrations internes et certains cas de saturation mémoire. Tous les messages d'erreur sont écrits sur la sortie d'erreur standard ; l'affichage normal des commandes exécutées est effectué sur la sortie standard.

Taper le caractère d'interruption (habituellement Control-C ou DEL) au prompt principal ou secondaire annule la saisie et revient au prompt principal.[1] Taper le caractère d'interruption pendant l'exécution d'une commande déclenche l'exception `KeyboardInterrupt`, qui peut être gérée par une instruction `try`.

2.2.2 Scripts Python exécutables

Sur les systèmes UNIX à la BSD, les scripts Python peuvent être rendus directement exécutables, comme les scripts shell, en plaçant la ligne

[1] Un problème avec la bibliothèque GNU Readline peut empêcher cela.

```
#! /usr/bin/env python
```

(en supposant que l'interpréteur est dans le PATH de l'utilisateur au lancement du script) et rendant le fichier exécutable. Le '#!' doit correspondre aux deux premiers caractères du fichier.

2.2.3 Encodage du fichier source

Dans les fichiers sources Python il est possible d'utiliser des encodages différents de l'ASCII. La meilleure façon de le faire consiste à mettre une ligne de commentaire spéciale après la ligne `#!` pour définir l'encodage du fichier source :

```
# -*- coding: encoding -*-
```

Avec cette déclaration, tous les caractères dans le fichier source seront traités comme ayant l'encodage *encoding*, et il sera possibe d'écrire directement des chaînes littérales Unicode dans l'encodage choisi. La liste des encodages possibles peut être trouvée dans *Python Library Reference*, dans la section sur les `codecs`.

Par exemple, pour écrire des chaînes littérales Unicode incluant le symbole monétaire Euro, l'encodage ISO-8859-15 peut être utilisé, le symbole de Euro ayant le valeur ordinale 164. Le script suivant imprime la valeur 8364 (le code Unicode correspondant au symbole Euro) puis termine :

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

Si votre éditeur est capable d'enregistrer des fichiers en UTF-8 avec une marque *UTF-8 byte order mark* (ou BOM), vous pouvez utiliser cela au lieu d'une déclaration d'encodage. IDLE accepte cette possibilité si la case `Options/General/Default Source Encoding/UTF-8` est cochée. Notez que cette signature n'est pas comprise dans les anciennes livraisons de Python (2.2 et plus anciennes), et n'est pas non plus comprise par le système d'exploitation pour les fichiers script avec des lignes `#!` (utilisées uniquement sur des systèmes UNIX).

En utilisant UTF-8 (soit à travers cette signature, soit au moyen d'une déclaration d'encodage), les caractères de la plupart des langues du monde peuvent figurer simultanément dans les chaînes littérales et les commentaires. L'emploi de caractères non ASCII dans les identificateurs n'est pas autorisé. Pour afficher tous ces caractères correctement votre éditeur doit reconnaître que le fichier est UTF-8 et il doit utiliser une police de caractères qui supporte tous les caractères dans le fichier.

2.2.4 Le fichier de démarrage interactif

Quand vous utilisez Python de façon interactive, il est souvent pratique que certaines commandes standard soient exécutées à chaque fois que l'interpréteur est lancé. Vous pouvez faire cela en donnant à la variable d'environnement PYTHONSTARTUP la valeur du nom d'un fichier contenant vos commandes de démarrage. Cela est analogue à l'utilisation du '.profile' pour les shells UNIX.

Ce fichier est seulement lu avant les sessions interactives, non pas lorsque Python lit ses commandes dans un script, ni lorsque '/dev/tty' est fourni comme source explicite pour les commandes (ce qui pour le reste se comporte comme une session interactive). Il est exécuté dans le même espace de noms que celui où sont exécutées les commandes interactives, de sorte que les objets qu'il définit ou importe peuvent être utilisés directement dans la session interactive. Vous pouvez aussi changer les invites `sys.ps1` et `sys.ps2` dans ce fichier.

Si vous voulez lire un fichier de lancement additionnel depuis le répertoire courant, vous pouvez programmer cela dans le fichier de démarrage global en utilisant du code similaire à '`if os.path.isfile('.pythonrc.py') : execfile('.pythonrc.py')`'. Si vous voulez utiliser le fichier de démarrage dans un script, vous devez le faire explicitement dans le script :

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

CHAPITRE

TROIS

Une introduction informelle à Python

Dans les exemples suivants, la saisie et l'affichage seront distingués par la présence ou l'absence d'invites (`>>>` et '`...` ') : pour reproduire l'exemple, vous devez taper tout ce qui suit l'invite, quand celle-ci apparaît ; les lignes qui ne commencent pas par une invite correspondent à l'affichage effectué par l'interpréteur. Notez qu'une invite secondaire seule sur une ligne signifie que vous devez taper une ligne vide ; cela marque la fin des commandes sur plusieurs lignes.

De nombreux exemples de ce manuel, même ceux qui sont saisis à l'invite interactive, contiennent des commentaires. Les commentaires en Python commencent par un caractère dièse, '#', et continuent jusqu'à la fin de la ligne physique. Un commentaire peut se trouver au début d'une ligne derrière un espace ou du code, mais pas à l'intérieur d'une chaîne de caractères litérale. Un caractère dièse à l'intérieur d'une chaîne est juste un caractère dièse.

Quelques exemples :

```
# voici le premier commentaire
SPAM = 1                 # et voici le deuxième commentaire
                         # ... et maintenant un troisième!
STRING = "# Ceci n'est pas un commentaire."
```

3.1 Utiliser Python comme calculatrice

Essayons quelques commandes Python simples. Lancez l'interpréteur et attendez l'apparition du prompt principal, `>>>`. (Ça ne devrait pas être très long.)

3.1.1 Nombres

L'interpréteur fonctionne comme une simple calculatrice : vous pouvez y taper une expression et il va en afficher la valeur. La syntaxe des expression est naturelle : les opérateurs `+`, `-`, `*` et `/` marchent exactement comme dans la plupart des langages (par exemple Pascal ou C) ; les parenthèses peuvent être utilisées pour les regrouper. Par exemple :

```
>>> 2+2
4
>>> # Ceci est un commentaire
... 2+2
4
>>> 2+2  # et un commentaire sur la même ligne que le code
4
>>> (50-5*6)/4
5
>>> # La division des entiers retourne l'entier immédiatement inférieur:
... 7/3
2
>>> 7/-3
-3
```

Comme en C, le signe égale (‘=’) est utilisé pour affecter une valeur à une variable. La valeur d’une affectation n’est pas affichée :

```
>>> largeur = 20
>>> hauteur = 5*9
>>> largeur * hauteur
900
```

Une valeur peut être affectée à plusieurs variables simultanément :

```
>>> x = y = z = 0  # Mettre à zéro x, y et z
>>> x
0
>>> y
0
>>> z
0
```

Il y a un support complet des nombres à virgule flottante ; les opérateurs en présence de types d’opérandes mélangés convertissent les opérandes entiers en virgule flottante :

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Les nombres complexes sont toujours représentés comme deux nombres en virgule flottante, les parties réelle et imaginaire. Pour extraire ces parties d’un nombre complexe *z*, utilisez `z.real` et `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Les fonctions de conversion en virgule flottante et en entier (`float()`, `int()` et `long()`) ne marchent pas pour les nombres complexes — il n'y a pas une façon correcte et unique de convertir un nombre complexe en un nombre réel. Utilisez `abs(`*z*`)` pour obtenir sa norme (en flottant) ou `z.real` pour sa partie réelle.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

En mode interactif, la dernière expression affichée est affectée à la variable _. Quand vous voulez utiliser Python comme calculatrice, c'est plus pratique pour continuer les calculs, par exemple :

```
>>> tva = 12.5 / 100
>>> prix = 100.50
>>> prix * tva
12.5625
>>> prix + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Cette variable doit être utilisée en lecture seule par l'utilisateur. Ne lui affectez pas une valeur de façon explicite — vous auriez alors créé une variable locale indépendante, avec le même nom, masquant la variable intégrée et son comportement magique.

3.1.2 Chaînes de caractères

En plus des nombres, Python peut aussi manipuler des chaînes, qui peuvent être exprimées de différentes façons. Elles peuvent être incluses entre simples quotes (apostrophes) ou doubles quotes (guillemets) :

```
>>> 'spam eggs'
'spam eggs'
>>> 'n\'est-ce pas'
"n'est-ce pas"
>>> "n'est-ce pas"
"n'est-ce pas"
>>> '"Oui," dit-il.'
'"Oui," dit-il.'
>>> "\"Oui,\" dit-il."
'"Oui," dit-il.'
>>> '"N\'est-ce pas," repondit-elle.'
'"N\'est-ce pas," repondit-elle.'
```

Notez[1] que les chaînes admettent ou non les caractères accentués en mode intéractif suivant votre plate-forme. Si les commandes sont lues depuis un fichier, la situation est légèrement différente : en général vous pourrez, mais les caractères accentués risquent d'être interprétés différemment si vous transférez vos fichiers entre des plate-formes différentes. Pour ces questions de portabilité, les identificateurs en Python sont limités au code ASCII 7 bits. Vous ne pourrez pas (en mode intéractif ou pas) utiliser des lettres accentuées dans les noms de variables, fonctions, modules, classes, etc.

Les textes dans les chaînes peuvent se poursuivre sur plusieurs lignes de plusieurs façons. Des lignes de continuation peuvent être utilisées, avec un antislash comme dernier caractère sur la ligne pour indiquer que la prochaine ligne est une continuation logique de la ligne :

```
salut = "Ceci est une chaîne plutot longue contenant\n\
plusieurs lignes de texte excatement comme on le ferait en C.\n\
    Notez que les blancs au début de la ligne sont\
 significatifs."

print salut
```

Notez que les retours chariot nécessiteraient toujours d'être intégrés dans la chaîne en utilisant `\n` ; le retour chariot qui suit l'antislash de fin est supprimé. Cet exemple s'afficherait de la façon suivante :

```
Ceci est une chaîne plutot longue contenant
plusieurs lignes de texte excatement comme on le ferait en C.
    Notez que les blancs au début de la ligne sont significatifs.
```

Si nous définissons les caractères de la chaîne comme une chaîne "raw" (brute), par contre, les séquences `\n` ne sont pas converties en caractères de retour chariot, mais l'antislash de fin de ligne, et le retour chariot du source sont tous les deux inclus dans la chaîne comme données. Ainsi, l'exemple :

```
salut = r"Ceci est une chaine assez longue contenant\n\
plusieurs lignes de texte comme vous pourriez vraiment le faire en C."

print salut
```

afficherait :

```
Ceci est une chaine assez longue contenant\n\
plusieurs lignes de texte comme vous pourriez vraiment le faire en C.
```

Ou bien, les chaînes peuvent être entourées par un couple de triple-quotes correspondantes : `"""` ou `'''`. Les fins de lignes n'ont pas besoin d'être préfixées lorsqu'on utilise les triple-quotes, mais elles seront incluses dans la chaîne.

```
print """
Usage: trucmuche [OPTIONS]
     -h                        Affiche cette notice d'usage
     -H hôte                   hôte auquel il faut se connecter
"""
```

produit l'affichage suivant :

[1] NDT : Ce paragraphe absent de l'édition originale a été ajouté par Daniel Calvelo Aros à l'intention des utilisateurs de Python francophones.

```
Usage: trucmuche [OPTIONS]
     -h                        Affiche cette notice d'usage
     -H hôte                   hôte auquel il faut se connecter
```

L'interpréteur affiche le résultat des opérations sur les chaînes de la même façon qu'à la saisie : entre quotes, et avec quotes et autres caractères bizarres préfixés par un antislash, pour afficher leur valeur exacte. La chaîne sera délimitée par des doubles quotes si elle contient une simple quote et aucune double quote, sinon, elle sera délimitée par des simples quotes. (L'instruction `print`, décrite plus loin, peut être utilisée pour écrire des chaînes sans quotes ni caractères préfixés.)

Les chaînes peuvent être concaténées (accolées) avec l'opérateur `+`, et répétées avec `*` :

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Deux chaînes de texte côte à côte sont automatiquement concaténées ; la première ligne ci-dessus aurait pu être écrite '`mot = 'Help' 'A''`; cela fonctionne seulement avec deux chaînes de texte, pas avec des expressions quelconques de type chaîne.

```
>>> import string
>>> 'cha' 'ine'                   #  <-  C'est ok
'chaine'
>>> string.strip('cha') + 'ine'   #  <-  C'est ok
'chaine'
>>> string.strip('cha') 'ine'     #  <-  Ca c'est faux
  File "<stdin>", line 1, in ?
    string.strip('cha') 'ine'
                        ^
SyntaxError: invalid syntax
```

Les chaînes peuvent être décomposées (indexées) ; comme en C, le premier caractère d'une chaîne est en position (index) 0. Il n'y a pas de type caractère spécifique ; un caractère est simplement une chaîne de taille un. Comme en Icon, les sous-chaînes peuvent être spécifiées avec la *notation de découpage* (*slice*) : deux indices séparés par deux-points.

```
>>> mot[4]
'A'
>>> mot[0:2]
'He'
>>> mot[2:4]
'lp'
```

A la différence des chaînes de caractères en C, les chaînes Python ne peuvent être modifiées. Faire une affectation à l'emplacement d'un indice dans la chaîne aboutit à une erreur :

```
>>> mot[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> mot[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Cependant, il est facile et efficace de créer une nouvelle chaîne avec un contenu combiné :

```
>>> 'x' + mot[1:]
'xelpA'
>>> 'Splat' + mot[4]
'SplatA'
```

Les indices de découpage ont des valeurs par défaut utiles ; un premier index non-défini prend pour valeur par défaut zéro, un second index omis prend pour valeur par défaut la taille de la chaîne qu'on est en train de découper.

```
>>> mot[:2]    # Les deux premiers caractères
'He'
>>> mot[2:]    # Tout sauf les deux premiers caractères
'lpA'
```

Voici un invariant utile des opérations de découpage : `s[ :i] + s[i :]` égale `s`.

```
>>> mot[:2] + mot[2:]
'HelpA'
>>> mot[:3] + mot[3:]
'HelpA'
```

Les indices de découpage erronés sont gérés de façon élégante : un index qui est trop grand est remplacé par la taille de la chaîne, un index de fin inférieur à l'indice de début retourne une chaîne vide.

```
>>> mot[1:100]
'elpA'
>>> mot[10:]
''
>>> mot[2:1]
''
```

Les indices peuvent être des nombres négatifs, pour compter à partir de la droite. Par exemple :

```
>>> mot[-1]     # Le dernier caractère
'A'
>>> mot[-2]     # L'avant dernier caractère
'p'
>>> mot[-2:]    # Les deux derniers caractères
'pA'
>>> mot[:-2]    # Tout sauf les deux derniers caractères
'Hel'
```

Mais notez que -0 est vraiment la même chose que 0, donc ça ne compte pas à partir de la droite !

```
>>> mot[-0]     # (puisque -0 égale 0)
'H'
```

Les indices de découpage négatifs hors limites sont tronqués, mais n'essayez pas ceci avec des indices d'accès à des éléments uniques (sans découpage) :

```
>>> mot[-100:]
'HelpA'
>>> mot[-10]    # erreur
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

La meilleure façon de se rappeler comment marchent les découpages est de penser aux indices comme pointant *entre* les caractères, avec le bord gauche du premier caractère numéroté 0. Alors le bord droit du dernier caractère d'une chaîne de *n* caractères porte l'index *n*, par exemple :

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

La première ligne de nombres donne la position des indices 0...5 dans la chaîne ; la seconde ligne donne les indice négatifs correspondants. Le découpage de *i* à *j* consiste en tous les caractères entre les extrémités étiquetées *i* et *j*, respectivement.

Pour les indices non négatifs, la longueur d'une tranche est la différence entre ses indices, si les deux sont à l'intérieur des limites. Par exemple, la longueur de `mot[1:3]` est 2.

La fonction intégrée `len()` retourne la longueur d'une chaîne :

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Voir aussi :

Types séquences (`http://docs.python.org/lib/typesseq.html`)

Les chaînes et les chaînes Unicode décrites à la prochaine section sont des exemples de types séquences et supportent les opérations communes supportées par de tels types.

Méthodes des chaînes (`http://docs.python.org/lib/string-methods.html`)

Les chaînes et les chaînes Unicode disposent d'un nombre important de méthodes effectuant des transformations basiques et des recherches.

Mise en forme de chaînes (`http://docs.python.org/lib/typesseq-strings.html`)

Les opérations de mise en forme appelées lorsque des chaînes et des chaînes Unicode sont les opérandes gauches de l'opérateur % sont décrites plus en détail dans cette page.

3.1.3 Chaînes Unicode

A partir de Python 2.0, un nouveau type de données destiné à stocker du texte est disponible pour les programmeurs : l'objet Unicode. Il peut être utilisé pour stocker et manipuler des données Unicode (voir http ://www.unicode.org/) et s'intègre bien avec les objets chaînes en fournissant des conversions automatiques là où c'est nécessaire.

Unicode offre l'avantage de fournir un numéro pour chaque caractère de chaque écriture utilisée dans les textes

modernes et anciens. Auparavant, il n'y avait que 256 numéros possibles pour les caractères d'écriture et les textes étaient donc typiquement associés à une page de codes qui réalisait l'association entre le numéros et les caractères d'écriture. Cela conduisait à beaucoup de confusion, spécialement en ce qui concerne l'internationalisation (écrite d'habitude comme '`i18n`' — '`i`' + caractères 18 + '`n`') des logiciels. Unicode résout ces problèmes en définissant une page de codes pour toutes les écritures.

Créer des chaînes Unicode en Python est exactement aussi simple que de créer des chaînes normales :

```
>>> u'Bonjour !'
u'Bonjour !'
```

Le '`u`' minuscule devant les guillemets indique qu'on souhaite créer une chaîne Unicode. Si vous désirez placer des caractères spéciaux dans la chaîne, vous pouvez le faire en utilisant l'encodage Python *Echappement-Unicode*. L'exemple suivant montre comment faire :

```
>>> u'Salut\u0020tout le monde !'
u'Salut tout le monde !'
```

La séquence d'échappement `\u0020` indique qu'il faut insérer le caractère Unicode dont la valeur ordinale est 0x0020 (le caractère espace) à l'endroit indiqué.

Les autres caractères sont interprétés en utilisant leurs valeurs numériques respectives directement comme des numéros Unicode. Si vous avez des textes de chaînes en encodage standard Latin-1 qui est utilisé dans de nombreux pays occidentaux, vous trouverez pratique que les 256 premiers caractères de l'Unicode soient les mêmes que les 256 caractères de Latin-1.

Pour les experts, il y a aussi un mode brut exactement comme pour les chaînes normales. Vous devez insérer 'ur' avant l'ouverture de la chaîne pour que Python utilise l'encodage *Raw-Unicode-Escape* (Echappement-Brut-Escape). Il n'appliquera la conversion `\uXXXX` ci-dessus que s'il y a un nombre impair d'antislash avant le petit 'u'.

```
>>> ur'Salut\u0020tout le monde !'
u'Salut tout le monde !'
>>> ur'Salut\\u0020tout le monde !'
u'Salut\\\\u0020tout le monde !'
```

Le mode brut est extrèmement utile lorsqu'il s'agit de saisir de nombreux antislash, comme ça peut être nécessaire dans les expressions rationnelles.

En dehors de ces encodages standards, Python fournit tout un ensemble d'autres moyens de créer des chaînes Unicode sur la base d'un encodage connu.

La fonction `unicode()` intégrée fournit un accès à tous les codecs (COdeurs et DECodeurs) Unicode enregistrés. Certains des encodages les mieux connus que ces codecs peuvent convertir sont *Latin-1*, *ASCII*, *UTF-8* et *UTF-16*. Les deux derniers sont des encodages à longueur variable qui permettent de stocker chaque caractères Unicode sur un ou plusieurs octets. L'encodage par défaut est normalement défini à ASCII, ce qui utilise les caractères de l'intervale 0 à 127 et rejette tout autre caractère avec une erreur. Quand une chaîne Unicode est affichée, écrite dans un fichier, ou convertie avec `str()`, la conversion survient en utilisant l'encodage par défaut.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

Pour convertir une chaîne Unicode vers une chaîne 8-bit en utilisant un encodage spécifique, les objets Unicode fournissent une méthode `encode()` qui prend un argument, le nom de l'encodage. Les noms en minuscules sont préférés pour les encodages.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Si vous avez des données dans un encodage spécifique et souhaitez produire une chaîne Unicode correspondante, vous pouvez utiliser la fonction `unicode()` avec le nom de l'encodage comme second argument.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Listes

Python connaît un grand nombre de types de données *composites*, utilisées pour regrouper un ensemble de valeurs. La plus riche en possibilités est la *liste*, qui peut être écrite comme une liste de valeurs (éléments) entre crochets et séparés par des virgules. Les éléments d'une liste n'ont pas nécessairement le même type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Comme les indices des chaînes, les indices des listes commencent à 0, et les listes peuvent être découpées, concaténées, et ainsi de suite :

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

A la différence des chaînes, qui sont *non-modifiables*, il est possible de changer les éléments individuels d'une liste :

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

L'affectation dans des tranches est aussi possible, et cela peut même changer la taille de la liste :

```
>>> # Remplacer certains éléments:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # En enlever certains:
... a[0:2] = []
>>> a
[123, 1234]
>>> # En insérer
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Insère (une copie de) soi-même au début
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

La fonction intégrée `len()` s'applique aussi aux listes :

```
>>> len(a)
8
```

Il est possible d'emboîter des listes (créer des listes contenant d'autres listes), par exemple :

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Notez que dans l'exemple précédent, `p[1]` et `q` se réfèrent réellement au même objet ! Nous reviendrons plus tard sur la *sémantique des objets*.

