The Python Language Reference

Version 2.7.18

Guido van Rossum
and the Python development team

mai 20, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

Cette documentation décrit la syntaxe et la « sémantique interne » du langage. Elle peut être laconique, mais essaye d'être exhaustive et exacte. La sémantique des objets natifs secondaires, des fonctions, et des modules est documentée dans library-index. Pour une présentation informelle du langage, voyez plutôt tutorial-index. Pour les développeurs C ou C++, deux manuels supplémentaires existent : extending-index survole l'écriture d'extensions, et c-api-index décrit l'interface C/C++ en détail.

CHAPITRE 1

Introduction

Ce manuel de référence décrit le langage de programmation Python. Il n'a pas vocation à être un tutoriel.

Nous essayons d'être le plus précis possible et nous utilisons le français (NdT : ou l'anglais pour les parties qui ne sont pas encore traduites) plutôt que des spécifications formelles, sauf pour la syntaxe et l'analyse lexicale. Nous espérons ainsi rendre ce document plus compréhensible pour un grand nombre de lecteurs, même si cela laisse un peu de place à l'ambiguïté. En conséquence, si vous arrivez de Mars et que vous essayez de ré-implémenter Python à partir de cet unique document, vous devrez faire des hypothèses et, finalement, vous aurez certainement implémenté un langage sensiblement différent. D'un autre côté, si vous utilisez Python et que vous vous demandez quelles règles s'appliquent pour telle partie du langage, vous devriez trouver une réponse satisfaisante ici. Si vous souhaitez voir une définition plus formelle du langage, nous acceptons toutes les bonnes volontés (ou bien inventez une machine pour nous cloner ☺).

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although alternate implementations exist), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short « implementation notes » sprinkled throughout the text.

Chaque implémentation de Python est livrée avec un certain nombre de modules natifs. Ceux-ci sont documentés dans library-index. Quelques modules natifs sont mentionnés quand ils interagissent significativement avec la définition du langage.

1.1 Autres implémentations

Bien qu'il existe une implémentation Python qui soit de loin la plus populaire, il existe d'autres implémentations qui présentent un intérêt particulier pour différents publics.

Parmi les implémentations les plus connues, nous pouvons citer :

CPython C'est l'implémentation originelle et la plus entretenue de Python, écrite en C. Elle implémente généralement en premier les nouvelles fonctionnalités du langage.

Jython Python implémenté en Java. Cette implémentation peut être utilisée comme langage de script pour les applications Java ou pour créer des applications utilisant des bibliothèques Java. Elle est également souvent utilisée

pour créer des tests de bibliothèques Java. Plus d'informations peuvent être trouvées sur the Jython website (site en anglais).

Python pour .NET Cette implémentation utilise en fait l'implémentation CPython, mais c'est une application .NET et permet un accès aux bibliothèques .NET. Elle a été créée par Brian Lloyd. Pour plus d'informations, consultez la page d'accueil Python pour .NET (site en anglais).

IronPython Un autre Python pour .NET. Contrairement à Python.NET, il s'agit d'une implémentation Python complète qui génère du code intermédiaire (IL) .NET et compile le code Python directement en assemblages .NET. Il a été créé par Jim Hugunin, le programmeur à l'origine de Jython. Pour plus d'informations, voir the IronPython website (site en anglais).

PyPy Une implémentation de Python complètement écrite en Python. Elle apporte des fonctionnalités avancées introuvables dans d'autres implémentations, telles que le fonctionnement sans pile (*stackless* en anglais) et un compilateur à la volée (*Just in Time compiler* en anglais). L'un des objectifs du projet est d'encourager l'expérimentation du langage lui-même en facilitant la modification de l'interpréteur (puisqu'il est écrit en Python). Des informations complémentaires sont disponibles sur la page d'accueil du projet PyPy (site en anglais).

Chacune de ces implémentations diffère d'une manière ou d'une autre par rapport au langage décrit dans ce manuel, ou comporte des spécificités que la documentation standard de Python ne couvre pas. Reportez-vous à la documentation spécifique à l'implémentation pour déterminer ce que vous devez savoir sur l'implémentation que vous utilisez.

1.2 Notations

Les descriptions de l'analyse lexicale et de la syntaxe utilisent une notation de grammaire BNF modifiée. Le style utilisé est le suivant :

```
name        ::=   lc_letter (lc_letter | "_")*
lc_letter   ::=   "a"..."z"
```

La première ligne indique qu'un `name` est un `lc_letter` suivi d'une suite de zéro ou plus `lc_letter`s ou tiret bas. Un `lc_letter` est, à son tour, l'un des caractères `'a'` à `'z'` (cette règle est effectivement respectée pour les noms définis dans les règles lexicales et grammaticales de ce document).

Chaque règle commence par un nom (qui est le nom que la règle définit) et `::=`. Une barre verticale (`|`) est utilisée pour séparer les alternatives ; c'est l'opérateur le moins prioritaire de cette notation. Une étoile (`*`) signifie zéro ou plusieurs répétitions de l'élément précédent ; de même, un plus (`+`) signifie une ou plusieurs répétitions, et une expression entre crochets (`[ ]`) signifie zéro ou une occurrence (en d'autres termes, l'expression encadrée est facultative). Les opérateurs `*` et `+` agissent aussi étroitement que possible ; les parenthèses sont utilisées pour le regroupement. Les chaînes littérales sont entourées de guillemets anglais `"`. L'espace n'est utilisée que pour séparer les lexèmes. Les règles sont normalement contenues sur une seule ligne ; les règles avec de nombreuses alternatives peuvent être formatées avec chaque ligne représentant une alternative (et donc débutant par une barre verticale, sauf la première).

Dans les définitions lexicales (comme dans l'exemple ci-dessus), deux autres conventions sont utilisées : deux caractères littéraux séparés par des points de suspension signifient le choix d'un seul caractère dans la plage donnée (en incluant les bornes) de caractères ASCII. Une phrase entre les signes inférieur et supérieur (`<...>`) donne une description informelle du symbole défini ; par exemple, pour décrire la notion de « caractère de contrôle » si nécessaire.

Même si la notation utilisée est presque la même, il existe une grande différence entre la signification des définitions lexicales et syntaxiques : une définition lexicale opère sur les caractères individuels de l'entrée, tandis qu'une définition syntaxique opère sur le flux de lexèmes générés par l'analyse lexicale. Toutes les notations sous la forme BNF dans le chapitre suivant (« Analyse lexicale ») sont des définitions lexicales ; les notations dans les chapitres suivants sont des définitions syntaxiques.

CHAPITRE 2

Analyse lexicale

Un programme Python est lu par un analyseur syntaxique (*parser* en anglais). En entrée de cet analyseur syntaxique, nous trouvons des lexèmes (*tokens* en anglais), produits par un analyseur lexical. Ce chapitre décrit comment l'analyseur lexical découpe le fichier en lexèmes.

Python uses the 7-bit ASCII character set for program text.

Nouveau dans la version 2.3 : An encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII.

For compatibility with older versions, Python only warns if it finds 8-bit characters ; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

The run-time character set depends on the I/O devices connected to the program but is generally a superset of ASCII.

Future compatibility note : It may be tempting to assume that the character set for 8-bit characters is ISO Latin-1 (an ASCII superset that covers most western languages that use the Latin alphabet), but it is possible that in the future Unicode text editors will become common. These generally use the UTF-8 encoding, which is also an ASCII superset, but with very different use for the characters with ordinals 128-255. While there is no consensus on this subject yet, it is unwise to assume either Latin-1 or UTF-8, even though the current implementation appears to favor Latin-1. This applies both to the source character set and the run-time character set.

2.1 Structure des lignes

Un programme en Python est divisé en *lignes logiques*.

2.1.1 Lignes logiques

La fin d'une ligne logique est représentée par le lexème NEWLINE. Les instructions ne peuvent pas traverser les limites des lignes logiques, sauf quand NEWLINE est autorisé par la syntaxe (par exemple, entre les instructions des instructions composées). Une ligne logique est constituée d'une ou plusieurs *lignes physiques* en fonction des règles, explicites ou implicites, de *continuation de ligne*.

2.1.2 Lignes physiques

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files and strings, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

Lorsque vous encapsulez Python, les chaînes de code source doivent être passées à l'API Python en utilisant les conventions du C standard pour les caractères de fin de ligne : le caractère `\n`, dont le code ASCII est LF.

2.1.3 Commentaires

Un commentaire commence par le caractère croisillon (`#`, *hash* en anglais et qui ressemble au symbole musical dièse, c'est pourquoi il est souvent improprement appelé caractère dièse) situé en dehors d'une chaine de caractères littérale et se termine à la fin de la ligne physique. Un commentaire signifie la fin de la ligne logique à moins qu'une règle de continuation de ligne implicite ne s'applique. Les commentaires sont ignorés au niveau syntaxique, ce ne sont pas des lexèmes.

2.1.4 Déclaration d'encodage

Si un commentaire placé sur la première ou deuxième ligne du script Python correspond à l'expression rationnelle `coding[=:]\s*([-\w.]+)`, ce commentaire est analysé comme une déclaration d'encodage ; le premier groupe de cette expression désigne l'encodage du fichier source. Cette déclaration d'encodage doit être seule sur sa ligne et, si elle est sur la deuxième ligne, la première ligne doit aussi être une ligne composée uniquement d'un commentaire. Les formes recommandées pour l'expression de l'encodage sont :

```
# -*- coding: <encoding-name> -*-
```

qui est reconnue aussi par GNU Emacs et :

```
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenaar's VIM. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`'\xef\xbb\xbf'`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's **notepad**).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, in particular to find the end of a string, and to interpret the contents of Unicode literals. String literals are converted to Unicode for syntactical analysis, then converted back to their original encoding before interpretation starts.

2.1.5 Continuation de ligne explicite

Deux lignes physiques, ou plus, peuvent être jointes pour former une seule ligne logique en utilisant la barre oblique inversée (\) selon la règle suivante : quand la ligne physique se termine par une barre oblique inversée qui ne fait pas partie d'une chaine de caractères ou d'un commentaire, la ligne immédiatement suivante lui est adjointe pour former une seule ligne logique, en supprimant la barre oblique inversée et le caractère de fin de ligne. Par exemple :

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

Une ligne que se termine par une barre oblique inversée ne peut pas avoir de commentaire. La barre oblique inversée ne permet pas de continuer un commentaire. La barre oblique inversée ne permet pas de continuer un lexème, sauf s'il s'agit d'une chaîne de caractères (par exemple, les lexèmes autres que les chaînes de caractères ne peuvent pas être répartis sur plusieurs lignes en utilisant une barre oblique inversée). La barre oblique inversée n'est pas autorisée ailleurs sur la ligne, en dehors d'une chaîne de caractères.

2.1.6 Continuation de ligne implicite

Les expressions entre parenthèses, crochets ou accolades peuvent être réparties sur plusieurs lignes sans utiliser de barre oblique inversée. Par exemple :

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Les lignes continuées implicitement peuvent avoir des commentaires. L'indentation des lignes de continuation n'est pas importante. Une ligne blanche est autorisée comme ligne de continuation. Il ne doit pas y avoir de lexème NEWLINE entre des lignes implicitement continuées. Les lignes continuées implicitement peuvent être utilisées dans des chaînes entre triples guillemets (voir ci-dessous) ; dans ce cas, elles ne peuvent pas avoir de commentaires.

2.1.7 Lignes vierges

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

2.1.8 Indentation

Des espaces ou tabulations au début d'une ligne logique sont utilisées pour connaître le niveau d'indentation de la ligne, qui est ensuite utilisé pour déterminer comment les instructions sont groupées.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes ; the whitespace up to the first backslash determines the indentation.

Note de compatibilité entre les plateformes : en raison de la nature des éditeurs de texte sur les plateformes non Unix, il n'est pas judicieux d'utiliser un mélange d'espaces et de tabulations pour l'indentation dans un seul fichier source. Il convient également de noter que des plateformes peuvent explicitement limiter le niveau d'indentation maximal.

Un caractère de saut de page peut être présent au début de la ligne ; il est ignoré pour les calculs d'indentation ci-dessus. Les caractères de saut de page se trouvant ailleurs avec les espaces en tête de ligne ont un effet indéfini (par exemple, ils peuvent remettre à zéro le nombre d'espaces).

Les niveaux d'indentation de lignes consécutives sont utilisés pour générer les lexèmes INDENT et DEDENT, en utilisant une pile, de cette façon :

Avant que la première ligne du fichier ne soit lue, un « zéro » est posé sur la pile ; il ne sera plus jamais enlevé. Les nombres empilés sont toujours strictement croissants de bas en haut. Au début de chaque ligne logique, le niveau d'indentation de la ligne est comparé au sommet de la pile. S'ils sont égaux, il ne se passe rien. S'il est plus grand, il est empilé et un lexème INDENT est produit. S'il est plus petit, il *doit* être l'un des nombres présents dans la pile ; tous les nombres de la pile qui sont plus grands sont retirés et, pour chaque nombre retiré, un lexème DEDENT est produit. À la fin du fichier, un lexème DEDENT est produit pour chaque nombre supérieur à zéro restant sur la pile.

Voici un exemple de code Python correctement indenté (bien que très confus) :

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

L'exemple suivant montre plusieurs erreurs d'indentation :

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

En fait, les trois premières erreurs sont détectées par l'analyseur syntaxique ; seule la dernière erreur est trouvée par l'analyseur lexical (l'indentation de `return r` ne correspond à aucun niveau dans la pile).

2.1.9 Espaces entre lexèmes

Sauf au début d'une ligne logique ou dans les chaînes de caractères, les caractères « blancs » espace, tabulation et saut de page peuvent être utilisés de manière interchangeable pour séparer les lexèmes. Un blanc n'est nécessaire entre deux lexèmes que si leur concaténation pourrait être interprétée comme un lexème différent (par exemple, ab est un lexème, mais a b comporte deux lexèmes).

2.2 Autres lexèmes

Outre NEWLINE, INDENT et DEDENT, il existe les catégories de lexèmes suivantes : *identifiants*, *mots clés*, *littéraux*, *opérateurs* et *délimiteurs*. Les blancs (autres que les fins de lignes, vus auparavant) ne sont pas des lexèmes mais servent à délimiter les lexèmes. Quand une ambiguïté existe, le lexème correspond à la plus grande chaîne possible qui forme un lexème licite, en lisant de la gauche vers la droite.

2.3 Identifiants et mots-clés

Identifiers (also referred to as *names*) are described by the following lexical definitions :

```
identifier   ::=   (letter|"_") (letter | digit | "_")*
letter       ::=   lowercase | uppercase
lowercase    ::=   "a"..."z"
uppercase    ::=   "A"..."Z"
digit        ::=   "0"..."9"
```

Les identifiants n'ont pas de limite de longueur. La casse est prise en compte.

2.3.1 Mots-clés

Les identifiants suivants sont des mots réservés par le langage et ne peuvent pas être utilisés en tant qu'identifiants normaux. Ils doivent être écrits exactement comme ci-dessous :

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Modifié dans la version 2.4 : `None` became a constant and is now recognized by the compiler as a name for the built-in object `None`. Although it is not a keyword, you cannot assign a different object to it.

Modifié dans la version 2.5 : Using *as* and *with* as identifiers triggers a warning. To use them as keywords, enable the `with_statement` future feature .

Modifié dans la version 2.6 : *as* and *with* are full keywords.

2.3.2 Classes réservées pour les identifiants

Certaines classes d'identifiants (outre les mots-clés) ont une signification particulière. Ces classes se reconnaissent par des caractères de soulignement en tête et en queue d'identifiant :

_* Not imported by `from module import *`. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation ; it is stored in the `__builtin__` module. When not in interactive mode, _ has no special meaning and is not defined. See section *L'instruction import*.

Note : Le nom _ est souvent utilisé pour internationaliser l'affichage ; reportez-vous à la documentation du module `gettext` pour plus d'informations sur cette convention.

__*__ Noms définis par le système. Ces noms sont définis par l'interpréteur et son implémentation (y compris la bibliothèque standard). Les noms actuels définis par le système sont abordés dans la section *Méthodes spéciales*, mais aussi ailleurs. D'autres noms seront probablement définis dans les futures versions de Python. Toute utilisation de noms de la forme `__*__`, dans n'importe quel contexte, qui n'est pas conforme à ce qu'indique explicitement la documentation, est sujette à des mauvaises surprises sans avertissement.

__* Noms privés pour une classe. Les noms de cette forme, lorsqu'ils sont utilisés dans le contexte d'une définition de classe, sont réécrits sous une forme modifiée pour éviter les conflits de noms entre les attributs « privés » des classes de base et les classes dérivées. Voir la section *Identifiants (noms)*.

2.4 Littéraux

Les littéraux sont des notations pour indiquer des valeurs constantes de certains types natifs.

2.4.1 String literals

Les chaînes de caractères littérales sont définies par les définitions lexicales suivantes :

```
stringliteral     ::=   [stringprefix](shortstring | longstring)
stringprefix      ::=   "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
                        | "b" | "B" | "br" | "Br" | "bR" | "BR"
shortstring       ::=   "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring        ::=   "'''" longstringitem* "'''"
                        | '"""' longstringitem* '"""'
shortstringitem   ::=   shortstringchar | escapeseq
longstringitem    ::=   longstringchar | escapeseq
shortstringchar   ::=   <any source character except "\" or newline or the quote>
longstringchar    ::=   <any source character except "\">
escapeseq         ::=   "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the *stringprefix* and the rest of the string literal. The source character set is defined by the encoding declaration ; it is ASCII if no encoding declaration is given in the source file ; see section *Déclaration d'encodage*.

In plain English : String literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter 'r' or 'R' ; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of 'u' or 'U' makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646.

Some additional escape sequences, described below, are available in Unicode strings. A prefix of `'b'` or `'B'` is ignored in Python 2 ; it indicates that the literal should become a bytes literal in Python 3 (e.g. when code is automatically converted with 2to3). A `'u'` or `'b'` prefix may be followed by an `'r'` prefix.

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A « quote » is the character used to open the string, i.e. either `'` or `"`.)

Unless an `'r'` or `'R'` prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are :

Séquence d'échappement	Signification	Notes
`\newline`	Ignoré	
`\\`	barre oblique inversée (`\`)	
`\'`	guillemet simple (`'`)	
`\"`	guillemet double (`"`)	
`\a`	cloche ASCII (BEL)	
`\b`	retour arrière ASCII (BS)	
`\f`	saut de page ASCII (FF)	
`\n`	saut de ligne ASCII (LF)	
`\N{name}`	Character named *name* in the Unicode database (Unicode only)	
`\r`	retour à la ligne ASCII (CR)	
`\t`	tabulation horizontale ASCII (TAB)	
`\uxxxx`	Character with 16-bit hex value *xxxx* (Unicode only)	(1)
`\Uxxxxxxxx`	Character with 32-bit hex value *xxxxxxxx* (Unicode only)	(2)
`\v`	tabulation verticale ASCII (VT)	
`\ooo`	caractère dont le code est *ooo* en octal	(3,5)
`\xhh`	caractère dont le code est *ooo* en hexadécimal	(4,5)

Notes :

(1) Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(2) Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default).

(3) Comme dans le C Standard, jusqu'à trois chiffres en base huit sont acceptés.

(4) Contrairement au C Standard, il est obligatoire de fournir deux chiffres hexadécimaux.

(5) In a string literal, hexadecimal and octal escapes denote the byte with the given value ; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the string*. (This behavior is useful when debugging : if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences marked as « (Unicode only) » in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

When an `'r'` or `'R'` prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal `r"\n"` consists of two characters : a backslash and a lowercase `'n'`. String quotes can be escaped with a backslash, but the backslash remains in the string ; for example, `r"\""` is a valid string literal consisting of two characters : a backslash and a double quote ; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw string cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, *not* as a line continuation.

When an `'r'` or `'R'` prefix is used in conjunction with a `'u'` or `'U'` prefix, then the `\uXXXX` and `\UXXXXXXXX` escape sequences are processed while *all other backslashes are left in the string*. For example, the string literal `ur"\u0062\n"` consists of three Unicode characters : "LATIN SMALL LETTER B", "REVERSE SOLIDUS", and "LATIN SMALL

LETTER N". Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, `\uXXXX` escape sequences are only recognized when there are an odd number of backslashes.

2.4.2 Concaténation de chaînes de caractères

Multiple adjacent string literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, `"hello" 'world'` is equivalent to `"helloworld"`. This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example :

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The "+" operator must be used to concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings).

2.4.3 Littéraux numériques

There are four types of numeric literals : plain integers, long integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

Notez que les littéraux numériques ne comportent pas de signe ; une phrase telle que `-1` est en fait une expression composée de l'opérateur unitaire `-` et du littéral `1`.

2.4.4 Integer and long integer literals

Integer and long integer literals are described by the following lexical definitions :

```
longinteger     ::=   integer ("l" | "L")
integer         ::=   decimalinteger | octinteger | hexinteger | bininteger
decimalinteger  ::=   nonzerodigit digit* | "0"
octinteger      ::=   "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger      ::=   "0" ("x" | "X") hexdigit+
bininteger      ::=   "0" ("b" | "B") bindigit+
nonzerodigit    ::=   "1"..."9"
octdigit        ::=   "0"..."7"
bindigit        ::=   "0" | "1"
hexdigit        ::=   digit | "a"..."f" | "A"..."F"
```

Although both lower case `'l'` and upper case `'L'` are allowed as suffix for long integers, it is strongly recommended to always use `'L'`, since the letter `'l'` looks too much like the digit `'1'`.

Plain integer literals that are above the largest representable plain integer (e.g., 2147483647 when using 32-bit arithmetic) are accepted as if they were long integers instead. [1] There is no limit for long integer literals apart from what can be stored in available memory.

