The Python Language Reference

Version 3.12.3

Guido van Rossum and the Python development team

mai 17, 2024

Python Software Foundation
Email : docs@python.org

Table des matières

1 Introduction **3**
- 1.1 Autres implémentations . . . 3
- 1.2 Notations . . . 4

2 Analyse lexicale **5**
- 2.1 Structure des lignes . . . 5
  - 2.1.1 Lignes logiques . . . 5
  - 2.1.2 Lignes physiques . . . 5
  - 2.1.3 Commentaires . . . 6
  - 2.1.4 Déclaration d'encodage . . . 6
  - 2.1.5 Continuation de ligne explicite . . . 6
  - 2.1.6 Continuation de ligne implicite . . . 7
  - 2.1.7 Lignes vierges . . . 7
  - 2.1.8 Indentation . . . 7
  - 2.1.9 Espaces entre lexèmes . . . 8
- 2.2 Autres lexèmes . . . 8
- 2.3 Identifiants et mots-clés . . . 8
  - 2.3.1 Mots-clés . . . 9
  - 2.3.2 Mots-clés ad hoc . . . 9
  - 2.3.3 Classes réservées pour les identifiants . . . 10
- 2.4 Littéraux . . . 10
  - 2.4.1 Littéraux de chaînes de caractères et de suites d'octets . . . 10
  - 2.4.2 Concaténation de chaînes de caractères . . . 13
  - 2.4.3 f-strings . . . 13
  - 2.4.4 Littéraux numériques . . . 15
  - 2.4.5 Entiers littéraux . . . 15
  - 2.4.6 Nombres à virgule flottante littéraux . . . 16
  - 2.4.7 Imaginaires littéraux . . . 16
- 2.5 Opérateurs . . . 16
- 2.6 Délimiteurs . . . 17

3 Modèle de données **19**
- 3.1 Objets, valeurs et types . . . 19
- 3.2 Hiérarchie des types standards . . . 20
  - 3.2.1 `None` . . . 20
  - 3.2.2 NotImplemented . . . 20
  - 3.2.3 Ellipse . . . 21
  - 3.2.4 `numbers.Number` . . . 21
  - 3.2.5 Séquences . . . 22
  - 3.2.6 Ensembles . . . 23
  - 3.2.7 Tableaux de correspondances . . . 23

3.2.8 Types appelables . . . 24
3.2.9 Modules . . . 28
3.2.10 Classes déclarées par le développeur . . . 28
3.2.11 Instances de classe . . . 29
3.2.12 Objets entrées-sorties (ou objets fichiers) . . . 29
3.2.13 Types internes . . . 29
3.3 Méthodes spéciales . . . 34
3.3.1 Personnalisation de base . . . 35
3.3.2 Personnalisation de l'accès aux attributs . . . 38
3.3.3 Personnalisation de la création de classes . . . 42
3.3.4 Personnalisation des instances et vérification des sous-classes . . . 45
3.3.5 Émulation de types génériques . . . 46
3.3.6 Émulation d'objets appelables . . . 48
3.3.7 Émulation de types conteneurs . . . 48
3.3.8 Émulation de types numériques . . . 50
3.3.9 Gestionnaire de contexte With . . . 52
3.3.10 Arguments positionnels dans le filtrage par motif sur les classes . . . 52
3.3.11 Emulating buffer types . . . 53
3.3.12 Recherche des méthodes spéciales . . . 53
3.4 Coroutines . . . 54
3.4.1 Objets *attendables* (*awaitable*) . . . 54
3.4.2 Objets coroutines . . . 55
3.4.3 Itérateurs asynchrones . . . 55
3.4.4 Gestionnaires de contexte asynchrones . . . 56

4 Modèle d'exécution **57**
4.1 Structure d'un programme . . . 57
4.2 Noms et liaisons . . . 57
4.2.1 Liaisons des noms . . . 57
4.2.2 Résolution des noms . . . 58
4.2.3 Annotation scopes . . . 59
4.2.4 Lazy evaluation . . . 59
4.2.5 Noms natifs et restrictions d'exécution . . . 60
4.2.6 Interaction avec les fonctionnalités dynamiques . . . 60
4.3 Exceptions . . . 61

5 Le système d'importation **63**
5.1 `importlib` . . . 64
5.2 Les paquets . . . 64
5.2.1 Paquets classiques . . . 64
5.2.2 Paquets espaces de nommage . . . 65
5.3 Recherche . . . 65
5.3.1 Cache des modules . . . 65
5.3.2 Chercheurs et chargeurs . . . 66
5.3.3 Points d'entrées automatiques pour l'importation . . . 66
5.3.4 Méta-chemins . . . 66
5.4 Chargement . . . 67
5.4.1 Chargeurs . . . 68
5.4.2 Sous-modules . . . 69
5.4.3 Spécificateurs de modules . . . 69
5.4.4 Attributs des modules importés . . . 70
5.4.5 module.__path__ . . . 71
5.4.6 Représentation textuelle d'un module . . . 71
5.4.7 Invalidation de *bytecode* mis en cache . . . 72
5.5 Le chercheur dans *path* . . . 72
5.5.1 Chercheurs d'entrée dans *path* . . . 73
5.5.2 Protocole des chercheurs d'entrée dans *path* . . . 74
5.6 Remplacement du système d'importation standard . . . 74

5.7 Importations relatives au paquet . . . 75
5.8 Cas particulier de __main__ . . . 75
5.8.1 __main__.__spec__ . . . 75
5.9 Références . . . 76

6 Expressions **77**
6.1 Conversions arithmétiques . . . 77
6.2 Atomes . . . 77
6.2.1 Identifiants (noms) . . . 78
6.2.2 Littéraux . . . 78
6.2.3 Formes parenthésées . . . 78
6.2.4 Agencements des listes, ensembles et dictionnaires . . . 79
6.2.5 Agencements de listes . . . 79
6.2.6 Agencements d'ensembles . . . 80
6.2.7 Agencements de dictionnaires . . . 80
6.2.8 Expressions génératrices . . . 81
6.2.9 Expressions yield . . . 81
6.3 Primaires . . . 86
6.3.1 Références à des attributs . . . 86
6.3.2 sélection (ou indiçage) . . . 86
6.3.3 Tranches . . . 87
6.3.4 Appels . . . 87
6.4 Expression await . . . 89
6.5 L'opérateur puissance . . . 89
6.6 Arithmétique unaire et opérations sur les bits . . . 90
6.7 Opérations arithmétiques binaires . . . 90
6.8 Opérations de décalage . . . 91
6.9 Opérations binaires bit à bit . . . 92
6.10 Comparaisons . . . 92
6.10.1 Comparaisons de valeurs . . . 92
6.10.2 Opérations de tests d'appartenance à un ensemble . . . 94
6.10.3 Comparaisons d'identifiants . . . 95
6.11 Opérations booléennes . . . 95
6.12 Expressions d'affectation . . . 95
6.13 Expressions conditionnelles . . . 96
6.14 Expressions lambda . . . 96
6.15 Listes d'expressions . . . 96
6.16 Ordre d'évaluation . . . 97
6.17 Priorités des opérateurs . . . 97

7 Les instructions simples **99**
7.1 Les expressions . . . 99
7.2 Les assignations . . . 100
7.2.1 Les assignations augmentées . . . 102
7.2.2 Les assignations annotées . . . 102
7.3 L'instruction assert . . . 103
7.4 L'instruction pass . . . 103
7.5 L'instruction del . . . 104
7.6 L'instruction return . . . 104
7.7 L'instruction yield . . . 104
7.8 L'instruction raise . . . 105
7.9 L'instruction break . . . 106
7.10 L'instruction continue . . . 107
7.11 L'instruction import . . . 107
7.11.1 L'instruction future . . . 108
7.12 L'instruction global . . . 109
7.13 L'instruction nonlocal . . . 110
7.14 The type statement . . . 110

8 Instructions composées **113**
8.1 L'instruction `if` . . . 114
8.2 L'instruction `while` . . . 114
8.3 L'instruction `for` . . . 114
8.4 L'instruction `try` . . . 115
8.4.1 clause `except` . . . 115
8.4.2 clause `except*` . . . 117
8.4.3 clause `else` . . . 118
8.4.4 clause `finally` . . . 118
8.5 L'instruction `with` . . . 118
8.6 L'instruction `match` . . . 120
8.6.1 Aperçu . . . 120
8.6.2 Gardes . . . 121
8.6.3 Bloc `case` attrape-tout . . . 122
8.6.4 Filtres . . . 122
8.7 Définition de fonctions . . . 128
8.8 Définition de classes . . . 130
8.9 Coroutines . . . 131
8.9.1 Définition de fonctions coroutines . . . 131
8.9.2 L'instruction `async for` . . . 131
8.9.3 L'instruction `async with` . . . 132
8.10 Type parameter lists . . . 133
8.10.1 Generic functions . . . 134
8.10.2 Generic classes . . . 135
8.10.3 Generic type aliases . . . 135

9 Composants de plus haut niveau **137**
9.1 Programmes Python complets . . . 137
9.2 Fichier d'entrée . . . 138
9.3 Entrée interactive . . . 138
9.4 Entrée d'expression . . . 138

10 Spécification complète de la grammaire **139**

A Glossaire **155**

B À propos de ces documents **171**
B.1 Contributeurs de la documentation Python . . . 171

C Histoire et licence **173**
C.1 Histoire du logiciel . . . 173
C.2 Conditions générales pour accéder à, ou utiliser, Python . . . 174
C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.12.3 . . . 174
C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0 . . . 175
C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1 . . . 176
C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2 . . . 177
C.2.5 LICENCE BSD ZERO-CLAUSE POUR LE CODE DANS LA DOCUMENTATION DE PYTHON 3.12.3 . . . 177
C.3 Licences et remerciements pour les logiciels tiers . . . 178
C.3.1 Mersenne twister . . . 178
C.3.2 Interfaces de connexion (*sockets*) . . . 179
C.3.3 Interfaces de connexion asynchrones . . . 179
C.3.4 Gestion de témoin (*cookie*) . . . 180
C.3.5 Traçage d'exécution . . . 180
C.3.6 Les fonctions UUencode et UUdecode . . . 181
C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*) . . . 181
C.3.8 test_epoll . . . 182
C.3.9 Select kqueue . . . 182
C.3.10 SipHash24 . . . 183

C.3.11 *strtod* et *dtoa* . . . 183
C.3.12 OpenSSL . . . 184
C.3.13 expat . . . 187
C.3.14 libffi . . . 187
C.3.15 zlib . . . 188
C.3.16 cfuhash . . . 188
C.3.17 libmpdec . . . 189
C.3.18 Ensemble de tests C14N du W3C . . . 189
C.3.19 Audioop . . . 190
C.3.20 asyncio . . . 190

D Copyright **191**

Index **193**

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 ☺).

S'agissant du manuel de référence d'un langage, il est dangereux de rentrer profondément dans les détails d'implémentation ; l'implémentation peut changer et d'autres implémentations du même langage peuvent fonctionner différemment. En même temps, CPython est l'implémentation de Python la plus répandue (bien que d'autres implémentations gagnent en popularité) et certaines de ses bizarreries méritent parfois d'être mentionnées, en particulier lorsque l'implémentation impose des limitations supplémentaires. Par conséquent, vous trouvez de courtes "notes d'implémentation" saupoudrées dans le texte.

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 implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.

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
: An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.

PyPy
: An implementation of Python written completely in Python. It supports several advanced features not found in other implementations like stackless support and a Just in Time compiler. One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page.

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

The descriptions of lexical analysis and syntax use a modified Backus–Naur form (BNF) grammar notation. This uses the following style of definition :

```
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 lit le texte du programme comme des suites de caractères Unicode ; l'encodage du fichier source peut être spécifié par une déclaration d'encodage et vaut par défaut UTF-8, voir la **PEP 3120** pour les détails. Si le fichier source ne peut pas être décodé, une exception `SyntaxError` (erreur de syntaxe) est levée.

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

Une ligne physique est une suite de caractères terminée par une séquence de fin de ligne. Dans les fichiers sources et les chaînes de caractères, n'importe quelle séquence de fin de ligne des plateformes standards peut être utilisée ; Unix utilise le caractère ASCII LF (pour *linefeed*, saut de ligne en français), Windows utilise la séquence CR LF (*carriage return* suivi de *linefeed*) et Macintosh utilisait le caractère ASCII CR. Toutes ces séquences peuvent être utilisées, quelle que soit la plateforme. La fin de l'entrée est aussi une fin de ligne physique implicite.

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>
```

qui est reconnue par VIM de Bram Moolenaar.

If no encoding declaration is found, the default encoding is UTF-8. If the implicit or explicit encoding of a file is UTF-8, an initial UTF-8 byte-order mark (b'xefxbbxbf') is ignored rather than being a syntax error.

Si un encodage est déclaré, le nom de l'encodage doit être reconnu par Python (voir standard-encodings). L'encodage est utilisé pour toute l'analyse lexicale, y compris les chaînes de caractères, les commentaires et les identifiants.

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

Une ligne logique qui ne contient que des espaces, tabulations, caractères de saut de page (*formfeed* en anglais) ou commentaires est ignorée (c'est-à-dire que le lexème NEWLINE n'est pas produit). Pendant l'édition interactive d'instructions, la gestion des lignes vierges peut différer en fonction de l'implémentation de la boucle REPL. Dans l'interpréteur standard, une ligne complètement vierge (c'est-à-dire ne contenant strictement rien, même pas une espace ou un commentaire) termine une instruction multi-lignes.

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.

Les tabulations sont remplacées (de la gauche vers la droite) par une à huit espaces de manière à ce que le nombre de caractères remplacés soit un multiple de huit (nous avons ainsi la même règle que celle d'Unix). Le nombre total d'espaces précédant le premier caractère non blanc détermine alors le niveau d'indentation de la ligne. L'indentation ne peut pas être répartie sur plusieurs lignes physiques à l'aide de barres obliques inversées ; les espaces jusqu'à la première barre oblique inversée déterminent l'indentation.

L'indentation est déclarée inconsistante et rejetée si, dans un même fichier source, le mélange des tabulations et des espaces est tel que la signification dépend du nombre d'espaces que représente une tabulation. Une exception `TabError` est levée dans ce cas.

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

Les identifiants (aussi appelés *noms*) sont décrits par les définitions lexicales suivantes.

La syntaxe des identifiants en Python est basée sur l'annexe UAX-31 du standard Unicode avec les modifications définies ci-dessous ; consultez la **PEP 3131** pour plus de détails.

Dans l'intervalle ASCII (*U+0001..U+007F*), les caractères licites pour les identifiants sont les mêmes que pour Python 2.x : les lettres minuscules et majuscules de `A` à `Z`, le souligné (ou *underscore*) _ et, sauf pour le premier caractère, les chiffres de `0` à `9`.

Python 3.0 introduit des caractères supplémentaires en dehors de l'intervalle ASCII (voir la **PEP 3131**). Pour ces caractères, la classification utilise la version de la « base de données des caractères Unicode » telle qu'incluse dans le module `unicodedata`.

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

```
identifier     ::=   xid_start xid_continue*
id_start       ::=   <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the ur
id_continue    ::=   <all characters in id_start, plus characters in the categories Mn, M
xid_start      ::=   <all characters in id_start whose NFKC normalization is in "id_start
xid_continue   ::=   <all characters in id_continue whose NFKC normalization is in "id_co
```

Les codes de catégories Unicode cités ci-dessus signifient :

- *Lu* — lettres majuscules
- *Ll* — lettres minuscules
- *Lt* — lettres majuscules particulières (catégorie *titlecase* de l'Unicode)
- *Lm* — lettres modificatives avec chasse
- *Lo* — autres lettres
- *Nl* — nombres lettres (par exemple, les nombres romains)
- *Mn* — symboles que l'on combine avec d'autres (accents ou autres) sans générer d'espace (*nonspacing marks* en anglais)
- *Mc* — symboles que l'on combine avec d'autres en générant une espace (*spacing combining marks* en anglais)
- *Nd* — chiffres (arabes et autres)
- *Pc* — connecteurs (tirets et autres lignes)
- *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
- *Other_ID_Continue* — pareillement

Tous les identifiants sont convertis dans la forme normale NFKC pendant l'analyse syntaxique : la comparaison des identifiants se base sur leur forme NFKC.

A non-normative HTML file listing all valid identifier characters for Unicode 15.0.0 can be found at https://www.unicode.org/Public/15.0.0/ucd/DerivedCoreProperties.txt

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 :

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2 Mots-clés ad hoc

Added in version 3.10.

Some identifiers are only reserved under specific contexts. These are known as *soft keywords*. The identifiers `match`, `case`, `type` and _ can syntactically act as keywords in certain contexts, but this distinction is done at the parser level, not when tokenizing.

As soft keywords, their use in the grammar is possible while still preserving compatibility with existing code that uses these names as identifier names.

`match`, `case`, and _ are used in the *`match`* statement. `type` is used in the *`type`* statement.

Modifié dans la version 3.12 : `type` is now a soft keyword.

2.3.3 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 :

_*
N'est pas importé par `from module import *`.

_
Dans un motif `case` d'une instruction *`match`*, _ est un *mot-clé ad hoc* qui décrit un motif *attrape-tout*.
De son côté, l'interpréteur interactif place le résultat de la dernière évaluation dans la variable _ (son emplacement se situe dans le module `builtins`, avec les fonctions natives telles que `print`).
Ailleurs, _ est un identifiant comme un autre. Il est souvent utilisé pour désigner des éléments « spéciaux », mais il n'est pas spécial pour Python en tant que tel.

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.
Il est aussi communément utilisé pour signifier que la variable n'est pas utilisée.

__*__
Noms définis par le système, appelés noms « *dunder* » (pour *Double Underscores*) de manière informelle. 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 Littéraux de chaînes de caractères et de suites d'octets

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

```
stringliteral     ::=   [stringprefix](shortstring | longstring)
stringprefix      ::=   "r" | "u" | "R" | "U" | "f" | "F"
                        | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring       ::=   "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring        ::=   "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem   ::=   shortstringchar | stringescapeseq
longstringitem    ::=   longstringchar | stringescapeseq
shortstringchar   ::=   <any source character except "\" or newline or the quote>
longstringchar    ::=   <any source character except "\">
stringescapeseq   ::=   "\" <any source character>
```

```
bytesliteral     ::=   bytesprefix(shortbytes | longbytes)
bytesprefix      ::=   "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes       ::=   "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
```

```
longbytes         ::=   "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem    ::=   shortbyteschar | bytesescapeseq
longbytesitem     ::=   longbyteschar | bytesescapeseq
shortbyteschar    ::=   <any ASCII character except "\" or newline or the quote>
longbyteschar     ::=   <any ASCII character except "\">
bytesescapeseq    ::=   "\" <any ASCII character>
```

Une restriction syntaxique non indiquée par ces règles est qu'aucun blanc n'est autorisé entre le *stringprefix* ou *bytesprefix* et le reste du littéral. Le jeu de caractères source est défini par la déclaration d'encodage ; il vaut UTF-8 si aucune déclaration d'encodage n'est donnée dans le fichier source ; voir la section *Déclaration d'encodage*.

In plain English : Both types of 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 give special meaning to otherwise ordinary characters like `n`, which means 'newline' when escaped (`\n`). It can also be used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. See *escape sequences* below for examples.

Les littéraux de suites d'octets sont toujours préfixés par `'b'` ou `'B'` ; cela crée une instance de type `bytes` au lieu du type `str`. Ils ne peuvent contenir que des caractères ASCII ; les octets dont la valeur est supérieure ou égale à 128 doivent être exprimés à l'aide d'échappements.

Les chaînes et suites d'octets littérales peuvent être préfixées par la lettre `'r'` ou `'R'` ; de telles chaînes sont appelées *chaines brutes* (*raw strings* en anglais) et traitent la barre oblique inversée comme un caractère normal. En conséquence, les chaînes littérales `'\U'` et `'\u'` ne sont pas considérées comme spéciales. Comme les littéraux Unicode de Python 2.x se comportent différemment, la syntaxe `'ur'` n'est pas reconnue en Python 3.x.

Added in version 3.3 : le préfixe `'rb'` a été ajouté comme synonyme de `'br'` pour les littéraux de suites d'octets.

la gestion du préfixe historique pour les chaînes Unicode (`u'chaine'`) a été réintroduite afin de simplifier la maintenance de code compatible Python 2.x et 3.x. Voir la **PEP 414** pour davantage d'informations.

Une chaîne littérale qui contient `'f'` ou `'F'` dans le préfixe est une *chaîne de caractères littérale formatée* ; lisez *f-strings*. Le `'f'` peut être combiné avec `'r'` mais pas avec `'b'` ou `'u'`, donc les chaînes de caractères formatées sont possibles mais les littéraux de suites d'octets ne peuvent pas l'être.

Dans les chaînes entre triples guillemets, les sauts de ligne et guillemets peuvent ne pas être échappés (et sont donc pris en compte), mais trois guillemets non échappés à la suite terminent le littéral (on entend par guillemet le caractère utilisé pour commencer le littéral, c'est-à-dire `'` ou `"`).

Escape sequences

À moins que le préfixe `'r'` ou `'R'` ne soit présent, les séquences d'échappement dans les littéraux de chaînes et suites d'octets sont interprétées comme elles le seraient par le C Standard. Les séquences d'échappement reconnues sont :

Séquence d'échappement	Signification	Notes
`\`<newline>	barre oblique inversée et retour à la ligne ignorés	(1)
`\\`	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)	
`\r`	retour à la ligne ASCII (CR)	
`\t`	tabulation horizontale ASCII (TAB)	
`\v`	tabulation verticale ASCII (VT)	
`\`*ooo*	caractère dont le code est *ooo* en octal	(2,4)
`\x`*hh*	caractère dont le code est *ooo* en hexadécimal	(3,4)

Les séquences d'échappement reconnues seulement dans les chaînes littérales sont :

Séquence d'échappement	Signification	Notes
`\N{`*name*`}`	caractère dont le nom est *name* dans la base de données Unicode	(5)
`\u`*xxxx*	caractère dont le code est *xxxx* en hexadécimal	(6)
`\U`*xxxxxxxx*	caractère dont le code est *xxxxxxxx* en hexadécimal sur 32 bits	(7)

Notes :

(1) A backslash can be added at the end of a line to ignore the newline :

```
>>> 'This string will not include \
... backslashes or newline characters.'
'This string will not include backslashes or newline characters.'
```

The same result can be achieved using *triple-quoted strings*, or parentheses and *string literal concatenation*.

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

Modifié dans la version 3.11 : Octal escapes with value larger than `0o377` produce a `DeprecationWarning`.

Modifié dans la version 3.12 : Octal escapes with value larger than `0o377` produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

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

(4) Dans un littéral de suite d'octets, un échappement hexadécimal ou octal est un octet dont la valeur est donnée. Dans une chaîne littérale, un échappement est un caractère Unicode dont le code est donné.

(5) Modifié dans la version 3.3 : Ajout du support pour les alias de noms [1].

(6) Exactement quatre chiffres hexadécimaux sont requis.

(7) N'importe quel caractère Unicode peut être encodé de cette façon. Exactement huit chiffres hexadécimaux sont requis.

Contrairement au C standard, toutes les séquences d'échappement non reconnues sont laissées inchangées dans la chaîne, c'est-à-dire que *la barre oblique inversée est laissée dans le résultat* (ce comportement est utile en cas de débogage : si une séquence d'échappement est mal tapée, la sortie résultante est plus facilement reconnue comme source de l'erreur). Notez bien également que les séquences d'échappement reconnues uniquement dans les littéraux de chaînes de caractères ne sont pas reconnues pour les littéraux de suites d'octets.

Modifié dans la version 3.6 : Unrecognized escape sequences produce a `DeprecationWarning`.

Modifié dans la version 3.12 : Unrecognized escape sequences produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

Même dans une chaîne littérale brute, les guillemets peuvent être échappés avec une barre oblique inversée mais la barre oblique inversée reste dans le résultat ; par exemple, `r"\""` est une chaîne de caractères valide composée de deux caractères : une barre oblique inversée et un guillemet double ; `r"\"` n'est pas une chaîne de caractères valide (même une chaîne de caractères brute ne peut pas se terminer par un nombre impair de barres obliques inversées). Plus précisément, *une chaîne littérale brute ne peut pas se terminer par une seule barre oblique inversée* (puisque la barre oblique inversée échappe le guillemet suivant). Notez également qu'une simple barre oblique inversée suivie d'un saut de ligne est interprétée comme deux caractères faisant partie du littéral et *non* comme une continuation de ligne.