3.2 Premiers pas vers la programmation

Bien sûr, nous pouvons utiliser Python pour des tâches plus compliquées que d'ajouter deux et deux. Par exemple, nous pouvons écrire une sous-séquence de la suite de *Fibonacci* de la façon suivante :

```
>>> # Suite de Fibonacci
... # La somme de deux éléments définit le suivant
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Cet exemple introduit plusieurs fonctionnalités nouvelles.

- La première ligne contient une *affectation multiple* : les variables `a` et `b` prennent simultanément les nouvelles valeurs 0 et 1. Sur la dernière ligne l'affectation multiple est utilisée à nouveau, montrant que les expressions en partie droite sont d'abord toutes évaluées avant qu'aucune affectation ne se fasse.
- La boucle `while` s'exécute tant que la condition (ici : `b < 10`) reste vraie. En Python, comme en C, toute valeur entière différente de zéro est vraie ; zéro est faux. La condition pourrait aussi être une chaîne ou une valeur de type liste, en fait n'importe quelle séquence ; n'importe quoi avec une longueur différente de zéro est vrai, les séquences vides correspondent à faux. Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs de comparaison standard sont écrits de la même façon qu'en C : `<`, `>`, `==`, `<=`, `>=` et `!=`.
- Le *corps* de la boucle est *indenté* : l'indentation est le moyen par lequel Python regroupe les instructions. Python ne fournit pas (encore) une fonction d'édition de ligne intelligente, donc vous devez insérer une tabulation ou un espace pour chaque ligne indentée. En pratique vous préparerez les saisies plus compliquées avec un éditeur de texte ; la plupart des éditeurs de texte ont une fonction d'auto-indentation. Lorsqu'une instruction composée est entrée en mode interactif, elle doit être suivie d'une ligne vide pour indiquer qu'elle est terminée (car l'interpréteur ne peut pas deviner si vous avez tapé la dernière ligne).
- L'instruction `print` écrit la valeur de la ou des expressions qui lui sont données. Elle diffère de la simple écriture de l'expression (comme tout-à-l'heure dans les exemples de la calculatrice) dans la mesure où elle accepte plusieurs expressions et chaînes. Les chaînes sont imprimées sans quotes, et un espace est inséré entre les éléments, ce qui vous permet de les afficher dans un format plus sympathique, comme ceci :

  ```
  >>> i = 256*256
  >>> print 'La valeur de i est', i
  La valeur de i est 65536
  ```

  Une virgule finale empêche le retour chariot après l'affichage :

  ```
  >>> a, b = 0, 1
  >>> while b < 1000:
  ...     print b,
  ...     a, b = b, a+b
  ...
  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
  ```

  Notez que l'interpréteur insère un retour chariot avant d'imprimer le prompt suivant si la ligne n'a pas été complétée.

CHAPITRE

QUATRE

D’autres outils de contrôle d’exécution

A part l’instruction `while` que l’on vient de découvrir, Python comprend les instructions de contrôle d’exécution habituelles connues dans d’autres langages, avec quelques adaptations.

4.1 Instructions `if`

Peut-être l’instruction la plus connue est-elle l’instruction `if`. Par exemple :

```
>>> x = int(raw_input("Entrez un entier : "))
>>> if x < 0:
...      x = 0
...      print ’Négatif changé en zéro’
... elif x == 0:
...      print ’Zéro’
... elif x == 1:
...      print ’Un seul’
... else:
...      print ’Plus’
...
```

Il peut y avoir aucune ou plusieurs sections `elif`, et la section `else` est optionnelle. Le mot-clé ‘`elif`’ est une abréviation de ‘else if’, et est utile pour éviter une indentation excessive. Une séquence `if` … `elif` … `elif` … est un substitut pour les instructions `switch` ou `case` qu’on trouve dans d’autres langages.

4.2 Instructions `for`

L’instruction `for` en Python diffère un petit peu de ce que vous avez pu utiliser en C ou en Pascal. Au lieu d’itérer toujours dans une progression arithmétique de nombres (comme en Pascal), ou de laisser l’utilisateur complètement libre dans les tests et les pas d’itération (comme en C), l’instruction `for` de Python itère parmi les éléments de n’importe quelle séquence (une liste ou une chaîne), dans l’ordre où ils apparaissent dans la séquence. Par exemple (aucun jeu de mots volontaire) :

```
>>> # Mesurer quelques chaînes:
... a = [’chat’, ’fenêtre’, ’défenestrer’]
>>> for x in a:
...     print x, len(x)
...
chat 4
fenêtre 7
défenestrer 11
```

Il n’est pas prudent de modifier la séquence sur laquelle on itère dans la boucle (cela peut seulement arriver pour

les types de séquences modifiables, tels que les listes). Si vous avez besoin de modifier la liste sur laquelle vous itérez (par exemple, pour dupliquer des éléments sélectionnés), vous devez itérer sur une copie. La notation de découpage rend cela particulièrement pratique :

```
>>> for x in a[:]: # fait une copie de la liste entière par découpage
...     if len(x) > 8: a.insert(0, x)
...
>>> a
['défenestrer', 'chat', 'fenêtre', 'défenestrer']
```

4.3 La fonction `range()`

Si vous avez besoin d'itérer sur une séquence de nombres, la fonction intégrée `range()` vient à point. Elle génère des listes contenant des progressions arithmétiques :

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

Le nombre de fin qui lui est passé n'est jamais dans la liste générée ; `range(10)` génère une liste de 10 valeurs, exactement les indices des éléments d'une séquence de longueur 10. Il est possible de faire commencer l'intervalle à un autre nombre, ou de spécifier un incrément différent (même négatif) :

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Pour parcourir les indices d'une séquence, combinez `range()` et `len()` comme ci-dessous :

```
>>> a = ['Marie', 'avait', 'un', 'petit', 'mouton']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Marie
1 avait
2 un
3 petit
4 mouton
```

4.4 Les instructions `break` et `continue`, et les clauses `else` dans les boucles

L'instruction `break`, comme en C, sort de la plus petite boucle `for` ou `while` englobante.

L'instruction `continue`, également empruntée au C, continue sur la prochaine itération de la boucle.

Les instructions de boucle ont une clause `else` ; elle est exécutée lorsque la boucle se termine par épuisement de la liste (avec `for`) ou quand la condition devient fausse (avec `while`), mais pas quand la boucle est interrompue par une instruction `break`. Cela est expliqué dans la boucle suivante, qui recherche des nombres premiers :

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'égale', x, '*', n/x
...            break
...     else:
...          # la boucle s'est terminée sans trouver de facteur
...          print n, 'est un nombre premier'
...
2 est un nombre premier
3 est un nombre premier
4 égale 2 * 2
5 est un nombre premier
6 égale 2 * 3
7 est un nombre premier
8 égale 2 * 4
9 égale 3 * 3
```

4.5 L'Instruction `pass`

L'instruction `pass` ne fait rien. Elle peut être utilisée lorsqu'une instruction est requise syntaxiquement mais que le programme ne nécessite aucune action. Par exemple :

```
>>> while 1:
...       pass # Attente active d'une interruption au clavier
...
```

4.6 Définition de fonctions

Nous pouvons créer une fonction qui écrit la série de Fibonacci jusqu'à une limite quelconque :

```
>>> def fib(n):    # écrit la série de Fibonacci jusqu'à n
...     """Affiche une suite de Fibonacci jusqu'à n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Maintenant on appelle la fonction qui vient juste d'être définie
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Le mot-clé `def` débute la *définition* d'une fonction. Il doit être suivi par le nom de la fonction et une liste entre parenthèses de paramètres formels. Les instructions qui forment le corps de la fonction commencent sur la ligne suivante, indentée par une tabulation. La première instruction du corps de la fonction peut éventuellement être un texte dans une chaîne de caractères ; cette chaîne est la chaîne de documentation de la fonction, ou *docstring*.

Il y a des outils qui utilisent les docstrings pour générer automatiquement de la documentation papier, ou pour permettre à l'utilisateur de naviguer interactivement dans le code ; c'est une bonne technique que d'inclure les docstrings dans le code que vous écrivez, donc essayez de vous y habituer.

L'*exécution* d'une fonction génère une nouvelle table de symboles, utilisée pour les variables locales de la fonction. Plus précisément, toutes les affectations de variables dans une fonction stockent la valeur dans la table de symboles locale ; alors que les références à des variables regardent en premier dans la table de symboles locale, puis dans

la table de symboles globale, et enfin dans la table des noms intégrés. Ainsi, on ne peut affecter directement une valeur aux variables globales à l'intérieur d'une fonction (à moins de les déclarer avec une instruction `global`), bien qu'on puisse y faire référence.

Les vrais paramètres (arguments) d'un appel de fonction sont introduits dans la table de symboles locale de la fonction appelée quand elle est appelée ; ainsi, les arguments sont passés en utilisant un *passage par valeur*.[1] Quand une fonction appelée appelle à son tour une autre fonction, une nouvelle table de symboles locaux est créée pour cet appel.

La définition d'une fonction introduit le nom de la fonction dans la table de symboles courante. La valeur du nom de la fonction a un type qui est reconnu par l'interpréteur comme une fonction définie par l'utilisateur. Cette valeur peut être affectée à un autre nom qui peut alors être utilisé aussi comme une fonction. Cela permet de disposer d'un mécanisme général de renommage :

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Vous pourriez objecter que `fib` n'est pas une fonction mais une procédure. En Python, comme en C, les procédures sont juste des fonctions qui ne retournent pas de valeur. En fait, techniquement parlant, les procédures retournent bien une valeur, bien qu'elle soit plutôt décevante. Cette valeur est appelée `None` (c'est un nom intégré). La valeur `None` n'est normalement pas affichée par l'interpréteur si elle devait être la seule valeur écrite. Vous pouvez le vérifier si vous y tenez vraiment :

```
>>> print fib(0)
None
```

Ecrire une fonction qui retourne une liste des nombres de la suite de Fibonacci, au lieu de les imprimer, est très simple :

```
>>> def fib2(n): # retourne la série de Fibonacci jusqu'à n
...     """Retourne une liste contenant la série de Fibonacci jusqu'à n"""
...     resultat = []
...     a, b = 0, 1
...     while b < n:
...         resultat.append(b)    # voir ci-dessous
...         a, b = b, a+b
...     return resultat
...
>>> f100 = fib2(100)    # on l'appelle
>>> f100                # écrire le résultat
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Cet exemple, comme d'habitude, démontre quelques nouvelles caractéristiques de Python :

- L'instruction `return` termine une fonction en renvoyant une valeur. `return` sans une expression en argument renvoie `None`. Parvenir jusqu'au bout de la procédure renvoie également `None`.
- L'instruction `result.append(b)` appelle une *méthode* de l'objet `result`. Une méthode est une fonction qui 'appartient' à un objet et est nommée `obj.nommethode`, où `obj` est un objet (cela pourrait être une expression), et `nommethode` est le nom d'une méthode qui est définie d'après le type de l'objet. Différents types définissent différentes méthodes. Les méthodes de types différents peuvent avoir le même nom sans que cela soit source d'ambiguïtés. (Il est possible de

[1] En réalité, *passage par référence d'objet* serait une meilleure description, puisque si un objet modifiable est passé, l'appelant verra tous les changements que l'appelé y effectue (les éléments insérés dans une liste).

définir vos propres types d'objets et méthodes, en utilisant des *classes*, de la façon décrite en 9.) La méthode `append()` montrée précédemment, est définie pour les objets listes ; elle ajoute un nouvel élément à la fin de la liste. Dans cet exemple, c'est équivalent à '`result = result + [b]`', mais en plus performant.

4.7 Encore plus sur la définition de fonctions

Il est aussi possible de définir des fonctions à nombre d'arguments variable. Il y a trois façons de faire, qui peuvent être combinées.

4.7.1 Valeurs d'argument par défaut

La technique la plus utile consiste à spécifier une valeur par défaut pour un ou plusieurs arguments. Cela crée une fonction qui peut être appelée avec moins d'arguments qu'il n'en a été défini.

```
def demande_ok(question, tentatives=4, plainte='Oui ou non, svp!'):
    while 1:
        ok = raw_input(question)
        if ok in ('o', 'ou', 'oui'): return 1
        if ok in ('n', 'no', 'non', 'niet'): return 0
        tentatives = tentatives - 1
        if tentatives < 0: raise IOError, 'utilisateur refusenik'
        print plainte
```

Cette fonction peut être appelée soit comme ceci : `demande_ok('Etes vous sûr de vouloir quitter?')` ou comme ceci : `demande_ok('OK pour écrasement du fichier?', 2)`.

Les valeurs par défaut sont évaluées au moment de la définition de la fonction dans la portée *de définition*, ainsi

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

affichera `5`.

Avertissement important : La valeur par défaut est évaluée seulement une fois. Cela est important lorsque la valeur par défaut est un objet modifiable comme une liste ou un dictionnaire. Par exemple, la fonction suivante accumule les arguments qui lui sont passés au fur et à mesure des appels :

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Cela affichera

```
[1]
[1, 2]
[1, 2, 3]
```

Si vous ne voulez pas que la valeur par défaut soit partagée entre des appels successifs, vous pouvez plutôt écrire la fonction comme ceci :

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Arguments à mot-clé

Les fonctions peuvent aussi être appelées en utilisant des arguments mots-clés de la forme '*motcle = valeur*'. Par exemple, la fonction suivante :

```
def perroquet(voltage, etat='c\'est du solide', action='voom', type='Bleu Norvégien'):
    print "-- Ce perroquet ne fera pas", action,
    print "si vous le mettez sous", voltage, "Volts."
    print "-- Beau plumage, le", type
    print "-- Ca", etat, "!"
```

pourrait être appelée de l'une des façons suivantes :

```
perroquet(1000)
perroquet(action = 'VOOOOOM', voltage = 1000000)
perroquet('un millier', etat = 'fait bouffer les pissenlits par la racine')
perroquet('un milion', 'vous dégoute de la vie', 'de bonds')
```

mais les appels suivants seraient tous invalides :

```
perroquet()                          # manque un argument obligatoire
perroquet(voltage=5.0, 'rend mort')  # un argument non-mot-clé suit un mot-clé
perroquet(110, voltage=220)          # doublon de valeurs pour un argument
perroquet(acteur='John Cleese')      # mot-clé inconnu
```

En général, une liste d'arguments doit être constituée de tous les arguments de position, suivis de tous les arguments mots-clés, où ces mots-clés doivent être choisis parmi les noms des paramètres formels. Il n'est pas important qu'un paramètre formel ait une valeur par défaut ou non. Aucun argument ne peut recevoir une valeur plus d'une fois — les noms de paramètre formel correspondant aux arguments de position ne peuvent être utilisés comme mots-clés dans les mêmes appels.

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Quand un paramètre formel de la forme `**`*nom* est présent en dernière position, il reçoit un dictionnaire contenant tous les arguments mots-clés dont les mots-clés ne correspondent pas à un paramètre formel. Cela peut être combiné avec un paramètre formel de la forme `*`*nom* (décrit dans la sous-section suivante) qui reçoit un tuple contenant les arguments positionnels au-delà de la liste de paramètres formels. (`*`*nom* doit être placé avant `**`*nom*.) Par exemple, nous définissons une fonction comme ceci :

```
def fromagerie(type, *arguments, **motcles):
    print "-- Avez-vous du", type, '?'
    print "-- Je suis désolé, plus personne n'a de", type
    for arg in arguments: print arg
    print '-'*40
    cles = motcles.keys()
    cles.sort()
    for mc in cles : print mc, ':', motcles[mc]
```

Elle pourrait être appelée comme ceci :

```
fromagerie('Camembert', "Il est très coulant, monsieur.",
           "Il est vraiment très, TRES coulant, monsieur.",
           client='John Cleese',
           proprietaire='Michael Palin',
           sketch='Sketch de la Fromagerie' )
```

et bien sûr, elle écrirait :

```
-- Avez-vous du Camembert ?
-- Je suis désolé, plus personne n'a de Camembert
Il est très coulant, monsieur.
Il est vraiment très, TRES coulant, monsieur.
----------------------------------------
client : John Cleese
proprietaire : Michael Palin
sketch : Sketch de la Fromagerie
```

Notez que la méthode `sort()` de la liste de des mots-clés des noms d'arguments est appelée avant d'imprimer le contenu du dictionnaire `motcles` ; si cela n'est pas fait, l'ordre dans lequel les arguments sont imprimés n'est pas défini.

4.7.3 Listes d'arguments arbitraires

Finalement, l'option la moins fréquemment utilisée est de spécifier qu'une fonction peut être appelée avec un nombre d'arguments arbitraire. Ces arguments seront récupérés dans un tuple. Avant le nombre variable d'arguments, zéro ou plus arguments normaux pourraient être présents.

```
def fprintf(fichier, format, *args):
    fichier.write(format % args)
```

4.7.4 Listes d'arguments à déballer

La situation inverse se produit lorsque les arguments sont dans une liste ou un n-uplet mais doivent être déballés en vue d'une fonction qui requiert des arguments positionnels séparés. Par exemple, la fonction intégrée `range()` attend deux arguments séparés `start` et `stop`. Si ces derniers ne sont pas disponibles séparément, écrivez l'appel de la fonction avec l'opérateur `*` afin de déballer les arguments depuis une liste ou un n-uplet :

```
>>> range(3, 6)        # appel normal avec des arguments séparés
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)       # appel avec des arguments déballés depuis une liste
[3, 4, 5]
```

4.7.5 Les formes lambda

Suite à la demande populaire, quelques caractéristiques trouvées habituellement dans les langages de programmation fonctionnelle et dans Lisp ont été ajoutées à Python. Avec le mot-clé `lambda`, de petites fonctions anonymes peuvent être créées. Voici une fonction qui retourne la somme de ses deux arguments : '`lambda a, b : a+b`'. Les formes Lambda peuvent être utilisées chaque fois qu'un objet fonction est requis. Elles sont limitées syntaxiquement à une expression unique. Sémantiquement, elles sont juste de l'enrobage syntaxique pour une définition de fonction normale. Comme les définitions de fonctions imbriquées, les formes lambda peuvent faire référence à des variables de la portée qui les contient :

```
>>> def fabrique_incrementeur(n):
...     return lambda x, incr=n: x+incr
...
>>> f = fabrique_incrementeur(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.6 Chaînes de documentation (Docstrings)

Il existe des conventions émergentes à propos du contenu et du formatage des chaînes de documentation.

La première ligne devrait toujours être un résumé concis des objectifs de l'objet. Afin d'être bref, il ne devrait pas répéter explicitement le nom ou le type de l'objet, puisque ces informations sont disponibles par d'autres moyens (sauf si le nom se trouve être un verbe décrivant l'utilisation d'une fonction). Cette ligne devrait toujours commencer par une lettre majuscule et finir par une virgule.

S'il y a d'autres lignes dans la chaîne de documentation, la deuxième ligne devrait être vide, séparant visuellement le résumé du reste de la description. Les lignes suivantes devraient constituer un ou plusieurs paragraphes décrivant les conventions d'appel des objets, ses effets de bord, etc.

L'interpréteur python ne supprime pas l'indentation des chaînes de texte multilignes en Python, donc les outils qui traitent la documentation doivent supprimer l'indentation. Cela peut se faire en utilisant la convention suivante. La première ligne non-vide *après* la première ligne de la chaîne détermine la quantité d'indentation pour toute la chaîne de documentation. (On ne peut pas utiliser la première ligne puisqu'elle est généralement adjacente aux quotes ouvrantes de la chaîne donc son indentation n'est pas apparente dans le texte de la chaîne.) Les espaces "équivalents" à cette indentation sont ensuite supprimés du début de toutes les lignes de la chaîne. Des lignes indentées de façon moins importante ne devraient pas apparaître, mais si elles le font, tous leurs espaces en début de ligne devraient être supprimés. L'équivalence de l'espacement devrait être testée après l'expansion des tabulations (à 8 espaces, normalement).

Voici un exemple de docstring multi-ligne :

```
>>> def ma_fonction():
...     """Ne fait rien, mais le documente.
...
...     Non, vraiment, elle ne fait rien.
...     """
...     pass
...
>>> print ma_fonction.__doc__
Ne fait rien, mais le documente.