Some examples of plain integer literals (first row) and long integer literals (second and third rows) :

1. In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L   0x100000000L
      79228162514264337593543950336             0xdeadbeef
```

2.4.5 Nombres à virgule flottante littéraux

Les nombres à virgule flottante littéraux sont décrits par les définitions lexicales suivantes :

```
floatnumber     ::=   pointfloat | exponentfloat
pointfloat      ::=   [intpart] fraction | intpart "."
exponentfloat   ::=   (intpart | pointfloat) exponent
intpart         ::=   digit+
fraction        ::=   "." digit+
exponent        ::=   ("e" | "E") ["+" | "-"] digit+
```

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals :

```
3.14    10.    .001    1e100    3.14e-10    0e0
```

Note that numeric literals do not include a sign ; a phrase like `-1` is actually an expression composed of the unary operator `-` and the literal `1`.

2.4.6 Imaginaires littéraux

Les nombres imaginaires sont décrits par les définitions lexicales suivantes :

```
imagnumber   ::=   (floatnumber | intpart) ("j" | "J")
```

Un littéral imaginaire produit un nombre complexe dont la partie réelle est `0.0`. Les nombres complexes sont représentés comme une paire de nombres à virgule flottante et possèdent les mêmes restrictions concernant les plages autorisées. Pour créer un nombre complexe dont la partie réelle est non nulle, ajoutez un nombre à virgule flottante à votre littéral imaginaire. Par exemple `(3+4j)`. Voici d'autres exemples de littéraux imaginaires :

```
3.14j   10.j    10j     .001j   1e100j  3.14e-10j
```

2.5 Opérateurs

Les lexèmes suivants sont des opérateurs :

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

The comparison operators `<>` and `!=` are alternate spellings of the same operator. `!=` is the preferred spelling ; `<>` is obsolescent.

2.6 Délimiteurs

Les lexèmes suivants servent de délimiteurs dans la grammaire :

```
(       )       [       ]       {       }       @
,       :       .       `       =       ;
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis in slices. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

Les caractères ASCII suivants ont une signification spéciale en tant que partie d'autres lexèmes ou ont une signification particulière pour l'analyseur lexical :

```
'       "       #       \
```

Les caractères ASCII suivants ne sont pas utilisés en Python. S'ils apparaissent en dehors de chaines littérales ou de commentaires, ils produisent une erreur :

```
$       ?
```

Notes

CHAPITRE 3

Modèle de données

3.1 Objets, valeurs et types

En Python, les données sont représentées sous forme *d'objets*. Toutes les données d'un programme Python sont représentées par des objets ou par des relations entre les objets (dans un certain sens, et en conformité avec le modèle de Von Neumann « d'ordinateur à programme enregistré », le code est aussi représenté par des objets).

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created ; you may think of it as the object's address in memory. The "`is`" operator compares the identity of two objects ; the `id()` function returns an integer representing its identity (currently implemented as its address). An object's *type* is also unchangeable. [1] An object's type determines the operations that the object supports (e.g., « does it have a length ? ») and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). The *value* of some objects can change. Objects whose value can change are said to be *mutable* ; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed ; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type ; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Un objet n'est jamais explicitement détruit ; cependant, lorsqu'il ne peut plus être atteint, il a vocation à être supprimé par le ramasse-miettes (*garbage-collector* en anglais). L'implémentation peut retarder cette opération ou même ne pas la faire du tout — la façon dont fonctionne le ramasse-miette est particulière à chaque implémentation, l'important étant qu'il ne supprime pas d'objet qui peut encore être atteint.

CPython implementation detail : CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (ex : always close files).

Notez que si vous utilisez les fonctionnalités de débogage ou de trace de l'implémentation, il est possible que des références qui seraient normalement supprimées soient toujours présentes. Notez aussi que capturer une exception avec l'instruction

1. Il *est* possible, dans certains cas, de changer le type d'un objet, sous certaines conditions. Cependant, ce n'est généralement pas une bonne idée car cela peut conduire à un comportement très étrange si ce n'est pas géré correctement.

try...*except* peut conserver des objets en vie.

Some objects contain references to « external » resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The "*try*...*finally*" statement provides a convenient way to do this.

Certains objets contiennent des références à d'autres objets ; on les appelle *conteneurs*. Comme exemples de conteneurs, nous pouvons citer les tuples, les listes et les dictionnaires. Les références sont parties intégrantes de la valeur d'un conteneur. Dans la plupart des cas, lorsque nous parlons de la valeur d'un conteneur, nous parlons des valeurs, pas des identifiants des objets contenus ; cependant, lorsque nous parlons de la muabilité d'un conteneur, seuls les identifiants des objets immédiatement contenus sont concernés. Ainsi, si un conteneur immuable (comme un tuple) contient une référence à un objet muable, sa valeur change si cet objet muable est modifié.

Presque tous les comportements d'un objet dépendent du type de l'objet. Même son identifiant est concerné dans un certain sens : pour les types immuables, les opérations qui calculent de nouvelles valeurs peuvent en fait renvoyer une référence à n'importe quel objet existant avec le même type et la même valeur, alors que pour les objets muables cela n'est pas autorisé. Par exemple, après `a = 1 ; b = 1`, `a` et `b` peuvent ou non se référer au même objet avec la valeur un, en fonction de l'implémentation. Mais après `c = [] ; d = []`, il est garanti que `c` et `d` font référence à deux listes vides distinctes nouvellement créées. Notez que `c = d = []` attribue le même objet à `c` et `d`.

3.2 Hiérarchie des types standards

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.).

Quelques descriptions des types ci-dessous contiennent un paragraphe listant des « attributs spéciaux ». Ces attributs donnent accès à l'implémentation et n'ont, en général, pas vocation à être utilisés. Leur définition peut changer dans le futur.

None Ce type ne possède qu'une seule valeur. Il n'existe qu'un seul objet avec cette valeur. Vous accédez à cet objet avec le nom natif `None`. Il est utilisé pour signifier l'absence de valeur dans de nombreux cas, par exemple pour des fonctions qui ne retournent rien explicitement. Sa valeur booléenne est fausse.

NotImplemented This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods may return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

Ellipse This type has a single value. There is a single object with this value. This object is accessed through the built-in name `Ellipsis`. It is used to indicate the presence of the `...` syntax in a slice. Its truth value is true.

`numbers.Number` Ces objets sont créés par les littéraux numériques et renvoyés en tant que résultats par les opérateurs et les fonctions arithmétiques natives. Les objets numériques sont immuables ; une fois créés, leur valeur ne change pas. Les nombres Python sont bien sûr très fortement corrélés aux nombres mathématiques mais ils sont soumis aux limitations des représentations numériques par les ordinateurs.

Python distingue les entiers, les nombres à virgule flottante et les nombres complexes :

`numbers.Integral` Ils représentent des éléments de l'ensemble mathématique des entiers (positifs ou négatifs).

There are three types of integers :

Plain integers These represent numbers in the range -2147483648 through 2147483647. (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation would fall outside this range, the result is normally returned as a long integer (in some cases, the exception `OverflowError` is raised instead). For the purpose of shift and mask operations, integers are assumed to have a binary, 2's complement notation using 32 or more bits, and hiding no bits from the user (i.e., all 4294967296 different bit patterns correspond to different values).

Long integers Ils représentent les nombres, sans limite de taille, sous réserve de pouvoir être stockés en mémoire (virtuelle). Afin de pouvoir effectuer des décalages et appliquer des masques, on considère qu'ils ont une représentation binaire. Les nombres négatifs sont représentés comme une variante du complément à 2, qui donne l'illusion d'une chaîne infinie de bits de signe s'étendant vers la gauche.

Booleans These represent the truth values False and True. The two objects representing the values `False` and `True` are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation, if it yields a result in the plain integer domain, will yield the same result in the long integer domain or when using mixed operands. The switch between domains is transparent to the programmer.

`numbers.Real` (`float`) Ils représentent les nombres à virgule flottante en double précision, tels que manipulés directement par la machine. Vous dépendez donc de l'architecture machine sous-jacente (et de l'implémentation C ou Java) pour les intervalles gérés et le traitement des débordements. Python ne gère pas les nombres à virgule flottante en précision simple ; les gains en puissance de calcul et mémoire, qui sont généralement la raison de l'utilisation des nombres en simple précision, sont annihilés par le fait que Python encapsule de toute façon ces nombres dans des objets. Il n'y a donc aucune raison de compliquer le langage avec deux types de nombres à virgule flottante.

`numbers.Complex` Ils représentent les nombres complexes, sous la forme d'un couple de nombres à virgule flottante en double précision, tels que manipulés directement par la machine. Les mêmes restrictions s'appliquent que pour les nombres à virgule flottante. La partie réelle et la partie imaginaire d'un nombre complexe `z` peuvent être demandées par les attributs en lecture seule `z.real` et `z.imag`.

Séquences Ils représentent des ensembles de taille finie indicés par des entiers positifs ou nuls. La fonction native `len()` renvoie le nombre d'éléments de la séquence. Quand la longueur d'une séquence est *n*, l'ensemble des indices contient les entiers 0, 1 …, *n-1*. On accède à l'élément d'indice *i* de la séquence *a* par `a[i]`.

Les séquences peuvent aussi être découpées en tranches (*slicing* en anglais) : `a[i:j]` sélectionne tous les éléments d'indice *k* tel que *i* <= *k* < *j*. Quand on l'utilise dans une expression, la tranche est du même type que la séquence. Ceci veut dire que l'ensemble des indices de la tranche est renuméroté de manière à partir de 0.

Quelques séquences gèrent le « découpage étendu » (*extended slicing* en anglais) avec un troisième paramètre : `a[i:j:k]` sélectionne tous les éléments de *a* d'indice *x* où `x = i + n*k`, avec *n* >= `0` et *i* <= *x* < *j*.

Les séquences se différencient en fonction de leur muabilité :

Séquences immuables Un objet de type de séquence immuable ne peut pas être modifié une fois qu'il a été créé. Si l'objet contient des références à d'autres objets, ces autres objets peuvent être muables et peuvent être modifiés ; cependant, les objets directement référencés par un objet immuable ne peuvent pas être modifiés.

Les types suivants sont des séquences immuables :

Chaînes de caractères The items of a string are characters. There is no separate character type ; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0–127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.

(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule ?)

Unicode The items of a Unicode object are Unicode code units. A Unicode code unit is represented by a Unicode object of one item and can hold either a 16-bit or 32-bit value representing a Unicode ordinal (the maximum value for the ordinal is given in `sys.maxunicode`, and depends on how Python is configured at compile time). Surrogate pairs may be present in the Unicode object, and will be reported as two separate items. The built-in functions `unichr()` and `ord()` convert between code units and nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode()` and the built-in function `unicode()`.

Tuples Les éléments d'un tuple sont n'importe quels objets Python. Les tuples de deux ou plus éléments sont formés par une liste d'expressions dont les éléments sont séparés par des virgules. Un tuple composé d'un seul élément (un « singleton ») est formé en suffixant une expression avec une virgule (une expression en tant que telle ne crée pas un tuple car les parenthèses doivent rester disponibles pour grouper les expressions). Un tuple vide peut être formé à l'aide d'une paire de parenthèses vide.

Séquences muables Les séquences muables peuvent être modifiées après leur création. Les notations de tranches et de sous-ensembles peuvent être utilisées en tant que cibles d'une assignation ou de l'instruction `del` (suppression).

Il existe aujourd'hui deux types intrinsèques de séquences muables :

Listes N'importe quel objet Python peut être élément d'une liste. Les listes sont créées en plaçant entre crochets une liste d'expressions dont les éléments sont séparés par des virgules (notez que les listes de longueur 0 ou 1 ne sont pas des cas particuliers).

Tableaux d'octets A bytearray object is a mutable array. They are created by the built-in `bytearray()` constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable bytes objects.

The extension module `array` provides an additional example of a mutable sequence type.

Ensembles Ils représentent les ensembles d'objets, non ordonnés, finis et dont les éléments sont uniques. Tels quels, ils ne peuvent pas être indicés. Cependant, il est possible d'itérer dessus et la fonction native `len()` renvoie le nombre d'éléments de l'ensemble. Les utilisations classiques des ensembles sont les tests d'appartenance rapides, la suppression de doublons dans une séquence et le calcul d'opérations mathématiques telles que l'intersection, l'union, la différence et le complémentaire.

Pour les éléments des ensembles, les mêmes règles concernant l'immuabilité s'appliquent que pour les clés de dictionnaires. Notez que les types numériques obéissent aux règles normales pour les comparaisons numériques : si deux nombres sont égaux (pour l'opération de comparaison, par exemple `1` et `1.0`), un seul élément est conservé dans l'ensemble.

Actuellement, il existe deux types d'ensembles natifs :

Ensembles Ils représentent les ensembles muables. Un ensemble est créé par la fonction native constructeur `set()` et peut être modifié par la suite à l'aide de différentes méthodes, par exemple `add()`.

Ensembles gelés Ils représentent les ensembles immuables. Ils sont créés par la fonction native constructeur `frozenset()`. Comme un ensemble gelé est immuable et *hachable*, il peut être utilisé comme élément d'un autre ensemble ou comme clé de dictionnaire.

Tableaux de correspondances Ils représentent les ensembles finis d'objets indicés par des ensembles index arbitraires. La notation `a[k]` sélectionne l'élément indicé par `k` dans le tableau de correspondance `a` ; elle peut être utilisée dans des expressions, comme cible d'une assignation ou avec l'instruction `del`. La fonction native `len()` renvoie le nombre d'éléments du tableau de correspondances.

Il n'existe actuellement qu'un seul type natif pour les tableaux de correspondances :

Dictionnaires Ils représentent les ensembles finis d'objets indicés par des valeurs presque arbitraires. Les seuls types de valeurs non reconnus comme clés sont les valeurs contenant des listes, des dictionnaires ou les autres types muables qui sont comparés par valeur plutôt que par l'identifiant de l'objet. La raison de cette limitation est qu'une implémentation efficace de dictionnaire requiert que l'empreinte par hachage des clés reste constante dans le temps. Les types numériques obéissent aux règles normales pour les comparaisons numériques : si deux nombres sont égaux pour l'opération de comparaison, par exemple `1` et `1.0`, alors ces deux nombres peuvent être utilisés indifféremment pour désigner la même entrée du dictionnaire.

Les dictionnaires sont muables : ils peuvent être créés par la notation `{...}` (reportez-vous à la section *Agencements de dictionnaires*).

The extension modules `dbm`, `gdbm`, and `bsddb` provide additional examples of mapping types.

Types appelables Ce sont les types sur lesquels on peut faire un appel de fonction (lisez la section *Appels*) :

Fonctions allogènes Un objet fonction allogène (ou fonction définie par l'utilisateur, mais ce n'est pas forcément l'utilisateur courant qui a défini cette fonction) est créé par la définition d'une fonction (voir la section *Définition de fonctions*). Il doit être appelé avec une liste d'arguments contenant le même nombre d'éléments que la liste des paramètres formels de la fonction.

Attributs spéciaux :

Attribut	Signification	
`__doc__` `func_doc`	The function's documentation string, or `None` if unavailable.	Accessible en écriture
`__name__` `func_name`	Nom de la fonction	Accessible en écriture
`__module__`	Nom du module où la fonction est définie ou `None` si ce nom n'est pas disponible.	Accessible en écriture
`__defaults__` `func_defaults`	Tuple contenant les valeurs des arguments par défaut pour ceux qui en sont dotés ou `None` si aucun argument n'a de valeur par défaut.	Accessible en écriture
`__code__` `func_code`	Objet code représentant le corps de la fonction compilée.	Accessible en écriture
`__globals__` `func_globals`	Référence pointant vers le dictionnaire contenant les variables globales de la fonction – l'espace de noms global du module dans lequel la fonction est définie.	Accessible en lecture seule
`__dict__` `func_dict`	Espace de noms accueillant les attributs de la fonction.	Accessible en écriture
`__closure__` `func_closure`	`None` ou tuple de cellules qui contient un lien pour chaque variable libre de la fonction.	Accessible en lecture seule

La plupart des attributs étiquetés « Accessible en écriture » vérifient le type de la valeur qu'on leur assigne.

Modifié dans la version 2.4 : `func_name` is now writable.

Modifié dans la version 2.6 : The double-underscore attributes `__closure__`, `__code__`, `__defaults__`, and `__globals__` were introduced as aliases for the corresponding `func_*` attributes for forwards compatibility with Python 3.

Les objets fonctions acceptent également l'assignation et la lecture d'attributs arbitraires. Vous pouvez utiliser cette fonctionnalité pour, par exemple, associer des métadonnées aux fonctions. La notation classique par point est utilisée pour définir et lire de tels attributs. *Notez que l'implémentation actuelle accepte seulement les attributs de fonction sur les fonctions définies par l'utilisateur. Les attributs de fonction pour les fonctions natives seront peut-être acceptés dans le futur.*

Vous trouvez davantage d'informations sur la définition de fonctions dans le code de cet objet ; la description des types internes est donnée plus bas.

User-defined methods A user-defined method object combines a class, a class instance (or `None`) and any callable object (normally a user-defined function).

Special read-only attributes : `im_self` is the class instance object, `im_func` is the function object ; `im_class` is the class of `im_self` for bound methods or the class that asked for the method for unbound methods ; `__doc__` is the method's documentation (same as `im_func.__doc__`) ; `__name__` is the method name (same as `im_func.__name__`) ; `__module__` is the name of the module the method was defined in, or `None` if unavailable.

Modifié dans la version 2.2 : `im_self` used to refer to the class that defined the method.

Modifié dans la version 2.6 : For Python 3 forward-compatibility, `im_func` is also available as `__func__`, and `im_self` as `__self__`.

Les méthodes savent aussi accéder (mais pas modifier) les attributs de la fonction de l'objet fonction sous-jacent.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object. When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object ; otherwise, the original method object is used as it is.

When a user-defined method object is created by retrieving a user-defined function object from a class, its `im_self` attribute is `None` and the method object is said to be unbound. When one is created by retrieving a user-defined function object from a class via one of its instances, its `im_self` attribute is the instance, and the method object is said to be bound. In either case, the new method's `im_class` attribute is the class from which the retrieval takes place, and its `im_func` attribute is the original function object.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the `im_func` attribute of the new instance is not the original method object but its `im_func` attribute.

When a user-defined method object is created by retrieving a class method object from a class or instance, its `im_self` attribute is the class itself, and its `im_func` attribute is the function object underlying the class method.

When an unbound user-defined method object is called, the underlying function (`im_func`) is called, with the restriction that the first argument must be an instance of the proper class (`im_class`) or of a derived class thereof.

When a bound user-defined method object is called, the underlying function (`im_func`) is called, inserting the class instance (`im_self`) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When a user-defined method object is derived from a class method object, the « class instance » stored in `im_self` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions ; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods ; this *only* happens when the function is an attribute of the class.

Fonctions générateurs A function or method which uses the *yield* statement (see section *L'instruction yield*) is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function : calling the iterator's `next()` method will cause the function to execute until it provides a value using the *yield* statement. When the function executes a *return* statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

Fonctions natives Un objet fonction native est une enveloppe autour d'une fonction C. Nous pouvons citer `len()` et `math.sin()` (`math` est un module standard natif) comme fonctions natives. Le nombre et le type des arguments sont déterminés par la fonction C. Des attributs spéciaux en lecture seule existent : `__doc__` contient la chaîne de documentation de la fonction (ou `None` s'il n'y en a pas) ; `__name__` est le nom de la fonction ; `__self__` est défini à `None` ; `__module__` est le nom du module où la fonction est définie ou `None` s'il n'est pas disponible.

Méthodes natives Ce sont des fonctions natives déguisées, contenant un objet passé à une fonction C en tant qu'argument supplémentaire implicite. Un exemple de méthode native est `une_liste.append()` (une_liste étant un objet liste). Dans ce cas, l'attribut spécial en lecture seule `__self__` est défini à l'objet *une_liste*.

Class Types Class types, or « new-style classes, » are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override *__new__()*. The arguments of the call are passed to *__new__()* and, in the typical case, to *__init__()* to initialize the new instance.

Classic Classes Class objects are described below. When a class object is called, a new class instance (also described below) is created and returned. This implies a call to the class's *__init__()* method if it has one. Any arguments are passed on to the *__init__()* method. If there is no *__init__()* method, the class must be called without arguments.

Instances de classes Class instances are described below. Class instances are callable only when the class has a *__call__()* method; `x(arguments)` is a shorthand for `x.__call__(arguments)`.

Modules Modules are imported by the *import* statement (see section *L'instruction import*). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the func_globals attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

L'assignation d'un attribut met à jour le dictionnaire d'espace de noms du module, par exemple `m.x = 1` est équivalent à `m.__dict__["x"] = 1`.

Attribut spécial en lecture seule : `__dict__` est l'objet dictionnaire répertoriant l'espace de noms du module.

en raison de la manière dont CPython nettoie les dictionnaires de modules, le dictionnaire du module est effacé quand le module n'est plus visible, même si le dictionnaire possède encore des références actives. Pour éviter ceci, copiez le dictionnaire ou gardez le module dans votre champ de visibilité tant que vous souhaitez utiliser le dictionnaire directement.

Predefined (writable) attributes : `__name__` is the module's name; `__doc__` is the module's documentation string, or `None` if unavailable; `__file__` is the pathname of the file from which the module was loaded, if it was loaded from a file. The `__file__` attribute is not present for C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

Classes Both class types (new-style classes) and class objects (old-style/classic classes) are typically created by class definitions (see section *Définition de classes*). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `C.x` is translated to `C.__dict__["x"]` (although for new-style classes in particular there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. For old-style classes, the search is depth-first, left-to-right in the order of occurrence in the base class list. New-style classes use the more complex C3 method resolution order which behaves correctly even in the presence of "diamond" inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by new-style classes can be found in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/2.3/mro/.

When a class attribute reference (for class `C`, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either `C` or one of its base classes, it is transformed into an unbound user-defined method object whose `im_class` attribute is `C`. When it would yield a class method object, it is transformed into a bound user-defined method object whose `im_self` attribute is `C`. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section *Implémentation de descripteurs* for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__` (note that only new-style classes support descriptors).

Les assignations d'un attribut de classe mettent à jour le dictionnaire de la classe, jamais le dictionnaire d'une classe de base.

Un objet classe peut être appelé (voir ci-dessus) pour produire une instance de classe (voir ci-dessous).

Special attributes : `__name__` is the class name; `__module__` is the module name in which the class was defined; `__dict__` is the dictionary containing the class's namespace; `__bases__` is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; `__doc__` is the class's documentation string, or `None` if undefined.

