

---

# Python Tutorial

*Version 2.7.15*

**Guido van Rossum  
and the Python development team**

janvier 07, 2019

Python Software Foundation  
Email : [docs@python.org](mailto:docs@python.org)



---

## Table des matières

---

<b>1</b>	<b>Mise en bouche</b>	<b>3</b>
<b>2</b>	<b>Mode d'emploi de l'interpréteur Python</b>	<b>5</b>
2.1	Lancement de l'interpréteur . . . . .	5
2.2	L'interpréteur et son environnement . . . . .	6
<b>3</b>	<b>Introduction informelle à Python</b>	<b>9</b>
3.1	Utilisation de Python comme une calculatrice . . . . .	9
3.2	Premiers pas vers la programmation . . . . .	18
<b>4</b>	<b>D'autres outils de contrôle de flux</b>	<b>21</b>
4.1	L'instruction <code>if</code> . . . . .	21
4.2	L'instruction <code>for</code> . . . . .	21
4.3	La fonction <code>range()</code> . . . . .	22
4.4	Les instructions <code>break</code> , <code>continue</code> et les clauses <code>else</code> au sein des boucles . . . . .	23
4.5	L'instruction <code>pass</code> . . . . .	24
4.6	Définir des fonctions . . . . .	24
4.7	D'avantage sur la définition des fonctions . . . . .	26
4.8	Aparté : le style de codage . . . . .	30
<b>5</b>	<b>Structures de données</b>	<b>33</b>
5.1	Compléments sur les listes . . . . .	33
5.2	L'instruction <code>del</code> . . . . .	39
5.3	Tuples et séquences . . . . .	39
5.4	Ensembles . . . . .	40
5.5	Dictionnaires . . . . .	41
5.6	Techniques de boucles . . . . .	42
5.7	Plus d'informations sur les conditions . . . . .	43
5.8	Comparer des séquences avec d'autres types . . . . .	44
<b>6</b>	<b>Modules</b>	<b>47</b>
6.1	Les modules en détail . . . . .	48
6.2	Modules standards . . . . .	50
6.3	La fonction <code>dir()</code> . . . . .	51
6.4	Les paquets . . . . .	52
<b>7</b>	<b>Les entrées/sorties</b>	<b>57</b>

7.1	Formatage de données . . . . .	57
7.2	Lecture et écriture de fichiers . . . . .	61
<b>8</b>	<b>Erreurs et exceptions</b>	<b>65</b>
8.1	Les erreurs de syntaxe . . . . .	65
8.2	Exceptions . . . . .	65
8.3	Gestion des exceptions . . . . .	66
8.4	Déclencher des exceptions . . . . .	68
8.5	Exceptions définies par l'utilisateur . . . . .	69
8.6	Définition d'actions de nettoyage . . . . .	70
8.7	Actions de nettoyage prédéfinies . . . . .	71
<b>9</b>	<b>Classes</b>	<b>73</b>
9.1	Objets et noms : préambule . . . . .	73
9.2	Portées et espaces de noms en Python . . . . .	74
9.3	Une première approche des classes . . . . .	75
9.4	Remarques diverses . . . . .	79
9.5	Héritage . . . . .	80
9.6	Variables privées et références locales aux classes . . . . .	82
9.7	Trucs et astuces . . . . .	83
9.8	Les exceptions sont aussi des classes . . . . .	83
9.9	Itérateurs . . . . .	84
9.10	Générateurs . . . . .	86
9.11	Expressions et générateurs . . . . .	86
<b>10</b>	<b>Survol de la bibliothèque standard</b>	<b>89</b>
10.1	Interface avec le système d'exploitation . . . . .	89
10.2	Jokers sur les noms de fichiers . . . . .	90
10.3	Paramètres passés en ligne de commande . . . . .	90
10.4	Redirection de la sortie d'erreur et fin d'exécution . . . . .	90
10.5	Recherche de motifs dans les chaînes . . . . .	90
10.6	Mathématiques . . . . .	91
10.7	Accès à internet . . . . .	91
10.8	Dates et heures . . . . .	91
10.9	Compression de données . . . . .	92
10.10	Mesure des performances . . . . .	92
10.11	Contrôle qualité . . . . .	93
10.12	Piles fournies . . . . .	93
<b>11</b>	<b>Brief Tour of the Standard Library — Part II</b>	<b>95</b>
11.1	Formatage de l'affichage . . . . .	95
11.2	Gabarits (Templates) . . . . .	96
11.3	Traitement des données binaires . . . . .	97
11.4	Fils d'exécution . . . . .	97
11.5	Journalisation . . . . .	98
11.6	Références faibles . . . . .	99
11.7	Outils pour les listes . . . . .	99
11.8	Arithmétique décimale à virgule flottante . . . . .	100
<b>12</b>	<b>Pour aller plus loin</b>	<b>103</b>
<b>13</b>	<b>Édition interactive des entrées et substitution d'historique</b>	<b>105</b>
13.1	Édition de ligne . . . . .	105
13.2	Substitution d'historique . . . . .	105
13.3	Raccourcis clavier . . . . .	106

13.4 Alternatives à l'interpréteur interactif . . . . .	107
<b>14 Arithmétique en nombres à virgule flottante : problèmes et limites</b>	<b>109</b>
14.1 Erreurs de représentation . . . . .	111
<b>15 Annexe</b>	<b>113</b>
15.1 Mode interactif . . . . .	113
<b>A Glossaire</b>	<b>115</b>
<b>B À propos de ces documents</b>	<b>125</b>
B.1 Contributeurs de la documentation Python . . . . .	125
<b>C Histoire et licence</b>	<b>127</b>
C.1 Histoire du logiciel . . . . .	127
C.2 Conditions générales pour accéder à, ou utiliser, Python . . . . .	128
C.3 Licences et Remerciements pour les logiciels inclus . . . . .	131
<b>D Copyright</b>	<b>143</b>
<b>Index</b>	<b>145</b>



Python est un langage de programmation puissant et facile à apprendre. Il dispose de structures de données de haut niveau et permet une approche simple mais efficace de la programmation orientée objet. Parce que sa syntaxe est élégante, que son typage est dynamique et qu'il est interprété, Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur la plupart des plateformes.

L'interpréteur Python et sa vaste bibliothèque standard sont disponibles librement, sous forme de sources ou de binaires, pour toutes les plateformes majeures depuis le site Internet <https://www.python.org/> et peuvent être librement redistribués. Ce même site distribue et pointe vers des modules, des programmes et des outils tiers. Enfin, il constitue une source de documentation.

L'interpréteur Python peut être facilement étendu par de nouvelles fonctions et types de données implémentés en C ou C++ (ou tout autre langage appellable depuis le C). Python est également adapté comme langage d'extension pour personnaliser des applications.

Dans ce tutoriel, nous introduisons, de façon informelle, les concepts de base ainsi que les fonctionnalités du langage Python et de son écosystème. Il est utile de disposer d'un interpréteur Python à portée de main pour mettre en pratique les notions abordées. Si ce n'est pas possible, pas de souci, les exemples sont inclus et le tutoriel est adapté à une lecture « hors ligne ».

Pour une description des objets et modules de la bibliothèque standard, reportez-vous à [library-index](#). [reference-index](#) présente le langage de manière plus formelle. Pour écrire des extensions en C ou en C++, lisez [extending-index](#) et [c-api-index](#). Des livres sont également disponibles qui couvrent Python dans le détail.

L'ambition de ce tutoriel n'est pas d'être exhaustif et de couvrir chaque fonctionnalité, ni même toutes les fonctionnalités les plus utilisées. Il vise, en revanche, à introduire les fonctionnalités les plus notables et à vous donner une bonne idée de la saveur et du style du langage. Après l'avoir lu, vous serez capable de lire et d'écrire des modules et des programmes Python et vous serez prêt à en apprendre davantage sur les modules de la bibliothèque Python décrits dans [library-index](#).

Pensez aussi à consulter le [Glossaire](#).



# CHAPITRE 1

---

## Mise en bouche

---

Lorsqu'on travaille beaucoup sur ordinateur, on finit souvent par vouloir automatiser certaines tâches : par exemple, effectuer une recherche et un remplacement sur un grand nombre de fichiers de texte ; ou renommer et réorganiser des photos d'une manière un peu compliquée. Pour vous, ce peut être créer une petite base de données, une application graphique ou un simple jeu.

Quand on est un développeur professionnel, le besoin peut se faire sentir de travailler avec des bibliothèques C/C++/Java, mais on trouve que le cycle habituel écriture/compilation/test/compilation est trop lourd. Vous écrivez peut-être une suite de tests pour une telle bibliothèque et vous trouvez que l'écriture du code de test est pénible. Ou bien vous avez écrit un logiciel qui a besoin d'être extensible grâce à un langage de script, mais vous ne voulez pas concevoir ni implémenter un nouveau langage pour votre application.

Python est le langage parfait pour vous.

Vous pouvez écrire un script shell Unix ou des fichiers batch Windows pour certaines de ces tâches. Les scripts shell sont appropriés pour déplacer des fichiers et modifier des données textuelles, mais pas pour une application ayant une interface graphique ni pour des jeux. Vous pouvez écrire un programme en C/C++/Java, mais cela peut prendre beaucoup de temps, ne serait-ce que pour avoir une première maquette. Python est plus facile à utiliser et il vous aidera à terminer plus rapidement votre travail, que ce soit sous Windows, Mac OS X ou Unix.

Python reste facile à utiliser, mais c'est un vrai langage de programmation : il offre une bien meilleure structure et prise en charge des grands programmes que les scripts shell ou les fichiers batch. Par ailleurs, Python permet de beaucoup mieux vérifier les erreurs que le langage C et, en tant que *langage de très haut niveau*, il possède nativement des types de données très évolués tels que les tableaux de taille variable ou les dictionnaires. Grâce à ses types de données plus universels, Python est utilisable pour des domaines beaucoup plus variés que ne peuvent l'être Awk ou même Perl. Pourtant, vous pouvez faire de nombreuses choses au moins aussi facilement en Python que dans ces langages.

Python vous permet de découper votre programme en modules qui peuvent être réutilisés dans d'autres programmes Python. Il est fourni avec une grande variété de modules standards que vous pouvez utiliser comme base de vos programmes, ou comme exemples pour apprendre à programmer. Certains de ces modules donnent accès aux entrées/sorties, aux appels système, aux sockets réseaux et même aux outils comme Tk pour créer des interfaces graphiques.

Python est un langage interprété, ce qui peut vous faire gagner un temps considérable pendant le développement de vos programmes car aucune compilation ni édition de liens n'est nécessaire. L'interpréteur peut être utilisé de manière interactive, pour vous permettre d'expérimenter les fonctionnalités du langage, d'écrire des programmes jetables ou de tester des fonctions lors d'un développement incrémental. C'est aussi une calculatrice de bureau bien pratique.

Python permet d'écrire des programmes compacts et lisibles. Les programmes écrits en Python sont généralement beaucoup plus courts que leurs équivalents en C, C++ ou Java. Et ceci pour plusieurs raisons :

- les types de données de haut niveau vous permettent d'exprimer des opérations complexes en une seule instruction ;
- les instructions sont regroupées entre elles grâce à l'indentation, plutôt que par l'utilisation d'accolades ;
- aucune déclaration de variable ou d'argument n'est nécessaire.

Python est *extensible* : si vous savez écrire un programme en C, une nouvelle fonction ou module peut être facilement ajouté à l'interpréteur afin de l'étendre, que ce soit pour effectuer des opérations critiques à vitesse maximale ou pour lier des programmes en Python à des bibliothèques disponibles uniquement sous forme binaire (par exemple des bibliothèques graphiques dédiées à un matériel). Une fois que vous êtes à l'aise avec ces principes, vous pouvez relier l'interpréteur Python à une application écrite en C et l'utiliser comme un langage d'extensions ou de commandes pour cette application.

Au fait, le nom du langage provient de l'émission de la BBC « Monty Python's Flying Circus » et n'a rien à voir avec les reptiles. Faire référence aux sketches des Monty Python dans la documentation n'est pas seulement permis, c'est encouragé !

Maintenant que vos papilles ont été chatouillées, nous allons pouvoir rentrer dans le vif du sujet Python. Et comme la meilleure façon d'apprendre un langage est de l'utiliser, ce tutoriel vous invite à jouer avec l'interpréteur au fur et à mesure de votre lecture.

Dans le prochain chapitre, nous expliquons l'utilisation de l'interpréteur. Ce n'est pas la section la plus passionnante, mais c'est un passage obligé pour que vous puissiez mettre en pratique les exemples donnés plus loin.

Le reste du tutoriel présente diverses fonctionnalités du langage et du système Python au travers d'exemples, en commençant par les expressions simples, les instructions et les types de données, jusqu'à aborder des concepts avancés comme les exceptions et les classes, en passant par les fonctions et modules.

---

Mode d'emploi de l'interpréteur Python

---

## 2.1 Lancement de l'interpréteur

L'interpréteur python est habituellement installé `/usr/local/bin/python` sur les machines où il est disponible. En mettant `/usr/local/bin` dans les chemins de recherche de votre shell Unix, vous permettez de lancer l'interpréteur Python en tapant la commande :

```
python
```

dans le shell. Le choix du répertoire où se trouve l'interpréteur étant une option d'installation, d'autres chemins sont possibles ; voyez avec votre guru Python local ou votre administrateur système. (Par exemple, `/usr/local/python` est une localisation courante.)

Sur les machines Windows, l'installation de Python se situe généralement dans `C:\Python27`, bien que vous puissiez changer cela lorsque vous lancer l'installateur. Pour ajouter ce chemin dans vos chemins de recherche, vous pouvez taper la commande suivante dans votre émulateur de terminal DOS

```
set path=%path%;C:\python27
```

Tapez un caractère de fin de fichier (**Ctrl-D** sous Unix, **Ctrl-Z** sous Windows) dans une invite de commande primaire provoque la fermeture de l'interpréteur avec un code de sortie nul. Si cela ne fonctionne pas, vous pouvez fermer l'interpréteur en tapant la commande `quit()`.

Les fonctionnalités d'édition de ligne en mode interactif ne sont pas très sophistiquées. Sur Unix, celui qui à installé Python peut avoir activé le support de *GNU readline*, qui ajoute quelques fonctionnalités d'édition, et la gestion de l'historique. Le moyen probablement le plus rapide pour savoir si l'édition est gérée est de taper **Control-P** au premier prompt Python. Si ça *beep* vous avez l'édition. Voir l'appendice *Édition interactive des entrées et substitution d'historique* à propos des raccourcis clavier. Si rien n'apparaît ou si **^P** est affiché, l'édition de la ligne n'est pas disponible, vous ne pourrez qu'utiliser que la touche retour arrière pour supprimer des caractères de la ligne actuelle.

L'interpréteur fonctionne de façon similaire au shell Unix : lorsqu'il est appelé avec l'entrée standard connectée à un périphérique tty, il lit et exécute les commandes de façon interactive ; lorsqu'il est appelé avec un nom de fichier en argument ou avec un fichier comme entrée standard, il lit et exécute un *script* depuis ce fichier.

Une autre façon de lancer l'interpréteur est `python -c commande [arg]` .... Cela exécute les instructions de *commande* de façon analogue à l'option `-c` du shell. Parce que les instructions Python contiennent souvent des espaces et d'autres caractères spéciaux pour le shell, il est généralement conseillé de mettre *commande* entre guillemets simples.

Certains modules Python sont aussi utiles en tant que scripts. Ils peuvent être appelés avec `python -m module [arg]` ... qui exécute le fichier source de *module* comme si vous aviez tapé son nom complet dans la ligne de commande.

Quand un fichier de script est utilisé, il est parfois utile de pouvoir lancer le script puis d'entrer dans le mode interactif après coup. Cela est possible en passant `-i` avant le script.

Tous les paramètres utilisables en ligne de commande sont documentés dans `using-on-general`.

### 2.1.1 Passage d'arguments

Lorsqu'ils sont connus de l'interpréteur, le nom du script et les arguments additionnels sont représentés sous forme d'une liste assignée à la variable `argv` du module `sys`. Vous pouvez y accéder en exécutant `import sys`. La liste contient au minimum un élément ; quand aucun script ni aucun argument n'est donné, `sys.argv[0]` est une chaîne vide. Quand `-` (qui représente l'entrée standard) est passé comme nom de script, `sys.argv[0]` contient `-`. Quand `-c commande` est utilisé, `sys.argv[0]` contient `-c`. Enfin, quand `-m module` est utilisé, le nom complet du module est assigné à `sys.argv[0]`. Les options trouvées après `-c commande` ou `-m module` ne sont pas lues comme options de l'interpréteur Python mais laissées dans `sys.argv` pour être utilisées par le module ou la commande.

### 2.1.2 Mode interactif

Lorsque des commandes sont lues depuis un tty, l'interpréteur est dit être en *mode interactif*. Dans ce mode, il demande la commande suivante avec le *prompt primaire*, en général trois signes plus-grand-que (`>>>`) ; pour les lignes de continuation, il affiche le *prompt secondaire*, par défaut trois points (`...`). L'interpréteur affiche un message de bienvenue indiquant son numéro de version et une notice de copyright avant d'afficher le premier prompt :

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les lignes de continuation sont nécessaires pour entrer une construction multi-lignes. Par exemple, regardez cette instruction `if` :

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

Pour plus d'informations sur le mode interactif, voir *Mode interactif*.

## 2.2 L'interpréteur et son environnement

### 2.2.1 Encodage du code source

By default, Python source files are treated as encoded in ASCII. To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows :

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

where *encoding* is one of the valid `codecs` supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be :

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a *UNIX « shebang » line*. In this case, the encoding declaration should be added as the second line of the file. For example :

```
#!/usr/bin/env python  
# -*- coding: cp1252 -*-
```



---

## Introduction informelle à Python

---

Dans les exemples qui suivent, les entrées et sorties se distinguent par la présence ou l'absence d'invite (`>>>` et `...`) : pour reproduire les exemples, vous devez taper tout ce qui est après l'invite, au moment où celle-ci apparaît ; les lignes qui n'affichent pas d'invite sont les sorties de l'interpréteur. Notez qu'une invite secondaire affichée seule sur une ligne dans un exemple indique que vous devez entrer une ligne vide ; ceci est utilisé pour terminer une commande multi-lignes.

Beaucoup d'exemples de ce manuel, même ceux saisis à l'invite de l'interpréteur, incluent des commentaires. Les commentaires en Python commencent avec un caractère croisillon, `#`, et s'étendent jusqu'à la fin de la ligne. Un commentaire peut apparaître au début d'une ligne ou à la suite d'un espace ou de code, mais pas à l'intérieur d'une chaîne de caractères littérale. Un caractère croisillon à l'intérieur d'une chaîne de caractères est juste un caractère croisillon. Comme les commentaires ne servent qu'à expliquer le code et ne sont pas interprétés par Python, ils peuvent être ignorés lorsque vous tapez les exemples.

Quelques exemples :

```
# this is the first comment
spam = 1 # and this is the second comment
        # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

### 3.1 Utilisation de Python comme une calculatrice

Essayons quelques commandes Python simples. Démarrez l'interpréteur et attendez l'invite primaire, `>>>`. Ça ne devrait pas être long.

#### 3.1.1 Les nombres

L'interpréteur agit comme une simple calculatrice : vous pouvez lui entrer une expression et il vous affiche la valeur. La syntaxe des expressions est simple : les opérateurs `+`, `-`, `*` et `/` fonctionnent comme dans la plupart des langages (par exemple, Pascal ou C) ; les parenthèses peuvent être utilisées pour faire des regroupements. Par exemple :

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

Les nombre entiers (comme 2, 4, 20) sont de type `int`, alors que les décimaux (comme 5.0, 1.6) sont de type `float`. Vous trouvez plus de détails sur les types numériques plus loin dans ce tutoriel.

Le type donné en résultat d'une division (/) dépend de ses opérandes. Si les deux opérandes sont de type `int`, c'est une *division entière* qui est effectuée, et un `int` est donné. Si l'une des opérandes est un `float`, une division classique est effectuée et un `float` est renvoyé. L'opérateur `//` permet quand à lui d'effectuer des division entières peu importe ses opérandes. Le reste de la division est calculé grâce à l'opérateur `%`

```
>>> 17 / 3 # int / int -> int
5
>>> 17 / 3.0 # int / float -> float
5.666666666666667
>>> 17 // 3.0 # explicit floor division discards the fractional part
5.0
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # result * divisor + remainder
17
```

En Python, il est possible de calculer des puissances avec l'opérateur `**`<sup>1</sup> :

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

Le signe égal (=) est utilisé pour affecter une valeur à une variable. Dans ce cas, aucun résultat n'est affiché avant l'invite suivante :

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Si une variable n'est pas « définie » (si aucune valeur ne lui a été affectée), son utilisation produit une erreur :

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Les nombres à virgule flottante sont tout à fait admis (NdT : Python utilise le point . comme séparateur

---

1. Puisque `**` est prioritaire sur `-`, `-3 ** 2` est interprété `-(3 ** 2)` et vaut donc `-9`. Pour éviter cela et obtenir `9`, utilisez des parenthèses : `(-3) ** 2`.

entre la partie entière et la partie décimale des nombres, c'est la convention anglo-saxonne); les opérateurs avec des opérandes de types différents convertissent l'opérande de type entier en type virgule flottante :

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

En mode interactif, la dernière expression affichée est affectée à la variable `_`. Ainsi, lorsque vous utilisez Python comme calculatrice, cela vous permet de continuer des calculs facilement, par exemple :

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

Cette variable doit être considérée comme une variable en lecture seule par l'utilisateur. Ne lui affectez pas de valeur explicitement — vous créeriez ainsi une variable locale indépendante, avec le même nom, qui masquerait la variable native et son fonctionnement magique.

En plus des `int` et des `float`, il existe les `Decimal` et les `Fraction`. Python gère aussi les nombres complexes, en utilisant le suffixe `j` ou `J` pour indiquer la partie imaginaire (tel que `3+5j`).

### 3.1.2 Chaînes de caractères

En plus des nombres, Python sait aussi manipuler des chaînes de caractères, qui peuvent être exprimées de différentes manières. Elles peuvent être écrites entre guillemets simples (`'...'`) ou entre guillemets (`"..."`) sans distinction<sup>2</sup>. `\` peut être utilisé pour protéger un guillemet :

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Dans une session interactive, la chaîne est affichée entre guillemets simples, et les caractères spéciaux sont protégés par des *backslash*. Bien que cela puisse paraître différent de ce qui a été donné (les guillemets peuvent changer), les deux chaînes sont équivalentes. La chaîne est entre guillemets si elles contient un apostrophe mais aucun guillemet, sinon elle est entre guillemets simples. L'instruction `print` donne un affichage plus lisible : sans guillemets et en affichant les caractères protégés et spéciaux :

2. Contrairement à d'autres langages, les caractères spéciaux comme `\n` ont la même signification entre guillemets (`"..."`) ou entre guillemets simples (`'...'`). La seule différence est que, dans une chaîne entre guillemets, il n'est pas nécessaire de protéger les guillemets simples et vice-versa.

```
>>> "Isn\t," they said.
Isn\t," they said.
>>> print "Isn\t," they said.
Isn\t," they said.
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print, \n is included in the output
First line.\nSecond line.
>>> print s # with print, \n produces a new line
First line.
Second line.
```

Si vous ne voulez pas que les caractères précédés d'un \ soient interprétés comme étant spéciaux, utilisez les chaînes brutes (raw strings en anglais) en préfixant la chaîne d'un r :

```
>>> print 'C:\some\name' # here \n means newline!
C:\some
ame
>>> print r'C:\some\name' # note the r before the quote
C:\some\name
```

Les chaînes de caractères peuvent s'étendre sur plusieurs lignes. Utilisez alors des triples guillemets, simples ou doubles : '''...''' ou """...""". Les retours à la ligne sont automatiquement inclus, mais on peut l'empêcher en ajoutant \ à la fin de la ligne. L'exemple suivant :

```
print """\
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
"""
```

produit l'affichage suivant (notez que le premier retour à la ligne n'est pas inclus) :

```
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

Les chaînes peuvent être concaténées (collées ensemble) avec l'opérateur + et répétées avec l'opérateur \* :

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Plusieurs chaînes de caractères, écrites littéralement (c'est à dire entre guillemets), côte à côte, sont automatiquement concaténées.

```
>>> 'Py' 'thon'
'Python'
```

Cette fonctionnalité est surtout intéressante pour couper des chaînes trop longues :

```
>>> text = ('Put several strings within parentheses '
...        'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Cela ne fonctionne cependant qu'avec les chaînes littérales, pas avec les variables ni les expressions :

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
...
SyntaxError: invalid syntax
```

Pour concaténer des variables, ou des variables avec des chaînes littérales, utilisez l'opérateur + :

```
>>> prefix + 'thon'
'Python'
```

Les chaînes de caractères peuvent être indexées (i.e. on peut accéder aux caractères par leur position), le premier caractère d'une chaîne étant à la position 0. Il n'existe pas de type distinct pour les caractères, un caractère est simplement une chaîne de longueur 1 :

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Les indices peuvent également être négatifs, on compte alors en partant de la droite. Par exemple :

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Notez que, comme -0 égale 0, les indices négatifs commencent par -1.