    Non, vraiment, elle ne fait rien.
```

CHAPITRE

CINQ

Structures de données

Ce chapitre décrit avec plus de détail quelques éléments que vous avez déjà étudié, et ajoute aussi quelques nouveautés.

5.1 Plus de détails sur les listes

Le type de données liste possède d'autres méthodes. Voici toutes les méthodes des objets listes :

`append(x)`
Equivalent à `a.insert(len(a), x)`.

`extend(L)`
Rallonge la liste en ajoutant à la fin tous les éléments de la liste donnée ; équivaut à `a[len(a):] = L`.

`insert(i, x)`
Insère un élément à une position donnée. Le premier argument est l'indice de l'élément avant lequel il faut insérer, donc `a.insert(0, x)` insère au début de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`.

`remove(x)`
Enlève le premier élément de la liste dont la valeur est `x`. Il y a erreur si cet élément n'existe pas.

`pop([i])`
Enlève l'élément présent à la position donnée dans la liste, et le renvoie. Si aucun indice n'est spécifié, `a.pop()` renvoie le dernier élément de la liste. L'élément est aussi supprimé de la liste.

`index(x)`
Retourne l'indice dans la liste du premier élément dont la valeur est `x`. Il y a erreur si cet élément n'existe pas.

`count(x)`
Renvoie le nombre de fois que `x` apparaît dans la liste.

`sort()`
Trie les éléments à l'intérieur de la liste.

`reverse()`
Renverse l'ordre des éléments à l'intérieur de la liste.

Un exemple qui utilise toutes les méthodes des listes :

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Utiliser les listes comme des piles

Les méthodes des listes rendent très facile l'utilisation d'une liste comme une pile, où le dernier élément ajouté est le premier élément récupéré (LIFO, "last-in, first-out"). Pour ajouter un élément au sommet de la pile, utilisez la méthode `append()`. Pour récupérer un élément du sommet de la pile, utilisez `pop()` sans indice explicite. Par exemple :

```
>>> pile = [3, 4, 5]
>>> pile.append(6)
>>> pile.append(7)
>>> pile
[3, 4, 5, 6, 7]
>>> pile.pop()
7
>>> pile
[3, 4, 5, 6]
>>> pile.pop()
6
>>> pile.pop()
5
>>> pile
[3, 4]
```

5.1.2 Utiliser les listes comme des files

Vous pouvez aussi utiliser facilement une liste comme une file, où le premier élément ajouté est le premier élément retiré (FIFO, "first-in, first-out"). Pour ajouter un élément à la fin de la file, utiliser `append()`. Pour récupérer un élément du devant de la file, utilisez `pop()` avec `0` pour indice. Par exemple ;

```
>>> file = ["Eric", "John", "Michael"]
>>> file.append("Terry")           # Terry arrive
>>> file.append("Graham")          # Graham arrive
>>> file.pop(0)
'Eric'
>>> file.pop(0)
'John'
>>> file
['Michael', 'Terry', 'Graham']
```

5.1.3 Outils de programmation fonctionnelle

Il y a trois fonctions intégrées qui sont très pratiques avec les listes : `filter()`, `map()`, et `reduce()`.

'`filter(`*fonction*`,` *sequence*`)`' renvoit une liste (du même type, si possible) contenant les seul éléments de la séquence pour lesquels *fonction*`(`*element*`)` est vraie. Par exemple, pour calculer quelques nombres premiers :

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

'`map(`*fonction*`,` *sequence*`)`' appelle *fonction*`(`*element*`)` pour chacun des éléments de la séquence et renvoie la liste des valeurs de retour. Par exemple, pour calculer les cubes :

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Plusieurs séquences peuvent être passées en paramètre ; la fonction doit alors avoir autant d'arguments qu'il y a de séquences et est appelée avec les éléments correspondants de chacune des séquences (ou `None` si l'une des séquences est plus courte que l'autre). Si `None` est passé en tant que fonction, une fonction retournant ses arguments lui est substituée.

En combinant ces deux cas spéciaux, on voit que '`map(None,` *liste1*`,` *liste2*`)`' est une façon pratique de transformer un couple de liste en une liste de couples. Par exemple :

```
>>> seq = range(8)
>>> def carre(x): return x*x
...
>>> map(None, seq, map(carre, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

'`reduce(`*fonction*`,` *sequence*`)`' renvoie une valeur unique construite par l'appel de la fonction binaire *fonction* sur les deux premiers éléments de la séquence, puis sur le résultat et l'élément suivant, et ainsi de suite. Par exemple, pour calculer la somme des nombres de 1 à 10 :

```
>>> def ajoute(x,y): return x+y
...
>>> reduce(ajoute, range(1, 11))
55
```

S’il y a seulement un élément dans la séquence, sa valeur est renvoyée ; si la séquence est vide, une exception est déclenchée.

Un troisième argument peut être transmis pour indiquer la valeur de départ. Dans ce cas, la valeur de départ est renvoyée pour une séquence vide, et la fonction est d’abord appliquée à la valeur de départ et au premier élément de la séquence, puis au résultat et à l’élément suivant, et ainsi de suite. Par exemple,

```
>>> def somme(seq):
...     def ajoute(x,y): return x+y
...     return reduce(ajoute, seq, 0)
...
>>> somme(range(1, 11))
55
>>> somme([])
0
```

5.1.4 *List Comprehensions*

Les *list comprehensions* fournissent une façon concise de créer des listes sans avoir recours à `map()`, `filter()` et/ou `lambda`. La définition de liste qui en résulte a souvent tendance à être plus claire que des listes construites avec ces outils. Chaque *list comprehension* consiste en une expression suivie d’une clause `for`, puis zéro ou plus clauses `for` ou `if`. Le résultat sera une liste résultant de l’évaluation de l’expression dans le contexte des clauses `for` et `if` qui la suivent. Si l’expression s’évalue en un tuple, elle doit être mise entre parenthèses.

```
>>> fruitfrais = [’  banane’, ’  myrtille ’, ’fruit de la passion  ’]
>>> [projectile.strip() for projectile in fruitfrais]
[’banane’, ’myrtille’, ’fruit de la passion’]
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]      # erreur - parenthèses obligatoires pour les tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

5.2 L'instruction `del`

Il y a un moyen d'enlever un élément d'une liste en ayant son indice au lieu de sa valeur : l'instruction `del`. Cela peut aussi être utilisé pour enlever des tranches dans une liste (ce que l'on a fait précédemment par remplacement de la tranche par une liste vide). Par exemple :

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` peut aussi être utilisé pour supprimer des variables complètes :

```
>>> del a
```

Faire par la suite référence au nom `a` est une erreur (au moins jusqu'à ce qu'une autre valeur ne lui soit affectée). Nous trouverons d'autres utilisations de `del` plus tard.

5.3 N-uplets (tuples) et séquences

Nous avons vu que les listes et les chaînes ont plusieurs propriétés communes, telles que l'indexation et les opérations de découpage. Elles sont deux exemples de types de données de type *séquence*. Puisque Python est un langage qui évolue, d'autres types de données de type séquence pourraient être ajoutés. Il y a aussi un autre type de données de type séquence standard : le *tuple* (n-uplet).

Un n-uplet consiste en un ensemble de valeurs séparées par des virgules, par exemple :

```
>>> t = 12345, 54321, 'salut!'
>>> t[0]
12345
>>> t
(12345, 54321, 'salut!')
>>> # Les Tuples peuvent être imbriqués:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'salut!'), (1, 2, 3, 4, 5))
```

Comme vous pouvez le voir, à l'affichage, les tuples sont toujours entre parenthèses, de façon à ce que des tuples de tuples puissent être interprétés correctement ; ils peuvent être saisis avec ou sans parenthèses, bien que des parenthèses soient souvent nécessaires (si le tuple fait partie d'une expression plus complexe).

Les tuples ont plein d'utilisations. Par exemple, les couples de coordonnées (x, y), les enregistrements des employés d'une base de données, etc. Les tuples, comme les chaînes, sont non-modifiables : il est impossible d'affecter individuellement une valeur aux éléments d'un tuple (bien que vous puissiez simuler quasiment cela avec le découpage et la concaténation).

Un problème particulier consiste à créer des tuples contenant 0 ou 1 élément : la syntaxe reconnaît quelques subtilités pour y arriver. Les tuples vides sont construits grâce à des parenthèses vides ; un tuple avec un élément est construit en faisant suivre une valeur d'une virgule (il ne suffit pas de mettre une valeur seule entre parenthèses). Moche, mais efficace. Par exemple :

```
>>> empty = ()
>>> singleton = 'salut',    # <-- notez la virgule en fin de ligne
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('salut',)
```

L'instruction `t = 12345, 54321, 'salut!'` est un exemple d' *emballage en tuple* (tuple packing) : les valeurs `12345`, `54321` et `'salut!'` sont emballées ensemble dans un tuple. L'opération inverse est aussi possible :

```
>>> x, y, z = t
```

Cela est appelé, fort judicieusement, *déballage de tuple* (tuple unpacking). Le déballage d'un tuple nécessite que la liste des variables à gauche ait un nombre d'éléments égal à la longueur du tuple. Notez que des affectations multiples ne sont en réalité qu'une combinaison d'emballage et déballage de tuples !

5.4 Ensembles

Python comporte également un type de données pour représenter des ensembles. Un `set` est une collection (non rangée) sans éléments dupliqués. Les emplois basiques sont le test d'appartenance et l'élimination des entrée dupliquées. Les objets ensembles supportent les opérations mathématiques comme l'union, l'intersection, la différence et la différence symétrique.

Voici une démonstration succincte :

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruits = set(basket) # create a set without duplicates
>>> fruits
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits # fast membership testing
True
>>> 'crabgrass' in fruits
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b # letters in both a and b
set(['a', 'c'])
>>> a ^ b # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.5 Dictionnaires

Un autre type de données intégré à Python est le *dictionnaire*. Les dictionnaires sont parfois trouvés dans d'autres langages sous le nom de "mémoires associatives" ou "tableaux associatifs". A la différence des séquences, qui sont indexées par un intervalle numérique, les dictionnaires sont indexés par des *clés*, qui peuvent être de n'importe quel type non-modifiable ; les chaînes et les nombres peuvent toujours être des clés. Les tuples peuvent être utilisés comme clés s'ils ne contiennent que des chaînes, des nombres ou des tuples. Vous ne pouvez pas utiliser des listes comme clés, puisque les listes peuvent être modifiées en utilisant leur méthode `append()`.

Il est préférable de considérer les dictionnaires comme des ensembles non ordonnés de couples *clé :valeur*, avec la contrainte que les clés soient uniques (à l'intérieur d'un même dictionnaire). Un couple d'accolades crée un dictionnaire vide : `{}`. Placer une liste de couples clé :valeur séparés par des virgules à l'intérieur des accolades ajoute les couples initiaux clé :valeur au dictionnaire ; c'est aussi de cette façon que les dictionnaires sont affichés.

Les opérations principales sur un dictionnaire sont le stockage d'une valeur à l'aide d'une certaine clé et l'extraction de la valeur en donnant la clé. Il est aussi possible de détruire des couples clé :valeur avec `del`. Si vous stockez avec une clé déjà utilisée, l'ancienne valeur associée à cette clé est oubliée. C'est une erreur d'extraire une valeur en utilisant une clé qui n'existe pas.

La méthode `keys()` d'un objet de type dictionnaire retourne une liste de toutes les clés utilisées dans le dictionnaire, dans un ordre quelconque (si vous voulez qu'elle soit triée, appliquez juste la méthode `sort()` à la liste des clés). Pour savoir si une clé particulière est dans le dictionnaire, utilisez la méthode `has_key()` du dictionnaire.

Voici un petit exemple utilisant un dictionnaire :

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```

Le constructeur `dict()` construit des dictionnaires directement à partir de listes de paires clé-valeur rangées comme des n-uplets. Lorsque les paires forment un motif, les *list list comprehensions* peuvent spécifier de manière compacte la liste de clés-valeurs.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)]) # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Plus loin dans ce tutoriel nous étudierons les "expressions générateurs" qui sont l'outil idéal pour fournir des paires clé-valeur au constructeur `dict()`.

5.6 Techniques de boucles

Lorsqu'on boucle sur un dictionnaire, les clés et les valeurs correspondantes peuvent être obtenues en même temps en utilisant la méthode `iteritems()`

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Lorsqu'on boucle sur une séquence, l'indice donnant la position et la valeur correspondante peuvent être obtenus en même temps en utilisant la fonction `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Pour boucler sur deux séquences, ou plus, en même temps, les éléments peuvent être appariés avec la fonction `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your %s? It is %s.' % (q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Pour boucler à l'envers sur une séquence, spécifiez d'abord la séquence à l'endroit, ensuite appelez la fonction `reversed()`.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Pour boucler sur une séquence comme si elle était triée, utilisez la fonction `sorted()` qui retourne une liste nouvelle triée tout en laissant la source inchangée.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

5.7 Plus de détails sur les conditions

Les conditions utilisées dans les instructions `while` et `if` peuvent contenir d'autres opérateurs en dehors des comparaisons.

Les opérateurs de comparaison `in` et `not in` vérifient si une valeur apparaît (ou non) dans une séquence. Les opérateurs `is` et `is not` vérifient si deux objets sont réellement le même objet ; cela se justifie seulement pour les objets modifiables comme les listes. Tous les opérateurs de comparaison ont la même priorité, qui est plus faible que celle de tous les opérateurs numériques.

Les comparaisons peuvent être enchaînées. Par exemple, `a < b == c` teste si `a` est strictement inférieur à `b` et de plus si `b` est égal à `c`.

Les comparaisons peuvent être combinées avec les opérateurs Booléens `and` (et) et `or` (ou), et le résultat d'une comparaison (ou de n'importe quel autre expression Booléenne) peut être inversé avec `not` (pas). Ces opérateurs ont encore une fois une priorité inférieure à celle des opérateurs de comparaison ; et entre eux, `not` a la plus haute priorité, et `or` la plus faible, de sorte que `A and not B or C` est équivalent à `(A and (not B)) or C`. Bien sûr, les parenthèses peuvent être utilisées pour exprimer les compositions désirées.

Les opérateurs Booléens `and` et `or` sont des opérateurs dits *court-circuit* : leurs arguments sont évalués de gauche à droite, et l'évaluation s'arrête dès que le résultat est trouvé. Par exemple, si `A` et `C` sont vrais mais que `B` est faux, `A and B and C` n'évalue pas l'expression `C`. En général, la valeur de retour d'un opérateur court-circuit, quand elle est utilisée comme une valeur générale et non comme un Booléen, est celle du dernier argument évalué.

Il est possible d'affecter le résultat d'une comparaison ou une autre expression Booléenne à une variable. Par exemple,

```
>>> chaine1, chaine2, chaine3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = chaine1 or chaine2 or chaine3
>>> non_null
'Trondheim'
```

Notez qu'en Python, au contraire du C, les affectations ne peuvent pas être effectuées à l'intérieur des expressions. Les programmeurs C ronchonneront peut-être, mais cela évite une classe de problèmes qu'on rencontre dans les programmes C : écrire = dans une expression alors qu'il fallait ==.

5.8 Comparer les séquences et d'autres types

Les objets de type séquence peuvent être comparés à d'autres objets appartenant au même type de séquence. La comparaison utilise l'ordre *lexicographique* : les deux premiers éléments sont d'abord comparés, et s'ils diffèrent cela détermine le résultat de la comparaison ; s'ils sont égaux, les deux éléments suivants sont comparés, et ainsi de suite, jusqu'à ce que l'une des deux séquences soit épuisée. Si deux éléments à comparer sont eux-mêmes des séquences du même type, la comparaison lexicographique est reconsidérée récursivement. Si la comparaison de tous les éléments de deux séquences les donne égaux, les séquences sont considérées comme égales. Si une séquence est une sous-séquence initiale de l'autre, la séquence la plus courte est la plus petite (inférieure). L'ordonnancement lexicographique pour les chaînes utilise l'ordonnancement ASCII pour les caractères. Quelques exemples de comparaisons de séquences du même type :

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Notez que la comparaison d'objets de types différents est licite. Le résultat est déterministe mais arbitraire : les types sont triés selon leur nom. Ainsi une liste (list) est toujours inférieure à une chaîne (string), une chaîne (string) est toujours inférieure à un n-uplet (tuple), etc. Les types numériques mélangés sont comparés en fonction de leur

valeur numérique, ainsi 0 est égal à 0.0, etc.[1]

[1] On ne doit pas se fier aux règles de comparaison pour des objets de types différents ; elles pourraient changer dans une version ultérieure du langage.

CHAPITRE

SIX

Modules

Si vous quittez l'interprèteur de Python et le lancez à nouveau, les définitions que vous avez faites (fonctions et variables) sont perdues. Par conséquent, si vous voulez écrire un programme plus long, vous feriez mieux d'utiliser à la place un éditeur de texte pour préparer le source pour l'interpréteur et de le lancer avec ce fichier comme entrée. Cela s'appelle créer un script. Quant votre programme devient plus long, vous pouvez vouloir le couper en plusieurs fichiers pour une maintenance plus facile. Vous pouvez également vouloir utiliser dans plusieurs programmes une fonction pratique que vous avez écrite sans copier sa définition dans chaque programme.

Pour supporter cela, Python offre un moyen de mettre des définitions dans un fichier et de les utiliser dans un script ou dans un session interactive de l'interpréteur. Un tel fichier s'appelle un *module* ; les définitions d'un module peuvent être *importées* dans un autre module ou dans le module *principal* (la collection de variables à laquelle vous avez accès dans un script exécuté depuis le plus haut niveau et dans le mode calculatrice).

Un module est un fichier contenant des définitions et des instructions Python. Le nom de fichier est le nom du module auquel est ajouté le suffixe '.py'. Dans un module, le nom du module (comme chaîne de caractères) est disponible comme valeur de la variable globale `__name__`. Par exemple, employez votre éditeur de texte préféré pour créer un fichier appelé 'fibo.py' dans le répertoire courant avec le contenu suivant :

```
# Module nombres de Fibonacci

def fib(n):    # écrit la série de Fibonacci jusqu'à n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # retourne la série de Fibonacci jusqu'à n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Maintenant lancez l'interpréteur Python et importez ce module avec la commande suivante :

```
>>> import fibo
```

Cela n'écrit pas les noms des fonctions définies dans `fibo` directement dans la table de symboles actuelle ; cela y insère seulement le nom de module `fibo`. En utilisant le nom de module vous pouvez accéder aux fonctions :

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

si vous avez l'intention d'utiliser souvent une fonction, vous pouvez l'affecter à un nom local :

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Encore plus sur les modules

Un module peut contenir des instructions exécutables aussi bien que des définitions de fonction. Ces instructions sont destinées à initialiser le module. On les exécute seulement la *première* fois que le module est importé quelque part.[1]

Chaque module a sa propre table de symboles privée, qui est utilisée comme table de symbole globale par toutes les fonctions définies dans le module. Ainsi, l'auteur d'un module peut utiliser des variables globales dans le module sans s'inquiéter des désaccords accidentels avec les variables globales d'un utilisateur. D'autre part, si vous savez que ce que vous faites, vous pouvez accéder aux variables globales d'un module avec la même notation que celle employée pour se référer à ses fonctions, `nommodule.nomelem`.

Les modules peuvent importer d'autres modules. Il est d'usage mais pas obligatoire de placer toutes les instructions `import` au début d'un module (ou d'un script). Les noms du module importé sont placés dans la table globale de symboles du module importateur.

Il y a une variante de l'instruction `import` qui importe des noms d'un module directement dans la table de symboles du module importateur. Par exemple :

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Cela n'introduit pas dans la table de symboles locale le nom du module duquel les éléments importés sont issus (ainsi dans l'exemple, `fibo` n'est pas défini).

Il y a même une variante pour importer tous les noms qu'un module définit :

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Cela importe tous les noms excepté ceux qui commencent par un tiret-bas (_).

6.1.1 Le chemin de recherche du module

Quand un module nommé `spam` est importé, l'interpréteur recherche un fichier nommé 'spam.py' dans le répertoire courant, et puis dans la liste de répertoires indiquée par la variable d'environnement PYTHONPATH. Elle a la même syntaxe que la variable du shell PATH, c'est-à-dire, une liste de noms de répertoire. Quand PYTHON-

[1] En fait, les définitions de fonctions sont aussi des 'instructions' qui sont 'exécutées' ; l'exécution insère le nom de la fonction dans la table de symboles globale du module.

PATH n'est pas renseigné, ou quand le fichier n'y est pas trouvé, la recherche continue dans un chemin d'accès par défaut, dépendant de l'installation ; sur UNIX, c'est habituellement '. :/usr/local/lib/python'.