Instances de classes A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object or an unbound user-defined method object whose associated class is the class (call it `C`) of the instance for which the attribute reference was initiated

or one of its bases, it is transformed into a bound user-defined method object whose `im_class` attribute is `C` and whose `im_self` attribute is the instance. Static method and class method objects are also transformed, as if they had been retrieved from class `C`; see above under « Classes ». See section *Implémentation de descripteurs* for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's `__dict__`. If no class attribute is found, and the object's class has a *`__getattr__()`* method, that is called to satisfy the lookup.

Les assignations et suppressions d'attributs mettent à jour le dictionnaire de l'instance, jamais le dictionnaire de la classe. Si la classe possède une méthode *`__setattr__()`* ou *`__delattr__()`*, elle est appelée au lieu de mettre à jour le dictionnaire de l'instance directement.

Les instances de classes peuvent prétendre être des nombres, des séquences ou des tableaux de correspondance si elles ont des méthodes avec des noms spéciaux. Voir la section *Méthodes spéciales*.

Attributs spéciaux : `__dict__` est le dictionnaire des attributs; `__class__` est la classe de l'instance.

Files A file object represents an open file. File objects are created by the `open()` built-in function, and also by `os.popen()`, `os.fdopen()`, and the `makefile()` method of socket objects (and perhaps by other functions or methods provided by extension modules). The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter's standard input, output and error streams. See bltin-file-objects for complete documentation of file objects.

Types internes Quelques types utilisés en interne par l'interpréteur sont accessibles à l'utilisateur. Leur définition peut changer dans les futures versions de l'interpréteur mais ils sont donnés ci-dessous à fin d'exhaustivité.

Objets Code Un objet code représente le code Python sous sa forme compilée en *bytecode*. La différence entre un objet code et un objet fonction est que l'objet fonction contient une référence explicite vers les globales de la fonction (le module dans lequel elle est définie) alors qu'un objet code ne contient aucun contexte; par ailleurs, les valeurs par défaut des arguments sont stockées dans l'objet fonction, pas dans l'objet code (parce que ce sont des valeurs calculées au moment de l'exécution). Contrairement aux objets fonctions, les objets codes sont immuables et ne contiennent aucune référence (directe ou indirecte) à des objets muables.

Attributs spéciaux en lecture seule : `co_name` donne le nom de la fonction; `co_argcount` est le nombre d'arguments positionnels (y compris les arguments avec des valeurs par défaut); `co_nlocals` est le nombre de variables locales utilisées par la fonction (y compris les arguments); `co_varnames` est un tuple contenant le nom des variables locales (en commençant par les noms des arguments); `co_cellvars` est un tuple contenant les noms des variables locales qui sont référencées par des fonctions imbriquées; `co_freevars` est un tuple contenant les noms des variables libres; `co_code` est une chaîne représentant la séquence des instructions de *bytecode*; `co_consts` est un tuple contenant les littéraux utilisés par le *bytecode*; `co_names` est un tuple contenant les noms utilisés par le *bytecode*; `co_filename` est le nom de fichier à partir duquel le code a été compilé; `co_firstlineno` est numéro de la première ligne de la fonction; `co_lnotab` est une chaîne qui code la correspondance entre les différents endroits du *bytecode* et les numéros de lignes (pour les détails, regardez le code source de l'interpréteur); `co_stacksize` est la taille de pile nécessaire (y compris pour les variables locales); `co_flags` est un entier qui code différents drapeaux pour l'interpréteur.

Les drapeaux suivants sont codés par des bits dans `co_flags` : le bit `0x04` est positionné à 1 si la fonction utilise la syntaxe `*arguments` pour accepter un nombre arbitraire d'arguments positionnels; le bit `0x08` est positionné à 1 si la fonction utilise la syntaxe `**keywords` pour accepter un nombre arbitraire d'arguments nommés; le bit `0x20` est positionné à 1 si la fonction est un générateur.

Les déclarations de fonctionnalité future `from __future__ import division` utilisent aussi des bits dans `co_flags` pour indiquer si l'objet code a été compilé avec une fonctionnalité future : le bit `0x2000` est positionné à 1 si la fonction a été compilée avec la division future activée; les bits `0x10` et `0x1000` étaient utilisés dans les versions antérieures de Python.

Les autres bits de `co_flags` sont réservés à un usage interne.

Si l'objet code représente une fonction, le premier élément dans `co_consts` est la chaîne de documentation de la fonction (ou `None` s'il n'y en a pas).

Objets cadres Un objet cadre représente le cadre d'exécution. Ils apparaissent dans des objets traces (voir plus loin).

Special read-only attributes : `f_back` is to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame ; `f_code` is the code object being executed in this frame ; `f_locals` is the dictionary used to look up local variables ; `f_globals` is used for global variables ; `f_builtins` is used for built-in (intrinsic) names ; `f_restricted` is a flag indicating whether the function is executing in restricted execution mode ; `f_lasti` gives the precise instruction (this is an index into the bytecode string of the code object).

Special writable attributes : `f_trace`, if not `None`, is a function called at the start of each source code line (this is used by the debugger) ; `f_exc_type`, `f_exc_value`, `f_exc_traceback` represent the last exception raised in the parent frame provided another exception was ever raised in the current frame (in all other cases they are `None`) ; `f_lineno` is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to f_lineno.

Objets traces Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *L'instruction try*.) It is accessible as `sys.exc_traceback`, and also as the third item of the tuple returned by `sys.exc_info()`. The latter is the preferred interface, since it works correctly when the program is using multiple threads. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream ; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

Attributs spéciaux en lecture seule : `tb_next` est le prochain niveau dans la pile d'appels (vers le cadre où l'exception a été levée) ou `None` s'il n'y a pas de prochain niveau ; `tb_frame` pointe vers le cadre d'exécution du niveau courant ; `tb_lineno` donne le numéro de ligne où l'exception a été levée ; `tb_lasti` indique l'instruction précise. Le numéro de ligne et la dernière instruction dans la trace peuvent différer du numéro de ligne de l'objet cadre si l'exception a eu lieu dans une instruction *`try`* sans qu'il n'y ait de clause `except` adéquate ou sans clause `finally`.

Objets tranches Slice objects are used to represent slices when *extended slice syntax* is used. This is a slice using two colons, or multiple slices or ellipses separated by commas, e.g., `a[i:j:step]`, `a[i:j, k:l]`, or `a[..., i:j]`. They are also created by the built-in `slice()` function.

Attributs spéciaux en lecture seule : `start` est la borne inférieure ; `stop` est la borne supérieure ; `step` est la valeur du pas ; chaque attribut vaut `None` s'il est omis. Ces attributs peuvent être de n'importe quel type.

Les objets tranches comprennent une méthode :

`slice.`**`indices`**(*self*, *length*)

This method takes a single integer argument *length* and computes information about the extended slice that the slice object would describe if applied to a sequence of *length* items. It returns a tuple of three integers ; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

Nouveau dans la version 2.3.

Objets méthodes statiques Les objets méthodes statiques permettent la transformation des objets fonctions en objets méthodes décrits au-dessus. Un objet méthode statique encapsule tout autre objet, souvent un objet méthode définie par l'utilisateur. Quand un objet méthode statique est récupéré depuis une classe ou une instance de classe, l'objet réellement renvoyé est un objet encapsulé, qui n'a pas vocation à être transformé encore une fois. Les objets méthodes statiques ne sont pas appelables en tant que tels, bien que les objets qu'ils encapsulent le soient souvent. Les objets méthodes statiques sont créés par le constructeur natif `staticmethod()`.

Objets méthodes de classes Un objet méthode de classe, comme un objet méthode statique, encapsule un autre objet afin de modifier la façon dont cet objet est récupéré depuis les classes et instances de classes. Le comportement des objets méthodes de classes dans le cas d'une telle récupération est décrit plus haut, dans « méthodes définies par l'utilisateur ». Les objets méthodes de classes sont créés par le constructeur natif `classmethod()`.

3.3 New-style and classic classes

Classes and instances come in two flavors : old-style (or classic) and new-style.

Up to Python 2.1 the concept of `class` was unrelated to the concept of `type`, and old-style classes were the only flavor available. For an old-style class, the statement `x.__class__` provides the class of *x*, but `type(x)` is always `<type 'instance'>`. This reflects the fact that all old-style instances, independent of their class, are implemented with a single built-in type, called `instance`.

New-style classes were introduced in Python 2.2 to unify the concepts of `class` and `type`. A new-style class is simply a user-defined type, no more, no less. If *x* is an instance of a new-style class, then `type(x)` is typically the same as `x.__class__` (although this is not guaranteed – a new-style class instance is permitted to override the value returned for `x.__class__`).

The major motivation for introducing new-style classes is to provide a unified object model with a full meta-model. It also has a number of practical benefits, like the ability to subclass most built-in types, or the introduction of « descriptors », which enable computed properties.

For compatibility reasons, classes are still old-style by default. New-style classes are created by specifying another new-style class (i.e. a type) as a parent class, or the « top-level type » `object` if no other parent is needed. The behaviour of new-style classes differs from that of old-style classes in a number of important details in addition to what `type()` returns. Some of these changes are fundamental to the new object model, like the way special methods are invoked. Others are « fixes » that could not be implemented before for compatibility concerns, like the method resolution order in case of multiple inheritance.

While this manual aims to provide comprehensive coverage of Python's class mechanics, it may still be lacking in some areas when it comes to its coverage of new-style classes. Please see https://www.python.org/doc/newstyle/ for sources of additional information.

Old-style classes are removed in Python 3, leaving only new-style classes.

3.4 Méthodes spéciales

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named *`__getitem__()`*, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `x.__getitem__(i)` for old-style classes and `type(x).__getitem__(x, i)` for new-style classes. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

Lorsque vous implémentez une classe qui émule un type natif, il est important que cette émulation n'implémente que ce qui fait sens pour l'objet qui est modélisé. Par exemple, la recherche d'éléments individuels d'une séquence peut faire sens, mais pas l'extraction d'une tranche (un exemple est l'interface de `NodeList` dans le modèle objet des documents W3C).

3.4.1 Personnalisation de base

`object.`**`__new__`**`(`*cls*`[, ...])`

Appelée pour créer une nouvelle instance de la classe *cls*. La méthode *`__new__()`* est statique (c'est un cas particulier, vous n'avez pas besoin de la déclarer comme telle) qui prend comme premier argument la classe pour laquelle on veut créer une instance. Les autres arguments sont ceux passés à l'expression de l'objet constructeur (l'appel à la classe). La valeur de retour de *`__new__()`* doit être l'instance du nouvel objet (classiquement une instance de *cls*).

Typical implementations create a new instance of the class by invoking the superclass's *`__new__()`* method using `super(currentclass, cls).__new__(cls[, ...])` with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

Si *`__new__()`* renvoie une instance de *cls*, alors la méthode *`__init__()`* de la nouvelle instance est invoquée avec `__init__(self[, ...])` où *self* est la nouvelle instance et les autres arguments sont les mêmes que ceux passés à *`__new__()`*.

Si *`__new__()`* ne renvoie pas une instance de *cls*, alors la méthode *`__init__()`* de la nouvelle instance n'est pas invoquée.

L'objectif de *`__new__()`* est principalement, pour les sous-classes de types immuables (comme `int`, `str` ou `tuple`), d'autoriser la création sur mesure des instances. Elle est aussi souvent surchargée dans les méta-classes pour particulariser la création des classes.

`object.`**`__init__`**`(`*self*`[, ...])`

Called after the instance has been created (by *`__new__()`*), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an *`__init__()`* method, the derived class's *`__init__()`* method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example : `BaseClass.__init__(self, [args...])`.

Because *`__new__()`* and *`__init__()`* work together in constructing objects (*`__new__()`* to create it, and *`__init__()`* to customise it), no non-`None` value may be returned by *`__init__()`*; doing so will cause a `TypeError` to be raised at runtime.

`object.`**`__del__`**`(`*self*`)`

Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a *`__del__()`* method, the derived class's *`__del__()`* method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended !) for the *`__del__()`* method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that *`__del__()`* methods are called for objects that still exist when the interpreter exits.

Note : `del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include : circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in `sys.exc_traceback` keeps the stack frame alive); or a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in `sys.last_traceback` keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the latter two situations can be resolved by storing `None` in `sys.exc_traceback` or `sys.last_traceback`. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level *`__del__()`* methods involved. Refer to the documentation for the `gc` module for more information about how *`__del__()`* methods are handled by the cycle detector, particularly the description of the `garbage` value.

> **Avertissement :** Due to the precarious circumstances under which *`__del__()`* methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. Also,

when *__del__()* is invoked in response to a module being deleted (e.g., when execution of the program is done), other globals referenced by the *__del__()* method may already have been deleted or in the process of being torn down (e.g. the import machinery shutting down). For this reason, *__del__()* methods should do the absolute minimum needed to maintain external invariants. Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted ; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the *__del__()* method is called.

See also the `-R` command-line option.

`object.`**`__repr__`**`(`*self*`)`
Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the « official » string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object. If a class defines *__repr__()* but not *__str__()*, then *__repr__()* is also used when an « informal » string representation of instances of that class is required.

Cette fonction est principalement utilisée à fins de débogage, il est donc important que la représentation donne beaucoup d'informations et ne soit pas ambigüe.

`object.`**`__str__`**`(`*self*`)`
Called by the `str()` built-in function and by the *print* statement to compute the « informal » string representation of an object. This differs from *__repr__()* in that it does not have to be a valid Python expression : a more convenient or concise representation may be used instead. The return value must be a string object.

`object.`**`__lt__`**`(`*self*, *other*`)`
`object.`**`__le__`**`(`*self*, *other*`)`
`object.`**`__eq__`**`(`*self*, *other*`)`
`object.`**`__ne__`**`(`*self*, *other*`)`
`object.`**`__gt__`**`(`*self*, *other*`)`
`object.`**`__ge__`**`(`*self*, *other*`)`
Nouveau dans la version 2.1.

These are the so-called « rich comparison » methods, and are called for comparison operators in preference to *__cmp__()* below. The correspondence between operator symbols and method names is as follows : `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` and `x<>y` call `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

Une méthode de comparaison riche peut renvoyer le singleton `NotImplemented` si elle n'implémente pas l'opération pour une paire donnée d'arguments. Par convention, `False` et `True` sont renvoyées pour une comparaison qui a réussi. Cependant, ces méthodes peuvent renvoyer n'importe quelle valeur donc, si l'opérateur de comparaison est utilisé dans un contexte booléen (par exemple dans une condition d'une instruction `if`), Python appelle `bool()` sur la valeur pour déterminer si le résultat est faux ou vrai.

There are no implied relationships among the comparison operators. The truth of `x==y` does not imply that `x!=y` is false. Accordingly, when defining *__eq__()*, one should also define *__ne__()* so that the operators will behave as expected. See the paragraph on *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does) ; rather, *__lt__()* and *__gt__()* are each other's reflection, *__le__()* and *__ge__()* are each other's reflection, and *__eq__()* and *__ne__()* are their own reflection.

Arguments to rich comparison methods are never coerced.

To automatically generate ordering operations from a single root operation, see `functools.total_ordering()`.

`object.`**`__cmp__`**`(`*self*, *other*`)`
Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer

if `self < other`, zero if `self == other`, a positive integer if `self > other`. If no *__cmp__()*, *__eq__()* or *__ne__()* operation is defined, class instances are compared by object identity (« address »). See also the description of *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys. (Note : the restriction that exceptions are not propagated by *__cmp__()* has been removed since Python 1.5.)

`object.`**`__rcmp__`**(*self*, *other*)
Modifié dans la version 2.1 : No longer supported.

`object.`**`__hash__`**(*self*)
Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. *__hash__()* should return an integer. The only required property is that objects which compare equal have the same hash value ; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example :

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

If a class does not define a *__cmp__()* or *__eq__()* method it should not define a *__hash__()* operation either ; if it defines *__cmp__()* or *__eq__()* but not *__hash__()*, its instances will not be usable in hashed collections. If a class defines mutable objects and implements a *__cmp__()* or *__eq__()* method, it should not implement *__hash__()*, since hashable collection implementations require that an object's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have *__cmp__()* and *__hash__()* methods by default ; with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns a result derived from `id(x)`.

Classes which inherit a *__hash__()* method from a parent class but change the meaning of *__cmp__()* or *__eq__()* such that the hash value returned is no longer appropriate (e.g. by switching to a value-based concept of equality instead of the default identity based equality) can explicitly flag themselves as being unhashable by setting `__hash__ = None` in the class definition. Doing so means that not only will instances of the class raise an appropriate `TypeError` when a program attempts to retrieve their hash value, but they will also be correctly identified as unhashable when checking `isinstance(obj, collections.Hashable)` (unlike classes which define their own *__hash__()* to explicitly raise `TypeError`).

Modifié dans la version 2.5 : *__hash__()* may now also return a long integer object ; the 32-bit integer is then derived from the hash of that object.

Modifié dans la version 2.6 : *__hash__* may now be set to `None` to explicitly flag instances of a class as unhashable.

`object.`**`__nonzero__`**(*self*)
Called to implement truth value testing and the built-in operation `bool()` ; should return `False` or `True`, or their integer equivalents `0` or `1`. When this method is not defined, *__len__()* is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither *__len__()* nor *__nonzero__()*, all its instances are considered true.

`object.`**`__unicode__`**(*self*)
Called to implement `unicode()` built-in ; should return a Unicode object. When this method is not defined, string conversion is attempted, and the result of string conversion is converted to Unicode using the system default encoding.

3.4.2 Personnalisation de l'accès aux attributs

Les méthodes suivantes peuvent être définies pour personnaliser l'accès aux attributs (utilisation, assignation, suppression de `x.name`) pour les instances de classes.

`object.`**`__getattr__`**(*self*, *name*)
Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for `self`). `name` is the attribute name. This method should return the (computed) attribute value or raise an `AttributeError` exception.

Note that if the attribute is found through the normal mechanism, *__getattr__()* is not called. (This is an intentional asymmetry between *__getattr__()* and *__setattr__()*.) This is done both for efficiency reasons and because otherwise *__getattr__()* would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the *__getattribute__()* method below for a way to actually get total control in new-style classes.

`object.`**`__setattr__`**(*self*, *name*, *value*)
Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

If *__setattr__()* wants to assign to an instance attribute, it should not simply execute `self.name = value` — this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., `self.__dict__[name] = value`. For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, `object.__setattr__(self, name, value)`.

`object.`**`__delattr__`**(*self*, *name*)
Comme *__setattr__()* mais pour supprimer un attribut au lieu de l'assigner. Elle ne doit être implémentée que si `del obj.name` a du sens pour cet objet.

More attribute access for new-style classes

The following methods only apply to new-style classes.

`object.`**`__getattribute__`**(*self*, *name*)
Appelée de manière inconditionnelle pour implémenter l'accès aux attributs des instances de la classe. Si la classe définit également *__getattr__()*, cette dernière n'est pas appelée à moins que *__getattribute__()* ne l'appelle explicitement ou ne lève une exception `AttributeError`. Cette méthode doit renvoyer la valeur (calculée) de l'attribut ou lever une exception `AttributeError`. Afin d'éviter une récursion infinie sur cette méthode, son implémentation doit toujours appeler la méthode de la classe de base avec le même paramètre *name* pour accéder à n'importe quel attribut dont elle a besoin. Par exemple, `object.__getattribute__(self, name)`.

Note : This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See *Special method lookup for new-style classes*.

Implémentation de descripteurs

Les méthodes qui suivent s'appliquent seulement quand une instance de la classe (dite classe *descripteur*) contenant la méthode apparaît dans une classe *propriétaire* (*owner* en anglais) ; la classe descripteur doit figurer dans le dictionnaire de la classe propriétaire ou dans le dictionnaire de la classe d'un des parents. Dans les exemples ci-dessous, « l'attribut » fait référence à l'attribut dont le nom est une clé du `__dict__` de la classe propriétaire.

`object.`**`__get__`**`(`*self*, *instance*, *owner*`)`
: Appelée pour obtenir l'attribut de la classe propriétaire (accès à un attribut de classe) ou d'une instance de cette classe (accès à un attribut d'instance). *owner* est toujours la classe propriétaire alors que *instance* est l'instance par laquelle on accède à l'attribut ou `None` lorsque l'on accède par la classe *owner*. Cette méthode doit renvoyer la valeur (calculée) de l'attribut ou lever une exception `AttributeError`.

`object.`**`__set__`**`(`*self*, *instance*, *value*`)`
: Appelée pour définir l'attribut d'une instance *instance* de la classe propriétaire à la nouvelle valeur *value*.

`object.`**`__delete__`**`(`*self*, *instance*`)`
: Appelée pour supprimer l'attribut de l'instance *instance* de la classe propriétaire.

Invocation des descripteurs

En général, un descripteur est un attribut d'objet dont le comportement est « lié » (*binding dehavior* en anglais), c'est-à-dire que les accès aux attributs ont été surchargés par des méthodes conformes au protocole des descripteurs : *`__get__()`*, *`__set__()`* et *`__delete__()`*. Si l'une de ces méthodes est définie pour un objet, il est réputé être un descripteur.

Le comportement par défaut pour la gestion d'un attribut est de définir, obtenir et supprimer cet attribut du dictionnaire de l'objet. Par exemple, pour `a.x` Python commence d'abord par rechercher `a.__dict__['x']`, puis `type(a).__dict__['x']` ; ensuite Python continue en remontant les classes de base de `type(a)`, en excluant les méta-classes.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called. Note that descriptors are only invoked for new style objects or classes (ones that subclass `object()` or `type()`).

Le point de départ pour une invocation de descripteur est la liaison `a.x`. La façon dont les arguments sont assemblés dépend de `a` :

Appel direct Le plus simple et le plus rare des appels est quand l'utilisateur code directement l'appel à la méthode du descripteur : `x.__get__(a)`.

Liaison avec une instance If binding to a new-style object instance, `a.x` is transformed into the call : `type(a).__dict__['x'].__get__(a, type(a))`.

Liaison avec une classe If binding to a new-style class, `A.x` is transformed into the call : `A.__dict__['x'].__get__(None, A)`.

Liaison super Si `a` est une instance de `super`, alors `super(B, obj).m()` recherche `obj.__class__.__mro__` pour la classe de base `A` immédiatement avant `B` puis invoque le descripteur avec l'appel suivant : `A.__dict__['m'].__get__(obj, obj.__class__)`.