En plus de l'indexation, le découpage (*slicing*) est géré. L'indexation est utilisée pour obtenir les caractères individuellement, alors que le découpage permet d'obtenir une sous-chaîne.

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Notez que le début est toujours inclus et la fin toujours exclue. Cela assure que `s[:i] + s[i:]` est toujours égal à `s` :

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Les valeurs par défaut des indices de tranches ont une utilité ; le premier indice vaut zéro par défaut (i.e. lorsqu'il est omis), le deuxième correspond par défaut à la taille de la chaîne de caractères :

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:]   # characters from the second-last (included) to the end
'on'
```

Pour mémoriser la façon dont les tranches fonctionnent, vous pouvez imaginer que les indices pointent *entre* les caractères, le côté gauche du premier caractère ayant la position 0. Le côté droit du dernier caractère d'une chaîne de  $n$  caractères a alors pour indice  $n$ . Par exemple :

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

La première ligne de nombres donne la position des indices 0...6 dans la chaîne ; la deuxième ligne donne l'indice négatif correspondant. La tranche de  $i$  à  $j$  est constituée de tous les caractères situés entre les bords libellés  $i$  et  $j$ , respectivement.

Pour des indices non négatifs, la longueur d'une tranche est la différence entre ces indices, si les deux sont entre les bornes. Par exemple, la longueur de `word[1:3]` est 2.

Utiliser un indice trop grand produit une erreur :

```
>>> word[42]    # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Cependant, les indices hors bornes sont gérés silencieusement lorsqu'ils sont utilisés dans des tranches :

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Les chaînes de caractères, en Python, ne peuvent pas être modifiées. On dit qu'elles sont *immuables*. Affecter une nouvelle valeur à un indice dans une chaîne produit une erreur :

```
>>> word[0] = 'J'
...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
...
TypeError: 'str' object does not support item assignment
```

Si vous avez besoin d'une chaîne différente, vous devez en créer une nouvelle :

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

La fonction native `len()` renvoie la longueur d'une chaîne :

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

Voir aussi :

**typeseq** Les chaînes de caractères, et les chaînes Unicode décrites dans la prochaine section, sont des exemples de *types de séquences*, et supportent donc les opérations classiques prises en charge par ces types.

**string-methods** Aussi bien les chaînes de caractères que les chaînes Unicode supportent un large éventail de méthodes de transformations basiques et de recherche.

**formatstrings** Informations sur le formatage des chaînes avec la méthode `str.format()`.

**string-formatting** Les anciennes opérations de formatage appelées lorsque les chaînes de caractères et les chaînes Unicode sont les opérandes placés à gauche de l'opérateur `%` sont décrites plus en détail ici.

### 3.1.3 Chaînes Unicode

À partir de Python 2.0, un nouveau type permettant de stocker du texte est mis à la disposition du programmeur : le type Unicode. Il peut être utilisé pour stocker et manipuler des données Unicode (voir <http://www.unicode.org/>) et s'intègre très bien avec les types de chaînes de caractères existant, en fournissant une conversion automatique lorsque c'est nécessaire.

Unicode a l'avantage de fournir une valeur ordinaire pour chaque caractère d'un script utilisé aussi bien dans d'anciens textes que dans des textes modernes. Auparavant, il n'y avait que 256 valeurs ordinales possibles pour chaque caractère. Chaque texte était typiquement associé à une page de codes qui associait une valeur ordinaire à chaque caractère. Ceci conduisait à une grande confusion, notamment pour tout ce qui touchait à l'internationalisation (souvent écrite *i18n* — 'i' + 18 caractères + 'n') des logiciels. Unicode résout ces problèmes en définissant une page de code unique pour tous les scripts.

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

```
>>> u'Hello World !'
u'Hello World !'
```

Le petit 'u' qui précède l'apostrophe indique que l'on veut créer une chaîne Unicode. Si vous voulez intégrer des caractères spéciaux dans la chaîne, vous pouvez le faire en utilisant l'encodage Python d'échappement des caractères Unicode. Comme dans cet exemple

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

La séquence d'échappement `\u0020` indique d'insérer le caractère Unicode de valeur ordinaire 0x0020 (un espace) à la position indiquée.

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

Pour les experts, il existe également un mode « brut » identique à celui disponible pour les chaînes de caractères. Vous devez préfixer la première apostrophe avec «ur» pour que Python utilise l'encodage *Unicode-Brut*. Il n'appliquera la conversion `\uXXXX` de l'exemple ci-dessous que s'il y a un nombre impair d'antislashes devant le petit «u»

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\u0020World !'
```

Le mode brut est le plus utile lorsque vous devez saisir de nombreux antislashes, comme il peut être nécessaire de le faire dans des expressions rationnelles.

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

La primitive `unicode()` donne accès à tous les codecs (CODEurs et DECodeurs) enregistrés. Certains des encodages les plus connus pris en charge par ces codecs sont *Latin-1*, *ASCII*, *UTF-8* et *UTF-16*. Les deux derniers sont des encodages à longueur variable qui stockent chaque caractère Unicode dans un ou plusieurs octets. L'encodage par défaut est normalement défini en ASCII, qui gère les caractères de valeur ordinaire 0 à 127 et rejette tous les autres caractères avec une erreur. Quand une chaîne Unicode est imprimée, écrite dans un fichier ou convertie avec la fonction `str()`, une conversion s'effectue en utilisant cet encodage par défaut

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xfc\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not
↳in range(128)
```

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

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

Si vous avez une donnée dans un encodage spécifique et voulez générer la chaîne Unicode correspondante, vous pouvez utiliser la fonction `unicode()` en fournissant le nom de l'encodage comme second argument

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

### 3.1.4 Listes

Python connaît différents types de données *combinés*, utilisés pour regrouper plusieurs valeurs. La plus souple est la *liste*, qui peut être écrite comme une suite, placée entre crochets, de valeurs (éléments) séparés par des virgules. Les éléments d'une liste ne sont pas obligatoirement tous du même type, bien qu'à l'usage ce soit souvent le cas.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Comme les chaînes de caractères (et toute autre type de *sequence*), les listes peuvent être indicées et découpées :

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Toutes les opérations par tranches renvoient une nouvelle liste contenant les éléments demandés. Cela signifie que l'opération suivante renvoie une copie (superficielle) de la liste :

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Les listes gèrent aussi les opérations comme la concaténation :

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Mais à la différence des chaînes qui sont *immuables*, les listes sont *muables* : il est possible de modifier leur contenu :

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Il est aussi possible d'ajouter de nouveaux éléments à la fin d'une liste avec la méthode `append()` (les méthodes sont abordées plus tard) :

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Des affectations de tranches sont également possibles, ce qui peut même modifier la taille de la liste ou la vider complètement :

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> letters[:] = []
>>> letters
[]
```

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

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Il est possible d'imbriquer des listes (i.e. créer des listes contenant d'autres listes). Par exemple :

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

## 3.2 Premiers pas vers la programmation

Bien entendu, on peut utiliser Python pour des tâches plus compliquées que d'additionner deux et deux. Par exemple, on peut écrire le début de la suite de Fibonacci comme ceci :

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Cet exemple introduit plusieurs nouvelles fonctionnalités.

- La première ligne contient une *affectation multiple* : les variables `a` et `b` se voient affecter simultanément leurs nouvelles valeurs 0 et 1. Cette méthode est encore utilisée à la dernière ligne, pour démontrer que les expressions sur la partie droite de l'affectation sont toutes évaluées avant que les affectations ne soient effectuées. Ces expressions en partie droite sont toujours évaluées de la gauche vers la droite.
- La boucle `while` s'exécute tant que la condition (ici : `b < 10`) reste vraie. En Python, comme en C, tout entier différent de zéro est vrai et zéro est faux. La condition peut aussi être une chaîne de caractères, une liste, ou en fait toute séquence ; une séquence avec une valeur non nulle est vraie, une séquence vide est fausse. Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs de comparaison standards sont écrits comme en C : `<` (inférieur), `>` (supérieur), `==` (égal), `<=` (inférieur ou égal), `>=` (supérieur ou égal) et `!=` (non égal).

- Le *corps* de la boucle est *indenté* : l'indentation est la méthode utilisée par Python pour regrouper des instructions. En mode interactif, vous devez saisir une tabulation ou des espaces pour chaque ligne indentée. En pratique, vous aurez intérêt à utiliser un éditeur de texte pour les saisies plus compliquées ; tous les éditeurs de texte dignes de ce nom disposent d'une fonction d'auto-indentation. Lorsqu'une expression composée est saisie en mode interactif, elle doit être suivie d'une ligne vide pour indiquer qu'elle est terminée (car l'analyseur ne peut pas deviner que vous venez de saisir la dernière ligne). Notez bien que toutes les lignes à l'intérieur d'un bloc doivent être indentées au même niveau.
- L'instruction `print` écrit la valeur de l'expression qui lui est fournie. Ce n'est pas la même chose que d'écrire l'expression que vous voulez écrire (comme nous l'avons fait dans l'exemple de la calculatrice), et diffère dans la façon dont elle gère les expressions et chaînes de caractères multiples. Les chaînes sont imprimées sans apostrophes et un espace est inséré entre les éléments de telle sorte que vous pouvez facilement formater les choses, comme ceci :

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

Une virgule à la fin de l'instruction supprime le saut de ligne

```
>>> 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 saut de ligne avant d'afficher l'invite suivante si la dernière ligne n'était pas complète.

## Notes



En plus de l'instruction `while` qui vient d'être présentée, Python dispose des instructions de contrôle de flux classiques que l'on trouve dans d'autres langages, mais toujours avec ses propres tournures.

### 4.1 L'instruction `if`

L'instruction `if` est sans doute la plus connue. Par exemple :

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

Il peut y avoir un nombre quelconque de parties `elif` et la partie `else` est facultative. Le mot clé `elif` est un raccourci pour *else if*, mais permet de gagner un niveau d'indentation. Une séquence `if ... elif ... elif ...` est par ailleurs équivalente aux instructions `switch` ou `case` disponibles dans d'autres langages.

### 4.2 L'instruction `for`

L'instruction `for` que propose Python est un peu différente de celle que l'on peut trouver en C ou en Pascal. Au lieu de toujours itérer sur une suite arithmétique de nombres (comme en Pascal), ou de donner à l'utilisateur la possibilité de définir le pas d'itération et la condition de fin (comme en C), l'instruction `for`

en Python itère sur les éléments d'une séquence (qui peut être une liste, une chaîne de caractères...), dans l'ordre dans lequel ils apparaissent dans la séquence. Par exemple :

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
...
cat 3
window 6
defenestrate 12
```

Si vous devez modifier la séquence sur laquelle s'effectue l'itération à l'intérieur de la boucle (par exemple pour dupliquer ou supprimer un élément), il est plus que recommandé de commencer par en faire une copie, celle-ci n'étant pas implicite. La notation « par tranches » rend cette opération particulièrement simple :

```
>>> for w in words[:]: # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

## 4.3 La fonction range()

Si vous devez itérer sur une suite de nombres, la fonction intégrée `range()` est faite pour cela. Elle génère une liste contenant une simple progression arithmétique :

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

Le dernier élément fourni en paramètre ne fait jamais partie de la liste générée ; `range(10)` génère une liste de 10 valeurs, dont les valeurs vont de 0 à 9. Il est possible de spécifier une valeur de début et/ou une valeur d'incrément différente(s) (y compris négative pour cette dernière, que l'on appelle également parfois le “pas”)

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

Pour itérer sur les indices d'une séquence, on peut combiner les fonctions `range()` et `len()` :

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Cependant, dans la plupart des cas, il est plus pratique d'utiliser la fonction `enumerate()`. Voyez pour cela *Techniques de boucles*.

## 4.4 Les instructions `break`, `continue` et les clauses `else` au sein des boucles

The `break` statement, like in C, breaks out of the innermost enclosing `for` or `while` loop.

Les boucles peuvent également disposer d'une instruction `else`; celle-ci est exécutée lorsqu'une boucle se termine alors que tous ses éléments ont été traités (dans le cas d'un `for`) ou que la condition devient fausse (dans le cas d'un `while`), mais pas lorsque la boucle est interrompue par une instruction `break`. L'exemple suivant, qui effectue une recherche de nombres premiers, en est une démonstration :

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...         else:
...             # loop fell through without finding a factor
...             print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Oui, ce code est correct. Regardez attentivement : l'instruction `else` est rattachée à la boucle `for`, et `non` à l'instruction `if`.)

Lorsqu'elle est utilisée dans une boucle, la clause `else` est donc plus proche de celle associée à une instruction `try` que de celle associée à une instruction `if` : la clause `else` d'une instruction `try` s'exécute lorsqu'aucune exception n'est déclenchée, et celle d'une boucle lorsque aucun `break` n'intervient. Plus d'informations sur l'instruction `try` et le traitement des exceptions, consultez *Gestion des exceptions*.

L'instruction `continue`, également empruntée au C, fait passer la boucle à son itération suivante :

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

## 4.5 L'instruction `pass`

L'instruction `pass` ne fait rien. Elle peut être utilisée lorsqu'une instruction est nécessaire pour fournir une syntaxe correcte, mais qu'aucune action ne doit être effectuée. Par exemple :

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

On utilise couramment cette instruction pour créer des classes minimales :

```
>>> class MyEmptyClass:
...     pass
...
```

Un autre cas d'utilisation du `pass` est de réserver un espace en phase de développement pour une fonction ou un traitement conditionnel, vous permettant ainsi de construire votre code à un niveau plus abstrait. L'instruction `pass` est alors ignorée silencieusement :

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

## 4.6 Définir des fonctions

On peut créer une fonction qui écrit la suite de Fibonacci jusqu'à une limite imposée :

```
>>> def fib(n): # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Le mot-clé `def` introduit une *définition* de fonction. Il doit être suivi du nom de la fonction et d'une liste, entre parenthèses, de ses paramètres. L'instruction qui constitue le corps de la fonction débute à la ligne suivante et doit être indentée.

La première instruction d'une fonction peut, de façon facultative, être une chaîne de caractères littérale ; cette chaîne de caractères sera alors la chaîne de documentation de la fonction, appelée *docstring* (consultez la section *Chaînes de documentation* pour en savoir plus). Il existe des outils qui utilisent ces chaînes de documentation pour générer automatiquement une documentation en ligne ou imprimée, ou pour permettre à l'utilisateur de naviguer de façon interactive dans le code ; prenez-en l'habitude, c'est une bonne pratique que de documenter le code que vous écrivez !

L'exécution d'une fonction introduit une nouvelle table de symboles utilisée par les variables locales de la fonction. Plus précisément, toutes les affectations de variables effectuées au sein d'une fonction stockent la valeur dans la table de symboles locale ; en revanche, les références de variables sont recherchées dans la table de symboles locale, puis dans la table de symboles locale des fonctions englobantes, puis dans la table de symboles globale et finalement dans la table de noms des primitives. Par conséquent, bien qu'elles puissent

être référencées, il est impossible d'affecter une valeur à une variable globale (sauf en utilisant une instruction `global`).

Les paramètres effectifs (arguments) d'une fonction sont introduits dans la table de symboles locale de la fonction appelée au moment où elle est appelée; par conséquent, les passages de paramètres se font *par valeur*, la *valeur* étant toujours une *référence* à un objet et non la valeur de l'objet lui-même.<sup>1</sup> Lorsqu'une fonction appelle une autre fonction, une nouvelle table de symboles locale est créée pour cet appel.

Une définition de fonction introduit le nom de la fonction dans la table de symboles courante. La valeur du nom de la fonction est un type qui est reconnu par l'interpréteur comme une fonction utilisateur. Cette valeur peut être affectée à un autre nom qui pourra alors être utilisé également comme une fonction. Ceci fournit un mécanisme de renommage général :

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Si vous venez d'autres langages, vous pouvez penser que `fib` n'est pas une fonction mais une procédure, puisqu'elle ne renvoie pas de résultat. En fait, même les fonctions sans instruction `return` renvoient une valeur, quoique ennuyeuse. Cette valeur est appelée `None` (c'est le nom d'une primitive). Écrire la valeur `None` est normalement supprimé par l'interpréteur lorsqu'il s'agit de la seule valeur écrite. Vous pouvez le voir si vous y tenez vraiment en utilisant `print`

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

Il est facile d'écrire une fonction qui renvoie une liste de la série de Fibonacci au lieu de l'afficher :

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100             # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Cet exemple, comme d'habitude, illustre de nouvelles fonctionnalités de Python :

- L'instruction `return` provoque la sortie de la fonction en renvoyant une valeur. `return` sans expression en paramètre renvoie `None`. Arriver à la fin d'une fonction renvoie également `None`.
- L'instruction `result.append(a)` appelle une *méthode* de l'objet `result` qui est une liste. Une méthode est une fonction qui « appartient » à un objet et qui est nommée `obj.methodname`, où `obj` est un objet (il peut également s'agir d'une expression) et `methodname` est le nom d'une méthode que le type de l'objet définit. Différents types définissent différentes méthodes. Des méthodes de différents types peuvent porter le même nom sans qu'il n'y ait d'ambiguïté (vous pouvez définir vos propres types d'objets et leurs méthodes en utilisant des *classes*, voir [Classes](#)). La méthode `append()` donnée dans

1. En fait, *appels par référence d'objets* serait sans doute une description plus juste dans la mesure où, si un objet muable est passé en argument, l'appelant verra toutes les modifications qui lui auront été apportées par l'appelé (insertion d'éléments dans une liste...).

cet exemple est définie pour les listes ; elles ajoute un nouvel élément à la fin de la liste. Dans cet exemple, elle est l'équivalent de `result = result + [a]`, mais elle est plus efficace.

## 4.7 D'avantage sur la définition des fonctions

Il est également possible de définir des fonctions avec un nombre variable d'arguments. Trois syntaxes peuvent être utilisées, éventuellement combinées.

### 4.7.1 Valeur par défaut des arguments

La forme la plus utile consiste à indiquer une valeur par défaut pour certains arguments. Ceci crée une fonction qui pourra être appelée avec moins d'arguments que ceux présents dans sa définition. Par exemple :

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Cette fonction peut être appelée de plusieurs façons :

- en ne fournissant que les arguments obligatoires : `ask_ok('Do you really want to quit?')`
- en fournissant une partie des arguments facultatifs : `ask_ok('OK to overwrite the file?', 2)`
- en fournissant tous les arguments : `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Cet exemple présente également le mot-clé `in`. Celui-ci permet de tester si une séquence contient une certaine valeur.

Les valeurs par défaut sont évaluées lors de la définition de la fonction dans la portée de *définition*, de telle sorte que :

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

Affiche 5.

**Avertissement important :** la valeur par défaut n'est évaluée qu'une seule fois. Ceci fait une différence lorsque cette valeur par défaut est un objet muable tel qu'une liste, un dictionnaire ou des instances de la plupart des classes. Par exemple, la fonction suivante accumule les arguments qui lui sont passés au fil des appels successifs :

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

(suite sur la page suivante)

(suite de la page précédente)

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

Ceci affiche :

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

Si vous ne voulez pas que cette valeur par défaut soit partagée entre des appels successifs, vous pouvez écrire la fonction de cette façon :

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

### 4.7.2 Les arguments nommés

Les fonctions peuvent également être appelées en utilisant des *arguments nommés* sous la forme `kwarg=value`. Par exemple, la fonction suivante :

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

accepte un argument obligatoire (`voltage`) et trois arguments facultatifs (`state`, `action` et `type`). Cette fonction peut être appelée de n'importe laquelle des façons suivantes :

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')   # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)   # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

mais tous les appels qui suivent sont incorrects :

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')              # non-keyword argument after a keyword argument
parrot(110, voltage=220)                  # duplicate value for the same argument
parrot(actor='John Cleese')              # unknown keyword argument
```

Dans un appel de fonction, les arguments nommés doivent suivre les arguments positionnés. Tous les arguments nommés doivent correspondre à l'un des arguments acceptés par la fonction (par exemple, `actor` n'est pas un argument accepté par la fonction `parrot`), mais leur ordre n'est pas important. Ceci inclut également les arguments facultatifs (`parrot(voltage=1000)` est également correct). Aucun argument ne peut recevoir une valeur plus d'une fois, comme l'illustre cet exemple incorrect du fait de cette restriction :

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for keyword argument 'a'
```

Quand un dernier paramètre formel est présent sous la forme **\*\*name**, il reçoit un dictionnaire (voir types-mapping) contenant tous les arguments nommés à l'exception de ceux correspondant à un paramètre formel. Ceci peut être combiné à un paramètre formel sous la forme **\*name** (décrit dans la section suivante) qui lui reçoit un tuple contenant les arguments positionnés au-delà de la liste des paramètres formels (**\*name** doit être présent avant **\*\*name**). Par exemple, si vous définissez une fonction comme ceci :

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

Elle pourrait être appelée comme ceci :

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

et, bien sûr, elle affiche :

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Notez que la liste des arguments nommés est créée en classant les clés du dictionnaire extraites par la méthode `keys()` avant de les afficher. Si cela n'est pas fait, l'ordre dans lequel les arguments sont affichés n'est pas défini.

### 4.7.3 Listes d'arguments arbitraires

Pour terminer, l'option la moins fréquente consiste à indiquer qu'une fonction peut être appelée avec un nombre arbitraire d'arguments. Ces arguments sont intégrés dans un tuple (voir *Tuples et séquences*). Avant le nombre variable d'arguments, zéro arguments normaux ou plus peuvent apparaître :

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

#### 4.7.4 Séparation des listes d'arguments

La situation inverse intervient lorsque les arguments sont déjà dans une liste ou un tuple mais doivent être séparés pour un appel de fonction nécessitant des arguments positionnés séparés. Par exemple, la primitive `range()` attend des arguments *start* et *stop* distincts. S'ils ne sont pas disponibles séparément, écrivez l'appel de fonction en utilisant l'opérateur `*` pour séparer les arguments présents dans une liste ou un tuple :

```
>>> range(3, 6)                # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)               # call with arguments unpacked from a list
[3, 4, 5]
```

De la même façon, les dictionnaires peuvent fournir des arguments nommés en utilisant l'opérateur `**` :

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin'
↳ demised !
```

#### 4.7.5 Fonctions anonymes

Avec le mot-clé `lambda`, vous pouvez créer de petites fonctions anonymes. En voici une qui renvoie la somme de ses deux arguments : `lambda a, b: a+b`. Les fonctions `lambda` peuvent être utilisées partout où un objet fonction est attendu. Elles sont syntaxiquement restreintes à une seule expression. Sémantiquement, elles ne sont que du sucre syntaxique pour une définition de fonction normale. Comme les fonctions imbriquées, les fonctions `lambda` peuvent référencer des variables de la portée englobante :

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

L'exemple précédent utilise une fonction anonyme pour renvoyer une fonction. Une autre utilisation classique est de donner une fonction minimaliste directement en tant que paramètre :

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

### 4.7.6 Chaînes de documentation

Il existe des conventions émergentes concernant le contenu et le format des chaînes de documentation.

La première ligne devrait toujours être courte et résumer de manière concise l'utilité de l'objet. Afin d'être bref, nul besoin de rappeler le nom de l'objet ou son type, qui sont accessibles par d'autres moyens (sauf si le nom est un verbe qui décrit une opération). Cette ligne devrait commencer avec une majuscule et se terminer par un point.

S'il y a d'autres lignes dans la chaîne de documentation, la deuxième ligne devrait être vide, pour la séparer visuellement du reste de la description. Les autres lignes peuvent alors constituer un ou plusieurs paragraphes décrivant le mode d'utilisation de l'objet, ses effets de bord, etc.

L'analyseur de code Python ne supprime pas l'indentation des chaînes de caractères littérales multi-lignes, donc les outils qui utilisent la documentation doivent si besoin faire cette opération eux-mêmes. La convention suivante s'applique : la première ligne non vide *après* la première détermine la profondeur d'indentation de l'ensemble de la chaîne de documentation (on ne peut pas utiliser la première ligne qui est généralement accolée aux guillemets d'ouverture de la chaîne de caractères et dont l'indentation n'est donc pas visible). Les espaces « correspondant » à cette profondeur d'indentation sont alors supprimés du début de chacune des lignes de la chaîne. Aucune ligne ne devrait présenter un niveau d'indentation inférieur mais si cela arrive, toutes les espaces situés en début de ligne doivent être supprimées. L'équivalent des espaces doit être testé après expansion des tabulations (normalement remplacés par 4 espaces).

Voici un exemple de chaîne de documentation multi-lignes :

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

## 4.8 Aparté : le style de codage

Maintenant que vous êtes prêt à écrire des programmes plus longs et plus complexes, il est temps de parler du *style de codage*. La plupart des langages peuvent être écrits (ou plutôt *formatés*) selon différents styles ; certains sont plus lisibles que d'autres. Rendre la lecture de votre code plus facile aux autres est toujours une bonne idée et adopter un bon style de codage peut énormément vous y aider.

En Python, la plupart des projets adhèrent au style défini dans la [PEP 8](#) ; elle met en avant un style de codage très lisible et agréable à l'œil. Chaque développeur Python se doit donc de la lire et de s'en inspirer autant que possible ; voici ses principaux points notables :

- Utilisez des indentations de 4 espaces et pas de tabulation.  
4 espaces constituent un bon compromis entre une indentation courte (qui permet une profondeur d'imbrication plus importante) et une longue (qui rend le code plus facile à lire). Les tabulations introduisent de la confusion et doivent être proscrites autant que possible.
- Faites en sorte que les lignes ne dépassent pas 79 caractères, au besoin en insérant des retours à la ligne.  
Vous facilitez ainsi la lecture pour les utilisateurs qui n'ont qu'un petit écran et, pour les autres, cela leur permet de visualiser plusieurs fichiers côte à côte.