En fait, les modules sont recherchés dans la liste de répertoires donnée par la variable `sys.path` qui est initialisée à partir du répertoire contenant le script d'entrée (ou le répertoire actuel), PYTHONPATH et le chemin par défaut, dépendant de l'installation. Cela permet aux programmes Python qui savent ce qu'ils font de modifier ou de remplacer le chemin d'accès aux modules. Notez que puisque le répertoire contenant le script en cours d'exécution est sur le chemin de recherche, il est important que le script n'ait pas le même nom qu'un module standard, ou bien Python essaiera de charger le script comme module quand ce module sera importé. Cela provoquera en général une erreur. Voyez la section 6.2, "Modules standards." pour plus d'informations.

6.1.2 Fichiers "compilés" de Python

Pour accélérer de manière importante le temps de lancement des petits programmes qui utilisent beaucoups de modules standard, si un fichier appelé 'spam.pyc' existe dans le répertoire où 'spam.py' se trouve, il est supposé contenir une version du module `spam` déjà compilée "en byte-code" L'heure de modification de la version de 'spam.py' employée pour créer 'spam.pyc' est enregistrée dans 'spam.pyc', et le fichier est ignoré si ceux-ci ne s'accordent pas.

Normalement, vous n'avez rien à faire pour créer le fichier 'spam.pyc'. Toutes les fois que 'spam.py' est compilé avec succès, une tentative est faite pour écrire la version compilée sur 'spam.pyc'. Il n'y a pas d'erreur si cette tentative échoue ; si pour une raison quelconque le fichier n'est pas écrit complètement, le fichier 'spam.pyc' résultant sera identifié comme incorrect et ainsi ignoré plus tard. Le contenu du fichier 'spam.pyc' est indépendant de la plate-forme, ainsi un répertoire de module de Python peut être partagé par des machines d'architectures différentes.

Quelques trucs pour les experts :

- Quand l'interpréteur de Python est appelé avec l'indicateur `-O`, du code optimisé est produit et enregistré dans des fichiers '.pyo'. L'optimiseur actuel n'aide pas beaucoup ; il retire seulement les instructions `assert` et des instructions `SET_LINENO`. Quand `-O` est utilisé, *tout* le byte-code est optimisé ; les fichiers `pyc` sont ignorés et des fichiers `py` sont compilés en byte-code optimisé.
- Passer deux options **-O** en paramètres à l'interpréteur Python (**-OO**) forcera le compilateur de *bytecode* à effectuer des optimisations qui pouraient dans certains cas rares avoir pour résultat des programmes ne fonctionnant pas correctement. Actuellement, seules les chaînes `__doc__` sont enlevées du *bytecode*, ce qui a pour résultat des fichiers '.pyo' plus compacts. Puisque certains programmes pourraient s'appuyer sur le fait que celles-ci soient disponibles, vous devriez utiliser cette option uniquement si vous savez ce que vous faites.
- Un programme ne fonctionne pas plus rapidement quand on le charge depuis un fichier ''.pyc'' ou '.pyo' que quand on le charge depuis un '.py' ; la seule chose qui est plus rapide pour les fichiers '.pyc' ou '.pyo' est la vitesse à laquelle ils sont chargés.
- Quand un script est exécuté en donnant son nom sur la ligne de commande, le byte-code pour le script n'est jamais écrit dans un fichier '.pyc' ou '.pyo'. Ainsi, le temps de démarrage d'une séquence type peut être réduit en déplaçant la majeure partie de son code dans un module et en ayant un petit script d'amorce qui importe ce module.
- Il est possible d' avoir un fichier appelé 'spam.pyc' (ou 'spam.pyo' quand `-O` est utilisé) sans module 'spam.py' dans le même module. Cela peut être employé pour distribuer une bibliothèque de code Python sous une forme qui est moyennement difficile à décompiler.
- Le module `compileall` peut créer des fichiers '.pyc' (ou des fichiers '.pyo' quand `-O` est utilisé) pour tous les modules présents dans un répertoire.

6.2 Modules standard

Python est livré avec une bibliothèque de modules standard, décrite dans un document séparé, *Python Library Reference* ("Library Reference" ci-après). Quelques modules sont intégrés dans l'interpréteur ; ceux-ci permettent d'accéder à des opérations qui ne font pas partie du noyau du langage mais sont néanmoins intégrées, pour des

raisons d'efficacité ou pour permettre d'accéder aux primitives du système d'exploitation telles que les appels système. La définition de l'ensemble de ces modules standards est une option de configuration qui dépend aussi de la plate-forme sous-jacente. Par exemple, le module `amoeba` est seulement fourni sur les systèmes qui supportent d'une façon ou d'une autre les primitives d'Amoeba. Un module particulier mérite une certaine attention : `sys`, qui est intégré dans chaque interpréteur de Python. Les variables `sys.ps1` et `sys.ps2` définissent les chaînes de caractères utilisées en tant qu'invites primaire et secondaire :

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Ces deux variables sont seulement définies si l'interpréteur est en mode interactif.

La variable `sys.path` est une liste de chaînes de caractères qui déterminent le chemin de recherche des modules pour l'interpréteur. Il est initialisé à un chemin par défaut à partir de la variable d'environnement PYTHONPATH, ou d'une valeur par défaut intégrée au programme si PYTHONPATH n'est pas renseigné. Vous pouvez la modifier en utilisant des opérations standard sur des listes :

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 La fonction `dir()`

La fonction intégrée `dir()` est employée pour découvrir les noms qu'un module définit. Elle renvoie une liste triée de chaînes de caractères :

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'argv', 'builtin_module_names',
 'byteorder', 'copyright', 'displayhook', 'exc_info', 'exc_type',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'getdefaultencoding',
 'getdlopenflags', 'getrecursionlimit', 'getrefcount', 'hexversion',
 'maxint', 'maxunicode', 'modules', 'path', 'platform', 'prefix', 'ps1',
 'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
 'version_info', 'warnoptions']
```

Sans arguments, `dir()` énumère les noms que vous avez définis :

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Notez qu'elle énumère tous les types de noms : les variables, les modules, les fonctions, etc.

`dir()` n'énumère pas les noms des fonctions et des variables intégrées. Si vous en voulez une liste, elles sont définies dans le module standard `__builtin__` :

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TypeError', 'UnboundLocalError',
 'UnicodeError', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'buffer', 'callable', 'chr', 'classmethod',
 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr',
 'dict', 'dir', 'divmod', 'eval', 'execfile', 'exit', 'file', 'filter',
 'float', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len',
 'license', 'list', 'locals', 'long', 'map', 'max', 'min', 'object',
 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range', 'raw_input',
 'reduce', 'reload', 'repr', 'round', 'setattr', 'slice', 'staticmethod',
 'str', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange',
 'zip']
```

6.4 Paquetages

Les paquetages sont un moyen de structurer l'espace des noms de modules Python en utilisant "les noms de modules pointés". Par exemple, le nom de module `A.B` désigne un sous-module nommé 'B' dans un module nommé 'A'. Tout comme l'utilisation des modules permet aux auteurs de différents modules de ne pas s'inquiéter au sujet des noms des variables globales de chacun des autres modules, l'utilisation des noms de modules pointés dispense l'auteur de paquetages multi-modules comme NumPy ou PIL de devoir s'inquiéter au sujet de leurs noms de modules.

Supposez que vous vouliez concevoir une collection de modules (un "paquetage") pour la manipulation uniforme des fichiers de sons et des données de son. Il y a beaucoup de formats de fichier de sons différents (habituellement reconnus par leur extension, par exemple : '.wav', '.ai', '.au'), ainsi vous pouvez avoir besoin de créer et mettre à jour une collection grandissante de module pour la conversion entre les divers formats de fichier. Il y a également beaucoup d'opérations différentes que vous pourriez vouloir exécuter sur des données de sons (comme le mixage, ajouter de l'écho, appliquer une fonction d'égalisation, créer un effet artificiel de stéréo), ainsi en complément, vous écrirez une série interminable de modules pour réaliser ces opérations. Voici une structure possible pour votre paquetage (exprimé en termes de système de fichiers hiérarchique) :

```
Sound/                          Paquetage de niveau supérieur
      __init__.py               Initialisation du paquetage sons
      Formats/                  Sous-paquetage pour la conversion des formats de fichiers
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Sous-paquetage pour les effets sonores
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Sous-paquetage pour les filtres
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Quand on importe un paquetage, Python examine les répertoires de `sys.path` à la recherche du sous-répertoire du paquetage.

Les fichiers '__init__.py' sont obligatoires pour que Python considère les répertoires comme contenant des paquetages ; cela est fait pour empêcher des répertoires avec un nom commun, tel que '`string`', de cacher involontairement les modules valides qui apparaissent plus tard dans le chemin de recherche de module. Dans le cas le plus simple, '__init__.py' peut juste être un fichier vide, mais doit pouvoir également exécuter du code d'initialisation pour le paquetage ou positionner la variable `__all__`, décrite ci-dessous.

Les utilisateurs du paquetage peuvent importer individuellement des modules du paquetage, par exemple :

```
import Sound.Effects.echo
```

Cela charge le sous-module `Sound.Effects.echo`. Il doit être référencé avec son nom complet.

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Une autre solution pour importer le sous-module est :

```
from Sound.Effects import echo
```

Cela charge le sous-module `echo`, et le rend également disponible sans son préfixe de paquetage, ainsi il peut être utilisé comme suit :

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Une autre variante consiste encore à importer la fonction ou la variable désirée directement :

```
from Sound.Effects.echo import echofilter
```

Encore une fois, cela charge le sous-module `echo`, et rend sa fonction echofilter disponible directement :

```
echofilter(input, output, delay=0.7, atten=4)
```

Notez qu'en utilisant `from` *paquetage* `import` *element*, l'élément peut être un sous-module (ou sous-paquetage) du paquetage, ou un autre nom défini dans le paquetage, comme une fonction, une classe ou une variable. L'instruction `import` teste d'abord si l'élément est défini dans le paquetage ; sinon, elle suppose que c'est un module et essaye de le charger. Si elle ne le trouve pas, `ImportError` est déclenché.

Au contraire, en utilisant la syntaxe `import` *element.souselement.soussouselement*, chaque élément excepté le dernier doit être un paquetage ; le dernier élément peut être un module ou un paquetage mais ne peut pas être une classe ou une fonction ou une variable définie dans l'élément précédent.

6.4.1 Importer * depuis un paquetage

Maintenant, qu'est-ce qui se produit quand l'utilisateur écrit `from Sound.Effects import *` ? Dans le meilleur des cas, on espérerait que cela s'adresse d'une façon ou d'une autre au système de fichiers, trouve quels sous-modules sont présents dans le paquetage, et les importe tous. Malheureusement, cette opération ne fonctionne pas très bien sur des plate-formes Mac et Windows, où le système de fichiers n'a pas toujours des informations précises sur la casse d'un nom de fichier ! Sur ces plate-formes, il n'y a aucun moyen garanti de savoir si un fichier 'ECHO.PY' devrait être importé en tant que module `echo`, `Echo` ou `ECHO`. (Par exemple, Windows 95 a la fâcheuse habitude de montrer tous les noms de fichier avec une première lettre en capitale.) La restriction de nom de fichier DOS 8+3 ajoute un autre problème intéressant pour les longs noms de modules.

La seule solution est que l'auteur de module fournisse un index explicite du module. L'instruction d'importation utilise la convention suivante : si le code '__init__.py' d'un paquetage définit une liste nommée `__all__`, celle-ci est utilisée comme la liste des noms de modules qui doivent être importés quand `from` *paquetage* `import *` est rencontré. Il appartient à l'auteur du paquetage de tenir cette liste à jour quand une nouvelle version du paquetage est livrée. Les auteurs de paquetage peuvent également décider de ne pas la supporter, s'ils ne souhaitent pas une utilisation d'importation par * de leur module. Par exemple, le fichier `Sounds/Effects/__init__.py` pourrait contenir le code suivant :

```
__all__ = ["echo", "surround", "reverse"]
```

Cela signifierait que `from Sound.Effects import *` importerait les trois sous-modules du paquetage `Sound`

Si `__all__` n'est pas défini, l'instruction `from Sound.Effects import *` *n'importe pas* dans l'espace des noms actuel l'ensemble des sous-modules du paquetage `Sound.Effects` ; elle s'assure seulement que le paquetage `Sound.Effects` a été importé (probablement en exécutant son code d'initialisation, '__init__.py') et puis importe tous les noms définis dans le module, quels qu'ils soient. Cela inclut tout nom défini (et tout sous-module chargé explicitement) par '__init__.py'. Elle inclut également tous les sous-modules du paquetage qui ont été chargés de façon explicite par des instructions d'importation précédentes. Considérons ce code :

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Dans cet exemple, les modules echo et surround sont importés dans l'espace des noms actuel parce qu'ils sont définis dans le paquetage `Sound.Effects` quand l'instruction `from...import` est exécutée. (Cela fonctionne également quand `__all__` est défini.)

Notez qu'en général la pratique de l'importation par * d'un module ou paquetage fait froncer les sourcils, puisqu'elle conduit souvent à un code très peu lisible. Cependant, il est correct de l'employer pour éviter la saisie au clavier lors des sessions interactives, et parce que certains modules sont conçus pour exporter seulement les noms qui correspondent à certains motifs.

Rappelez-vous, il n'y a rien d'incorrect à utiliser `from Paquetage import sous_module_specifique` ! En fait, c'est la notation recommandée à moins que le module importateur ne doive utiliser des sous-modules avec le même nom, issus de paquetages différents.

6.4.2 Références intra-paquetage

Les sous-modules doivent souvent se faire référence mututellement. Par exemple, le module `surround` pourrait utiliser le module `echo`. En fait, de telles références sont si communes que l'instruction `import` regarde d'abord dans le paquetage contenant avant de regarder dans le chemin de recherche standard de module. Ainsi, le module surround peut simplement utiliser `import echo` ou `from echo import echofilter`. Si le module importé n'est pas trouvé dans le paquetage actuel (le paquetage dont le module actuel est un sous-module), l'instruction `import` recherche un module au niveau supérieur avec le nom donné.

Quand des paquetages sont structurés dans des sous-paquetages (comme avec le paquetage `Sound` dans l'exemple), il n'y a aucun raccourci pour se référer à des sous-modules des paquetages enfants de mêmes parents — le nom complet du sous-paquetage doit être utilisé. Par exemple, si le module `Sound.Filters.vocoder` doit utiliser le module `echo` du paquetage `Sound.Effects`, il peut utiliser `from Sound.Effects import echo`.

6.4.3 Paquetages dans des répertoires multiples

Les paquetages possèdent un attribut plus spécial, `__path__`. Celui-ci est initialisé de manière à être une liste comportant le nom du répertoire contenant les fichiers `'__init__.py'` des paquetage, cette initialisation se faisant avant que le code contenu dans ces fichiers soit exécuté. Cette variable peut être modifiée ; faire cela affecte les recherches ultérieures de modules et de sous-paquetages contenus dans le paquetage. Cette possibilité n'est pas souvent nécessaire, mais elle peut être utilisée pour étendre l'ensemble des modules qui se trouvent dans un paquetage.

CHAPITRE

SEPT

Entrées et sorties

Il y a plusieurs manières de présenter l'affichage produit par un programme ; les données peuvent être imprimées sous une forme humainement lisible, ou être écrites dans un fichier pour un usage ultérieur. Ce chapitre présentera certaines de ces possibilités.

7.1 Un formatage de sortie plus fantaisiste

Jusqu'ici nous avons rencontré deux manières d'afficher des valeurs : *les instructions d'expression* et l'instruction `print`. (Une troisième manière est d'utiliser la méthode `write()` des objets fichier ; le fichier de sortie standard peut être référencé par `sys.stdout`. Voyez le manuel Library Reference pour plus d'informations.)

Vous souhaiterez souvent avoir plus de contrôle sur le formatage de vos sorties que d'imprimer simplement des valeurs séparées par des espaces. Il y a deux manières de formater vos sorties ; la première manière est de faire toutes les manipulations de chaînes de caractères vous-même ; en utilisant les opérations de concaténation et de découpage de chaînes de caractères, vous pouvez créer n'importe quel format que vous puissiez imaginer. Le module standard `string` contient quelques opérations utiles pour remplir des chaînes de caractères à une largeur de colonne donnée ; celles-ci seront discutées sous peu. La deuxième manière est d'utiliser l'opérateur `%` avec une chaîne de caractères comme argument de gauche, `%` interprète l'argument de gauche comme une chaîne de formatage comme pour la fonction `sprintf()` du langage C à appliquer à l'argument de droite, et retourne une chaîne de caractères résultant de cette opération de formatage.

Il reste naturellement une question : comment convertissez-vous des valeurs en chaînes de caractères ? Heureusement, Python a des moyens de convertir n'importe quelle valeur en chaîne de caractères : passez-la à la fonction `repr()`, ou écrivez juste la valeur entre des guillemets renversés (anti-quotes : `“`, équivalent à `repr()`).

La fonction `str()` est faite pour renvoyer des représentatuons de valeurs qui sont assez faciles à lire par les humains, alors que `repr()` est faite pour générer des représentations qui puissent être lues par l'interpréteur (ou générer une `SyntaxError` s'il n'y a pas de syntaxe équivalente). Pour des objets qui n'ont pas de représentation particulière pour la consommation humaine, `str()` renverra la même valeur que `repr()`. De nombreuses valeurs, comme les nombres ou les structures comme les listes et les dictionnaires, ont la même représentation dans les deux fonctions. Les chaînes et les nombres à virgule flottante, en particulier, ont deux représentations distinctes.

Quelques exemples :

```
>>> s = 'Salut, tout le monde.'
>>> str(s)
'Salut, tout le monde.'
>>> `s`
"'Salut, tout le monde.'"
>>> str(0.1)
'0.1'
>>> `0.1`
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'La valeur de x est ' + `x` + ', et y est ' + `y` + '...'
>>> print s
La valeur de x est 32.5, et y est 40000...
>>> # Les anti-quotes marchent avec d'autres types en dehors des nombres:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[32.4, 40000]'
>>> # Convertir une chaîne ajoute des quotes de chaîne et des antislash:
... salut = 'salut, monde\n'
>>> saluts = `salut`
>>> print saluts
'salut, monde\n'
>>> # L'argument des anti-quotes peut être un tuple:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

Voici deux manières d'écrire une table des carrés et des cubes :

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(`x`, 2), string.rjust(`x*x`, 3),
...     # Notez la virgule à la fin de la ligne précédente
...     print string.rjust(`x*x*x`, 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Notez qu'un espace entre chaque colonne a été ajouté à cause de la façon dont `print` fonctionne : elle ajoute

toujours des espaces entre ses arguments.)

Cet exemple présente la fonction `string.rjust()`, qui justifie à droite une chaîne de caractères dans un champ d'une largeur donnée en la complétant avec des espaces du côté gauche. Il y a les fonctions semblables `string.ljust()` et `string.center()`. Ces fonctions n'écrivent rien, elles renvoient juste une nouvelle chaîne de caractères. Si la chaîne de caractères d'entrée est trop longue, elles ne la tronquent pas, mais la renvoient sans changement ; cela gâchera votre présentation de colonne mais c'est habituellement mieux que l'alternative, qui serait de tricher au sujet d'une valeur. (Si vous voulez vraiment la troncature vous pouvez toujours ajouter une opération de découpage, comme '`string.ljust(x, n)[0 :n]`'.)

Il y a une autre fonction, `string.zfill()`, qui complète une chaîne de caractères numérique du côté gauche avec des zéros. Elle sait gérer les signes positifs et négatifs :

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

L'utilisation de l'opérateur % ressemble à ceci :

```
>>> import math
>>> print 'La valeur de PI est approximativement %5.3f.' % math.pi
La valeur de PI est approximativement 3.142.
```

S'il y a plus d'un descripteur de format dans la chaîne de caractères, vous devez passer un tuple comme opérande de droite, comme dans cet exemple :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> for nom, telephone in table.items():
...     print '%-10s ==> %10d' % (nom, telephone)
...
Jack       ==>       4098
Dcab       ==>    8637678
Sjoerd     ==>       4127
```

La plupart des formats fonctionnent exactement comme en C et exigent que vous passiez le type approprié ; cependant, si vous ne le faites pas vous obtenez une exception, pas un core dump. Le format de `%s` est moins strict : si l'argument correspondant n'est pas un objet chaîne de caractères, il est converti en chaîne de caractères en utilisant la fonction intégrée `str()`. Utiliser `*` pour passer la largeur ou la précision comme argument (entier) séparé est possible. Les formats `%n` et `%p` du C ne sont pas supportés.

Si vous avez une chaîne de caractères de formatage vraiment longue que vous ne voulez pas fractionner, il serait élégant de pouvoir référencer les variables à formater par leur nom et pas par leur position. Cela peut être fait en utilisant la forme `%(name)format`, comme montré ici :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

C'est particulièrement utile en combinaison avec la fonction intégrée `vars()`, qui renvoie un dictionnaire contenant toutes les variables locales.

7.2 Lire et écrire des fichiers

`open()` renvoie un objet de type fichier, et est utilisée plus généralement avec deux arguments : '`open(`*nomfichier*`, `*mode*`)`'.