1. https://www.unicode.org/Public/15.0.0/ucd/NameAliases.txt

2.4.2 Concaténation de chaînes de caractères

Plusieurs chaînes de caractères ou suites d'octets adjacentes (séparées par des blancs), utilisant éventuellement des conventions de guillemets différentes, sont autorisées. La signification est la même que leur concaténation. Ainsi, `"hello" 'world'` est l'équivalent de `"helloworld"`. Cette fonctionnalité peut être utilisée pour réduire le nombre de barres obliques inverses, pour diviser de longues chaînes de caractères sur plusieurs lignes ou même pour ajouter des commentaires à des portions de chaînes de caractères. Par exemple :

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

Notez que cette fonctionnalité agit au niveau syntaxique mais est implémentée au moment de la compilation. Pour concaténer les expressions des chaînes de caractères au moment de l'exécution, vous devez utiliser l'opérateur +. Notez également que la concaténation littérale peut utiliser un style différent de guillemets pour chaque composant (et même mélanger des chaînes de caractères brutes et des chaînes de caractères entre triples guillemets). Enfin, les chaînes de caractères formatées peuvent être concaténées avec des chaînes de caractères ordinaires.

2.4.3 f-strings

Added in version 3.6.

Une *chaine de caractères littérale formatée* ou *f-string* est une chaine de caractères littérale préfixée par `'f'` ou `'F'`. Ces chaines peuvent contenir des champs à remplacer, c'est-à-dire des expressions délimitées par des accolades `{}`. Alors que les autres littéraux de chaines ont des valeurs constantes, les chaines formatées sont de vraies expressions évaluées à l'exécution.

Les séquences d'échappement sont décodées comme à l'intérieur des chaînes de caractères ordinaires (sauf lorsqu'une chaîne de caractères est également marquée comme une chaîne brute). Après décodage, la grammaire s'appliquant au contenu de la chaîne de caractères est :

```
f_string            ::=  (literal_char | "{{" | "}}" | replacement_field)*
replacement_field   ::=  "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression        ::=  (conditional_expression | "*" or_expr)
                         ("," conditional_expression | "," "*" or_expr)* [","]
                         | yield_expression
conversion          ::=  "s" | "r" | "a"
format_spec         ::=  (literal_char | replacement_field)*
literal_char        ::=  <any code point except "{", "}" or NULL>
```

Les portions qui sont en dehors des accolades sont traitées comme les littéraux, sauf les doubles accolades `'{{'` ou `'}}'` qui sont remplacées par la simple accolade correspondante. Une simple accolade ouvrante `'{'` marque le début du champ à remplacer, qui commence par une expression Python. Pour afficher à la fois le texte de l'expression et sa valeur une fois évaluée (utile lors du débogage), un signe égal `'='` peut être ajouté après l'expression. Ensuite, il peut y avoir un champ de conversion, introduit par un point d'exclamation `'!'`. Une spécification de format peut aussi être rajoutée, introduite par le caractère deux-points `':'`. Le champ à remplacer se termine par une accolade fermante `'}'`.

Expressions in formatted string literals are treated like regular Python expressions surrounded by parentheses, with a few exceptions. An empty expression is not allowed, and both *lambda* and assignment expressions `:=` must be surrounded by explicit parentheses. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right. Replacement expressions can contain newlines in both single-quoted and triple-quoted f-strings and they can contain comments. Everything that comes after a `#` inside a replacement field is a comment (even closing braces and quotes). In that case, replacement fields must be closed in a different line.

```
>>> f"abc{a # This is a comment }"
... + 3}"
'abc5'
```

Modifié dans la version 3.7 : Avant Python 3.7, il était illégal d'utiliser *await* ainsi que les compréhensions utilisant *async for* dans les expressions au sein des chaînes de caractères formatées littérales à cause d'un problème dans l'implémentation.

Modifié dans la version 3.12 : Prior to Python 3.12, comments were not allowed inside f-string replacement fields.

Lorsqu'un signe égal `'='` est présent, la sortie comprend le texte de l'expression, le signe `'='` et la valeur calculée. Les espaces après l'accolade ouvrante `'{'`, dans l'expression et après le signe `'='` sont conservées à l'affichage. Par défaut, le signe `'='` utilise la `repr()` de l'expression, sauf si un format est indiqué. Quand le format est indiqué, c'est `str()` de l'expression qui est utilisée à moins qu'une conversion `!r` ne soit déclarée.

Added in version 3.8 : le signe égal `'='`.

Si une conversion est spécifiée, le résultat de l'évaluation de l'expression est converti avant d'être formaté. La conversion `'!s'` appelle `str()` sur le résultat, `'!r'` appelle `repr()` et `'!a'` appelle `ascii()`.

The result is then formatted using the `format()` protocol. The format specifier is passed to the *__format__()* method of the expression or conversion result. An empty string is passed when the format specifier is omitted. The formatted result is then included in the final value of the whole string.

Top-level format specifiers may include nested replacement fields. These nested fields may include their own conversion fields and format specifiers, but may not include more deeply nested replacement fields. The format specifier mini-language is the same as that used by the `str.format()` method.

Les chaînes formatées littérales peuvent être concaténées mais les champs à remplacer ne peuvent pas être divisés entre les littéraux.

Quelques exemples de chaines formatées littérales :

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

Reusing the outer f-string quoting type inside a replacement field is permitted :

```
>>> a = dict(x=2)
>>> f"abc {a["x"]} def"
'abc 2 def'
```

Modifié dans la version 3.12 : Prior to Python 3.12, reuse of the same quoting type of the outer f-string inside a replacement field was not possible.

Backslashes are also allowed in replacement fields and are evaluated the same way as in any other context :

```
>>> a = ["a", "b", "c"]
>>> print(f"List a contains:\n{"\n".join(a)}")
List a contains:
a
b
c
```

Modifié dans la version 3.12 : Prior to Python 3.12, backslashes were not permitted inside an f-string replacement field.

Une chaine formatée littérale ne peut pas être utilisée en tant que *docstring*, même si elle ne comporte pas d'expression.

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

Consultez aussi la **PEP 498** qui propose l'ajout des chaines formatées littérales et `str.format()` qui utilise un mécanisme similaire pour formater les chaînes de caractères.

2.4.4 Littéraux numériques

Il existe trois types de littéraux numériques : les entiers, les nombres à virgule flottante et les nombres imaginaires. Il n'y a pas de littéraux complexes (les nombres complexes peuvent être construits en ajoutant un nombre réel et un nombre imaginaire).

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`.

2.4.5 Entiers littéraux

Les entiers littéraux sont décrits par les définitions lexicales suivantes :

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

Il n'y a pas de limite pour la longueur des entiers littéraux, sauf celle relative à la capacité mémoire.

Les tirets bas sont ignorés pour déterminer la valeur numérique du littéral. Ils peuvent être utilisés pour grouper les chiffres afin de faciliter la lecture. Un souligné peut être placé entre des chiffres ou après la spécification de la base telle que `0x`.

Notez que placer des zéros en tête de nombre pour un nombre décimal différent de zéro n'est pas autorisé. Cela permet d'éviter l'ambigüité avec les littéraux en base octale selon le style C que Python utilisait avant la version 3.0.

Quelques exemples d'entiers littéraux :

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

Modifié dans la version 3.6 : Les tirets bas ne sont pas autorisés pour grouper les littéraux.

2.4.6 Nombres à virgule flottante littéraux

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

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

Notez que la partie entière et l'exposant sont toujours interprétés comme étant en base 10. Par exemple, `077e010` est licite et désigne le même nombre que `77e10`. La plage autorisée pour les littéraux de nombres à virgule flottante dépend de l'implémentation. Comme pour les entiers littéraux, les soulignés permettent de grouper des chiffres.

Quelques exemples de nombres à virgule flottante littéraux :

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

Modifié dans la version 3.6 : Les tirets bas ne sont pas autorisés pour grouper les littéraux.

2.4.7 Imaginaires littéraux

Les nombres imaginaires sont décrits par les définitions lexicales suivantes :

```
imagnumber   ::=   (floatnumber | digitpart) ("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   3.14_15_93j
```

2.5 Opérateurs

Les lexèmes suivants sont des opérateurs :

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

2.6 Délimiteurs

Les lexèmes suivants servent de délimiteurs dans la grammaire :

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

Le point peut aussi apparaître dans les littéraux de nombres à virgule flottante et imaginaires. Une suite de trois points possède une signification spéciale : c'est une ellipse littérale. La deuxième partie de la liste, les opérateurs d'affectation augmentés, servent de délimiteurs pour l'analyseur lexical mais sont aussi des opérateurs.

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.

Particularité de l'implémentation CPython : en CPython, `id(x)` est l'adresse mémoire où est stocké `x`.

Le type de l'objet détermine les opérations que l'on peut appliquer à l'objet (par exemple, « a-t-il une longueur ? ») et définit aussi les valeurs possibles pour les objets de ce type. La fonction `type()` renvoie le type de l'objet (qui est lui-même un objet). Comme l'identifiant, le *type* d'un objet ne peut pas être modifié [1].

La *valeur* de certains objets peut changer. Les objets dont la valeur peut changer sont dits mutables ; les objets dont la valeur est définitivement fixée à leur création sont dits *immuables* (*immutable* en anglais). La valeur d'un objet conteneur immuable qui contient une référence vers un objet mutable peut varier lorsque la valeur de l'objet mutable change ; cependant, le conteneur est quand même considéré comme immuable parce que l'ensemble des objets qu'il contient ne peut pas être modifié. Ainsi, l'immuabilité n'est pas strictement équivalente au fait d'avoir une valeur non modifiable, c'est plus subtil. La muabilité d'un objet est définie par son type ; par exemple, les nombres, les chaînes de caractères et les *n*-uplets sont immuables alors que les dictionnaires et les listes sont mutables.

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.

Particularité de l'implémentation CPython : CPython utilise aujourd'hui un mécanisme de compteur de références avec une détection, en temps différé et optionnelle, des cycles d'objets. Ce mécanisme supprime la plupart des objets dès qu'ils ne sont plus accessibles mais il ne garantit pas la suppression des objets où il existe des références circulaires. Consultez la documentation du module `gc` pour tout ce qui concerne la suppression des cycles. D'autres implémentations agissent différemment et CPython pourrait évoluer. Ne vous reposez pas sur la finalisation immédiate des objets devenus inaccessibles (ainsi, vous devez toujours fermer les fichiers explicitement).

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.

Note that the use of the implementation's tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a *try*...*except* statement may keep objects alive.

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 and the *with* statement provide convenient ways to do this.

Certains objets contiennent des références à d'autres objets ; on les appelle *conteneurs*. Comme exemples de conteneurs, nous pouvons citer les *n*-uplets, 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 *n*-uplet) contient une référence à un objet mutable, sa valeur change si cet objet mutable 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 mutables 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

Vous trouvez ci-dessous une liste des types natifs de Python. Des modules d'extension (écrits en C, Java ou d'autres langages) peuvent définir des types supplémentaires. Les futures versions de Python pourront ajouter des types à cette hiérarchie (par exemple les nombres rationnels, des tableaux d'entiers stockés efficacement, etc.), bien que de tels ajouts se trouvent souvent plutôt dans la bibliothèque standard.

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.

3.2.1 `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 renvoient rien explicitement. Sa valeur booléenne est fausse.

3.2.2 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 should 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.) It should not be evaluated in a boolean context.

Consultez implementing-the-arithmetic-operations pour davantage de détails.

Modifié dans la version 3.9 : Evaluating `NotImplemented` in a boolean context is deprecated. While it currently evaluates as true, it will emit a `DeprecationWarning`. It will raise a `TypeError` in a future version of Python.

3.2.3 Ellipse

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 littéral `...` ou le nom natif `Ellipsis`. Sa valeur booléenne est vraie.

3.2.4 `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.

Les représentations sous forme de chaînes de caractères des objets numériques, produites par *`__repr__()`* et *`__str__()`*, ont les propriétés suivantes :

- Ce sont des littéraux numériques valides qui, s'ils sont passés au constructeur de leur classe, produisent un objet qui a la valeur numérique de l'objet d'origine.
- La représentation est en base 10, si possible.
- Les zéros en tête, sauf en ce qui concerne un zéro seul avant la virgule (représentée par un point en Python conformément à la convention anglo-saxonne), ne sont pas affichés.
- Les zéros en fin, sauf en ce qui concerne un zéro seul après la virgule, ne sont pas affichés.
- Le signe n'est affiché que lorsque le nombre est négatif.

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).

Note : Les règles pour la représentation des entiers ont pour objet de donner l'interprétation la plus naturelle pour les opérations de décalage et masquage qui impliquent des entiers négatifs.

Il existe deux types d'entiers :

Entiers (`int`)
: 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.

Booléens (`bool`)
: Ils représentent les valeurs *faux* et *vrai*. Deux objets, `False` et `True`, sont les seuls objets booléens. Le type booléen est un sous-type du type entier et les valeurs booléennes se comportent comme les valeurs 0 (pour `False`) et 1 (pour `True`) dans presque tous les contextes. L'exception concerne la conversion en chaîne de caractères où `"False"` et `"True"` sont renvoyées.

`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 (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`.

3.2.5 Séquences

These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is *n*, the index set contains the numbers 0, 1, ..., *n*-1. Item *i* of sequence *a* is selected by `a[i]`. Some sequences, including built-in sequences, interpret negative subscripts by adding the sequence length. For example, `a[-2]` equals `a[n-2]`, the second to last item of sequence a with length `n`.

Sequences also support slicing : `a[i:j]` selects all items with index *k* such that *i* <= *k* < *j*. When used as an expression, a slice is a sequence of the same type. The comment above about negative indexes also applies to negative slice positions.

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 mutables 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
: A string is a sequence of values that represent Unicode code points. All the code points in the range `U+0000 - U+10FFFF` can be represented in a string. Python doesn't have a `char` type ; instead, every code point in the string is represented as a string object with length `1`. The built-in function `ord()` converts a code point from its string form to an integer in the range `0 - 10FFFF` ; `chr()` converts an integer in the range `0 - 10FFFF` to the corresponding length `1` string object. `str.encode()` can be used to convert a `str` to `bytes` using the given text encoding, and `bytes.decode()` can be used to achieve the opposite.

***n*-uplets (*tuples* en anglais)**
: Les éléments d'un *n*-uplet peuvent être n'importe quel objet Python. Les *n*-uplets de deux éléments ou plus sont formés par une liste d'expressions dont les éléments sont séparés par des virgules. Un *n*-uplet 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 *n*-uplet car les parenthèses doivent rester disponibles pour grouper les expressions). Un *n*-uplet vide est formé à l'aide d'une paire de parenthèses vide.

Chaînes d'octets (ou *bytes*)
: Les objets *bytes* sont des tableaux immuables. Les éléments sont des octets (donc composés de 8 bits), représentés par des entiers dans l'intervalle 0 à 255 inclus. Les littéraux *bytes* (tels que `b'abc'`) et la fonction native constructeur `bytes()` peuvent être utilisés pour créer des objets *bytes*. Aussi, un objet *bytes* peut être décodé vers une chaîne *via* la méthode `decode()`.

Séquences mutables

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

Note : The `collections` and `array` module provide additional examples of mutable sequence types.

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

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
: Un objet *bytearray* est un tableau mutable. Il est créé par la fonction native constructeur `bytearray()`. À part la propriété d'être mutable (et donc de ne pas pouvoir calculer son empreinte par hachage), un tableau d'octets possède la même interface et les mêmes fonctionnalités qu'un objet immuable `bytes`.

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

3.2.7 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 correspondances `a` ; elle peut être utilisée dans des expressions, comme cible d'une affectation 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 mutables 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 préservent l'ordre d'insertion, ce qui signifie que les clés sont renvoyées séquentiellement dans le même ordre que celui de l'insertion. Remplacer une clé existante ne change pas l'ordre. Par contre, la retirer puis la réinsérer la met à la fin et non à sa précédente position.

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

Les modules d'extensions `dbm.ndbm` et `dbm.gnu` apportent d'autres exemples de types tableaux de correspondances, de même que le module `collections`.

Modifié dans la version 3.7 : les dictionnaires ne conservaient pas l'ordre d'insertion dans les versions antérieures à Python 3.6. Dans CPython 3.6, l'ordre d'insertion était déjà conservé, mais considéré comme un détail d'implémentation et non comme une garantie du langage.

3.2.8 Types appelables

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

Fonctions définies par l'utilisateur

Un objet 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.

Special read-only attributes

Attribut	Signification
`function.__globals__`	A reference to the `dictionary` that holds the function's *global variables* -- the global namespace of the module in which the function was defined.
`function.__closure__`	`None` or a `tuple` of cells that contain bindings for the function's free variables. Un objet cellule possède un attribut `cell_contents`. Il peut être utilisé pour obtenir la valeur de la cellule et pour en définir la valeur.

Special writable attributes

Most of these attributes check the type of the assigned value :

Attribut	Signification
`function.__doc__`	The function's documentation string, or `None` if unavailable. Not inherited by subclasses.
`function.__name__`	The function's name. See also : `__name__` `attributes`.
`function.__qualname__`	The function's *qualified name*. See also : `__qualname__` `attributes`. Added in version 3.3.
`function.__module__`	Nom du module où la fonction est définie ou `None` si ce nom n'est pas disponible.
`function.__defaults__`	A `tuple` containing default *parameter* values for those parameters that have defaults, or `None` if no parameters have a default value.
`function.__code__`	The *code object* representing the compiled function body.
`function.__dict__`	The namespace supporting arbitrary function attributes. See also : `__dict__` `attributes`.
`function.__annotations__`	A `dictionary` containing annotations of *parameters*. The keys of the dictionary are the parameter names, and `'return'` for the return annotation, if provided. See also : annotations-howto.
`function.__kwdefaults__`	A `dictionary` containing defaults for keyword-only *parameters*.
`function.__type_params__`	A `tuple` containing the *type parameters* of a *generic function*. Added in version 3.12.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes.

Particularité de l'implémentation CPython : CPython's current implementation only supports function attributes on user-defined functions. Function attributes on *built-in functions* may be supported in the future.

Additional information about a function's definition can be retrieved from its *code object* (accessible via the *__code__* attribute).

Méthodes d'instances

Un objet méthode d'instance combine une classe, une instance de classe et tout objet appelable (normalement une fonction définie par l'utilisateur).

Special read-only attributes :

`method.__self__`	Refers to the class instance object to which the method is *bound*
`method.__func__`	Refers to the original *function object*
`method.__doc__`	The method's documentation (same as `method.__func__.__doc__`). A `string` if the original function had a docstring, else `None`.
`method.__name__`	The name of the method (same as `method.__func__.__name__`)
`method.__module__`	The name of the module the method was defined in, or `None` if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying *function object*.

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* or a `classmethod` object.

When an instance method object is created by retrieving a user-defined *function object* from a class via one of its instances, its `__self__` attribute is the instance, and the method object is said to be *bound*. The new method's `__func__` attribute is the original function object.

When an instance method object is created by retrieving a `classmethod` object from a class or instance, its `__self__` attribute is the class itself, and its `__func__` attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (`__func__`) is called, inserting the class instance (`__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 an instance method object is derived from a `classmethod` object, the "class instance" stored in `__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 instance method object happens each time the attribute is retrieved from the 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 génératrices (ou générateurs)

Une fonction ou une méthode qui utilise l'instruction `yield` (voir la section *L'instruction yield*) est appelée *fonction génératrice*. Une telle fonction, lorsqu'elle est appelée, renvoie toujours un objet *itérateur* qui peut être utilisé pour exécuter le corps de la fonction : appeler la méthode `iterator.__next__()` de l'itérateur exécute la fonction jusqu'à ce qu'elle renvoie une valeur à l'aide de l'instruction `yield`. Quand la fonction exécute l'instruction `return` ou se termine, une exception `StopIteration` est levée et l'itérateur a atteint la fin de l'ensemble de valeurs qu'il peut renvoyer.

Fonctions coroutines

Une fonction ou méthode définie en utilisant *async def* est appelée *fonction coroutine*. Une telle fonction, quand elle est appelée, renvoie un objet *coroutine*. Elle peut contenir des expressions *await* ou *async with* ou des instructions *async for*. Voir également la section *Objets coroutines*.

Fonctions génératrices (ou générateurs) asynchrones

Une fonction ou une méthode définie avec *async def* et qui utilise l'instruction *yield* est appelée *fonction génératrice asynchrone*. Une telle fonction, quand elle est appelée, renvoie un objet *itérateur asynchrone* qui peut être utilisé dans des instructions *async for* pour exécuter le corps de la fonction.

Appeler la méthode *aiterator.__anext__* de l'itérateur asynchrone renvoie un *awaitable* qui, lorsqu'on l'attend, s'exécute jusqu'à ce qu'il fournisse une valeur à l'aide de l'expression *yield*. Quand la fonction exécute une instruction *return* (sans valeur) ou arrive à la fin, une exception `StopAsyncIteration` est levée et l'itérateur asynchrone a atteint la fin de l'ensemble des valeurs qu'il peut produire.

Fonctions natives

A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes :

- `__doc__` is the function's documentation string, or `None` if unavailable. See *function.__doc__*.
- `__name__` is the function's name. See *function.__name__*.
- `__self__` is set to `None` (but see the next item).
- `__module__` is the name of the module the function was defined in or `None` if unavailable. See *function.__module__*.

Méthodes natives

This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *alist*. (The attribute has the same semantics as it does with *other instance methods*.)

Classes

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.

Instances de classe

Les instances d'une classe peuvent devenir des appelables si vous définissez la méthode *__call__()* de leur classe.

3.2.9 Modules

Modules are a basic organizational unit of Python code, and are created by the *import system* as invoked either by the *import* statement, or by calling functions such as `importlib.import_module()` and built-in `__import__()`. A module object has a namespace implemented by a `dictionary` object (this is the dictionary referenced by the *__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'affectation d'un attribut met à jour le dictionnaire d'espace de nommage du module, par exemple `m.x = 1` est équivalent à `m.__dict__["x"] = 1`.

Attributs prédéfinis (accessibles en écriture) :

__name__
Nom du module.

__doc__
Chaîne de documentation du module (*docstring* en anglais), ou `None` si le module n'en a pas.

__file__
Chemin vers le fichier à partir duquel le module a été chargé, s'il a été chargé depuis un fichier. L'attribut *__file__* peut être manquant pour certains types de modules, tels que les modules C qui sont statiquement liés à l'interpréteur. Pour les modules d'extension chargés dynamiquement à partir d'une bibliothèque partagée, c'est le chemin vers le fichier de la bibliothèque partagée.

__annotations__
Dictionnaire des *annotations de variable* trouvées lors de l'exécution du code du module. Pour plus de détails sur l'attribut `__annotations__`, voir annotations-howto.

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

Particularité de l'implémentation CPython : 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.

3.2.10 Classes déclarées par le développeur

Custom class types 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 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. This search of the base classes uses the 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 Python can be found at python_2.3_mro.

When a class attribute reference (for class `C`, say) would yield a class method object, it is transformed into an instance method object whose *__self__* attribute is `C`. When it would yield a `staticmethod` 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__`.

Les affectations 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).

Attributs spéciaux :

__name__
Nom de la classe.

__module__
Nom du module où la classe a été définie.

__dict__
Dictionnaire qui forme l'espace de nommage de la classe.

__bases__
N-uplet des classes mères, dans le même ordre que dans la définition de la classe.

__doc__
Chaîne de documentation de la classe (*docstring* en anglais), ou bien `None` si la classe n'en a pas.

__annotations__
Dictionnaire des *annotations de variable* trouvées lors de l'exécution du code de la classe. Pour plus de détails sur l'attribut `__annotations__`, voir annotations-howto.

__type_params__
A tuple containing the *type parameters* of a *generic class*.

3.2.11 Instances de classe

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, it is transformed into an instance method object whose *__self__* attribute is the instance. Static method and class method objects are also transformed ; 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 affectations 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 correspondances 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.

3.2.12 Objets entrées-sorties (ou objets fichiers)

Un *objet fichier* représente un fichier ouvert. Différents raccourcis existent pour créer des objets fichiers : la fonction native `open()` et aussi `os.popen()`, `os.fdopen()` ou la méthode `makefile()` des objets connecteurs (et sûrement d'autres fonctions ou méthodes fournies par les modules d'extensions).

Les objets `sys.stdin`, `sys.stdout` et `sys.stderr` sont initialisés à des objets fichiers correspondant à l'entrée standard, la sortie standard et le flux d'erreurs de l'interpréteur ; ils sont tous ouverts en mode texte et se conforment donc à l'interface définie par la classe abstraite `io.TextIOBase`.

3.2.13 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 *code intermédiaire*. 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 mutables.