Pour des liaisons avec des instances, la priorité à l'invocation du descripteur dépend des méthodes que le descripteur a définies. Un descripteur peut définir n'importe quelle combinaison de *`__get__()`*, *`__set__()`* et *`__delete__()`*. S'il ne définit pas *`__get__()`*, alors accéder à l'attribut retourne l'objet descripteur lui-même sauf s'il existe une valeur dans le dictionnaire de l'objet instance. Si le descripteur définit *`__set__()`* ou *`__delete__()`*, c'est un descripteur de données ; s'il ne définit aucune méthode, c'est un descripteur hors-donnée. Normalement, les descripteurs de données définissent à la fois *`__get__()`* et *`__set__()`*, alors que les descripteurs hors-données définissent seulement la méthode *`__get__()`*. Les descripteurs de données qui définissent *`__set__()`* et *`__get__()`* sont toujours prioritaires face à une redéfinition du dictionnaire de l'instance. En revanche, les descripteurs hors-données peuvent être shuntés par les instances.

Les méthodes Python (y compris `staticmethod()` et `classmethod()`) sont implémentées comme des descripteurs hors-donnée. De la même manière, les instances peuvent redéfinir et surcharger les méthodes. Ceci permet à chaque instance d'avoir un comportement qui diffère des autres instances de la même classe.

La fonction `property()` est implémentée en tant que descripteur de données. Ainsi, les instances ne peuvent pas surcharger le comportement d'une propriété.

`__slots__`

By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining *__slots__* in a new-style class definition. The *__slots__* declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because *__dict__* is not created for each instance.

`__slots__`

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. If defined in a new-style class, *__slots__* reserves space for the declared variables and prevents the automatic creation of *__dict__* and *__weakref__* for each instance.

Nouveau dans la version 2.2.

Note sur l'utilisation de *__slots__*

- When inheriting from a class without *__slots__*, the *__dict__* attribute of that class will always be accessible, so a *__slots__* definition in the subclass is meaningless.
- Sans variable *__dict__*, les instances ne peuvent pas assigner de nouvelles variables (non listées dans la définition de *__slots__*). Les tentatives d'assignation sur un nom de variable non listé lève `AttributeError`. Si l'assignation dynamique de nouvelles variables est nécessaire, ajoutez `'__dict__'` à la séquence de chaînes dans la déclaration *__slots__*.
  Modifié dans la version 2.3 : Previously, adding `'__dict__'` to the *__slots__* declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.
- Sans variable *__weakref__* pour chaque instance, les classes qui définissent *__slots__* ne gèrent pas les références faibles vers leurs instances. Si vous avez besoin de gérer des références faibles, ajoutez `'__weakref__'` à la séquence de chaînes dans la déclaration de *__slots__*.
  Modifié dans la version 2.3 : Previously, adding `'__weakref__'` to the *__slots__* declaration would not enable support for weak references.
- Les *__slots__* sont implémentés au niveau de la classe en créant des descripteurs (*Implémentation de descripteurs*) pour chaque nom de variable. Ainsi, les attributs de classe ne peuvent pas être utilisés pour des valeurs par défaut aux variables d'instances définies par *__slots__* ; sinon, l'attribut de classe surchargerait l'assignation par descripteur.
- The action of a *__slots__* declaration is limited to the class where it is defined. As a result, subclasses will have a *__dict__* unless they also define *__slots__* (which must only contain names of any *additional* slots).
- Si une classe définit un *slot* déjà défini dans une classe de base, la variable d'instance définie par la classe de base est inaccessible (sauf à utiliser le descripteur de la classe de base directement). Cela rend la signification du programme indéfinie. Dans le futur, une vérification sera ajoutée pour empêcher cela.
- Nonempty *__slots__* does not work for classes derived from « variable-length » built-in types such as `long`, `str` and `tuple`.
- Tout itérable qui n'est pas une chaîne peut être assigné à un *__slots__*. Les tableaux de correspondance peuvent aussi être utilisés ; cependant, dans le futur, des significations spéciales pourraient être associées à chacune des clés.
- Les assignations de *__class__* ne fonctionnent que si les deux classes ont le même *__slots__*.
  Modifié dans la version 2.6 : Previously, *__class__* assignment raised an error if either new or old class had *__slots__*.

3.4.3 Personnalisation de la création de classes

By default, new-style classes are constructed using `type()`. A class definition is read into a separate namespace and the value of class name is bound to the result of `type(name, bases, dict)`.

When the class definition is read, if *__metaclass__* is defined then the callable assigned to it will be called instead of `type()`. This allows classes or functions to be written which monitor or alter the class creation process :

— Modifying the class dictionary prior to the class being created.

— Returning an instance of another class – essentially performing the role of a factory function.

These steps will have to be performed in the metaclass's *__new__()* method – `type.__new__()` can then be called from this method to create a class with different properties. This example adds a new element to the class dictionary before creating the class :

```
class metacls(type):
    def __new__(mcs, name, bases, dict):
        dict['foo'] = 'metacls was here'
        return type.__new__(mcs, name, bases, dict)
```

You can of course also override other class methods (or add new methods) ; for example defining a custom *__call__()* method in the metaclass allows custom behavior when the class is called, e.g. not always creating a new instance.

__metaclass__

This variable can be any callable accepting arguments for `name`, `bases`, and `dict`. Upon class creation, the callable is used instead of the built-in `type()`.

Nouveau dans la version 2.2.

The appropriate metaclass is determined by the following precedence rules :

— If `dict['__metaclass__']` exists, it is used.

— Otherwise, if there is at least one base class, its metaclass is used (this looks for a *__class__* attribute first and if not found, uses its type).

— Otherwise, if a global variable named __metaclass__ exists, it is used.

— Otherwise, the old-style, classic metaclass (types.ClassType) is used.

The potential uses for metaclasses are boundless. Some ideas that have been explored including logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.4.4 Personnalisation des instances et vérification des sous-classes

Nouveau dans la version 2.6.

Les méthodes suivantes sont utilisées pour surcharger le comportement par défaut des fonctions natives `isinstance()` et `issubclass()`.

En particulier, la méta-classe `abc.ABCMeta` implémente ces méthodes pour autoriser l'ajout de classes de base abstraites (ABC pour *Abstract Base Classes* en anglais) en tant que « classes de base virtuelles » pour toute classe ou type (y compris les types natifs).

`class.`**__instancecheck__**(*self*, *instance*)

Renvoie `True` si *instance* doit être considérée comme une instance (directe ou indirecte) de *class*. Si elle est définie, est elle appelée pour implémenter `isinstance(instance, class)`.

`class.`**__subclasscheck__**(*self*, *subclass*)

Renvoie `True` si *subclass* doit être considérée comme une sous-classe (directe ou indirecte) de *class*. Si elle est définie, appelée pour implémenter `issubclass(subclass, class)`.

Notez que ces méthodes sont recherchées dans le type (la méta-classe) d'une classe. Elles ne peuvent pas être définies en tant que méthodes de classe dans la classe réelle. C'est cohérent avec la recherche des méthodes spéciales qui sont appelées pour les instances, sauf qu'ici l'instance est elle-même une classe.

Voir aussi :

PEP 3119 – Introduction aux classes de bases abstraites Inclut la spécification pour la personnalisation du comportement de `isinstance()` et `issubclass()` à travers *`__instancecheck__()`* et *`__subclasscheck__()`*, avec comme motivation pour cette fonctionnalité l'ajout les classes de base abstraites (voir le module `abc`) au langage.

3.4.5 Émulation d'objets appelables

`object.`**`__call__`**(*self*[, *args...*])

Appelée quand l'instance est « appelée » en tant que fonction ; si la méthode est définie, `x(arg1, arg2, ...)` est un raccourci pour `x.__call__(arg1, arg2, ...)`.

3.4.6 Émulation de types conteneurs

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping ; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N` where *N* is the length of the sequence, or slice objects, which define a range of items. (For backwards compatibility, the method *`__getslice__()`* (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `setdefault()`, `iterkeys()`, `itervalues()`, `iteritems()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard dictionary objects. The `UserDict` module provides a `DictMixin` class to help create those methods from a base set of *`__getitem__()`*, *`__setitem__()`*, *`__delitem__()`*, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods *`__add__()`*, *`__radd__()`*, *`__iadd__()`*, *`__mul__()`*, *`__rmul__()`* and *`__imul__()`* described below ; they should not define *`__coerce__()`* or other numerical operators. It is recommended that both mappings and sequences implement the *`__contains__()`* method to allow efficient use of the `in` operator ; for mappings, `in` should be equivalent of `has_key()` ; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the *`__iter__()`* method to allow efficient iteration through the container ; for mappings, *`__iter__()`* should be the same as `iterkeys()` ; for sequences, it should iterate through the values.

`object.`**`__len__`**(*self*)

Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an object that doesn't define a *`__nonzero__()`* method and whose *`__len__()`* method returns zero is considered to be false in a Boolean context.

CPython implementation detail : In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a *`__nonzero__()`* method.

`object.`**`__getitem__`**(*self*, *key*)

Appelée pour implémenter l'évaluation de `self[key]`. Pour les types séquences, les clés autorisées sont les entiers et les objets tranches (*slice*). Notez que l'interprétation spéciale des indices négatifs (si la classe souhaite émuler un type séquence) est du ressort de la méthode *`__getitem__()`*. Si *key* n'est pas du bon type, une `TypeError` peut être levée ; si la valeur est en dehors de l'ensemble des indices de la séquence (après interprétation éventuelle des valeurs négatives), une `IndexError` doit être levée. Pour les tableaux de correspondances, si *key* n'existe pas dans le conteneur, une `KeyError` doit être levée.

Note : *for* s'attend à ce qu'une `IndexError` soit levée en cas d'indice illégal afin de détecter correctement la fin de la séquence.

`object.`**`__setitem__`**(*self*, *key*, *value*)
Appelée pour implémenter l'assignation à `self[key]`. La même note que pour *__getitem__()* s'applique. Elle ne doit être implémentée que pour les tableaux de correspondances qui autorisent les modifications de valeurs des clés, ceux pour lesquels on peut ajouter de nouvelles clés ou, pour les séquences, celles dont les éléments peuvent être remplacés. Les mêmes exceptions que pour la méthode *__getitem__()* doivent être levées en cas de mauvaises valeurs de clés.

`object.`**`__delitem__`**(*self*, *key*)
Appelée pour implémenter la suppression de `self[key]`. La même note que pour *__getitem__()* s'applique. Elle ne doit être implémentée que pour les tableaux de correspondances qui autorisent les suppressions de clés ou pour les séquences dont les éléments peuvent être supprimés de la séquence. Les mêmes exceptions que pour la méthode *__getitem__()* doivent être levées en cas de mauvaises valeurs de clés.

`object.`**`__missing__`**(*self*, *key*)
Appelée par `dict.`*__getitem__()* pour implémenter `self[key]` dans les sous-classes de dictionnaires lorsque la clé n'est pas dans le dictionnaire.

`object.`**`__iter__`**(*self*)
This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method `iterkeys()`.

Les objets itérateurs doivent aussi implémenter cette méthode ; ils doivent alors se renvoyer eux-mêmes. Pour plus d'information sur les objets itérateurs, lisez typeiter.

`object.`**`__reversed__`**(*self*)
Appelée (si elle existe) par la fonction native `reversed()` pour implémenter l'itération en sens inverse. Elle doit renvoyer un nouvel objet itérateur qui itère sur tous les objets du conteneur en sens inverse.

Si la méthode *__reversed__()* n'est pas fournie, la fonction native `reversed()` se replie sur le protocole de séquence (*__len__()* et *__getitem__()*). Les objets qui connaissent le protocole de séquence ne doivent fournir *__reversed__()* que si l'implémentation qu'ils proposent est plus efficace que celle de `reversed()`.

Nouveau dans la version 2.6.

Les opérateurs de tests d'appartenance (*in* et *not in*) sont normalement implémentés comme des itérations sur la séquence. Cependant, les objets conteneurs peuvent fournir les méthodes spéciales suivantes avec une implémentation plus efficace, qui ne requièrent d'ailleurs pas que l'objet soit une séquence.

`object.`**`__contains__`**(*self*, *item*)
Appelée pour implémenter les opérateurs de test d'appartenance. Elle doit renvoyer `True` si *item* est dans *self* et `False` sinon. Pour les tableaux de correspondances, seules les clés sont considérées (pas les valeurs des paires clés-valeurs).

Pour les objets qui ne définissent pas *__contains__()*, les tests d'appartenance essaient d'abord d'itérer avec *__iter__()* puis avec le vieux protocole d'itération sur les séquences *via* *__getitem__()*, reportez-vous à *cette section dans la référence du langage*.

3.4.7 Additional methods for emulation of sequence types

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define *__getslice__()* ; mutable sequences might define all three methods.

`object.`**`__getslice__`**(*self*, *i*, *j*)

Obsolète depuis la version 2.0 : Support slice objects as parameters to the *__getitem__()* method. (However, built-in types in CPython currently still implement *__getslice__()*. Therefore, you have to override it in derived classes when implementing slicing.)

Called to implement evaluation of `self[i:j]`. The returned object should be of the same type as *self*. Note that missing *i* or *j* in the slice expression are replaced by zero or `sys.maxsize`, respectively. If negative indexes are used in the slice, the length of the sequence is added to that index. If the instance does not implement the *__len__()* method, an `AttributeError` is raised. No guarantee is made that indexes adjusted this way are not still negative. Indexes which are greater than the length of the sequence are not modified. If no *__getslice__()* is found, a slice object is created instead, and passed to *__getitem__()* instead.

`object.`**`__setslice__`**(*self*, *i*, *j*, *sequence*)

Called to implement assignment to `self[i:j]`. Same notes for *i* and *j* as for *__getslice__()*.

This method is deprecated. If no *__setslice__()* is found, or for extended slicing of the form `self[i:j:k]`, a slice object is created, and passed to *__setitem__()*, instead of *__setslice__()* being called.

`object.`**`__delslice__`**(*self*, *i*, *j*)

Called to implement deletion of `self[i:j]`. Same notes for *i* and *j* as for *__getslice__()*. This method is deprecated. If no *__delslice__()* is found, or for extended slicing of the form `self[i:j:k]`, a slice object is created, and passed to *__delitem__()*, instead of *__delslice__()* being called.

Notice that these methods are only invoked when a single slice with a single colon is used, and the slice method is available. For slice operations involving extended slice notation, or in absence of the slice methods, *__getitem__()*, *__setitem__()* or *__delitem__()* is called with a slice object as argument.

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods *__getitem__()*, *__setitem__()* and *__delitem__()* support slice objects as arguments) :

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to `max()` ; these are necessary because of the handling of negative indices before the `__*slice__()` methods are called. When negative indexes are used, the `__*item__()` methods receive them as provided, but the `__*slice__()` methods get a « cooked » form of the index values. For each negative index value, the length of the

sequence is added to the index before calling the method (which may still result in a negative index) ; this is the customary handling of negative indexes by the built-in sequence types, and the `__*item__()` methods are expected to do this as well. However, since they should already be doing that, negative indexes cannot be passed in ; they must be constrained to the bounds of the sequence before being passed to the `__*item__()` methods. Calling `max(0, i)` conveniently returns the proper value.

3.4.8 Émulation de types numériques

Les méthodes suivantes peuvent être définies pour émuler des objets numériques. Les méthodes correspondant à des opérations qui ne sont pas autorisées pour la catégorie de nombres considérée (par exemple, les opérations bit à bit pour les nombres qui ne sont pas entiers) doivent être laissées indéfinies.

`object.`**`__add__`**(*self*, *other*)
`object.`**`__sub__`**(*self*, *other*)
`object.`**`__mul__`**(*self*, *other*)
`object.`**`__floordiv__`**(*self*, *other*)
`object.`**`__mod__`**(*self*, *other*)
`object.`**`__divmod__`**(*self*, *other*)
`object.`**`__pow__`**(*self*, *other*[, *modulo*])
`object.`**`__lshift__`**(*self*, *other*)
`object.`**`__rshift__`**(*self*, *other*)
`object.`**`__and__`**(*self*, *other*)
`object.`**`__xor__`**(*self*, *other*)
`object.`**`__or__`**(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a class that has an *`__add__()`* method, `x.__add__(y)` is called. The *`__divmod__()`* method should be the equivalent to using *`__floordiv__()`* and *`__mod__()`* ; it should not be related to *`__truediv__()`* (described below). Note that *`__pow__()`* should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.

Si l'une de ces méthodes n'autorise pas l'opération avec les arguments donnés, elle doit renvoyer `NotImplemented`.

`object.`**`__div__`**(*self*, *other*)
`object.`**`__truediv__`**(*self*, *other*)

The division operator (`/`) is implemented by these methods. The *`__truediv__()`* method is used when `__future__.division` is in effect, otherwise *`__div__()`* is used. If only one of these two methods is defined, the object will not support division in the alternate context ; `TypeError` will be raised instead.

`object.`**`__radd__`**(*self*, *other*)
`object.`**`__rsub__`**(*self*, *other*)
`object.`**`__rmul__`**(*self*, *other*)
`object.`**`__rdiv__`**(*self*, *other*)
`object.`**`__rtruediv__`**(*self*, *other*)
`object.`**`__rfloordiv__`**(*self*, *other*)
`object.`**`__rmod__`**(*self*, *other*)
`object.`**`__rdivmod__`**(*self*, *other*)
`object.`**`__rpow__`**(*self*, *other*)
`object.`**`__rlshift__`**(*self*, *other*)
`object.`**`__rrshift__`**(*self*, *other*)
`object.`**`__rand__`**(*self*, *other*)
`object.`**`__rxor__`**(*self*, *other*)
`object.`**`__ror__`**(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`,

`**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation and the operands are of different types. [2] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an `__rsub__()` method, `y.__rsub__(x)` is called if `x.__sub__(y)` returns *NotImplemented*.

Notez que la fonction ternaire `pow()` n'essaie pas d'appeler `__rpow__()` (les règles de coercition seraient trop compliquées).

Note : Si le type de l'opérande de droite est une sous-classe du type de l'opérande de gauche et que cette sous-classe fournit la méthode symétrique pour l'opération, cette méthode sera appelée avant la méthode originelle de l'opérande gauche. Ce comportement permet à des sous-classes de surcharger les opérations de leurs ancêtres.

`object.`**`__iadd__`**(*self*, *other*)
`object.`**`__isub__`**(*self*, *other*)
`object.`**`__imul__`**(*self*, *other*)
`object.`**`__idiv__`**(*self*, *other*)
`object.`**`__itruediv__`**(*self*, *other*)
`object.`**`__ifloordiv__`**(*self*, *other*)
`object.`**`__imod__`**(*self*, *other*)
`object.`**`__ipow__`**(*self*, *other*[, *modulo*])
`object.`**`__ilshift__`**(*self*, *other*)
`object.`**`__irshift__`**(*self*, *other*)
`object.`**`__iand__`**(*self*, *other*)
`object.`**`__ixor__`**(*self*, *other*)
`object.`**`__ior__`**(*self*, *other*)

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, to execute the statement `x += y`, where *x* is an instance of a class that has an `__iadd__()` method, `x.__iadd__(y)` is called. If *x* is an instance of a class that does not define a `__iadd__()` method, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`.

`object.`**`__neg__`**(*self*)
`object.`**`__pos__`**(*self*)
`object.`**`__abs__`**(*self*)
`object.`**`__invert__`**(*self*)

Appelée pour implémenter les opérations arithmétiques unaires (`-`, `+`, `abs()` et `~`).

`object.`**`__complex__`**(*self*)
`object.`**`__int__`**(*self*)
`object.`**`__long__`**(*self*)
`object.`**`__float__`**(*self*)

Called to implement the built-in functions `complex()`, `int()`, `long()`, and `float()`. Should return a value of the appropriate type.

`object.`**`__oct__`**(*self*)
`object.`**`__hex__`**(*self*)

Called to implement the built-in functions `oct()` and `hex()`. Should return a string value.

`object.`**`__index__`**(*self*)

Called to implement `operator.index()`. Also called whenever Python needs an integer object (such as in slicing). Must return an integer (int or long).

Nouveau dans la version 2.5.

2. Pour des opérandes de même type, on considère que si la méthode originelle (telle que `__add__()`) échoue, l'opération n'est pas autorisée et donc la méthode symétrique n'est pas appelée.

`object.__coerce__`(*self*, *other*)
Called to implement « mixed-mode » numeric arithmetic. Should either return a 2-tuple containing *self* and *other* converted to a common numeric type, or `None` if conversion is impossible. When the common type would be the type of `other`, it is sufficient to return `None`, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here). A return value of `NotImplemented` is equivalent to returning `None`.

3.4.9 Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely ; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3, coercion will not be supported.

- If the left operand of a % operator is a string or Unicode object, no coercion takes place and the string formatting operation is invoked instead.
- It is no longer recommended to define a coercion operation. Mixed-mode operations on types that don't define coercion pass the original arguments to the operation.
- New-style classes (those derived from `object`) never invoke the *__coerce__()* method in response to a binary operator ; the only time *__coerce__()* is invoked is when the built-in function `coerce()` is called.
- For most intents and purposes, an operator that returns `NotImplemented` is treated the same as one that is not implemented at all.
- Below, `__op__()` and `__rop__()` are used to signify the generic method names corresponding to an operator ; `__iop__()` is used for the corresponding in-place operator. For example, for the operator "+", *__add__()* and *__radd__()* are used for the left and right variant of the binary operator, and *__iadd__()* for the in-place variant.
- For objects *x* and *y*, first `x.__op__(y)` is tried. If this is not implemented or returns `NotImplemented`, `y.__rop__(x)` is tried. If this is also not implemented or returns `NotImplemented`, a `TypeError` exception is raised. But see the following exception :
- Exception to the previous item : if the left operand is an instance of a built-in type or a new-style class, and the right operand is an instance of a proper subclass of that type or class and overrides the base's `__rop__()` method, the right operand's `__rop__()` method is tried *before* the left operand's `__op__()` method.
  This is done so that a subclass can completely override binary operators. Otherwise, the left operand's `__op__()` method would always accept the right operand : when an instance of a given class is expected, an instance of a subclass of that class is always acceptable.
- When either operand type defines a coercion, this coercion is called before that type's `__op__()` or `__rop__()` method is called, but no sooner. If the coercion returns an object of a different type for the operand whose coercion is invoked, part of the process is redone using the new object.
- When an in-place operator (like "+=") is used, if the left operand implements `__iop__()`, it is invoked without any coercion. When the operation falls back to `__op__()` and/or `__rop__()`, the normal coercion rules apply.
- In `x + y`, if *x* is a sequence that implements sequence concatenation, sequence concatenation is invoked.
- In `x * y`, if one operand is a sequence that implements sequence repetition, and the other is an integer (`int` or `long`), sequence repetition is invoked.
- Rich comparisons (implemented by methods *__eq__()* and so on) never use coercion. Three-way comparison (implemented by *__cmp__()*) does use coercion under the same conditions as other binary operations use it.
- In the current implementation, the built-in numeric types `int`, `long`, `float`, and `complex` do not use coercion. All these types implement a *__coerce__()* method, for use by the built-in `coerce()` function.
  Modifié dans la version 2.7 : The complex type no longer makes implicit calls to the *__coerce__()* method for mixed-type binary arithmetic operations.