```
>>> f=open('/tmp/fichiertravail', 'w')
>>> print f
<open file '/tmp/fichiertravail', mode 'w' at 80a0960>
```

Le premier argument est une chaîne de caractères contenant le nom du fichier. Le deuxième argument est une autre chaîne de caractères contenant quelques caractères décrivant la manière d'utiliser le fichier. *mode* vaut `'r'` quand le fichier doit être seulement lu, `'w'` pour seulement écrit (un fichier déjà existant avec le même nom sera effacé), et `'a'` ouvre le fichier en ajout ; les données écrites dans le fichier seront automatiquement ajoutées à la fin. `'r+'` ouvre le fichier pour la lecture et l'écriture. L'argument *mode* est facultatif ; `'r'` sera pris par défaut s'il est omis.

Sur Windows et Macintosh, `'b'` ajouté au mode ouvre fichier en mode binaire, donc il y a aussi des modes comme `'rb'`, `'wb'`, et `'r+b'`. Windows fait la distinction entre fichier texte et binaire ; les caractères de fin de ligne dans des fichiers texte sont automatiquement modifiés légèrement quand des données sont lues ou écrites. Cette modification imperceptible des données du fichier marche très bien pour des fichiers textes ASCII, mais elle altèrera des données binaires comme dans des fichiers JPEG ou '.EXE'. Faites très attention à utiliser le mode binaire en lisant et en écrivant de tels fichiers. (notez que la sémantique précise du mode texte sur le Macintosh dépend de la bibliothèque C utilisée.)

7.2.1 Méthodes des objets fichiers

Le reste des exemples dans cette section supposera qu'un objet fichier appelé `f` a déjà été créé.

Pour lire le contenu d'un fichier, appeler `f.read(`*taille*`)`, qui lit une certaine quantité de données et les retourne en tant que chaîne de caractères. *taille* est un argument numérique facultatif. Quand *taille* est omis ou négatif, le contenu entier du fichier sera lu et retourné ; c'est votre problème si le fichier est deux fois plus grand que la mémoire de votre machine. Autrement, au plus *taille* octets sont lus et retournés. Si la fin du fichier a été atteinte, `f.read()` renverra une chaîne de caractères vide (`""`).

```
>>> f.read()
'Ceci est le fichier entier.\n'
>>> f.read()
''
```

`f.readline()` lit une seule ligne à partir du fichier ; un caractère de fin de ligne (`\n`) est laissé à l'extrémité de la chaîne de caractères lue, et est seulement omis sur la dernière ligne du fichier si le fichier ne se termine pas par une fin de ligne. Cela rend la valeur de retour non ambiguë ; si `f.readline()` renvoie une chaîne de caractères vide, la fin du fichier a été atteinte, alors qu'une fin de ligne est représentée par `'\n'`, une chaîne de caractères contenant seulement une seule fin de ligne.

```
>>> f.readline()
'Ceci est la première ligne du fichier.\n'
>>> f.readline()
'Deuxième ligne du fichier\n'
>>> f.readline()
''
```

`f.readlines()` renvoie une liste contenant toutes les lignes de données dans le fichier. Si un paramètre optionnel `sizehint` est donné, alors elle lit le nombre d'octets indiqué, plus autant d'octets qu'il en faut pour compléter la dernière ligne commencée, et renvoie la liste des lignes ainsi lues. Cela est souvent utile pour permettre la lecture par lignes efficace, sans devoir charger entièrement le fichier en mémoire. La liste retournée est entièrement faite de lignes complètes.

```
>>> f.readlines()
['Ceci est la première ligne du fichier.\n', 'Deuxième ligne du fichier\n']
```

`f.write(`*chaine*`)` écrit le contenu de *chaine* dans le fichier, en retournant `None`.

```
>>> f.write('Voici un test\n')
```

Pour écrire quelque chose d'autre qu'une chaîne il est nécessaire de commencer par le convertir en chaîne :

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` renvoie un nombre entier donnant la position actuelle dans le fichier associé à l'objet fichier, mesurée en octets depuis le début du fichier. Pour changer la position dans l'objet fichier, employez '`f.seek(`*decalage*`,` *point_depart*`)`'. La position est calculée en ajoutant *decalage* à un point de référence ; le point de référence est choisi par l'argument *point_depart*. Une valeur de 0 pour *point_depart* fait démarrer au début du fichier, 1 utilise la position courante du fichier, et 2 utilise la fin de fichier comme point de référence. *point_depart* peut être omis et prend alors 0 pour valeur par défaut comme point de référence.

```
>>> f=open('/tmp/fichiertravail', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Saute jusqu'au 6ème octet dans le fichier
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Saute jusqu'au 3ème octet avant la fin
>>> f.read(1)
'd'
```

Quand vous en avez terminé avec un fichier, appeler `f.close()` pour le fermer et libérer toutes les ressources système utilisées par le fichier ouvert. Après avoir appelé `f.close()`, les tentatives d'utiliser l'objet fichier échoueront automatiquement.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Les objets fichier ont quelques méthodes supplémentaires, telles que `isatty()` et `truncate()` qui sont moins fréquemment utilisées ; consultez la Library Reference pour un guide complet des objets fichier.

7.2.2 Le module `pickle`

Les chaînes de caractères peuvent facilement être écrites et lues dans un fichier. Les nombres demandent un peu plus d'effort, puisque la méthode `read()` renvoie seulement les chaînes de caractères, qui devront être passées vers une fonction comme `int()`, qui prend une chaîne de caractères comme `'123'` et renvoie sa valeur numérique 123. Cependant, quand vous voulez sauvegarder des types de données plus complexes comme des listes, des dictionnaires, ou des instances de classe, les choses deviennent beaucoup plus compliquées.

Plutôt que faire écrire et déboguer constamment par les utilisateurs le code pour sauvegarder des types de données complexes, Python fournit un module standard appelé `pickle`. C'est un module étonnant qui peut prendre presque n'importe quel objet Python (même quelques formes de code Python !), et le convertir en une représentation sous forme de chaîne de caractères ; ce processus s'appelle *pickling*. Reconstruire l'objet à partir de sa représentation en chaîne de caractères s'appelle *unpickling*. Entre pickling et unpickling, la chaîne de caractères

représentant l'objet a pu avoir été enregistrée dans un fichier ou des données, ou avoir été envoyée à une machine éloignée via une connexion réseau.

Si vous avez un objet `x`, et un objet fichier `f` ouvert en écriture, la voie la plus simple de "pickler" l'objet prend seulement une ligne de code :

```
pickle.dump(x, f)
```

Pour "unpickler" l'objet, si `f` est un objet fichier ouvert en lecture :

```
x = pickle.load(f)
```

(il y a d'autres variantes pour cela, utilisées pour "pickler" beaucoup d'objets ou quand vous ne voulez pas écrire les données "picklées" dans un fichier ; consultez la documentation complète pour `pickle` dans la Library Reference.)

`pickle` est le moyen standard pour enregistrer des objets Python et les réutiliser dans d'autres programmes ou dans une future invocation du même programme ; le terme technique pour cela est la *persistance* d'un objet. Puisque `pickle` est très largement répandu, beaucoup d'auteurs qui écrivent des extensions pour Python prennent soin de s'assurer que de nouveaux types de données tels que des matrices peuvent être correctement "picklés" et "unpicklés".

CHAPITRE

HUIT

Erreurs et exceptions

Jusqu'ici nous avons à peine mentionné les messages d'erreur, mais si vous avez essayé les exemples vous en avez certainement croisé. Il y a (au moins) deux types distincts d'erreurs : les *erreurs de syntaxe* et les *exceptions*.

8.1 Erreurs de syntaxe

Les erreurs de syntaxe, ou erreurs d'interprétation, sont peut-être les formes de messages d'erreur les plus courantes que vous rencontrerez pendant votre apprentissage de Python :

```
>>> while 1 print 'Bonjour'
  File "<stdin>", line 1, in ?
    while 1 print 'Bonjour'
                ^
SyntaxError: invalid syntax
```

L'interpréteur affiche la ligne où l'erreur a eu lieu, et une petite 'flèche' qui marque le premier point de la ligne où l'erreur a été détectée. L'erreur est causée par le (ou du moins détectée au) lexème qui *précède* la flèche : dans l'exemple, l'erreur est détectée au mot-clé `print`, vu qu'il manque un deux-points (' `:`') juste avant. Un nom et un numéro de ligne de fichier sont aussi donnés pour que vous sachiez où aller regarder si les commandes proviennent d'un script.

8.2 Exceptions

Même lorsqu'une instruction ou une expression est syntaxiquement correcte, elle peut provoquer une erreur lorsqu'on essaye de l'exécuter. Les erreurs détectées à l'exécution sont appelées *exceptions* et ne sont pas fatales : vous allez bientôt apprendre à les gérer dans des programmes Python. Néanmoins, la plupart des exceptions n'est pas gérée par un programme et entraîne des messages d'erreur comme ci-dessous :

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: spam
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de plusieurs types, le type est écrit dans le message : dans l'exemple, les types sont `ZeroDivisionError` (erreur de division par zéro), `NameError` (erreur de nom) et `TypeError` (erreur de type). La chaîne de caractères affichée comme type de l'exception est le nom intégré pour l'exception qui a eu lieu. C'est le cas pour toutes les exceptions intégrées, mais pas forcément pour les exceptions définies par l'utilisateur (même si c'est une convention utile). Les noms des exceptions standard sont des identifiants intégrés (et non des mots-clés réservés).

Le reste de la ligne contient des détails, dont l'interprétation dépend du type de l'exception.

La partie précédant le message d'erreur affiche le contexte dans lequel l'exception a eu lieu, sous forme de trace de la pile. En général, elle contient une trace de pile avec des lignes de code source ; toutefois, les lignes lues depuis l'entrée standard ne seront pas affichées.

La *Python Library Reference* donne la liste des exceptions intégrées et leur signification.

8.3 Gestion des exceptions

Il est possible d'écrire des programmes qui prennent en charge des exceptions spécifiques. Regardez l'exemple suivant, qui interroge l'utilisateur jusqu'à ce qu'un entier valide ait été saisi, mais lui permet d'interrompre le programme en utilisant `Control-C` ou une autre combinaison de touches reconnue par le système d'exploitation (il faut savoir qu'une interruption produite par l'utilisateur est signalée en levant l'exception `KeyboardInterrupt`).

```
>>> while 1:
...     try:
...         x = int(raw_input("Veuillez entrer un nombre: "))
...         break
...     except ValueError:
...         print "Aille! Ce n'était pas un nombre valide.  Essayez encore..."
...
```

L'instruction `try` fonctionne ainsi :

- D'abord, la *clause d'essai* (clause *try* : les instructions entre les mots-clés `try` et `except`) est exécutée.
- S'il ne se produit pas d'exception, la *clause d'exception* (clause *except*) est ignorée, et l'exécution du `try` est terminée.
- Si une exception se produit à un moment de l'exécution de la clause d'essai, le reste de la clause try est ignoré. Puis si son type correspond à l'exception donnée après le mot-clé `except`, la clause except est exécutée, puis l'exécution reprend après l'instruction `try`.
- Si une exception se produit qui ne correspond pas à l'exception donnée dans la clause except, elle est renvoyée aux instructions `try` extérieures ; s'il n'y a pas de prise en charge, il s'agit d'une *exception non gérée* et l'exécution est arrêtée avec un message, comme vu précédemment.

Une instruction `try` peut avoir plus d'une clause d'exception, de façon à définir des gestionnaires d'exception différents pour des exceptions différentes. Au plus une clause d'exception sera exécutée. Dans une clause d'exception ne seront gérées que les exceptions survenant dans la clause d'essai correspondante, et non pas celles provenant d'autres clauses d'exception. Une clause d'exception peut nommer plusieurs exceptions dans une liste parenthésée, par exemple :

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

La dernière clause d'exception peut omettre le(s) nom(s) d'exception, et sert alors de passe-partout. Utilisez cela avec une précaution extrême : il est facile de cacher de cette façon une vraie erreur de programmation !

```
import string, sys

try:
    f = open('monfichier.txt')
    c = f.readline()
    i = int(string.strip(c))
except IOError, (errno, strerror):
    print "Erreur(s) E/S: c"  (errno, strerror)
except ValueError:
    print "N'a pas pu convertir la donnée en entier."
except:
    print "Erreur non prévue:", sys.exc_info()[0]
    raise
```

L'instruction `try`…`except` admet une *clause par défaut* (clause *else*) qui doit suivre toutes les clauses d'exception. Elle est utile pour placer le code qui doit être exécuté si la clause d'essai ne déclenche pas d'exception. Par exemple :

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'impossible d'ouvrir', arg
    else:
        print arg, 'comporte', len(f.readlines()), 'lignes'
        f.close()
```

L'utilisation de la clause `else` est meilleure que l'ajout d'un code supplémentaire à la clause `try` parce qu'elle évite d'intercepter de façon accidentelle une exception qui n'a pas été déclenchée par le code qui est protégé par l'instruction `try` … `except`.

Quand une exception survient, elle peut avoir une valeur associée, appelée aussi l'*argument* de l'exception. La présence et le type de l'argument dépendent du type de l'exception.

La clause d'exception peut spécifier une variable après le nom de l'excption (ou la liste). La variable sera liée à une instance de l'exception, avec les arguments rangés dans `instance.args`. Par commodité, l'instance de l'exception définit `__getitem__` et `__str__`, ainsi les arguments peuvent être accédés ou imprimés directement sans avoir à référencer `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception, inst:
...     print type(inst) # the exception instance
...     print inst.args # arguments stored in .args
...     print inst # __str__ allows args to printed directly
...     x, y = inst # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Si une exception a un argument, celui-ci sera affiché dans la dernière partie ('détail') du message pour une exception non gérée.

Les clauses d'exception ne prennent pas en charge uniquement les exceptions qui surviennent dans la clause

d’essai, mais aussi celles qui surviennent dans les fonctions appelées (même de façon indirecte) dans la clause d’essai. Par exemple :

```
>>> def ceci_ne_marche_pas():
...     x = 1/0
...
>>> try:
...     ceci_ne_marche_pas()
... except ZeroDivisionError, detail:
...     print 'Gestion d'erreur à l'exécution:', detail
...
Gestion d'erreur à l'exécution: integer division or modulo
```

8.4 Déclencher des exceptions

L’instruction `raise` permet au programmeur de déclencher une exception. Par exemple :

```
>>> raise NameError, 'Coucou'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: Coucou
```

Le premier argument de `raise` nomme l’exception qui doit être déclenchée. Le second argument (optionnel) spécifie l’argument de l’exception.

Si vous avez besoin de déterminer si une exception est levée mais ne souhaitez pas la traîter, une forme plus simple de l’instruction `raise` permet de re-déclencher l’exception :

```
>>> try:
...     raise NameError, 'SalutToi'
... except NameError:
...     print 'Une exception au vol!'
...     raise
...
Une exception au vol!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: SalutToi
```

8.5 Exceptions définies par l’utilisateur

Les programmes peuvent nommer leurs propres exceptions en créant une nouvelle classe d’exception. Les exceptions devraient typiquement être dérivées de la classe `Exception`, soit directement, soit indirectement. Par exemple :

```
>>> class MonErreur(Exception):
...     def __init__(self, valeur):
...         self.valeur = valeur
...     def __str__(self):
...         return 'self.valeur'
...
>>> try:
...     raise MonErreur(2*2)
... except MonErreur, e:
...     print 'Mon exception s'est produite, valeur:', e.valeur
...
Mon exception s'est produite, valeur: 4
>>> raise MonErreur, 'zut!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MonErreur: 'zut!'
```

Dans cet exemple, la méthode `__init__` de `Exception` a été surchargée. Le nouveau comportement crée simplement l'attribut `valeur`. Cela remplace le comportement par défaut qui est de créer l'attribut `args`.

Des classes d'exceptions peuvent être définies qui font tout ce que d'autres classes peuvent faire, mais sont généralement gardées assez simples, offrant souvent seulement un certain nombre d'attributs qui permettent de donner des informations sur l'erreur qui soient extraits par les gestionaires pour cette exception. Quand on crée un module qui peut déclencher plusieurs erreurs distinctes, une pratique courante est de créer une classe de base pour les exceptions définies par ce module, et des sous-classes de celle-ci pour créer des classes d'exceptions spécifiques pour différentes conditions d'erreurs :

```
class Erreur(Exception):
    """Classe de base pour les exceptions dans ce module."""
    pass

class EntreeErreur(Erreur):
    """Exception déclenchée pour les erreurs dans l'entrée.

    Attributs:
        expression -- expression d'entrée sur laquelle l'erreur s'est produite
        message -- explication de l'erreur
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionErreur(Erreur):
    """Déclenchée quand une opération tente d'effectuer une transition
    d'état qui n'est pas autorisée.

    Attributs:
        precedent -- état au début de la transition
        suivant -- nouvel état essayé
        message -- explication de pourquoi cette transition spécifique
                   n'est pas autorisée
    """

    def __init__(self, precedent, suivant, message):
        self.precedent = precedent
        self.suivant = suivant
        self.message = message
```

La plupart des exceptions sont définies avec des noms qui se terminent en "Error," de façon similaire au nommage des exceptions standard.

Plusieurs modules standard définissent leurs propres exceptions pour rapporter les erreurs qui peuvent survenir dans les fonctions qu'ils définissent. Plus d'informations au sujet des classes sont données dans le chapitre 9, "Classes."

8.6 Définir les actions de nettoyage

L'instruction `try` admet une autre clause optionnelle qui permet de définir les actions de nettoyage qui doivent être exécutées impérativement. Par exemple :

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Adieu, monde!'
...
Adieu, monde!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Une *clause de finalisation* (clause *finally*) est exécutée qu'une exception ait eu lieu ou non dans la clause d'essai. Si une exception a été déclenchée, elle est déclenchée à nouveau après l'exécution de la clause de finalisation. La clause de finalisation est aussi exécutée "en sortant" lorsque l'instruction `try` est interrompue par les instructions `break` ou `return`.

Le code dans la clause finally est utile pour libérer des ressources externes (telles que des fichiers ou des connections réseau), indépendamment du fait que l'utilisation de la ressource ait été réussie.

Une instruction `try` doit avoir ou bien une ou plusieurs clauses d'exception, ou bien une clause de finalisation, mais pas les deux.

CHAPITRE

NEUF

Classes

Le mécanisme de classe en Python permet d'introduire les classes avec un minimum de syntaxe et sémantique nouvelles. C'est un mélange des mécanismes de classe de C++ et Modula-3. Comme les modules, les classes en Python n'installent pas de barrière absolue entre la définition et l'utilisateur, mais appellent plutôt à la politesse de l'utilisateur pour éviter l'"effraction de la définition". Les caractéristiques les plus importantes des classes sont pleinement présentes : le mécanisme d'héritage permet la multiplicité des classes de base, une classe dérivée peut surcharger n'importe quelle méthode de sa ou ses classes de base, une méthode peut appeler une méthode de sa classe de base avec le même nom. Les objets peuvent contenir un nombre arbitraire de données privées.

Dans la terminologie du C++, tous les membres d'une classe (dont les données membres) sont *publics*, et toutes les fonctions membres sont *virtuelles*. Il n'y a pas de constructeurs ou de destructeurs particuliers. Comme en Modula-3, il n'y a pas de raccourcis pour faire référence aux membres d'un objet à partir de ses méthodes : une méthode est déclarée avec un premier argument explicite qui représente l'objet, qui est fourni implicitement à l'appel. Comme en Smalltalk, les classes sont elles-mêmes des objets, mais dans un sens plus large : en Python, tous les types de données sont des objets. Cela fournit la sémantique pour l'importation et le renommage. Mais, comme en C++ et en Modula-3, les types intégrés ne peuvent pas être utilisés comme classes de base pour des extensions par l'utilisateur. En plus, comme en C++ mais contrairement à Modula-3, la plupart des opérateurs intégrés qui ont une syntaxe particulière (opérateurs arithmétiques, indiçage, etc.) peuvent être redéfinis pour des instances de classe.

9.1 Un mot sur la terminologie

A défaut d'une terminologie universellement acceptée pour parler des classes, j'utiliserai à l'occasion des termes de Smalltalk et C++. (J'utiliserais volontiers des termes de Modula-3, puisque sa sémantique orientée objet est plus proche de celle de Python que celle de C++, mais j'imagine que peu de lecteurs ont entendu parler de ce langage.)

Les objets possèdent une individualité, et des noms multiples (dans des portées multiples) peuvent être liés au même objet. Cela s'appelle aliasing dans d'autres langages. On ne le remarque pas de prime abord dans Python, et on peut l'ignorer pour le traitement des types de base non modifiables (nombres, chaînes de caractères, tuples). Néanmoins, l'aliasing a un effet (voulu !) sur la sémantique du code Python qui met en jeu des objets modifiables comme listes, dictionnaires et la plupart des types représentant des entités à l'exception du programme (fichiers, fenêtres, etc.). Cela est mis à profit dans les programmes, puisque les alias se comportent comme des pointeurs à plusieurs points de vue. Par exemple, le passage en paramètre d'un objet n'est pas coûteux puisque seul un pointeur est transmis par l'implémentation ; et si une fonction modifie un objet reçu en argument, l'appelant verra la modification — ce qui élimine le besoin d'avoir deux mécanismes de passage d'arguments comme en Pascal.