- Utilisez des lignes vides pour séparer les fonctions et les classes, ou pour scinder de gros blocs de code à l'intérieur de fonctions.
- Lorsque c'est possible, placez les commentaires sur leur propres lignes.
- Utilisez les chaînes de documentation.
- Utilisez des espaces autour des opérateurs et après les virgules, mais pas juste à l'intérieur des parenthèses : `a = f(1, 2) + g(3, 4)`.
- Nommez toujours vos classes et fonctions de la même manière ; la convention est d'utiliser une notation **CamelCase** pour les classes, et **minuscules\_avec\_trait\_bas** pour les fonctions et méthodes. Utilisez toujours **self** comme nom du premier argument des méthodes (voyez *[Une première approche des classes](#)* pour en savoir plus sur les classes et les méthodes).
- N'utilisez pas d'encodages exotiques dès lors que votre code est sensé être utilisé dans des environnements internationaux. L'ASCII est encore ce qui marche le mieux dans la plupart des cas.

## Notes



Ce chapitre reprend plus en détail quelques points déjà décrits précédemment et introduit également de nouvelles notions.

## 5.1 Compléments sur les listes

Le type liste dispose de méthodes supplémentaires. Voici toutes les méthodes des objets de type liste :

**list.append(*x*)**

Ajoute un élément à la fin de la liste ; équivalent à `a[len(a):] = [x]`.

**list.extend(*L*)**

Étend la liste en y ajoutant tous les éléments de la liste fournie ; équivalent à `a[len(a):] = L`.

**list.insert(*i*, *x*)**

Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste et `a.insert(len(a), x)` est équivalent à `a.append(x)`.

**list.remove(*x*)**

Supprime de la liste le premier élément dont la valeur est *x*. Une exception est levée s'il existe aucun élément avec cette valeur.

**list.pop([*i*])**

Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, `a.pop()` enlève et renvoie le dernier élément de la liste (les crochets autour du *i* dans la signature de la méthode indiquent que ce paramètre est facultatif et non que vous devez placer des crochets dans votre code ! Vous retrouverez cette notation fréquemment dans le Guide de Référence de la Bibliothèque Python).

**list.index(*x*)**

Retourne la position du premier élément de la liste ayant la valeur *x*. Une exception est levée s'il n'existe aucun élément avec cette valeur.

**list.count(*x*)**

Renvoie le nombre d'éléments ayant la valeur *x* dans la liste.

```
list.sort(cmp=None, key=None, reverse=False)
```

Classe les éléments sur place (les arguments peuvent personnaliser le classement, voir `sorted()` pour leur explication).

```
list.reverse()
```

Inverse l'ordre des éléments de la liste, en place.

L'exemple suivant utilise la plupart des méthodes des listes :

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. This is a design principle for all mutable data structures in Python.

### 5.1.1 Utilisation des listes comme des piles

Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti » ou LIFO pour *last-in, first-out* en anglais). Pour ajouter un élément sur la pile, utilisez la méthode `append()`. Pour récupérer l'objet au sommet de la pile, utilisez la méthode `pop()` sans indicateur de position. Par exemple :

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

(suite sur la page suivante)

(suite de la page précédente)

```
5
>>> stack
[3, 4]
```

### 5.1.2 Utilisation des listes comme des files

Il est également possible d'utiliser une liste comme une file, où le premier élément ajouté est le premier récupéré (« premier entré, premier sorti » ou FIFO pour *first-in, first-out*) ; toutefois, les listes ne sont pas très efficaces pour réaliser ce type de traitement. Alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertions ou de retraits en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).

Pour implémenter une file, utilisez la classe `collections.deque` qui a été conçue pour réaliser rapidement les opérations d'ajouts et de retraits aux deux extrémités. Par exemple :

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                # The first to arrive now leaves
'Eric'
>>> queue.popleft()                # The second to arrive now leaves
'John'
>>> queue                          # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

### 5.1.3 Outils de programmation fonctionnelle

Il existe trois fonctions primitives qui sont très utiles lorsqu'elles sont utilisées avec des listes : `filter()`, `map()` et `reduce()`.

`filter(function, sequence)` retourne une séquence composée des éléments de la séquence initiale pour lesquels la fonction `function(item)` est vraie. Si `sequence` est de type `str`, `unicode`, ou `tuple`, le résultat sera du même type ; sinon, il sera toujours de type `list`. Par exemple, pour construire une séquence de nombres divisibles par 2 ou 5

```
>>> def f(x): return x % 3 == 0 or x % 5 == 0
...
>>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

`map(function, sequence)` appelle `function(item)` pour chaque élément de la séquence et retourne une liste contenant l'ensemble des résultats. Par exemple, pour calculer des cubes

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

Plus d'une séquence peut être passée en paramètre ; dans ce cas, la fonction doit avoir autant d'arguments qu'il y a de séquences, et elle sera appelée avec l'élément correspondant de chaque séquence (ou `None` si certaines séquences sont plus courtes que d'autres).

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` retourne une seule valeur construite en appelant la fonction *function* avec comme paramètres les deux premiers éléments de la séquence, puis le premier résultat et l'élément suivant, et ainsi de suite. Par exemple, pour calculer la somme des nombres de 1 à 10

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

S'il n'y a qu'un seul élément dans la séquence, sa valeur est retournée ; si la séquence est vide, une exception est levée.

Un troisième paramètre peut être ajouté pour indiquer la valeur de départ. Dans ce cas, cette valeur est retournée dans le cas d'une séquence vide ; sinon, la fonction est d'abord appelée avec comme paramètres cette valeur de départ et le premier élément de la séquence, puis ce premier résultat et l'élément suivant, et ainsi de suite. Par exemple

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

N'utilisez pas la fonction `sum()` fournie dans cet exemple. Effectuer des sommes de nombres est un besoin tellement courant qu'il existe une fonction native `sum(sequence)` qui fait exactement la même chose.

### 5.1.4 Compréhensions de listes

Les compréhensions de listes fournissent un moyen de construire des listes de manière très concise. Une application classique est la construction de nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque élément d'une autre séquence ; ou de créer une sous-séquence des éléments satisfaisant une condition spécifique.

Par exemple, supposons que l'on veuille créer une liste de carrés, comme :

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut obtenir le même résultat avec

```
squares = [x**2 for x in range(10)]
```

C'est également l'équivalent de `squares = map(lambda x: x**2, range(10))`, mais en plus concis et plus lisible.

Une compréhension de liste consiste à placer entre crochets une expression suivie par une clause `for` puis par zéro ou plus clauses `for` ou `if`. Le résultat est une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent. Par exemple, cette compréhension de liste combine les éléments de deux listes s'ils ne sont pas égaux :

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

et c'est équivalent à :

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Notez que l'ordre des instructions `for` et `if` est le même dans ces différents extraits de code.

Si l'expression est un tuple (c'est-à-dire `(x, y)` dans cet exemple), elle doit être entourée par des parenthèses :

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
    ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Les compréhensions de listes peuvent contenir des expressions complexes et des fonctions imbriquées :

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

### Compréhensions de listes imbriquées

La première expression dans une compréhension de liste peut être n'importe quelle expression, y compris une autre compréhension de liste.

Voyez l'exemple suivant d'une matrice de 3 par 4, implémentée sous la forme de 3 listes de 4 éléments :

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Cette compréhension de liste transpose les lignes et les colonnes :

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Comme nous l'avons vu dans la section précédente, la compréhension de liste imbriquée est évaluée dans le contexte de l'instruction `for` qui la suit, donc cet exemple est équivalent à :

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

lequel à son tour est équivalent à :

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Dans des cas concrets, il est toujours préférable d'utiliser des fonctions natives plutôt que des instructions de contrôle de flux complexes. La fonction `zip()` ferait dans ce cas un excellent travail :

```
>>> zip(*matrix)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Voyez *Séparation des listes d'arguments* pour plus de détails sur l'astérisque de cette ligne.

## 5.2 L'instruction del

Il existe un moyen de retirer un élément d'une liste à partir de sa position au lieu de sa valeur : l'instruction `del`. Elle diffère de la méthode `pop()` qui, elle, renvoie une valeur. L'instruction `del` peut également être utilisée pour supprimer des tranches d'une liste ou la vider complètement (ce que nous avons fait auparavant en affectant une liste vide à la tranche). Par exemple :

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` peut aussi être utilisée pour supprimer des variables :

```
>>> del a
```

À partir de là, référencer le nom `a` est une erreur (au moins jusqu'à ce qu'une autre valeur lui soit affectée). Vous trouverez d'autres utilisations de la fonction `del` plus tard.

## 5.3 Tuples et séquences

Nous avons vu que les listes et les chaînes de caractères ont beaucoup de propriétés en commun, comme l'indexation et les opérations sur des tranches. Ce sont deux exemples de *séquences* (voir `typeseq`). Comme Python est un langage en constante évolution, d'autres types de séquences y seront peut-être ajoutés. Il existe également un autre type standard de séquence : le *tuple*.

Un tuple consiste en différentes valeurs séparées par des virgules, comme par exemple :

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
```

*# Tuples may be nested:*

```
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

*# Tuples are immutable:*

```
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

*# but they can contain mutable objects:*

```
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Comme vous pouvez le voir, les tuples sont toujours affichés entre parenthèses, de façon à ce que des tuples

imbriqués soient interprétés correctement ; ils peuvent être entrés avec ou sans parenthèses, même si celles-ci sont souvent nécessaires (notamment lorsqu'un tuple fait partie d'une expression plus longue). Il n'est pas possible d'affecter de valeur à un élément d'un tuple ; par contre, il est possible de créer des tuples contenant des objets muables, comme des listes.

Si les tuples peuvent sembler similaires aux listes, ils sont souvent utilisés dans des cas différents et pour des raisons différentes. Les tuples sont *immuables* et contiennent souvent des séquences hétérogènes d'éléments qui sont accédés par « déballage » (voir plus loin) ou indexation (ou même par attributs dans le cas des *namedtuples*). Les listes sont souvent *muables* et contiennent des éléments homogènes qui sont accédés par itération sur la liste.

Un problème spécifique est la construction de tuples ne contenant aucun ou un seul élément : la syntaxe a quelques tournures spécifiques pour s'en accommoder. Les tuples vides sont construits par une paire de parenthèses vides ; un tuple avec un seul élément est construit en faisant suivre la valeur par une virgule (il n'est pas suffisant de placer cette valeur entre parenthèses). Pas très joli, mais efficace. Par exemple :

```
>>> empty = ()
>>> singleton = 'hello',      # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

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

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

Ceci est appelé, de façon plus ou moins appropriée, un *déballage de séquence* et fonctionne pour toute séquence placée à droite de l'expression. Ce déballage requiert autant de variables dans la partie gauche qu'il y a d'éléments dans la séquence. Notez également que cette affectation multiple est juste une combinaison entre un emballage de tuple et un déballage de séquence.

## 5.4 Ensembles

Python fournit également un type de donnée pour les *ensembles*. Un ensemble est une collection non ordonnée sans élément dupliqué. Des utilisations basiques concernent par exemple des tests d'appartenance ou des suppressions de doublons. Les ensembles savent également effectuer les opérations mathématiques telles que les unions, intersections, différences et différences symétriques.

Des accolades ou la fonction `set()` peuvent être utilisés pour créer des ensembles. Notez que pour créer un ensemble vide, `{}` ne fonctionne pas, cela crée un dictionnaire vide. Utilisez plutôt `set()`.

Voici une brève démonstration :

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

(suite sur la page suivante)

(suite de la page précédente)

```

>>> # 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'])

```

Tout comme pour les *compréhensions de listes*, il est possible d'écrire des compréhensions d'ensembles :

```

>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])

```

## 5.5 Dictionnaires

Un autre type de donnée très utile, natif dans Python, est le *dictionnaire* (voir *typesmapping*). Ces dictionnaires sont parfois présents dans d'autres langages sous le nom de « mémoires associatives » ou de « tableaux associatifs ». À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des *clés*, qui peuvent être de n'importe quel type immuable ; les chaînes de caractères et les nombres peuvent toujours être des clés. Des tuples peuvent être utilisés comme clés s'ils ne contiennent que des chaînes, des nombres ou des tuples ; si un tuple contient un objet muable, de façon directe ou indirecte, il ne peut pas être utilisé comme une clé. Vous ne pouvez pas utiliser des listes comme clés, car les listes peuvent être modifiées en place en utilisant des affectations par position, par tranches ou via des méthodes comme `append()` ou `extend()`.

Le plus simple est de considérer les dictionnaires comme des ensembles non ordonnés de paires *clé : valeur*, les clés devant être uniques (au sein d'un dictionnaire). Une paire d'accolades crée un dictionnaire vide : `{}`. Placer une liste de paires clé :valeur séparées par des virgules à l'intérieur des accolades ajoute les valeurs correspondantes au dictionnaire ; c'est également de cette façon que les dictionnaires sont affichés.

Les opérations classiques sur un dictionnaire consistent à stocker une valeur pour une clé et à extraire la valeur correspondant à une clé. Il est également possible de supprimer une paire clé :valeur avec `del`. Si vous stockez une valeur pour une clé qui est déjà utilisée, l'ancienne valeur associée à cette clé est perdue. Si vous tentez d'extraire une valeur associée à une clé qui n'existe pas, une exception est levée.

La méthode `keys()` d'un dictionnaire retourne une liste de toutes les clés utilisées dans le dictionnaire, dans un ordre arbitraire (si vous voulez qu'elle soit triée, appliquez-lui la fonction `sorted()`). Pour tester si une clé est dans le dictionnaire, utilisez le mot-clé `in`.

Voici un petit exemple utilisant un dictionnaire :

```

>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel

```

(suite sur la page suivante)

(suite de la page précédente)

```
{'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']
>>> 'guido' in tel
True
```

Le constructeur `dict()` fabrique un dictionnaire directement à partir d'une liste de paires clé-valeur stockées sous la forme de tuples :

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

De plus, il est possible de créer des dictionnaires par compréhension depuis un jeu de clef et valeurs :

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Lorsque les clés sont de simples chaînes de caractères, il est parfois plus facile de spécifier les paires en utilisant des paramètres nommés :

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

## 5.6 Techniques de boucles

Lorsque vous itérez sur une séquence, la position et la valeur correspondante peuvent être récupérées 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 faire des boucles sur deux séquences ou plus en même temps, les éléments peuvent être associés par 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 {0}? It is {1}'.format(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 faire une boucle en sens inverse sur une séquence, commencez par spécifier la séquence dans son ordre normal, puis appliquez la fonction `reversed()`

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

Pour faire une boucle selon un certain classement sur une séquence, utilisez la fonction `sorted()`, elle renvoie une nouvelle liste classée sans altérer la source :

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

Lorsque vous faites une boucle sur un dictionnaire, les clés et leurs valeurs peuvent être récupérées 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
```

Il est parfois tentant de modifier une liste pendant son itération. Cependant, c'est souvent plus simple et plus sûr de créer une nouvelle liste à la place.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

## 5.7 Plus d'informations sur les conditions

Les conditions utilisées dans une instruction `while` ou `if` peuvent contenir n'importe quel opérateur, pas seulement des comparaisons.

Les opérateurs de comparaison `in` et `not in` testent si une valeur est présente ou non dans une séquence. Les opérateurs `is` et `is not` testent si deux objets sont vraiment le même objet ; ceci n'est important que

pour des objets muables comme des listes. Tous les opérateurs de comparaison ont la même priorité, qui est plus faible que celle des opérateurs numériques.

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

Les comparaisons peuvent être combinées en utilisant les opérateurs booléens `and` et `or`, le résultat d'une comparaison (ou de toute expression booléenne) pouvant être inversé avec `not`. Ces opérateurs ont une priorité inférieure à celle des opérateurs de comparaison ; entre eux, `not` a la priorité la plus élevée et `or` la plus faible, de telle sorte que `A and not B or C` est équivalent à `(A and (not B)) or C`. Comme toujours, des parenthèses peuvent être utilisées pour exprimer l'instruction désirée.

Les opérateurs booléens `and` et `or` sont appelés opérateurs *en circuit court* : leurs arguments sont évalués de la gauche vers la droite et l'évaluation s'arrête dès que le résultat est déterminé. Par exemple, si `A` et `C` sont vrais et `B` est faux, `A and B and C` n'évalue pas l'expression `C`. Lorsqu'elle est utilisée en tant que valeur et non en tant que booléen, la valeur de retour d'un opérateur en circuit court est celle du dernier argument évalué.

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

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Notez qu'en Python, à la différence du C, des affectations ne peuvent pas intervenir à l'intérieur d'expressions. Les programmeurs C râleront peut-être après cela, mais cela évite des erreurs fréquentes que l'on rencontre en C, lorsque l'on tape `=` alors que l'on voulait faire un test avec `==`.

## 5.8 Comparer des séquences avec d'autres types

Des séquences peuvent être comparées avec d'autres séquences du même type. La comparaison utilise un ordre *lexicographique* : les deux premiers éléments de chaque séquence sont comparés, et s'ils diffèrent cela détermine le résultat de la comparaison ; s'ils sont égaux, les deux éléments suivants sont comparés à leur tour, et ainsi de suite jusqu'à ce que l'une des séquences soit épuisée. Si deux éléments à comparer sont eux-mêmes des séquences du même type, alors la comparaison lexicographique est effectuée récursivement. Si tous les éléments des deux séquences sont égaux, les deux séquences sont alors considérées comme égales. Si une séquence est une sous-séquence de l'autre, la séquence la plus courte est celle dont la valeur est inférieure. La comparaison lexicographique des chaînes de caractères utilise l'ordre ASCII des caractères. Voici quelques exemples de comparaisons entre séquences de 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)
```

La comparaison d'objets de types différents est autorisée. Le résultat est déterminé mais arbitraire : les types sont ordonnés par leur nom. Ainsi, une liste (type `list`) est toujours inférieure à une chaîne de caractères (type `str`), une chaîne de caractères toujours inférieure à un tuple...<sup>1</sup> Des types numériques mélangés sont comparés en fonction de leur valeur numérique, donc 0 est égal à 0.0, etc.

---

1. Les règles de comparaison d'objets de types différents ne doivent pas être considérées comme fiables ; elles peuvent être amenées à changer dans une future version du langage.

## Notes



Lorsque vous quittez et entrez à nouveau dans l'interpréteur Python, tout ce que vous avez déclaré dans la session précédente est perdu. Afin de rédiger des programmes plus longs, vous devez utiliser un éditeur de texte, préparer votre code dans un fichier et exécuter Python avec ce fichier en paramètre. Cela s'appelle créer un *script*. Lorsque votre programme grandit, vous pouvez séparer votre code dans plusieurs fichiers. Ainsi, il vous est facile de réutiliser des fonctions écrites pour un programme dans un autre sans avoir à les copier.

Pour gérer cela, Python vous permet de placer des définitions dans un fichier et de les utiliser dans un script ou une session interactive. Un tel fichier est appelé un *module* et les définitions d'un module peuvent être importées dans un autre module ou dans le module *main* (qui est le module qui contient vos variables et définitions lors de l'exécution d'un script au niveau le plus haut ou en mode interactif).

Un module est un fichier contenant des définitions et des instructions. Son nom de fichier est le nom du module suffixé de `.py`. À l'intérieur d'un module, son propre nom est accessible par la variable `__name__`. Par exemple, prenez votre éditeur favori et créez un fichier `fibonacci.py` dans le répertoire courant qui contient :

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Maintenant, ouvrez un interpréteur et importez le module en tapant :

```
>>> import fibo
```

Cela n'importe pas les noms des fonctions définies dans `fibo` directement dans la table des symboles courants mais y ajoute simplement `fibo`. Vous pouvez donc appeler les fonctions *via* le nom du module :

```
>>> 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, il est possible de lui assigner un nom local :

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

## 6.1 Les modules en détail

Un module peut contenir aussi bien des instructions que des déclarations de fonctions. Ces instructions permettent d'initialiser le module. Elles ne sont exécutées que la *première* fois que le nom d'un module est trouvé dans un `import`<sup>1</sup> (elles sont aussi exécutées lorsque le fichier est exécuté en tant que script).

Chaque module possède sa propre table de symboles, utilisée comme table de symboles globaux par toutes les fonctions définies par le module. Ainsi l'auteur d'un module peut utiliser des variables globales dans un module sans se soucier de collisions de noms avec des variables globales définies par l'utilisateur du module. Cependant, si vous savez ce que vous faites, vous pouvez modifier une variable globale d'un module avec la même notation que pour accéder aux fonctions : `nommodule.nomelement`.

Des modules peuvent importer d'autres modules. Il est courant, mais pas obligatoire, de ranger tous les `import` au début du module (ou du script). Les noms des modules importés sont insérés dans la table des symboles globaux du module qui importe.

Il existe une variante de l'instruction `import` qui importe les noms d'un module directement dans la table de symboles du module qui l'importe, 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'insère pas le nom du module depuis lequel les définitions sont récupérées dans la table des symboles locaux (dans cet exemple, `fibo` n'est pas défini).

Il existe même une variante permettant d'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
```

Importe tous les noms sauf ceux commençant par un tiret bas (`_`).

---

1. En réalité, la déclaration d'une fonction est elle-même une instruction ; son exécution enregistre le nom de la fonction dans la table des symboles globaux du module.

Notez qu'en général, importer `*` d'un module ou d'un paquet est déconseillé. Souvent, le code devient difficilement lisible. Son utilisation en mode interactif est acceptée pour gagner quelques secondes.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `fib`.

It can also be used when utilising `from` with similar effects :

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

**Note :** Pour des raisons d'efficience, chaque module n'est importé qu'une fois par session de l'interprète. Si vous changez vos modules, vous devrez donc redémarrer votre interprète, ou, si vous souhaitez simplement tester un seul module de manière interactive, utiliser la fonction `reload()`, tel que : `reload(modulename)`.

### 6.1.1 Exécuter des modules comme des scripts

Lorsque vous exécutez un module Python avec :

```
python fibo.py <arguments>
```

le code du module est exécuté comme si vous l'aviez importé mais son `__name__` vaut `"__main__"`. Donc, en ajoutant ces lignes à la fin du module :

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

vous pouvez rendre le fichier utilisable comme script aussi bien que comme module importable, car le code qui analyse la ligne de commande n'est lancé que si le module est exécuté comme fichier « main » :

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Si le fichier est importé, le code n'est pas exécuté :