Special read-only attributes

codeobject.**co_name**	The function name
codeobject.**co_qualname**	The fully qualified function name Added in version 3.11.
codeobject.**co_argcount**	The total number of positional *parameters* (including positional-only parameters and parameters with default values) that the function has
codeobject.**co_posonlyargcount**	The number of positional-only *parameters* (including arguments with default values) that the function has
codeobject.**co_kwonlyargcount**	The number of keyword-only *parameters* (including arguments with default values) that the function has
codeobject.**co_nlocals**	The number of *local variables* used by the function (including parameters)
codeobject.**co_varnames**	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
codeobject.**co_cellvars**	A `tuple` containing the names of *local variables* that are referenced by nested functions inside the function
codeobject.**co_freevars**	A `tuple` containing the names of free variables in the function
codeobject.**co_code**	A string representing the sequence of *bytecode* instructions in the function
codeobject.**co_consts**	A `tuple` containing the literals used by the *bytecode* in the function
codeobject.**co_names**	A `tuple` containing the names used by the *bytecode* in the function
codeobject.**co_filename**	The name of the file from which the code was compiled
codeobject.**co_firstlineno**	The line number of the first line of the function
codeobject.**co_lnotab**	A string encoding the mapping from *bytecode* offsets to line numbers. For details, see the source code of the interpreter. Obsolète depuis la version 3.12 : This attribute of code objects is deprecated, and may be removed in Python 3.14.
codeobject.**co_stacksize**	The required stack size of the code object
codeobject.**co_flags**	An `integer` encoding a number of flags for the interpreter.

The following flag bits are defined for *co_flags* : bit `0x04` is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments ; bit `0x08` is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments ; bit `0x20` is set if the function is a generator. See inspect-module-co-flags for details on the semantics of each flags that might be present.

Future feature declarations (`from __future__ import division`) also use bits in *co_flags* to indicate whether a code object was compiled with a particular feature enabled : bit `0x2000` is set if the function was compiled with future division enabled ; bits `0x10` and `0x1000` were used in earlier versions of Python.

Other bits in *co_flags* are reserved for internal use.

If a code object represents a function, the first item in *co_consts* is the documentation string of the function, or `None` if undefined.

Methods on code objects

`codeobject.`**`co_positions`**`()`

Returns an iterable over the source code positions of each *bytecode* instruction in the code object.

The iterator returns `tuples` containing the `(start_line, end_line, start_column, end_column)`. The *i-th* tuple corresponds to the position of the source code that compiled to the *i-th* instruction. Column information is 0-indexed utf-8 byte offsets on the given source line.

L'information sur la position peut être manquante. Ce peut être le cas si (liste non exhaustive) :

— l'interpréteur est lancé avec l'option `-X no_debug_ranges` ;
— le fichier *.pyc* est le produit d'une compilation avec l'option `-X no_debug_ranges` ;
— le *n*-uplet de position correspond à des instructions artificielles ;
— les lignes et colonnes ne peuvent pas être représentées en tant que nombre, en raison de limitations dues à l'implémentation ;

Dans ce cas, certains ou tous les éléments du *n*-uplet peuvent valoir `None`.

Added in version 3.11.

Note : cette fonctionnalité nécessite de stocker les positions de colonne dans les objets code, ce qui peut conduire à une légère augmentation de l'utilisation du disque par les fichiers Python compilés ou de l'utilisation de la mémoire. Pour éviter de stocker cette information supplémentaire ou pour désactiver l'affichage supplémentaire dans la pile d'appels, vous pouvez activer l'option de ligne de commande `-X no_debug_ranges` ou la variable d'environnement `PYTHONNODEBUGRANGES`.

`codeobject.`**`co_lines`**`()`

Returns an iterator that yields information about successive ranges of *bytecode*s. Each item yielded is a `(start, end, lineno) tuple` :

— `start` (an `int`) represents the offset (inclusive) of the start of the *bytecode* range
— `end` (an `int`) represents the offset (exclusive) of the end of the *bytecode* range
— `lineno` is an `int` representing the line number of the *bytecode* range, or `None` if the bytecodes in the given range have no line number

The items yielded will have the following properties :

— The first range yielded will have a `start` of 0.
— The `(start, end)` ranges will be non-decreasing and consecutive. That is, for any pair of `tuples`, the `start` of the second will be equal to the `end` of the first.
— No range will be backwards : `end >= start` for all triples.
— The last `tuple` yielded will have `end` equal to the size of the *bytecode*.

Zero-width ranges, where `start == end`, are allowed. Zero-width ranges are used for lines that are present in the source code, but have been eliminated by the *bytecode* compiler.

Added in version 3.10.

Voir aussi :

PEP 626 - Precise line numbers for debugging and other tools.
The PEP that introduced the `co_lines()` method.

codeobject.**replace**(***kwargs*)

Return a copy of the code object with new values for the specified fields.

Added in version 3.8.

Objets cadres

Frame objects represent execution frames. They may occur in *traceback objects*, and are also passed to registered trace functions.

Special read-only attributes

frame.**f_back**	Points to the previous stack frame (towards the caller), or None if this is the bottom stack frame
frame.**f_code**	The *code object* being executed in this frame. Accessing this attribute raises an auditing event object.__getattr__ with arguments obj and "f_code".
frame.**f_locals**	The dictionary used by the frame to look up *local variables*
frame.**f_globals**	The dictionary used by the frame to look up *global variables*
frame.**f_builtins**	The dictionary used by the frame to look up *built-in (intrinsic) names*
frame.**f_lasti**	The "precise instruction" of the frame object (this is an index into the *bytecode* string of the *code object*)

Special writable attributes

frame.**f_trace**	If not None, this is a function called for various events during code execution (this is used by debuggers). Normally an event is triggered for each new source line (see *f_trace_lines*).
frame.**f_trace_lines**	Set this attribute to False to disable triggering a tracing event for each source line.
frame.**f_trace_opcodes**	Set this attribute to True to allow per-opcode events to be requested. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.
frame.**f_lineno**	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 this attribute.

Frame object methods

Les objets cadres comprennent une méthode :

`frame.`**`clear`**`()`

This method clears all references to *local variables* held by the frame. Also, if the frame belonged to a *generator*, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its *traceback* for later use).

`RuntimeError` est levée si le cadre est en cours d'exécution.

Added in version 3.4.

Objets traces d'appels

Traceback objects represent the stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling `types.TracebackType`.

Modifié dans la version 3.7 : Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, 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 the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

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`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the *`tb_next`* attributes should be linked to form a full stack trace.

Special read-only attributes :

`traceback.`**`tb_frame`**	Points to the execution *frame* of the current level. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.
`traceback.`**`tb_lineno`**	Gives the line number where the exception occurred
`traceback.`**`tb_lasti`**	Indicates the "precise instruction".

The line number and last instruction in the traceback may differ from the line number of its *frame object* if the exception occurred in a *`try`* statement with no matching except clause or with a *`finally`* clause.

`traceback.`**`tb_next`**

The special writable attribute `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

Modifié dans la version 3.7 : This attribute is now writable

Objets tranches

Un objet tranche est utilisé pour représenter des découpes des méthodes *__getitem__()*. Ils sont aussi créés par la fonction native `slice()`.

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*)

Cette méthode prend un argument entier *length* et calcule les informations de la tranche que l'objet *slice* décrit s'il est appliqué à une séquence de *length* éléments. Elle renvoie un triplet d'entiers ; respectivement, ce sont les indices de *début* et *fin* ainsi que le *pas* de découpe. Les indices manquants ou en dehors sont gérés de manière cohérente avec les tranches normales.

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 sont aussi appelables. Les objets méthodes statiques sont créés par le constructeur natif `staticmethod()`.

Objets méthodes de classes

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under *"instance methods"*. Class method objects are created by the built-in `classmethod()` constructor.

3.3 Méthodes spéciales

Une classe peut implémenter certaines opérations que l'on invoque par une syntaxe spéciale (telles que les opérations arithmétiques ou la découpe en tranches) en définissant des méthodes aux noms particuliers. C'est l'approche utilisée par Python pour la *surcharge d'opérateur*, permettant à une classe de définir son propre comportement vis-à-vis des opérateurs du langage. Par exemple, si une classe définit une méthode *__getitem__()* et que `x` est une instance de cette classe, alors `x[i]` est globalement équivalent à `type(x).__getitem__(x, i)`. Sauf lorsque c'est mentionné, toute tentative d'appliquer une opération alors que la méthode appropriée n'est pas définie lève une exception (typiquement `AttributeError` ou `TypeError`).

Définir une méthode spéciale à `None` indique que l'opération correspondante n'est pas disponible. Par exemple, si une classe assigne `None` à *__iter__()*, vous ne pouvez pas itérer sur la classe et appeler `iter()` sur une instance lève `TypeError` (sans se replier sur *__getitem__()*)[2].

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).

2. Les méthodes *__hash__()*, *__iter__()*, *__reversed__()* et *__contains__()* ont une gestion particulière pour cela ; les autres lèvent toujours `TypeError`, mais le font en considérant que `None` n'est pas un appelable.

3.3.1 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().__new__(cls[, ...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.

Si *__new__()* est appelée pendant la construction de l'objet et 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 au constructeur de l'objet.

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 métaclasses pour particulariser la création des classes.

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

Appelée après la création de l'instance (par *__new__()*), mais avant le retour vers l'appelant. Les arguments sont ceux passés à l'expression du constructeur de classe. Si une classe de base possède une méthode *__init__()*, la méthode *__init__()* de la classe dérivée, si elle existe, doit explicitement appeler cette méthode pour assurer une initialisation correcte de la partie classe de base de l'instance ; par exemple : `super().__init__([args...])`.

Comme *__new__()* et *__init__()* travaillent ensemble pour créer des objets (*__new__()* pour le créer, *__init__()* pour le particulariser), *__init__()* ne doit pas renvoyer de valeur `None` ; sinon une exception `TypeError` est levée à l'exécution.

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

Appelée au moment où une instance est sur le point d'être détruite. On l'appelle aussi finaliseur ou (improprement) destructeur. Si une classe de base possède une méthode *__del__()*, la méthode *__del__()* de la classe dérivée, si elle existe, doit explicitement l'appeler pour s'assurer de l'effacement correct de la partie classe de base de l'instance.

Il est possible (mais pas recommandé) que la méthode *__del__()* retarde la destruction de l'instance en créant une nouvelle référence vers cet objet. Python appelle ceci la *résurrection* d'objet. En fonction de l'implémentation, *__del__()* peut être appelée une deuxième fois au moment où l'objet ressuscité va être détruit ; l'implémentation actuelle de *CPython* ne l'appelle qu'une fois.

Il n'est pas garanti que soient appelées les méthodes *__del__()* des objets qui existent toujours quand l'interpréteur termine.

Note : `del x` n'appelle pas directement `x.__del__()` — la première décrémente le compteur de références de `x`. La seconde n'est appelée que quand le compteur de références de `x` atteint zéro.

Particularité de l'implémentation CPython : It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the *cyclic garbage collector*. A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

Voir aussi :

Documentation du module `gc`.

Avertissement : en raison des conditions particulières qui règnent quand *__del__()* est appelée, les exceptions levées pendant son exécution sont ignorées et, à la place, un avertissement est affiché sur `sys.stderr`. En particulier :

— *__del__()* peut être invoquée quand du code arbitraire est en cours d'exécution, et ce dans n'importe quel fil d'exécution. Si *__del__()* a besoin de poser un verrou ou d'accéder à tout autre ressource bloquante, elle peut provoquer un blocage mutuel (*deadlock* en anglais) car la ressource peut être déjà utilisée par le code qui est interrompu pour exécuter la méthode *__del__()*.

— *__del__()* peut être exécutée pendant que l'interpréteur se ferme. En conséquence, les variables globales auxquelles elle souhaite accéder (y compris les autres modules) peuvent déjà être détruites ou assignées à `None`. Python garantit que les variables globales dont le nom commence par un tiret bas sont supprimées de leur module avant que les autres variables globales ne le soient ; si aucune autre référence vers ces variables globales n'existe, cela peut aider à s'assurer que les modules importés soient toujours accessibles au moment où la méthode *__del__()* est appelée.

`object.`**`__repr__`**(*self*)

Appelée par la fonction native `repr()` pour calculer la représentation « officielle » en chaîne de caractères d'un objet. Tout est fait pour que celle-ci ressemble à une expression Python valide pouvant être utilisée pour recréer un objet avec la même valeur (dans un environnement donné). Si ce n'est pas possible, une chaîne de la forme `<...une description utile...>` est renvoyée. La valeur renvoyée doit être un objet chaîne de caractères. Si une classe définit *__repr__()* mais pas *__str__()*, alors *__repr__()* est aussi utilisée quand une représentation « informelle » en chaîne de caractères est demandée pour une instance de cette classe.

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*)

Appelée par `str(objet)` ainsi que les fonctions natives `format()` et `print()` pour calculer une chaîne de caractères « informelle » ou joliment mise en forme de représentation de l'objet. La valeur renvoyée doit être un objet string.

Cette méthode diffère de *object.__repr__()* car il n'est pas attendu que *__str__()* renvoie une expression Python valide : une représentation plus agréable à lire ou plus concise peut être utilisée.

L'implémentation par défaut du type natif `object` appelle *object.__repr__()*.

`object.`**`__bytes__`**(*self*)

Appelée par bytes pour calculer une représentation en chaîne *bytes* d'un objet. Elle doit renvoyer un objet `bytes`.

`object.`**`__format__`**(*self*, *format_spec*)

Appelée par la fonction native `format()` et, par extension, lors de l'évaluation de *chaînes de caractères littérales formatées* et la méthode `str.format()`. Elle produit une chaîne de caractères « formatée » représentant un objet. L'argument `format_spec` est une chaîne de caractères contenant la description des options de formatage voulues. L'interprétation de l'argument `format_spec` est laissée au type implémentant *__format__()*. Cependant, la plupart des classes délèguent le formatage aux types natifs ou utilisent une syntaxe similaire pour les options de formatage.

Lisez formatspec pour une description de la syntaxe standard du formatage.

La valeur renvoyée doit être un objet chaîne de caractères.

Modifié dans la version 3.4 : la méthode `__format__` de `object` lui-même lève une `TypeError` si vous lui passez une chaîne non vide.

Modifié dans la version 3.7 : `object.__format__(x, '')` est maintenant équivalent à `str(x)` plutôt qu'à `format(str(x), '')`.

`object.`**`__lt__`**(*self*, *other*)

`object.`**`__le__`**(*self*, *other*)

`object.`**`__eq__`**(*self*, *other*)

`object.`**`__ne__`**(*self*, *other*)

`object.`**`__gt__`**(*self*, *other*)

`object.`**`__ge__`**(*self*, *other*)

Ce sont les méthodes dites de « comparaisons riches ». La correspondance entre les symboles opérateurs et les noms de méthodes est la suivante : `x<y` appelle `x.__lt__(y)`, `x<=y` appelle `x.__le__(y)`, `x==y`

appelle `x.__eq__(y)`, `x!=y` appelle `x.__ne__(y)`, `x>y` appelle `x.__gt__(y)` et `x>=y` appelle `x.__ge__(y)`.

A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call `bool()` on the value to determine if the result is true or false.

By default, `object` implements *`__eq__()`* by using `is`, returning `NotImplemented` in the case of a false comparison : `True if x is y else NotImplemented`. For *`__ne__()`*, by default it delegates to *`__eq__()`* and inverts the result unless it is `NotImplemented`. There are no other implied relationships among the comparison operators or default implementations ; for example, the truth of `(x<y or x==y)` does not imply `x<=y`. To automatically generate ordering operations from a single root operation, see `functools.total_ordering()`.

Lisez le paragraphe *`__hash__()`* pour connaître certaines notions importantes relatives à la création d'objets *hachables* qui acceptent les opérations de comparaison personnalisées et qui sont utilisables en tant que clés de dictionnaires.

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. If the operands are of different types, and the right operand's type is a direct or indirect subclass of the left operand's type, the reflected method of the right operand has priority, otherwise the left operand's method has priority. Virtual subclassing is not considered.

When no appropriate method returns any value other than `NotImplemented`, the `==` and `!=` operators will fall back to `is` and `is not`, respectively.

`object.`**`__hash__`**`(`*`self`*`)`

Appelée par la fonction native `hash()` et par les opérations sur les membres de collections hachées (ce qui comprend `set`, `frozenset` et `dict`). La méthode `__hash__()` doit renvoyer un entier. La seule propriété requise est que les objets qui sont égaux pour la comparaison doivent avoir la même valeur de hachage ; il est conseillé de mélanger les valeurs de hachage des composants d'un objet qui jouent un rôle dans la comparaison des objets, en les emballant dans un *n*-uplet dont on calcule l'empreinte. Par exemple :

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

Note : `hash()` limite la valeur renvoyée d'un objet ayant une méthode *`__hash__()`* personnalisée à la taille d'un `Py_ssize_t`. C'est classiquement 8 octets pour une implémentation 64 bits et 4 octets sur une implémentation 32 bits. Si la méthode *`__hash__()`* d'un objet doit être interopérable sur des plateformes ayant des implémentations différentes, assurez-vous de vérifier la taille du hachage sur toutes les plateformes. Une manière facile de le faire est la suivante : `python -c "import sys; print(sys.hash_info.width)"`.

If a class does not define an *`__eq__()`* method it should not define a *`__hash__()`* operation either ; if it defines *`__eq__()`* but not *`__hash__()`*, its instances will not be usable as items in hashable collections. If a class defines mutable objects and implements an *`__eq__()`* method, it should not implement *`__hash__()`*, since the implementation of *hashable* collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

Les classes définies par l'utilisateur possèdent des méthodes *`__eq__()`* et *`__hash__()`* par défaut ; ces méthodes répondent que tous les objets sont différents (sauf avec eux-mêmes) et `x.__hash__()` renvoie une valeur telle que `x == y` implique à la fois `x is y` et `hash(x) == hash(y)`.

Une classe qui surcharge *`__eq__()`* et qui ne définit pas *`__hash__()`* a sa méthode *`__hash__()`* implicitement assignée à `None`. Quand la méthode *`__hash__()`* d'une classe est `None`, une instance de cette classe lève `TypeError` quand un programme essaie de demander son empreinte et elle est correctement identifiée comme *non hachable* quand on vérifie `isinstance(obj, collections.abc.Hashable)`.

Si une classe qui surcharge *`__eq__()`* a besoin de conserver l'implémentation de *`__hash__()`* de la classe parente, vous devez l'indiquer explicitement à l'interpréteur en définissant `__hash__ = <ClasseParente>.__hash__`.

Si une classe ne surcharge pas *__eq__()* et veut supprimer le calcul des empreintes, elle doit inclure `__hash__ = None` dans la définition de la classe. Une classe qui définit sa propre méthode *__hash__()* qui lève explicitement `TypeError` serait incorrectement identifiée comme hachable par un appel à `isinstance(obj, collections.abc.Hashable)`.

Note : par défaut, les valeurs renvoyées par *__hash__()* pour les chaînes et les *bytes* sont « salées » avec une valeur aléatoire non prévisible. Bien qu'une empreinte reste constante tout au long d'un processus Python, sa valeur n'est pas prévisible entre deux invocations de Python.

This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.

Modifier les empreintes obtenues par hachage modifie l'ordre d'itération sur les *sets*. Python n'a jamais donné de garantie sur cet ordre (d'ailleurs, l'ordre n'est pas le même entre les implémentations 32 et 64 bits).

Voir aussi `PYTHONHASHSEED`.

Modifié dans la version 3.3 : la randomisation des empreintes est activée par défaut.

`object.`**`__bool__`**`(`*self*`)`

Called to implement truth value testing and the built-in operation `bool()` ; should return `False` or `True`. 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 `__bool__()`, all its instances are considered true.

3.3.2 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*`)`

Appelée lorsque l'accès par défaut à l'attribut échoue en levant `AttributeError` (soit *__getattribute__()* lève `AttributeError` car *name* n'est pas un attribut de l'instance ou un attribut dans l'arborescence de la classe de `self`; ou *__get__()* de la propriété *name* lève `AttributeError`). Cette méthode doit renvoyer soit la valeur (calculée) de l'attribut, soit lever une exception `AttributeError`.

Notez que si l'attribut est trouvé par le mécanisme normal, *__getattr__()* n'est pas appelée (c'est une asymétrie voulue entre *__getattr__()* et *__setattr__()*). Ce comportement est adopté à la fois pour des raisons de performance et parce que, sinon, *__getattr__()* n'aurait aucun moyen d'accéder aux autres attributs de l'instance. Notez que, au moins pour ce qui concerne les variables d'instance, vous pouvez simuler un contrôle total en n'insérant aucune valeur dans le dictionnaire des attributs de l'instance (mais en les insérant dans un autre objet à la place). Lisez la partie relative à la méthode *__getattribute__()* ci-dessous pour obtenir un contrôle total effectif sur l'accès aux attributs.

`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 *Recherche des méthodes spéciales*.

Lève un événement d'audit `object.__getattr__` avec les arguments `obj` et `name`.

`object.`**`__setattr__`**`(`*self*, *name*, *value*`)`

Appelée lors d'une assignation d'attribut. Elle est appelée à la place du mécanisme normal (c'est-à-dire stocker la valeur dans le dictionnaire de l'instance). *name* est le nom de l'attribut, *value* est la valeur à assigner à cet attribut.

Si *`__setattr__()`* veut assigner un attribut d'instance, elle doit appeler la méthode de la classe de base avec le même nom, par exemple `object.__setattr__(self, name, value)`.

Lève un événement d'audit `object.__setattr__` avec les arguments `obj`, `name` et `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.

Lève un événement d'audit `object.__deltattr__` avec les arguments `obj` et `name`.

`object.`**`__dir__`**`(`*self*`)`

Called when `dir()` is called on the object. An iterable must be returned. `dir()` converts the returned iterable to a list and sorts it.

Personnalisation de l'accès aux attributs d'un module

Les noms spéciaux `__getattr__` et `__dir__` peuvent aussi être personnalisés pour accéder aux attributs du module. La fonction `__getattr__` au niveau du module doit accepter un argument qui est un nom d'attribut et doit renvoyer la valeur calculée ou lever une `AttributeError`. Si un attribut n'est pas trouvé dans l'objet module en utilisant la recherche normale, c'est-à-dire *`object.__getattribute__()`*, alors Python recherche `__getattr__` dans le `__dict__` du module avant de lever une `AttributeError`. S'il la trouve, il l'appelle avec le nom de l'attribut et renvoie le résultat.

The `__dir__` function should accept no arguments, and return an iterable of strings that represents the names accessible on module. If present, this function overrides the standard `dir()` search on a module.

Pour une personnalisation plus fine du comportement d'un module (assignation des attributs, propriétés, etc.), vous pouvez assigner l'attribut `__class__` d'un objet module à une sous-classe de `types.ModuleType`. Par exemple :

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

Note : définir `__getattr__` du module et `__class__` pour le module impacte uniquement les recherches qui utilisent la syntaxe d'accès aux attributs — accéder directement aux globales d'un module (soit par le code dans le module, soit *via* une référence au dictionnaire des variables globales du module) fonctionne toujours de la même façon.

Modifié dans la version 3.5 : l'attribut `__class__` du module est maintenant en lecture-écriture.

Added in version 3.7 : attributs `__getattr__` et `__dir__` du module.

Voir aussi :

PEP 562 — `__getattr__` et `__dir__` pour un module
Décrit les fonctions `__getattr__` et `__dir__` des modules.

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=None*`)`

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). L'argument optionnel *owner* est 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*.

Il convient que cette méthode renvoie la valeur calculée de l'attribut ou lève une exception `AttributeError`.

La **PEP 252** spécifie que *`__get__()`* soit un appelable avec un ou deux arguments. Les descripteurs natifs de Python suivent cette spécification ; cependant, il est probable que des outils tiers aient des descripteurs qui requièrent les deux arguments. L'implémentation de *`__getattribute__()`* de Python passe toujours les deux arguments, qu'ils soient requis ou non.

`object.`**`__set__`**`(`*self*, *instance*, *value*`)`

Appelée pour définir l'attribut d'une instance *instance* de la classe propriétaire à la nouvelle valeur *value*.

Notez que ajouter *`__set__()`* ou *`__delete__()`* modifie la nature du descripteur vers un « descripteur de donnée ». Reportez-vous à *Invocation des descripteurs* pour plus de détails.

`object.`**`__delete__`**`(`*self*, *instance*`)`

Appelée pour supprimer l'attribut de l'instance *instance* de la classe propriétaire.

Instances of descriptors may also have the `__objclass__` attribute present :

`object.`**`__objclass__`**

The attribute `__objclass__` is interpreted by the `inspect` module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).

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 métaclasses.

Cependant, si la valeur cherchée est un objet qui définit une des méthodes de descripteur, alors Python modifie son comportement et invoque la méthode du descripteur à la place. Le moment où cela intervient dans la recherche citée ci-dessus dépend de l'endroit où a été définie la méthode de descripteur et comment elle a été appelée.

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

Si elle est liée à un objet instance, `a.x` est transformé en l'appel suivant : `type(a).__dict__['x'].__get__(a, type(a))`.

Liaison avec une classe
: Si elle est liée à une classe, `A.x` est transformé en l'appel suivant : `A.__dict__['x'].__get__(None, A)`.