9.2 Les portées et les espaces de noms en Python

Avant d'introduire les classes, je dois vous dire quelques mots sur les règles de portée en Python. Les définitions de classe jouent astucieusement avec les espaces de noms, et vous devez savoir comment fonctionnent les portées et les espaces de noms pour comprendre ce qui se passe. En fait, la connaissance de ce sujet est utile à tout programmeur Python avancé.

D'abord quelques définitions.

Un *espace de noms* (*name space*) est une relation entre des noms et des objets. La plupart des espaces de noms sont actuellement implémentés comme des dictionnaires, mais cela n'est pas visible (sauf sur les performances peut-être) et pourrait changer dans le futur. Quelques exemples d'espaces de noms : l'ensemble des noms intégrés (les fonctions telles que `abs()`, et les noms d'exception intégrés) ; les noms globaux dans un module ; les noms locaux au cours d'un appel de fonction. En un sens, l'ensemble des attributs d'un objet constitue aussi un espace de noms. La chose importante à savoir sur les espaces de noms est qu'il n'y a absolument aucune relation entre les noms contenus dans les différents espaces de noms ; par exemple, deux modules différents peuvent définir tous les deux une fonction "maximise" sans confusion possible — les utilisateurs des modules doivent préfixer par le nom du module à l'utilisation.

Au passage, j'utilise le mot *attribut* (*attribute*) pour n'importe quel nom qui suit un point — par exemple, dans l'expression `z.real`, `real` est un attribut de l'objet `z`. Strictement parlant, les références à des noms dans des modules sont des attributs ; dans l'expression `nommod.nomfonc`, `nommod` est un objet module et `nomfonc` en est un attribut. Dans ce cas, il y a un rapport direct entre les attributs du module et les noms globaux définis dans le module : ils partagent le même espace de noms ![1]

Les attributs peuvent être en lecture seule ou bien modifiables. Dans ce cas, on peut affecter des valeurs à des attributs. Les attributs d'un module sont modifiables : vous pouvez faire '`nommod.la_reponse = 42`'. Les attributs modifiables peuvent aussi être effacés avec l'instruction `del`. Par exemple, '`del nommod.la_reponse`' enlèvera l'attribut `la_reponse` de l'objet nommé par `nommod`.

Les espaces de noms sont créés à des moments différents et ont des durées de vie différentes. L'espace de noms qui contient les noms intégrés est créé au lancement de l'interpréteur Python, et n'est jamais effacé. L'espace de noms global pour un module est créé quand la définition du module est chargée ; normalement, les espaces de noms des modules vivent jusqu'à la mort de l'interpréteur. Les instructions exécutées à l'invocation de l'interpréteur par le niveau supérieur, qu'elles soient lues depuis un fichier ou entrées de façon interactive, sont considérées comme faisant partie d'un module appelé `__main__`, elles ont donc leur propre espace de noms global. (En fait, les noms intégrés font aussi partie d'un module appelé `__builtin__`.)

L'espace de noms local à une fonction est créé quand celle-ci est appelée et il est effacé quand la fonction se termine ou déclenche une exception qui n'est pas gérée dans la fonction. (En fait, oublié décrit mieux ce qui se passe vraiment.) Evidemment, les appels récursifs ont chacun leur propre espace de noms.

Une *portée* (*scope*) est une région textuelle d'un programme Python dans laquelle un espace de noms est directement accessible. "Directement accessible" veut dire qu'une référence non qualifiée à un nom cherchera ce nom dans cet espace de noms.

Bien qu'elles soient déterminées statiquement, les portées sont utilisées dynamiquement. A n'importe quel moment de l'exécution, exactement trois portées imbriquées sont utilisées (exactement trois espaces de noms sont accessibles directement) : la portée immédiate, qui est explorée en premier, contient les noms locaux, la portée intermédiaire, explorée ensuite, contient les noms globaux du module courant, et la portée extérieure (explorée en dernier) correspond à l'espace de noms contenant les noms intégrés.

Si un nom est déclaré global alors les références et affectations le concernant sont directement adressées à la portée qui porte les noms globaux du module. Autrement, toutes les variables trouvées ailleurs que dans la portée la plus intérieure sont en lecture seulement.

Normalement, la portée locale fait référence aux noms de la fonction courante (textuellement). En dehors des fonctions, la portée locale fait référence au même espace de noms que la portée globale : l'espace de noms du module. Les définitions de classe placent encore un autre espace de noms dans la portée locale.

Il est important de voir que les portées sont déterminées de façon textuelle : la portée globale d'une fonction définie dans un module est l'espace de noms de ce module, peu importe d'où ou à travers quel alias cette fonction est appelée. D'un autre côté, la recherche de noms elle-même est effectuée dynamiquement, à l' exécution — toutefois, la définition du langage tend à évoluer vers la résolution statique de noms, au moment de la "compilation", alors ne vous basez pas sur la résolution dynamique de noms ! (En fait, les variables locales sont déjà déterminées de façon statique.)

[1] Sauf pour une chose. Les objets module possèdent un attribut secret en lecture exclusive qui s'appelle `__dict__` et qui renvoie le dictionnaire utilisé pour implémenter l'espace de noms du module ; le nom `__dict__` est un attribut mais pas un nom global. Evidemment, l'utiliser casse l'abstraction de l' implémentation des espaces de noms, et son usage doit être restreint à des choses telles que les débogueurs post-mortem.

Un point titilleux de Python est que les affectations se font toujours dans la portée immédiate. L'affectation ne copie pas de données — elle ne fait qu'affecter un nom à un objet. Cela est vrai aussi de l'effacement : l'instruction '`del x`' enlève le lien vers `x` de l'espace de noms référencé par la portée locale. En fait, toute opération qui introduit de nouveaux noms utilise la portée locale : en particulier, les instructions d'importation et les définitions de fonction lient le nom du module ou de la fonction à la portée locale. (L'instruction `global` peut être utilisée pour indiquer que certaines variables vivent dans la portée globale.)

9.3 Une première approche des classes

Les classes introduisent un peu de syntaxe nouvelle, trois nouveaux types d'objet, et quelques points de sémantique supplémentaires.

9.3.1 Syntaxe de la définition de classe

La forme la plus simple de définition de classe ressemble à ceci :

```
class NomClasse:
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

Les définitions de classe, comme les définitions de fonction (instructions `def`) doivent être exécutées pour entrer en effet. (Vous pourriez placer une définition de classe dans une branche d'une instruction `if`, ou à l'intérieur d'une fonction.)

Dans la pratique, les instructions à l'intérieur d'une définition de classe seront souvent des définitions de fonction, mais d'autres instructions sont acceptées, et parfois s'avèrent utiles — plus de détails sur le sujet ci-dessous. Les définitions de fonction à l'intérieur d'une classe ont normalement une forme particulière de liste d'arguments, dictée par les conventions d'appel de méthode — cela aussi est expliqué plus loin.

A l'entrée d'une définition de fonction, un nouvel espace de noms est créé et utilisé comme portée locale — ainsi, toute affectation de variables rentre dans cet espace de noms. En particulier, les définitions de fonctions y rattachent le nom des nouvelles fonction.

Lorsque la définition de la classe est achevée de façon normale (par la fin), un *objet classe* (*class object*) est créé. Il s'agit essentiellemnt d'un enrobage autour du contenu de l'espace de noms créé par la définition de classe ; nous verrons davantage de caractéristiques des objets classes dans la section suivante. La portée locale d'origine (celle en cours avant le début de la définition de classe) est réinstallée, et l'objet classe est lié ici au nom donné dans l'en-tête de la définition de classe (`NomClasse` dans l'exemple).

9.3.2 Objets classes

Les objets classe admettent deux sortes d'opérations : la référenciation des attributs et l'instanciation.

Les *références aux attributs* (*attribute references*) utilisent la syntaxe standard utilisée pour toutes les références d'attribut en Python : `obj.nom`. Les noms d'attribut valides sont ceux qui étaient dans l'espace de noms de la classe quand l'objet classe a été créé. Donc, si la définition de classe ressemble à :

```
class MaClasse:
    "Une classe simple pour exemple''
    i = 12345
    def f(self):
        return 'bonjour'
```

alors `MaClasse.i` et `MaClasse.f` sont des références d'attribut valides, qui renvoient un entier et un objet fonction, respectivement. On peut affecter une valeur aux attributs de classe, donc vous pouvez changer la valeur de `MaClasse.i` par affectation. `__doc__` est un attribut valide, en lecture exclusive, qui renvoie la docstring correspondant à la classe : `"Une classe simple pour exemple"`).

L'*instantiation* de classe utilise la notation d'appel de fonction. Faites comme si l'objet classe était une fonction sans paramètres qui renvoie une instance nouvelle de la classe. Par exemple, (avec la classe précédente) :

```
x = MaClasse()
```

crée une nouvelle *instance* de la classe et affecte cet objet à la variable locale `x`.

L'opération d'instanciation ("appeller" un objet classe) crée un objet vide. De nombreuses classes aiment bien créer les objets dans un état initial connu. Ainsi une classe peut définir une méthode spéciale nommée `__init__()`, comme ceci :

```
    def __init__(self):
        self.donnee = []
```

Quand une classe définit une méthode `__init__()`, l'instanciation de la classe appelle automatiquement `__init__()` pour l'instance de la classe nouvellement créée. Ainsi, dans cet exemple, une instance nouvelle, initialisée, peut être obtenue par :

```
x = MaClasse()
```

Bien-sûr, la méthode `__init__()` peut avoir des arguments pour offrir plus de souplesse. Dans ce cas, les arguments fournis à l'opérateur d'instanciation de la classe sont passés à `__init__()`. Par exemple,

```
>>> class Complexe:
...     def __init__(self, partiereelle, partieimaginaire):
...         self.r = partiereelle
...         self.i = partieimaginaire
...
>>> x = Complexe(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objets instances

Que peut-on faire avec les objets instances ? Les seules opérations acceptées par des objets instance sont des références à leurs attributs. Il y a deux sortes de noms d'attributs valides.

J'appellerai la première *données attributs* (*data attributes*). Ils correspondent aux "variables d'instance" (instance variables) en Smalltalk, et aux "données membres" (data members) en C++. Les données attributs n'ont pas besoin d'être déclarées ; comme les variables locales, elles apparaissent lorsqu'on leur affecte une valeur pour la première fois. Par exemple, si `x` est l'instance de `MaClasse` créée précédemment, le morceau de code suivant affichera la valeur `16`, sans laisser de trace :

```
x.compteur = 1
while x.compteur < 10:
    x.compteur = x.compteur * 2
print x.compteur
del x.compteur
```

La seconde sorte de référence d'attribut acceptée par les objets instance sont les *méthodes* (*methods*). Une méthode est une fonction qui "appartient" à un objet. (En Python, le terme méthode n'est pas exclusif aux instances de

classe : d'autres types d'objet peuvent avoir des méthodes. Par exemple, les objets liste ont des méthodes appelées append, insert, remove, sort, etc. Néanmoins, dans ce qui suit, nous allons utiliser le terme méthode pour désigner exclusivement les méthodes d'un objet instance de classe, sauf mention explicite.)

Les noms de méthodes valides pour un objet instance dépendent de sa classe. Par définition, tous les attributs d'une classe qui sont des fonctions (définies par l'utilisateur) définissent des méthodes correspondantes pour les instances. Ainsi, dans notre exemple, `x.f` est une référence valide à une méthode, puisque `MaClasse.f` est une fonction, mais `x.i` ne l'est pas, vu que `MaClasse.i` ne l'est pas. Toutefois `x.f` n'est pas la même chose que `MaClasse.f` — c'est un *objet méthode* (*method object*), et non pas un objet fonction.

9.3.4 Objets méthodes

D'habitude, une méthode est appelée de façon directe :

```
x.f()
```

Dans notre exemple, cela renverrait la chaîne `'salut monde'`. Or, il n'est pas nécessaire d'appeler une méthode tout de suite : `x.f` est un objet méthode, il peut être rangé quelque part et être appelé plus tard, par exemple :

```
xf = x.f
while 1:
    print xf()
```

continuera à afficher '`bonjour`' jusqu'à la fin des temps.

Que se passe-t-il exactement lorsqu'une méthode est appelée ? Vous avez peut-être remarqué que `x.f()` a été appelée sans argument ci-dessus, alors que la définition de fonction pour `f` en spécifiait un. Qu'est-il arrivé à l'argument ? On se doute bien que Python déclenche une exception quand une fonction qui requiert un argument est appelée sans argument — même si l'argument n'est pas effectivement utilisé...

En fait, vous avez peut-être deviné la réponse : la particularité des méthodes est que l'objet est passé comme premier argument à la fonction. Dans notre exemple, l'appel `x.f()` est l'équivalent exact de `MaClasse.f(x)`. En général, appeler une méthode avec une liste de *n* arguments équivaut à appeler la fonction correspondante avec une liste d'arguments qui est le résultat de l'insertion de l'objet avant le premier argument.

Si vous n'avez toujours pas compris comment fonctionnent les méthodes, un regard sur l'implémentation va peut-être clarifier les choses. Lorsqu'un attribut d'une instance est référencé et qu'il n'est pas une donnée attribut, une recherche est entamée dans sa classe. Si le nom correspond à un attribut de classe valide qui est un objet fonction, un objet méthode est créé en empaquetant (des pointeurs sur) l'objet instance et l'objet fonction trouvé dans un objet abstrait : c'est l'objet méthode. Lorsque l'objet méthode est appelé avec une liste d'arguments, il est depaqueté, une nouvelle liste d'arguments est construite à partir de l'objet instance et de la liste d'arguments originelle, puis l'objet fonction est appelé avec cette nouvelle liste d'arguments.

9.4 Quelques remarques

Les données attributs écrasent les méthodes de même nom ; pour éviter des conflits de noms accidentels, qui peuvent causer des bogues difficiles à trouver dans des programmes conséquents, il est sage d'utiliser une convention qui minimise les chances de conflit. Des conventions possibles comprennent la mise en majuscules les noms des méthodes, préfixer les noms des données attributs avec une même courte chaîne de caractères (peut-être un simple tiret-bas), ou utiliser des verbes pour les méthodes et des substantifs pour les données.

Les données attributs peuvent être référencées par des méthodes aussi bien que par les utilisateurs ordinaires ("clients") d'un objet. Autrement dit, les classes ne sont pas utilisables pour implémenter des types abstraits purs. En fait, rien dans Python ne permet d'assurer le secret des données — tout est basé sur des conventions. (D'un autre côté, l'implémentation de Python, écrite en C, peut cacher complètement les détails d'implémentation et de contrôle d'accès à un objet si besoin est ; cela peut être utilisé par les extensions de Python écrites en C.)

Les clients doivent utiliser les données attributs avec précaution — ils peuvent bouleverser des invariants entretenus par les méthodes en écrasant leurs données attributs. Notez bien que les clients peuvent rajouter des données attributs de leur cru à un objet instance sans affecter la validité des méthodes, pourvu que les conflits de noms soient évités — là encore, une convention de nommage peut éviter bien des migraines.

Il n'y a pas de raccourci pour faire référence à des données attributs (ou à d'autres méthodes !) à partir d'une méthode. Je trouve que cela augmente en fait la lisibilité des méthodes : il n'y a aucune chance de confondre les variables locales et les variables d'instance lorsqu'on examine une méthode.

Par convention, le premier argument d'une méthode est souvent appelé `self`. Ce n'est qu'une convention : le mot `self` n'a absolument aucun sens spécial pour Python. (Remarquez, tout de même, que si votre code ne suit pas la convention, il peut se révéler moins lisible par d'autres programmeurs Python, et il est aussi concevable qu'un *explorateur de classes* soit écrit, qui se base sur cette convention.)

Tout objet fonction qui est aussi un attribut d'une classe définit une méthode pour les instances de cette classe. Il n'est pas nécessaire que la définition de la fonction soit textuellement comprise dans la définition de la classe : affecter un objet fonction à une variable locale dans la classe marche aussi. Par exemple :

```
# Fonction définie en dehors de la classe
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'bonjour'
    h = g
```

Maintenant, `f`, `g` et `h` sont tous trois des attributs de la classe `C` qui font référence à des objets fonctions, et par conséquent ils sont tous les trois des méthodes des instances de `C` — `h` équivaut exactement à `g`. Remarquez que cette pratique ne sert souvent qu'à embrouiller le lecteur du programme.

Les méthodes peuvent appeler d'autres méthodes en utilisant les attributs méthodes de l'argument `self` :

```
class Sac:
    def vider(self):
        self.donnees = []
    def ajouter(self, x):
        self.donnees.append(x)
    def ajouterdoublon(self, x):
        self.ajouter(x)
        self.ajouter(x)
```

Les méthodes peuvent faire référence à des noms globaux de la même façon que les fonctions ordinaires. La portée globale associée à une méthode est celle du module qui contient la définition de la classe. (La classe elle-même ne sert jamais de portée globale !) Alors qu'on trouve rarement de bonnes raisons pour utiliser des données globales dans une méthode, il y a plusieurs utilisations légitimes de la portée globale : ne serait-ce que le fait que les fonctions et les modules importés dans la portée globale peuvent être utilisés par les méthodes, de même que les fonctions et les classes qui y sont définies. D'habitude, la classe qui contient la méthode est elle-même définie dans cette portée globale, et dans la section suivante nous allons voir quelques bonnes raisons pour lesquelles une méthode pourrait vouloir faire référence à sa propre classe !

9.5 Héritage

Bien sûr, une caractéristique du langage ne serait pas digne du mot "classe" si elle ne permettait pas l'héritage. La syntaxe pour définir une classe dérivée ressemble à ceci :

```
class NomClasseDerivee(NomClasseDeBase):
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

Le nom `NomClasseDeBase` doit être défini dans une portée contenant la définition de la classe dérivée. A la place d'un nom de classe de base, une expression est acceptée. Cela est utile lorsque la classe de base est définie dans un autre module,

```
class NomClasseDerivee(nommod.NomClasseDeBase):
```

L'exécution d'une définition de classe dérivée se déroule comme pour une classe de base. Quand l'objet classe est construit, la classe de base est mémorisée. Cela est employé dans la résolution des références d'attribut : si l'attribut demandé n'est pas trouvé dans la classe, il est recherché dans la classe de base. Cette règle est employée récursivement si la classe de base est elle-même dérivée depuis une autre classe.

Il n'y a rien de spécial dans l'instantiation d'une classe dérivée : `NomClasseDerivee()` crée une nouvelle instance de la classe. Les références aux méthodes sont résolues ainsi : l'attribut est recherché dans la classe correspondante, en descendant la chaîne des classes de base si besoin est, et la référence à la méthode est valide si cette recherche aboutit à un objet fonction.

Les classe dérivées peuvent redéfinir les méthodes de leurs classes de base. Puisque les méthodes ne jouissent pas de privilèges particuliers lorsqu'elles appellent d'autres méthodes du même objet, la méthode d'une classe de base qui appelle une autre méthode définie dans la même classe de base peut en définitive appeler une méthode d'une classe dérivée qui a redéfini cette méthode. (Pour les programmeurs C++ : toutes les méthodes en Python sont des "fonctions virtuelles".)

Une méthode d'une classe dérivée qui redéfinit une fonction peut en fait vouloir étendre et non pas remplacer la méthode de la classe de base de même nom. Il y un moyen simple d'appeler la méthode de la classe de base directement : simplement appelez '`NomClasseDeBase.nommethode(self, arguments)`'. Cela peut parfois s'avérer utile pour les clients aussi. (Remarquez que ça ne marche que si la classe de base est définie ou importée dans la portée globale.)

9.5.1 Héritage multiple

Python supporte aussi une forme limitée d'héritage multiple. Une définition de classe avec plusieurs classes de base ressemble à :

```
class NomClasseDerivee(Base1, Base2, Base3):
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

La seule règle permettant d'expliquer la sémantique de l'héritage multiple est la règle de résolution utilisée pour les références aux attributs. La résolution se fait en profondeur d'abord, de gauche à droite. Donc, si un attribut n'est pas trouvé dans `NomClasseDerivee`, il est cherché dans `Base1`, puis (récursivement) dans les classes de base de `Base1`, et seulement s'il n'y est pas trouvé, il est recherché dans `Base2`, et ainsi de suite.

(Pour certains la recherche en largeur d'abord — chercher dans `Base2` est `Base3` avant les classes de base de `Base1` — semble plus naturelle. Pourtant, cela nécessite que vous sachiez si un attribut particulier de `Base1` est défini dans `Base1` ou dans une de ses classes de base avant de pouvoir considérer les conflits de nom avec `Base2`. La règle en profondeur d'abord ne fait pas de différence entre les attributs directs et hérités de `Base1`.)