```
>>> import fibo
>>>
```

C'est typiquement utilisé soit pour proposer une interface utilisateur pour un module, soit pour lancer les tests sur le module (exécuter le module en tant que script lance les tests).

### 6.1.2 Les dossiers de recherche de modules

Lorsqu'un module nommé par exemple `spam` est importé, il est d'abord recherché parmi les modules natifs puis, s'il n'est pas trouvé, l'interpréteur cherche un fichier nommé `spam.py` dans une liste de dossiers donnée par la variable `sys.path`. Par défaut, `sys.path` est initialisée à :

- le dossier contenant le script d'entrée (ou le dossier courant).
- `PYTHONPATH` (une liste de dossiers, utilisant la même syntaxe que la variable shell `PATH`) ;
- les valeurs par défaut dépendantes de l'installation.

Après leur initialisation, les programmes Python peuvent modifier leur `sys.path`. Le dossier contenant le script courant est placé au début de la liste des dossiers à rechercher, avant les dossiers de bibliothèques. Cela signifie qu'un module dans ce dossier, ayant le même nom qu'un module, sera chargé à sa place. C'est une erreur typique, à moins que ce ne soit voulu. Voir *Modules standards* pour plus d'informations.

### 6.1.3 Fichiers Python « compilés »

Un gain de temps important au démarrage pour les courts programmes utilisant beaucoup de modules standards, si un fichier `spam.pyc` existe dans le dossier où `spam.py` est trouvé, il est supposé contenir une version compilée du module : `mod:spam`. La date de modification du fichier `spam.py` est utilisée pour créer le fichier `spam.pyc`, et stockée dans le fichier `spam.pyc`, ainsi le fichier `.pyc` est ignoré si ces deux dates ne coïncident pas.

Normalement, vous n'avez rien à faire pour créer un fichier `spam.pyc`. Lorsque le fichier `spam.py` est compilé, Python essaye d'écrire la version compilée dans `spam.pyc`. Ce n'est pas une erreur si cette tentative échoue, et si pour n'importe quelle raison, le fichier `.pyc` viendrait à ne pas être écrit entièrement, il serait reconnu comme invalide et ignoré. Le contenu du fichier `spam.pyc` est indépendant de la plateforme, donc un dossier contenant des modules Python peut être partagé entre différentes machines de différentes architectures.

Astuces pour les experts :

- Lorsque l'interpréteur Python est invoqué avec l'option `-O`, un code optimisé est généré et stocké dans des fichiers en `.pyo`. L'optimiseur, aujourd'hui, n'aide pas beaucoup, il ne fait que supprimer les instructions `assert`. Lorsque l'option `-O` est utilisée, tous les *bytecodes* sont optimisés, les fichiers `.pyc` sont ignorés, et les fichiers `.py` compilés en un bytecode optimisé.
- Donner deux `-O` à l'interpréteur (`-OO`) engendrera un bytecode qui pourrait dans de rares cas provoquer un dysfonctionnement du programme. Actuellement seulement les chaînes `__doc__` sont retirés du bytecode, rendant les `.pyo` plus petits. Puisque certains programmes peuvent en avoir besoin, vous ne devriez utiliser cette option que si vous savez ce que vous faites.
- Un programme ne s'exécute pas plus vite lorsqu'il est lu depuis un `.pyc` ou `.pyo`, la seule chose qui est plus rapide est la vitesse à laquelle ils sont chargés.
- Lorsqu'un programme est lancé en donnant son nom à une invite de commande, le bytecode du script n'est jamais écrit dans un `.pyc` ni `.pyo`. Cependant son temps de chargement peut être réduit en déplaçant la plupart de son code vers un module, et n'utiliser qu'un script léger pour importer ce module. Il est aussi possible d'exécuter un fichier `.pyc` ou `.pyo` directement depuis l'invite de commande.
- Il est possible d'avoir un fichier `spam.pyc` (ou `spam.pyo` lorsque `-O` a été utilisée) sans `spam.py` pour un même module. Cela permet de distribuer une bibliothèque Python dans un format qui est modérément compliqué à rétro-ingénier.
- Le module `compileall` peut créer des fichiers `.pyc` (ou `.pyo` si l'option `-O` est fournie) pour chaque module d'un dossier.

## 6.2 Modules standards

Python est accompagné d'une bibliothèque de modules standards, décrits dans la documentation de la Bibliothèque Python, plus loin. Certains modules sont intégrés dans l'interpréteur, ils exposent des outils qui ne font pas partie du langage mais qui font tout de même partie de l'interpréteur, soit pour le côté pratique, soit pour exposer des outils essentiels tels que l'accès aux appels système. La composition de ces modules est configurable à la compilation et dépend aussi de la plateforme cible. Par exemple, le module `winreg` n'est proposé que sur les systèmes Windows. Un module mérite une attention particulière, le module `sys`, qui est présent dans tous les interpréteurs Python. Les variables `sys.ps1` et `sys.ps2` définissent les chaînes d'invites principales et secondaires :

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

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

La variable `sys.path` est une liste de chaînes qui détermine les chemins de recherche de modules pour l'interpréteur. Elle est initialisée à un chemin par défaut pris de la variable d'environnement `PYTHONPATH` ou d'une valeur par défaut interne si `PYTHONPATH` n'est pas définie. `sys.path` est modifiable en utilisant les opérations habituelles des listes :

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

## 6.3 La fonction `dir()`

La fonction interne `dir()` est utilisée pour trouver quels noms sont définis par un module. Elle donne une liste de chaînes classées par ordre lexicographique :

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions']
```

Sans paramètre, `dir()` liste les noms actuellement définis :

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

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

La fonction `dir()` ne liste pas les noms des fonctions et variables natives. Si vous en voulez la liste, ils sont définis dans le module standard `__builtin__` :

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
'__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

## 6.4 Les paquets

Packages are a way of structuring Python's module namespace by using « dotted module names ». For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Imaginez que vous voulez construire un ensemble de modules (un « paquet ») pour gérer uniformément les fichiers contenant du son et des données sonores. Il existe un grand nombre de formats de fichiers pour stocker du son (généralement identifiés par leur extension, par exemple `.wav`, `.aiff`, `.au`), vous avez donc besoin de créer et maintenir un nombre croissant de modules pour gérer la conversion entre tous ces formats. Vous voulez aussi pouvoir appliquer un certain nombre d'opérations sur ces sons : mixer, ajouter de l'écho, égaliser, ajouter un effet stéréo artificiel, etc. Donc, en plus des modules de conversion, vous allez écrire une myriade de modules permettant d'effectuer ces opérations. Voici une structure possible pour votre paquet (exprimée sous la forme d'une arborescence de fichiers :

sound/	Top-level package
__init__.py	Initialize the sound package

(suite sur la page suivante)

(suite de la page précédente)

```

formats/                Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
effects/                Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
filters/               Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

Lorsqu'il importe des paquets, Python cherche dans chaque dossier de `sys.path` un sous-dossier du nom du paquet.

Les fichiers `__init__.py` sont nécessaires pour que Python considère les dossiers comme contenant des paquets, cela évite que des dossiers ayant des noms courants comme `string` ne masquent des modules qui auraient été trouvés plus tard dans la recherche des dossiers. Dans le plus simple des cas, `__init__.py` peut être vide, mais il peut aussi exécuter du code d'initialisation pour son paquet ou configurer la variable `__all__` (documentée plus loin).

Les utilisateurs d'un module peuvent importer ses modules individuellement, par exemple :

```
import sound.effects.echo
```

charge le sous-module `sound.effects.echo`. Il doit alors être référencé par son nom complet.

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

Une autre manière d'importer des sous-modules est :

```
from sound.effects import echo
```

charge aussi le sous-module `echo` et le rend disponible sans avoir à indiquer le préfixe du paquet. Il peut donc être utilisé comme ceci :

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

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

```
from sound.effects.echo import echofilter
```

Le sous-module `echo` est toujours chargé mais ici la fonction `echofilter()` est disponible directement :

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

Notez que lorsque vous utilisez `from package import element`, `element` peut aussi bien être un sous-module, un sous-paquet ou simplement un nom déclaré dans le paquet (une variable, une fonction ou une classe). L'instruction `import` cherche en premier si `element` est défini dans le paquet ; s'il ne l'est pas, elle cherche à charger un module et, si elle n'en trouve pas, une exception `ImportError` est levée.

Au contraire, en utilisant la syntaxe `import element.souselement.soussouselement`, chaque `element` sauf le dernier doit être un paquet. Le dernier `element` peut être un module ou un paquet, mais ne peut être ni une fonction, ni une classe, ni une variable définie dans l'élément précédent.

### 6.4.1 Importer \* depuis un paquet

Qu'arrive-il lorsqu'un utilisateur écrit `from sound.effects import *`? Idéalement, on pourrait espérer que Python aille chercher tous les sous-modules du paquet sur le système de fichiers et qu'ils seraient tous importés. Cela pourrait être long et importer certains sous-modules pourrait avoir des effets secondaires indésirables ou, du moins, désirés seulement lorsque le sous-module est importé explicitement.

La seule solution, pour l'auteur du paquet, est de fournir un index explicite du contenu du paquet. L'instruction `import` utilise la convention suivante : si le fichier `__init__.py` du paquet définit une liste nommée `__all__`, cette liste est utilisée comme liste des noms de modules devant être importés lorsque `from package import *` est utilisé. Il est de la responsabilité de l'auteur du paquet de maintenir cette liste à jour lorsque de nouvelles versions du paquet sont publiées. Un auteur de paquet peut aussi décider de ne pas autoriser d'importer `*` pour son paquet. Par exemple, le fichier `sound/effects/__init__.py` peut contenir le code suivant :

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

Cela signifie que `from sound.effects import *` importe les trois sous-modules explicitement désignés du paquet `sound`.

Si `__all__` n'est pas définie, l'instruction `from sound.effects import *` n'importe *pas* tous les sous-modules du paquet `sound.effects` dans l'espace de noms courant mais s'assure seulement que le paquet `sound.effects` a été importé (i.e. que tout le code du fichier `__init__.py` a été exécuté) et importe ensuite les noms définis dans le paquet. Cela inclut tous les noms définis (et sous-modules chargés explicitement) par `__init__.py`. Sont aussi inclus tous les sous-modules du paquet ayant été chargés explicitement par une instruction `import`. Typiquement :

```
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 de noms courant lorsque `from...import` est exécuté parce qu'ils sont définis dans le paquet `sound.effects` (cela fonctionne aussi lorsque `__all__` est définie).

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Rappelez-vous que rien ne vous empêche d'utiliser `from paquet import sous_module_specifique` ! C'est d'ailleurs la manière recommandée, à moins que le module qui fait les imports ait besoin de sous-modules ayant le même nom mais provenant de paquets différents.

### 6.4.2 Références internes dans un paquet

Les sous modules ont souvent besoin de s'utiliser entre eux. Par exemple le module `surround` pourrait utiliser le module `echo`. Dans les faits ces références sont si communes que l'instruction clef `import` regarde d'abord

dans le paquet avant de chercher dans les modules standard. Donc le module `surround` peut simplement utiliser `import echo` ou `from echo import echofilter`. Si le module n'est pas trouvé dans le paquet courant (le paquet pour lequel le module actuel est un sous module), l'instruction `import` cherchera le nom donné en dehors du paquet.

Lorsque les paquets sont organisés en sous-paquets (comme le paquet `sound` par exemple), vous pouvez utiliser des imports absolus pour cibler des paquets voisins. Par exemple, si le module `sound.filters.vocoder` a besoin du module `echo` du paquet `sound.effects`, il peut utiliser `from sound.effects import echo`.

Depuis Python 2.5, en plus des imports relatifs décrits plus haut, vous pouvez écrire des imports relatifs explicites via la syntaxe `from module import name`. Ces imports relatifs explicites utilisent le préfixe point (`.`) pour indiquer le paquet courant ou le paquet parent. Pour le module `surround` par exemple vous pourriez utiliser :

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Notez que les imports relatifs, implicites et explicites, sont basés sur le nom du module courant. Puisque le nom du module principal est toujours `"__main__"`, les modules conçus comme modules principaux d'une application doivent toujours utiliser des imports absolus.

### 6.4.3 Paquets dans plusieurs dossiers

Les paquets possèdent un attribut supplémentaire, `__path__`, qui est une liste initialisée avant l'exécution du fichier `__init__.py`, contenant le nom de son dossier dans le système de fichiers. Cette liste peut être modifiée, altérant ainsi les futures recherches de modules et sous-paquets contenus dans le paquet.

Bien que cette fonctionnalité ne soit que rarement utile, elle peut servir à élargir la liste des modules trouvés dans un paquet.

#### Notes



Il existe bien des moyens de présenter les sorties d'un programmes ; les données peuvent être affichées sous une forme lisible par un être humain ou sauvegardées dans un fichier pour une utilisation future. Ce chapitre présente quelques possibilités.

### 7.1 Formatage de données

Jusque là, nous avons rencontré deux moyens d'écrire des données : les *déclarations d'expressions* et l'instruction `print`. (Une troisième méthode consiste à utiliser la méthode `write()` des fichiers ; le fichier de sortie standard peut être référence en tant que `sys.stdout`. Voyez le Guide de Référence de la Bibliothèque Standard pour en savoir plus.)

Mais vous voudrez souvent avoir plus de contrôle sur le formatage de vos sorties que de simplement imprimer des valeurs séparées par des espaces. Il y a deux méthodes pour formater ces sorties ; la première est de gérer les chaînes de caractères par vous même ; en utilisant des tranches sur vos chaînes et des opérations de concaténation, vous pouvez créer n'importe quelle présentation imaginable. Mais les chaînes de caractères disposent également de méthodes qui facilitent, par exemple, les alignements de chaînes de caractères selon une certaine largeur de colonne, et qui seront présentées rapidement. La seconde méthode consiste à utiliser la méthode `format()`.

Le module `string` contient une classe `Template` qui permet aussi de remplacer des valeurs au sein de chaînes de caractères.

Mais une question demeure, bien sûr : comment convertir des valeurs en chaînes de caractères ? Heureusement, Python fournit plusieurs moyens de convertir n'importe quelle valeur en chaîne : les fonctions `repr()` et `str()`.

La fonction `str()` est destinée à renvoyer des représentations de valeurs qui soient lisibles par un être humain, alors que la fonction `repr()` est destinée à générer des représentations qui puissent être lues par l'interpréteur (ou forceront une `SyntaxError` s'il n'existe aucune syntaxe équivalente). Pour les objets qui n'ont pas de représentation humaine spécifique, `str()` renverra la même valeur que `repr()`. Beaucoup de valeurs, comme les nombres ou les structures telles que les listes ou les dictionnaires, ont la même représentation en utilisant les deux fonctions. Les chaînes de caractères et les nombres à virgule flottante, en revanche, ont deux représentations distinctes.

Quelques exemples :

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs'))))
"(32.5, 40000, ('spam', 'eggs'))"
```

Voici deux façons d'écrire une table de carrés et de cubes :

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(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 '{0:2d} {1:3d} {2:4d}'.format(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
```

(suite sur la page suivante)

(suite de la page précédente)

```
9 81 729
10 100 1000
```

(Note that in the first example, one space between each column was added by the way `print` works : by default it adds spaces between its arguments.)

Cet exemple illustre l'utilisation de la méthode `str.rjust()` des chaînes de caractères ; elle justifie à droite une chaîne dans un champ d'une largeur donnée en ajoutant des espaces sur la gauche. Il existe des méthodes similaires `str.ljust()` et `str.center()`. Ces méthodes n'écrivent rien, elles renvoient simplement une nouvelle chaîne. Si la chaîne passée en paramètre est trop longue, elle n'est pas tronquée mais renvoyée sans modification ; cela peut chambouler votre mise en page mais c'est souvent préférable à l'alternative, qui pourrait mentir sur une valeur (et si vous voulez vraiment tronquer vos valeurs, vous pouvez toujours utiliser une tranche, comme dans `x.ljust(n)[:n]`).

Il existe une autre méthode, `str.zfill()`, qui comble une chaîne numérique à gauche avec des zéros. Elle comprend les signes plus et moins :

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

L'utilisation de base de la méthode `str.format()` ressemble à ceci :

```
>>> print 'We are the {} who say "{}!".format('knights', 'Ni')
We are the knights who say "Ni!"
```

Les crochets et les caractères qu'ils contiennent (appelés les champs de formatage) sont remplacés par les objets passés en paramètres à la méthode `str.format()`. Un nombre entre crochets se réfère à la position de l'objet passé à la méthode `str.format()`

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

Si des arguments nommés sont utilisés dans la méthode `str.format()`, leurs valeurs sont utilisées en se basant sur le nom des arguments :

```
>>> print 'This {food} is {adjective}.'.format(
...     food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Les arguments positionnés et nommés peuvent être combinés arbitrairement :

```
>>> print 'The story of {0}, {1}, and {other}'.format('Bill', 'Manfred',
...                                                other='Georg')
The story of Bill, Manfred, and Georg.
```

'!s' (appliquer `str()`) et '!r' (appliquer `repr()`) peuvent être utilisées pour convertir les valeurs avant leur formatage :

```
>>> import math
>>> print 'The value of PI is approximately {}'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

Un caractère ':' suivi d'une spécification de formatage peuvent suivre le nom du champ. Ceci offre un niveau de contrôle plus fin sur la façon dont les valeurs sont formatées. L'exemple suivant arrondit Pi à trois chiffres après la virgule (NdT : qui, en notation anglo-saxonne, est un point).

```
>>> import math
>>> print 'The value of PI is approximately {:.3f}'.format(math.pi)
The value of PI is approximately 3.142.
```

Indiquer un entier après le ':' indique la largeur minimale de ce champ en nombre de caractères. C'est utile pour faire de jolis tableaux :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '{0:10} ==> {1:10d}'.format(name, phone)
...
Jack          ==>      4098
Dcab          ==>      7678
Sjoerd        ==>      4127
```

Si vous avez une chaîne de formatage vraiment longue que vous ne voulez pas découper, il est possible de référencer les variables à formater par leur nom plutôt que par leur position. Utilisez simplement un dictionnaire et la notation entre crochets '[]' pour accéder aux clés :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Vous pouvez obtenir le même résultat en passant le tableau comme des arguments nommés en utilisant la notation \*\*

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

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

Pour avoir une description complète du formatage des chaînes de caractères avec la méthode `str.format()`, lisez : [formatstrings](#).

### 7.1.1 Anciennes méthodes de formatage de chaînes

L'opérateur % peut aussi être utilisé pour formater des chaînes. Il interprète l'argument de gauche pratiquement comme une chaîne de formatage de la fonction `sprintf()` à appliquer à l'argument de droite, et il renvoie la chaîne résultant de cette opération de formatage. Par exemple :

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

Vous trouverez plus d'informations dans la section string-formatting.

## 7.2 Lecture et écriture de fichiers

`open()` retourne un objet fichier, et est le plus souvent utilisé avec deux arguments : `open(filename, mode)`.

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

Le premier argument est une chaîne contenant le nom du fichier. Le deuxième argument est une autre chaîne contenant quelques caractères décrivant la façon dont le fichier est utilisé. *mode* peut être `'r'` quand le fichier n'est accédé qu'en lecture, `'w'` en écriture seulement (un fichier existant portant le même nom sera alors écrasé) et `'a'` ouvre le fichier en mode ajout (toute donnée écrite dans le fichier est automatiquement ajoutée à la fin). `'r+'` ouvre le fichier en mode lecture/écriture. L'argument *mode* est optionnel, sa valeur par défaut est `'r'`.

Sous Windows, `'b'` ajouté au mode ouvre le fichier en mode binaire, et il existe donc des modes comme `'rb'`, `'wb'` et `'r+b'`. Python sous Windows fait effectivement une distinction entre fichiers texte et binaires ; le caractère de fin de ligne des fichiers texte est automatiquement modifié lorsqu'une donnée est lue ou écrite. Cette modification sous le manteau des données du fichier n'est pas un problème pour les fichiers textes en ASCII, mais elle peut corrompre les données binaires comme celles des fichiers JPEG ou EXE. Soyez donc attentifs à toujours bien utiliser le mode binaire lorsque vous travaillez avec de tels fichiers. Sous Unix, le mode binaire n'a aucun effet mais vous pouvez l'utiliser pour tous les fichiers binaires pour que votre code soit indépendant de la plate-forme.

### 7.2.1 Méthodes des objets fichiers

Les derniers exemples de cette section supposent qu'un objet fichier appelé `f` a déjà été créé.

Pour lire le contenu d'un fichier, appelez `f.read(size)`, qui lit une certaine quantité de données et les retourne sous la forme d'une chaîne de caractères. *size* est un argument numérique optionnel. Quand *size* est omis ou négatif, le contenu entier du fichier est lu et retourné ; c'est votre problème si le fichier est deux fois plus gros que la mémoire de votre machine. Sinon, au plus *size* octets sont lus et retournés. Lorsque la fin du fichier est atteinte, `f.read()` renvoie une chaîne vide (`""`)

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` lit une seule ligne du fichier ; un caractère de fin de ligne (`\n`) est laissé à la fin de la chaîne, et n'est omis que sur la dernière ligne du fichier si celui-ci ne se termine pas un caractère de fin de ligne. Ceci permet de rendre la valeur de retour non ambiguë : si `f.readline()` retourne une chaîne vide, c'est que la fin du fichier a été atteinte, alors qu'une ligne vide est représentée par `'\n'`, une chaîne de caractères ne contenant qu'une fin de ligne :

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
```

(suite sur la page suivante)

(suite de la page précédente)

```
'Second line of the file\n'
>>> f.readline()
''
```

Pour lire ligne à ligne, vous pouvez aussi boucler sur l'objet fichier. C'est plus efficace en terme de gestion mémoire, plus rapide et donne un code plus simple :

```
>>> for line in f:
    print line,

This is the first line of the file.
Second line of the file
```

Pour construire une liste avec toutes les lignes d'un fichier, il est aussi possible d'utiliser `list(f)` ou `f.readlines()`.

`f.write(string)` écrit le contenu de *string* dans le fichier, et renvoie `None`

```
>>> f.write('This is a test\n')
```

Pour écrire autre chose qu'une chaîne, il faut commencer par la convertir en chaîne

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

`f.tell()` retourne un entier indiquant la position actuelle dans le fichier, mesurée en octets à partir du début du fichier. Pour modifier la position dans le fichier, utilisez `f.seek(offset, from_what)`. La position est calculée en ajoutant *offset* à un point de référence ; ce point de référence est sélectionné par l'argument *from\_what* : 0 pour le début du fichier, 1 pour la position actuelle, et 2 pour la fin du fichier. *from\_what* peut être omis et sa valeur par défaut est 0, utilisant le début du fichier comme point de référence :

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

Quand vous avez terminé d'utiliser un fichier, appeler `f.close()` pour le fermer et libérer toutes les ressources système qu'il a pu utiliser. Après l'appel de `f.close()`, toute tentative d'utilisation de l'objet fichier échouera systématiquement :

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

C'est une bonne pratique d'utiliser le mot-clé `with` lorsque vous traitez des fichiers. Ceci procure l'avantage de toujours fermer correctement le fichier, même si une exception a été déclenchée. C'est aussi beaucoup plus court que d'utiliser l'équivalent avec des blocs `try- finally`

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Les fichiers disposent de méthodes supplémentaires, telles que `isatty()` et `truncate()` qui sont moins souvent utilisées ; consultez la Référence de la Bibliothèque Standard pour avoir un guide complet des objets fichiers.

### 7.2.2 Sauvegarde de données structurées avec le module `json`

Les chaînes de caractères peuvent facilement être écrites dans un fichier et relues. Les nombres nécessitent un peu plus d'effort, car la méthode `read()` ne renvoie que des chaînes. Elles doivent donc être passées à une fonction comme `int()`, qui prend une chaîne comme `'123'` en entrée et renvoie sa valeur numérique 123. Mais dès que vous voulez enregistrer des types de données plus complexes comme des listes, des dictionnaires ou des instances de classes, le traitement lecture/écriture à la main devient vite compliqué.

Plutôt que de passer son temps à écrire et déboguer du code permettant de sauvegarder des types de données compliqués, Python permet d'utiliser **JSON** (**J**ava**S**cript **O**bject **N**otation), un format répandu de représentation et d'échange de données. Le module standard appelé `json` peut transformer des données hiérarchisées Python en une représentation sous forme de chaîne de caractères. Ce processus est nommé *sérialiser*. Reconstruire les données à partir de leur représentation sous forme de chaîne est appelé *désérialiser*. Entre sa sérialisation et sa dé-sérialisation, la chaîne représentant les données peut avoir été stockée ou transmise à une autre machine.

---

**Note :** Le format JSON est couramment utilisé dans les applications modernes pour échanger des données. Beaucoup de développeurs le maîtrise, ce qui en fait un format de prédilection pour l'interopérabilité.

---

Si vous avez un objet `x`, vous pouvez voir sa représentation JSON en tapant simplement :

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Une autre variante de la fonction `dumps()`, appelée `dump()` sérialise simplement l'objet dans un fichier. Donc si `f` est un *file object* ouvert en écriture, on peut faire :

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

Pour décoder l'objet à nouveau, si `f` est un *file object* ouvert en lecture :

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

Cette méthode de sérialisation peut sérialiser des listes et des dictionnaires. Mais sérialiser d'autres types de données requiert un peu plus de travail. La documentation du module `json` explique comment faire.

**Voir aussi :**

Le module `pickle`

Au contraire de *JSON*, *pickle* est un protocole permettant la sérialisation d'objets Python arbitrairement complexes. Il est donc spécifique à Python et ne peut pas être utilisé pour communiquer avec d'autres langages. Il est aussi, par défaut, une source de vulnérabilité : dé-sérialiser des données au format *pickle* provenant d'une source malveillante et particulièrement habile peut mener à exécuter du code arbitraire.



---

Erreurs et exceptions

---

Jusqu'ici, les messages d'erreurs ont seulement été mentionnés. Mais si vous avez essayé les exemples vous avez certainement vu plus que cela. En fait, il y a au moins deux types d'erreurs à distinguer : les *erreurs de syntaxe* et les *exceptions*.

## 8.1 Les erreurs de syntaxe

Les erreurs de syntaxe, qui sont des erreurs d'analyse du code, sont peut-être celles que vous rencontrez le plus souvent lorsque vous êtes encore en phase d'apprentissage de Python :

```
>>> while True print 'Hello world'
      File "<stdin>", line 1
        while True print 'Hello world'
                        ^
SyntaxError: invalid syntax
```

L'analyseur repère la ligne incriminée et affiche une petite “flèche” pointant vers le premier endroit de la ligne où l'erreur a été détectée. L'erreur est causée (ou, au moins, a été détectée comme telle) par le symbole placé *avant* la flèche, ici car il manque deux points (':') avant lui, dans l'exemple l'erreur est détectée au mot clef **print**, car il manque deux points (':') juste avant. Le nom de fichier et le numéro de ligne sont affichés pour vous permettre de localiser facilement l'erreur lorsque le code provient d'un script.

## 8.2 Exceptions

Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution. Les erreurs détectées durant l'exécution sont appelées des *exceptions* et ne sont pas toujours fatales : nous apprendrons bientôt comment les traiter dans vos programmes. La plupart des exceptions toutefois ne sont pas prises en charge par les programmes, ce qui génère des messages d'erreurs comme celui-ci :

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de différents types et ce type est indiqué dans le message : les types indiqués dans l'exemple sont `ZeroDivisionError`, `NameError` et `TypeError`. Le texte affiché comme type de l'exception est le nom de l'exception native qui a été déclenchée. Ceci est vrai pour toutes les exceptions natives mais n'est pas une obligation pour les exceptions définies par l'utilisateur (même si c'est une convention bien pratique). Les noms des exceptions standards sont des identifiants natifs (pas des mots réservés).

Le reste de la ligne fournit plus de détails en fonction du type de l'exception et de ce qui l'a causée.