Liaison super
: Une recherche avec un point telle que `super(A, a).x` cherche `a.__class__.__mro__` pour une classe de base `B` qui suit (dans l'ordre MRO) `A`, puis renvoie `B.__dict__['x'].__get__(a, A)`. Si ce n'est pas un descripteur, `x` est renvoyé inchangé.

For instance bindings, the precedence of descriptor invocation depends on which descriptor methods are defined. A descriptor can define any combination of *__get__()*, *__set__()* and *__delete__()*. If it does not define `__get__()`, then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines `__set__()` and/or `__delete__()`, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both `__get__()` and `__set__()`, while non-data descriptors have just the `__get__()` method. Data descriptors with `__get__()` and `__set__()` (and/or `__delete__()`) defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Les méthodes Python (y compris celles décorées par `@staticmethod` et `@classmethod`) sont implémentées comme des descripteurs hors-données. 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.

créneaux prédéfinis (`__slots__`)

Les créneaux prédéfinis (`__slots__`) vous permettent de déclarer des membres d'une donnée (comme une propriété) et d'interdire la création de `__dict__` ou de *__weakref__* (à moins qu'ils ne soient explicitement déclarés dans le `__slots__` ou présent dans le parent).

L'espace gagné par rapport à l'utilisation d'un `__dict__` peut être significatif. La recherche d'attribut peut aussi s'avérer beaucoup plus rapide.

`object.`**`__slots__`**
: Cette variable de classe peut être assignée avec une chaîne, un itérable ou une séquence de chaînes avec les noms de variables utilisés par les instances. *__slots__* réserve de la place pour ces variables déclarées et interdit la création automatique de `__dict__` et *__weakref__* pour chaque instance.

Notes on using *__slots__* :

- Lorsque vous héritez d'une classe sans *__slots__*, les attributs `__dict__` et *__weakref__* des instances sont toujours accessibles.
- 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__*.
- 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__*.
- Les *__slots__* sont implémentés au niveau de la classe en créant des *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.
- L'action de la déclaration du *__slots__* ne se limite pas à la classe où il est défini. Les *__slots__* déclarés par les parents sont disponibles dans les classes enfants. Cependant, les sous-classes enfants ont un `__dict__` et un *__weakref__* à moins qu'elles ne définissent aussi un *__slots__* (qui ne doit contenir alors que les noms *supplémentaires* aux créneaux déjà prédéfinis).
- 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.
- `TypeError` will be raised if nonempty *__slots__* are defined for a class derived from a `"variable-length" built-in type` such as `int`, `bytes`, and `tuple`.

— Tout *itérable*, sauf les chaînes de caractères, peuvent être affectés à *__slots__*.
— Si vous affectez *__slots__* à un `dictionnaire`, les clés du dictionnaires seront les noms du *slot*. Les valeurs du dictionnaire peuvent être utilisées en tant que chaines de description (*docstrings*) et sont reconnues par `inspect.getdoc()` qui les affiche dans la sortie de `help()`.
— Les assignations de `__class__` ne fonctionnent que si les deux classes ont le même *__slots__*.
— L'héritage multiple avec plusieurs classes parentes qui ont des *__slots__* est possible, mais seul un parent peut avoir des attributs créés par *__slots__* (les autres classes parentes doivent avoir des *__slots__* vides). La violation de cette règle lève `TypeError`.
— Si un *itérateur* est utilisé pour *__slots__*, alors un *descripteur* est créé pour chacune des valeurs de l'itérateur. Cependant, l'attribut *__slots__* est un itérateur vide.

3.3.3 Personnalisation de la création de classes

Quand une classe hérite d'une classe parente, la méthode *`__init_subclass__()`* de la classe parente est appelée. Ainsi, il est possible d'écrire des classes qui modifient le comportement des sous-classes. Ce comportement est corrélé aux décorateurs de classes mais, alors que les décorateurs de classes agissent seulement sur la classe qu'ils décorent, `__init_subclass__` agit uniquement sur les futures sous-classes de la classe qui définit cette méthode.

classmethod `object.`**`__init_subclass__`**`(`*cls*`)`

Cette méthode est appelée quand la classe est sous-classée. *cls* est alors la nouvelle sous-classe. Si elle est définie en tant que méthode d'instance normale, cette méthode est implicitement convertie en méthode de classe.

Keyword arguments which are given to a new class are passed to the parent class's `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in :

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

L'implémentation par défaut de `object.__init_subclass__` ne fait rien sans argument, mais lève une erreur si elle est appelée avec un argument ou plus.

Note : l'indication de métaclasse `metaclass` est absorbée par le reste du mécanisme de types et n'est jamais passée à l'implémentation de `__init_subclass__`. La métaclasse réelle (plutôt que l'indication explicite) peut être récupérée par `type(cls)`.

Added in version 3.6.

Lorsqu'une classe est créée, `type.__new__()` exécute le point d'entrée `____set_name__()` de toute variable de la classe qui en possède un.

`object.`**`__set_name__`**`(`*self*, *owner*, *name*`)`

Appelée automatiquement au moment où la classe propriétaire *owner* est créée. L'objet *self* a été assigné à *name* dans *owner* :

```
class A:
    x = C()  # Automatically calls: x.__set_name__(A, 'x')
```

Si l'affectation se produit après la création de la classe, le point d'entrée *`__set_name__()`* n'est pas appelé automatiquement. Mais il est autorisé d'appeler *`__set_name__()`* manuellement :

```
class A:
   pass

c = C()
```

(suite sur la page suivante)

(suite de la page précédente)

```
A.x = c                  # The hook is not called
c.__set_name__(A, 'x')   # Manually invoke the hook
```

Consultez *Création de l'objet classe* pour davantage de détails.

Added in version 3.6.

Métaclasses

Par défaut, les classes sont construites en utilisant `type()`. Le corps de la classe est exécuté dans un nouvel espace de nommage et le nom de la classe est lié localement au résultat de `type(name, bases, namespace)`.

Le déroulement de création de la classe peut être personnalisé en passant l'argument nommé `metaclass` dans la ligne de définition de la classe ou en héritant d'une classe existante qui comporte déjà un tel argument. Dans l'exemple qui suit, `MyClass` et `MySubclass` sont des instances de `Meta` :

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Tout autre argument nommé spécifié dans la définition de la classe est passé aux opérations de métaclasses décrites auparavant.

Quand la définition d'une classe est exécutée, les différentes étapes suivies sont :

- les entrées MRO sont résolues ;
- la métaclasse appropriée est déterminée ;
- l'espace de nommage de la classe est préparé ;
- le corps de la classe est exécuté ;
- l'objet classe est créé.

Résolution des entrées MRO

`object.`**`__mro_entries__`**`(`*self*, *bases*`)`

If a base that appears in a class definition is not an instance of `type`, then an `__mro_entries__()` method is searched on the base. If an `__mro_entries__()` method is found, the base is substituted with the result of a call to `__mro_entries__()` when creating the class. The method is called with the original bases tuple passed to the *bases* parameter, and must return a tuple of classes that will be used instead of the base. The returned tuple may be empty : in these cases, the original base is ignored.

Voir aussi :

`types.resolve_bases()`
Dynamically resolve bases that are not instances of `type`.

`types.get_original_bases()`
Retrieve a class's "original bases" prior to modifications by *`__mro_entries__()`*.

PEP 560
Core support for typing module and generic types.

Détermination de la métaclasse appropriée

La métaclasse appropriée pour une définition de classe est déterminée de la manière suivante :

- si aucune classe et aucune métaclasse n'est donnée, alors `type()` est utilisée ;
- si une métaclasse explicite est donnée et que *ce n'est pas* une instance de `type()`, alors elle est utilisée directement en tant que métaclasse ;
- si une instance de `type()` est donnée comme métaclasse explicite ou si *bases* est définie, alors la métaclasse la plus dérivée est utilisée.

La métaclasse la plus dérivée est choisie à partir des métaclasses explicitement spécifiées (s'il y en a) et les métaclasses (c'est-à-dire les `type(cls)`) de toutes les classes de base spécifiées. La métaclasse la plus dérivée est celle qui est un sous-type de *toutes* ces métaclasses candidates. Si aucune des métaclasses candidates ne remplit ce critère, alors la définition de la classe échoue en levant `TypeError`.

Préparation de l'espace de nommage de la classe

Une fois que la métaclasse appropriée est identifiée, l'espace de nommage de la classe est préparé. Si la métaclasse possède un attribut `__prepare__`, il est appelé avec `namespace = metaclass.__prepare__(name, bases, **kwds)` (où les arguments nommés supplémentaires, s'il y en a, sont les arguments de la définition de la classe). La méthode `__prepare__` doit être implémentée comme une `méthode de classe`). L'espace de nommage renvoyé par `__prepare__` est passé à `__new__`, mais quand l'instance finale est créée, l'espace de nommage est copié vers un nouveau `dict`.

Si la métaclasse ne possède pas d'attribut `__prepare__`, alors l'espace de nommage de la classe est initialisé en tant que tableau de correspondances ordonné.

Voir aussi :

PEP 3115 — Métaclasses dans Python 3000
introduction de la fonction automatique `__prepare__` de l'espace de nommage

Exécution du corps de la classe

Le corps de la classe est exécuté (approximativement) avec `exec(body, globals(), namespace)`. La principale différence avec un appel normal à `exec()` est que la portée lexicale autorise le corps de la classe (y compris les méthodes) à faire référence aux noms de la portée courante et des portées externes lorsque la définition de classe a lieu dans une fonction.

Cependant, même quand une définition de classe intervient dans une fonction, les méthodes définies à l'intérieur de la classe ne peuvent pas voir les noms définis en dehors de la portée de la classe. On accède aux variables de la classe *via* le premier paramètre des méthodes d'instance ou de classe, ou *via* la référence implicite `__class__` incluse dans la portée lexicale et décrite dans la section suivante.

Création de l'objet classe

Quand l'espace de nommage a été rempli en exécutant le corps de la classe, l'objet classe est créé en appelant `metaclass(name, bases, namespace, **kwds)` (les arguments nommés supplémentaires passés ici sont les mêmes que ceux passés à `__prepare__`).

Cet objet classe est celui qui est référencé par la forme sans argument de `super()`. `__class__` est une référence implicite créée par le compilateur si une méthode du corps de la classe fait référence soit à `__class__`, soit à `super`. Ceci permet que la forme sans argument de `super()` identifie la classe en cours de définition en fonction de la portée lexicale, tandis que la classe ou l'instance utilisée pour effectuer l'appel en cours est identifiée en fonction du premier argument transmis à la méthode.

Particularité de l'implémentation CPython : dans CPython 3.6 et suivants, la cellule `__class__` est passée à la métaclasse en tant qu'entrée `__classcell__` dans l'espace de nommage de la classe. Si elle est présente, elle doit être propagée à l'appel `type.__new__` pour que la classe soit correctement initialisée. Ne pas le faire se traduit par un `RuntimeError` dans Python 3.8.

Quand vous utilisez la métaclasse par défaut `type` ou toute autre métaclasse qui finit par appeler `type.__new__`, les étapes de personnalisation supplémentaires suivantes sont suivies après la création de l'objet classe :

1) `type.__new__` récupère, dans l'espace de nommage de la classe, tous les descripteurs qui définissent une méthode *`__set_name__()`* ;
2) Toutes ces méthodes `__set_name__` sont appelées avec la classe en cours de définition et le nom assigné à chaque descripteur ;
3) La méthode automatique *`__init_subclass__()`* est appelée sur le parent immédiat de la nouvelle classe en utilisant l'ordre de résolution des méthodes.

Après la création de l'objet classe, il est passé aux décorateurs de la classe, y compris ceux inclus dans la définition de la classe (s'il y en a) et l'objet résultant est lié à l'espace de nommage local en tant que classe définie.

Quand une nouvelle classe est créée *via* `type.__new__`, l'objet fourni en tant que paramètre d'espace de nommage est copié vers un nouveau tableau de correspondances ordonné et l'objet original est laissé de côté. La nouvelle copie est encapsulée dans un mandataire en lecture seule qui devient l'attribut `__dict__` de l'objet classe.

Voir aussi :

PEP 3135 — Nouvelle méthode super
Décrit la référence à la fermeture (*closure* en anglais) de la `__class__` implicite

Cas d'utilisations des métaclasses

Les utilisations possibles des métaclasses sont immenses. Quelques pistes ont déjà été explorées comme l'énumération, la gestion des traces, le contrôle des interfaces, la délégation automatique, la création automatique de propriétés, les mandataires, les *frameworks* ainsi que le verrouillage ou la synchronisation automatique de ressources.

3.3.4 Personnalisation des instances et vérification des sous-classes

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

En particulier, la métaclasse `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, elle est 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 métaclasse) 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 des classes de base abstraites (voir le module `abc`) au langage.

3.3.5 Émulation de types génériques

Lors de l'utilisation d'*annotations de types*, il est souvent utile de *paramètrer* un *type générique* en se servant de la notation crochets de Python. Par exemple, l'annotation `list[int]` peut être utilisée pour signifier une `liste` dans laquelle tous les éléments sont de type `entiers`.

Voir aussi :

PEP 343 — Indications de types
Introduction à l'annotation de types en Python (document en anglais)

Types alias génériques
Documentation pour les objets qui représentent des classes génériques paramétrées

Generics, Types génériques définis par l'utilisateur et classe `typing.Generic` (classe de base abstraite pour les types génériques)
Documentation sur la manière d'implémenter des classes génériques qui peuvent être paramétrées à l'exécution et comprises par les vérificateurs statiques de types.

Généralement, une classe ne peut être paramétrée que si elle définit une méthode spéciale de classe `__class_getitem__()`.

classmethod `object.__class_getitem__`(*cls*, *key*)

Renvoie un objet représentant la spécialisation d'une classe générique en fonction des arguments types trouvés dans *key*.

Lorsqu'elle est définie dans une classe, `__class_getitem__()` est automatiquement une méthode de classe. Ainsi, il est superflu de la décorer avec `@classmethod` lors de sa définition.

Intention de *__class_getitem__*

Le but de *`__class_getitem__()`* est de permettre la paramétrisation à l'exécution des classes génériques de la bibliothèque standard de façon à pouvoir appliquer plus facilement des *annotations de type* à ces classes.

Pour implémenter des classes génériques particularisées pouvant être paramétrées à l'exécution, et comprises par les vérificateurs statiques de type, vous pouvez soit hériter d'une classe de la bibliothèque standard qui implémente déjà *`__class_getitem__()`*, ou hériter de `typing.Generic`, qui a sa propre implémentation de `__class_getitem__()`.

Les implémentations particularisées de *`__class_getitem__()`* sur des classes définies ailleurs que la bibliothèque standard peuvent ne pas être comprises par des vérificateurs de types tiers tels que *mypy*. L'utilisation de `__class_getitem__()` pour tout autre objectif que l'annotation de type n'est pas conseillée.

__class_getitem__ contre *__getitem__*

D'habitude, l'*indiçage* d'un objet en utilisant des crochets appelle la méthode *`__getitem__()`* de l'instance, définie dans la classe de l'objet. Cependant, si l'objet dont on cherche un indice est lui-même une classe, la méthode de classe *`__class_getitem__()`* peut être appelée à la place. `__class_getitem__()` doit renvoyer un objet GenericAlias si elle est correctement définie.

Lorsqu'on lui présente l'*expression* `obj[x]`, l'interpréteur Python suit une sorte de processus suivant pour décider s'il faut appeler *`__getitem__()`* ou *`__class_getitem__()`* :

```
from inspect import isclass

def subscribe(obj, x):
    """Return the result of the expression 'obj[x]'"""

    class_of_obj = type(obj)

    # If the class of obj defines __getitem__,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # call class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # Else, if obj is a class and defines __class_getitem__,
    # call obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # Else, raise an exception
    else:
        raise TypeError(
            f"'{class_of_obj.__name__}' object is not subscriptable"
        )
```

En Python, toutes les classes sont des instances d'autres classes. La classe d'une classe est appelée la *métaclasse* de la classe et la plupart des classes ont la classe `type` comme métaclasse. `type` ne définit pas *`__getitem__()`*, ce qui veut dire que des expressions telles que `list[int]`, `dict[str, float]` et `tuple[str, bytes]` aboutissent toutes à l'appel de *`__class_getitem__()`* :

```
>>> # list has class "type" as its metaclass, like most classes:
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" calls "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ returns a GenericAlias object:
>>> type(list[int])
<class 'types.GenericAlias'>
```

Cependant, si une classe a une métaclasse particularisée qui définit *`__getitem__()`*, l'indiçage de la classe peut conduire à un comportement différent. Un exemple peut être trouvé dans le module `enum` :

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # Enum classes have a custom metaclass:
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta defines __getitem__,
>>> # so __class_getitem__ is not called,
>>> # and the result is not a GenericAlias object:
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

Voir aussi :

PEP 560 — Gestion de base pour les types modules et les types génériques
Introduction de *`__class_getitem__()`*, et présentation des cas où un *indiçage* conduit à l'appel de `__class_getitem__()` au lieu de *`__getitem__()`*

3.3.6 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 `type(x).__call__(x, arg1, ...)`.

3.3.7 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. It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard `dictionary` objects. The `collections.abc` module provides a `MutableMapping` *abstract base 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 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 search the mapping's keys ; 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 iterate through the object's keys ; 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 *`__bool__()`* method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

Particularité de l'implémentation CPython : 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 *`__bool__()`* method.

`object.__length_hint__`(*self*)

Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer >= 0. The return value may also be `NotImplemented`, which is treated the same as if the `__length_hint__` method didn't exist at all. This method is purely an optimization and is never required for correctness.

Added in version 3.4.

Note : le découpage est effectué uniquement à l'aide des trois méthodes suivantes. Un appel comme

```
a[1:2] = b
```

est traduit en

```
a[slice(1, 2, None)] = b
```

et ainsi de suite. Les éléments manquants sont remplacés par `None`.

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

Called to implement evaluation of `self[key]`. For *sequence* types, the accepted keys should be integers. Optionally, they may support `slice` objects as well. Negative index support is also optional. If *key* is of an

inappropriate type, `TypeError` may be raised ; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For *mapping* types, if *key* is missing (not in the container), `KeyError` should be raised.

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.

Note : quand on vous *spécifiez un indice* pour une *classe*, la méthode de classe spéciale *__class_getitem__()* peut être appelée au lieu de `__getitem__()`. Reportez-vous à *__class_getitem__ contre __getitem__* pour plus de détails.

`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*)

Cette méthode est appelée quand un *itérateur* est requis pour un conteneur. Cette méthode doit renvoyer un nouvel objet itérateur qui peut itérer sur tous les objets du conteneur. Pour les tableaux de correspondances, elle doit itérer sur les clés du conteneur.

`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()`.

Les opérateurs de tests d'appartenance (*in* et *not in*) sont normalement implémentés comme des itérations sur un conteneur. 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 itérable.

`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*.

3.3.8 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.`**`__matmul__`**(*self*, *other*)
`object.`**`__truediv__`**(*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*)

Ces méthodes sont appelées pour implémenter les opérations arithmétiques binaires (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). Par exemple, pour évaluer l'expression `x + y`, où *x* est une instance d'une classe qui possède une méthode *`__add__()`*, `type(x).__add__(x, y)` est appelée. La méthode *`__divmod__()`* doit être l'équivalent d'appeler *`__floordiv__()`* et *`__mod__()`* ; elle ne doit pas être reliée à *`__truediv__()`*. Notez que *`__pow__()`* doit être définie de manière à accepter un troisième argument optionnel si la version ternaire de la fonction native `pow()` est autorisée.

If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

`object.`**`__radd__`**(*self*, *other*)
`object.`**`__rsub__`**(*self*, *other*)
`object.`**`__rmul__`**(*self*, *other*)
`object.`**`__rmatmul__`**(*self*, *other*)
`object.`**`__rtruediv__`**(*self*, *other*)
`object.`**`__rfloordiv__`**(*self*, *other*)
`object.`**`__rmod__`**(*self*, *other*)
`object.`**`__rdivmod__`**(*self*, *other*)
`object.`**`__rpow__`**(*self*, *other*[, *modulo*])
`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[3] and the operands are of different types.[4] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an *`__rsub__()`* method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns `NotImplemented`.

3. "Does not support" here means that the class has no such method, or the method returns `NotImplemented`. Do not set the method to `None` if you want to force fallback to the right operand's reflected method—that will instead have the opposite effect of explicitly *blocking* such fallback.

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

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 une implémentation différente de la méthode symétrique pour l'opération, cette méthode est 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.`**`__imatmul__`**(*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, or if that method returns `NotImplemented`, the augmented assignment falls back to the normal methods. For instance, if *x* is an instance of a class with an *`__iadd__()`* method, `x += y` is equivalent to `x = x.__iadd__(y)` . If *`__iadd__()`* does not exist, or if `x.__iadd__(y)` returns `NotImplemented`, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`. In certain situations, augmented assignment can result in unexpected errors (see faq-augmented-assignment-tuple-error), but this behavior is in fact part of the data model.

`object.`**`__neg__`**(*self*)

`object.`**`__pos__`**(*self*)

`object.`**`__abs__`**(*self*)

`object.`**`__invert__`**(*self*)

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

`object.`**`__complex__`**(*self*)

`object.`**`__int__`**(*self*)

`object.`**`__float__`**(*self*)

Appelées pour implémenter les fonctions natives `complex()`, `int()` et `float()`. Elles doivent renvoyer une valeur du type approprié.

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

Appelée pour implémenter `operator.index()` et lorsque Python a besoin de convertir sans perte un objet numérique en objet entier (pour un découpage ou dans les fonctions natives `bin()`, `hex()` et `oct()`). La présence de cette méthode indique que l'objet numérique est un type entier. Elle doit renvoyer un entier.

Si *`__int__()`*, *`__float__()`* et *`__complex__()`* ne sont pas définies, alors les fonctions natives `int()`, `float()` et `complex()` redirigent par défaut vers *`__index__()`*.

`object.`**`__round__`**(*self*[, *ndigits*])

`object.`**`__trunc__`**(*self*)

`object.`**`__floor__`**(*self*)

object.**__ceil__**(*self*)

Appelées pour implémenter la fonction native `round()` et les fonctions du module `math` `trunc()`, `floor()` et `ceil()`. À moins que *ndigits* ne soit passé à `__round__()`, toutes ces méthodes doivent renvoyer la valeur de l'objet tronquée pour donner un `Integral` (typiquement un `int`).

La fonction native `int()` se replie sur *`__trunc__()`* dans le cas où ni *`__int__()`* ni *`__index__()`* ne sont définies.

Modifié dans la version 3.11 : la délégation de `int()` vers `__trunc__()` est obsolète.

3.3.9 Gestionnaire de contexte With

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.

Note that *`__exit__()`* methods should not reraise the passed-in exception ; this is the caller's responsibility.

Voir aussi :

PEP 343 — L'instruction `with`
La spécification, les motivations et des exemples de l'instruction *`with`* en Python.

3.3.10 Arguments positionnels dans le filtrage par motif sur les classes

When using a class name in a pattern, positional arguments in the pattern are not allowed by default, i.e. `case MyClass(x, y)` is typically invalid without special support in `MyClass`. To be able to use that kind of pattern, the class needs to define a *__match_args__* attribute.

object.**__match_args__**

Cet attribut de la classe est un *n*-uplet de chaînes. Lorsque la classe apparaît dans un filtre avec des arguments positionnels, ils sont convertis en arguments nommés avec les noms du *n*-uplet, dans l'ordre. Si l'attribut n'est pas défini, tout se passe comme si sa valeur était le *n*-uplet vide `()`.

Ainsi, si `UneClasse.__match_args__` est mis à `("gauche", "milieu", "droite")`, le filtre `case UneClasse(x, y)` est équivalent à `case UneClasse(gauche=x, milieu=y)`. Le filtre doit comporter au maximum autant d'arguments positionnels que la longueur `__match_args__`. Dans le cas contraire, le filtrage lève l'exception `TypeError`.

Added in version 3.10.

Voir aussi :

PEP 634 — Filtrage par motif structurel
Spécification de l'instruction `match`.

3.3.11 Emulating buffer types

The buffer protocol provides a way for Python objects to expose efficient access to a low-level memory array. This protocol is implemented by builtin types such as `bytes` and `memoryview`, and third-party libraries may define additional buffer types.

While buffer types are usually implemented in C, it is also possible to implement the protocol in Python.

`object.__buffer__`(*self*, *flags*)

Called when a buffer is requested from *self* (for example, by the `memoryview` constructor). The *flags* argument is an integer representing the kind of buffer requested, affecting for example whether the returned buffer is read-only or writable. `inspect.BufferFlags` provides a convenient way to interpret the flags. The method must return a `memoryview` object.

`object.__release_buffer__`(*self*, *buffer*)

Called when a buffer is no longer needed. The *buffer* argument is a `memoryview` object that was previously returned by *__buffer__()*. The method must release any resources associated with the buffer. This method should return `None`. Buffer objects that do not need to perform any cleanup are not required to implement this method.