Il est clair que l'utilisation banalisée de l'héritage multiple est un cauchemar de maintenance, étant donné que Python se base sur des conventions pour éviter les conflits de noms accidentels. Un problème bien connu de l'héritage multiple est celui d'une classe dérivée de deux autres classes qui ont une même classe de base en commun. S'il reste facile de voir ce qui se passe dans ce cas (l'instance aura une seule copie des "variables d'instance" ou des données attributs utilisés par la classe de base commune), il n'est pas clair que cette sémantique soit utile de quelque façon que ce soit.

9.6 Variables privées

Il y a un support limité pour des identificateurs privés dans une classe. Tout identificateur de la forme `__spam` (au moins deux tirets-bas au début, au plus un tiret-bas à la fin) est maintenant textuellement remplacé par `_nomclasse__spam`, où `nomclasse` est le nom de classe courant, duquel les tirets-bas de début on été enlevés. Ce brouillage (mangling) est réalisé indépendamment de la position syntaxique de l'identificateur, donc il peut être utilisé pour définir des variables de classe et d'instance privées, des méthodes, des globales, et même pour enregistrer des variables d'instance privées de cette classe dans des instances d'*autres* classes. Le nom brouillé peut être tronqué s'il dépasse 255 caractères. En dehors des classes, ou lorsque le nom de la classe ne contient que des tirets-bas, le brouillage n'a pas lieu.

Le brouillage de noms permet aux classes de définir simplement des variables d'instance et des méthodes "privées", sans avoir à se préoccuper des variables d'instance définies par des classes dérivées, ou des problèmes avec des variables d'instance définies en dehors de la classe. Remarquez que les règles de brouillage ont été dessinées surtout pour éviter des accidents ; il reste possible d'accéder ou de modifier une variable considérée comme privée. Cela peut être utile dans des circonstances particulières telles que dans le débogueur, et c'est une des raisons pour lesquelles ce trou n'est pas comblé. (Petite bogue : dériver une classe en utilisant le même nom de classe permet l'utilisation des variables privées de la classe de base.)

Remarquez que le code passé en argument à `exec`, `eval()` ou `evalfile()` ne considère pas le nom de classe de la classe appelante comme étant le nom de classe courant ; c'est un effet similaire à celui de l'instruction `global`, limité à du code qui a été compilé en même temps. La même restriction s'applique à `getattr()`, `setattr()` et `delattr()`, de même qu'aux références directes à `__dict__`.

9.7 En vrac

Il est parfois utile de disposer d'un type de données semblable au "record" du Pascal ou au "struct" du C, pour lier quelques données nommées. Une définition de classe vide peut servir à cela :

```
class Employe:
    pass

john = Employe() # Crée un enregistrement vide d'Employe

# Remplit les champs de l'enregistrement
john.nom = 'John Doe'
john.dept = 'computer lab'
john.salaire = 1000
```

Un bout de code Python qui attend des données d'un certain type abstrait peut souvent recevoir à la place une classe qui simule les méthodes de ce type de données. Par exemple, si vous avez une fonction qui met en forme des données issues d'un objet fichier, vous pouvez définir une classe avec des méthodes `read()` et `readline()` qui prend les données d'un tampon et passer celui-ci comme argument.

Les objets méthode d'instance possèdent eux-mêmes des attributs : `m.im_self` est l'objet duquel la méthode est instance, et `m.im_func` est l'objet fonction correspondant à la méthode.

9.8 Les exceptions sont des classes aussi

Les exceptions définies par l'utilisateur sont construites comme des classes. En utilisant ce mécanisme, on peut définir des hiérarchies extensibles d'exceptions.

Il y a deux formes sémantiques valides pour l'instruction `raise` :

```
raise Classe, instance

raise instance
```

Dans la première forme, `instance` doit être une instance de `Classe` ou d'une de ses classes dérivées. La seconde forme est un raccourci pour

```
raise instance.__class__, instance
```

Une clause d'exception peut lister des classes et des chaînes de caractères. Une classe dans une clause d'exception est compatible avec une exception si celle ci est la même classe ou une classe qui en est dérivée (mais pas en sens inverse — une clause d'exception qui liste une classe dérivée n'est pas compatible avec une classe de base). Par exemple, le code suivant affichera B, C, D, dans cet ordre :

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Remarquez que si les clauses d'exception avaient été inversées (‘`except B`’ en premier), le code aurait affiché B, B, B — c'est la première clause except qui convient qui est exécutée.

Lorsqu'un message d'erreur est affiché pour une exception non gérée qui est une classe, le nom de la classe est affiché, puis deux-points, un espace, et finalement, l'instance convertie en chaîne de caractères à travers la fonction intégrée `str()`.

9.9 Itérateurs

A l'heure qu'il est vous avez probablement remarqué que la plupart des conteneurs peuvent être parcourus en utilisant une instruction `for` :

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Ce style d'accès est clair, concis et pratique. L'utilisation d'itérateurs imprègne Python et l'unifie. En coulisse, l'instruction `for` appelle `iter()` sur l'objet conteneur. Cette fonction renvoie un objet itérateur définissant une méthode `next()` qui accède les éléments dans le conteneur, un à la fois. Lorsqu'il n'y a plus d'éléments, `iter()` lève une exception `StopIteration` qui dit à la boucle `for` de se terminer. L'exemple que voici montre comment cela fonctionne :

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel
    it. next()
StopIteration
```

Ayant vu la mécanique qu'il y a derrière le protocole d'un itérateur, il est facile d'ajouter un comportement d'itérateur à vos classes. Définissez une méthode `__iter__()` qui renvoie un objet ayant une méthode `next()`. Si la classe définit `next()`, alors `__iter__()` peut se limiter à renvoyer `self` :

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s
```

9.10 Generateurs

Les générateurs sont un outil simple et puissant pour créer des itérateurs. Ils sont écrits comme des fonctions ordinaires mais utilisent l'instruction `yield` chaque fois qu'ils veulent renvoyer une donnée. Chaque fois que `next()` est appelé, le générateur reprend là où il s'était interrompu (il mémorise les valeurs des données et quelle instruction a été exécutée en dernier). L'exemple suivant montre à quel point il est trivial de créer un générateur :

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Tout ce qui peut être fait avec des générateurs peut aussi être fait avec des itérateurs basés sur des classes comme décrit dans la section précédente. Ce qui rend les générateurs si compacts est le fait que les méthodes (__iter__()) et (next()) sont créées automatiquement.

Un autre point clé est que les variables locales et l'état de l'exécution sont sauvés automatiquement entre les appels. Cela rend la fonction facile à écrire et beaucoup plus claire qu'une approche utilisant les variables d'instance comme `self.index` et `self.data`.

En plus de la création de méthodes et la sauvegarde de l'état automatiques, lorsque les générateurs terminent ils lèvent automatiquement l'exception `StopIteration`. Ensemble, ces particularités facilitent la création d'itérateurs sans plus d'effort que l'écriture d'une fonction ordinaire.

9.11 Expressions générateurs

Certains générateurs simples peuvent être codés succinctement comme des expressions qui utilisent une syntaxe similaire à celle des *list comprehensions* mais avec des parenthèses au lieu de crochets. Ces expressions sont destinées aux situations où le générateur est immédiatement utilisé par une fonction englobante. Les expressions générateurs sont plus compactes mais moins souples que les définitions de générateurs complètes et ont tendance à être plus économes en mémoire que les *list comprehensions* équivalentes.

Exemples :

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

CHAPITRE

DIX

Petit tour dans la bibliothèque standard

10.1 Interface avec le système d'exploitation

Le module `os` fournit des tas de fonctions pour interagir ave le système d'exploitation :

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # Return the current working directory
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Assurez-vous d'utiliser le style "`import os`" au lieu de "`from os import *`". Cela amènerait `os.open()` à faire de l'ombre à la fonction intégrée `open()` qui aopère de manière très différente.

Les fonctions intégrées `dir()` et `help()` sont utiles comme aides interactives pour travailler avec de grand modules comme `os` :

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Pour les tâches courantes de gestion de fichiers et répertoires, la module `shutil` fournit une interface de haut niveau facile à utiliser :

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2 Fichiers et jockers

Le module `glob` fournit une fonction pour construire des listes de fichiers à partir de recherches avec jockers dans des répertoires :

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Arguments de la ligne de commande

Les scripts utilitaires requièrent souvent le traitement des arguments de la ligne de commande. Ces arguments sont stockés sous forme de liste dans l'attribut `argv` du module `sys`. Par exemple, la sortie suivante résulte de l'exécution de `demo.py one two three` sur la ligne de commande :

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Le module `getopt` traite `sys.argv` en utilisant les mêmes conventions que la fonction UNIX `getopt()`. Un traitement de la ligne de commande plus puissant et flexible est fourni par le module `optparse`.

10.4 Redirection de la sortie et terminaison du programme

Le module `sys` possède également des attributs pour reprsenter `stdin`, `stdout` et `stderr`. Ce dernier est utile pour rendre les messages d'avertissent et d'erreur visibles même lorsque `stdout` a été rédirigé :

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

La manière la plus directe de terminer un script consiste à utiliser `sys.exit()`.

10.5 Appariement de chaînes

Le module `re` fournit des outils d'expressions régulières pour un traitement avancé des chaînes. Pour des appariements et des traitements complexes, les expressions régulières offrent ds solutions succinctes et optimisées :

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Lorsque seules des opérations simples sont requises, les méthodes des chaînes sont préférables car elles sont plus simples à lire et à débugger :

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Mathématiques

Le module `math` donne accès aux fonctions mathématiques à virgule flottante de la bibliothèque C sous-jacente :

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Le module `random` fournit des outils pour faire des sélections aléatoires :

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()                   # random float
0.17970987693706186
>>> random.randrange(6)               # random integer chosen from range(6)
4
```

10.7 Accès à Internet

Il y a un certain nombre de modules pour accéder à Internet et pour traiter les protocoles de l'Internet. Deux des plus simples sont `urllib2` pour récupérer des données depuis des url et `smtplib` pour envoyer du courrier :

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
... if 'EST' in line:          # look for Eastern Standard Time ...
print line <BR>Nov. 25, 09:43:32 PM EST
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
"""To: jcaesar@example.org
From: soothsayer@example.org

Beware the Ides of March.
""")
>>> server.quit()
```

10.8 Dates et heures

Le module `datetime` fournit des classes pour manipuler les dates et les heures aussi bien de manière simple que de manière complexe. Bien que l'arithmétique des dates et des heures est supportée, le centre d'attention de l'implémentation a été porté sur l'efficacité de l'extraction des membres en vue de la mise en forme et du traitement de l'affichage. Ce module supporte aussi les objets liés aux fuseaux horaires.

```
# dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compression de données

Les formats communs pour archiver et compresser des données sont supportés par des modules, dont : `zlib`, `gzp`, `bz2`, `zipfile` et `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Mesure des performances

Certains utilisateurs de Python nourrissent un profond intérêt pour la connaissance des performances relatives de différentes approches d'un même problème. Python fournit un outil de mesure qui répond immédiatement à ces questions.

Par exemple, il peut être tentant d'utiliser l'emballage et le déballage des n-uplets à la place de l'approche traditionnelle pour échanger des arguments. Le module `timeit` montre rapidement un modeste avantage en performances :

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

Contrastant avec la finesse de la granularité de `timeit`, les modules `profile` et `pstats` fournissent des outils pour identifier les sections critiques dans de larges blocs de code.

10.11 Contrôle de qualité

Une approche de l'écriture du logiciel de haute qualité consiste à écrire des tests pour chaque fonction au fur et à mesure qu'elle est développée et à faire tourner ces tests fréquemment durant le processus de développement.

Le module `doctest` fournit un outil pour examiner un module et valider les tests immergés dans las chaînes de documentation du programme. La construction d'un test est aussi simple que copier-coller un appel typique et son résultat dans la chaîne de documentation. Cela améliora la documentation en donnant un exemple à l'utilisateur et permet au module `doctest` de s'assurer que le code reste fidèle à la documentation :

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()       # automatically validate the embedded tests
```

Le module `unittest` n'est pas un module `doctest` sans effort, mais il permet de maintenir dans un fichier séparé un plus vaste ensemble de tests.

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main()   # Calling from the command line invokes all tests
```

10.12 Les piles sont fournies avec l'appareil

Python a une philosophie "piles comprises". C'est dans la sophistication et la robustesse de ses paquetages les plus gros que cela se voit le mieux. Par exemple :

- Les modules `xmlrpclib` et `SimpleXMLRPCServer` font que l'implémentation des appels de procédures éloignées (RPC) est une tâche presque triviale. Malgré les noms, aucune connaissance ou manipulation directe de XML n'est nécessaire.
- Le paquetage `email` est une bibliothèque pour gérer les messages électroniques, y compris les documents MIME et les autres documents basés sur la RFC-2822. Contrairement à `smtplib` et `poplib`, qui envoient et reçoivent effectivement des messages, le paquetage `email` a une boîte à outils complète pour construire ou décoder des messages à la structure complexe (ce qui inclut les attachements) et pour implémenter les protocoles Internet pour le codage et les entêtes.
- Les paquetages `xml.dom` et `xml.sax` fournissent un support robuste pour analyser ces formats d'échange de données très répandus. De même, le module `csv` effectue des lectures et des écritures directement dans un format de base de données commun. Ensemble, ces modules et paquetages simplifient grandement l'échange de données entre des applications Python et d'autres outils.
- L'internationalisation est supportée par un certain nombre de modules, incluant les paquetages `gettext`, `locale` et `codecs`.

CHAPITRE

ONZE

Petit tour dans la bibliothèque standard - Deuxième partie

Cette seconde visite guidée concerne des modules plus avancés qui répondent aux besoins de la programmation professionnelle. Ces modules apparaissent rarement dans les petits scripts.

11.1 Mise en forme des sorties

Le module `repr` fournit une version de `repr()` pour un affichage abrégé de conteneurs volumineux ou profondément imbriqués :

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

Le module `pprint` offre un contrôle plus sophistiqué sur l'impression d'objets intégrés aussi bien que d'objets définis par l'utilisateur, d'une manière qui est lisible par l'interpréteur. Lorsque le résultat est plus long qu'une ligne, l'*imprimeur-enjoliveur* ajoute des fins-de-ligne et une indentation pour révéler plus clairement la structure des données :

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]] ...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

Le module `textwrap` formate les paragraphes de texte de sorte à remplir une largeur d'écran donnée :

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

Le module `locale` accède à une base de données de formats de données spécifiques à des cultures. L'attribut `grouping` des fonctions de formatage de `locale` fournit un moyen direct pour mettre en forme des nombres avec des séparateurs de groupes de chiffres :

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format("%s%.*f", (conv['currency_symbol'],
...       conv['int_frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Modèles

Le module `string` inclut une souple classe `Template` avec une syntaxe simplifiée adaptée à l'édition par les utilisateurs finaux. Cela permet que les utilisateurs personnalisent leurs applications sans devoir les altérer.

Ce format utilise des noms garde-place formés par '$' suivi d'un identificateur Python valide (des caractères alphanumériques et des blancs soulignés). En entourant ces noms par des accolades on leur permet d'être suivis de caractères alphanumériques sans espaces. En écrivant '$$' on crée un '$' simple.

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

La méthode `substitute` lève une erreur `KeyError` lorsqu'un garde-place n'est pas fourni dans un fictionnaire ou dans un argument à mot-clé. Pour des applications de style *mailing-fusion*, l'information fournie par l'utilisateur peut être incomplète et la méthode `safe_substitute` peut être plus appropriée – elle laisse les garde-place inchangés lorsque la donnée est omise :

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Les sous-classes des modèles peuvent spécifier un délimiteur personnalisé. Par exemple, un utilitaire de renommage pour un butineur de photos peut choisir d'utiliser des signes pourcent pour des garde-place comme la date courante, le numéro de série de l'image ou le format du fichier :

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '%s --> %s' % (filename, newname)
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Une autre application pour ces modèles est de séparer la logique du programme d'avec les détails des divers formats de sortie. Il devient possible de substituer des modèles personnalisés par des fichiers XML, des rapports en texte plat ou des rapports web en HTML.

11.3 Travail avec des enregistrements binaires

Le module `struct` fournit les fonctions `pack()` et `unpack()` pour travailler avec des enregistrements ayant des formats binaires de longueurs variables. L'exemple suivant montre comment boucler à travers l'information de tête d'un fichier zip (pour `pack` les codes "H" et "L" représentent des nombres sans signe de deux et quatre octets respectivement) :

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('LLLHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size     # skip to the next header
```

11.4 Multi-threading

Le *threading* est une technique qui consiste à découpler les tâches qui ne sont pas séquentiellement dépendantes. Les *threads* peuvent être utilisés pour améliorer la réactivité d'applications qui acceptent des saisies de l'utilisateur pendant que d'autres tâches sont effectuées en arrière-plan. Un emploi apparenté à celui-là consiste à faire tourner

des entrées/sorties en parallèle avec des calculs dans un autre *thread*.

Le code suivant montre comment le module de haut niveau `threading` peut faire tourner des tâches en arrière-plan pendant que le programme principal continue à tourner :

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'
background.join()              # Wait for the background task to finish
print 'Main program waited until background was done.'
```

Le principal défi des applications multi-*threaded* est la coordination des *threads* qui partagent des données ou d'autres ressources. A cet effet, le module `threading` fournit un certain nombre de primitives de synchronisation incluant des horloges, des événements, des variables de condition et des sémaphores.

Bien que ces outils soient puissants, des erreurs de conception mineures peuvent être à l'origine de problèmes difficiles à reproduire. Si bien que l'approche préférée de la coordination de tâches consiste à concentrer tous les accès à une ressource donnée dans un même *thread* et à utiliser le module `Queue` pour passer à ce *thread* les requêtes provenant des autres *threads*. Les applications qui utilisent des objets `Queue` pour la communication inter-*threads* et la coordination sont plus faciles à concevoir, plus lisibles et plus fiables.

11.5 Journalisation

Le module `logging` offre un système complet et flexible de journalisation. Dans l'emploi le plus simple, les messages sont envoyés à un fichier ou à `sys.stderr` :

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Cela produit la sortie suivante :

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Par défaut, les messages d'information et de debugging sont supprimés et la sortie est envoyée sur la sortie d'erreur standard. D'autres options de sortie comprennent le déroutement de messages à travers du courrier électronique, des datagrammes, des *sockets* ou vers un serveur HTTP. De nouveaux filtres peuvent choisir les différents chemins selon la priorité du message : `DEBUG`, `INFO`, `WARNING`, `ERROR` et `CRITICAL`.

Le système de journalisation peut être configuré directement depuis Python ou peut être chargé à partir d'un fichier de configuration éditable par l'utilisateur afin de personnaliser la journaisation sans altérer l'application.

11.6 Références faibles

Python fait une gestion automatique de la mémoire (comptage de références pour la plupart des objets et ramasse-miettes pour éliminer les cycles). Lorsque la dernière référence sur une mémoire est éliminée, cette dernière est rapidement libérée.

Cette approche est parfaite pour la plupart des applications. Cependant, dans certaines occasions il est nécessaire de suivre la trace de certains objets, mais uniquement lorsqu'ils sont utilisés par quelque autre objet. Malheureusement, pour suivre la trace d'un objet il faut avoir une référence dessus, ce qui rend l'objet permanent. Le module `weakref` fournit des outils pour pister des objets sans créer des références dessus. Lorsque l'objet n'est plus nécessaire, il est automatiquement enlevé de la table de `weakref` et une action enregistrée pour l'objet est déclenchée. Les applications typiques de cela comprennent la mise en cache d'objets dont la création est coûteuse :

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in -toplevel-
    d['primary']                # entry was automatically removed
  File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Outils pour travailler avec des listes

Une grande partie des besoins des structures de données peuvent être comblés avec le type intégré liste. Parfois, cependant, il faut des implémentations alternatives des listes, réalisant d'autres compromis à propos des performances.

Le module `array` fournit un objet `array()` qui est comme une liste mais contient uniquement des données homogènes, mémorisées avec plus de compacité. L'exemple suivant montre un tableau de nombres mémorisés comme des entiers sans signe codés sur deux octets (code "H") au lieu des 16 octets par élément pour une liste Python ordinaire d'objets int.

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Le module `collections` fournit un objet `deque()` qui est comme une liste avec des opérations rapides pour ajouter un élément (à droite) ou en enlever un à gauche, mais les recerches au milieu de la liste sont lentes. Ces objets conviennent bien à l'implémentation des queues et des recherches en largeur dans les arbres :

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

En plus des alternatives à l'implémentation des listes la bibliothèque offre aussi des outils comme le module `bisect` avec des fonctions pour manipuler des listes triées :

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

Le module `heapq` fournit des fonctions pour l'implémentation de teas basés sur des listes ordinaires. La plus petite valeur est toujours conservée à l'emplacement zéro. Cela est utile pour les applications qui accèdent de manière répétée au plus petit élément mais ne veulent pas exécuter un tri de liste complet :

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]   # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Arithmétique décimale à virgule flottante

Le module `decimal` offre un type de données `Decimal` destiné à faire de l'arithmétique décimale à virgule flottante. Comparée à l'implémentation intégrée des nombres binaires en virgule flottante, `float`, cette nouvelle classe est particulièrement utile aux applications financières et aux autres usages qui requièrent une représentation décimale exacte, le contrôle de la précision, le contrôle des arrondis, pour satisfaire les prescriptions légales ou réglementaires, le repérage des chiffres décimaux significatifs et aussi les applications dont l'utilisateur espère que les résultats coïncideront avec ceux de calculs faits à la main.

Par exemple, le calcul de 5% du prix d'une communication téléphonique de 70 centimes donne des résultats différents en nombres décimaux à virgule flottante et en nombre binaires à virgules flottante. La différence devient significative si les résultats sont arrondis au centime le plus proche :

```
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

Le résultat décimal garde un zéro à la fin, impliquant automatiquement un résultat à quatre chiffres après la virgule, ce qui correspond au produit de deux facteurs ayant deux chiffres après la virgule. `Decimal` reproduit les mathématiques comme on les fait à la main et évite les problèmes qui peuvent survenir lorsque le binaire à virgule flottante ne peut pas représenter exactement les quantités décimales.

```
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Le module `Decimal` fournit une arithmétique avec autant de précision que nécessaire :

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
```

CHAPITRE

DOUZE

Et maintenant ?