La partie précédente dans le message d'erreur indique le contexte dans lequel s'est produite l'exception, sous la forme d'une trace de pile d'exécution. En général, celle-ci contient les lignes du code source ; toutefois, les lignes lues à partir de l'entrée standard ne sont pas affichées.

Vous trouvez la liste des exceptions natives et leur signification dans `bltin-exceptions`.

## 8.3 Gestion des exceptions

Il est possible d'écrire des programmes qui prennent en charge certaines exceptions. Regardez l'exemple suivant, qui demande une saisie à l'utilisateur jusqu'à ce qu'un entier valide ait été entré, mais permet à l'utilisateur d'interrompre le programme (en utilisant `Control-C` ou un autre raccourci que le système accepte) ; notez qu'une interruption générée par l'utilisateur est signalée en levant l'exception `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
... 
```

L'instruction `try` fonctionne comme ceci :

- premièrement, la *clause try* (instruction(s) placée(s) entre les mots-clés `try` et `except`) est exécutée ;
- si aucune exception n'intervient, la clause `except` est sautée et l'exécution de l'instruction `try` est terminée ;
- si une exception intervient pendant l'exécution de la clause `try`, le reste de cette clause est sauté. Si le type d'exception levée correspond à un nom indiqué après le mot-clé `except`, la clause `except` correspondante est exécutée, puis l'exécution continue après l'instruction `try` ;
- si une exception intervient et ne correspond à aucune exception mentionnée dans la clause `except`, elle est transmise à l'instruction `try` de niveau supérieur ; si aucun gestionnaire d'exception n'est trouvé, il s'agit d'une *exception non gérée* et l'exécution s'arrête avec un message comme indiqué ci-dessus.

Une instruction `try` peut comporter plusieurs clauses `except` pour permettre la prise en charge de différentes exceptions. Mais un seul gestionnaire, au plus, sera exécuté. Les gestionnaires ne prennent en charge que les exceptions qui interviennent dans la clause `try` correspondante, pas dans d'autres gestionnaires de la même instruction `try`. Mais une même clause `except` peut citer plusieurs exceptions sous la forme d'un tuple entre parenthèses, comme dans cet exemple :

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

Notez que les parenthèses autour de ce tuple sont nécessaires, car `except ValueError, e:` est la syntaxe utilisée pour ce qui s'écrit désormais `except ValueError as e:` dans les dernières versions de Python (comme décrit ci-dessous). L'ancienne syntaxe est toujours supportée pour la compatibilité ascendante. Ce qui signifie que `except RuntimeError, TypeError:` n'est pas l'équivalent de `except (RuntimeError, TypeError):` mais de `except RuntimeError as TypeError:`, ce qui n'est pas ce que l'on souhaite.

La dernière clause `except` peut omettre le(s) nom(s) d'exception(s) et joue alors le rôle de joker. C'est toutefois à utiliser avec beaucoup de précautions car il est facile de masquer une vraie erreur de programmation par ce biais. Elle peut aussi être utilisée pour afficher un message d'erreur avant de propager l'exception (en permettant à un appelant de gérer également l'exception) :

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

L'instruction `try ... except` accepte également une *clause else* optionnelle qui, lorsqu'elle est présente, doit se placer après toutes les clauses `except`. Elle est utile pour du code qui doit être exécuté lorsqu'aucune exception n'a été levée par la clause `try`. Par exemple :

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Il vaut mieux utiliser la clause `else` plutôt que d'ajouter du code à la clause `try` car cela évite de capturer accidentellement une exception qui n'a pas été levée par le code initialement protégé par l'instruction `try ... except`.

Quand une exception intervient, une valeur peut lui être associée, que l'on appelle *l'argument* de l'exception. La présence de cet argument et son type dépendent du type de l'exception.

La clause `except` peut spécifier un nom de variable après le nom de l'exception (ou le tuple). Cette variable est liée à une instance d'exception avec les arguments stockés dans `instance.args`. Pour plus de commo-

dité, l'instance de l'exception définit la méthode `__str__()` afin que les arguments puissent être imprimés directement sans avoir à référencer `.args`.

On peut aussi instancier une exception et lui ajouter autant d'attributs que nécessaire avant de la déclencher

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

Si une exception a un argument, il est imprimé dans la dernière partie (“detail”) du message des exceptions non gérées.

Les gestionnaires d'exceptions n'interceptent pas que les exceptions qui sont levées immédiatement dans leur clause `try`, mais aussi celles qui sont levées au sein de fonctions appelées (parfois indirectement) dans la clause `try`. Par exemple :

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

## 8.4 Déclencher des exceptions

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

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Le seul argument à `raise` indique l'exception à déclencher. Cela peut être soit une instance d'exception, soit une classe d'exception (une classe dérivée de `Exception`).

Si vous avez besoin de savoir si une exception a été levée mais que vous n'avez pas intention de la gérer, une forme plus simple de l'instruction `raise` permet de propager l'exception :

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

## 8.5 Exceptions définies par l'utilisateur

Les programmes peuvent nommer leur propres exceptions en créant de nouvelles classes (voir *Classes* à propos des classes Python). Ces exceptions doivent typiquement être dérivées de la classe `Exception`, directement ou indirectement. Par exemple :

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: 'oops!'
```

Dans cet exemple, la méthode `__init__()` de la classe `Exception` a été surchargée. Le nouveau comportement crée simplement l'attribut *value*. Ceci remplace le comportement par défaut qui crée l'attribut *args*.

Les classes d'exceptions peuvent être définies pour faire tout ce qu'une autre classe peut faire. Elles sont le plus souvent gardées assez simples, n'offrant que les attributs permettant aux gestionnaires de ces exceptions d'extraire les informations relatives à l'erreur qui s'est produite. Lorsque l'on crée un module qui peut déclencher plusieurs types d'erreurs distincts, une pratique courante est de créer une classe de base pour l'ensemble des exceptions définies dans ce module et de créer des sous-classes spécifiques d'exceptions pour les différentes conditions d'erreurs :

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
```

(suite sur la page suivante)

```

    expr -- input expression in which the error occurred
    msg  -- explanation of the error
"""

def __init__(self, expr, msg):
    self.expr = expr
    self.msg = msg

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """

    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg

```

Most exceptions are defined with names that end in « Error », similar to the naming of the standard exceptions.

Beaucoup de modules standards définissent leurs propres exceptions pour signaler les erreurs possibles dans les fonctions qu'ils définissent. Plus d'informations sur les classes sont présentées dans le chapitre *Classes*.

## 8.6 Définition d'actions de nettoyage

L'instruction `try` a une autre clause optionnelle qui est destinée à définir des actions de nettoyage devant être exécutées dans certaines circonstances. Par exemple :

```

>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>

```

La clause *finally* est toujours exécutée avant de sortir de l'instruction `try`, qu'une exception soit survenu ou non. Lorsqu'une exception est survenue dans la clause `try` et n'a pas été attrapée par un `except` (ou qu'elle s'est produite dans un `except` ou `else`), elle sera relancée après la fin de l'exécution de la clause *finally*. La clause *finally* est aussi exécutée « en sortant » lorsque n'importe quelle autre clause du `try` est interrompue que ce soit avec un `break`, `continue` ou `return`. Un exemple plus compliqué (ayant un `except` et un *finally* dans le même `try` fonctionne comme en Python 2.6) :

```

>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

Comme vous pouvez le voir, la clause `finally` est exécutée dans tous les cas. L'exception de type `TypeError`, déclenchée en divisant deux chaînes de caractères, n'est pas prise en charge par la clause `except` et est donc propagée après que la clause `finally` a été exécutée.

Dans les vraies applications, la clause `finally` est notamment utile pour libérer des ressources externes (telles que des fichiers ou des connexions réseau), quelle qu'ait été l'utilisation de ces ressources.

## 8.7 Actions de nettoyage prédéfinies

Certains objets définissent des actions de nettoyage standards qui doivent être exécutées lorsque l'objet n'est plus nécessaire, indépendamment du fait que l'opération ayant utilisé l'objet ait réussi ou non. Regardez l'exemple suivant, qui tente d'ouvrir un fichier et d'afficher son contenu à l'écran :

```

for line in open("myfile.txt"):
    print line,

```

Le problème avec ce code est qu'il laisse le fichier ouvert pendant une durée indéterminée après que le code ait fini de s'exécuter. Ce n'est pas un problème avec des scripts simples, mais peut l'être au sein d'applications plus conséquentes. L'instruction `with` permet d'utiliser certains objets comme des fichiers d'une façon qui assure qu'ils seront toujours nettoyés rapidement et correctement :

```

with open("myfile.txt") as f:
    for line in f:
        print line,

```

Dès que l'instruction est exécutée, le fichier *f* est toujours fermé, même si un problème est intervenu pendant l'exécution de ces lignes. D'autres objets qui fournissent des actions de nettoyage prédéfinies l'indiquent dans leur documentation.



La notion de classes en Python s'inscrit dans le langage avec un minimum de syntaxe et de sémantique nouvelles. C'est un mélange des mécanismes rencontrés dans C++ et Modula-3. Les classes fournissent toutes les fonctionnalités standards de la programmation orientée objet : l'héritage de classes autorise les héritages multiples, une classe dérivée peut surcharger les méthodes de sa ou ses classes de base et une méthode peut appeler la méthode d'une classe de base qui possède le même nom. Les objets peuvent contenir n'importe quel nombre ou type de données. De la même manière que les modules, les classes participent à la nature dynamique de Python : elles sont créées pendant l'exécution et peuvent être modifiées après leur création.

Dans la terminologie C++, les membres des classes (y compris les données) sont *publics* (sauf exception, voir *Variables privées et références locales aux classes*) et toutes les fonctions membres sont *virtuelles*. Comme avec Modula-3, il n'y a aucune façon d'accéder aux membres d'un objet à partir de ses méthodes : une méthode est déclarée avec un premier argument explicite représentant l'objet et cet argument est transmis de manière implicite lors de l'appel. Comme avec Smalltalk, les classes elles-mêmes sont des objets. Il existe ainsi une sémantique pour les importer et les renommer. Au contraire de C++ et Modula-3, les types natifs peuvent être utilisés comme classes de base pour être étendus par l'utilisateur. Enfin, comme en C++, la plupart des opérateurs natifs avec une syntaxe spéciale (opérateurs arithmétiques, indigage, etc.) peuvent être redéfinis pour les instances de classes.

Par manque d'ontologie pour parler des classes, nous utilisons parfois des termes de Smalltalk et C++. Nous voulions utiliser les termes de Modula-3 puisque sa sémantique orientée objet est plus proche de celle de Python que C++, mais il est probable que seul un petit nombre de lecteurs les connaissent.

## 9.1 Objets et noms : préambule

Les objets possèdent une existence propre et plusieurs noms peuvent être utilisés (dans divers contextes) pour faire référence à un même objet. Ce concept est connu sous le nom d'alias dans d'autres langages. Il n'apparaît pas au premier coup d'œil en Python et il peut être ignoré tant qu'on travaille avec des types de base immuables (nombres, chaînes, tuples). Cependant, les alias peuvent produire des effets surprenants sur la sémantique d'un code Python mettant en jeu des objets muables comme les listes, les dictionnaires et la plupart des autres types. En général, leur utilisation est bénéfique au programme car les alias se comportent, d'un certain point de vue, comme des pointeurs. Par exemple, transmettre un objet n'a aucun coût car c'est simplement un pointeur qui est transmis par l'implémentation ; et si une fonction modifie un objet passé en

argument, le code à l'origine de l'appel voit le changement. Ceci élimine le besoin d'avoir deux mécanismes de transmission d'arguments comme en Pascal.

## 9.2 Portées et espaces de noms en Python

Avant de présenter les classes, nous devons aborder la notion de portée en Python. Les définitions de classes font d'habiles manipulations avec les espaces de noms, vous devez donc savoir comment les portées et les espaces de noms fonctionnent. Soit dit en passant, la connaissance de ce sujet est aussi utile aux développeurs Python expérimentés.

Commençons par quelques définitions.

Un *espace de noms* est une table de correspondance entre des noms et des objets. La plupart des espaces de noms sont actuellement implémentés sous forme de dictionnaires Python, mais ceci n'est normalement pas visible (sauf pour les performances) et peut changer dans le futur. Comme exemples d'espaces de noms, nous pouvons citer les primitives (fonctions comme `abs()` et les noms des exceptions de base) ; les noms globaux dans un module ; et les noms locaux lors d'un appel de fonction. D'une certaine manière, l'ensemble des attributs d'un objet forme lui-même un espace de noms. L'important à retenir concernant les espaces de noms est qu'il n'y a absolument aucun lien entre les noms de différents espaces de noms ; par exemple, deux modules différents peuvent définir une fonction `maximize` sans qu'il n'y ait de confusion. Les utilisateurs des modules doivent préfixer le nom de la fonction avec celui du module.

À ce propos, nous utilisons le mot *attribut* pour tout nom suivant un point. Par exemple, dans l'expression `z.real`, `real` est un attribut de l'objet `z`. Rigoureusement parlant, les références à des noms dans des modules sont des références d'attributs : dans l'expression `nommodule.nomfonction`, `nommodule` est un objet module et `nomfonction` est un attribut de cet objet. Dans ces conditions, il existe une correspondance directe entre les attributs du module et les noms globaux définis dans le module : ils partagent le même espace de noms <sup>1</sup> !

Les attributs peuvent être en lecture seule ou modifiables. S'ils sont modifiables, l'affectation à un attribut est possible. Les attributs de modules sont modifiables : vous pouvez écrire `nommodule.la_reponse = 42`. Les attributs modifiables peuvent aussi être effacés avec l'instruction `del`. Par exemple, `del nommodule.la_reponse` supprime l'attribut `la_reponse` de l'objet nommé `nommodule`.

Les espaces de noms sont créés à différents moments et ont différentes durées de vie. L'espace de noms contenant les primitives est créé au démarrage de l'interpréteur Python et n'est jamais effacé. L'espace de nom global pour un module est créé lorsque la définition du module est lue. Habituellement, les espaces de noms des modules durent aussi jusqu'à l'arrêt de l'interpréteur. Les instructions exécutées par la première invocation de l'interpréteur, qu'ils soient lus depuis un fichier de script ou de manière interactive, sont considérés comme faisant partie d'un module appelé `__main__`, de façon qu'elles possèdent leur propre espace de noms. (les primitives vivent elles-mêmes dans un module, appelé `builtins`.)

L'espace des noms locaux d'une fonction est créé lors de son appel, puis effacé lorsqu'elle renvoie un résultat ou lève une exception non prise en charge (en fait, « oublié » serait une meilleure façon de décrire ce qui se passe réellement). Bien sûr, des invocations récursives ont chacune leur propre espace de noms.

La *portée* est la zone textuelle d'un programme Python où un espace de noms est directement accessible. « Directement accessible » signifie ici qu'une référence non qualifiée à un nom est cherchée dans l'espace de noms.

Bien que les portées soient déterminées de manière statique, elles sont utilisées de manière dynamique. À n'importe quel moment de l'exécution, il y a au minimum trois portées imbriquées dont les espaces de noms sont directement accessibles :

- la portée la plus au centre, celle qui est consultée en premier, contient les noms locaux ;

---

1. Il existe une exception : les modules disposent d'un attribut secret en lecture seule appelé `__dict__` qui renvoie le dictionnaire utilisé pour implémenter l'espace de noms du module ; le nom `__dict__` est un attribut mais pas un nom global. Évidemment, si vous l'utilisez, vous brisez l'abstraction de l'implémentation des espaces de noms. Il est donc réservé à des choses comme les débogueurs post-mortem.

- les portées des fonctions englobantes, qui sont consultées en commençant avec la portée englobante la plus proche, contiennent des noms non-locaux mais aussi non-globaux ;
- l'avant-dernière portée contient les noms globaux du module courant ;
- la portée englobante, consultée en dernier, est l'espace de noms contenant les primitives.

Si un nom est déclaré global, toutes les références et affectations vont directement dans la portée intermédiaire contenant les noms globaux du module. Dans les autres cas, toutes les variables trouvées au dehors du scope le plus proche seront en lecture seule (toute tentative de modifier une telle variable créera simplement une *nouvelle* variable locale dans la portée la plus au centre, en laissant inchangée la variable du même nom dans la portée englobante).

Habituellement, la portée locale référence les noms locaux de la fonction courante. En dehors des fonctions, la portée locale référence le même espace de noms que la portée globale : l'espace de noms du module. Les définitions de classes créent un nouvel espace de noms dans la portée locale.

Il est important de réaliser que les portées sont déterminées de manière textuelle : la portée globale d'une fonction définie dans un module est l'espace de noms de ce module, quelle que soit la provenance de l'appel à la fonction. En revanche, la recherche réelle des noms est faite dynamiquement au moment de l'exécution. Cependant la définition du langage est en train d'évoluer vers une résolution statique des noms au moment de la « compilation », donc ne vous basez pas sur une résolution dynamique (en réalité, les variables locales sont déjà déterminées de manière statique) !

Une particularité de Python est que si aucune instruction `global` n'est active, les affectations de noms vont toujours dans la portée la plus proche. Les affectations ne copient aucune donnée : elles se contentent de lier des noms à des objets. Ceci est également vrai pour l'effacement : l'instruction `del x` supprime la liaison de `x` dans l'espace de noms référencé par la portée locale. En réalité, toutes les opérations qui impliquent des nouveaux noms utilisent la portée locale : en particulier, les instructions `import` et les définitions de fonctions effectuent une liaison du module ou du nom de fonction dans la portée locale. (L'instruction `global` peut être utilisée pour indiquer qu'une variable particulière doit être dans l'espace de noms global.)

## 9.3 Une première approche des classes

Le concept de classe introduit un peu de syntaxe nouvelle, trois nouveaux types d'objets ainsi que quelques nouveaux éléments de sémantique.

### 9.3.1 Syntaxe de définition des classes

La forme la plus simple de définition d'une classe est la suivante :

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Les définitions de classes, comme les définitions de fonctions (définitions `def`), doivent être exécutées avant d'avoir un effet. Vous pouvez tout à fait placer une définition de classe dans une branche d'une instruction conditionnelle `if` ou encore à l'intérieur d'une fonction.

Dans la pratique, les déclarations dans une définition de classe sont généralement des définitions de fonctions mais d'autres déclarations sont permises et parfois utiles (nous revenons sur ce point plus tard). 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 aux méthodes (à nouveau, tout ceci est expliqué plus loin).

Quand une classe est définie, un nouvel espace de noms est créé et utilisé comme portée locale — Ainsi, toutes les affectations de variables locales entrent dans ce nouvel espace de noms. En particulier, les définitions de

fonctions y lient le nom de la nouvelle fonction.

À la fin de la définition d'une classe, un *objet classe* est créé. C'est, pour simplifier, une encapsulation du contenu de l'espace de noms créé par la définition de classe. Nous revoyons les objets classes dans la prochaine section. La portée locale initiale (celle qui prévaut avant le début de la définition de la classe) est ré-instanciée et l'objet de classe est lié ici au nom de classe donné dans l'en-tête de définition de classe (`ClassName` dans l'exemple).

### 9.3.2 Objets classes

Les objets classes prennent en charge deux types d'opérations : des références à des attributs et l'instanciation.

Les *références d'attributs* utilisent la syntaxe standard utilisée pour toutes les références d'attributs en Python : `obj.nom`. Les noms d'attribut valides sont tous les noms qui se trouvaient dans l'espace de noms de la classe quand l'objet classe a été créé. Donc, si la définition de classe est de cette forme :

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

alors `MyClass.i` et `MyClass.f` sont des références valides à des attributs, renvoyant respectivement un entier et un objet fonction. Les attributs de classes peuvent également être affectés, de sorte que vous pouvez modifier la valeur de `MyClass.i` par affectation. `__doc__` est aussi un attribut valide, renvoyant la *docstring* appartenant à la classe : "A simple example class".

L'*instanciation* de classes utilise la notation des fonctions. Considérez simplement que l'objet classe est une fonction sans paramètre qui renvoie une nouvelle instance de la classe. Par exemple (en considérant la classe définie ci-dessus) :

```
x = MyClass()
```

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

L'opération d'instanciation (en « appelant » un objet classe) crée un objet vide. De nombreuses classes aiment créer des instances personnalisées correspondant à un état initial spécifique. À cet effet, une classe peut définir une méthode spéciale nommée `__init__()`, comme ceci :

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

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

```
x = MyClass()
```

Bien sûr, la méthode `__init__()` peut avoir des arguments pour une plus grande flexibilité. Dans ce cas, les arguments donnés à l'opérateur d'instanciation de classe sont transmis à `__init__()`. Par exemple :

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
```

(suite sur la page suivante)

(suite de la page précédente)

```

...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)

```

### 9.3.3 Objets instances

Maintenant, que pouvons-nous faire avec des objets instances ? Les seules opérations comprises par les objets instances sont des références d'attributs. Il y a deux sortes de noms d'attributs valides, les attributs “données” et les méthodes.

Les *attributs “données”* correspondent à des « variables d'instance » en Smalltalk et aux « membres de données » en C++. Les attributs “données” n'ont pas à être déclarés. Comme les variables locales, ils existent dès lors qu'ils sont assignés une première fois. Par exemple, si `x` est l'instance de `MyClass` créée ci-dessus, le code suivant affiche la valeur 16, sans laisser de trace :

```

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

```

L'autre type de référence à un attribut d'instance est une *méthode*. Une méthode est une fonction qui « appartient à » un objet (en Python, le terme de méthode n'est pas unique aux instances de classes : d'autres types d'objets peuvent aussi avoir des méthodes. Par exemple, les objets listes ont des méthodes appelées `append`, `insert`, `remove`, `sort` et ainsi de suite. Toutefois, dans la discussion qui suit, sauf indication contraire, nous utilisons le terme de méthode exclusivement en référence à des méthodes d'objets instances de classe).

Les noms de méthodes valides d'un objet instance dépendent de sa classe. Par définition, tous les attributs d'une classe qui sont des objets fonction définissent les méthodes correspondantes de ses instances. Donc, dans notre exemple, `x.f` est une référence valide à une méthode car `MyClass.f` est une fonction, mais pas `x.i` car `MyClass.i` n'en est pas une. Attention cependant, `x.f` n'est pas la même chose que `MyClass.f` — Il s'agit d'un *objet méthode*, pas d'un objet fonction.

### 9.3.4 Objets méthode

Le plus souvent, une méthode est appelée juste après avoir été liée :

```
x.f()
```

Dans l'exemple de la classe `MyClass`, cela renvoie la chaîne de caractères `hello world`. Toutefois, il n'est pas nécessaire d'appeler la méthode directement : `x.f` est un objet méthode, il peut être gardé de côté et être appelé plus tard. Par exemple :

```

xf = x.f
while True:
    print xf()

```

affiche `hello world` jusqu'à la fin des temps.

Que se passe-t-il exactement quand une méthode est appelée ? Vous avez dû remarquer que `x.f()` a été appelée dans le code ci-dessus sans argument, alors que la définition de la méthode `f()` spécifie bien qu'elle

prend un argument. Qu'est-il arrivé à l'argument ? Python doit sûrement lever une exception lorsqu'une fonction qui requiert un argument est appelée sans – même si l'argument n'est pas utilisé...

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

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object : this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

### 9.3.5 Classes et variables d'instance

En général, les variables d'instance stockent des informations relatives à chaque instance alors que les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe :

```
class Dog:

    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'
```

Comme nous l'avons vu dans *Objets et noms : préambule*, les données partagées *mutable* (telles que les listes, dictionnaires, etc...) peuvent avoir des effets surprenants. Par exemple, la liste *tricks* dans le code suivant ne devrait pas être utilisée en tant que variable de classe car, dans ce cas, une seule liste est partagée par toutes les instances de *Dog* :

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Une conception correcte de la classe est d'utiliser une variable d'instance à la place :

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

## 9.4 Remarques diverses

Les attributs “données” surchargent les méthodes avec le même nom ; pour éviter des conflits de nommage, qui peuvent causer des bugs difficiles à trouver dans de grands programmes, il est sage d'adopter certaines conventions qui minimisent les risques de conflits. Parmi les conventions possibles, on peut citer la mise en majuscule des noms de méthodes, le préfixe des noms d'attributs “données” par une chaîne courte et unique (parfois juste la caractères souligné) ou l'utilisation de verbes pour les méthodes et de noms pour les attributs “données”.

Les attributs “données” peuvent être référencés par des méthodes comme par des utilisateurs ordinaires (« clients ») d'un objet. En d'autres termes, les classes ne sont pas utilisables pour implémenter des types de données purement abstraits. En fait, il n'est pas possible en Python d'imposer de masquer des données — tout est basé sur des conventions (d'un autre côté, l'implémentation de Python, écrite en C, peut complètement masquer les détails d'implémentation et contrôler l'accès à un objet si nécessaire ; ceci peut être utilisé par des extensions de Python écrites en C).

Les clients doivent utiliser les attributs “données” avec précaution — ils pourraient mettre le désordre dans les invariants gérés par les méthodes avec leurs propres valeurs d'attributs. Remarquez que les clients peuvent ajouter leurs propres attributs “données” à une instance d'objet sans altérer la validité des méthodes, pour autant que les noms n'entrent pas en conflit — là aussi, adopter une convention de nommage peut éviter bien des problèmes.

Il n'y a pas de notation abrégée pour référencer des attributs “données” (ou les autres méthodes !) depuis les méthodes. Nous pensons que ceci améliore en fait la lisibilité des méthodes : il n'y a aucune chance de confondre variables locales et variables d'instances quand on regarde le code d'une méthode.

Souvent, le premier argument d'une méthode est nommé `self`. Ce n'est qu'une convention : le nom `self` n'a aucune signification particulière en Python. Notez cependant que si vous ne suivez pas cette convention, votre

code risque d'être moins lisible pour d'autres programmeurs Python et il est aussi possible qu'un programme qui fasse l'introspection de classes repose sur une telle convention.

Tout objet fonction qui est un attribut de classe définit une méthode pour des instances de cette classe. Il n'est pas nécessaire que le texte de définition de la fonction soit dans la définition de la classe : il est possible d'affecter un objet fonction à une variable locale de la classe. Par exemple :

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Maintenant, `f`, `g` et `h` sont toutes des attributs de la classe `C` et font référence à des fonctions objets. Par conséquent, ce sont toutes des méthodes des instances de `C` — `h` est exactement identique à `g`. Remarquez qu'en pratique, ceci ne sert qu'à embrouiller le lecteur d'un programme.

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

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Les méthodes peuvent faire référence à des noms globaux de la même manière que les fonctions. La portée globale associée à une méthode est le module contenant la définition de la classe (la classe elle-même n'est jamais utilisée en tant que portée globale). Alors qu'il est rare d'avoir une bonne raison d'utiliser des données globales dans une méthode, il y a de nombreuses utilisations légitimes de la portée globale : par exemple, les fonctions et modules importés dans une portée globale peuvent être utilisés par des méthodes, de même que les fonctions et classes définies dans cette même portée. Habituellement, la classe contenant la méthode est elle-même définie dans cette portée globale et, dans la section suivante, nous verrons de bonnes raisons pour qu'une méthode référence sa propre classe.

Toute valeur est un objet et a donc une *classe* (appelée aussi son *type*). Elle est stockée dans `objet.__class__`.

## 9.5 Héritage

Bien sûr, ce terme de « classe » ne serait pas utilisé s'il n'y avait pas d'héritage. La syntaxe pour définir une sous-classe est de cette forme :

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Le nom `BaseClassName` doit être défini dans une portée contenant la définition de la classe dérivée. À la place du nom d'une classe de base, une expression est aussi autorisée. Ceci peut être utile, par exemple, lorsque la classe est définie dans un autre module :

```
class DerivedClassName(modname.BaseClassName):
```

L'exécution d'une définition de classe dérivée se déroule comme pour une classe de base. Quand l'objet de la classe est construit, la classe de base est mémorisée. Elle est utilisée pour la résolution des références d'attributs : si un attribut n'est pas trouvé dans la classe, la recherche se poursuit en regardant dans la classe de base. Cette règle est appliquée récursivement si la classe de base est elle-même dérivée d'une autre classe.

Il n'y a rien de particulier dans l'instanciation des classes dérivées : `DerivedClassName()` crée une nouvelle instance de la classe. Les références aux méthodes sont résolues comme suit : l'attribut correspondant de la classe est recherché, en remontant la hiérarchie des classes de base si nécessaire, et la référence de méthode est valide si cela conduit à une fonction.

Les classes dérivées peuvent surcharger des méthodes de leurs classes de base. Comme les méthodes n'ont aucun privilège particulier quand elles appellent d'autres méthodes d'un même objet, une méthode d'une classe de base qui appelle une autre méthode définie dans la même classe peut en fait appeler une méthode d'une classe dérivée qui la surcharge (pour les programmeurs C++ : toutes les méthodes de Python sont en effet « virtuelles »).

Une méthode dans une classe dérivée peut aussi, en fait, vouloir étendre plutôt que simplement remplacer la méthode du même nom de sa classe de base. L'appel direct à la méthode de la classe de base s'écrit simplement `BaseClassName.nomMethode(self, arguments)`. C'est parfois utile également aux clients (notez bien que ceci ne fonctionne que si la classe de base est accessible en tant que `BaseClassName` dans la portée globale).

Python définit deux fonctions primitives pour gérer l'héritage :

- utilisez `isinstance()` pour tester le type d'une instance : `isinstance(obj, int)` renvoie `True` seulement si `obj.__class__` est égal à `int` ou à une autre classe dérivée de `int` ;
- Utilisez `issubclass()` pour tester l'héritage d'une class : `issubclass(bool, int)` renvoie `True` car la class `bool` est une sous-classe de `int`. Par contre, `issubclass(unicode, str)` renvoie `False` car `unicode` n'est pas une sous-classe de `str` (ils partagent seulement un ancêtre commun, `basestring`).

### 9.5.1 Héritage multiple

Python propose également une forme d'héritage multiple. Une définition de classe ayant plusieurs classes de base ressemble à :

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Pour les anciennes classes, la seule règle est : en profondeur, de gauche à droite. Ainsi, si un attribut n'est pas trouvé dans `NomDeLaClasseDérivée`, il est recherché dans `Base1`, puis (récursivement) dans les classes

de base de **Base1** ; s'il n'y est pas trouvé, il est recherché dans **Base2** et ses classes de base, et ainsi de suite.

(Pour certaines personnes, commencer la recherche « en largeur » — chercher dans **Base2** et **Base3** avant d'aller dans les classes de base de **Base1** — peut sembler plus naturel. Toutefois, ceci nécessiterait de savoir si un attribut particulier de **Base1** est actuellement défini dans **Base1** ou dans l'une de ses classes de base avant de pouvoir envisager les conséquences d'un conflit de nom avec un attribut de **Base2**. La règle qui consiste à rechercher d'abord en profondeur ne fait aucune différence entre des attributs définis localement et des attributs hérités de **Base1**).

Pour les *nouvelles classes*, l'ordre de résolution des méthodes change dynamiquement pour gérer les appels coopératifs de la fonction `super()`. Cette approche est connue dans certains langages supportant l'héritage multiple sous le nom de la « méthode la plus proche » (« call-next-method »), et est plus puissante que le seul appel de la méthode « `super` » que l'on trouve dans les langages ne gérant que l'héritage simple.

Avec les nouvelles classes, l'ordre défini dynamiquement est nécessaire car tous les cas d'héritage multiple comportent un arbre d'héritage en losange (au moins l'une des classes parentes peut être accédée via plusieurs chemins à partir d'une même sous-classe). Par exemple, toutes les nouvelles classes héritent de `object`, donc n'importe quel arbre d'héritage multiple fournit plus d'un chemin pour atteindre `object`. Pour qu'une classe de base ne soit pas appelée plusieurs fois, l'algorithme dynamique linéarise l'ordre de recherche d'une façon qui préserve l'ordre d'héritage, de la gauche vers la droite, spécifié dans chaque classe, qui appelle chaque classe parente une seule fois, qui est monotone (ce qui signifie qu'une classe peut être sous-classée sans affecter l'ordre d'héritage de ses parents). Prises ensemble, ces propriétés permettent de concevoir des classes de façon fiable et extensible dans un contexte d'héritage multiple. Pour plus de détail, consultez <http://www.python.org/download/releases/2.3/mro/>.

## 9.6 Variables privées et références locales aux classes

Les membres « privés », qui ne peuvent être accédés que depuis l'intérieur d'un objet, n'existent pas en Python. Toutefois, il existe une convention respectée par la majorité du code Python : un nom préfixé par un tiret bas (comme `_spam`) doit être considéré comme une partie non publique de l'API (qu'il s'agisse d'une fonction, d'une méthode ou d'un attribut « données »). Il doit être vu comme un détail d'implémentation pouvant faire l'objet de modifications futures sans préavis.

Dès lors qu'il y a un cas d'utilisation valable pour avoir des attributs privés aux classes (notamment pour éviter des conflits avec des noms définis dans des sous-classes), il existe un support (certes limité) pour un tel mécanisme, appelé *name mangling*. Tout identifiant de la forme `__spam` (avec au moins deux tirets bas en tête et au plus un à la fin) est remplacé textuellement par `__classname__spam`, où `classname` est le nom de la classe sans le ou les premiers tirets-bas. Ce « découpage » est effectué sans tenir compte de la position syntaxique de l'identifiant, tant qu'il est présent dans la définition d'une classe.

Ce changement de nom est utile pour permettre à des sous-classes de surcharger des méthodes sans casser les appels de méthodes à l'intérieur d'une classe. Par exemple :

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # private copy of original update() method

class MappingSubclass(Mapping):
```

(suite sur la page suivante)

(suite de la page précédente)

```
def update(self, keys, values):
    # provides new signature for update()
    # but does not break __init__()
    for item in zip(keys, values):
        self.items_list.append(item)
```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Notez que ces règles sont conçues avant tout pour éviter les accidents; il reste possible d'accéder ou de modifier une variable considérée comme privée. Ceci peut même être utile dans certaines circonstances, comme au sein du débogueur.

Notez que le code passé à `exec()`, `eval()` ou `execfile()` ne considère pas le nom de la classe appelante comme étant la classe courante; le même effet s'applique à la directive `global`, dont l'effet est de la même façon restreint au code compilé dans le même ensemble de byte-code. Les mêmes restrictions s'appliquent à `getattr()`, `setattr()` et `delattr()`, ainsi qu'aux références directes à `__dict__`.

## 9.7 Trucs et astuces

Il est parfois utile d'avoir un type de donnée similaire au « record » du Pascal ou au *struct* du C, qui regroupent ensemble quelques attributs “données” nommés. La définition d'une classe vide remplit parfaitement ce besoin :

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

À du code Python qui s'attend à recevoir un type de donnée abstrait spécifique, on peut souvent fournir une classe qui simule les méthodes de ce type. Par exemple, à une fonction qui formate des données extraites d'un objet fichier, vous pouvez lui passer comme argument une instance d'une classe qui implémente les méthodes `read()` et `readline()` en puisant ses données à partir d'un tampon de chaînes de caractères.

Les objets méthodes d'instances ont également des attributs : `m.im_self` est l'instance d'objet avec la méthode `m()`, et `m.im_func` est l'objet fonction correspondant à la méthode.

## 9.8 Les exceptions sont aussi des classes

Les exceptions définies par l'utilisateur sont également définies par des classes. En utilisant ce mécanisme, il est possible de créer des hiérarchies d'exceptions extensibles.

Il y a deux nouvelles formes (sémantiques) pour l'instruction `raise` :

```
raise Class, instance

raise instance
```

Dans la première forme, `instance` doit être une instance de `Class` ou d'une classe dérivée. La seconde forme est un raccourci pour :

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

Une classe dans une clause `except` est compatible avec une exception si elle est de la même classe ou d'une de ses classes dérivées. Mais l'inverse n'est pas vrai, une clause `except` spécifiant une classe dérivée n'est pas compatible avec une classe de base. Par exemple, le code suivant affiche B, C et 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"
```

Notez que si les clauses `except` avaient été inversées (avec `except B` en premier), il aurait affiché B, B, B — la première clause `except` correspondante étant déclenchée.

Quand un message d'erreur est imprimé pour une exception non traitée, la classe de l'exception est indiquée, suivie de deux points, d'un espace et de l'instance convertie en chaîne de caractères via la fonction `str()`.

## 9.9 Itérateurs

Vous avez maintenant certainement remarqué que l'on peut itérer sur la plupart des objets conteneurs 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 mode d'accès est simple, concis et pratique. L'utilisation d'itérateurs imprègne et unifie Python. En arrière plan, l'instruction `for` appelle la fonction `iter()` sur l'objet conteneur. Cette fonction renvoie un

itérateur qui définit la méthode `next()`, laquelle accède aux éléments du conteneur un par un. Lorsqu'il n'y a plus d'élément, `next()` lève une exception `StopIteration` qui indique à la boucle de l'instruction `for` de se terminer. Cet exemple montre comment tout 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 "<stdin>", line 1, in <module>
    it.next()
StopIteration
```

Une fois compris les mécanismes de gestion des itérateurs, il est simple d'ajouter ce comportement à vos classes. Définissez une méthode `__iter__()`, qui retourne un objet disposant d'une méthode `next()`. Si la classe définit elle-même la méthode `next()`, alors `__iter__()` peut simplement 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]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s
```

## 9.10 Générateurs

Les *générateurs* sont des outils simples et puissants pour créer des itérateurs. Ils sont écrits comme des fonctions classiques mais utilisent l'instruction `yield` lorsqu'ils veulent renvoyer des données. À chaque fois qu'il est appelé par `next()`, le générateur reprend son exécution là où il s'était arrêté (en conservant tout son contexte d'exécution). Un exemple montre très bien combien les générateurs sont simples à créer :

```
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 également être fait avec des itérateurs basés sur des classes, comme décrit dans le paragraphe précédent. Si qui fait que les générateurs sont si compacts est que les méthodes `__iter__()` et `next()` sont créées automatiquement.

Une autre fonctionnalité clé est que les variables locales ainsi que le contexte d'exécution sont sauvegardés automatiquement entre les appels. Cela simplifie d'autant plus l'écriture de ces fonctions et rend leur code beaucoup plus lisible qu'avec une approche utilisant des variables d'instance telles que `self.index` et `self.data`.

En plus de la création automatique de méthodes et de la sauvegarde du contexte d'exécution, les générateurs lèvent automatiquement une exception `StopIteration` lorsqu'ils terminent leur exécution. La combinaison de ces fonctionnalités rend très simple la création d'itérateurs, sans plus d'effort que l'écriture d'une fonction classique.

## 9.11 Expressions et générateurs

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

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())
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> 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']
```

## Notes



### 10.1 Interface avec le système d'exploitation

Le module `os` propose des dizaines de fonctions pour interagir avec le système d'exploitation :

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Veillez bien à utiliser `import os` plutôt que `from os import *`, sinon `os.open()` cache la primitive `open()` qui fonctionne différemment.

Les primitives `dir()` et `help()` sont des aides utiles lorsque vous travaillez en mode interactif avec des gros 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 la gestion des fichiers et dossiers, le module `shutil` expose une interface plus abstraite et plus facile à utiliser :

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

## 10.2 Jokers sur les noms de fichiers

Le module `glob` fournit une fonction pour construire des listes de fichiers à partir de motifs :

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

## 10.3 Paramètres passés en ligne de commande

Typiquement, les outils en ligne de commande ont besoin de lire les paramètres qui leur sont donnés. Ces paramètres sont stockés dans la variable `argv` du module `sys` sous la forme d'une liste. Par exemple, l'affichage suivant vient de l'exécution de `python demo.py one two three` depuis la ligne de commande :

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

Le module `getopt` analyse `sys.argv` en utilisant les conventions habituelles de la fonction Unix `getopt()`. Des outils d'analyse des paramètres de la ligne de commande plus flexibles et avancés sont disponibles dans le module `argparse`.

## 10.4 Redirection de la sortie d'erreur et fin d'exécution

Le module `sys` a aussi des attributs pour `stdin`, `stdout` et `stderr`. Ce dernier est utile pour émettre des messages d'avertissement ou d'erreur qui restent visibles même si `stdout` est redirigé :

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

Le moyen le plus direct de terminer un script est d'utiliser `sys.exit()`.

## 10.5 Recherche de motifs dans les chaînes

Le module `re` fournit des outils basés sur les expressions rationnelles permettant des opérations complexes sur les chaînes. C'est une solution optimisée, utilisant une syntaxe concise, pour rechercher des motifs complexes ou effectuer des remplacements complexes dans les chaînes :

```
>>> import re
>>> re.findall(r'\b[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 les opérations sont simples, il est préférable d'utiliser les méthodes des chaînes. Elles sont plus lisibles et plus faciles à déboguer :

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

## 10.6 Mathématiques

Le module `math` donne accès aux fonctions sur les nombres à virgule flottante (*float* en anglais) de la bibliothèque C :

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

Le module `random` offre des outils pour faire des tirages 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 existe de nombreux modules pour accéder à internet et traiter les protocoles. Deux des plus simples sont `urllib2` pour récupérer des données depuis des URLs et `smtplib` pour envoyer des emails :

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:    # look for Eastern 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()
```

Notez que le deuxième exemple a besoin d'un serveur mail tournant localement.

## 10.8 Dates et heures

Le module `datetime` propose des classes pour manipuler les dates et les heures de manière simple ou plus complexe. Bien que faire des calculs de dates et d'heures soit possible, la priorité de l'implémentation est mise sur l'extraction efficace des attributs pour le formatage et la manipulation. Le module gère aussi les objets dépendant des 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 de compression et d'archivage les plus courants sont directement gérés par les modules comme : `zlib`, `gzip`, `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 sont très intéressés par les performances de différentes approches d'un même problème. Python propose un outil de mesure répondant simplement à ces questions.

Par exemple, pour échanger deux variables, il peut être tentant d'utiliser l'empaquetage et le dépaquetage de tuples plutôt que la méthode traditionnelle. Le module `timeit` montre rapidement le léger gain de performance obtenu :

```
>>> 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
```

En opposition à `timeit` et sa granularité fine, `profile` et `pstats` fournissent des outils permettant d'identifier les parties les plus gourmandes en temps d'exécution dans des volumes de code plus grands.

## 10.11 Contrôle qualité

Une approche possible pour développer des application de très bonne qualité est d'écrire des tests pour chaque fonction au fur et à mesure de son développement, puis d'exécuter ces tests fréquemment lors du processus de développement.

Le module `doctest` cherche des tests dans les chaînes de documentation. Un test ressemble à un simple copier-coller d'un appel et son résultat depuis le mode interactif. Cela améliore la documentation en fournissant des exemples tout en prouvant qu'ils sont justes :

```
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` requiert plus d'efforts que le module `doctest` mais il permet de construire un jeu de tests plus complet que l'on fait évoluer dans un fichier séparé :

```
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)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

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

## 10.12 Piles fournies

Python adopte le principe des « piles fournies ». Vous pouvez le constater au travers des fonctionnalités évoluées et solides fournies par ses plus gros paquets. Par exemple :

- Les modules `xmlrpclib` et `SimpleXMLRPCServer` permettent d'appeler des fonctions à distance quasiment sans effort. En dépit du nom des modules, aucune connaissance du XML n'est nécessaire.
- Le paquet `email` est une bibliothèque pour gérer les messages électroniques, incluant les MIME et autre encodages basés sur la RFC 2822. Contrairement à `smtplib` et `poplib`, qui envoient et reçoivent des messages, le paquet `email` est une boîte à outils pour construire, lire des structures de messages complexes (comprenant des pièces jointes), ou implémenter des encodages et protocoles.
- Les paquets `xml.dom` et `xml.sax` fournissent un moyen solide de parser ce format populaire. Aussi, le module `csv` peut lire et écrire dans un format de base de donnée commun. Ensemble, ces modules et paquets simplifient grandement l'échange de donnée entre les applications Python et les autres outils.
- L'internationalisation est possible grâce à de nombreux paquets tels que `gettext`, `locale` ou `codecs`.



---

Brief Tour of the Standard Library — Part II

---

Cette deuxième partie aborde des modules plus à destination des programmeurs professionnels. Ces modules sont rarement nécessaires dans de petits scripts.

## 11.1 Formatage de l’affichage

Le module `repr` fournit une version de `repr()` spécialisées dans l’affichage raccourci de grands conteneurs ou de conteneurs profondément imbriqués :

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

Le module `pprint` propose un contrôle plus fin de l’affichage des objets, aussi bien natifs que ceux définis par l’utilisateur, de manière à être lisible par l’interpréteur. Lorsque le résultat fait plus d’une ligne, il est séparé sur plusieurs lignes et est indenté pour rendre la structure plus visible :

```
>>> 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 des paragraphes de texte pour tenir sur un écran d’une largeur donnée :

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
```

(suite sur la page suivante)

(suite de la page précédente)

```
... 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` utilise une base de données des formats spécifiques à chaque région pour les dates, nombres, etc. L'attribut `grouping` de la fonction de formatage permet de formater directement des nombres avec un séparateur :

```
>>> 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_string("%s%.*f", (conv['currency_symbol'],
...                               conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

## 11.2 Gabarits (Templates)

Le module `string` contient une classe polyvalente : `Template`. Elle permet d'écrire des gabarits (*templates* en anglais) avec une syntaxe simple, dans le but d'être utilisable par des non-développeurs. Ainsi, vos utilisateurs peuvent personnaliser leur application sans la modifier.

Le format utilise des marqueurs formés d'un \$ suivi d'un identifiant Python valide (caractères alphanumériques et tirets-bas). Entourer le marqueur d'accolades permet de lui coller d'autres caractères alphanumériques sans intercaler une espace. Écrire \$\$ produit 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 exception `KeyError` lorsqu'un marqueur n'a pas été fourni, ni dans un dictionnaire, ni sous forme d'un paramètre nommé. Dans certains cas, lorsque la donnée à appliquer peut n'être fournie que partiellement par l'utilisateur, la méthode `safe_substitute()` est plus appropriée car elle laisse tels quels les marqueurs manquants :

```
>>> 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 classes filles de `Template` peuvent définir leur propre délimiteur. Typiquement, un script de renommage de photos par lots peut choisir le symbole pourcent comme marqueur pour les champs tels que la date actuelle, le numéro de l'image ou son format :

```
>>> 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 '{0} --> {1}'.format(filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Une autre utilisation des gabarits consiste à séparer la logique métier des détails spécifiques à chaque format de sortie. Il est ainsi possible de générer des gabarits spécifiques pour les fichiers XML, texte, HTML ...

## 11.3 Traitement des données binaires

Le module `struct` expose les fonctions `pack()` et `unpack()` permettant de travailler avec des données binaires. L'exemple suivant montre comment parcourir une entête de fichier ZIP sans recourir au module `zipfile`. Les marqueurs "H" et "I" représentent des nombres entiers non signés, stockés respectivement sur deux et quatre octets. Le "<" indique qu'ils ont une taille standard et utilisent la convention petit-boutiste :

```
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('<IIHH', 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 Fils d'exécution

Des tâches indépendantes peuvent être exécutées de manière non séquentielle en utilisant des fils d'exécution (*threading* en anglais). Les fils d'exécution peuvent être utilisés pour améliorer la réactivité d'une application

qui interagit avec l'utilisateur pendant que d'autres traitements sont exécutés en arrière-plan. Une autre utilisation typique est de séparer sur deux fils d'exécution distincts les entrées / sorties et le calcul.

Le code suivant donne un exemple d'utilisation du module `threading` exécutant des tâches en arrière-plan pendant que le programme principal continue de s'exécuter :

```
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 avec plusieurs fils d'exécution consiste à coordonner ces fils qui partagent des données ou des ressources. Pour ce faire, le module `threading` expose quelques outils dédiés à la synchronisation comme les verrous (*locks* en anglais), les événements (*events* en anglais), les variables conditionnelles (*condition variables* en anglais) et les sémaphores (*semaphore* en anglais).

Bien que ces outils soient puissants, de simples erreurs de conception engendrent parfois des problèmes difficiles à reproduire. L'approche privilégiée pour coordonner des tâches est de canaliser tous les accès à une ressource dans un seul *thread*, et d'utiliser le module `Queue` pour fournir à ce *thread* les demandes des autres *threads*. Les applications utilisant des `Queue.Queue` pour la communication et la coordination inter-*thread* sont plus simples à concevoir, plus lisibles, et plus fiables.

## 11.5 Journalisation

Le module `logging` est un système de journalisation complet. Dans son utilisation la plus élémentaire, les messages sont simplement envoyés dans un fichier ou sur `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 l'affichage suivant :

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
```

(suite sur la page suivante)

(suite de la page précédente)

```
CRITICAL:root:Critical error -- shutting down
```

Par défaut, les messages d'information et de débogage sont ignorés, les autres sont envoyés vers la sortie standard. Il est aussi possible d'envoyer les messages par courriel, datagrammes, sur des sockets ou vers un serveur HTTP. Des nouveaux filtres permettent d'utiliser des sorties différentes en fonction de la priorité du message : `DEBUG`, `INFO`, `WARNING`, `ERROR` et `CRITICAL`.

La configuration de la journalisation peut être effectuée directement dans le code Python ou peut être chargée depuis un fichier de configuration, permettant de personnaliser la journalisation sans modifier l'application.

## 11.6 Références faibles

Python gère lui-même la mémoire (par comptage des références pour la plupart des objets et en utilisant un *ramasse-miettes* (*garbage collector* en anglais) pour éliminer les cycles). La mémoire est libérée rapidement lorsque sa dernière référence est supprimée.