Added in version 3.12.

Voir aussi :

PEP 688 - Making the buffer protocol accessible in Python
Introduces the Python `__buffer__` and `__release_buffer__` methods.

`collections.abc.Buffer`
ABC for buffer types.

3.3.12 Recherche des méthodes spéciales

Pour les classes définies par le développeur, l'invocation implicite de méthodes spéciales n'est garantie que si ces méthodes sont définies par le type d'objet, pas dans le dictionnaire de l'objet instance. Ce comportement explique pourquoi le code suivant lève une exception :

```
>>> class C:
...     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 métaclasse » 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 métaclasse 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
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).

3.4 Coroutines

3.4.1 Objets *attendables* (*awaitable*)

Un objet *awaitable* implémente généralement une méthode *__await__()*. Les objets *coroutine* renvoyés par les fonctions *async def* sont des *attendables* (*awaitable*).

Note : les objets *itérateur de générateur* renvoyés par les générateurs décorés par `types.coroutine()` sont aussi des *attendables* (*awaitable*), mais ils n'implémentent pas *__await__()*.

`object.`**`__await__`**(*self*)

Doit renvoyer un *itérateur*. Doit être utilisé pour implémenter les objets *awaitable*. Par exemple, `asyncio.Future` implémente cette méthode pour être compatible avec les expressions *await*.

Note : The language doesn't place any restriction on the type or value of the objects yielded by the iterator returned by `__await__`, as this is specific to the implementation of the asynchronous execution framework (e.g. `asyncio`) that will be managing the *awaitable* object.

Added in version 3.5.

Voir aussi :

PEP 492 pour les informations relatives aux objets *attendables* (*awaitable*).

3.4.2 Objets coroutines

Les objets *coroutine* sont des objets *awaitable*. L'exécution d'une coroutine peut être contrôlée en appelant *__await__()* et en itérant sur le résultat. Quand la coroutine a fini de s'exécuter et termine, l'itérateur lève `StopIteration` et l'attribut `value` de l'exception contient la valeur de retour. Si la coroutine lève une exception, elle est propagée par l'itérateur. Les coroutines ne doivent pas lever directement des exceptions `StopIteration` non gérées.

Les coroutines disposent aussi des méthodes listées ci-dessous, analogues à celles des générateurs (voir *Méthodes des générateurs-itérateurs*). Cependant, au contraire des générateurs, vous ne pouvez pas itérer directement sur des coroutines.

Modifié dans la version 3.5.2 : utiliser *await* plus d'une fois sur une coroutine lève une `RuntimeError`.

`coroutine.`**`send`**`(`*value*`)`

Starts or resumes execution of the coroutine. If *value* is `None`, this is equivalent to advancing the iterator returned by *__await__()*. If *value* is not `None`, this method delegates to the *send()* method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above.

`coroutine.`**`throw`**`(`*value*`)`

`coroutine.`**`throw`**`(`*type*[, *value*[, *traceback*]]`)`

Lève l'exception spécifiée dans la coroutine. Cette méthode délègue à la méthode *throw()* de l'itérateur qui a causé la suspension de la coroutine, s'il possède une telle méthode. Sinon, l'exception est levée au point de suspension. Le résultat (valeur de retour, `StopIteration` ou une autre exception) est le même que lorsque vous itérez sur la valeur de retour de *__await__()*, décrite ci-dessus. Si l'exception n'est pas gérée par la coroutine, elle est propagée à l'appelant.

Modifié dans la version 3.12 : The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`coroutine.`**`close`**`()`

Demande à la coroutine de faire le ménage et de se terminer. Si la coroutine est suspendue, cette méthode délègue d'abord à la méthode *close()* de l'itérateur qui a causé la suspension de la coroutine, s'il possède une telle méthode. Ensuite, elle lève `GeneratorExit` au point de suspension, ce qui fait le ménage dans la coroutine immédiatement. Enfin, la coroutine est marquée comme ayant terminé son exécution, même si elle n'a jamais démarré.

Les objets coroutines sont automatiquement fermés en utilisant le processus décrit au-dessus au moment où ils sont détruits.

3.4.3 Itérateurs asynchrones

Un *itérateur asynchrone* peut appeler du code asynchrone dans sa méthode `__anext__`.

Les itérateurs asynchrones peuvent être utilisés dans des instructions *async for*.

`object.`**`__aiter__`**`(`*self*`)`

Doit renvoyer un objet *itérateur asynchrone*.

`object.`**`__anext__`**`(`*self*`)`

Doit renvoyer un *attendable* (*awaitable*) qui se traduit par la valeur suivante de l'itérateur. Doit lever une `StopAsyncIteration` quand l'itération est terminée.

Un exemple d'objet itérateur asynchrone :

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self
```

(suite sur la page suivante)

(suite de la page précédente)

```
    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Added in version 3.5.

Modifié dans la version 3.7 : avant Python 3.7, *__aiter__()* pouvait renvoyer un *attendable* (*awaitable*) qui se résolvait potentiellement en un *itérateur asynchrone*.

À partir de Python 3.7, *__aiter__()* doit renvoyer un objet itérateur asynchrone. Renvoyer autre chose entraine une erreur `TypeError`.

3.4.4 Gestionnaires de contexte asynchrones

Un *gestionnaire de contexte asynchrone* est un *gestionnaire de contexte* qui est capable de suspendre son exécution dans ses méthodes `__aenter__` et `__aexit__`.

Les gestionnaires de contexte asynchrones peuvent être utilisés dans des instructions *async with*.

`object.`**`__aenter__`**(*self*)

Semantically similar to *__enter__()*, the only difference being that it must return an *awaitable*.

`object.`**`__aexit__`**(*self*, *exc_type*, *exc_value*, *traceback*)

Semantically similar to *__exit__()*, the only difference being that it must return an *awaitable*.

Un exemple de classe de gestionnaire de contexte asynchrone :

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Added in version 3.5.

CHAPITRE 4

Modèle d'exécution

4.1 Structure d'un programme

Un programme Python est construit à partir de blocs de code. Un *bloc* est un morceau de texte de programme Python qui est exécuté en tant qu'unité. Les éléments suivants sont des blocs : un module, un corps de fonction et une définition de classe. Chaque commande écrite dans l'interpréteur interactif de Python est un bloc. Un fichier de script (un fichier donné en entrée standard à l'interpréteur ou spécifié en tant qu'argument de ligne de commande à l'interpréteur) est un bloc de code. Une commande de script (une commande spécifiée en ligne de commande à l'interpréteur avec l'option `-c`) est un bloc de code. Un module exécuté en tant que script principal (module `__main__`) depuis la ligne de commande en utilisant l'option `-m` est aussi un bloc de code. La chaîne passée en argument aux fonctions natives `eval()` et `exec()` est un bloc de code.

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.

4.2 Noms et liaisons

4.2.1 Liaisons des noms

Les *noms* sont des références aux objets. Ils sont créés lors des opérations de liaisons de noms (*name binding* en anglais).

Les noms sont liés *via* les constructions suivantes :

- paramètres formels de fonctions,
- définitions de classes,
- définitions de fonctions,
- expressions d'affectation,
- *cibles* qui sont des identifiants, lorsque c'est une affectation :
  - de l'entête d'une boucle `for`,
  - after `as` in a `with` statement, `except` clause, `except*` clause, or in the as-pattern in structural pattern matching,
  - dans un champ de recherche d'un filtrage par motifs
- des instructions `import`.
- `type` statements.
- *type parameter lists*.

L'instruction `import` sous la forme `from ... import *` lie tous les noms définis dans le module importé, sauf ceux qui commencent par le caractère souligné. Cette écriture ne peut être utilisée qu'au niveau du module.

Une cible qui apparaît dans une instruction *del* est aussi considérée comme une liaison à un nom dans ce cadre (bien que la sémantique véritable soit de délier le nom).

Chaque affectation 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).

Si un nom est lié dans un bloc, c'est une variable locale de ce bloc, à moins qu'il ne soit déclaré *nonlocal* ou *global*. Si un nom est lié au niveau du module, c'est une variable globale (les variables du bloc de code de niveau module sont locales et globales). Si une variable est utilisée dans un bloc de code alors qu'elle n'y est pas définie, c'est une *variable libre*.

Chaque occurrence d'un nom dans un programme fait référence à la *liaison* de ce nom établie par les règles de résolution des noms suivantes.

4.2.2 Résolution des noms

La *portée* définit la visibilité d'un nom dans un bloc. Si une variable locale est définie dans un bloc, sa portée comprend ce bloc. Si la définition intervient dans le bloc d'une fonction, la portée s'étend à tous les blocs contenus dans celui qui comprend la définition, à moins qu'un bloc intérieur ne définisse une autre liaison pour ce nom.

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.

Quand un nom n'est trouvé nulle part, une exception `NameError` est levée. Si la portée courante est celle d'une fonction et que le nom fait référence à une variable locale qui n'a pas encore été liée au moment où le nom est utilisé, une exception `UnboundLocalError` est levée. `UnboundLocalError` est une sous-classe de `NameError`.

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. See the FAQ entry on UnboundLocalError for examples.

If the *global* statement occurs within a block, all uses of the names specified in the statement refer to the bindings of those names 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 `builtins`. The global namespace is searched first. If the names are not found there, the builtins namespace is searched next. If the names are also not found in the builtins namespace, new variables are created in the global namespace. The global statement must precede all uses of the listed names.

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.

The *nonlocal* statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. `SyntaxError` is raised at compile time if the given name does not exist in any enclosing function scope. *Type parameters* cannot be rebound with the `nonlocal` statement.

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

Class definition blocks and arguments to `exec()` and `eval()` are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. 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 comprehensions and generator expressions, but it does not include *annotation scopes*, which have access to their enclosing class scopes. This means that the following will fail :

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

However, the following will succeed :

```
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)   # <type 'A.Nested'>
```

4.2.3 Annotation scopes

Type parameter lists and `type` statements introduce *annotation scopes*, which behave mostly like function scopes, but with some exceptions discussed below. *Annotations* currently do not use annotation scopes, but they are expected to use annotation scopes in Python 3.13 when **PEP 649** is implemented.

Annotation scopes are used in the following contexts :

- Type parameter lists for *generic type aliases*.
- Type parameter lists for *generic functions*. A generic function's annotations are executed within the annotation scope, but its defaults and decorators are not.
- Type parameter lists for *generic classes*. A generic class's base classes and keyword arguments are executed within the annotation scope, but its decorators are not.
- The bounds and constraints for type variables (*lazily evaluated*).
- The value of type aliases (*lazily evaluated*).

Annotation scopes differ from function scopes in the following ways :

- Annotation scopes have access to their enclosing class namespace. If an annotation scope is immediately within a class scope, or within another annotation scope that is immediately within a class scope, the code in the annotation scope can use names defined in the class scope as if it were executed directly within the class body. This contrasts with regular functions defined within classes, which cannot access names defined in the class scope.
- Expressions in annotation scopes cannot contain `yield`, `yield from`, `await`, or `:=` expressions. (These expressions are allowed in other scopes contained within the annotation scope.)
- Names defined in annotation scopes cannot be rebound with `nonlocal` statements in inner scopes. This includes only type parameters, as no other syntactic elements that can appear within annotation scopes can introduce new names.
- While annotation scopes have an internal name, that name is not reflected in the *__qualname__* of objects defined within the scope. Instead, the `__qualname__` of such objects is as if the object were defined in the enclosing scope.

Added in version 3.12 : Annotation scopes were introduced in Python 3.12 as part of **PEP 695**.

4.2.4 Lazy evaluation

The values of type aliases created through the `type` statement are *lazily evaluated*. The same applies to the bounds and constraints of type variables created through the *type parameter syntax*. This means that they are not evaluated when the type alias or type variable is created. Instead, they are only evaluated when doing so is necessary to resolve an attribute access.

Example :

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

Here the exception is raised only when the `__value__` attribute of the type alias or the `__bound__` attribute of the type variable is accessed.

This behavior is primarily useful for references to types that have not yet been defined when the type alias or type variable is created. For example, lazy evaluation enables creation of mutually recursive type aliases :

```
from typing import Literal

type SimpleExpr = int | Parenthesized
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

Lazily evaluated values are evaluated in *annotation scope*, which means that names that appear inside the lazily evaluated value are looked up as if they were used in the immediately enclosing scope.

Added in version 3.12.

4.2.5 Noms natifs et restrictions d'exécution

Particularité de l'implémentation CPython : L'utilisateur ne doit pas toucher à `__builtins__` ; c'est et cela doit rester réservé aux besoins de l'implémentation. Les utilisateurs qui souhaitent surcharger des valeurs dans l'espace de nommage natif doivent *`importer`* le module `builtins` et modifier ses attributs judicieusement.

L'espace de nommage natif associé à l'exécution d'un bloc de code est effectivement trouvé en cherchant le nom `__builtins__` dans l'espace de nommage globaux ; ce doit être un dictionnaire ou un module (dans ce dernier cas, le dictionnaire du module est utilisé). Par défaut, lorsque l'on se trouve dans le module `__main__`, `__builtins__` est le module natif `builtins` ; lorsque l'on se trouve dans tout autre module, `__builtins__` est un pseudonyme du dictionnaire du module `builtins` lui-même.

4.2.6 Interaction avec les fonctionnalités dynamiques

La résolution des noms de variables libres intervient à l'exécution, pas à la compilation. Cela signifie que le code suivant affiche 42 :

```
i = 10
def f():
    print(i)
i = 42
f()
```

Les fonctions `eval()` et `exec()` n'ont pas accès à l'environnement complet pour résoudre les noms. Les noms doivent être résolus dans les espaces de nommage locaux et globaux de l'appelant. Les variables libres ne sont pas résolues dans l'espace de nommage englobant le plus proche mais dans l'espace de nommage globaux [1]. Les fonctions `eval()` et `exec()` possèdent des arguments optionnels pour surcharger les espaces de nommage globaux et locaux. Si seulement un espace de nommage est spécifié, il est utilisé pour les deux.

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

4.3 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).

Quand une exception n'est pas du tout gérée, l'interpréteur termine l'exécution du programme ou retourne à la boucle interactive. Dans ces cas, il affiche une trace de la pile d'appels, sauf si l'exception est `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 non virtuelles*. L'instance peut être transmise au gestionnaire et peut apporter des informations complémentaires sur les conditions de l'exception.

Note : Les messages d'exception ne font pas partie de l'API Python. Leur contenu peut changer d'une version de Python à une autre sans avertissement et le code ne doit pas reposer sur ceux-ci s'il doit fonctionner sur plusieurs versions de l'interpréteur.

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

CHAPITRE 5

Le système d'importation

Le code Python d'un *module* peut accéder à du code d'un autre module par un mécanisme qui consiste à *importer* cet autre module. L'instruction *import* est la façon la plus courante de faire appel à ce système d'importation, mais ce n'est pas la seule. Les fonctions telles que `importlib.import_module()` et `__import__()` peuvent aussi être utilisées pour mettre en œuvre le mécanisme d'importation.

L'instruction *import* effectue deux opérations ; elle cherche le module dont le nom a été donné puis elle lie le résultat de cette recherche à un nom dans la portée locale. L'opération de recherche de l'instruction `import` consiste à appeler la fonction `__import__()` avec les arguments adéquats. La valeur renvoyée par `__import__()` est utilisée pour effectuer l'opération de liaison avec le nom fourni à l'instruction `import`. Reportez-vous à l'instruction `import` pour les détails exacts de l'opération de liaison avec le nom.

Un appel direct à `__import__()` effectue seulement la recherche du module et, s'il est trouvé, l'opération de création du module. Bien que des effets collatéraux puissent se produire, tels que l'importation de paquets parents et la mise à jour de divers caches (y compris `sys.modules`), il n'y a que l'instruction *import* qui déclenche l'opération de liaison avec le nom.

Quand une instruction *import* est exécutée, la fonction native `__import__()` est appelée. D'autres mécanismes d'appel au système d'importation (tels que `importlib.import_module()`) peuvent choisir d'ignorer `__import__()` et utiliser leurs propres solutions pour implémenter la sémantique d'importation.

Quand un module est importé pour la première fois, Python recherche le module et, s'il est trouvé, crée un objet module [1] en l'initialisant. Si le module n'est pas trouvé, une `ModuleNotFoundError` est levée. Python implémente plusieurs stratégies pour rechercher le module d'un nom donné quand le mécanisme d'importation est invoqué. Ces stratégies peuvent être modifiées et étendues par divers moyens décrits dans les sections suivantes.

Modifié dans la version 3.3 : le système d'importation a été mis à jour pour implémenter complètement la deuxième partie de la **PEP 302**. Il n'existe plus de mécanisme implicite d'importation (le système d'importation complet est exposé *via* `sys.meta_path`). En complément, la gestion des paquets dans l'espace des noms natif a été implémentée (voir la **PEP 420**).

1. Voir `types.ModuleType`.

5.1 importlib

Le module `importlib` fournit une API riche pour interagir avec le système d'importation. Par exemple, `importlib.import_module()` fournit une API (que nous vous recommandons) plus simple que la fonction native `__import__()` pour mettre en œuvre le mécanisme d'importation. Reportez-vous à la documentation de la bibliothèque `importlib` pour obtenir davantage de détails.

5.2 Les paquets

Python ne connait qu'un seul type d'objet module et tous les modules sont donc de ce type, que le module soit implémenté en Python, en C ou quoi que ce soit d'autre. Pour aider à l'organisation des modules et fournir une hiérarchie des noms, Python développe le concept de *paquets*.

Vous pouvez vous représenter les paquets comme des répertoires dans le système de fichiers et les modules comme des fichiers dans ces répertoires. Mais ne prenez pas trop cette analogie au pied de la lettre car les paquets et les modules ne proviennent pas obligatoirement du système de fichiers. Dans le cadre de cette documentation, nous utilisons cette analogie bien pratique des répertoires et des fichiers. Comme les répertoires du système de fichiers, les paquets sont organisés de manière hiérarchique et les paquets peuvent eux-mêmes contenir des sous-paquets ou des modules.

Il est important de garder à l'esprit que tous les paquets sont des modules mais que tous les modules ne sont pas des paquets. Formulé autrement, les paquets sont juste un certain type de modules. Spécifiquement, tout module qui contient un attribut `__path__` est réputé être un paquet.

Tous les modules ont un nom. Les noms des sous-paquets sont séparés du nom du paquet parent par un point (`.`), à l'instar de la syntaxe standard d'accès aux attributs en Python. Ainsi, vous pouvez avoir un paquet nommé `email` qui possède un sous-paquet nommé `email.mime` et un module dans ce sous-paquet nommé `email.mime.text`.

5.2.1 Paquets classiques

Python définit deux types de paquets, les *paquets classiques* et les *paquets espaces de nommage*. Les paquets classiques sont les paquets traditionnels tels qu'ils existaient dans Python 3.2 et antérieurs. Un paquet classique est typiquement implémenté sous la forme d'un répertoire contenant un fichier `__init__.py`. Quand un paquet classique est importé, ce fichier `__init__.py` est implicitement exécuté.

Par exemple, l'arborescence suivante définit un paquet `parent` au niveau le plus haut avec trois sous-paquets :

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

Importer `parent.one` exécute implicitement `parent/__init__.py` et `parent/one/__init__.py`. Les importations postérieures de `parent.two` ou `parent.three` respectivement exécutent `parent/two/__init__.py` ou `parent/three/__init__.py` respectivement.

5.2.2 Paquets espaces de nommage

Un paquet-espace de nommage est la combinaison de plusieurs *portions* où chaque portion fournit un sous-paquet au paquet parent. Les portions peuvent être situées à différents endroits du système de fichiers. Les portions peuvent aussi être stockées dans des fichiers zip, sur le réseau ou à tout autre endroit dans lequel Python cherche pendant l'importation. Les paquets-espaces de nommage peuvent correspondre directement à des objets du système de fichiers, ou pas ; ils peuvent être des modules virtuels qui n'ont aucune représentation concrète.

Les paquets-espaces de nommage n'utilisent pas une liste ordinaire pour leur attribut `__path__`. Ils utilisent en lieu et place un type itérable personnalisé qui effectue automatiquement une nouvelle recherche de portions de paquets à la tentative suivante d'importation dans ce paquet si le chemin de leur paquet parent (ou `sys.path` pour les paquets de plus haut niveau) change.

Pour les paquets-espaces de nommage, il n'existe pas de fichier `parent/__init__.py`. En fait, il peut y avoir plusieurs répertoires `parent` trouvés pendant le processus d'importation, où chacun est apporté par une portion différente. Ainsi, `parent/one` n'est pas forcément physiquement à côté de `parent/two`. Dans ce cas, Python crée un paquet-espace de nommage pour le paquet de plus haut niveau `parent` dès que lui ou l'un de ses sous-paquet est importé.

Voir aussi la **PEP 420** pour les spécifications des paquets-espaces de nommage.

5.3 Recherche

Pour commencer la recherche, Python a besoin du *nom qualifié* du module (ou du paquet, mais ici cela ne fait pas de différence) que vous souhaitez importer. Le nom peut être donné en argument à l'instruction *`import`* ou comme paramètre aux fonctions `importlib.import_module()` ou `__import__()`.

Le nom est utilisé dans plusieurs phases de la recherche et peut être un chemin séparé par des points pour un sous-module, par exemple `truc.machin.bidule`. Dans ce cas, Python essaie d'abord d'importer `truc` puis `truc.machin` et enfin `truc.machin.bidule`. Si n'importe laquelle des importations intermédiaires échoue, une `ModuleNotFoundError` est levée.

5.3.1 Cache des modules

Le premier endroit vérifié pendant la recherche d'une importation est `sys.modules`. Ce tableau de correspondances est utilisé comme cache de tous les modules déjà importés, y compris les chemins intermédiaires. Ainsi, si `truc.machin.bidule` a déjà été importé, `sys.modules` contient les entrées correspondantes à `truc`, `truc.machin` et `truc.machin.bidule`. À chaque chemin correspond une clé.

Pendant l'importation, le nom de module est cherché dans `sys.modules` et, s'il est trouvé, la valeur associée est le module recherché et le processus est fini. Cependant, si la valeur est `None`, alors une `ModuleNotFoundError` est levée. Si le nom du module n'est pas trouvé, Python continue la recherche du module.

`sys.modules` est accessible en lecture-écriture. Supprimer une clé peut ne pas détruire le module associé (car d'autres modules contiennent possiblement des références vers ce module), mais cela invalide l'entrée du cache pour ce nom de module. Python cherche alors un nouveau module pour ce nom. La clé peut aussi être assignée à `None` de manière à forcer une `ModuleNotFoundError` lors de la prochaine importation du module.

Attention cependant : s'il reste une référence à l'objet module et que vous invalidez l'entrée dans le cache de `sys.modules` puis ré-importez le module, les deux objets modules ne seront pas les mêmes. À l'inverse, `importlib.reload()` ré-utilise le *même* objet module et ré-initialise simplement le contenu du module en ré-exécutant le code du module.

5.3.2 Chercheurs et chargeurs

Si le module n'est pas trouvé dans `sys.modules`, alors Python utilise son protocole d'importation pour chercher et charger le module. Ce protocole se compose de deux objets conceptuels : les *chercheurs* et les *chargeurs*. Le travail du chercheur consiste à trouver, à l'aide de différentes stratégies, le module dont le nom a été fourni. Les objets qui implémentent ces deux interfaces sont connus sous le vocable « *importateurs* » (ils renvoient une référence vers eux-mêmes quand ils trouvent un module qui répond aux attentes).

Python inclut plusieurs chercheurs et importateurs par défaut. Le premier sait comment trouver les modules natifs et le deuxième sait comment trouver les modules figés. Un troisième chercheur recherche les modules dans *import path*. *import path* est une énumération sous forme de liste de chemins ou de fichiers zip. Il peut être étendu pour rechercher aussi dans toute ressource qui dispose d'un identifiant pour la localiser, une URL par exemple.

Le mécanisme d'importation est extensible, vous pouvez donc ajouter de nouveaux chercheurs pour étendre le domaine de recherche des modules.

Les chercheurs ne chargent pas les modules. S'il trouve le module demandé, un chercheur renvoie un *spécificateur de module*, qui contient toutes les informations nécessaires pour importer le module ; celui-ci sera alors utilisé par le mécanisme d'importation pour charger le module.

Les sections suivantes décrivent plus en détail le protocole utilisé par les chercheurs et les chargeurs, y compris la manière de les créer et les enregistrer pour étendre le mécanisme d'importation.

Modifié dans la version 3.4 : dans les versions précédentes de Python, les chercheurs renvoyaient directement les *chargeurs*. Dorénavant, ils renvoient des spécificateurs de modules qui *contiennent* les chargeurs. Les chargeurs sont encore utilisés lors de l'importation mais ont moins de responsabilités.