3.4.10 Gestionnaire de contexte With

Nouveau dans la version 2.5.

Un *gestionnaire de contexte* est un objet qui met en place un contexte prédéfini au moment de l'exécution de l'instruction `with`. Le gestionnaire de contexte gère l'entrée et la sortie de ce contexte d'exécution pour tout un bloc de code. Les gestionnaires de contextes sont normalement invoqués en utilisant une instruction `with` (décrite dans la section *L'instruction with*), mais ils peuvent aussi être directement invoqués par leurs méthodes.

Les utilisations classiques des gestionnaires de contexte sont la sauvegarde et la restauration d'états divers, le verrouillage et le déverrouillage de ressources, la fermeture de fichiers ouverts, etc.

Pour plus d'informations sur les gestionnaires de contexte, lisez typecontextmanager.

`object.`**`__enter__`**`(`*self*`)`
Entre dans le contexte d'exécution relatif à cet objet. L'instruction `with` lie la valeur de retour de cette méthode à une (ou plusieurs) cible spécifiée par la clause `as` de l'instruction, si elle est spécifiée.

`object.`**`__exit__`**`(`*self*, *exc_type*, *exc_value*, *traceback*`)`
Sort du contexte d'exécution relatif à cet objet. Les paramètres décrivent l'exception qui a causé la sortie du contexte. Si l'on sort du contexte sans exception, les trois arguments sont à `None`.

Si une exception est indiquée et que la méthode souhaite supprimer l'exception (c'est-à-dire qu'elle ne veut pas que l'exception soit propagée), elle doit renvoyer `True`. Sinon, l'exception est traitée normalement à la sortie de cette méthode.

Notez qu'une méthode `__exit__()` ne doit pas lever à nouveau l'exception qu'elle reçoit ; c'est du ressort de l'appelant.

Voir aussi :

PEP 343 - The « with » statement La spécification, les motivations et des exemples de l'instruction `with` en Python.

3.4.11 Special method lookup for old-style classes

For old-style classes, special methods are always looked up in exactly the same way as any other method or attribute. This is the case regardless of whether the method is being looked up explicitly as in `x.__getitem__(i)` or implicitly as in `x[i]`.

This behaviour means that special methods may exhibit different behaviour for different instances of a single old-style class if the appropriate special attributes are set differently :

```
>>> class C:
...     pass
...
>>> c1 = C()
>>> c2 = C()
>>> c1.__len__ = lambda: 5
>>> c2.__len__ = lambda: 9
>>> len(c1)
5
>>> len(c2)
9
```

3.4.12 Special method lookup for new-style classes

For new-style classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception (unlike the equivalent example with old-style classes) :

```
>>> class C(object):
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

La raison de ce comportement vient de certaines méthodes spéciales telles que *__hash__()* et *__repr__()* qui sont implémentées par tous les objets, y compris les objets types. Si la recherche effectuée par ces méthodes utilisait le processus normal de recherche, elles ne fonctionneraient pas si on les appelait sur l'objet type lui-même :

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

Essayer d'invoquer une méthode non liée d'une classe de cette manière est parfois appelé « confusion de méta-classe » et se contourne en shuntant l'instance lors de la recherche des méthodes spéciales :

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

En plus de shunter les attributs des instances pour fonctionner correctement, la recherche des méthodes spéciales implicites shunte aussi la méthode *__getattribute__()* même dans la méta-classe de l'objet :

```
>>> class Meta(type):
...    def __getattribute__(*args):
...       print "Metaclass getattribute invoked"
...       return type.__getattribute__(*args)
...
>>> class C(object):
...     __metaclass__ = Meta
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print "Class getattribute invoked"
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
```

(suite sur la page suivante)

(suite de la page précédente)

```
10
>>> len(c)                      # Implicit lookup
10
```

En shuntant le mécanisme de *__getattribute__()* de cette façon, cela permet d'optimiser la vitesse de l'interpréteur moyennant une certaine manœuvre dans la gestion des méthodes spéciales (la méthode spéciale *doit* être définie sur l'objet classe lui-même afin d'être invoquée de manière cohérente par l'interpréteur).

Notes

CHAPITRE 4

Modèle d'exécution

4.1 Noms et liaisons

Names refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of Python program text that is executed as a unit. The following are blocks : a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the "**-c**" option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the *exec* statement is a code block. The expression read and evaluated by the built-in function `input()` is a code block.

Un bloc de code est exécuté dans un *cadre d'exécution*. Un cadre contient des informations administratives (utilisées pour le débogage) et détermine où et comment l'exécution se poursuit après la fin de l'exécution du bloc de code.

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block ; it does not extend to the code blocks of methods – this includes generator expressions since they are implemented using a function scope. This means that the following will fail :

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

Quand un nom est utilisé dans un bloc de code, la résolution utilise la portée la plus petite. L'ensemble de toutes les portées visibles dans un bloc de code s'appelle *l'environnement* du bloc.

If a name is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a *free variable*.

When a name is not found at all, a `NameError` exception is raised. If the name refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

The following constructs bind names : formal parameters to functions, *import* statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, *for* loop header, in the second position of an *except* clause header or after *as* in a *with* statement. The *import* statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a *del* statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced by an enclosing scope ; the compiler will report a `SyntaxError`.

Chaque assignation ou instruction *import* a lieu dans un bloc défini par une définition de classe ou de fonction ou au niveau du module (le bloc de code de plus haut niveau).

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `__builtin__`. The global namespace is searched first. If the name is not found there, the builtins namespace is searched. The global statement must precede all uses of the name.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace ; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `__builtin__` (note : no "s") ; when in any other module, `__builtins__` is an alias for the dictionary of the `__builtin__` module itself. `__builtins__` can be set to a user-created dictionary to create a weak form of restricted execution.

CPython implementation detail : Users should not touch `__builtins__` ; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should *import* the `__builtin__` (no "s") module and modify its attributes appropriately.

L'espace de noms pour un module est créé automatiquement la première fois que le module est importé. Le module principal d'un script s'appelle toujours `__main__`.

L'instruction *global* a la même porte qu'une opération de liaison du même bloc. Si la portée englobante la plus petite pour une variable libre contient une instruction *global*, la variable libre est considérée globale.

A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution. The namespace of the class definition becomes the attribute dictionary of the class. Names defined at the class scope are not visible in methods.

4.1.1 Interaction avec les fonctionnalités dynamiques

There are several cases where Python statements are illegal when used in conjunction with nested scopes that contain free variables.

If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time.

If the wild card form of import — `import *` — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

If *exec* is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError` unless the exec explicitly specifies the local namespace for the *exec*. (In other words, `exec obj` would be illegal, but `exec obj in ns` would be legal.)

The `eval()`, `execfile()`, and `input()` functions and the *exec* statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace. [1] The *exec* statement and the `eval()` and `execfile()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.2 Exceptions

Les exceptions sont un moyen de sortir du flot normal d'exécution d'un bloc de code de manière à gérer des erreurs ou des conditions exceptionnelles. Une exception est *levée* au moment où l'erreur est détectée ; elle doit être *gérée* par le bloc de code qui l'entoure ou par tout bloc de code qui a, directement ou indirectement, invoqué le bloc de code où l'erreur s'est produite.

L'interpréteur Python lève une exception quand il détecte une erreur à l'exécution (telle qu'une division par zéro). Un programme Python peut aussi lever explicitement une exception avec l'instruction *raise*. Les gestionnaires d'exception sont spécifiés avec l'instruction *try* ... *except*. La clause *finally* d'une telle instruction peut être utilisée pour spécifier un code de nettoyage qui ne gère pas l'exception mais qui est exécuté quoi qu'il arrive (exception ou pas).

Python utilise le modèle par *terminaison* de gestion des erreurs : un gestionnaire d'exception peut trouver ce qui est arrivé et continuer l'exécution à un niveau plus élevé mais il ne peut pas réparer l'origine de l'erreur et ré-essayer l'opération qui a échoué (sauf à entrer à nouveau dans le code en question par le haut).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is `SystemExit`.

Les exceptions sont identifiées par des instances de classe. La clause *except* sélectionnée dépend de la classe de l'instance : elle doit faire référence à la classe de l'instance ou à une de ses classes ancêtres. L'instance peut être transmise au gestionnaire et peut apporter des informations complémentaires sur les conditions de l'exception.

Exceptions can also be identified by strings, in which case the *except* clause is selected by object identity. An arbitrary value can be raised along with the identifying string which can be passed to the handler.

Note : Messages to exceptions are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

Reportez-vous aussi aux descriptions de l'instruction *try* dans la section *L'instruction try* et de l'instruction *raise* dans la section *L'instruction raise*.

Notes

1. En effet, le code qui est exécuté par ces opérations n'est pas connu au moment où le module est compilé.

CHAPITRE 5

Expressions

Ce chapitre explique la signification des éléments des expressions en Python.

Notes sur la syntaxe : dans ce chapitre et le suivant, nous utilisons la notation BNF étendue pour décrire la syntaxe, pas l'analyse lexicale. Quand une règle de syntaxe est de la forme

```
name   ::=   othername
```

et qu'aucune sémantique n'est donnée, la sémantique de `name` est la même que celle de `othername`.

5.1 Conversions arithmétiques

When a description of an arithmetic operator below uses the phrase « the numeric arguments are converted to a common type, » the arguments are coerced using the coercion rules listed at *Coercion rules*. If both arguments are standard numeric types, the following coercions are applied :

- Si l'un des deux arguments est du type nombre complexe, l'autre est converti en nombre complexe ;
- sinon, si l'un des arguments est un nombre à virgule flottante, l'autre est converti en nombre à virgule flottante ;
- otherwise, if either argument is a long integer, the other is converted to long integer ;
- otherwise, both must be plain integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string left argument to the "%" operator). Extensions can define their own coercions.

5.2 Atomes

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is :

```
atom       ::=  identifier | literal | enclosure
enclosure  ::=  parenth_form | list_display
                | generator_expression | dict_display | set_display
                | string_conversion | yield_atom
```

5.2.1 Identifiants (noms)

Un identifiant qui apparaît en tant qu'atome est un nom. Lisez la section *Identifiants et mots-clés* pour la définition lexicale et la section *Noms et liaisons* pour la documentation sur les noms et les liaisons afférentes.

Quand un nom est lié à un objet, l'évaluation de l'atome produit cet objet. Quand le nom n'est pas lié, toute tentative de l'évaluer lève une exception `NameError`.

Transformation des noms privés : lorsqu'un identificateur qui apparaît textuellement dans la définition d'une classe commence par deux (ou plus) caractères de soulignement et ne se termine pas par deux (ou plus) caractères de soulignement, il est considéré comme un *nom privé <private name>* de cette classe. Les noms privés sont transformés en une forme plus longue avant que le code ne soit généré pour eux. La transformation insère le nom de la classe, avec les soulignés enlevés et un seul souligné inséré devant le nom. Par exemple, l'identificateur `__spam` apparaissant dans une classe nommée `Ham` est transformé en `_Ham__spam`. Cette transformation est indépendante du contexte syntaxique dans lequel l'identificateur est utilisé. Si le nom transformé est extrêmement long (plus de 255 caractères), l'implémentation peut le tronquer. Si le nom de la classe est constitué uniquement de traits de soulignement, aucune transformation n'est effectuée.

5.2.2 Littéraux

Python supports string literals and various numeric literals :

```
literal  ::=  stringliteral | integer | longinteger
              | floatnumber | imagnumber
```

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section *Littéraux* for details.

Tous les littéraux sont de types immuables et donc l'identifiant de l'objet est moins important que sa valeur. Des évaluations multiples de littéraux avec la même valeur (soit la même occurrence dans le texte du programme, soit une autre occurrence) résultent dans le même objet ou un objet différent avec la même valeur.

5.2.3 Formes parenthésées

Une forme parenthésée est une liste d'expressions (cette liste est en fait optionnelle) placée à l'intérieur de parenthèses :

```
parenth_form   ::=   "(" [expression_list] ")"
```

Une liste d'expressions entre parenthèses produit ce que la liste de ces expressions produirait : si la liste contient au moins une virgule, elle produit un n-uplet (type *tuple*) ; sinon, elle produit l'expression elle-même (qui constitue donc elle-même la liste d'expressions).

Une paire de parenthèses vide produit un objet *tuple* vide. Comme les *tuples* sont immuables, la règle pour les littéraux s'applique (c'est-à-dire que deux occurrences du *tuple* vide peuvent, ou pas, produire le même objet).

Notez que les *tuples* ne sont pas créés par les parenthèses mais par l'utilisation de la virgule. L'exception est le tuple vide, pour lequel les parenthèses *sont requises* (autoriser que « rien » ne soit pas parenthésé dans les expressions aurait généré des ambigüités et aurait permis à certaines coquilles de passer inaperçu).

5.2.4 Agencements de listes

Un agencement de liste est une suite (possiblement vide) d'expressions à l'intérieur de crochets :

```
list_display          ::=   "[" [expression_list | list_comprehension] "]"
list_comprehension    ::=   expression list_for
list_for              ::=   "for" target_list "in" old_expression_list [list_iter]
old_expression_list   ::=   old_expression [("," old_expression)+ [","]]
old_expression        ::=   or_test | old_lambda_expr
list_iter             ::=   list_for | list_if
list_if               ::=   "if" old_expression [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is supplied, it consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new list are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached [1].

5.2.5 Displays for sets and dictionaries

For constructing a set or a dictionary Python provides special syntax called « displays », each of them in two flavors :

- soit le contenu du conteneur est listé explicitement,
- soit il est calculé à l'aide de la combinaison d'une boucle et d'instructions de filtrage, appelée une *compréhension* (dans le sens de ce qui sert à définir un concept, par opposition à *extension*).

Les compréhensions sont constituées des éléments de syntaxe communs suivants :

```
comprehension   ::=   expression comp_for
comp_for        ::=   "for" target_list "in" or_test [comp_iter]
comp_iter       ::=   comp_for | comp_if
comp_if         ::=   "if" expression_nocond [comp_iter]
```

1. In Python 2.3 and later releases, a list comprehension « leaks » the control variables of each `for` it contains into the containing scope. However, this behavior is deprecated, and relying on it will not work in Python 3.

Une compréhension est constituée par une seule expression suivie par au moins une clause *for* et zéro ou plus clauses *for* ou *if*. Dans ce cas, les éléments du nouveau conteneur sont ceux qui auraient été produits si l'on avait considéré toutes les clauses *for* ou *if* comme des blocs, imbriqués de la gauche vers la droite, et évalué l'expression pour produire un élément à chaque fois que le bloc le plus imbriqué était atteint.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don't « leak » in the enclosing scope.

5.2.6 Générateurs (expressions)

Une expression générateur est une notation concise pour un générateur, entourée de parenthèses :

```
generator_expression  ::=  "(" expression comp_for ")"
```

Une expression générateur produit un nouvel objet générateur. Sa syntaxe est la même que celle des compréhensions, sauf qu'elle est entourée de parenthèses au lieu de crochets ou d'accolades.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for generator object (in the same fashion as normal generators). However, the leftmost *for* clause is immediately evaluated, so that an error produced by it can be seen before any other possible error in the code that handles the generator expression. Subsequent *for* clauses cannot be evaluated immediately since they may depend on the previous *for* loop. For example : `(x*y for x in range(10) for y in bar(x))`.

The parentheses can be omitted on calls with only one argument. See section *Appels* for the detail.

5.2.7 Agencements de dictionnaires

Un agencement de dictionnaire est une série (possiblement vide) de couples clés-valeurs entourée par des accolades :

```
dict_display        ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list      ::=  key_datum ("," key_datum)* [","]
key_datum           ::=  expression ":" expression
dict_comprehension  ::=  expression ":" expression comp_for
```

Un agencement de dictionnaire produit un nouvel objet dictionnaire.

Si une séquence (dont les éléments sont séparés par des virgules) de couples clés-valeurs est fournie, les couples sont évalués de la gauche vers la droite pour définir les entrées du dictionnaire : chaque objet clé est utilisé comme clé dans le dictionnaire pour stocker la donnée correspondante. Cela signifie que vous pouvez spécifier la même clé plusieurs fois dans la liste des couples clés-valeurs et, dans ce cas, la valeur finalement stockée dans le dictionnaire est la dernière donnée.

Une compréhension de dictionnaire, au contraire des compréhensions de listes ou d'ensembles, requiert deux expressions séparées par une virgule et suivies par les clauses usuelles « for » et « if ». Quand la compréhension est exécutée, les éléments clés-valeurs sont insérés dans le nouveau dictionnaire dans l'ordre dans lequel ils sont produits.

Les restrictions relatives aux types des clés sont données dans la section *Hiérarchie des types standards* (pour résumer, le type de la clé doit être *hachable*, ce qui exclut tous les objets muables). Les collisions entre les clés dupliquées ne sont pas détectées ; la dernière valeur (celle qui apparaît le plus à droite dans l'agencement) stockée prévaut pour une clé donnée.

5.2.8 Agencements d'ensembles

Un agencement d'ensemble (type *set*) est délimité par des accolades et se distingue de l'agencement d'un dictionnaire par le fait qu'il n'y a pas de « deux points » `:` pour séparer les clés et les valeurs :

```
set_display  ::=  "{" (expression_list | comprehension) "}"
```

Un agencement d'ensemble produit un nouvel objet ensemble muable, le contenu étant spécifié soit par une séquence d'expression, soit par une compréhension. Quand une liste (dont les éléments sont séparés par des virgules) est fournie, ses éléments sont évalués de la gauche vers la droite et ajoutés à l'objet ensemble. Quand une compréhension est fournie, l'ensemble est construit à partir des éléments produits par la compréhension.

Un ensemble vide ne peut pas être construit par `{}` ; cette écriture construit un dictionnaire vide.

5.2.9 String conversions

A string conversation is an expression list enclosed in reverse (a.k.a. backward) quotes :

```
string_conversion  ::=  "`" expression_list "`"
```

A string conversion evaluates the contained expression list and converts the resulting object into a string according to rules specific to its type.

If the object is a string, a number, `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function `eval()` to yield an expression with the same value (or an approximation, if floating point numbers are involved).

(In particular, converting a string adds quotes around it and converts « funny » characters to escape sequences that are safe to print.)

Recursive objects (for example, lists or dictionaries that contain a reference to themselves, directly or indirectly) use `...` to indicate a recursive reference, and the result cannot be passed to `eval()` to get an equal value (`SyntaxError` will be raised instead).

The built-in function `repr()` performs exactly the same conversion in its argument as enclosing it in parentheses and reverse quotes does. The built-in function `str()` performs a similar but more user-friendly conversion.

5.2.10 Expressions `yield`

```
yield_atom        ::=  "(" yield_expression ")"
yield_expression  ::=  "yield" [expression_list]
```

Nouveau dans la version 2.5.

The *`yield`* expression is only used when defining a generator function, and can only be used in the body of a function definition. Using a *`yield`* expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of a generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first *`yield`* expression, where it is suspended again, returning the value of *`expression_list`* to generator's caller. By suspended we mean that all local state is retained, including the current bindings of local variables,

the instruction pointer, and the internal evaluation stack. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the *yield* expression was just another external call. The value of the *yield* expression after resuming depends on the method which resumed the execution.

All of this makes generator functions quite similar to coroutines ; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where should the execution continue after it yields ; the control is always transferred to the generator's caller.

Méthodes des générateurs-itérateurs

Cette sous-section décrit les méthodes des générateurs-itérateurs. Elles peuvent être utilisées pour contrôler l'exécution des fonctions générateurs.

Notez que l'appel à une méthode ci-dessous d'un générateur alors que le générateur est déjà en cours d'exécution lève une exception `ValueError`.

`generator.`**`next`**`()`
: Starts the execution of a generator function or resumes it at the last executed *yield* expression. When a generator function is resumed with a *next()* method, the current *yield* expression always evaluates to `None`. The execution then continues to the next *yield* expression, where the generator is suspended again, and the value of the *expression_list* is returned to *next()*'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

`generator.`**`send`**`(`*value*`)`
: Resumes the execution and « sends » a value into the generator function. The `value` argument becomes the result of the current *yield* expression. The *send()* method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When *send()* is called to start the generator, it must be called with `None` as the argument, because there is no *yield* expression that could receive the value.

`generator.`**`throw`**`(`*type*`[`, *value*`[`, *traceback*`]]`)
: Raises an exception of type `type` at the point where generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

`generator.`**`close`**`()`
: Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function then raises `StopIteration` (by exiting normally, or due to already being closed) or `GeneratorExit` (by not catching the exception), close returns to its caller. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. *close()* does nothing if the generator has already exited due to an exception or normal exit.

Voici un exemple simple qui montre le comportement des générateurs et des fonctions générateurs :

```
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception, e:
...                 value = e
...     finally:
...         print "Don't forget to clean up when 'close()' is called."
...
>>> generator = echo(1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
1
>>> print generator.next()
None
>>> print generator.send(2)
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

Voir aussi :

PEP 342 – Coroutines *via* des générateurs améliorés Proposition d'améliorer l'API et la syntaxe des générateurs, de manière à pouvoir les utiliser comme de simples coroutines.

5.3 Primaires

Les primaires (*primary* dans la grammaire formelle ci-dessous) représentent les opérations qui se lient au plus proche dans le langage. Leur syntaxe est :

```
primary   ::=   atom | attributeref | subscription | slicing | call
```

5.3.1 Références à des attributs

Une référence à un attribut (*attributeref* dans la grammaire formelle ci-dessous) est une primaire suivie par un point et un nom :

```
attributeref   ::=   primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

5.3.2 Sélections

Une sélection (*subscription* dans la grammaire formelle ci-dessous) désigne un élément dans un objet séquence (chaîne, n-uplet ou liste) ou tableau de correspondances (dictionnaire) :

```
subscription   ::=   primary "[" expression_list "]"
```

The primary must evaluate to an object of a sequence or mapping type.