Cette approche fonctionne bien pour la majorité des applications mais, parfois, il est nécessaire de surveiller un objet seulement durant son utilisation par quelque chose d'autre. Malheureusement, le simple fait de le suivre crée une référence qui rend l'objet permanent. Le module `weakref` expose des outils pour suivre les objets sans pour autant créer une référence. Lorsqu'un objet n'est pas utilisé, il est automatiquement supprimé du tableau des références faibles et une fonction de rappel (*callback* en anglais) est appelée. Un exemple typique est le cache d'objets coûteux à créer :

```
>>> 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 "<stdin>", line 1, in <module>
    d['primary']                             # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

## 11.7 Outils pour les listes

Beaucoup de structures de données peuvent être représentées avec les listes natives. Cependant, d'autres besoins peuvent émerger pour des structures ayant des caractéristiques différentes, typiquement en termes de performance.

Le module `array` fournit un objet `array()` ne permettant de stocker que des listes homogènes mais d'une manière plus compacte. L'exemple suivant montre une liste de nombres stockés chacun sur deux octets non signés (marqueur "H") plutôt que d'utiliser 16 octets comme l'aurait fait une liste classique :

```
>>> 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 la classe `deque()`. Elle ressemble à une liste mais est plus rapide pour l'insertion ou l'extraction des éléments par la gauche et plus lente pour accéder aux éléments du milieu. Ces objets sont particulièrement adaptés pour construire des queues ou des algorithmes de parcours d'arbres en largeur (ou BFS, pour *Breadth First Search* en anglais) :

```
>>> 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 de fournir des implémentations de listes alternatives, la bibliothèque fournit des outils tels que `bisect`, un module contenant des fonctions de manipulation de 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` permet d'implémenter des tas (*heap* en anglais) à partir de simples listes. La valeur la plus faible est toujours à la première position (indice 0). C'est utile dans les cas où l'application accède souvent à l'élément le plus petit mais sans vouloir classer entièrement la liste :

```
>>> 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` expose la classe `Decimal` : elle est spécialisée dans le calcul de nombres décimaux représentés en virgule flottante. Par rapport à la classe native `float`, elle est particulièrement utile pour :

- les applications traitant de finance et autres utilisations nécessitant une représentation décimale exacte,
- le contrôle de la précision,
- le contrôle sur les arrondis pour répondre à des obligations légales ou réglementaires,
- suivre les décimales significatives, ou
- les applications pour lesquelles l'utilisateur attend des résultats identiques aux calculs faits à la main.

Par exemple, calculer 5 % de taxe sur une facture de 70 centimes donne un résultat différent en nombre à virgule flottante binaire et décimale. La différence devient significative lorsqu'on arrondit le résultat au centime près :

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01')) # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

Le résultat d'un calcul donné par `Decimal` conserve les zéros non-significatifs. La classe conserve automatiquement quatre décimales significatives pour des opérandes à deux décimales significatives. La classe `Decimal` imite les mathématiques telles qu'elles pourraient être effectuées à la main, évitant les problèmes typiques de l'arithmétique binaire à virgule flottante qui n'est pas capable de représenter exactement certaines quantités décimales.

La représentation exacte de la classe `Decimal` lui permet de faire des calculs de modulo ou des tests d'égalité qui ne seraient pas possibles avec une représentation à virgule flottante binaire :

```
>>> 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` permet de faire des calculs avec autant de précision que nécessaire :

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



La lecture de ce tutoriel a probablement renforcé votre intérêt pour Python et vous devez être impatient de l'utiliser pour résoudre des vrais problèmes. Où aller pour en apprendre plus ?

Ce tutoriel fait partie de la documentation de Python, et la documentation de Python est vaste :

- `library-index` :  
Nous vous conseillons de naviguer dans le manuel, c'est une référence complète (quoique laconique) sur les types, fonctions et modules de la bibliothèque standard. La distribution standard de Python inclut *énormément* de code supplémentaire. Il existe des modules pour lire les courriels, récupérer des documents *via* HTTP, générer des nombres aléatoires, analyser les paramètres de la ligne de commande, écrire des programmes CGI, compresser des données et beaucoup d'autres fonctions. Une balade dans la documentation de la bibliothèque vous donnera une idée de ce qui est disponible.
- `install-index` explique comment installer des modules externes écrits par d'autres utilisateurs de Python.
- `reference-index` contient une explication détaillée de la syntaxe et sémantique de Python. C'est une lecture fastidieuse, mais elle a sa place dans une documentation exhaustive.

D'autres ressources :

- <https://www.python.org> est le site principal pour Python. Il contient du code, de la documentation, des liens vers d'autres sites traitant de Python partout sur Internet. Il est répliqué dans différents endroits autour du globe, comme en Europe, au Japon et en Australie. Ces répliques peuvent dans certains cas être plus rapides que le site principal, en fonction de l'endroit où vous vous trouvez.
- <https://docs.python.org/fr/> offre un accès rapide à la documentation de Python en français.
- <https://pypi.org> : The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- <https://code.activestate.com/recipes/langs/python/> : « The Python Cookbook » est un recueil assez imposant d'exemples de code, de modules et de scripts. Les contributions les plus remarquables y sont regroupées dans le livre « Python Cookbook » (O'Reilly & Associates, ISBN 0-596-00797-3).

Pour les questions et problèmes liés à Python, vous pouvez écrire sur le newsgroup `comp.lang.python`, ou les envoyer à la mailing-list `python-list@python.org`. Le newsgroup et la mailing list sont liés, les messages écrits d'un côté sont automatiquement transférés de l'autre. Il y a environ 120 messages par jour, (avec des pics autour de plusieurs centaines) demandant (et répondant) à des questions, suggérant de nouvelles fonctionnalités, et annonçant de nouveaux modules. Avant d'y écrire, assurez-vous d'avoir lu la Foire Aux Questions (autrement nommée FAQ). Les archives de la mailing list sont disponibles à <https://mail.python>.

[org/pipermail/](http://org/pipermail/). La FAQ répond à la plupart des questions qui reviennent fréquemment, et peut contenir la solution à votre problème.

---

### Édition interactive des entrées et substitution d'historique

---

Certaines versions de l'interpréteur Python prennent en charge l'édition de la ligne d'entrée courante et la substitution d'historique, similaires aux facilités que l'on trouve dans le shell Korn et dans le shell GNU Bash. C'est implémenté en utilisant la bibliothèque [GNU Readline](#), qui supporte l'édition en style Emacs et en style Vi. La bibliothèque a sa propre documentation qui ne va pas être dupliquée ici ; toutefois, les bases seront expliquées rapidement. L'éditeur interactif et la gestion d'historique décrits ici sont disponibles en option sous les versions Unix et Cygwin de l'interpréteur.

Ce chapitre *ne décrit pas* les possibilités d'édition du paquet PythonWin de Mark Hammond ou de l'environnement basé sur Tk, IDLE, distribué avec Python. Le rappel d'historique des lignes de commandes qui fonctionne dans des fenêtres DOS sous NT ou d'autres variétés de Windows est encore une autre histoire.

#### 13.1 Édition de ligne

Si elle est supportée, l'édition de la ligne d'entrée est active lorsque l'interpréteur affiche une invite primaire ou secondaire. La ligne courante peut être éditée en utilisant les caractères de contrôles traditionnels d'Emacs. Les plus importants d'entre eux sont : **C-A** (Contrôle-A) déplace le curseur au début de la ligne, **C-E** à la fin, **C-B** d'un caractère vers la gauche, **C-F** d'un caractère vers la droite. La touche retour arrière efface le caractère à la gauche du curseur, **C-D** le caractère à sa droite. **C-K** supprime le reste de la ligne à la droite du curseur, **C-Y** rappelle la dernière chaîne supprimée. **C-tiret bas** annule la dernière modification ; cette commande peut être répétée pour cumuler ses effets.

#### 13.2 Substitution d'historique

La substitution d'historique fonctionne comme ceci. Toutes les lignes non vides reçues sont enregistrées dans un tampon d'historique et, à chaque fois qu'une invite est affichée, vous êtes positionné sur une nouvelle ligne à la fin de ce tampon. **C-P** remonte d'une ligne (en arrière) dans ce tampon, **C-N** descend d'une ligne. Chaque ligne présente dans le tampon d'historique peut être éditée ; un astérisque apparaît en face de l'invite pour indiquer qu'une ligne a été modifiée. Presser la touche **Entrée** envoie la ligne en cours à l'interpréteur. **C-R** démarre une recherche incrémentale en arrière ; **C-S** démarre une recherche en avant.

## 13.3 Raccourcis clavier

Les raccourcis clavier ainsi que d'autres paramètres de la bibliothèque Readline peuvent être personnalisés en inscrivant des commandes dans un fichier d'initialisation appelé `~/.inputrc`. Les raccourcis sont de la forme

```
key-name: function-name
```

ou :

```
"string": function-name
```

et les options peuvent être définies avec

```
set option-name value
```

Par exemple :

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Notez que le raccourci par défaut pour la touche **Tab** en Python est d'insérer un caractère de tabulation au lieu de la fonction par défaut de complétion des noms de fichiers de la bibliothèque Readline. Si vous y tenez, vous pouvez surcharger ce comportement en indiquant

```
Tab: complete
```

dans votre fichier `~/.inputrc` (bien sûr, avec cela il devient plus difficile d'entrer des lignes de continuation indentées si vous êtes habitué à utiliser **Tab** dans ce but).

La complétude automatique des noms de variables et de modules est disponible de manière optionnelle. Pour l'activer dans le mode interactif de l'interpréteur, ajoutez les lignes suivantes à votre fichier de démarrage :<sup>1</sup>

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

Ceci lie la touche **Tab** à la fonction de complétude, donc taper la touche **Tab** deux fois de suite suggère les options disponibles ; la recherche s'effectue dans les noms d'instructions Python, les noms de variables locales et les noms de modules disponibles. Pour les expressions pointées telles que `string.a`, l'expression est évaluée jusqu'au dernier `'.'` avant de suggérer les options disponibles à partir des attributs de l'objet résultant de cette évaluation. Notez bien que ceci peut exécuter une partie du code de l'application si un objet disposant d'une méthode `__getattr__()` fait partie de l'expression.

Un fichier de démarrage plus complet peut ressembler à cet exemple. Notez que celui-ci supprime les noms qu'il crée dès qu'ils ne sont plus nécessaires ; ceci est possible car le fichier de démarrage est exécuté dans le même espace de noms que les commandes interactives, et supprimer les noms évite de générer des effets

---

1. Python exécute le contenu d'un fichier identifié par la variable d'environnement `PYTHONSTARTUP` lorsque vous démarrez un interpréteur interactif. Pour personnaliser Python y compris en mode non interactif, consultez *Modules de personnalisation*.

de bord dans l'environnement interactif. Vous pouvez trouver pratique de conserver certains des modules importés, tels que `os`, qui s'avère nécessaire dans la plupart des sessions avec 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: "export PYTHONSTARTUP=~/.pystartup" in bash.

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
```

## 13.4 Alternatives à l'interpréteur interactif

Cette facilité constitue un énorme pas en avant comparé aux versions précédentes de l'interpréteur. Toutefois, il reste des fonctions à implémenter comme l'indentation correcte sur les lignes de continuation (l'analyseur sait si une indentation doit suivre) ; le mécanisme de complétion devrait utiliser la table de symboles de l'interpréteur. Une commande pour vérifier (ou même suggérer) les correspondances de parenthèses, de guillemets..., serait également utile.

Une alternative améliorée de l'interpréteur interactif est développée depuis maintenant quelques temps : [IPython](#). Il fournit la complétion, l'exploration d'objets et une gestion avancée de l'historique. Il peut également être personnalisé en profondeur et embarqué dans d'autres applications. Un autre environnement interactif amélioré similaire est [bpython](#).

### Notes



---

Arithmétique en nombres à virgule flottante : problèmes et limites

---

Les nombres à virgule flottante sont représentés, au niveau matériel, en fractions de nombres binaires (base 2). Par exemple, la fraction décimale :

0.125

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

0.001

a la valeur  $0/2 + 0/4 + 1/8$ . Ces deux fractions ont une valeur identique, la seule différence est que la première est une fraction décimale, la seconde est une fraction binaire.

Malheureusement, la plupart des fractions décimales ne peuvent pas avoir de représentation exacte en fractions binaires. Par conséquent, en général, les nombres à virgule flottante que vous donnez sont seulement approximatés en fractions binaires pour être stockés dans la machine.

Le problème est plus simple à aborder en base 10. Prenons par exemple, la fraction  $1/3$ . Vous pouvez l'approximer en une fraction décimale :

0.3

ou, mieux,

0.33

ou, mieux,

0.333

etc. Peu importe le nombre de décimales que vous écrivez, le résultat ne vaut jamais exactement  $1/3$ , mais c'est une estimation s'en approchant toujours mieux.

De la même manière, peu importe combien de décimales en base 2 vous utilisez, la valeur décimale 0.1 ne peut pas être représentée exactement en fraction binaire. En base 2,  $1/10$  est le nombre périodique suivant :

```
0.00011001100110011001100110011001100110011001100110011...
```

Arrêtez à n'importe quelle quantité finie de bits, et vous obtiendrez une approximation.

Pour Python, sur une machine typique, 53 bits sont utilisés pour la précision d'un flottant, donc la valeur stockée lorsque vous entrez le nombre décimal 0.1 est la fraction binaire

```
0.0001100110011001100110011001100110011001100110011010
```

qui est proche, mais pas exactement égale, à 1/10.

Il est facile d'oublier que la valeur stockée est une approximation de la fraction décimale d'origine, du fait de la manière dont les flottants sont affichés dans l'interpréteur. Python n'affiche qu'une approximation décimale de la valeur stockée en binaire. Si Python devait afficher la vraie valeur décimale de l'approximation binaire stockée pour 0.1, il afficherait :

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

C'est bien plus de décimales que ce qu'attendent la plupart des utilisateurs, donc Python affiche une valeur arrondie afin d'améliorer la lisibilité :

```
>>> 0.1
0.1
```

Il est important de comprendre qu'en réalité, c'est une illusion : la valeur stockée n'est pas exactement 1/10, c'est simplement à *l'affichage* que la valeur stockée est arrondie. Ceci devient évident dès que vous effectuez des opérations arithmétiques avec ces valeurs :

```
>>> 0.1 + 0.2
0.30000000000000004
```

Ce comportement est inhérent à la nature même de la représentation des nombres à virgule flottante dans la machine : ce n'est pas un bogue dans Python et ce n'est pas non plus un bogue dans votre code. Vous pouvez observer le même type de comportement dans tous les autres langages utilisant le support matériel pour le calcul des nombres à virgule flottante (bien que certains langages ne rendent pas visible la différence par défaut, ou pas dans tous les modes d'affichage).

Une autre surprise est inhérente à celle-ci. Par exemple, si vous tentez d'arrondir la valeur 2.675 à deux décimales, vous obtiendrez

```
>>> round(2.675, 2)
2.67
```

La documentation de la primitive `round()` indique qu'elle arrondit à la valeur la plus proche en s'éloignant de zéro. Puisque la fraction décimale est exactement à mi-chemin entre 2.67 et 2.68, vous devriez vous attendre à obtenir (une approximation binaire de) 2.68. Ce n'est pourtant pas le cas, car lorsque la fraction décimale 2.675 est convertie en flottant, elle est stockée par une approximation dont la valeur exacte est

```
2.67499999999999982236431605997495353221893310546875
```

Puisque l'approximation est légèrement plus proche de 2.67 que 2.68, l'arrondi se fait vers le bas.

Si vous êtes dans une situation où les arrondis de nombres décimaux à mi-chemin ont de l'importance, vous devriez utiliser le module `decimal`. Soit dit en passant, le module `decimal` propose aussi un moyen pratique de « voir » la valeur exacte stockée pour n'importe quel flottant.

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

Une autre conséquence du fait que 0.1 n'est pas exactement stocké 1/10 est que la somme de dix valeurs de 0.1 ne donne pas 1.0 non plus :

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

L'arithmétique des nombres binaires à virgule flottante réserve beaucoup de surprises de ce genre. Le problème avec « 0.1 » est expliqué en détails ci-dessous, dans la section « Erreurs de représentation ». Voir [The Perils of Floating Point](#) pour une liste plus complète de ce genre de surprises.

Il est vrai qu'il n'existe pas de réponse simple, cependant ne vous méfiez pas trop des nombres à virgule flottante ! Les erreurs, en Python, dans les opérations de nombres à virgule flottante sont dues au matériel sous-jacent, et sur la plupart des machines ne sont pas plus importantes que 1 sur  $2^{53}$  par opération. C'est plus que nécessaire pour la plupart des tâches, mais vous devez garder à l'esprit que ce ne sont pas des opérations décimales, et que chaque opération sur des nombres à virgule flottante peut souffrir d'une nouvelle erreur.

Bien que des cas pathologiques existent, pour la plupart des cas d'utilisations courants vous obtiendrez le résultat attendu à la fin en arrondissant simplement au nombre de décimales désirées à l'affichage. Pour un contrôle fin sur la manière dont les flottants sont affichés, consultez dans `formatstrings` les spécifications de formatage de la méthode `str.format()`

## 14.1 Erreurs de représentation

Cette section explique en détail l'exemple du « 0.1 » et montre comment vous pouvez effectuer une analyse exacte de ce type de cas par vous-même. Nous supposons que la représentation binaire des nombres flottants vous est familière.

Le terme *Representation error* signifie que la plupart des fractions décimales ne peuvent être représentées exactement en binaire. C'est la principale raison pour laquelle Python (ou Perl, C, C++, Java, Fortran, et beaucoup d'autres) n'affiche habituellement pas le résultat exact en décimal :

```
>>> 0.1 + 0.2
0.30000000000000004
```

Pourquoi ? 1/10 et 2/10 ne sont pas représentable de manière exacte en fractions binaires. Cependant, toutes les machines d'aujourd'hui (Juillet 2010) suivent la norme IEEE-754 en ce qui concerne l'arithmétique des nombres à virgule flottante, et la plupart des plateformes utilisent un « IEEE-754 double precision » pour représenter les floats de Python. Les « IEEE-754 double precision » utilisent 53 bits de précision, donc à la lecture l'ordinateur essaie de convertir 0.1 dans la fraction la plus proche possible de la forme  $J/2^{**N}$  avec  $J$  un nombre entier d'exactement 53 bits. Réécrire :

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

en :

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

en se rappelant que  $J$  fait exactement 53 bits (donc  $\geq 2^{52}$  mais  $< 2^{53}$ ), la meilleure valeur possible pour  $N$  est 56 :

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

Donc 56 est la seule valeur possible pour  $N$  qui laisse exactement 53 bits pour  $J$ . La meilleure valeur possible pour  $J$  est donc ce quotient, arrondi :

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

Puisque la retenue est plus grande que la moitié de 10, la meilleure approximation est obtenue en arrondissant par le haut :

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

Par conséquent la meilleure approximation possible pour  $1/10$  en « IEEE-754 double precision » est cette au desus de  $2^{56}$ , soit :

```
7205759403792794 / 72057594037927936
```

Notez que puisque l'arrondi a été fait vers le haut, le résultat est en réalité légèrement plus grand que  $1/10$  ; si nous n'avions pas arrondi par le haut, le quotient aurait été légèrement plus petit que  $1/10$ . Mais dans aucun cas il ne vaut *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 utilisant les nombres à virgule flottante double précision de l'« IEEE-754 » :

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

Si on multiplie cette fraction par  $10^{30}$ , on peut observer les valeurs de ses 30 décimales de poids fort.

```
>>> 7205759403792794 * 10**30 // 2**56
1000000000000000005551115123125L
```

signifiant que la valeur exacte stockée dans l'ordinateur est approximativement égale à la valeur décimale 0.1000000000000000005551115123125. Dans les versions antérieures à Python 2.7 et Python 3.1, Python arrondissait ces valeurs à 17 décimales significatives, affichant “0.10000000000000001”. Dans les versions actuelles de Python, la valeur affichée est la valeur dont la fraction est la plus courte possible tout en redonnant exactement la même représentation une fois reconverti en binaire, affichant simplement “0.1”.

## 15.1 Mode interactif

### 15.1.1 Gestion des erreurs

Quand une erreur se produit, l'interpréteur affiche un message d'erreur et la trace d'appels. En mode interactif, il revient à l'invite de commande primaire ; si l'entrée provient d'un fichier, l'interpréteur se termine avec un code de sortie non nul après avoir affiché la trace d'appels (les exceptions gérées par une clause **except** dans une instruction **try** ne sont pas considérées comme des erreurs dans ce contexte). Certaines erreurs sont inconditionnellement fatales et provoquent la fin du programme avec un code de sortie non nul ; les incohérences internes et, dans certains cas, les pénuries de mémoire sont traitées de la sorte. Tous les messages d'erreur sont écrits sur le flux d'erreur standard ; l'affichage normal des commandes exécutées est écrit sur la sortie standard.

Taper le caractère d'interruption (généralement **Ctrl+C** ou **Supprimer**) au niveau de l'invite de commande primaire annule l'entrée et revient à l'invite<sup>1</sup>. Saisir une interruption tandis qu'une commande s'exécute lève une exception **KeyboardInterrupt** qui peut être gérée par une instruction **try**.

### 15.1.2 Scripts Python exécutables

Sur les systèmes Unix, un script Python peut être rendu directement exécutable, comme un script *shell*, en ajoutant la ligne :

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

(en supposant que l'interpréteur est dans le **PATH** de l'utilisateur) au début du script et en rendant le fichier exécutable. **'#!'** doivent être les deux premiers caractères du fichier. Sur certaines plateformes, cette première ligne doit finir avec une fin de ligne de type Unix (**'\n'**) et pas de type Windows (**'\r\n'**). Notez que le caractère croisillon, **'#'**, est utilisé pour initier un commentaire en Python.

Un script peut être rendu exécutable en utilisant la commande **chmod**.

1. Un problème avec GNU Readline peut l'en empêcher.

```
$ chmod +x myscript.py
```

Sur les système Windows il n'y a pas de « mode exécutable ». L'installateur Python associe automatiquement les fichiers en `.py` avec `python.exe` de telle sorte qu'un double clic sur un fichier Python le lance comme un script. L'extension peut aussi être `.pyw`. Dans ce cas, la console n'apparaît pas.

### 15.1.3 Configuration du mode interactif

En mode interactif, il peut être pratique de faire exécuter quelques commandes au lancement de l'interpréteur. Configurez la variable d'environnement `PYTHONSTARTUP` avec le nom d'un fichier contenant les instructions à exécuter, à la même manière du `.profile` pour un *shell* Unix.

Ce fichier n'est lu qu'en mode interactif, pas quand Python lit les instructions depuis un fichier, ni quand `/dev/tty` est donné explicitement comme fichier source (pour tout le reste, Python se comporte alors comme dans une session interactive). Les instructions de ce fichier sont exécutées dans le même espace de noms que vos commandes, donc les objets définis et modules importés peuvent être utilisés directement dans la session interactive. Dans ce fichier, il est aussi possible de changer les invites de commande `sys.ps1` et `sys.ps2`.