5.3.3 Points d'entrées automatiques pour l'importation

Le mécanisme d'importation est conçu pour être extensible ; vous pouvez y insérer des *points d'entrée automatique* (*hooks* en anglais). Il existe deux types de points d'entrée automatique pour l'importation : les *méta-points d'entrée* et les *points d'entrée sur le chemin des importations*.

Les méta-points d'entrée sont appelés au début du processus d'importation, juste après la vérification dans le cache `sys.modules` mais avant tout le reste. Cela permet aux méta-points d'entrée de surcharger le traitement effectué sur `sys.path`, les modules figés ou même les modules natifs. L'enregistrement des méta-points d'entrée se fait en ajoutant de nouveaux objets chercheurs à `sys.meta_path`, comme décrit ci-dessous.

Les points d'entrée sur le chemin des importations sont appelés pendant le traitement de `sys.path` (ou `package.__path__`), au moment où le chemin qui leur correspond est atteint. Les points d'entrée sur le chemin des importations sont enregistrés en ajoutant de nouveaux appelables à `sys.path_hooks`, comme décrit ci-dessous.

5.3.4 Méta-chemins

Quand le module demandé n'est pas trouvé dans `sys.modules`, Python recherche alors dans `sys.meta_path` qui contient une liste d'objets chercheurs dans des méta-chemins. Ces chercheurs sont interrogés dans l'ordre pour voir s'ils savent prendre en charge le module passé en paramètre. Les chercheurs dans les méta-chemins implémentent une méthode `find_spec()` qui prend trois arguments : un nom, un chemin d'importation et (optionnellement) un module cible. Un chercheur dans les méta-chemins peut utiliser n'importe quelle stratégie pour déterminer s'il est apte à prendre en charge le module.

Si un chercheur dans les méta-chemins sait prendre en charge le module donné, il renvoie un objet spécificateur. S'il ne sait pas, il renvoie `None`. Si le traitement de `sys.meta_path` arrive à la fin de la liste sans qu'aucun chercheur n'a renvoyé un objet spécificateur, alors une `ModuleNotFoundError` est levée. Toute autre exception levée est simplement propagée à l'appelant, mettant fin au processus d'importation.

La méthode `find_spec()` des chercheurs dans les méta-chemins est appelée avec deux ou trois arguments. Le premier est le nom complètement qualifié du module à importer, par exemple `truc.machin.bidule`. Le deuxième argument est l'ensemble des chemins dans lesquels chercher. Pour les modules de plus haut niveau, le deuxième argument est `None` mais pour les sous-modules ou les paquets, le deuxième argument est la valeur de l'attribut `__path__` du paquet parent. Si l'attribut `__path__` approprié n'est pas accessible, une `ModuleNotFoundError` est levée.

Le troisième argument est un objet module existant qui va être la cible du chargement (plus tard). Le système d'importation ne passe le module cible en paramètre que lors d'un rechargement.

Le méta-chemin peut être parcouru plusieurs fois pour une seule requête d'importation. Par exemple, si nous supposons qu'aucun des modules concernés n'a déjà été mis en cache, importer `truc.machin.bidule` effectue une première importation au niveau le plus haut, en appelant `c_m_c.find_spec("truc", None, None)` pour chaque chercheur dans les méta-chemins (`c_m_c`). Après que `truc` a été importé, `truc.machin` est importé en parcourant le méta-chemin une deuxième fois, appelant `c_m_c.find_spec("truc.machin", truc.__path__, None)`. Une fois `truc.machin` importé, le parcours final appelle `c_m_c.find_spec("truc.machin.bidule", truc.machin.__path__, None)`.

Quelques chercheurs dans les méta-chemins ne gèrent que les importations de plus haut niveau. Ces importateurs renvoient toujours `None` si on leur passe un deuxième argument autre que `None`.

Le `sys.meta_path` de Python comprend trois chercheurs par défaut : un qui sait importer les modules natifs, un qui sait importer les modules figés et un qui sait importer les modules depuis un *chemin des importations* (c'est le *chercheur dans path*).

Modifié dans la version 3.4 : The `find_spec()` method of meta path finders replaced `find_module()`, which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement `find_spec()`.

Modifié dans la version 3.10 : Use of `find_module()` by the import system now raises `ImportWarning`.

Modifié dans la version 3.12 : `find_module()` has been removed. Use `find_spec()` instead.

5.4 Chargement

Quand un spécificateur de module est trouvé, le mécanisme d'importation l'utilise (et le chargeur qu'il contient) pour charger le module. Voici à peu près ce qui se passe au sein de l'importation pendant la phase de chargement :

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # It is assumed 'exec_module' will also be defined on the loader.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)

if spec.loader is None:
    # unsupported
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

Notez les détails suivants :

— S'il existe un objet module dans `sys.modules` avec le même nom, *import* l'aurait déjà renvoyé.

— Le module existe dans `sys.modules` avant que le chargeur exécute le code du module. C'est crucial car le code du module peut (directement ou indirectement) s'importer lui-même ; l'ajouter à `sys.modules` avant évite les récursions infinies dans le pire cas et le chargement multiple dans le meilleur des cas.
— Si le chargement échoue, le module en cause (et seulement ce module) est enlevé de `sys.modules`. Tout module déjà dans le cache de `sys.modules` et tout module qui a été chargé avec succès par effet de bord doit rester dans le cache. C'est différent dans le cas d'un rechargement où même le module qui a échoué est conservé dans `sys.modules`.
— Après que le module est créé mais avant son exécution, le mécanisme d'importation définit les attributs relatifs à l'importation (`_init_module_attrs` dans l'exemple de pseudo-code ci-dessus), comme indiqué brièvement dans une *section* que nous abordons ensuite.
— L'exécution du module est le moment clé du chargement dans lequel l'espace de nommage du module est peuplé. L'exécution est entièrement déléguée au chargeur qui doit décider ce qui est peuplé et comment.
— Le modulé créé pendant le chargement et passé à `exec_module()` peut ne pas être celui qui est renvoyé à la fin de l'importation [2].

Modifié dans la version 3.4 : le système d'importation a pris en charge les responsabilités des chargeurs. Celles-ci étaient auparavant effectuées par la méthode `importlib.abc.Loader.load_module()`.

5.4.1 Chargeurs

Les chargeurs de modules fournissent la fonction critique du chargement : l'exécution du module. Le mécanisme d'importation appelle la méthode `importlib.abc.Loader.exec_module()` avec un unique argument, l'objet module à exécuter. Toute valeur renvoyée par `exec_module()` est ignorée.

Les chargeurs doivent satisfaire les conditions suivantes :
— Si le module est un module Python (par opposition aux modules natifs ou aux extensions chargées dynamiquement), le chargeur doit exécuter le code du module dans l'espace des noms globaux du module (`module.__dict__`).
— Si le chargeur ne peut pas exécuter le module, il doit lever une `ImportError`, alors que toute autre exception levée durant `exec_module()` est propagée.

Souvent, le chercheur et le chargeur sont le même objet ; dans ce cas, la méthode `find_spec()` doit juste renvoyer un spécificateur avec le chargeur défini à `self`.

Les chargeurs de modules peuvent choisir de créer l'objet module pendant le chargement en implémentant une méthode `create_module()`. Elle prend un argument, l'objet spécificateur du module et renvoie le nouvel objet du module à utiliser pendant le chargement. Notez que `create_module()` n'a besoin de définir aucun attribut sur l'objet module. Si cette méthode renvoie `None`, le mécanisme d'importation crée le nouveau module lui-même.

Added in version 3.4 : la méthode `create_module()` des chargeurs.

Modifié dans la version 3.4 : la méthode `load_module()` a été remplacée par `exec_module()` et le mécanisme d'import assume toutes les responsabilités du chargement.

Par compatibilité avec les chargeurs existants, le mécanisme d'importation utilise la méthode `load_module()` des chargeurs si elle existe et si le chargeur n'implémente pas `exec_module()`. Cependant, `load_module()` est déclarée obsolète et les chargeurs doivent implémenter `exec_module()` à la place.

La méthode `load_module()` *doit* implémenter toutes les fonctionnalités de chargement décrites ci-dessus en plus de l'exécution du module. Toutes les contraintes s'appliquent aussi, avec quelques précisions supplémentaires :
— S'il y a un objet module existant avec le même nom dans `sys.modules`, le chargeur doit utiliser le module existant (sinon, `importlib.reload()` ne fonctionnera pas correctement). Si le module considéré n'est pas trouvé dans `sys.modules`, le chargeur doit créer un nouvel objet module et l'ajouter à `sys.modules`.
— Le module *doit* exister dans `sys.modules` avant que le chargeur n'exécute le code du module, afin d'éviter les récursions infinies ou le chargement multiple.
— Si le chargement échoue, le chargeur ne doit enlever de `sys.modules` **que** le (ou les) module ayant échoué et seulement si le chargeur lui-même a chargé le module explicitement.

Modifié dans la version 3.5 : un avertissement `DeprecationWarning` est levé quand `exec_module()` est définie mais `create_module()` ne l'est pas.

2. L'implémentation de *importlib* évite d'utiliser directement la valeur de retour. À la place, elle récupère l'objet module en recherchant le nom du module dans `sys.modules`. L'effet indirect est que le module importé peut remplacer le module de même nom dans `sys.modules`. C'est un comportement spécifique à l'implémentation dont le résultat n'est pas garanti pour les autres implémentations de Python.

Modifié dans la version 3.6 : une exception `ImportError` est levée quand `exec_module()` est définie mais `create_module()` ne l'est pas.

Modifié dans la version 3.10 : l'utilisation de `load_module()` lève un `ImportWarning`.

5.4.2 Sous-modules

Quand un sous-module est chargé, quel que soit le mécanisme (par exemple avec les instructions `import`, `import-from` ou avec la fonction native `__import__()`), une liaison est créée dans l'espace de nommage du module parent vers l'objet sous-module. Par exemple, si le paquet `spam` possède un sous-module `foo`, après l'importation de `spam.foo`, `spam` possède un attribut `foo` qui est lié au sous-module. Supposons que nous ayons l'arborescence suivante :

```
spam/
    __init__.py
    foo.py
```

et que le contenu de `spam/__init__.py` contienne :

```
from .foo import Foo
```

alors exécuter les lignes suivantes crée des liens vers `foo` et `Foo` dans le module `spam` :

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.Foo
<class 'spam.foo.Foo'>
```

Connaissant la façon habituelle dont Python effectue les liens, cela peut sembler surprenant. Mais c'est en fait une fonctionnalité fondamentale du système d'importation. Si vous avez quelque part `sys.modules['spam']` et `sys.modules['spam.foo']` (comme dans c'est le cas ci-dessus après l'importation), alors le dernier doit apparaître comme l'attribut `foo` du premier.

5.4.3 Spécificateurs de modules

Le mécanisme d'importation utilise diverses informations de chaque module pendant l'importation, spécialement avant le chargement. La plupart de ces informations sont communes à tous les modules. Le but d'un spécificateur de module est d'encapsuler ces informations relatives à l'importation au sein de chaque module.

Utiliser un spécificateur pendant l'importation permet de transférer l'état entre les composants du système d'importation, par exemple entre le chercheur qui crée le spécificateur de module et le chargeur qui l'exécute. Surtout, cela permet au mécanisme d'importation d'effectuer toutes les opérations classiques de chargement, alors que c'était le chargeur qui en avait la responsabilité quand il n'y avait pas de spécificateur.

Le spécificateur de module est accessible par l'attribut `__spec__` de l'objet module. Lisez `ModuleSpec` pour davantage d'informations sur le contenu du spécificateur de module.

Added in version 3.4.

5.4.4 Attributs des modules importés

Le mécanisme d'importation renseigne ces attributs pour chaque objet module pendant le chargement, sur la base du spécificateur de module et avant que le chargeur n'exécute le module.

It is **strongly** recommended that you rely on *__spec__* and its attributes instead of any of the other individual attributes listed below.

__name__

L'attribut __name__ doit contenir le nom complètement qualifié du module. Ce nom est utilisé pour identifier de manière non équivoque le module dans le mécanisme d'importation.

__loader__

L'attribut __loader__ doit pointer vers l'objet chargeur que le mécanisme d'importation a utilisé pour charger le module. L'utilisation principale concerne l'introspection, mais il peut être utilisé pour d'autres fonctionnalités relatives au chargement. Par exemple, obtenir des données par l'intermédiaire du chargeur.

It is **strongly** recommended that you rely on *__spec__* instead of this attribute.

Modifié dans la version 3.12 : The value of __loader__ is expected to be the same as __spec__.loader. The use of __loader__ is deprecated and slated for removal in Python 3.14.

__package__

The module's __package__ attribute may be set. Its value must be a string, but it can be the same value as its __name__. When the module is a package, its __package__ value should be set to its __name__. When the module is not a package, __package__ should be set to the empty string for top-level modules, or for submodules, to the parent package's name. See **PEP 366** for further details.

This attribute is used instead of __name__ to calculate explicit relative imports for main modules, as defined in **PEP 366**.

It is **strongly** recommended that you rely on *__spec__* instead of this attribute.

Modifié dans la version 3.6 : la valeur de __package__ devrait être la même que celle de __spec__.parent.

Modifié dans la version 3.10 : ImportWarning is raised if import falls back to __package__ instead of parent.

Modifié dans la version 3.12 : Raise DeprecationWarning instead of ImportWarning when falling back to __package__.

__spec__

L'attribut __spec__ doit contenir un lien vers le spécificateur de module qui a été utilisé lors de l'importation du module. Définir __spec__ correctement s'applique aussi pour *l'initialisation des modules au démarrage de l'interpréteur*. La seule exception concerne __main__ où la valeur de __spec__ est *None dans certains cas*.

When __spec__.parent is not set, __package__ is used as a fallback.

Added in version 3.4.

Modifié dans la version 3.6 : __spec__.parent est utilisé par défaut quand __package__ n'est pas défini.

__path__

Si le module est un paquet (classique ou espace de nommage), l'attribut __path__ de l'objet module doit être défini. La valeur doit être un itérable mais peut être vide si __path__ n'a pas de sens dans le contexte. Si __path__ n'est pas vide, il doit produire des chaînes lorsque l'on itère dessus. Vous trouvez plus de détails sur la sémantique de __path__ *plus loin ci-dessous*.

Les modules qui ne sont pas des paquets ne doivent pas avoir d'attribut __path__.

__file__

__cached__

__file__ est optionnel (si elle est spécifiée, la valeur doit être une chaîne). Cela indique le chemin vers le fichier depuis lequel le module a été chargé (s'il a été chargé à partir d'un fichier) ou le chemin du fichier de la bibliothèque partagée pour les modules d'extension chargés dynamiquement depuis une bibliothèque partagée. Il se peut qu'il n'existe pas pour certains types de modules, tels que les modules C qui sont liés statiquement

à l'interpréteur, où le système d'importation le laisse indéfini parce que sa sémantique serait mauvaise (par exemple, un module chargé depuis une base de données).

Si `__file__` est défini, il peut être judicieux de définir l'attribut `__cached__` dont la valeur est le chemin vers une version compilée du code (par exemple, le fichier *bytecode*). Le fichier n'a pas besoin d'exister pour définir cet attribut : le chemin peut simplement pointer vers l'endroit où le fichier compilé aurait été placé (voir la **PEP 3147**).

Notez que `__cached__` peut être défini même si `__file__` n'est pas défini. Cependant, ce scénario semble rare. Au final, c'est le chargeur qui utilise les spécifications du module fournies par le chercheur (spécifications à partir desquelles sont dérivées `__file__` et `__cached__`). Donc, si le chargeur peut charger depuis un module mis en cache mais ne peut pas charger depuis un fichier, ce scénario a du sens.

It is **strongly** recommended that you rely on *`__spec__`* instead of `__cached__`.

5.4.5 module.__path__

Par définition, si un module possède un attribut `__path__`, c'est un paquet.

L'attribut `__path__` d'un paquet est utilisé pendant l'importation des sous-paquets. Dans le mécanisme d'importation, son fonctionnement ressemble beaucoup à `sys.path`, c'est-à-dire qu'il fournit une liste d'emplacements où rechercher les modules pendant l'importation. Cependant, `__path__` est beaucoup plus contraint que `sys.path`.

`__path__` doit être un itérable de chaînes de caractères, mais il peut être vide. Les mêmes règles que pour `sys.path` s'appliquent au `__path__` d'un paquet et `sys.path_hooks` (dont la description est donnée plus bas) est consulté pendant le parcours de `__path__` du paquet.

Le fichier `__init__.py` d'un paquet peut définir ou modifier l'attribut `__path__` d'un paquet, et c'est ainsi qu'étaient implémentés les paquets-espaces de nommage avant la **PEP 420**. Depuis l'adoption de la **PEP 420**, les paquets-espaces de nommage n'ont plus besoin d'avoir des fichiers `__init__.py` qui ne font que de la manipulation de `__path__` ; le mécanisme d'importation définit automatiquement `__path__` correctement pour un paquet-espace de nommage.

5.4.6 Représentation textuelle d'un module

Par défaut, tous les modules ont une représentation textuelle utilisable. Cependant, en utilisant les attributs définis ci-dessus et dans le spécificateur de module, vous pouvez explicitement mieux contrôler l'affichage des objets modules.

Si le module possède un spécificateur (`__spec__`), le mécanisme d'importation essaie de générer une représentation avec celui-ci. S'il échoue ou s'il n'y a pas de spécificateur, le système d'importation construit une représentation par défaut en utilisant toute information disponible sur le module. Il tente d'utiliser `module.__name__`, `module.__file__` et `module.__loader__` comme entrées pour la représentation, avec des valeurs par défaut lorsque l'information est manquante.

Les règles exactes utilisées sont :

- Si le module possède un attribut `__spec__`, la valeur est utilisée pour générer la représentation. Les attributs *name*, *loader*, *origin* et *has_location* sont consultés.
- Si le module possède un attribut `__file__`, il est utilisé pour construire la représentation du module.
- Si le module ne possède pas d'attribut `__file__` mais possède un `__loader__` qui n'est pas `None`, alors la représentation du chargeur est utilisée pour construire la représentation du module.
- Sinon, il utilise juste le `__name__` du module dans la représentation.

Modifié dans la version 3.12 : Use of `module_repr()`, having been deprecated since Python 3.4, was removed in Python 3.12 and is no longer called during the resolution of a module's repr.

5.4.7 Invalidation de *bytecode* mis en cache

Avant que Python ne charge du *bytecode* en cache à partir d'un fichier `.pyc`, il vérifie si ce cache est bien à jour par rapport au fichier source `.py`. Python effectue cette vérification en stockant l'horodatage de la dernière modification de la source ainsi que sa taille dans le fichier cache au moment où il l'écrit. À l'exécution, le système d'importation valide le fichier cache en comparant les métadonnées que le cache contient avec les métadonnées de la source.

Python gère également les fichiers caches « avec empreintes », qui stockent une empreinte (*hash* en anglais) du contenu de la source plutôt que des métadonnées. Il existe deux variations des fichiers `.pyc` avec empreintes : vérifiés et non-vérifiés. Pour les fichiers `.pyc` avec empreinte vérifiés, Python valide le fichier cache en calculant l'empreinte du fichier source et compare les empreintes. Si l'empreinte stockée dans le fichier cache est invalide, Python la recalcule et écrit un nouveau fichier cache avec empreinte. Pour les fichiers `.pyc` avec empreinte non vérifiés, Python considère simplement que le fichier cache est valide s'il existe. La validation (ou non) des fichiers `.pyc` avec empreinte peut être définie avec l'option `--check-hash-based-pycs`.

Modifié dans la version 3.7 : ajout des fichiers `.pyc` avec empreinte. Auparavant, Python gérait les caches de *bytecode* sur la base de l'horodatage.

5.5 Le chercheur dans *path*

Comme indiqué précédemment, Python est livré par défaut avec plusieurs chercheurs dans les méta-chemins. L'un deux, appelé *chercheur dans path* (`PathFinder`), recherche dans le *chemin des importations* qui contient une liste *d'entrées dans path*. Chaque entrée désigne un emplacement où rechercher des modules.

Le chercheur dans *path* en tant que tel ne sait pas comment importer quoi que ce soit. Il ne fait que parcourir chaque entrée de *path* et associe à chacune d'elle un « chercheur d'entrée dans *path* » qui sait comment gérer le type particulier de chemin considéré.

L'ensemble par défaut des « chercheurs d'entrée dans *path* » implémente toute la sémantique pour trouver des modules dans le système de fichiers, gérer des fichiers spéciaux tels que le code source Python (fichiers `.py`), le *bytecode* Python (fichiers `.pyc`) et les bibliothèques partagées (par exemple les fichiers `.so`). Quand le module `zipimport` de la bibliothèque standard le permet, les « chercheurs d'entrée dans *path* » par défaut savent aussi gérer tous ces types de fichiers (autres que les bibliothèques partagées) encapsulés dans des fichiers zip.

Les chemins ne sont pas limités au système de fichiers. Ils peuvent faire référence à des URL, des requêtes dans des bases de données ou tout autre emplacement qui peut être spécifié dans une chaîne de caractères.

Le chercheur dans *path* fournit aussi des points d'entrées (ou *hooks*) et des protocoles de manière à pouvoir étendre et personnaliser les types de chemins dans lesquels chercher. Par exemple, si vous voulez pouvoir chercher dans des URL réseau, vous pouvez écrire une fonction « point d'entrée » qui implémente la sémantique HTTP pour chercher des modules sur la toile. Ce point d'entrée (qui doit être un appelable) doit renvoyer un *chercheur d'entrée dans path* qui gère le protocole décrit plus bas et qui sera utilisé pour obtenir un chargeur de module sur la toile.

Avertissement : cette section et la précédente utilisent toutes les deux le terme *chercheur*, dans un cas *chercheur dans les méta-chemins* et dans l'autre *chercheur d'entrée dans path*. Ces deux types de chercheurs sont très similaires, gèrent des protocoles similaires et fonctionnent de manière semblable pendant le processus d'importation, mais il est important de garder à l'esprit qu'ils sont subtilement différents. En particulier, les chercheurs dans les méta-chemins opèrent au début du processus d'importation, comme clé de parcours de `sys.meta_path`.

Au contraire, les « chercheurs d'entrée dans *path* » sont, dans un sens, un détail d'implémentation du chercheur dans *path* et, en fait, si le chercheur dans *path* était enlevé de `sys.meta_path`, aucune des sémantiques des « chercheurs d'entrée dans *path* » ne serait invoquée.

5.5.1 Chercheurs d'entrée dans *path*

Le *chercheur dans path* (*path based finder* en anglais) est responsable de trouver et charger les modules et les paquets Python dont l'emplacement est spécifié par une chaîne dite *d'entrée dans path*. La plupart de ces entrées désignent des emplacements sur le système de fichiers, mais il n'y a aucune raison de les limiter à ça.

En tant que chercheur dans les méta-chemins, un *chercheur dans path* implémente le protocole `find_spec()` décrit précédemment. Cependant, il autorise des points d'entrée (*hooks* en anglais) supplémentaires qui peuvent être utilisés pour personnaliser la façon dont les modules sont trouvés et chargés depuis le *chemin des importations*.

Trois variables sont utilisées par le *chercheur dans path* : `sys.path`, `sys.path_hooks` et `sys.path_importer_cache`. L'attribut `__path__` des objets paquets est aussi utilisé. Il permet de personnaliser encore davantage le mécanisme d'importation.

`sys.path` contains a list of strings providing search locations for modules and packages. It is initialized from the `PYTHONPATH` environment variable and various other installation- and implementation-specific defaults. Entries in `sys.path` can name directories on the file system, zip files, and potentially other "locations" (see the `site` module) that should be searched for modules, such as URLs, or database queries. Only strings should be present on `sys.path`; all other data types are ignored.

Le *chercheur dans path* est un *chercheur dans les méta-chemins*, donc le mécanisme d'importation commence la recherche dans le *chemin des importations* par un appel à la méthode `find_spec()` du chercheur dans *path*, comme décrit précédemment. Quand l'argument *path* de `find_spec()` est donné, c'est une liste de chemins à parcourir, typiquement un attribut `__path__` pour une importation à l'intérieur d'un paquet. Si l'argument *path* est `None`, cela indique une importation de niveau le plus haut et `sys.path` est utilisée.

The path based finder iterates over every entry in the search path, and for each of these, looks for an appropriate *path entry finder* (`PathEntryFinder`) for the path entry. Because this can be an expensive operation (e.g. there may be `stat()` call overheads for this search), the path based finder maintains a cache mapping path entries to path entry finders. This cache is maintained in `sys.path_importer_cache` (despite the name, this cache actually stores finder objects rather than being limited to *importer* objects). In this way, the expensive search for a particular *path entry* location's *path entry finder* need only be done once. User code is free to remove cache entries from `sys.path_importer_cache` forcing the path based finder to perform the path entry search again.

Si une entrée n'est pas présente dans le cache, le chercheur dans *path* itère sur chaque *callable* de `sys.path_hooks`. Chaque *point d'entrée sur une entrée de path* de cette liste est appelé avec un unique argument, l'entrée dans laquelle chercher. L'appelable peut soit renvoyer un *chercheur d'entrée dans path* apte à prendre en charge l'entrée ou lever une `ImportError`. Une `ImportError` est utilisée par le chercheur dans *path* pour signaler que le point d'entrée n'a pas trouvé de *chercheur d'entrée dans path* pour cette *entrée*. L'exception est ignorée et l'itération sur le *chemin des importations* se poursuit. Le point d'entrée doit attendre qu'on lui passe soit une chaîne de caractères soit une chaîne d'octets ; l'encodage des chaînes d'octets est à la main du point d'entrée (par exemple, ce peut être l'encodage du système de fichiers, de l'UTF-8 ou autre chose) et, si le point d'entrée n'arrive pas à décoder l'argument, il doit lever une `ImportError`.

Si l'itération sur `sys.path_hooks` se termine sans qu'aucun *chercheur d'entrée dans path* ne soit renvoyé, alors la méthode `find_spec()` du chercheur dans *path* stocke `None` dans le `sys.path_importer_cache` (pour indiquer qu'il n'y a pas de chercheur pour cette entrée) et renvoie `None`, indiquant que ce *chercheur dans les méta-chemins* n'a pas trouvé le module.

Si un *chercheur d'entrée dans path* *est* renvoyé par un des *points d'entrée* de `sys.path_hooks`, alors le protocole suivant est utilisé pour demander un spécificateur de module au chercheur, spécificateur qui sera utilisé pour charger le module.

Le répertoire de travail courant — noté sous la forme d'une chaîne de caractères vide — est géré d'une manière légèrement différente des autres entrées de `sys.path`. D'abord, si le répertoire de travail courant s'avère ne pas exister, aucune valeur n'est stockée dans `sys.path_importer_cache`. Ensuite, la valeur pour le répertoire de travail courant est vérifiée à chaque recherche de module. Enfin, le chemin utilisé pour `sys.path_importer_cache` et renvoyée par `importlib.machinery.PathFinder.find_spec()` est le nom réel du répertoire de travail courant et non pas la chaîne vide.

5.5.2 Protocole des chercheurs d'entrée dans *path*

Afin de gérer les importations de modules, l'initialisation des paquets et d'être capables de contribuer aux portions des paquets-espaces de nommage, les chercheurs d'entrée dans *path* doivent implémenter la méthode `find_spec()`.

La méthode `find_spec()` prend deux arguments : le nom complètement qualifié du module en cours d'importation et (optionnellement) le module cible. `find_spec()` renvoie un spécificateur de module pleinement peuplé. Ce spécificateur doit avoir son chargeur (attribut `loader`) défini, à une exception près.

To indicate to the import machinery that the spec represents a namespace *portion*, the path entry finder sets `submodule_search_locations` to a list containing the portion.

Modifié dans la version 3.4 : `find_spec()` replaced `find_loader()` and `find_module()`, both of which are now deprecated, but will be used if `find_spec()` is not defined.

Les vieux chercheurs d'entrée dans *path* peuvent implémenter une des deux méthodes obsolètes à la place de `find_spec()`. Ces méthodes sont toujours prises en compte dans le cadre de la compatibilité descendante. Cependant, si `find_spec()` est implémentée par le chercheur d'entrée dans *path*, les méthodes historiques sont ignorées.

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace *portion*.

À fin de compatibilité descendante avec d'autres implémentations du protocole d'importation, beaucoup de chercheurs d'entrée dans *path* gèrent aussi la méthode traditionnelle `find_module()` que l'on trouve dans les chercheurs dans les méta-chemins. Cependant, les méthodes `find_module()` des chercheurs d'entrée dans *path* ne sont jamais appelées avec un argument *path* (il est convenu qu'elles enregistrent les informations relatives au chemin approprié au moment de leur appel initial au point d'entrée).

La méthode `find_module()` des chercheurs d'entrée dans *path* est obsolète car elle n'autorise pas le chercheur d'entrée dans *path* à contribuer aux portions d'espaces de nommage des paquets-espaces de nommage. Si à la fois `find_loader()` et `find_module()` sont définies pour un chercheur d'entrée dans *path*, le système d'importation utilise toujours `find_loader()` plutôt que `find_module()`.

Modifié dans la version 3.10 : Calls to `find_module()` and `find_loader()` by the import system will raise `ImportWarning`.

Modifié dans la version 3.12 : `find_module()` and `find_loader()` have been removed.

5.6 Remplacement du système d'importation standard

La manière la plus fiable de remplacer tout le système d'importation est de supprimer le contenu par défaut de `sys.meta_path` et de le remplacer complètement par un chercheur dans les méta-chemins sur mesure.

S'il convient juste de modifier le comportement de l'instruction import sans affecter les autres API qui utilisent le système d'importation, alors remplacer la fonction native `__import__()` peut être suffisant. Cette technique peut aussi être employée au niveau d'un module pour n'altérer le comportement des importations qu'à l'intérieur de ce module.

Pour empêcher sélectivement l'importation de certains modules par un point d'entrée placé en tête dans le méta-chemin (plutôt que de désactiver complètement le système d'importation), il suffit de lever une `ModuleNotFoundError` directement depuis `find_spec()` au lieu de renvoyer `None`. En effet, ce dernier indique que la recherche dans le méta-chemin peut continuer alors que la levée de l'exception termine immédiatement la recherche.

5.7 Importations relatives au paquet

Les importations relatives commencent par une suite de points. Un seul point avant indique une importation relative, démarrant avec le paquet actuel. Deux points ou plus avant indiquent une importation relative au parent du paquet actuel, un niveau par point avant le premier. Par exemple, en ayant le contenu suivant :

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

Dans `subpackage1/moduleX.py` ou `subpackage1/__init__.py`, les importations suivantes sont des importations relatives valides :

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

Les importations absolues peuvent utiliser soit la syntaxe `import <>`, soit `from <> import <>`, mais les importations relatives doivent seulement utiliser la deuxième forme, la raison étant :