Si la primaire est un tableau de correspondances, la liste d'expressions (*expression_list* dans la grammaire formelle ci-dessous) doit pouvoir être évaluée comme un objet dont la valeur est une des clés du tableau de correspondances et la

sélection désigne la valeur qui correspond à cette clé (la liste d'expressions est un n-uplet sauf si elle comporte exactement un élément).

If the primary is a sequence, the expression list must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g., `x[-1]` selects the last item of `x`.) The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

Les éléments des chaînes sont des caractères. Un caractère n'est pas un type en tant que tel, c'est une chaîne de longueur un.

5.3.3 Tranches

Une tranche (*slicing* dans la grammaire formelle ci-dessous) sélectionne un intervalle d'éléments d'un objet séquence (par exemple une chaîne, un n-uplet ou une liste, respectivement les types *string*, *tuple* et *list*). Les tranches peuvent être utilisées comme des expressions ou des cibles dans les assignations ou les instructions `del`. La syntaxe est la suivante :

```
slicing            ::=  simple_slicing | extended_slicing
simple_slicing     ::=  primary "[" short_slice "]"
extended_slicing   ::=  primary "[" slice_list "]"
slice_list         ::=  slice_item ("," slice_item)* [","]
slice_item         ::=  expression | proper_slice | ellipsis
proper_slice       ::=  short_slice | long_slice
short_slice        ::=  [lower_bound] ":" [upper_bound]
long_slice         ::=  short_slice ":" [stride]
lower_bound        ::=  expression
upper_bound        ::=  expression
stride             ::=  expression
ellipsis           ::=  "..."
```

There is ambiguity in the formal syntax here : anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice nor ellipses). Similarly, when the slice list has exactly one short slice and no trailing comma, the interpretation as a simple slicing takes priority over that as an extended slicing.

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers ; defaults are zero and the `sys.maxint`, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index *k* such that `i <= k < j` where *i* and *j* are the specified lower and upper bounds. This may be an empty sequence. It is not an error if *i* or *j* lie outside the range of valid indexes (such items don't exist so they aren't selected).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items ; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in `Ellipsis` object. The conversion of a proper slice is a slice object (see section *Hiérarchie des types standards*) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

5.3.4 Appels

Un appel (*call* dans la grammaire ci-dessous) appelle un objet appelable (par exemple, une *fonction*) avec, possiblement, une liste d'*arguments* :

```
call                 ::=  primary "(" [argument_list [","]
                          | expression genexpr_for] ")"
argument_list        ::=  positional_arguments ["," keyword_arguments]
                          ["," "*" expression] ["," keyword_arguments]
                          ["," "**" expression]
                          | keyword_arguments ["," "*" expression]
                          ["," "**" expression]
                          | "*" expression ["," keyword_arguments] ["," "**" expression]
                          | "**" expression
positional_arguments ::=  expression ("," expression)*
keyword_arguments    ::=  keyword_item ("," keyword_item)*
keyword_item         ::=  identifier "=" expression
```

A trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and certain class instances themselves are callable ; extensions may define additional callable object types). All argument expressions are evaluated before the call is attempted. Please refer to section *Définition de fonctions* for the syntax of formal *parameter* lists.

Si des arguments par mots-clés sont présents, ils sont d'abord convertis en arguments positionnels, comme suit. Pour commencer, une liste de *slots* vides est créée pour les paramètres formels. S'il y a N arguments positionnels, ils sont placés dans les N premiers *slots*. Ensuite, pour chaque argument par mot-clé, l'identifiant est utilisé pour déterminer le *slot* correspondant (si l'identifiant est le même que le nom du premier paramètre formel, le premier *slot* est utilisé, et ainsi de suite). Si le *slot* est déjà rempli, une exception `TypeError` est levée. Sinon, la valeur de l'argument est placée dans le *slot*, ce qui le remplit (même si l'expression est `None`, cela remplit le *slot*). Quand tous les arguments ont été traités, les *slots* qui sont toujours vides sont remplis avec la valeur par défaut correspondante dans la définition de la fonction (les valeurs par défaut sont calculées, une seule fois, lorsque la fonction est définie ; ainsi, un objet mutable tel qu'une liste ou un dictionnaire utilisé en tant valeur par défaut sera partagé entre tous les appels qui ne spécifient pas de valeur d argument pour ce *slot* ; on évite généralement de faire ça). S'il reste des *slots* pour lesquels aucune valeur par défaut n'est définie, une exception `TypeError` est levée. Sinon, la liste des *slots* remplie est utilisée en tant que liste des arguments pour l'appel.

Une implémentation peut fournir des fonctions natives dont les paramètres positionnels n'ont pas de nom, même s'ils sont « nommés » pour les besoins de la documentation. Ils ne peuvent donc pas être spécifiés par mot-clé. En CPython, les fonctions implémentées en C qui utilisent `PyArg_ParseTuple()` pour analyser leurs arguments en font partie.

S'il y a plus d'arguments positionnels que de *slots* de paramètres formels, une exception `TypeError` est levée, à moins qu'un paramètre formel n'utilise la syntaxe `*identifier` ; dans ce cas, le paramètre formel reçoit un n-uplet contenant les arguments positionnels en supplément (ou un n-uplet vide s'il n'y avait pas d'arguments positionnel en trop).

Si un argument par mot-clé ne correspond à aucun nom de paramètre formel, une exception `TypeError` est levée, à moins qu'un paramètre formel n'utilise la syntaxe `**identifier` ; dans ce cas, le paramètre formel reçoit un dictionnaire contenant les arguments par mot-clé en trop (en utilisant les mots-clés comme clés et les arguments comme valeurs pour ce dictionnaire), ou un (nouveau) dictionnaire vide s'il n'y a pas d'argument par mot-clé en trop.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an iterable. Elements from this iterable are treated as if they were additional positional arguments ; if there are positional arguments *x1*, …, *xN*, and `expression` evaluates to a sequence *y1*, …, *yM*, this is equivalent to a call with M+N positional arguments *x1*, …, *xN*, *y1*, …, *yM*.

A consequence of this is that although the `*expression` syntax may appear *after* some keyword arguments, it is

processed *before* the keyword arguments (and the `**expression` argument, if any – see below). So :

```
>>> def f(a, b):
...     print a, b
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

Il est inhabituel que les syntaxes d'arguments par mots-clés et `*expression` soient utilisés simultanément dans un même appel, ce qui fait que la confusion reste hypothétique.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a mapping, the contents of which are treated as additional keyword arguments. In the case of a keyword appearing in both `expression` and as an explicit keyword argument, a `TypeError` exception is raised.

Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names. Formal parameters using the syntax `(sublist)` cannot be used as keyword argument names ; the outermost sublist corresponds to a single unnamed argument slot, and the argument value is assigned to the sublist using the usual tuple assignment rules after all other parameter processing is done.

Un appel renvoie toujours une valeur, possiblement `None`, à moins qu'il ne lève une exception. La façon dont celle valeur est calculée dépend du type de l'objet appelable.

Si c'est —

une fonction définie par l'utilisateur : le bloc de code de la fonction est exécuté, il reçoit la liste des arguments. La première chose que le bloc de code fait est de lier les paramètres formels aux arguments ; ceci est décrit dans la section *Définition de fonctions*. Quand le bloc de code exécute l'instruction *`return`*, cela spécifie la valeur de retour de l'appel de la fonction.

une fonction ou une méthode native : le résultat dépend de l'interpréteur ; lisez built-in-funcs pour une description des fonctions et méthodes natives.

un objet classe : une nouvelle instance de cette classe est renvoyée.

une méthode d'instance de classe : la fonction correspondante définie par l'utilisateur est appelée, avec la liste d'arguments qui est plus grande d'un élément que la liste des arguments de l'appel : l'instance est placée en tête des arguments.

une instance de classe : la classe doit définir une méthode *`__call__()`* ; l'effet est le même que si cette méthode était appelée.

5.4 L'opérateur puissance

L'opérateur puissance est plus prioritaire que les opérateurs unaires sur sa gauche ; il est moins prioritaire que les opérateurs unaires sur sa droite. La syntaxe est :

```
power   ::=   primary ["**" u_expr]
```

Ainsi, dans une séquence sans parenthèse de puissance et d'opérateurs unaires, les opérateurs sont évalués de droite à gauche (ceci ne contraint pas l'ordre d'évaluation des opérandes) : `-1**2` donne `-1`.

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments : it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type. The result type is that of the arguments after coercion.

With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative ; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised).

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `ValueError`.

5.5 Arithmétique unaire et opérations sur les bits

Toute l'arithmétique unaire et les opérations sur les bits ont la même priorité :

```
u_expr   ::=   power | "-" u_expr | "+" u_expr | "~" u_expr
```

L'opérateur unaire − (moins) produit l'opposé de son argument numérique.

L'opérateur unaire + (plus) produit son argument numérique inchangé.

The unary ~ (invert) operator yields the bitwise inversion of its plain or long integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers.

Dans ces trois cas, si l'argument n'est pas du bon type, une exception `TypeError` est levée.

5.6 Opérations arithmétiques binaires

Les opérations arithmétiques binaires suivent les conventions pour les priorités. Notez que certaines de ces opérations s'appliquent aussi à des types non numériques. À part l'opérateur puissance, il n'y a que deux niveaux, le premier pour les opérateurs multiplicatifs et le second pour les opérateurs additifs :

```
m_expr   ::=   u_expr | m_expr "*" u_expr | m_expr "//" u_expr | m_expr "/" u_expr
               | m_expr "%" u_expr
a_expr   ::=   m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed ; a negative repetition factor yields an empty sequence.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type ; the result is that of mathematical division with the "floor" function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

L'opérateur `%` (modulo) produit le reste de la division entière du premier argument par le second. Les arguments numériques sont d'abord convertis vers un type commun. Un zéro en second argument lève une exception `ZeroDivisionError`. Les arguments peuvent être des nombres à virgule flottante, par exemple `3.14%0.7` vaut `0.34` (puisque `3.14` égale `4*0.7+0.34`). L'opérateur modulo produit toujours un résultat du même signe que le se-

cond opérande (ou zéro) ; la valeur absolue du résultat est strictement inférieure à la valeur absolue du second opérande [2].

The integer division and modulo operators are connected by the following identity : `x == (x/y)*y + (x%y)`. Integer division and modulo are also connected with the built-in function `divmod()` : `divmod(x, y) == (x/y, x%y)`. These identities don't hold for floating point numbers ; there similar identities hold approximately where `x/y` is replaced by `floor(x/y)` or `floor(x/y) - 1` [3].

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string and unicode objects to perform string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section string-formatting.

Obsolète depuis la version 2.3 : The floor division operator, the modulo operator, and the `divmod()` function are no longer defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The + (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

L'opérateur − (soustraction) produit la différence entre ses arguments. Les arguments numériques sont d'abord convertis vers un type commun.

5.7 Opérations de décalage

Les opérations de décalage sont moins prioritaires que les opérations arithmétiques :

```
shift_expr   ::=   a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by *n* bits is defined as division by `pow(2, n)`. A left shift by *n* bits is defined as multiplication with `pow(2, n)`. Negative shift counts raise a `ValueError` exception.

Note : Dans l'implémentation actuelle, l'opérande de droite doit être au maximum `sys.maxsize`. Si l'opérande de droite est plus grand que `sys.maxsize`, une exception `OverflowError` est levée.

---

2. Bien que `abs(x%y) < abs(y)` soit vrai mathématiquement, ce n'est pas toujours vrai pour les nombres à virgule flottante en raison des arrondis. Par exemple, en supposant que Python tourne sur une plateforme où les *float* sont des nombres à double précision IEEE 754, afin que `-1e-100 % 1e100` soit du même signe que `1e100`, le résultat calculé est `-1e-100 + 1e100`, qui vaut exactement `1e100` dans ce standard. Or, la fonction `math.fmod()` renvoie un résultat dont le signe est le signe du premier argument, c'est-à-dire `-1e-100` dans ce cas. La meilleure approche dépend de l'application.

3. If x is very close to an exact integer multiple of y, it's possible for `floor(x/y)` to be one larger than `(x-x%y)/y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

5.8 Opérations binaires bit à bit

Chacune des trois opérations binaires bit à bit possède une priorité différente :

```
and_expr   ::=   shift_expr | and_expr "&" shift_expr
xor_expr   ::=   and_expr | xor_expr "^" and_expr
or_expr    ::=   xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be plain or long integers. The arguments are converted to a common type.

5.9 Comparaisons

Au contraire du C, toutes les opérations de comparaison en Python possèdent la même priorité, qui est plus faible que celle des opérations arithmétiques, décalages ou binaires bit à bit. Toujours contrairement au C, les expressions telles que `a < b < c` sont interprétées comme elles le seraient conventionnellement en mathématiques :

```
comparison      ::=   or_expr ( comp_operator or_expr )*
comp_operator   ::=   "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
                      | "is" ["not"] | ["not"] "in"
```

Les comparaisons produisent des valeurs booléennes : `True` ou `False`.

Les comparaisons peuvent être enchaînées arbitrairement, par exemple `x < y <= z` est équivalent à `x < y and y <= z`, sauf que `y` est évalué seulement une fois (mais dans les deux cas, `z` n'est pas évalué du tout si `x < y` s'avère être faux).

Formellement, si *a*, *b*, *c*, …, *y*, *z* sont des expressions et *op1*, *op2*, …, *opN* sont des opérateurs de comparaison, alors `a op1 b op2 c ... y opN z` est équivalent à `a op1 b and b op2 c and ... y opN z`, sauf que chaque expression est évaluée au maximum une fois.

Notez que `a op1 b op2 c` n'implique aucune comparaison entre *a* et *c*. Ainsi, par exemple, `x < y > z` est parfaitement légal (mais peut-être pas très élégant).

The forms <> and != are equivalent; for consistency with C, != is preferred; where != is mentioned below <> is also accepted. The <> spelling is considered obsolescent.

5.9.1 Value comparisons

The operators <, >, ==, >=, <=, and != compare the values of two objects. The objects do not need to have the same type.

Chapter *Objets, valeurs et types* states that objects have a value (in addition to type and identity). The value of an object is a rather abstract notion in Python : For example, there is no canonical access method for an object's value. Also, there is no requirement that the value of an object should be constructed in a particular way, e.g. comprised of all its data attributes. Comparison operators implement a particular notion of what the value of an object is. One can think of them as defining the value of an object indirectly, by means of their comparison implementation.

Types can customize their comparison behavior by implementing a `__cmp__()` method or *rich comparison methods* like `__lt__()`, described in *Personnalisation de base*.

The default behavior for equality comparison (== and !=) is based on the identity of the objects. Hence, equality comparison of instances with the same identity results in equality, and equality comparison of instances with different identities results in inequality. A motivation for this default behavior is the desire that all objects should be reflexive (i.e. `x is y` implies `x == y`).

The default order comparison (<, >, <=, and >=) gives a consistent but arbitrary order.

(This unusual definition of comparison was used to simplify the definition of operations like sorting and the `in` and `not in` operators. In the future, the comparison rules for objects of different types are likely to change.)

The behavior of the default equality comparison, that instances with different identities are always unequal, may be in contrast to what types will need that have a sensible definition of object value and value-based equality. Such types will need to customize their comparison behavior, and in fact, a number of built-in types have done that.

The following list describes the comparison behavior of the most important built-in types.

- Numbers of built-in numeric types (typesnumeric) and of the standard library types `fractions.Fraction` and `decimal.Decimal` can be compared within and across their types, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.
- Strings (instances of `str` or `unicode`) compare lexicographically using the numeric equivalents (the result of the built-in function `ord()`) of their characters. [4] When comparing an 8-bit string and a Unicode string, the 8-bit string is converted to Unicode. If the conversion fails, the strings are considered unequal.
- Instances of `tuple` or `list` can be compared only within each of their types. Equality comparison across these types results in unequality, and ordering comparison across these types gives an arbitrary order.
  These sequences compare lexicographically using comparison of corresponding elements, whereby reflexivity of the elements is enforced.
  In enforcing reflexivity of elements, the comparison of collections assumes that for a collection element `x`, `x == x` is always true. Based on that assumption, element identity is compared first, and element comparison is performed only for distinct elements. This approach yields the same result as a strict element comparison would, if the compared elements are reflexive. For non-reflexive elements, the result is different than for strict element comparison.
  Lexicographical comparison between built-in collections works as follows :
  - For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).

4. The Unicode standard distinguishes between *code points* (e.g. U+0041) and *abstract characters* (e.g. « LATIN CAPITAL LETTER A »). While most abstract characters in Unicode are only represented using one code point, there is a number of abstract characters that can in addition be represented using a sequence of more than one code point. For example, the abstract character « LATIN CAPITAL LETTER C WITH CEDILLA » can be represented as a single *precomposed character* at code position U+00C7, or as a sequence of a *base character* at code position U+0043 (LATIN CAPITAL LETTER C), followed by a *combining character* at code position U+0327 (COMBINING CEDILLA).

The comparison operators on unicode strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, `u"\u00C7" == u"\u0043\u0327"` is `False`, even though both strings represent the same abstract character « LATIN CAPITAL LETTER C WITH CEDILLA ».

To compare strings at the level of abstract characters (that is, in a way intuitive to humans), use `unicodedata.normalize()`.

— Collections are ordered the same as their first unequal elements (for example, `cmp([1,2,x], [1,2,y])` returns the same as `cmp(x,y)`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).

— Mappings (instances of `dict`) compare equal if and only if they have equal *(key, value)* pairs. Equality comparison of the keys and values enforces reflexivity.
Outcomes other than equality are resolved consistently, but are not otherwise defined. [5]

— Most other objects of built-in types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

User-defined classes that customize their comparison behavior should follow some consistency rules, if possible :

— Equality comparison should be reflexive. In other words, identical objects should compare equal :

`x is y` implies `x == y`

— Comparison should be symmetric. In other words, the following expressions should have the same result :

`x == y` and `y == x`
`x != y` and `y != x`
`x < y` and `y > x`
`x <= y` and `y >= x`

— Comparison should be transitive. The following (non-exhaustive) examples illustrate that :

`x > y and y > z` implies `x > z`
`x < y and y <= z` implies `x < z`

— Inverse comparison should result in the boolean negation. In other words, the following expressions should have the same result :

`x == y` and `not x != y`
`x < y` and `not x >= y` (for total ordering)
`x > y` and `not x <= y` (for total ordering)

The last two expressions apply to totally ordered collections (e.g. to sequences, but not to sets or mappings). See also the `total_ordering()` decorator.

— The `hash()` result should be consistent with equality. Objects that are equal should either have the same hash value, or be marked as unhashable.

Python does not enforce these consistency rules.

5.9.2 Membership test operations

The operators *in* and *not in* test for membership. `x in s` evaluates to `True` if *x* is a member of *s*, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which *in* tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

For the string and bytes types, `x in y` is `True` if and only if *x* is a substring of *y*. An equivalent test is `y.find(x) != -1`. Empty strings are always considered to be a substring of any other string, so `"" in "abc"` will return `True`.

For user-defined classes which define the *__contains__()* method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

For user-defined classes which do not define *__contains__()* but do define *__iter__()*, `x in y` is `True` if some value `z` with `x == z` is produced while iterating over `y`. If an exception is raised during the iteration, it is as if *in* raised that exception.

5. Earlier versions of Python used lexicographic comparison of the sorted (key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.

Lastly, the old-style iteration protocol is tried : if a class defines *__getitem__()*, `x in y` is `True` if and only if there is a non-negative integer index *i* such that `x == y[i]`, and all lower integer indices do not raise `IndexError` exception. (If any other exception is raised, it is as if *in* raised that exception).

L'opérateur *not in* est défini comme produisant le contraire de *in*.

5.9.3 Identity comparisons

The operators *is* and *is not* test for object identity : `x is y` is true if and only if *x* and *y* are the same object. `x is not y` yields the inverse truth value. [6]

5.10 Opérations booléennes

```
or_test    ::=   and_test | or_test "or" and_test
and_test   ::=   not_test | and_test "and" not_test
not_test   ::=   comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false : `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. (See the *__nonzero__()* special method for a way to change this.)

L'opérateur *not* produit `True` si son argument est faux, `False` sinon.

L'expression `x and y` commence par évaluer *x* ; si *x* est faux, sa valeur est renvoyée ; sinon, *y* est évalué et la valeur résultante est renvoyée.

L'expression `x or y` commence par évaluer *x* ; si *x* est vrai, sa valeur est renvoyée ; sinon, *y* est évalué et la valeur résultante est renvoyée.

(Note that neither *and* nor *or* restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because *not* has to invent a value anyway, it does not bother to return a value of the same type as its argument, so e.g., `not 'foo'` yields `False`, not `''`.)

5.11 Conditional Expressions

Nouveau dans la version 2.5.