Si vous voulez exécuter d'autres fichiers du dossier courant au démarrage, vous pouvez le programmer dans le fichier de démarrage global, par exemple avec le code suivant : `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Et si vous voulez exécuter le fichier de démarrage depuis un script, vous devez le faire explicitement dans le script :

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
        exec(startup_file)
```

### 15.1.4 Modules de personnalisation

Python peut être personnalisé *via* les modules `sitecustomize` et `usercustomize`. Pour découvrir comment ils fonctionnent, vous devez d'abord trouver l'emplacement de votre dossier « site-packages » utilisateur. Démarrez Python et exécutez ce code :

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

Vous pouvez maintenant y créer un fichier `usercustomize.py` et y écrire ce que vous voulez. Il est toujours pris en compte par Python, peu importe le mode, sauf lorsque vous démarrez avec l'option `-s` qui désactive l'import automatique.

`sitecustomize` fonctionne de la même manière mais est généralement créé par un administrateur et stocké dans le dossier site-packages global. Il est importé avant `usercustomize`. Pour plus de détails, consultez la documentation de `site`.

### Notes

>>> L'invite de commande utilisée par défaut dans l'interpréteur interactif. On la voit souvent dans des exemples de code qui peuvent être exécutés interactivement dans l'interpréteur.

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

**2to3** Outil qui essaie de convertir du code pour Python 2.x en code pour Python 3.x en gérant la plupart des incompatibilités qui peuvent être détectées en analysant la source et parcourant son arbre syntaxique.

*2to3* est disponible dans la bibliothèque standard sous le nom de `lib2to3`; un point d'entrée indépendant est fourni via `Tools/scripts/2to3`. Cf. *2to3-reference*.

**classe de base abstraite** Les classes de base abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes, ou subitement fausse (par exemple avec les méthodes magiques). Les ABC introduisent des sous-classes virtuelles, qui n'héritent pas d'une classe mais qui sont quand même reconnues par `isinstance()` ou `issubclass()` (Voir la documentation du module `abc`). Python contient de nombreuses ABC pour les structures de données (dans le module `collections`), les nombres (dans le module `numbers`), les flux (dans le module `io`). Vous pouvez créer vos propres ABC avec le module `abc`.

**argument** Une valeur, donnée à une *fonction* ou à une *méthode* lors de son appel. Il existe deux types d'arguments :

- *argument nommé* : un argument précédé d'un identifiant (comme `name=`) ou un dictionnaire précédé de `**`, lors d'un appel de fonction. Par exemple, 3 et 5 sont tous les deux des arguments nommés dans l'appel à `complex()` ici :

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argument positionnel* : Un argument qui n'est pas nommé. Les arguments positionnels apparaissent au début de la liste des arguments, ou donnés sous forme d'un *itérable* précédé par `*`. Par exemple, 3 et 5 sont tous les deux des arguments positionnels dans les appels suivants :

```
complex(3, 5)
complex(*(3, 5))
```

Les arguments se retrouvent dans le corps de la fonction appelée parmi les variables locales. Voir la section [calls](#) à propos des règles dictant cette affectation. Syntaxiquement, toute expression est acceptée comme argument, et c'est la valeur résultante de l'expression qui sera affectée à la variable locale.

Voir aussi [parameter](#) dans le glossaire, et la question dans la FAQ à propos de la différence entre argument et paramètre.

**attribut** Valeur associée à un objet et désignée par son nom via une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, il sera référencé par *o.a*.

**BDFL** Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de [Guido van Rossum](#), le créateur de Python.

**Objet bytes-compatible** Un objet gérant le `bufferobjects`, comme les classes `str`, `bytearray`, ou `memoryview`. Les objets bytes-compatibles peuvent manipuler des données binaires et ainsi servir à leur compression, sauvegarde, ou envoi sur une socket. Certaines actions nécessitent que la donnée binaire soit modifiable, ce qui n'est pas possible avec tous les objets byte-compatibles.

**code intermédiaire (*bytecode*)** Le code source, en Python, est compilé en un bytecode, la représentation interne à CPython d'un programme Python. Le bytecode est stocké dans un fichier nommé `.pyc` ou `.pyo`. Ces caches permettent de charger les fichiers plus rapidement lors de la deuxième exécution (en évitant ainsi de recommencer la compilation en bytecode). On dit que ce *langage intermédiaire* est exécuté sur une *machine virtuelle* qui exécute des instructions machine pour chaque instruction du bytecode. Notez que le bytecode n'a pas vocation à fonctionner entre différentes machines virtuelle Python, encore moins entre différentes version de Python.

La documentation du module `dis` fournit une liste des instructions du code intermédiaire.

**classe** Modèle pour créer des objets définis par l'utilisateur. Une définition de classe (*class*) contient normalement des définitions de méthodes qui agissent sur les instances de la classe.

**classic class** Any class which does not inherit from `object`. See [new-style class](#). Classic classes have been removed in Python 3.

**coercition** The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

**nombre complexe** Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de  $-1$ , souvent écrite *i* en mathématiques ou *j* par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe *j*, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

**gestionnaire de contexte** Objet contrôlant l'environnement à l'intérieur d'un bloc `with` en définissant les méthodes `__enter__()` et `__exit__()`. Consultez la [PEP 343](#).

**CPython** L'implémentation canonique du langage de programmation Python, tel que distribué sur [python.org](#). Le terme « CPython » est utilisé dans certains contextes lorsqu'il est nécessaire de distinguer cette implémentation des autres comme *Jython* ou *IronPython*.

**décorateur** Fonction dont la valeur de retour est une autre fonction. Un décorateur est habituellement utilisé pour transformer une fonction via la syntaxe `@wrapper`, dont les exemples typiques sont : `classmethod()` et `staticmethod()`.

La syntaxe des décorateurs est simplement du sucre syntaxique, les définitions des deux fonctions suivantes sont sémantiquement équivalentes :

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation définitions de fonctions et définitions de classes pour en savoir plus sur les décorateurs.

**descripteur** Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Pour plus d'informations sur les méthodes des descripteurs, consultez `descriptors`.

**dictionnaire** An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

**vue de dictionnaire** The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See `dict-views`.

**docstring** Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et placée dans l'attribut `__doc__` de la classe, de la fonction ou du module. Comme cette chaîne est disponible par introspection, c'est l'endroit idéal pour documenter l'objet.

**duck-typing** Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (*Si ça a un bec et que ça cancanne, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`. Notez cependant que le *duck-typing* peut travailler de pair avec les *classes de base abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

**EAFP** Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés `try` et `except`. Cette technique de programmation contraste avec le style *LYBL* utilisé couramment dans les langages tels que C.

**expression** A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

**module d'extension** Module écrit en C ou C++, utilisant l'API C de Python pour interagir avec Python et le code de l'utilisateur.

**objet fichier** Objet exposant une ressource via une API orientée fichier (avec les méthodes `read()` ou `write()`). En fonction de la manière dont il a été créé, un objet fichier peut interfacer l'accès à un fichier sur le disque ou à un autre type de stockage ou de communication (typiquement l'entrée standard, la sortie standard, un tampon en mémoire, une socket réseau, ...). Les objets fichiers sont aussi appelés *file-like-objects* ou *streams*.

There are actually three categories of file objects : raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

**objet fichier-compatible** Synonyme de *objet fichier*.

**chercheur** An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See [PEP 302](#) for details.

**division entière** Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut 2, contrairement à `11 / 4` qui vaut 2.75. Notez que `(-11) // 4` vaut -3 car l'arrondi se fait à l'entier inférieur. Voir la [PEP 328](#).

**fonction** Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et *fonction*.

**\_\_future\_\_** A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to 2. If the module in which it is executed had enabled *true division* by executing :

```
from __future__ import division
```

the expression `11/4` would evaluate to 2.75. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

**ramasse-miettes** (*garbage collection*) Le mécanisme permettant de libérer de la mémoire lorsqu'elle n'est plus utilisée. Python utilise un ramasse-miettes par comptage de référence, et un ramasse-miettes cyclique capable de détecter et casser les références circulaires.

**générateur** A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**expression génératrice** Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une expression `for` définissant une variable de boucle, un intervalle et une expression `if` optionnelle. Toute cette expression génère des valeurs pour la fonction qui l'entoure :

```
>>> sum(i*i for i in range(10))      # sum of squares 0, 1, 4, ... 81
285
```

**GIL** Voir *global interpreter lock*.

**verrou global de l'interpréteur** (*global interpreter lock* en anglais) Mécanisme utilisé par l'interpréteur *CPython* pour s'assurer qu'un seul fil d'exécution (*thread* en anglais) n'exécute le *bytecode* à la fois. Cela simplifie l'implémentation de *CPython* en rendant le modèle objet (incluant des parties critiques comme la classe native `dict`) implicitement protégé contre les accès concourants. Verrouiller l'interpréteur entier rend plus facile l'implémentation de multiples fils d'exécution (*multi-thread* en anglais), au détriment malheureusement de beaucoup du parallélisme possible sur les machines ayant plusieurs processeurs.

Cependant, certains modules d'extension, standards ou non, sont conçus de manière à libérer le GIL lorsqu'ils effectuent des tâches lourdes tel que la compression ou le hachage. De la même manière, le GIL est toujours libéré lors des entrées / sorties.

Les tentatives précédentes d'implémenter un interpréteur Python avec une granularité de verrouillage plus fine ont toutes échouées, à cause de leurs mauvaises performances dans le cas d'un processeur unique. Il est admis que corriger ce problème de performance induit mènerait à une implémentation beaucoup plus compliquée et donc plus coûteuse à maintenir.

**hachable** An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

La hachabilité permet à un objet d'être utilisé comme clé de dictionnaire ou en tant que membre d'un ensemble (type *set*), car ces structures de données utilisent ce *hash*.

Tous les types immuables fournis par Python sont hachables, et aucun type mutable (comme les listes ou les dictionnaires) ne l'est. Toutes les instances de classes définies par les utilisateurs sont hachables par défaut, elles sont toutes différentes selon `__eq__`, sauf comparées à elles mêmes, et leur empreinte (*hash*) est calculée à partir de leur `id()`.

**IDLE** Environnement de développement intégré pour Python. IDLE est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

**immuable** Objet dont la valeur ne change pas. Les nombres, les chaînes et les n-uplets sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée. Ils jouent un rôle important quand une valeur de *hash* constante est requise, typiquement en clé de dictionnaire.

**integer division** Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also [future](#).

**importer** Processus rendant le code Python d'un module disponible dans un autre.

**importateur** Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

**interactif** Python a un interpréteur interactif, ce qui signifie que vous pouvez écrire des expressions et des instructions à l'invite de l'interpréteur. L'interpréteur Python va les exécuter immédiatement et vous en présenter le résultat. Démarrez juste `python` (probablement depuis le menu principal de votre ordinateur). C'est un moyen puissant pour tester de nouvelles idées ou étudier de nouveaux modules (souvenez-vous de `help(x)`).

**interprété** Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

**itérable** An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

**itérateur** An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception

is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Vous trouverez davantage d'informations dans `typeiter`.

**fonction clé** Une fonction clé est un objet callable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction locale `strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions clé pour maîtriser comment les éléments sont triés ou groupés. Typiquement les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, et `itertools.groupby()`.

La méthode `str.lower()` peut servir en fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clé au besoin avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Finalement le module `operator` fournit des constructeurs de fonctions clé : `attrgetter()`, `itemgetter()`, et `methodcaller()`. Voir `Comment Trier` pour avoir des exemples de création et d'utilisation de fonctions clés.

**argument nommé** Voir *argument*.

**lambda** An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

**LBYL** Regarde avant de tomber, (*Look before you leap* en anglais). Ce style de programmation consiste à vérifier des conditions avant d'effectuer des appels ou des accès. Ce style contraste avec le style *EAFP* et se caractérise par la présence de beaucoup d'instructions `if`.

Dans un environnement avec plusieurs fils d'exécution (*multi-threaded* en anglais), le style *LBYL* peut engendrer un séquençage critique (*race condition* en anglais) entre le « regarde » et le « tomber ». Par exemple, le code `if key in mapping: return mapping[key]` peut échouer si un autre fil d'exécution supprime la clé `key` du `mapping` après le test mais avant l'accès. Ce problème peut être résolu avec des verrous (*locks*) ou avec l'approche *EAFP*.

**list** A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is  $O(1)$ .

**liste en compréhension (ou liste en intension)** A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

**chargeur** An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See [PEP 302](#) for details.

**Tableau de correspondances** Un conteneur permettant d'accéder à des éléments par clé et implémente les méthodes spécifiées dans `Mapping` ou `~collections.MutableMapping` :ref:`classes de base abstraites`. Les classes suivantes sont des exemples de mapping : `dict`, `collections.defaultdict`, `collections.OrderedDict`, et `collections.Counter`.

**métaclasses** Classe d'une classe. Les définitions de classe créent un nom pour la classe, un dictionnaire de classe et une liste de classes parentes. La métaclasses a pour rôle de réunir ces trois paramètres pour construire la classe. La plupart des langages orientés objet fournissent une implémentation par défaut. La particularité de Python est la possibilité de créer des métaclasses personnalisées. La plupart des utilisateurs n'aura jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûr les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : `metaclasses`.

**méthode** Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

**ordre de résolution des méthodes** L'ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais) est, lors de la recherche d'un attribut dans les classes parentes, la façon dont l'interpréteur Python classe ces classes parentes. Voir [The Python 2.3 Method Resolution Order](#) pour plus de détails sur l'algorithme utilisé par l'interpréteur Python depuis la version 2.3.

**module** Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de noms et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*. Voir aussi *paquet*.

**MRO** Voir *ordre de résolution des méthodes*.

**muable** Un objet muable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

**n-uplet nommé** (*named-tuple* en anglais) Classe qui, comme un *n-uplet* (*tuple* en anglais), a ses éléments accessibles par leur indice. Et en plus, les éléments sont accessibles par leur nom. Par exemple, `time.localtime()` donne un objet ressemblant à un *n-uplet*, dont `year` est accessible par son indice : `t[0]` ou par son nom : `t.tm_year`).

Un *n-uplet nommé* peut être un type natif tel que `time.struct_time` ou il peut être construit comme une simple classe. Un *n-uplet nommé* complet peut aussi être créé via la fonction `collections.namedtuple()`. Cette dernière approche fournit automatiquement des fonctionnalités supplémentaires, tel qu'une représentation lisible comme `Employee(name='jones', title='programmer')`.

**espace de noms** The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

**portée imbriquée** The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

**nouvelle classe** Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattr__()`.

More information can be found in newstyle.

**objet** N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

**paquet** *module* Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

**paramètre** A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters :

- *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, `foo` et `bar` dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

- *positional-only* : l'argument ne peut être donné que par sa position. Python n'a pas de syntaxe pour déclarer de tels paramètres, cependant des fonctions natives, comme `abs()`, en utilisent.

- *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une \*. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

- *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par \*\*. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

See also the [argument](#) glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

**PEP** Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See [PEP 1](#).

**argument positionnel** Voir [argument](#).

**Python 3000** Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

**Pythonique** Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

```
for i in range(len(food)):
    print food[i]
```

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print piece
```

**nombre de références** Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Le module `sys` définit une fonction `getrefcount()` que les développeurs peuvent utiliser pour obtenir le nombre de références à un objet donné.

**\_\_slots\_\_** A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

**séquence** An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

**tranche** An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`.

The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

**méthode spéciale** (*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans `specialnames`.

**instruction** Une instruction (*statement* en anglais) est un composant d'un « bloc » de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

**struct sequence** A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

**chaîne entre triple guillemets** Chaîne qui est délimitée par trois guillemets simples (') ou trois guillemets doubles ("). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un `\`. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

**type** Le type d'un objet Python détermine quel genre d'objet c'est. Tous les objets ont un type. Le type d'un objet peut être obtenu via son attribut `__class__` ou via `type(obj)`.

**retours à la ligne universels** A manner of interpreting text streams in which all of the following are recognized as ending a line : the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See [PEP 278](#) and [PEP 3116](#), as well as `str.splitlines()` for an additional use.

**environnement virtuel** Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

**machine virtuelle** Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *bytecode* produit par le compilateur de *bytecode*.

**Le zen de Python** Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant « `import this` » dans une invite Python interactive.



---

### À propos de ces documents

---

Ces documents sont générés à partir de sources en [reStructuredText](#) par [Sphinx](#), un analyseur de documents spécialement conçu pour la documentation Python.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page [reporting-bugs](#) qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

- Fred L. Drake, Jr., créateur des outils originaux de la documentation Python et rédacteur de la plupart de son contenu ;
- le projet [Docutils](#) pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
- Fredrik Lundh pour son projet [Alternative Python Reference](#), dont Sphinx a pris beaucoup de bonnes idées.

### B.1 Contributeurs de la documentation Python

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez [Misc/ACKS](#) dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse – Merci !



---

Histoire et licence

---

## C.1 Histoire du logiciel

Python a été créé au début des années 1990 par Guido van Rossum, au Stichting Mathematisch Centrum (CWI, voir <https://www.cwi.nl/>) au Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclut de nombreuses contributions de la part d'autres personnes.

En 1995, Guido continua son travail sur Python au Corporation for National Research Initiatives (CNRI, voir <https://www.cnri.reston.va.us/>) de Reston, en Virginie, d'où il diffusa plusieurs versions du logiciel.

En mai 2000, Guido et l'équipe de développement centrale de Python sont parti vers BeOpen.com pour former l'équipe BeOpen PythonLabs. En octobre de la même année, l'équipe de PythonLabs est partie vers Digital Creations (désormais Zope Corporation ; voir <http://www.zope.com/>). En 2001, la Python Software Foundation (PSF, voir <http://www.python.org/psf/>) voit le jour. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de Python. Zope Corporation en est un sponsor.

Toutes les versions de Python sont Open Source (voir <https://www.opensource.org/> pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
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
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et supérieur	2.1.1	2001-maintenant	PSF	oui

---

**Note :** Compatible GPL ne signifie pas que nous distribuons Python sous licence GPL. Toutes les licences Python, excepté la licence GPL, vous permettent la distribution d'une version modifiée sans rendre open source ces changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non compatible GPL » ne le peuvent pas.

---

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

## C.2 Conditions générales pour accéder à, ou utiliser, Python

### C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.15

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.7.15 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 2.7.15 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2019 Python Software Foundation; All Rights Reserved" are retained in Python 2.7.15 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 2.7.15 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 2.7.15.
4. PSF is making Python 2.7.15 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 2.7.15 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.15 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.15, 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 2.7.15, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

### LICENCE D'UTILISATION LIBRE BEOPEN PYTHON 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.

## C.2.3 LICENCE D'UTILISATION CNRI POUR 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,

(suite sur la page suivante)

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.

## C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 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.

## C.3 Licences et Remerciements pour les logiciels inclus

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

### C.3.1 Mersenne twister

Le module `_random` inclut du code construit à partir d'un téléchargement depuis <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. Voici mot pour mot les commentaires du code original :

A C-program for MT19937, with initialization improved 2002/1/26.  
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`  
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:

1. Redistributions of source code must retain the above copyright  
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  
notice, this list of conditions and the following disclaimer in the  
documentation and/or other materials provided with the distribution.

(suite sur la page suivante)

(suite de la page précédente)

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

### C.3.2 Interfaces de connexion (sockets)

Le module `socket` utilise les fonctions `getaddrinfo()` et `getnameinfo()` codées dans des fichiers source séparés et provenant du projet WIDE : <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.3 Virgule flottante et contrôle d'exception

Le code source pour le module `fpectl` inclut la note suivante :

```

-----
/                               Copyright (c) 1996.                               \
|                               The Regents of the University of California.          |
|                               All rights reserved.                                |
|                                                                                 |
|  Permission to use, copy, modify, and distribute this software for                |
|  any purpose without fee is hereby granted, provided that this en-               |
|  tire notice is included in all copies of any software which is or               |
|  includes a copy or modification of this software and in all                     |
|  copies of the supporting documentation for such software.                       |
|                                                                                 |
|  This work was produced at the University of California, Lawrence                  |
|  Livermore National Laboratory under contract no. W-7405-ENG-48                   |
|  between the U.S. Department of Energy and The Regents of the                    |
|  University of California for the operation of UC LLNL.                           |
|                                                                                 |
|                               DISCLAIMER                                             |
|                                                                                 |
|  This software was prepared as an account of work sponsored by an                 |
|  agency of the United States Government. Neither the United States                |
|  Government nor the University of California nor any of their em-                 |
|  ployees, makes any warranty, express or implied, or assumes any                  |
|  liability or responsibility for the accuracy, completeness, or                    |
|  usefulness of any information, apparatus, product, or process                    |
|  disclosed, or represents that its use would not infringe                        |
|  privately-owned rights. Reference herein to any specific commer-                 |
|  cial products, process, or service by trade name, trademark,                     |
|  manufacturer, or otherwise, does not necessarily constitute or                   |
|  imply its endorsement, recommendation, or favoring by the United                 |
|  States Government or the University of California. The views and                 |
|  opinions of authors expressed herein do not necessarily state or                 |
|  reflect those of the United States Government or the University                   |
|  of California, and shall not be used for advertising or product                  |
|  \ endorsement purposes.                                                           /
-----

```

### C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice :

```

Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

```

(suite sur la page suivante)

(suite de la page précédente)

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch  
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at

<http://www.ietf.org/rfc/rfc1321.txt>

The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 1321; now handles byte order either statically or dynamically.  
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.  
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5); added conditionalization for C++ compilation from Martin Purschke <purschke@bnl.gov>.  
1999-05-03 lpd Original version.

### C.3.5 Interfaces de connexion asynchrones

Les modules `asynchat` et `asyncore` contiennent la note suivante :

Copyright 1996 by Sam Rushing

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 Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,

(suite sur la page suivante)

(suite de la page précédente)

```
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING 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.
```

### C.3.6 Gestion de témoin (*cookie*)

The Cookie module contains the following notice :

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>
```

```
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
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

```
Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley 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.
```

### C.3.7 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com
```

```
Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro
```

```
Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke
```

```
Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro
```

(suite sur la page suivante)

(suite de la page précédente)

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of neither Automatrix, Bioreason or Mojam Media be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

### C.3.8 Les fonctions UUencode et UUdecode

Le module uu contient la note suivante :

Copyright 1994 by Lance Ellinghouse  
Cathedral City, California Republic, United States of America.

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 Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE 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.

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

### C.3.9 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

The xmlrpclib module contains the following notice :

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB  
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

(suite sur la page suivante)

(suite de la page précédente)

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR 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.

### C.3.10 test\_epoll

Le module `test_epoll` contient la note suivante :

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### C.3.11 Select kqueue

Le module `select` contient la note suivante pour l'interface `kqueue` :

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes  
All rights reserved.

Redistribution and use in source and binary forms, with or without

(suite sur la page suivante)

(suite de la page précédente)

modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.12 *strtod* et *dtoa*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversions de *doubles C* vers et depuis les chaînes, et tiré d'un fichier du même nom par David M. Gay, actuellement disponible sur <http://www.netlib.org/fp/>. Le fichier original, tel que récupéré le 16 mars 2009, contient la licence suivante :

```

/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 *****/

```

### C.3.13 OpenSSL

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et Mac OS X peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

## LICENSE ISSUES

=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact [openssl-core@openssl.org](mailto:openssl-core@openssl.org).

## OpenSSL License

-----

```

/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT

```

(suite sur la page suivante)

(suite de la page précédente)

```

* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

-----

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to.  The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code.  The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*    "This product includes cryptographic software written by
*     Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the rouines from the library
*    being used are not cryptographic related :-).

```

(suite sur la page suivante)

(suite de la page précédente)

```

* 4. If you include any Windows specific code (or a derivative thereof) from
*     the apps directory (application code) you must include an acknowledgement:
*     "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed.  i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

### C.3.14 expat

Le module `pyexpat` est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```

### C.3.15 libffi

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008 Red Hat, Inc and others.
```

```
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

```
The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

### C.3.16 zlib

Le module `zlib` est compilé en utilisant une copie du code source de *zlib* si la version de *zlib* trouvée sur le système est trop vieille pour être utilisée :

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler
```

```
This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```
Jean-loup Gailly  
jloup@gzip.org
```

```
Mark Adler  
madler@alumni.caltech.edu
```

## ANNEXE D

---

### Copyright

---

Python et cette documentation sont :

Copyright © 2001-2019 Python Software Foundation. All rights reserved.

Copyright © 2000 *BeOpen.com*. Tous droits réservés.

Copyright © 1995-2000 *Corporation for National Research Initiatives*. Tous droits réservés.

Copyright © 1991-1995 *Stichting Mathematisch Centrum*. Tous droits réservés.

---

Voir [Histoire et licence](#) pour des informations complètes concernant la licence et les permissions.



## Non-alphabetical

..., [115](#)

\*

état, [28](#)

\*\*

état, [29](#)

2to3, [115](#)

>>>, [115](#)

\_\_all\_\_, [54](#)

\_\_builtin\_\_

module, [52](#)

\_\_future\_\_, [118](#)

\_\_slots\_\_, [122](#)

## A

argument, [115](#)

argument nommé, [120](#)

argument positionnel, [122](#)

attribut, [116](#)

## B

BDFL, [116](#)

## C

chaîne entre triple guillemets, [123](#)

chargeur, [120](#)

chercheur, [118](#)

classe, [116](#)

classe de base abstraite, [115](#)

classic class, [116](#)

code intermédiaire (bytecode), [116](#)

coding

style, [30](#)

coercition, [116](#)

compileall

module, [50](#)

CPython, [116](#)

## D

décorateur, [117](#)

descripteur, [117](#)

dictionnaire, [117](#)

division entière, [118](#)

docstring, [117](#)

docstrings, [24](#), [30](#)

documentation strings, [24](#), [30](#)

duck-typing, [117](#)

## E

EAFP, [117](#)

environnement virtuel, [123](#)

espace de noms, [121](#)

état

\*, [28](#)

\*\*, [29](#)

for, [21](#)

expression, [117](#)

expression génératrice, [118](#)

## F

file

objet, [61](#)

fonction, [118](#)

fonction clé, [120](#)

fonction de base

help, [89](#)

open, [61](#)

unicode, [16](#)

for

état, [21](#)

## G

générateur, [118](#)

generator, [118](#)

generator expression, [118](#)

gestionnaire de contexte, [116](#)

GIL, [118](#)

## H

hachable, [119](#)

help

fonction de base, 89

## I

IDLE, 119

immuable, 119

importateur, 119

importer, 119

instruction, 123

integer division, 119

interactif, 119

interprété, 119

itérable, 119

itérateur, 119

## J

json

module, 63

## L

lambda, 120

LBYL, 120

Le zen de Python, 123

list, 120

liste en compréhension (ou liste en intension), 120

## M

machine virtuelle, 123

mangling

name, 82

métaclasses, 120

method

objet, 77

méthode, 121

méthode spéciale, 123

module, 121

\_\_builtin\_\_, 52

compileall, 50

json, 63

readline, 106

rlcompleter, 106

search path, 49

sys, 50

module d'extension, 117

MRO, 121

muable, 121

## N

name

mangling, 82

nombre complexe, 116

nombre de références, 122

nouvelle classe, 121

n-uplet nommé, 121

## O

objet, 121

file, 61

method, 77

Objet bytes-compatible, 116

objet fichier, 118

objet fichier-compatible, 118

open

fonction de base, 61

ordre de résolution des méthodes, 121

## P

paquet, 121

paramètre, 121

PATH, 50, 113

path

module search, 49

PEP, 122

portée imbriquée, 121

Python 3000, 122

Python Enhancement Proposals

PEP 1, 122

PEP 8, 30

PEP 278, 123

PEP 302, 118, 120

PEP 328, 118

PEP 343, 116

PEP 3116, 123

Pythonique, 122

PYTHONPATH, 50, 51

PYTHONSTARTUP, 106, 114

## R

ramasse-miettes, 118

readline

module, 106

retours à la ligne universels, 123

rlcompleter

module, 106

## S

search

path, module, 49

séquence, 122

strings, documentation, 24, 30

struct sequence, 123

style

coding, 30

sys

module, 50

## T

Tableau de correspondances, 120

tranche, 122

type, [123](#)

## U

unicode

fonction de base, [16](#)

## V

variable d'environnement

PATH, [50](#), [113](#)

PYTHONPATH, [50](#), [51](#)

PYTHONSTARTUP, [106](#), [114](#)

verrou global de l'interpréteur, [118](#)

vue de dictionnaire, [117](#)