```
import XXX.YYY.ZZZ
```

devrait exposer `XXX.YYY.ZZZ` comme une expression utilisable, mais `.moduleY` n'est pas une expression valide.

5.8 Cas particulier de `__main__`

Le module `__main__` est un cas particulier pour le système d'importation de Python. Comme indiqué par *ailleurs*, le module `__main__` est initialisé directement au démarrage de l'interpréteur, un peu comme `sys` et `builtins`. Cependant, au contraire des deux cités précédemment, ce n'est pas vraiment un module natif. Effectivement, la manière dont est initialisé `__main__` dépend des drapeaux et options avec lesquels l'interpréteur est lancé.

5.8.1 `__main__.__spec__`

En fonction de la manière dont `__main__` est initialisé, `__main__.__spec__` est défini de manière conforme ou mis à `None`.

Quand Python est démarré avec l'option `-m`, `__spec__` est défini à la valeur du spécificateur du module ou paquet correspondant. Python peuple aussi `__spec__` quand le module `__main__` est chargé en tant que partie de l'exécution d'un répertoire, d'un fichier zip ou d'une entrée de `sys.path`.

Dans les autres cas, `__main__.__spec__` est mis à `None`, car le code qui peuple `__main__` ne trouve pas de correspondance directe avec un module que l'on importe :

— invite de commande interactive
— l'option `-c`
— lecture depuis l'entrée standard
— lecture depuis un fichier de code source ou de *bytecode*

Notez que `__main__.__spec__` vaut toujours `None` dans le dernier cas, *même si* le fichier pourrait techniquement être importé directement en tant que module. Utilisez l'option `-m` si vous souhaitez disposer de métadonnées valides du module dans `__main__`.

Notez aussi que même quand `__main__` correspond à un module importable et que `__main__.__spec__` est défini en conséquence, ils seront toujours considérés comme des modules *distincts*. Cela est dû au fait que le bloc encadré par `if __name__ == "__main__":` ne s'exécute que quand le module est utilisé pour peupler l'espace de nommage de `__main__`, et pas durant une importation normale.

5.9 Références

Le mécanisme d'importation a considérablement évolué depuis les débuts de Python. La spécification des paquets originale est toujours disponible, bien que quelques détails ont changé depuis l'écriture de ce document.

La spécification originale de `sys.meta_path` se trouve dans la **PEP 302**. La **PEP 420** contient des extensions significatives.

PEP 420 introduced *namespace packages* for Python 3.3. **PEP 420** also introduced the `find_loader()` protocol as an alternative to `find_module()`.

La **PEP 366** décrit l'ajout de l'attribut `__package__` pour les importations relatives explicites dans les modules principaux.

La **PEP 328** a introduit les importations absolues et les importations relatives explicites. Elle a aussi proposé `__name__` pour la sémantique que la **PEP 366** attribuait à `__package__`.

PEP 338 définit l'exécution de modules en tant que scripts.

PEP 451 ajoute l'encapsulation dans les objets spécificateurs de l'état des importations, module par module. Elle reporte aussi la majorité des responsabilités des chargeurs vers le mécanisme d'importation. Ces changements permettent de supprimer plusieurs API dans le système d'importation et d'ajouter de nouvelles méthodes aux chercheurs et chargeurs.

Notes

CHAPITRE 6

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`.

6.1 Conversions arithmétiques

Quand la description d'un opérateur arithmétique ci-dessous utilise la phrase « les arguments numériques sont convertis vers un type commun », cela signifie que l'implémentation de l'opérateur fonctionne de la manière suivante pour les types natifs :

- 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 ;
- sinon, les deux doivent être des entiers et aucune conversion n'est nécessaire.

Des règles supplémentaires s'appliquent pour certains opérateurs (par exemple, une chaîne comme opérande de gauche pour l'opérateur `%`). Les extensions doivent définir leurs propres règles de conversion.

6.2 Atomes

Les atomes sont les éléments de base des expressions. Les atomes les plus simples sont les identifiants et les littéraux. Les expressions entre parenthèses, crochets ou accolades sont aussi classées syntaxiquement comme des atomes. La syntaxe pour les atomes est :

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

6.2.1 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* 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.

6.2.2 Littéraux

Python gère les littéraux de chaînes de caractères, de chaînes d'octets et de plusieurs autres types numériques :

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

L'évaluation d'un littéral produit un objet du type donné (chaîne de caractères, chaîne d'octets, entier, nombre à virgule flottante, nombre complexe) avec la valeur donnée. La valeur peut être approximée dans le cas des nombres à virgule flottante et des nombres imaginaires (complexes). Reportez-vous à la section *Littéraux* pour les détails.

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.

6.2.3 Formes parenthésées

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

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

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 *n*-uplet) ; sinon, elle produit l'expression elle-même (qui constitue donc elle-même la liste d'expressions).

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

Notez que les *n*-uplets ne sont pas créés par les parenthèses mais par l'utilisation de la virgule. L'exception est le *n*-uplet 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).

6.2.4 Agencements des listes, ensembles et dictionnaires

Pour construire une liste, un ensemble ou un dictionnaire, Python fournit des syntaxes spéciales dites « agencements » (*displays* en anglais), chaque agencement comportant deux variantes :

— 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   ::=   assignment_expression comp_for
comp_for        ::=   ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter       ::=   comp_for | comp_if
comp_if         ::=   "if" or_test [comp_iter]
```

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.

Cependant, à part l'expression de l'itérable dans la clause `for` la plus à gauche, la compréhension est exécutée dans une portée séparée, implicitement imbriquée. Ceci assure que les noms assignés dans la liste cible ne « fuient » pas en dehors de cette portée.

L'expression de l'itérable dans la clause `for` la plus à gauche est évaluée directement dans la portée englobante puis passée en tant qu'argument à la portée implicite imbriquée. Les clauses `for` suivantes et les filtres conditionnels de la clause `for` la plus à gauche ne peuvent pas être évalués dans la portée englobante, car ils peuvent dépendre de valeurs obtenues à partir de l'itérable le plus à gauche. Par exemple : `[x*y for x in range(10) for y in range(x, x+10)]`.

Pour assurer que le résultat de la compréhension soit un conteneur du type approprié, les expressions `yield` et `yield from` sont interdites dans la portée implicite imbriquée.

Depuis Python 3.6, dans une fonction *`async def`*, une clause `async for` peut être utilisée pour itérer sur un *itérateur asynchrone*. Une compréhension dans une fonction `async def` consiste alors à avoir, après cette expression de tête, une clause `for` ou `async for` suivie par des clauses `for` ou `async for` additionnelles facultatives et, possiblement, des expressions *`await`*. Si la compréhension contient des clauses `async for`, des expressions `await` ou d'autres compréhensions asynchrones, elle est appelée *compréhension asynchrone*. Une compréhension asynchrone peut suspendre l'exécution de la fonction coroutine dans laquelle elle apparaît. Voir aussi la **PEP 530**.

Added in version 3.6 : Les compréhensions asynchrones ont été introduites.

Modifié dans la version 3.8 : `yield` et `yield from` sont interdites dans la portée implicite imbriquée.

Modifié dans la version 3.11 : les compréhensions asynchrones sont maintenant autorisées dans les compréhensions des fonctions asynchrones. Les compréhensions englobantes deviennent implicitement asynchrones.

6.2.5 Agencements de listes

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

```
list_display   ::=   "[" [starred_list | comprehension] "]"
```

Un agencement de liste produit un nouvel objet liste, dont le contenu est spécifié soit par une liste d'expression soit par une compréhension. Quand une liste d'expressions (dont les éléments sont séparés par des virgules) est fournie, ces éléments sont évalués de la gauche vers la droite et placés dans l'objet liste, dans cet ordre. Quand c'est une compréhension qui est fournie, la liste est construite à partir des éléments produits par la compréhension.

6.2.6 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   ::=   "{" (starred_list | comprehension) "}"
```

Un agencement d'ensemble produit un nouvel objet ensemble mutable, 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.

6.2.7 Agencements de dictionnaires

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

```
dict_display        ::=   "{" [dict_item_list | dict_comprehension] "}"
dict_item_list      ::=   dict_item ("," dict_item)* [","]
dict_item           ::=   expression ":" expression | "**" or_expr
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 valeur 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 double astérisque `**` demande de *dépaqueter le dictionnaire*. L'opérande doit être un *tableau de correspondances*. Chaque élément du tableau de correspondances est ajouté au nouveau dictionnaire. Les valeurs les plus récentes remplacent les valeurs déjà définies par des couples clés-valeurs antérieurs ou par d'autres dépaquetages de dictionnaires antérieurs.

Added in version 3.5 : le dépaquetage peut se faire vers un agencement de dictionnaire, proposé à l'origine par la **PEP 448**.

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 mutables). 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.

Modifié dans la version 3.8 : Avant Python 3.8, dans les compréhensions de dictionnaires, l'ordre d'évaluation entre les clés et les valeurs n'était pas bien défini. Dans CPython, la valeur était évaluée avant la clé. À partir de la version 3.8, la clé est évaluée avant la valeur, comme proposé par la **PEP 572**.

6.2.8 Expressions génératrices

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

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

Une expression génératrice 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.

Les variables utilisées dans une expression génératrice sont évaluées paresseusement, au moment où la méthode *__next__()* de l'objet générateur est appelée (de la même manière que pour les générateurs classiques). Cependant, l'expression de l'itérable dans la clause `for` la plus à gauche est immédiatement évaluée, de manière à ce qu'une erreur dans cette partie soit signalée à l'endroit où l'expression génératrice est définie plutôt qu'à l'endroit où la première valeur est récupérée. Les clauses `for` suivantes ne peuvent pas être évaluées dans la portée implicite imbriquée car elles peuvent dépendre de valeurs obtenues à partir de boucles `for` plus à gauche. Par exemple, `(x*y for x in range(10) for y in range(x, x+10))`.

Les parenthèses peuvent être omises pour les appels qui ne possèdent qu'un seul argument. Voir la section *Appels* pour les détails.

Pour éviter d'interférer avec l'opération attendue de l'expression génératrice elle-même, les expressions `yield` et `yield from` sont interdites dans les générateurs définis de manière implicite.

Si une expression génératrice contient une ou des expressions `async for` ou *await*, elle est appelée *expression génératrice asynchrone*. Une expression génératrice asynchrone produit un nouvel objet générateur asynchrone qui est un itérateur asynchrone (voir *Itérateurs asynchrones*).

Added in version 3.6 : les expressions génératrices asynchrones ont été introduites.

Modifié dans la version 3.7 : Avant Python 3.7, les expressions génératrices asynchrones ne pouvaient apparaître que dans les coroutines *async def*. À partir de la version 3.7, toute fonction peut utiliser des expressions génératrices asynchrones.

Modifié dans la version 3.8 : `yield` et `yield from` sont interdites dans la portée implicite imbriquée.

6.2.9 Expressions `yield`

```
yield_atom        ::=  "(" yield_expression ")"
yield_from        ::=  "yield" "from" expression
yield_expression  ::=  "yield" expression_list | yield_from
```

Une expression `yield` est utilisée pour définir une *fonction génératrice* ou une *fonction génératrice asynchrone* et ne peut donc être utilisée que dans le corps de la définition d'une fonction. L'utilisation d'une expression `yield` dans le corps d'une fonction entraîne que cette fonction devient une fonction génératrice et son utilisation dans le corps d'une fonction *async def* entraine que cette fonction coroutine devient une fonction génératrice asynchrone. Par exemple :

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

En raison des effets de bords sur la portée contenant, les expressions `yield` ne sont pas autorisées dans la portée implicite utilisée dans l'implémentation des compréhensions et des expressions génératrices.

Modifié dans la version 3.8 : Les expressions `yield` sont interdites dans la portée implicite imbriquée utilisée dans l'implémentation des compréhensions et des expressions génératrices.

Les fonctions génératrices sont décrites plus loin alors que les fonctions générateurs asynchrones sont décrites sépa-

rément dans la section *Fonctions génératrices asynchrones*.

Lorsqu'une fonction génératrice est appelée, elle renvoie un itérateur que l'on appelle générateur. Ce générateur contrôle l'exécution de la fonction génératrice. L'exécution commence lorsque l'une des méthodes du générateur est appelée. À ce moment, l'exécution se déroule jusqu'à la première expression `yield`, où elle se suspend, renvoyant la valeur de *expression_list* à l'appelant du générateur ou `None` si *expression_list* est omis. Cette suspension conserve tous les états locaux, y compris les liaisons en cours des variables locales, le pointeur d'instruction, la pile d'évaluation interne et l'état de tous les gestionnaires d'exceptions. Lorsque l'exécution reprend en appelant l'une des méthodes du générateur, la fonction s'exécute exactement comme si l'expression `yield` n'avait été qu'un simple appel externe. La valeur de l'expression `yield` après reprise dépend de la méthode qui a permis la reprise de l'exécution. Si c'est *__next__()* qui a été utilisée (généralement *via* un *for* ou la fonction native `next()`) alors le résultat est `None`. Sinon, si c'est *send()* qui a été utilisée, alors le résultat est la valeur transmise à cette méthode.

Tout ceci rend les fonctions génératrices très similaires aux coroutines : elles produisent plusieurs objets *via* des expressions `yield`, elles possèdent plus qu'un seul point d'entrée et leur exécution peut être suspendue. La seule différence est qu'une fonction génératrice ne peut pas contrôler où l'exécution doit se poursuivre après une instruction `yield`; ce contrôle est toujours du ressort de l'appelant au générateur.

Les expressions `yield` sont autorisées partout dans un bloc *try*. Si l'exécution du générateur ne reprend pas avant qu'il ne soit finalisé (parce que son compteur de référence est tombé à zéro ou parce qu'il est nettoyé par le ramasse-miettes), la méthode *close()* du générateur-itérateur est appelée, ce qui permet l'exécution de toutes les clauses *finally* en attente.

L'expression passée à `yield from <expr>` doit être un itérateur. Toutes les valeurs produites par cet itérateur sont directement passées à l'appelant des méthodes du générateur courant. Toute valeur passée par *send()* ou toute exception passée par *throw()* est transmise à l'itérateur sous-jacent s'il possède les méthodes appropriées. Si ce n'est pas le cas, alors *send()* lève une `AttributeError` ou une `TypeError`, alors que *throw()* ne fait que propager l'exception immédiatement.

Quand l'itérateur sous-jacent a terminé, l'attribut `value` de l'instance `StopIteration` qui a été levée devient la valeur produite par l'expression `yield`. Elle peut être définie explicitement quand vous levez `StopIteration` ou automatiquement que le sous-itérateur est un générateur (en renvoyant une valeur par le sous-générateur).

Modifié dans la version 3.3 : `yield from <expr>` a été ajoutée pour déléguer le contrôle du flot d'exécution à un sous-itérateur.

Les parenthèses peuvent être omises quand l'expression `yield` est la seule expression à droite de l'instruction de l'instruction d'affectation.

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
Proposition d'améliorer l'API et la syntaxe des générateurs, de manière à pouvoir les utiliser comme de simples coroutines.

PEP 380 -- Syntaxe pour déléguer à un sous-générateur
Proposition d'introduire la syntaxe *yield_from*, de manière à déléguer facilement l'exécution à un sous-générateur.

PEP 525 : Générateurs asynchrones
La proposition qui a amélioré la **PEP 492** en ajoutant des capacités de générateur pour les coroutines.

Méthodes des générateurs-itérateurs

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

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__()`

Démarre l'exécution d'une fonction génératrice ou la reprend à la dernière expression `yield` exécutée. Quand une fonction génératrice est reprise par une méthode *`__next__()`*, l'expression `yield` en cours s'évalue toujours à `None`. L'exécution continue ensuite jusqu'à l'expression `yield` suivante, où le générateur est à nouveau suspendu et la valeur de *`expression_list`* est renvoyée à la méthode *`__next__()`* de l'appelant. Si le générateur termine sans donner une autre valeur, une exception `StopIteration` est levée.

Cette méthode est normalement appelée implicitement, par exemple par une boucle *`for`* ou par la fonction native `next()`.

`generator.send(`*value*`)`

Reprend l'exécution et « envoie » une valeur à la fonction génératrice. L'argument *value* devient le résultat de l'expression `yield` courante. La méthode *`send()`* renvoie la valeur suivante produite par le générateur ou lève `StopIteration` si le générateur termine sans produire de nouvelle valeur. Quand *`send()`* est utilisée pour démarrer le générateur, elle doit avoir `None` comme argument, car il n'y a aucune expression `yield` qui peut recevoir la valeur.

`generator.throw(`*value*`)`

`generator.throw(`*type*`[,` *value*`[,` *traceback*`]])`

Lève une exception à l'endroit où le générateur est en pause et renvoie la valeur suivante produite par la fonction génératrice. Si le générateur termine sans produire de nouvelle valeur, une exception `StopIteration` est levée. Si la fonction génératrice ne gère pas l'exception passée ou lève une autre exception, alors cette exception est propagée vers l'appelant.

Dans son utilisation typique, elle est appelée avec une seule instance d'exception, de façon similaire à l'utilisation du mot-clé *`raise`*.

Cependant, pour assurer la rétrocompatibilité, la deuxième signature est prise en charge, suivant une convention des anciennes versions de Python. L'argument *type* doit être une classe d'exception et *value* doit être une instance d'exception. Si *value* n'est pas fournie, le constructeur de *type* est appelé pour obtenir une instance. Si *traceback* est fournie, elle est liée sur l'exception, sinon tout attribut `__traceback__` existant stocké dans *value* est possiblement effacé.

Modifié dans la version 3.12 : The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`generator.close()`

Lève une `GeneratorExit` à l'endroit où la fonction génératrice a été mise en pause. Si la fonction génératrice termine, est déjà fermée ou lève `GeneratorExit` (parce qu'elle ne gère pas l'exception), *close* revient vers l'appelant. Si le générateur produit une valeur, une `RuntimeError` est levée. Si le générateur lève une autre exception, elle est propagée à l'appelant. La méthode *`close()`* ne fait rien si le générateur a déjà terminé en raison d'une exception ou d'une fin normale.