La lecture de ce tutoriel aura probablement renforcé votre intérêt dans l'utilisation de Python — vous devriez être impatient d'appliquer Python à la résolution de vos problèmes du monde réel. Où pouvez-vous aller maintenant ?

Vous devriez lire, ou au moins feuilleter, la *Python Library Reference*, qui offre un matériel de référence complet (quoique succinct) sur des types, des fonctions et des modules qui peuvent vous faire économiser beaucoup de temps en écrivant des programmes en Python. La distribution standard de Python inclut *énormément* de code, en C comme en Python ; il y a des modules pour lire des boîtes aux lettres UNIX, rechercher des documents via HTTP, générer des nombres aléatoires, interpréter des options de ligne de commande, écrire des programmes CGI, comprimer des données, et bien davantage ; parcourir la Library Reference devrait vous donner une idée de ce qui est disponible.

Le site Web principal pour Python est http ://www.python.org ; vous y trouverez du code, de la documentation, et des liens vers des pages en rapport avec Python dans le Web. Ce site Web dispose de miroirs dans plusieurs endroits du monde, en Europe, au Japon et en Australie ; un miroir peut être plus rapide d'accès que le site principal, suivant votre position géographique. Un site plus informel est http ://starship.skyport.net, qui contient nombre de pages personnelles en rapport avec Python ; beaucoup de gens y ont placé des programmes téléchargeables.

Pour des problèmes ou des questions sur Python, vous pouvez poster dans le groupe de discussion comp.lang.python, ou écrire à la liste de diffusion python-list@cwi.nl. Le groupe et la liste sont en passerelle, donc les messages de l'un passent automatiquement dans l'autre. Il y a entre 35 et 45 messages par jour, avec des questions (et des réponses), des suggestions d'apports nouveaux, et des annonces de nouveaux modules. Avant de poster, soyez sûr d'avoir consulté la liste de Foire Aux Questions (Frequently Asked Questions ou FAQ) située à http ://www.python.org/doc/FAQ.html, et d'avoir regardé dans le répertoire 'Misc/' de la distribution source de Python. La FAQ répond à plusieurs des questions qui reviennent sans cesse, et peut déjà contenir la solution à votre problème.

ANNEXE

A

Edition d'entrée interactive et substitution historique

Quelques versions de l'interpréteur Python supportent l'édition de la ligne d'entrée courante et la substitution historique, des commodités semblables à celles du shell Korn et du shell GNU Bash. Cela est implémenté en utilisant la librairie *GNU Readline*, qui supporte l'édition à la vi et à la emacs. Cette librairie a sa propre documentation, que je ne dupliquerai pas ici ; toutefois, les bases peuvent être vite expliquées.

Ce chapitre *ne* documente *pas* les capacités d'édition du paquetage PythonWin de Mark Hammond ou l'environnement basé sur Tk, IDLE, distribué avec Python. Le rappel d'historique de ligne de commande qui fonctionne dans les fenêtres DOS sous NT et d'autres variantes de DOS et Windows est encore un autre bestiau.

A.1 Edition de ligne

Si elle est supportée, l'édition de ligne d'entrée est active lorsque l'interpréteur affiche un prompt primaire ou secondaire. La ligne courante peut être éditée en utilisant les caractères de contrôle conventionnels d'Emacs. Les plus importants sont : C-A (Control-A) déplace le curseur en début de ligne, C-E en fin de ligne, C-B déplace d'une position vers la gauche, C-F vers la droite. Backspace efface le caractère à gauche du curseur, C-D le caractère à droite. C-K tue (efface) le reste de la ligne à droite du curseur, C-Y rappelle la dernière chaîne tuée. C-tiret-bas défait le dernier changement réalisé ; il peut être répété pour un effet cumulé.

A.2 Substitution historique

La substitution historique fonctionne ainsi. Toutes les lignes d'entrée non-vides sont enregistrées dans un tampon d'historique, et lorsqu'un nouveau prompt est affiché, vous êtes dans une nouvelle ligne à la fin de ce tampon. C-P déplace d'une ligne vers le haut (vers l'arrière) dans l'historique, C-N vers le bas. Toute ligne de l'historique peut être éditée ; un astérisque apparaît en début de ligne avant le prompt pour indiquer que la ligne a été modifiée. Appuyer sur Entrée passe la ligne courante à l'interpréteur. C-R commence une recherche incrémentale en arrière ; C-S une recherche en avant.

A.3 Définition des touches

Les associations des touches aux commandes et d'autres paramètres de la bibliothèque Readline peuvent être redéfinis en plaçant des commandes d'initialisation dans le fichier '$HOME/.inputrc'. Les définitions de raccourcis clavier ont la forme

```
nom-de-touche: nom-de-fonction
```

ou bien

```
"chaîne": nom-de-fonction
```

et les options sont modifiées avec

```
set nom-option valeur
```

Par exemple :

```
# Je préfère l'édition à la vi:
set editing-mode vi

# Edition sur une seule ligne:
set horizontal-scroll-mode On

# Redéfinir quelques touches:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Remarquez que l'action liée a la tabulation en Python est d'insérer une tabulation, au lieu de l'action par défaut de Readline, qui est la complétion de noms. Si vous insistez, vous pouvez redéfinir cela avec

```
Tab: complete
```

dans le fichier '$HOME/.inputrc'. (Bien sûr, cela rend plus difficile l'indentation des lignes de continuation...)

La complétion automatique des noms de variable et de module est disponible en option. Pour l'activer dans le mode interactif de l'interpréteur, rajouter ceci à votre fichier '$HOME/.pythonrc' :

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Cela lie la touche `Tab` à la fonction de complétion, donc appuyer deux fois sur `Tab` affiche les suggestions de complétion ; celles-ci sont cherchées parmi les noms d'instructions, les variables locales actuelles et les noms de module disponibles. Pour des expressions avec un point comme `string.a`, la complétion sera réalisée jusqu'au '`.`' puis les complétions correspondant aux attributs suivants seront suggérées. Remarquez que cela peut amener à exécuter du code propre à une application si un objet avec une méthode `__getattr__()` apparaît dans l'expression.

Un fichier de démarrage plus capable pourrait ressembler à cet exemple. Notez que cela efface les noms qu'il crée une fois qu'ils ne sont plus utilisés ; c'est fait car le fichier de démarrage est exécuté dans le même espace de noms que les commandes interactives, et enlever les noms évite de créer des effets de bord dans les environnement interactifs. Vous pourriez trouver pratique de garder certains des modules importés, comme `os`, qui s'avère être nécessaire dans la plupart des sessions dans l'interpréteur.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it, e.g. "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

A.4 Commentaire

Cette commodité est un énorme pas en avant par rapport aux versions plus anciennes de l'interpréteur ; néanmoins, quelques vœux restent à remplir : ce serait bien que l'indentation soit suggérée sur les lignes de continuation (l'interpréteur sait si un lexème d'indentation est requis par la suite). Le mécanisme de complétion pourrait se servir de la table de symboles de l'interpréteur. Une commande pour vérifier (voire suggérer) l'équilibre des parenthèses, des quotes, etc. serait aussi très utile.

ANNEXE

B

Arithmétique à virgule flottante : problèmes et limites

Les nombres à virgule flottante sont représentés dans le matériel de l'ordinateur comme des fractions de base 2 (binaires). Par exemple, la fraction décimale

```
0.125
```

a la valeur 1/10 + 2/100 + 5/1000, et de la même façon, la fraction binaire

```
0.001
```

a la valeur 0/2 + 0/4 + 1/8. Ces deux fractions ont des valeurs identiques, mais la seule différence réelle étant que la première est écrite en notation fractionnaire base 10, et la seconde en base 2.

Malheureusement, la plupart des fractions décimales ne peuvent pas être représentées exactement comme des fractions binaires. Une conséquence est que, en général, les nombres à virgule flottante décimaux que vous entrez sont seulement des approximations des nombres binaires à virgule flottante effectivement stockés dans la machine.

Le problème est facile à comprendre tout d'abord en base 10. Considérons la fraction 1/3. Vous pouvez en faire une approximation en fraction de base 10 :

```
0.3
```

ou, mieux,

```
0.33
```

ou, mieux,

```
0.333
```

et ainsi de suite. Peu importe combien de chiffres vous souhaitez écrire, le résultat ne sera jamais exactement 1/3, mais sera une approximation toujours meilleure de 1/3.

Dela même façon, peu importe combien de chiffres de base 2 vous souhaitez utiliser, la valeur décimale 0.1 ne peut être représentée exactement comme fraction de base 2. En base 2, 1/10 est une fraction se répétant indéfiniment

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

Arrêtez à n'importe quel nombre fini de bits, et vous aurez une approximation. Cela explique que vous voyiez des choses comme :

```
>>> 0.1
0.10000000000000001
```

Sur la plupart des machines aujourd'hui, c'est ce que vous verrez si vous entrez 0.1 lors du prompt Python. Vous pourriez voir autre chose, pourtant, car le nombre de bits utilisé par le matériel pour stocker les valeurs en virgule flottante peut varier entre les machines, et Python affiche juste une approximation décimale de la vraie valeur décimale de l'approcimation binaire stockée dans la machine. Sur la plupart des machines, si Python devait afficher la vraie valeur décimale de l'approximation binaire stockées pour 0.1, il devrait afficher

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

à la place ! Le prompt Python utilise (implicitement) la fonction intégrée `repr()` pour obtenir une version en chaine de tout ce qu'il affiche. Pour les flottants, `repr(`*flottant*`)` arrondit la vraie valeur décimale à 17 décimales significatives, donnant

```
0.10000000000000001
```

`repr(`*flottant*`)` produit 17 décimales significatives parce qu'il s'avère que c'est suffisant (sur la plupart des machines) de façon à ce que `eval(repr(`*x*`)) ==` *x* exactement pour tous les flottants finis *x*, mais que l'arrondissement à 16 décimales n'est pas suffisant pour assurer cela.

Notez que c'est dans la vraie nature de la virgule flottante en binaire : ce n'est pas un bogue de Python, ce n'est pas non plus un bogue dans votre code, et vous verrez la même chose dans tout langage qui supporte l'arithmétique à virgule flottante de votre matériel (bien que certains langages puissent ne pas *afficher* la différence par défaut, ou dans tous les modes d'affichage).

La fonction intégrée `str()` de Python produit seulement 12 décimales significatives, et vous pourriez souhaiter utiliser cela à la place. Il n'est pas habituel que `eval(str(`*x*`))` reproduise *x*, mais la sortie pourrait être plus agréable à regarder :

```
>>> print str(0.1)
0.1
```

Il est important de comprendre que cela est, en vérité, une illusion : la valeur dans la machine n'est pas exactement 1/10, vous arrondissez seulement l'*affichage* de la vraie valeur de la machine.

D'autres surprises découlent de celle-ci. Par exemple, après avoir vu

```
>>> 0.1
0.10000000000000001
```

vous pourriez être tentés d'utiliser la fonction `round()` pour la tronquer vers la seule décimale que vous attendez. Mais ça ne change rien :

```
>>> round(0.1, 1)
0.10000000000000001
```

Le problème est que la valeur binaire à virgule flottante stockée pour "0.1" était déjà la meilleure approximation binaire possible de 1/10, donc essayer de l'arrondir à nouveau ne peut rien améliorer : c'était déjà aussi bon que possible.

Une autre conséquence est que puisque 0.1 n'est pas exactement 1/10, ajouter 0.1 à lui-même 10 fois pourrait ne pas atteindre exactement 1.0, soit :

```
>>> somme = 0.0
>>> for i in range(10):
...     somme += 0.1
...
>>> somme
0.99999999999999989
```

L'arithmétique binaire à virgule flottante réserve de nombreuses surprises comme celle-ci. Le problème avec 0.1 est expliqué en détails précis ci-dessous, dans la section B.1, "Erreur de représentation". Voir *The Perils of Floating Point* pour un compte-rendu plus complet d'autres surprises courantes.

Comme il est dit près de la fin, "il n'y a pas de réponses faciles." Cependant, ne soyez pas trop inquiets ! Les erreurs dans les opérations flottantes en Python sont héritées des matériels à virgule flottante, et sur la plupart des machines sont de l'ordre de pas plus de 1 sur 2**53 par opération. C'est plus qu'adéquat pour la plupart des tâches, mais vous devez conserver à l'esprit que ce n'est pas de l'arithmétique décimale, et que toute opération flottante peut souffrir d'une nouvelle erreur d'arrondi.

Alors que des cas pathologiques existent effectivement, pour la plupart des utilisations courantes de l'arithmétique à virgule flottante, vous verrez le résultat que vous attendez au final si vous arrondissez simplement l'affichage de vos résultats finaux au nombre de décimales que vous attendez. `str()` suffit généralement, et pour un contrôle plus fin voir la discussion de l'opérateur de format `%` de Python : les codes de format `%g`, `%f` et `%e` fournissent des façons adaptables et simples d'arrondir les résultats pour l'affichage.

B.1 Erreur de représentation

Cette section explique l'exemple de "0.1" en détail, et montre comment vous pouvez effectuer une analyse exacte de cas similaires vous-même. Une familiarité basique avec la représentation à virgule flottante est supposée.

Erreur de représentation renvoie au fait que certaines (la plupart, en réalité) des fractions décimales ne peuvent être représentées exactement comme fractions binaires (base 2). C'est la raison principale pour laquelle Python (ou Perl, C, C++, Java, Fortran, et beaucoup d'autres) n'afficheront souvent pas le nombre décimal exact que vous attendez :

```
>>> 0.1
0.10000000000000001
```

Pourquoi cela ? 1/10 n'est pas exactement représentable en tant que fraction binaire. Presque toutes les machines d'aujourd'hui (juin 2005) utilisent l'arithmétique à virgule flottante IEEE-754, et presque toutes les plate-formes associent les flottants Python à des "double précision" IEEE-754. Les doubles 754 contiennent 53 bits de précision, donc en entrée l'ordinateur s'efforce de convertir 0.1 vers la fraction la plus proche qu'il peut de forme *J*/2***N* où *J* est un entier contenant exactement 53 bits. Réécrire

```
 1 / 10 ~= J / (2**N)
```

en

```
J ~= 2**N / 10
```

et se souvenir que *J* a exactement 53 bits (est `>= 2**52` mais `< 2**53`), la meilleure valeur pour *N* est 56 :

```
>>> 2L**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

C'est à dire que 56 est la seule valeur pour *N* qui laisse *J* à exactement 53 bits. La meilleure valeur possible pour *J*

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

Puisque le reste est supérieur à la moitié de 10, la meilleure aproximation est obtenue en arrondissant au supérieur :

```
>>> q+1
7205759403792794L
```

Ainsi la meilleure approximation possible de 1/10 en double précision 754 est cela sur 2**56, ou

```
7205759403792794 / 72057594037927936
```

Notez que depuis que nous avons arrondi, c'est un peu plus grand que 1/10 ; si nous n'avions pas arrondi, le quotient aurait été un peu plus petit que 1/10. Mais dans aucun cas il ne peut être *exactement* 1/10 !

Donc l'ordinateur ne "voit" jamais 1/10 : ce qu'il voit est la fraction exacte donnée ci-dessus, la meilleure approximation double 754 qu'il puisse obtenir :

```
>>> .1 * 2L**56
7205759403792794.0
```

Si on multiplie cette fraction par 10**30, nous pouvons voir la valeur (tronquée) de ses 30 décimales les plus significatives :

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000005551115123125L
```

ce qui signifie que le nombre exact stocké dans l'ordinateur est approximativement égal à la valeur décimale 0.100000000000000005551115123125. Arrondir cela à 17 chiffres significatifs donne le 0.10000000000000001 que Python affiche (enfin, qu'il affichera sur toute plateforme conforme-754 qui effectue les meilleures conversions de saisie et d'affichage possibles dans sa bibliothèque C — la votre peut-être pas !).

ANNEXE

C

Historique et licence

C.1 Histoire de Python

Python a été créé dans les années 90 par Guido van Rossum au *Stichting Mathematisch* (*CWI*, voir http ://www.cwi.nl/) aux Pays-Bas, comme successeur d'un langage appelé *ABC*. Guido est resté l'auteur principal de Python, même si ce dernier inclut maintenant de nombreuses contributions d'autres personnes.

En 1995, Guido a continué son travail sur Python à la *Corporation for National Research Initiatives* (*CNRI*, voir http ://www.cnri.reston.va.us/) à Reston, Virginie, où il a développé et mis à disposition plusieurs versions du logiciel.

En mai 2000, Guido et l'équipe de développement du cœur de Python ont déménagé chez *BeOpen.com* pour former le *BeOpen PythonLabs team*. En octobre de la même année, l'équipe de PythonLabs a déménagé chez *Digital Creations* (maintenant *Zope Corporation*, voir http ://www.zope.com/). En 2001 a été créée la *Python Software Foundation* (*PSF*, voir http ://www.python.org/psf/), une organisation sans but lucratif spécifiquement créée pour gérer la propriété intellectuelle de Python. *Zope Corporation* est un membre et un sponsor de la *PSF*.

Toutes les versions de Python sont *Open Source* (voir http ://www.opensource.org/ pour la définition de cette expression). Historiquement, la plupart – mais non la totalité – des versions de Python sont aussi compatibles *GPL*. La table suivante résume les diverses livraisons.

Produit	Dérivé de	Année	Propriétaire	compatible GPL ?
de 0.9.0 à 1.2	–	1991-1995	CWI	oui
de 1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.2	2.1.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2.1	2.2	2002	PSF	oui
2.2.2	2.2.1	2002	PSF	oui
2.2.3	2.2.2	2002-2003	PSF	oui
2.3	2.2.2	2002-2003	PSF	oui
2.3.1	2.3	2002-2003	PSF	oui
2.3.2	2.3.1	2003	PSF	oui

Note: "Compatible GPL" ne signifie pas que nous distribuons Python sous licence GPL. Contrairement à GPL, toutes les licences Python vous permettent de distribuer une version modifiée sans rendre vos changements *open source*. Une licence compatible GPL rend possible de combiner Python avec d'autres logiciels distribués sous GPL, les autres licences non.

Merci aux nombreux volontaires extérieurs qui ont travaillé sous la direction de Guido pour rendre possibles ces productions.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.4.1

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.4.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.4.1 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2003 Python Software Foundation ; All Rights Reserved” are retained in Python 2.4.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.4.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.4.1.
4. PSF is making Python 2.4.1 available to Licensee on an “AS IS” basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.4.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.4.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.4.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.4.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (“BeOpen”), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (“Licensee”) accessing and otherwise using this software in source or binary form and its associated documentation (“the Software”).
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create

any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http ://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives ; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes) : "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle) : 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL : http ://hdl.handle.net/1895.22/1013."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice

and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

==========

La version traduite en français de ce document, résultat du travail initial de plusieurs traducteurs (Olivier Berger, Daniel Calvelo Aros, Bruno Liénard) a été assemblée et mise à jour par Olivier Berger dans le cadre du projet bénévole de traduction en français de la documentation Python. Pour contribuer au projet, ou obtenir une mise à jour de ce document, veuillez vous reporter au site Internet suivant : http ://sourceforge.net/projects/frpython. Ce document est issu de la traduction du document original en anglais V1.5.1 de Guido Van Rossum, avec son aimable permission. Il a été mis-à-jour régulièrement pour tenir compte des évolutions du document original. La présente version est à jour par rapport à la V2.0.1. Les questions concernant le langage Python ou le contenu du présent document sont à adresser directement à l'auteur.

==========

La présente mise à jour de la traduction, synchrone avec la dernière version américaine en date – *Release 2.4.1* – a été réalisée par Henri Garreta. Merci de lui envoyer (henri.garreta@univmed.fr) les signalements d'erreurs et les remarques concernant la traduction.