```
conditional_expression   ::=   or_test ["if" or_test "else" expression]
expression               ::=   conditional_expression | lambda_expr
```

Les expressions conditionnelles (parfois appelées « opérateur ternaire ») sont les moins prioritaires de toutes les opérations Python.

The expression `x if C else y` first evaluates the condition, *C* (*not x*) ; if *C* is true, *x* is evaluated and its value is returned ; otherwise, *y* is evaluated and its value is returned.

6. En raison du ramasse-miettes automatique et de la nature dynamique des descripteurs, vous pouvez être confronté à un comportement semblant bizarre lors de certaines utilisations de l'opérateur *is*, par exemple si cela implique des comparaisons entre des méthodes d'instances ou des constantes. Allez vérifier dans la documentation pour plus d'informations.

Voir la **PEP 308** pour plus de détails sur les expressions conditionnelles.

5.12 Expressions lambda

```
lambda_expr       ::=   "lambda" [parameter_list]: expression
old_lambda_expr   ::=   "lambda" [parameter_list]: old_expression
```

Lambda expressions (sometimes called lambda forms) have the same syntactic position as expressions. They are a shorthand to create anonymous functions ; the expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with

```
def <lambda>(parameters):
    return expression
```

See section *Définition de fonctions* for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements.

5.13 Listes d'expressions

```
expression_list   ::=   expression ( "," expression )* [","]
```

An expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

La virgule finale est nécessaire pour créer un singleton (c'est-à-dire un n-uplet composé d'un seul élément) : elle est optionnelle dans tous les autres cas. Une expression seule sans virgule finale ne crée pas un n-uplet mais produit la valeur de cette expression (pour créer un *tuple* vide, utilisez une paire de parenthèses vide : `()`).

5.14 Ordre d'évaluation

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

Dans les lignes qui suivent, les expressions sont évaluées suivant l'ordre arithmétique de leurs suffixes :

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

5.15 Priorités des opérateurs

The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right — see section *Comparaisons* — and exponentiation, which groups from right to left).

Opérateur	Description
`lambda`	Expression lambda
`if` – `else`	Expressions conditionnelle
`or`	OR (booléen)
`and`	AND (booléen)
`not` x	NOT (booléen)
`in`, `not in`, `is`, `is not`, <, <=, >, >=, <>, !=, ==	Comparaisons, y compris les tests d'appartenance et les tests d'identifiants
|	OR (bit à bit)
^	XOR (bit à bit)
&	AND (bit à bit)
<<, >>	décalages
+, –	Addition et soustraction
*, /, //, %	Multiplication, division, remainder [7]
+x, –x, ~x	NOT (positif, négatif, bit à bit)
**	Puissance [8]
x[indice], x[indice:indice], x(arguments...), x.attribut	indiçage, tranches, appel, référence à un attribut
(expressions...), [expressions...], {key: value...}, `expressions...`	Binding or tuple display, list display, dictionary display, string conversion

Notes

7. L'opérateur % est aussi utilisé pour formater les chaînes de caractères ; il y possède la même priorité.
8. L'opérateur puissance ** est moins prioritaire qu'un opérateur unaire arithmétique ou bit à bit sur sa droite. Ainsi, 2**-1 vaut 0.5.

CHAPITRE 6

Les instructions simples

Simple statements are comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is :

```
simple_stmt   ::=   expression_stmt
                  | assert_stmt
                  | assignment_stmt
                  | augmented_assignment_stmt
                  | pass_stmt
                  | del_stmt
                  | print_stmt
                  | return_stmt
                  | yield_stmt
                  | raise_stmt
                  | break_stmt
                  | continue_stmt
                  | import_stmt
                  | future_stmt
                  | global_stmt
                  | exec_stmt
```

6.1 Les expressions

Les expressions sont utilisées (généralement de manière interactive) comme instructions pour calculer et écrire des valeurs, appeler une procédure (une fonction dont le résultat renvoyé n'a pas d'importance ; en Python, les procédures renvoient la valeur `None`). D'autres utilisations des expressions sont autorisées et parfois utiles. La syntaxe pour une expression en tant qu'instruction est :

```
expression_stmt   ::=   expression_list
```

Ce genre d'instruction évalue la liste d'expressions (qui peut se limiter à une seule expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output (see section *The print statement*) on a line by itself. (Expression statements yielding `None` are not written, so that procedure calls do not cause any output.)

6.2 Les assignations

Les assignations sont utilisées pour lier ou relier des noms à des valeurs et modifier des attributs ou des éléments d'objets muables :

```
assignment_stmt   ::=   (target_list "=")+ (expression_list | yield_expression)
target_list       ::=   target ("," target)* [","]
target            ::=   identifier
                        | "(" target_list ")"
                        | "[" [target_list] "]"
                        | attributeref
                        | subscription
                        | slicing
```

(See section *Primaires* for the syntax definitions for the last three symbols.)

Une assignation évalue la liste d'expressions (gardez en mémoire que ce peut être une simple expression ou une liste dont les éléments sont séparés par des virgules, cette dernière produisant un n-uplet) et assigne l'unique objet résultant à chaque liste cible, de la gauche vers la droite.

Une assignation est définie récursivement en fonction de la forme de la cible (une liste). Quand la cible est une partie d'un objet muable (une référence à un attribut, une sélection ou une tranche), l'objet muable doit effectuer l'assignation au final et décider de sa validité, voire lever une exception si l'assignation n'est pas acceptable. Les règles suivies par les différents types et les exceptions levées sont données dans les définitions des types d'objets (voir la section *Hiérarchie des types standards*).

Assignment of an object to a target list is recursively defined as follows.

- If the target list is a single target : The object is assigned to that target.
- If the target list is a comma-separated list of targets : The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

L'assignation d'un objet vers une cible unique est définie récursivement comme suit.

- Si la cible est une variable (un nom) :
  - If the name does not occur in a `global` statement in the current code block : the name is bound to the object in the current local namespace.
  - Otherwise : the name is bound to the object in the current global namespace.

  Le lien du nom est modifié si le nom était déjà lié. Ceci peut faire que le compteur de références de l'objet auquel le nom était précédemment lié tombe à zéro, entrainant la dé-allocation de l'objet et l'appel de son destructeur (s'il existe).
- If the target is a target list enclosed in parentheses or in square brackets : The object must be an iterable with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.
- Si la cible est une référence à un attribut : l'expression primaire de la référence est évaluée. Elle doit produire un objet avec des attributs que l'on peut assigner : si ce n'est pas le cas, une `TypeError` est levée. Python demande alors à cet objet d'assigner l'attribut donné ; si ce n'est pas possible, une exception est levée (habituellement, mais pas nécessairement, `AttributeError`).

Note : si l'objet est une instance de classe et que la référence à l'attribut apparaît des deux côtés de l'opérateur d'assignation, l'expression « à droite », `a.x` peut accéder soit à l'attribut d'instance ou (si cet attribut d'instance n'existe pas) à l'attribut de classe. L'expression cible « à gauche » `a.x` est toujours définie comme un attribut d'instance, en le créant si nécessaire. Ainsi, les deux occurrences de `a.x` ne font pas nécessairement référence au même attribut : si l'expression « à droite » fait référence à un attribut de classe, l'expression « à gauche » crée un nouvel attribut d'instance comme cible de l'assignation :

```
class Cls:
    x = 3               # class variable
inst = Cls()
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
```

Cette description ne s'applique pas nécessairement aux attributs des descripteurs, telles que les propriétés créées avec `property()`.

— If the target is a subscription : The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.
If the primary is a mutable sequence object (such as a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).
Si la primaire est un objet tableau de correspondances (tel qu'un dictionnaire), la sélection doit être d'un type compatible avec le type des clés ; Python demande alors au tableau de correspondances de créer un couple clé-valeur qui associe la sélection à l'objet assigné. Ceci peut remplacer une correspondance déjà existante pour une clé donnée ou insérer un nouveau couple clé-valeur.

— If the target is a slicing : The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present ; defaults are zero and the sequence's length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

Dans l'implémentation actuelle, la syntaxe pour les cibles est similaire à celle des expressions. Toute syntaxe invalide est rejetée pendant la phase de génération de code, ce qui produit des messages d'erreurs moins détaillés.

WARNING : Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are "safe" (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables are not safe ! For instance, the following program prints `[0, 2]` :

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.2.1 Les assignations augmentées

Une assignation augmentée est la combinaison, dans une seule instruction, d'une opération binaire et d'une assignation :

```
augmented_assignment_stmt   ::=   augtarget augop (expression_list | yield_expression)
augtarget                   ::=   identifier | attributeref | subscription | slicing
augop                       ::=   "+=" | "-=" | "*=" | "/=" | "//=" | "%=" | "**="
                                  | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section *Primaires* for the syntax definitions for the last three symbols.)

Une assignation augmentée évalue la cible (qui, au contraire des assignations normales, ne peut pas être un dépaquetage) et la liste d'expressions, effectue l'opération binaire (spécifique au type d'assignation) sur les deux opérandes et assigne le résultat à la cible originale. La cible n'est évaluée qu'une seule fois.

Une assignation augmentée comme `x += 1` peut être ré-écrite en `x = x + 1` pour obtenir un effet similaire, mais pas exactement équivalent. Dans la version augmentée. `x` n'est évalué qu'une seule fois. Aussi, lorsque c'est possible, l'opération concrète est effectuée *sur place*, c'est-à-dire que plutôt que de créer un nouvel objet et l'assigner à la cible, c'est le vieil objet qui est modifié.

À l'exception de l'assignation de tuples et de cibles multiples dans une seule instruction, l'assignation effectuée par une assignation augmentée est traitée de la même manière qu'une assignation normale. De même, à l'exception du comportement possible *sur place*, l'opération binaire effectuée par assignation augmentée est la même que les opérations binaires normales.

Pour les cibles qui sont des références à des attributs, la même *mise en garde sur les attributs de classe et d'instances* s'applique que pour les assignations normales.

6.3 L'instruction `assert`

Les instructions `assert` sont une manière pratique d'insérer des tests de débogage au sein d'un programme :

```
assert_stmt   ::=   "assert" expression ["," expression]
```

La forme la plus simple, `assert expression`, est équivalente à :

```
if __debug__:
    if not expression: raise AssertionError
```

La forme étendue, `assert expression1, expression2`, est équivalente à :

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

Ces équivalences supposent que `__debug__` et `AssertionError` font référence aux variables natives ainsi nommées. Dans l'implémentation actuelle, la variable native `__debug__` vaut `True` dans des circonstances normales, `False` quand les optimisations sont demandées (ligne de commande avec l'option *-O*). Le générateur de code actuel ne produit aucun code pour une instruction `assert` quand vous demandez les optimisations à la compilation. Notez qu'il est superflu d'inclure le code source dans le message d'erreur pour l'expression qui a échoué : il est affiché dans la pile d'appels.

Assigner vers `__debug__` est illégal. La valeur de cette variable native est déterminée au moment où l'interpréteur démarre.

6.4 L'instruction `pass`

```
pass_stmt   ::=   "pass"
```

pass est une opération vide — quand elle est exécutée, rien ne se passe. Elle est utile comme bouche-trou lorsqu'une instruction est syntaxiquement requise mais qu'aucun code ne doit être exécuté. Par exemple :

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

6.5 L'instruction `del`

```
del_stmt   ::=   "del" target_list
```

La suppression est récursivement définie de la même manière que l'assignation. Plutôt que de détailler cela de manière approfondie, voici quelques indices.

La suppression d'une liste cible (*target_list* dans la grammaire ci-dessus) supprime récursivement chaque cible, de la gauche vers la droite.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a *global* statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

It is illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

La suppression d'une référence à un attribut, une sélection ou une tranche est passée à l'objet primaire concerné : la suppression d'une tranche est en général équivalente à l'assignation d'une tranche vide du type adéquat (mais ceci est au final déterminé par l'objet que l'on tranche).

6.6 The `print` statement

```
print_stmt   ::=   "print" ([expression ("," expression)* [","]]
                   | ">>" expression [("," expression)+ [","]])
```

print evaluates each expression in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case (1) when no characters have yet been written to standard output, (2) when the last character written to standard output is a whitespace character except `' '`, or (3) when the last write operation on standard output was not a *print* statement. (In some cases it may be functional to write an empty string to standard output for this reason.)

Note : Objects which act like file objects but which are not the built-in file objects often do not properly emulate this aspect of the file object's behavior, so it is best not to rely on this.

A `'\n'` character is written at the end, unless the *print* statement ends with a comma. This is the only action if the statement contains just the keyword *print*.

Standard output is defined as the file object named `stdout` in the built-in module `sys`. If no such object exists, or if it does not have a `write()` method, a `RuntimeError` exception is raised.

print also has an extended form, defined by the second portion of the syntax described above. This form is sometimes referred to as « *print* chevron. » In this form, the first expression after the >> must evaluate to a « file-like » object, specifically an object that has a `write()` method as described above. With this extended form, the subsequent expressions are printed to this file object. If the first expression evaluates to `None`, then `sys.stdout` is used as the file for output.

6.7 L'instruction `return`

```
return_stmt   ::=   "return" [expression_list]
```

return ne peut être placée qu'à l'intérieur d'une définition de fonction, pas à l'intérieur d'une définition de classe.

Si une liste d'expressions (*expression_list* dans la grammaire ci-dessus) est présente, elle est évaluée, sinon `None` est utilisée comme valeur par défaut.

return quitte l'appel à la fonction courante avec la liste d'expressions (ou `None`) comme valeur de retour.

Quand *return* fait sortir d'une instruction *try* avec une clause *finally*, cette clause *finally* est exécutée avant de réellement quitter la fonction.

In a generator function, the *return* statement is not allowed to include an *expression_list*. In that context, a bare *return* indicates that the generator is done and will cause `StopIteration` to be raised.

6.8 L'instruction `yield`

```
yield_stmt   ::=   yield_expression
```

The *yield* statement is only used when defining a generator function, and is only used in the body of the generator function. Using a *yield* statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator's *next()* method repeatedly until it raises an exception.

When a *yield* statement is executed, the state of the generator is frozen and the value of *expression_list* is returned to *next()*'s caller. By « frozen » we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack : enough information is saved so that the next time *next()* is invoked, the function can proceed exactly as if the *yield* statement were just another external call.

As of Python version 2.5, the *yield* statement is now allowed in the *try* clause of a *try* ... *finally* construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's `close()` method will be called, allowing any pending *finally* clauses to execute.

Pour tous les détails sur la sémantique de *yield*, reportez-vous à la section *Expressions yield*.

Note : In Python 2.2, the *yield* statement was only allowed when the `generators` feature has been enabled. This `__future__` import statement was used to enable the feature :

```
from __future__ import generators
```

Voir aussi :

PEP 255 : Générateurs simples La proposition d'ajouter à Python des générateurs et l'instruction *yield*.

PEP 342 – Coroutines *via* des générateurs améliorés The proposal that, among other generator enhancements, proposed allowing *yield* to appear inside a *try* … *finally* block.

6.9 L'instruction `raise`

```
raise_stmt  ::=  "raise" [expression ["," expression ["," expression]]]
```

If no expressions are present, *raise* re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `TypeError` exception is raised indicating that this is an error (if running under IDLE, a `Queue.Empty` exception is raised instead).

Otherwise, *raise* evaluates the expressions to get three objects, using `None` as the value of omitted expressions. The first two objects are used to determine the *type* and *value* of the exception.

If the first object is an instance, the type of the exception is the class of the instance, the instance itself is the value, and the second object must be `None`.

If the first object is a class, it becomes the type of the exception. The second object is used to determine the exception value : If it is an instance of the class, the instance becomes the exception value. If the second object is a tuple, it is used as the argument list for the class constructor ; if it is `None`, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If a third object is present and not `None`, it must be a traceback object (see section *Hiérarchie des types standards*), and it is substituted instead of the current location as the place where the exception occurred. If the third object is present and not a traceback object or `None`, a `TypeError` exception is raised. The three-expression form of *raise* is useful to re-raise an exception transparently in an except clause, but *raise* with no expressions should be preferred if the exception to be re-raised was the most recently active exception in the current scope.

Des informations complémentaires sur les exceptions sont disponibles dans la section *Exceptions* et sur la gestion des exceptions dans la section *L'instruction try*.

6.10 L'instruction `break`

```
break_stmt  ::=  "break"
```

Une instruction *break* ne peut apparaître qu'à l'intérieur d'une boucle *for* ou *while*, mais pas dans une définition de fonction ou de classe à l'intérieur de cette boucle.

Elle termine la boucle la plus imbriquée, shuntant l'éventuelle clause *else* de la boucle.

Si une boucle *for* est terminée par un *break*, la cible qui contrôle la boucle garde sa valeur.

Quand *break* passe le contrôle en dehors d'une instruction *try* qui comporte une clause *finally*, cette clause *finally* est exécutée avant de quitter la boucle.

6.11 L'instruction continue

```
continue_stmt   ::=   "continue"
```

L'instruction *continue* ne peut apparaître qu'à l'intérieur d'une boucle *for* ou *while*, mais pas dans une définition de fonction ou de classe ni dans une clause *finally*, à l'intérieur de cette boucle. Elle fait continuer le flot d'exécution au prochain cycle de la boucle la plus imbriquée.

Quand *continue* passe le contrôle en dehors d'une instruction *try* qui comporte une clause *finally*, cette clause *finally* est exécutée avant de commencer le cycle suivant de la boucle.

6.12 L'instruction import

```
import_stmt       ::=   "import" module ["as" name] ( "," module ["as" name] )*
                        | "from" relative_module "import" identifier ["as" name]
                        ( "," identifier ["as" name] )*
                        | "from" relative_module "import" "(" identifier ["as" name]
                        ( "," identifier ["as" name] )* [","] ")"
                        | "from" module "import" "*"
module            ::=   (identifier ".")* identifier
relative_module   ::=   "."* module | "."+
name              ::=   identifier
```

Import statements are executed in two steps : (1) find a module, and initialize it if necessary ; (2) define a name or names in the local namespace (of the scope where the *import* statement occurs). The statement comes in two forms differing on whether it uses the *from* keyword. The first form (without *from*) repeats these steps for each identifier in the list. The form with *from* performs step (1) once, and then performs step (2) repeatedly.

To understand how step (1) occurs, one must first understand how Python handles hierarchical naming of modules. To help organize modules and provide a hierarchy in naming, Python has a concept of packages. A package can contain other packages and modules while modules cannot contain other modules or packages. From a file system perspective, packages are directories and modules are files.

Once the name of the module is known (unless otherwise specified, the term « module » will refer to both packages and modules), searching for the module or package can begin. The first place checked is `sys.modules`, the cache of all modules that have been imported previously. If the module is found there then it is used in step (2) of import.

If the module is not found in the cache, then `sys.meta_path` is searched (the specification for `sys.meta_path` can be found in **PEP 302**). The object is a list of *finder* objects which are queried in order as to whether they know how to load the module by calling their `find_module()` method with the name of the module. If the module happens to be contained within a package (as denoted by the existence of a dot in the name), then a second argument to `find_module()` is given as the value of the `__path__` attribute from the parent package (everything up to the last dot in the name of the module being imported). If a finder can find the module it returns a *loader* (discussed later) or returns `None`.

If none of the finders on `sys.meta_path` are able to find the module then some implicitly defined finders are queried. Implementations of Python vary in what implicit meta path finders are defined. The one they all do define, though, is one

that handles sys.path_hooks, sys.path_importer_cache, and sys.path.

The implicit finder searches for the requested module in the « paths » specified in one of two places (« paths » do not have to be file system paths). If the module being imported is supposed to be contained within a package then the second argument passed to find_module(), __path__ on the parent package, is used as the source of paths. If the module is not contained in a package then sys.path is used as the source of paths.

Once the source of paths is chosen it is iterated over to find a finder that can handle that path. The dict at sys.path_importer_cache caches finders for paths and is checked for a finder. If the path does not have a finder cached then sys.path_hooks is searched by calling each object in the list with a single argument of the path, returning a finder or raises ImportError. If a finder is returned then it is cached in sys.path_importer_cache and then used for that path entry. If no finder can be found but the path exists then a value of None is stored in sys.path_importer_cache to signify that an implicit, file-based finder that handles modules stored as individual files should be used for that path. If the path does not exist then a finder which always returns None is placed in the cache for the path.

If no finder can find the module then ImportError is raised. Otherwise some finder returned a loader whose load_module() method is called with the name of the module to load (see **PEP 302** for the original definition of loaders). A loader has several responsibilities to perform on a module it loads. First, if the module already exists in sys.modules (a possibility if the loader is called outside of the import machinery) then it is to use that module for initialization and not a new module. But if the module does not exist in sys.modules then it is to be added to that dict before initialization begins. If an error occurs during loading of the module and it was added to sys.modules it is to be removed from the dict. If an error occurs but the module was already in sys.modules it is left in the dict.

The loader must set several attributes on the module. __name__ is to be set to the name of the module. __file__ is to be the « path » to the file unless the module is built-in (and thus listed in sys.builtin_module_names) in which case the attribute is not set. If what is being imported is a package then __path__ is to be set to a list of paths to be searched when looking for modules and packages contained within the package being imported. __package__ is optional but should be set to the name of package that contains the module or package (the empty string is used for module not contained in a package). __loader__ is also optional but should be set to the loader object that is loading the module.

If an error occurs during loading then the loader raises ImportError if some other exception is not already being propagated. Otherwise the loader returns the module that was loaded and initialized.

When step (1) finishes without raising an exception, step (2) can begin.

The first form of *import* statement binds the module name in the local namespace to the module object, and then goes on to import the next identifier, if any. If the module name is followed by *as*, the name following *as* is used as the local name for the module.

The *from* form does not bind the module name : it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local namespace to the object thus found. As with the first form of *import*, an alternate local name can be supplied by specifying « *as* localname ». If a name is not found, ImportError is raised. If the list of identifiers is replaced by a star ('*'), all public names defined in the module are bound in the local namespace of the *import* statement..

The *public names* defined by a module are determined by checking the module's namespace for a variable named __all__; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in __all__ are all considered public and are required to exist. If __all__ is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character ('_'). __all__ should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The *from* form with * may only occur in a module scope. If the wild card form of import — import * — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a SyntaxError.

Quand vous spécifiez les modules à importer, vous n'avez pas besoin de spécifier les noms absolus des modules. Quand un module ou un paquet est contenu dans un autre paquet, il est possible d'effectuer une importation relative à l'intérieur du

même paquet de plus haut niveau sans avoir à mentionner le nom du paquet. En utilisant des points en entête du module ou du paquet spécifié après *from*, vous pouvez spécifier combien de niveaux vous souhaitez remonter dans la hiérarchie du paquet courant sans spécifier de nom exact. Un seul point en tête signifie le paquet courant où se situe le module qui effectue l'importation. Deux points signifient de remonter d'un niveau. Trois points, remonter de deux niveaux et ainsi de suite. Ainsi, si vous exécutez `from . import mod` dans un module du paquet `pkg`, vous importez finalement `pkg.mod`. Et si vous exécutez `from ..souspkg2 import mod` depuis `pkg.souspkg1`, vous importez finalement `pkg.souspkg2.mod`. La spécification des importations relatives se situe dans la **PEP 328**.

`importlib.import_module()` is provided to support applications that determine which modules need to be loaded dynamically.

6.12.1 L'instruction future

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_statement   ::=   "from" "__future__" "import" feature ["as" name]
                         ("," feature ["as" name])*
                         | "from" "__future__" "import" "(" feature ["as" name]
                         ("," feature ["as" name])* [","] ")"
feature            ::=   identifier
name               ::=   identifier
```

Une instruction *future* doit apparaître en haut du module. Les seules lignes autorisées avant une instruction *future* sont :

- la chaîne de documentation du module (si elle existe),
- des commentaires,
- des lignes vides et
- d'autres instructions *future*.

The features recognized by Python 2.6 are `unicode_literals`, `print_function`, `absolute_import`, `division`, `generators`, `nested_scopes` and `with_statement`. `generators`, `with_statement`, `nested_scopes` are redundant in Python version 2.6 and above because they are always enabled.

Une instruction *future* est reconnue et traitée spécialement au moment de la compilation : les modifications à la sémantique des constructions de base sont souvent implémentées en générant un code différent. Il peut même arriver qu'une nouvelle fonctionnalité ait une syntaxe incompatible (tel qu'un nouveau mot réservé) ; dans ce cas, le compilateur a besoin d'analyser le module de manière différente. De telles décisions ne peuvent pas être différées au moment de l'exécution.

Pour une version donnée, le compilateur sait quelles fonctionnalités ont été définies et lève une erreur à la compilation si une instruction *future* contient une fonctionnalité qui lui est inconnue.

La sémantique à l'exécution est la même que pour toute autre instruction d'importation : il existe un module standard `__future__`, décrit plus loin, qui est importé comme les autres au moment ou l'instruction *future* est exécutée.

La sémantique particulière à l'exécution dépend des fonctionnalités apportées par l'instruction *future*.

Notez que l'instruction suivante est tout à fait normale :

```
import __future__ [as name]
```

Ce n'est pas une instruction *future* ; c'est une instruction d'importation ordinaire qui n'a aucune sémantique particulière ou restriction de syntaxe.

Code compiled by an *exec* statement or calls to the built-in functions `compile()` and `execfile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future state-

ment. This can, starting with Python 2.2 be controlled by optional arguments to `compile()` — see the documentation of that function for details.

Une instruction *future* entrée à l'invite de l'interpréteur interactif est effective pour le reste de la session de l'interpréteur. Si l'interpréteur est démarré avec l'option `-i`, qu'un nom de script est passé pour être exécuté et que ce script contient une instruction *future*, elle est effective pour la session interactive qui démarre après l'exécution du script.

Voir aussi :

PEP 236 – retour vers le __future__ La proposition originale pour le mécanisme de __future__.

6.13 L'instruction `global`

```
global_stmt   ::=   "global" identifier ("," identifier)*
```

L'instruction *global* est une déclaration qui couvre l'ensemble du bloc de code courant. Elle signifie que les noms (*identifier* dans la grammaire ci-dessus) listés doivent être interprétés comme globaux. Il est impossible d'assigner une variable globale sans *global*, mais rappelez-vous que les variables libres peuvent faire référence à des variables globales sans avoir été déclarées en tant que telles.

Les noms listés dans l'instruction *global* ne doivent pas être utilisés, dans le même bloc de code, avant l'instruction *global*.

Names listed in a *global* statement must not be defined as formal parameters or in a *for* loop control target, *class* definition, function definition, or *import* statement.

CPython implementation detail : The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

Programmer's note : *global* is a directive to the parser. It applies only to code parsed at the same time as the *global* statement. In particular, a *global* statement contained in an *exec* statement does not affect the code block *containing* the *exec* statement, and code contained in an *exec* statement is unaffected by *global* statements in the code containing the *exec* statement. The same applies to the `eval()`, `execfile()` and `compile()` functions.

6.14 The `exec` statement

```
exec_stmt   ::=   "exec" or_expr ["in" expression ["," expression]]
```

This statement supports dynamic execution of Python code. The first expression should evaluate to either a Unicode string, a *Latin-1* encoded string, an open file object, a code object, or a tuple. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed. For the interpretation of a tuple, see below. In all cases, the code that's executed is expected to be valid as file input (see section *Fichier d'entrée*). Be aware that the *return* and *yield* statements may not be used outside of function definitions even within the context of code passed to the *exec* statement.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are given as *globals*

1. Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use *universal newlines* mode to convert Windows or Mac-style newlines.

and *locals*, the code will be executed as if it were embedded in a class definition.

The first expression may also be a tuple of length 2 or 3. In this case, the optional parts must be omitted. The form `exec(expr, globals)` is equivalent to `exec expr in globals`, while the form `exec(expr, globals, locals)` is equivalent to `exec expr in globals, locals`. The tuple form of `exec` provides compatibility with Python 3, where `exec` is a function rather than a statement.

Modifié dans la version 2.4 : Formerly, *locals* was required to be a dictionary.

As a side effect, an implementation may insert additional keys into the dictionaries given besides those corresponding to variable names set by the executed code. For example, the current implementation may add a reference to the dictionary of the built-in module `__builtin__` under the key `__builtins__` (!).

Programmer's hints : dynamic evaluation of expressions is supported by the built-in function `eval()`. The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use by *exec*.

Notes

CHAPITRE 7

Instructions composées

Les instructions composées contiennent d'autres (groupes d') instructions ; elles affectent ou contrôlent l'exécution de ces autres instructions d'une manière ou d'une autre. En général, une instruction composée couvre plusieurs lignes bien que, dans sa forme la plus simple, une instruction composée peut tenir sur une seule ligne.

The *if*, *while* and *for* statements implement traditional control flow constructs. *try* specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more "clauses." A clause consists of a header and a "suite." The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements ; the following is illegal, mostly because it wouldn't be clear to which *if* clause a following *else* clause would belong :

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the *print* statements are executed :

```
if x < y < z: print x; print y; print z
```

En résumé :

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | funcdef
                    | classdef
                    | decorated
suite           ::=   stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=   stmt_list NEWLINE | compound_stmt
```

```
stmt_list       ::=  simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a NEWLINE possibly followed by a DEDENT. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the “dangling *else*” problem is solved in Python by requiring nested *if* statements to be indented).

L’agencement des règles de grammaire dans les sections qui suivent place chaque clause sur une ligne séparée pour plus de clarté.

7.1 L’instruction `if`

L’instruction *if* est utilisée pour exécuter des instructions en fonction d’une condition :

```
if_stmt   ::=  "if" expression ":" suite
               ( "elif" expression ":" suite )*
               ["else" ":" suite]
```

Elle sélectionne exactement une des suites en évaluant les expressions une par une jusqu’à ce qu’une soit vraie (voir la section *Opérations booléennes* pour la définition de vrai et faux) ; ensuite cette suite est exécutée (et aucune autre partie de l’instruction *if* n’est exécutée ou évaluée). Si toutes les expressions sont fausses, la suite de la clause *else*, si elle existe, est exécutée.

7.2 L’instruction `while`

L’instruction *while* est utilisée pour exécuter des instructions de manière répétée tant qu’une expression est vraie :

```
while_stmt   ::=  "while" expression ":" suite
                  ["else" ":" suite]
```

Python évalue l’expression de manière répétée et, tant qu’elle est vraie, exécute la première suite ; si l’expression est fausse (ce qui peut arriver même lors du premier test), la suite de la clause *else*, si elle existe, est exécutée et la boucle se termine.

Une instruction *break* exécutée dans la première suite termine la boucle sans exécuter la suite de la clause *else*. Une instruction *continue* exécutée dans la première suite saute le reste de la suite et retourne au test de l’expression.

7.3 L’instruction `for`

L’instruction *for* est utilisée pour itérer sur les éléments d’une séquence (par exemple une chaîne, un tuple ou une liste) ou un autre objet itérable :

```
for_stmt   ::=  "for" target_list "in" expression_list ":" suite
                ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite

is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the *else* clause, if present, is executed, and the loop terminates.

A *break* statement executed in the first suite terminates the loop without executing the *else* clause's suite. A *continue* statement executed in the first suite skips the rest of the suite and continues with the next item, or with the *else* clause if there was no next item.

The suite may assign to the variable(s) in the target list ; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint : the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `range(3)` returns the list `[0, 1, 2]`.

Note : There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4 L'instruction `try`

L'instruction *try* spécifie les gestionnaires d'exception ou le code de nettoyage pour un groupe d'instructions :

```
try_stmt    ::=   try1_stmt | try2_stmt
try1_stmt   ::=   "try" ":" suite
                  ("except" [expression [("as" | ",") identifier]] ":" suite)+
                  ["else" ":" suite]
                  ["finally" ":" suite]
try2_stmt   ::=   "try" ":" suite
                  "finally" ":" suite
```

Modifié dans la version 2.5 : In previous versions of Python, *try*...*except*...*finally* did not work. *try*...*except* had to be nested in *try*...*finally*.

The *except* clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the *try* suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last ; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is « compatible » with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, or a tuple containing an item compatible with the exception.

Si aucune clause `except` ne correspond à l'exception, la recherche d'un gestionnaire d'exception se poursuit dans le code englobant et dans la pile d'appels [1].

1. L'exception est propagée à la pile d'appels à moins qu'il n'y ait une clause *finally* qui lève une autre exception, ce qui entraîne la perte de l'ancienne exception. Cette nouvelle exception entraîne la perte pure et simple de l'ancienne.

Si l'évaluation d'une expression dans l'en-tête d'une clause `except` lève une exception, la recherche initiale d'un gestionnaire est annulée et une recherche commence pour la nouvelle exception dans le code englobant et dans la pile d'appels (c'est traité comme si l'instruction *try* avait levé l'exception).

When a matching except clause is found, the exception is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause's suite is executed, details about the exception are assigned to three variables in the `sys` module : `sys.exc_type` receives the object identifying the exception ; `sys.exc_value` receives the exception's parameter ; `sys.exc_traceback` receives a traceback object (see section *Hiérarchie des types standards*) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(exc_type, exc_value, exc_traceback)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional *else* clause is executed if the control flow leaves the *try* suite, no exception was raised, and no *return*, *continue*, or *break* statement was executed. Exceptions in the *else* clause are not handled by the preceding *except* clauses.

If *finally* is present, it specifies a "cleanup" handler. The *try* clause is executed, including any *except* and *else* clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The *finally* clause is executed. If there is a saved exception, it is re-raised at the end of the *finally* clause. If the *finally* clause raises another exception or executes a *return* or *break* statement, the saved exception is discarded :

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

L'information relative à l'exception n'est pas disponible pour le programme pendant l'exécution de la clause *finally*.

Lorsqu'une instruction *return*, *break* ou *continue* est exécutée dans la suite d'une instruction *try* d'une construction *try*…*finally*, la clause *finally* est aussi exécutée à la sortie. Une instruction *continue* est illégale dans une clause *finally* (la raison est que l'implémentation actuelle pose problème — il est possible que cette restriction soit levée dans le futur).

La valeur de retour d'une fonction est déterminée par la dernière instruction *return* exécutée. Puisque la clause *finally* s'exécute toujours, une instruction *return* exécutée dans le *finally* sera toujours la dernière clause exécutée :

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Vous trouvez des informations supplémentaires relatives aux exceptions dans la section *Exceptions* et, dans la section *L'instruction raise*, des informations relatives à l'utilisation de l'instruction *raise* pour produire des exceptions.

7.5 L'instruction `with`

Nouveau dans la version 2.5.

L'instruction *`with`* est utilisée pour encapsuler l'exécution d'un bloc avec des méthodes définies par un gestionnaire de contexte (voir la section *Gestionnaire de contexte With*). Cela permet d'utiliser de manière simple le patron de conception classique *`try`*...*`except`*...*`finally`*.

```
with_stmt   ::=   "with" with_item (","  with_item)* ":" suite
with_item   ::=   expression ["as" target]
```

L'exécution de l'instruction *`with`* avec un seul « élément » (*item* dans la grammaire) se déroule comme suit :

1. L'expression de contexte (l'expression donnée dans le *`with_item`*) est évaluée pour obtenir un gestionnaire de contexte.
2. La méthode *`__exit__()`* du gestionnaire de contexte est chargée pour une utilisation ultérieure.
3. La méthode *`__enter__()`* du gestionnaire de contexte est invoquée.
4. Si une cible (*target* dans la grammaire ci-dessus) a été incluse dans l'instruction *`with`*, la valeur de retour de *`__enter__()`* lui est assignée.

   Note : L'instruction *`with`* garantit que si la méthode *`__enter__()`* se termine sans erreur, alors la méthode *`__exit__()`* est toujours appelée. Ainsi, si une erreur se produit pendant l'assignation à la liste cible, elle est traitée de la même façon qu'une erreur se produisant dans la suite. Voir l'étape 6 ci-dessous.

   ---

5. La suite est exécutée.
6. The context manager's *`__exit__()`* method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to *`__exit__()`*. Otherwise, three `None` arguments are supplied.
   If the suite was exited due to an exception, and the return value from the *`__exit__()`* method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the *`with`* statement.
   Si l'on est sorti de la suite pour une raison autre qu'une exception, la valeur de retour de *`__exit__()`* est ignorée et l'exécution se poursuit à l'endroit normal pour le type de sortie prise.

Avec plus d'un élément, les gestionnaires de contexte sont traités comme si plusieurs instructions *`with`* étaient imbriquées :

```
with A() as a, B() as b:
    suite
```

est équivalente à :

```
with A() as a:
    with B() as b:
        suite
```

Note : In Python 2.5, the *`with`* statement is only allowed when the `with_statement` feature has been enabled. It is always enabled in Python 2.6.

Modifié dans la version 2.7 : Prise en charge de multiples expressions de contexte.

Voir aussi :

PEP 343 - The « with » statement La spécification, les motivations et des exemples de l'instruction *`with`* en Python.

7.6 Définition de fonctions

Une définition de fonction définit un objet fonction allogène (voir la section *Hiérarchie des types standards*) :

```
decorated       ::=  decorators (classdef | funcdef)
decorators      ::=  decorator+
decorator       ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
funcdef         ::=  "def" funcname "(" [parameter_list] ")" ":" suite
dotted_name     ::=  identifier ("." identifier)*
parameter_list  ::=  (defparameter ",")*
                     ( "*" identifier ["," "**" identifier]
                     | "**" identifier
                     | defparameter [","] )
defparameter    ::=  parameter ["=" expression]
sublist         ::=  parameter ("," parameter)* [","]
parameter       ::=  identifier | "(" sublist ")"
funcname        ::=  identifier
```

Une définition de fonction est une instruction qui est exécutée. Son exécution lie le nom de la fonction, dans l'espace de noms local courant, à un objet fonction (un objet qui encapsule le code exécutable de la fonction). Cet objet fonction contient une référence à l'espace des noms globaux courant comme espace des noms globaux à utiliser lorsque la fonction est appelée.

La définition de la fonction n'exécute pas le corps de la fonction ; elle n'est exécutée que lorsque la fonction est appelée [2].

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code :

```
@f1(arg)
@f2
def func(): pass
```

est équivalente à :

```
def func(): pass
func = f1(arg)(f2(func))
```

When one or more top-level *parameters* have the form *parameter* = *expression*, the function is said to have « default parameter values. » For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same « pre-computed » value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary : if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g. :

2. Une chaîne littérale apparaissant comme première instruction dans le corps de la fonction est transformée en attribut `__doc__` de la fonction et donc en *docstring* de la fonction.

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Appels*. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form « `*identifier` » is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form « `**identifier` » is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section *Expressions lambda*. Note that the lambda expression is merely a shorthand for a simplified function definition ; a function defined in a « *def* » statement can be passed around or assigned to another name just like a function defined by a lambda expression. The « *def* » form is actually more powerful since it allows the execution of multiple statements.

Programmer's note : Functions are first-class objects. A « `def` » form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section *Noms et liaisons* for details.

7.7 Définition de classes

Une définition de classe définit un objet classe (voir la section *Hiérarchie des types standards*) :

```
classdef      ::=  "class" classname [inheritance] ":" suite
inheritance   ::=  "(" [expression_list] ")"
classname     ::=  identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section *Noms et liaisons*), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. [3] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

Programmer's note : Variables defined in the class definition are class variables ; they are shared by all instances. To create instance variables, they can be set in a method with `self.name = value`. Both class and instance variables are accessible through the notation « `self.name` », and an instance variable hides a class variable with the same name when accessed in this way. Class variables can be used as defaults for instance variables, but using mutable values there can lead to unexpected results. For *new-style class*es, descriptors can be used to create instance variables with different implementation details.

Class definitions, like function definitions, may be wrapped by one or more *decorator* expressions. The evaluation rules for the decorator expressions are the same as for functions. The result must be a class object, which is then bound to the class name.

3. Une chaîne littérale apparaissant comme première instruction dans le corps de la classe est transformée en élément `__doc__` de l'espace de noms et donc en *docstring* de la classe.

Notes

CHAPITRE 8

Composants de plus haut niveau

L'entrée de l'interpréteur Python peut provenir d'un certain nombre de sources : d'un script passé en entrée standard ou en argument de programme, tapée de manière interactive, à partir d'un fichier source de module, etc. Ce chapitre donne la syntaxe utilisée dans ces différents cas.

8.1 Programmes Python complets

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment : all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `__builtin__` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

La syntaxe d'un programme Python complet est celle d'un fichier d'entrée, dont la description est donnée dans la section suivante.

L'interpréteur peut également être invoqué en mode interactif ; dans ce cas, il ne lit et n'exécute pas un programme complet mais lit et exécute une seule instruction (éventuellement composée) à la fois. L'environnement initial est identique à celui d'un programme complet ; chaque instruction est exécutée dans l'espace de noms de `__main__`.

A complete program can be passed to the interpreter in three forms : with the `-c` *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode ; otherwise, it executes the file as a complete program.

8.2 Fichier d'entrée

Toutes les entrées lues à partir de fichiers non interactifs sont de la même forme :

```
file_input   ::=   (NEWLINE | statement)*
```

Cette syntaxe est utilisée dans les situations suivantes :

- lors de l'analyse d'un programme Python complet (à partir d'un fichier ou d'une chaîne de caractères) ;
- lors de l'analyse d'un module ;
- when parsing a string passed to the `exec` statement ;

8.3 Entrée interactive

L'entrée en mode interactif est analysée à l'aide de la grammaire suivante :

```
interactive_input   ::=   [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Notez qu'une instruction composée (de niveau supérieur) doit être suivie d'une ligne blanche en mode interactif ; c'est nécessaire pour aider l'analyseur à détecter la fin de l'entrée.

8.4 Entrée d'expression

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` must have the following form :

```
eval_input   ::=   expression_list NEWLINE*
```

The input line read by `input()` must have the following form :

```
input_input   ::=   expression_list NEWLINE
```

Note : to read "raw" input line without interpretation, you can use the built-in function `raw_input()` or the `readline()` method of file objects.

CHAPITRE 9

Spécification complète de la grammaire

Ceci est la grammaire de Python, exhaustive, telle qu'elle est lue par le générateur d'analyseur, et utilisée pour analyser des fichiers sources en Python :

```
# Grammar for Python

# Note:  Changing the grammar specified in this file will most likely
#        require corresponding changes in the parser module
#        (../Modules/parsermodule.c).  If you can't make the changes to
#        that module yourself, please co-ordinate the required changes
#        with someone who can; ask around on python-dev for help.  Fred
#        Drake <fdrake@acm.org> will probably be listening there.

# NOTE WELL: You should also follow all the steps listed in PEP 306,
# "How to Change Python's Grammar"

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() and input() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')'
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
fpdef: NAME | '(' fplist ')'
fplist: fpdef (',' fpdef)* [',']
```

(suite sur la page suivante)

(suite de la page précédente)

```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal assignments, additional restrictions enforced by the interpreter
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test [',' test [',' test]]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' ('.'* dotted_name | '.'+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
exec_stmt: 'exec' expr ['in' test [',' test]]
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test [('as' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

# Backward compatibility cruft to support:
# [ x for x in lambda: True, lambda: False if x() ]
# even while also allowing:
# lambda x: 5 if x else 2
# (But not a mix of the two)
testlist_safe: old_test [(',' old_test)+ [',']]
old_test: or_test | old_lambdef
old_lambdef: 'lambda' [varargslist] ':' old_test
```

(suite sur la page suivante)

(suite de la page précédente)

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
listmaker: test ( list_for | (',' test)* [','] )
testlist_comp: test ( comp_for | (',' test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [',')) |
                  (test (comp_for | (',' test)* [','])) )

classdef: 'class' NAME ['(' [testlist] ')'] ':' suite

arglist: (argument ',')* (argument [',']
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
argument: test [comp_for] | test '=' test

list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]

comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]

testlist1: test (',' test)*

# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [testlist]
```

ANNEXE A

Glossaire

>>> 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 *Appels* 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 *Implémentation de descripteurs*.

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, elles 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 cancane, 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 *LBYL* 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 *statement*s 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 *Définition de fonctions*.

__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 appelable 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 clef pour maîtriser comment les éléments dont 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 clef pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clef au besoin avec des expressions *lambda*, comme `lambda r: (r[0], r[2])`. Finalement le module `operator` fournit des constructeurs de fonctions clef : `attrgetter()`, `itemgetter()`, et `methodcaller()`. Voir Comment Trier pour avoir des exemple 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 devant 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 séquencement 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.

magic method An informal synonym for *special method*.

Tableau de correspondances Un conteneur permettant d'accéder à des éléments par clef 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`.

métaclasse 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 métaclasse 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 : *Personnalisation de la création de classes*.

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 *`__getattribute__()`*.

More information can be found in *New-style and classic classes*.

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 *Définition de fonctions* 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 *Méthodes spéciales*.

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 similiar 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.

ANNEXE B

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 !

ANNEXE C

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 Viriginie, d'où il diffusa plusieurs versions du logiciel.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation ; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

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 2.7.18

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 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 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 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 2.7.18 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
   2.7.18.

4. PSF is making Python 2.7.18 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 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, 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 2.7.18, 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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,
   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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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).
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.

 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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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:

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,
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

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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:

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 MERCHANT-
ABILITY 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
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, contiens 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
 * 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.

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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
   *    "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
   * 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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
   *    "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 :-).
   * 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        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

ANNEXE D

Copyright

Python et cette documentation sont :

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

Index

Non alphabétique

A

B

C

D

E

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z