Exemples

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

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
```

(suite sur la page suivante)

(suite de la page précédente)

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

Pour des exemples d'utilisation de `yield from`, lisez la pep-380 dans « Les nouveautés de Python ».

Fonctions génératrices asynchrones

La présence d'une expression *yield* dans une fonction ou une méthode définie en utilisant *async def* transforme cette fonction en fonction *générateur asynchrone*.

Quand une fonction génératrice asynchrone est appelée, elle renvoie un itérateur asynchrone, autrement appelé objet générateur asynchrone. Cet objet contrôle l'exécution de la fonction génératrice. Un objet générateur asynchrone est typiquement utilisé dans une instruction *async for* à l'intérieur d'une fonction coroutine de la même manière qu'un objet générateur serait utilisé dans une instruction *for*.

L'appel d'une méthode du générateur asynchrone renvoie un objet *awaitable* et l'exécution commence au moment où l'on atteint une instruction `await` le concernant. À ce moment, l'exécution se déroule jusqu'à la première expression `yield`, où elle est suspendue et renvoie la valeur de *expression_list* à la coroutine en attente. Comme pour un générateur, la suspension signifie que tous les états locaux sont conservés, y compris les liaisons des variables locales, le pointeur d'instruction, la pile d'évaluation interne et l'état de tous les gestionnaires d'exceptions. Lorsque l'exécution reprend parce que l'appelant a atteint une instruction `await` sur l'objet suivant retourné par les méthodes du générateur asynchrone, la fonction s'exécute exactement comme si l'expression `yield` n'avait été qu'un simple appel externe. La valeur de l'expression `yield` au moment de la reprise dépend de la méthode qui a relancé l'exécution. Si c'est *__anext__()* qui a été utilisée, alors le résultat est `None`. Sinon, si c'est *asend()* qui a été utilisée, alors le résultat est la valeur transmise à cette méthode.

Si un générateur asynchrone se termine précipitamment en raison d'un *break*, de l'annulation de la tâche de l'appelant ou d'une exception, le code de nettoyage du générateur asynchrone est exécuté et lève possiblement des exceptions, accède à des variables de contexte dans un contexte inattendu — peut-être parce que la tâche de laquelle il dépend est finie, ou pendant la fermeture de la boucle d'événements quand le point d'entrée du ramasse-miettes a déjà été appelé. Afin d'éviter cette situation, l'appelant doit explicitement fermer le générateur asynchrone en appelant la méthode *aclose()* pour « finaliser » le générateur et le détacher de la boucle d'événements.

Dans une fonction génératrice asynchrone, les expressions `yield` sont autorisées n'importe où dans une construction *try*. Cependant, si l'exécution d'un générateur asynchrone n'a pas repris avant que le générateur ne soit finalisé (parce que son compteur de référence a atteint zéro ou parce qu'il est nettoyé par le ramasse-miettes), alors une expression `yield` dans une construction `try` pourrait ne pas atteindre la clause *finally* en attente. Dans ce cas, c'est la responsabilité de la boucle d'événements ou du programmateur exécutant le générateur asynchrone d'appeler la méthode *aclose()* du générateur asynchrone et d'exécuter l'objet coroutine résultant, permettant ainsi à toute clause `finally` en attente d'être exécutée.

Pour effectuer correctement la finalisation, une boucle d'événements doit définir une fonction *finalizer* qui prend un générateur-itérateur asynchrone, appelle sans doute *aclose()* et exécute la coroutine. Ce *finalizer* peut s'enregistrer en appelant `sys.set_asyncgen_hooks()`. Lors de la première itération, un générateur-itérateur asynchrone stocke le *finalizer* enregistré à appeler lors de la finalisation. Pour un exemple de référence rela-

tif à une méthode de *finalizer*, regardez l'implémentation de `asyncio.Loop.shutdown_asyncgens` dans Lib/asyncio/base_events.py.

L'expression `yield from <expr>` produit une erreur de syntaxe quand elle est utilisée dans une fonction génératrice asynchrone.

Méthodes des générateurs-itérateurs asynchrones

Cette sous-section décrit les méthodes des générateurs-itérateurs asynchrones. Elles sont utilisées pour contrôler l'exécution des fonctions génératrices.

coroutine agen.**__anext__**()

Renvoie un *awaitable* qui, quand il a la main, démarre l'exécution du générateur asynchrone ou reprend son exécution à l'endroit de la dernière expression `yield` exécutée. Quand une fonction génératrice asynchrone est reprise par une méthode *`__anext__()`*, l'expression `yield` en cours s'évalue toujours à `None` dans le *awaitable* renvoyé, et elle continue son exécution jusqu'à l'expression `yield` suivante. La valeur de *`expression_list`* de l'expression `yield` est la valeur de l'exception `StopIteration` levée par la coroutine qui termine. Si le générateur asynchrone termine sans produire d'autre valeur, le *awaitable* lève une exception `StopAsyncIteration` qui signale que l'itération asynchrone est terminée.

Cette méthode est normalement appelée implicitement par une boucle *`async for`*.

coroutine agen.**asend**(*value*)

Renvoie un *awaitable* qui, lorsqu'il a la main, reprend l'exécution du générateur asynchrone. Comme pour la méthode *`send()`* d'un générateur, elle « envoie » une valeur *value* à la fonction génératrice asynchrone et cet argument devient le résultat de l'expression `yield` courante. Le *awaitable* renvoyé par la méthode *`asend()`* renvoie la valeur suivante produite par le générateur comme valeur de l'exception `StopIteration` levée ou lève `StopAsyncIteration` si le générateur asynchrone termine sans produire de nouvelle valeur. Quand *`asend()`* est appelée pour démarrer le générateur asynchrone, l'argument doit être `None` car il n'y a pas d'expression `yield` pour recevoir la valeur.

coroutine agen.**athrow**(*value*)

coroutine agen.**athrow**(*type*[, *value*[, *traceback*]])

Renvoie un *awaitable* qui lève une exception du type `type` à l'endroit où le générateur asynchrone a été mis en pause et renvoie la valeur suivante produite par la fonction génératrice comme valeur de l'exception `StopIteration` qui a été levée. Si le générateur asynchrone termine sans produire de nouvelle valeur, une exception `StopAsyncIteration` est levée par le *awaitable*. Si la fonction génératrice ne traite pas l'exception reçue ou lève une autre exception alors, quand le *awaitable* est lancé, cette exception est propagée vers l'appelant du *awaitable*.

Modifié dans la version 3.12 : The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

coroutine agen.**aclose**()

Renvoie un *awaitable* qui, quand il s'exécute, lève une exception `GeneratorExit` dans la fonction génératrice asynchrone à l'endroit où le générateur était en pause. Si la fonction génératrice asynchrone termine normalement, est déjà fermée ou lève `GeneratorExit` (parce qu'elle ne gère pas l'exception), alors le *awaitable* renvoyé lève une exception `StopIteration`. Tout nouveau *awaitable* produit par un appel postérieur au générateur asynchrone lève une exception `StopAsyncIteration`. Si le générateur asynchrone produit une valeur, une `RuntimeError` est levée par le *awaitable*. Si le générateur asynchrone lève une autre exception, elle est propagée à l'appelant du *awaitable*. Si le générateur asynchrone a déjà terminé (soit par une exception, soit normalement), alors tout nouvel appel à *`aclose()`* renvoie un *awaitable* qui ne fait rien.

6.3 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
```

6.3.1 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, which most objects do. This object is then asked to produce the attribute whose name is the identifier. The type and value produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

This production can be customized by overriding the *__getattribute__()* method or the *__getattr__()* method. The `__getattribute__()` method is called first and either returns a value or raises `AttributeError` if the attribute is not available.

If an `AttributeError` is raised and the object has a `__getattr__()` method, that method is called as a fallback.

6.3.2 sélection (ou indiçage)

L'indiçage d'une instance de *classe conteneur* sélectionne généralement un élément du conteneur. L'indiçage d'une *classe générique* renvoie généralement un objet GenericAlias.

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

Lorsqu'on accède à l'indice d'un objet, l'interpréteur évalue la primaire et la liste d'expressions.

L'évaluation de la primaire doit produire un objet qui gère l'indiçage. Un objet est susceptible de gérer l'indiçage s'il définit la ou les deux méthodes *__getitem__()* et *__class_getitem__()*. Quand on spécifie un indice du primaire, le résultat de l'évaluation de la liste d'expression est passé à l'une de ces méthodes. Pour plus de détails sur le choix de `__class_getitem__` ou `__getitem__` pour l'appel, lisez *__class_getitem__ contre __getitem__*.

Si la liste d'expressions contient au moins une virgule, elle est considérée comme un `n-uplet` contenant les éléments de la liste d'expressions. Sinon, la liste d'expressions est évaluée à la valeur du seul membre de la liste.

Pour les objets natifs, deux types d'objets gèrent la sélection *via* *__getitem__()* :

1. 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. Un exemple de classe implémentant le concept de tableau de correspondances est la classe `dict`.
2. Si la primaire est une *séquence*, la liste d'expressions (*expression_list* dans la grammaire) doit pouvoir être évaluée comme un `entier` ou une `tranche` (comme expliqué dans la section suivante). Des exemples de classes natives implémentant le concept de séquence sont les `chaînes`, `listes` et les `n-uplets`.

The formal syntax makes no special provision for negative indices in *sequences*. However, built-in sequences all provide a *__getitem__()* method that interprets negative indices by adding the length of the sequence to the index so that, for example, `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). Since the support for negative indices and slicing occurs in the object's `__getitem__()` method, subclasses overriding this method will need to explicitly add that support.

Une `chaîne` est une espèce particulière de séquence dont les éléments sont des *caractères*. Un caractère n'est pas un type en tant que tel, c'est une chaîne de longueur un.

6.3.3 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 affectations ou les instructions `del`. La syntaxe est la suivante :

```
slicing        ::=   primary "[" slice_list "]"
slice_list     ::=   slice_item ("," slice_item)* [","]
slice_item     ::=   expression | proper_slice
proper_slice   ::=   [lower_bound] ":" [upper_bound] [ ":" [stride] ]
lower_bound    ::=   expression
upper_bound    ::=   expression
stride         ::=   expression
```

Il existe une ambigüité dans la syntaxe formelle ci-dessus : tout ce qui ressemble à une liste d'expressions (*expression_list* vue avant) ressemble aussi à une liste de tranches (*slice_list* dans la grammaire ci-dessus). En conséquence, toute sélection (*subscription* dans la grammaire) peut être interprétée comme une tranche. Plutôt que de compliquer encore la syntaxe, l'ambigüité est levée en disant que, dans ce cas, l'interprétation en tant que sélection (*subscription*) est prioritaire sur l'interprétation en tant que tranche (c'est le cas si la liste de tranches (*slice_list*) ne contient aucune tranche en tant que telle).

The semantics for a slicing are as follows. The primary is indexed (using the same `__getitem__()` method as normal subscription) 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 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.

6.3.4 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 [","] | comprehension] ")"
argument_list           ::=   positional_arguments ["," starred_and_keywords]
                              ["," keywords_arguments]
                              | starred_and_keywords ["," keywords_arguments]
                              | keywords_arguments
positional_arguments    ::=   positional_item ("," positional_item)*
positional_item         ::=   assignment_expression | "*" expression
starred_and_keywords    ::=   ("*" expression | keyword_item)
                              ("," "*" expression | "," keyword_item)*
keywords_arguments      ::=   (keyword_item | "**" expression)
                              ("," keyword_item | "," "**" expression)*
keyword_item            ::=   identifier "=" expression
```

Une virgule finale (optionnelle) peut être présente, après les arguments positionnels et nommés, mais elle n'affecte

pas la sémantique.

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 all objects having a *__call__()* method are callable). 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 nommé, 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, l'argument est placé 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.

Particularité de l'implémentation CPython : 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 fournis comme arguments nommés. 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'argument positionnel en trop).

Si un argument nommé 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 nommés 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 nommé en trop.

Si la syntaxe `*expression` apparaît dans l'appel de la fonction, `expression` doit pouvoir s'évaluer à un *itérable*. Les éléments de ces itérables sont traités comme s'ils étaient des arguments positionnels supplémentaires. Pour l'appel `f(x1, x2, *y, x3, x4)`, si *y* s'évalue comme une séquence *y1* … *yM*, c'est équivalent à un appel avec M+4 arguments positionnels *x1*, *x2*, *y1* … *yM*, *x3*, *x4*.

Une conséquence est que bien que la syntaxe `*expression` puisse apparaître *après* les arguments par nommés explicites, ils sont traités *avant* les arguments nommés (et avant tout argument `**expression` -- voir ci-dessous). Ainsi :

```
>>> 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 rare.

Si la syntaxe `**expression` apparaît dans un appel de fonction, `expression` doit pouvoir s'évaluer comme un *tableau de correspondances*, dont le contenu est traité comme des arguments par mots-clés supplémentaires. Si un paramètre correspondant à une clé a déjà été fourni (en tant qu'argument nommé explicite, en provenance d'un autre dépaquetage), une exception `TypeError` est levée.

Lorsque `**expression` est utilisée, chaque clé de ce tableau de correspondances doit être une chaîne. Chaque valeur du tableau est affectée au premier paramètre formel éligible à l'affectation par mot-clé dont le nom est égal à la clé. Une clé n'a pas besoin d'être un identifiant Python (par exemple, `"max-temp °F"` est acceptable, bien qu'elle ne corresponde à aucun paramètre formel qui pourrait être déclaré). S'il n'y a pas de correspondance avec un paramètre formel, la paire clé-valeur est collectée par le paramètre `**`, s'il y en a un. S'il n'y a pas de paramètre `**`, une exception `TypeError` est levée.

Les paramètres formels qui utilisent la syntaxe `*identifier` ou `**identifier` ne peuvent pas être utilisés comme arguments positionnels ou comme noms d'arguments par mots-clés.

Modifié dans la version 3.5 : Les appels de fonction acceptent n'importe quel nombre de dépaquetages par `*` ou `**`. Des arguments positionnels peuvent suivre les dépaquetages d'itérables (`*`) et les arguments par mots-clés peuvent suivre les dépaquetages de dictionnaires (`**`). Proposé pour la première fois par la **PEP 448**.

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 :
: The class must define a `__call__()` method ; the effect is then the same as if that method was called.

6.4 Expression `await`

Suspend l'exécution de la *coroutine* sur un objet *awaitable*. Ne peut être utilisée qu'à l'intérieur d'une *coroutine function*.

```
await_expr   ::=   "await" primary
```

Added in version 3.5.

6.5 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   ::=   (await_expr | 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`.

L'opérateur puissance possède la même sémantique que la fonction native `pow()` lorsqu'elle est appelée avec deux arguments : il produit son argument de gauche élevé à la puissance de son argument de droite. Les arguments numé-

riques sont d'abord convertis vers un type commun et le résultat est de ce type.

Pour les opérandes entiers, le résultat est du même type à moins que le deuxième argument ne soit négatif ; dans ce cas, tous les arguments sont convertis en nombres à virgule flottante et le résultat est un nombre à virgule flottante. Par exemple, `10**2` renvoie `100` mais `10**-2` renvoie `0.01`.

Élever `0.0` à une puissance négative entraîne une `ZeroDivisionError`. Élever un nombre négatif à une puissance fractionnaire renvoie un nombre `complexe` (dans les versions antérieures, cela levait une `ValueError`).

This operation can be customized using the special *__pow__()* method.

6.6 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
```

The unary - (minus) operator yields the negation of its numeric argument ; the operation can be overridden with the *__neg__()* special method.

The unary + (plus) operator yields its numeric argument unchanged ; the operation can be overridden with the *__pos__()* special method.

The unary ~ (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers or to custom objects that override the *__invert__()* special method.

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

6.7 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 "@" m_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
```

L'opérateur `*` (multiplication) produit le produit de ses arguments. Les deux arguments doivent être des nombres ou alors le premier argument doit être un entier et l'autre doit être une séquence. Dans le premier cas, les nombres sont convertis dans un type commun puis sont multipliés entre eux. Dans le dernier cas, la séquence est répétée ; une répétition négative produit une séquence vide.

This operation can be customized using the special *__mul__()* and *__rmul__()* methods.

L'opérateur `@` (prononcé *at* en anglais) a vocation à multiplier des matrices. Aucun type Python natif n'implémente cet opérateur.

Added in version 3.5.

Les opérateurs `/` (division) et `//` (division entière ou *floor division* en anglais) produisent le quotient de leurs arguments. Les arguments numériques sont d'abord convertis vers un type commun. La division d'entiers produit un nombre à virgule flottante alors que la division entière d'entiers produit un entier ; le résultat est celui de la division mathématique suivie de la fonction `floor` appliquée au résultat. Une division par zéro lève une exception `ZeroDivisionError`.

This operation can be customized using the special *__truediv__()* and *__floordiv__()* methods.

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 second opérande (ou zéro) ; la valeur absolue du résultat est strictement inférieure à la valeur absolue du second opérande [1].

Les opérateurs division entière et modulo sont liés par la relation suivante : `x == (x//y)*y + (x%y)`. La division entière et le module sont aussi liés à la fonction native `divmod()` : `divmod(x, y) == (x//y, x%y)` [2].

En plus de calculer le modulo sur les nombres, l'opérateur `%` est aussi surchargé par les objets chaînes de caractères pour effectuer le formatage de chaîne « à l'ancienne ». La syntaxe pour le formatage de chaînes est décrit dans la référence de la bibliothèque Python, dans la section old-string-formatting.

The *modulo* operation can be customized using the special *__mod__()* method.

L'opérateur de division entière, l'opérateur modulo et la fonction `divmod()` ne sont pas définis pour les nombres complexes. À la place, vous pouvez, si cela a du sens pour ce que vous voulez faire, les convertir vers des nombres à virgule flottante en utilisant la fonction `abs()`.

L'opérateur + (addition) produit la somme de ses arguments. Les arguments doivent être tous les deux des nombres ou des séquences du même type. Dans le premier cas, les nombres sont convertis vers un type commun puis sont additionnés entre eux. Dans le dernier cas, les séquences sont concaténées.

This operation can be customized using the special *__add__()* and *__radd__()* methods.

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

This operation can be customized using the special *__sub__()* method.

6.8 Opérations de décalage

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

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

Ces opérateurs prennent des entiers comme arguments. Ils décalent le premier argument vers la gauche ou vers la droite du nombre de bits donné par le deuxième argument.

This operation can be customized using the special *__lshift__()* and *__rshift__()* methods.

Un décalage à droite de *n* bits est défini comme la division entière par `pow(2,n)`. Un décalage à gauche de *n* bits est défini comme la multiplication par `pow(2,n)`.

1. 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.

2. Si x est très proche d'un multiple entier de y, il est possible que `x/y` soit supérieur de un par rapport à `(x-x%y)//y` en raison des arrondis. Dans de tels cas, Python renvoie le second résultat afin d'avoir `divmod(x,y)[0] * y + x % y` le plus proche de `x`.

6.9 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 integers or one of them must be a custom object overriding *__and__()* or *__rand__()* special methods.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding *__xor__()* or *__rxor__()* special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding *__or__()* or *__ror__()* special methods.

6.10 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 donnent des valeurs booléennes (`True` ou `False`). Cependant, les *méthodes de comparaison riche* définies par l'utilisateur peuvent renvoyer des non-booléens. Dans ce cas, le résultat de la comparaison est converti en booléen avec `bool()` dans les contextes qui attendent un booléen.

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).

6.10.1 Comparaisons de valeurs

Les opérateurs <, >, ==, >=, <= et != comparent les valeurs de deux objets. Les objets n'ont pas besoin d'être du même type.

Le chapitre *Objets, valeurs et types* indique que les objets ont une valeur (en plus d'un type et d'un identifiant). La valeur d'un objet est une notion plutôt abstraite en Python : par exemple, il n'existe pas de méthode canonique pour accéder à la valeur d'un objet. De la même manière, il n'y a aucune obligation concernant la construction de la valeur d'un objet, par exemple qu'elle prenne en compte toutes les données de ses attributs. Les opérateurs de comparaison implémentent une notion particulière de ce qu'est la valeur d'un objet. Vous pouvez vous le représenter comme une définition indirecte de la valeur d'un objet, *via* l'implémentation de leur comparaison.

Because all types are (direct or indirect) subtypes of `object`, they inherit the default comparison behavior from `object`. Types can customize their comparison behavior by implementing *rich comparison methods* like

`__lt__()`, described in *Personnalisation de base*.

Le comportement par défaut pour le test d'égalité (== et !=) se base sur les identifiants des objets. Ainsi, un test d'égalité entre deux instances qui ont le même identifiant est vrai, un test d'égalité entre deux instances qui ont des identifiants différents est faux. La raison de ce choix est que Python souhaite que tous les objets soient réflexifs, c'est-à-dire que `x is y` implique `x == y`.

La relation d'ordre (<, >, <= et >=) n'est pas fournie par défaut ; une tentative se solde par une `TypeError`. La raison de ce choix est qu'il n'existe pas d'invariant similaire à celui de l'égalité.

Le comportement du test d'égalité par défaut, à savoir que les instances avec des identités différentes ne sont jamais égales, peut être en contradiction avec les types qui définissent la « valeur » d'un objet et se basent sur cette « valeur » pour l'égalité. De tels types doivent personnaliser leurs tests de comparaison et, en fait, c'est ce qu'ont fait un certain nombre de types natifs.

La liste suivante décrit le comportement des tests d'égalité pour les types natifs les plus importants.

- Beaucoup de types numériques natifs (typesnumeric) et de types de la bibliothèque standard `fractions.Fraction` ainsi que `decimal.decimal` peuvent être comparés, au sein de leur propre classe ou avec d'autres objets de classes différentes. Une exception notable concerne les nombres complexes qui ne gèrent pas la relation d'ordre. Dans les limites des types concernés, la comparaison mathématique équivaut à la comparaison algorithmique, sans perte de précision.
  Les valeurs non numériques `float('NaN')` et `decimal.Decimal('NaN')` sont spéciales : toute comparaison entre un nombre et une valeur non numérique est fausse. Une implication contre-intuitive à cela est que les valeurs non numériques ne sont pas égales à elles-mêmes. Par exemple, avec `x = float('NaN')`, les expressions `3 < x`, `x < 3` et `x == x` sont toutes fausses, mais l'expression `x != x` est vraie. Ce comportement est en accord avec IEEE 754.
- `None` and `NotImplemented` are singletons. **PEP 8** advises that comparisons for singletons should always be done with `is` or `is not`, never the equality operators.
- Les séquences binaires (instances du type `bytes` ou `bytearray`) peuvent être comparées au sein de la classe et entre classes. La comparaison est lexicographique, en utilisant la valeur numérique des éléments.
- Les chaînes de caractères (instances de `str`) respectent l'ordre lexicographique en utilisant la valeur Unicode (le résultat de la fonction native `ord()`) des caractères [3].
  Les chaînes de caractères et les séquences binaires ne peuvent pas être comparées directement.
- Les séquences (instances de `tuple`, `list` ou `range`) peuvent être comparées uniquement entre instances de même type, en sachant que les intervalles (*range*) ne gèrent pas la relation d'ordre. Le test d'égalité entre ces types renvoie faux et une comparaison entre instances de types différents lève une `TypeError`.
  Les séquences se comparent lexicographiquement en comparant les éléments correspondants. Les conteneurs natifs supposent généralement que les objets identiques sont égaux à eux-mêmes. Cela leur permet d'économiser les tests d'égalité pour des objets identiques afin d'améliorer les performances et de conserver leurs invariants internes.
  L'ordre lexicographique pour les collections natives fonctionne comme suit :
  - Deux collections sont égales si elles sont du même type, ont la même longueur et si les éléments correspondants de chaque paire sont égaux. Par exemple, `[1,2] == (1,2)` est faux car les types sont différents.
  - Les collections qui gèrent la relation d'ordre sont ordonnées comme leur premier élément différent (par exemple, `[1,2,x] <= [1,2,y]` a la même valeur que `x <= y`). Si un élément n'a pas de correspondant, la collection la plus courte est la plus petite (par exemple, `[1,2] < [1,2,3]` est vrai).
- Les tableaux de correspondances (instances de `dict`) sont égales si et seulement si toutes leurs paires `(clé, valeur)` sont égales. L'égalité des clés et des valeurs met en œuvre la réflexivité.
  Les comparaisons (<, >, <= et >=) lèvent `TypeError`.

3. Le standard Unicode distingue les *points codes* (*code points* en anglais, par exemple *U+0041*) et les *caractères abstraits* (*abstract characters* en anglais, par exemple « LATIN CAPITAL LETTER A »). Bien que la plupart des caractères abstraits de l'Unicode ne sont représentés que par un seul point code, il y a un certain nombre de caractères abstraits qui peuvent être représentés par une séquence de plus qu'un point code. Par exemple, le caractère abstrait « LATIN CAPITAL LETTER C WITH CEDILLA » peut être représenté comme un unique *caractère précomposé* au point code *U+00C7*, ou en tant que séquence d'un *caractère de base* à la position *U+0043* (LATIN CAPITAL LETTER C) du code, suivi par un *caractère combiné* à la position *U+0327* (*COMBINING CEDILLA*) du code.

   Les opérateurs de comparaison des chaînes opèrent au niveau des points codes Unicode. Cela peut être déroutant pour des humains. Par exemple, `"\u00C7" == "\u0043\u0327"` renvoie `False`, bien que les deux chaînes représentent le même caractère abstrait "LATIN CAPITAL LETTER C WITH CEDILLA".

   Pour comparer des chaînes au niveau des caractères abstraits (afin d'avoir quelque chose d'intuitif pour les humains), utilisez `unicodedata.normalize()`.

— Les ensembles (instances de `set` ou `frozenset`) peuvent être comparés au sein de leur propre type et entre types différents.
Les opérateurs d'inclusion et de sur-ensemble sont définis. Ces relations ne sont pas des relations d'ordre total (par exemple, les deux ensembles `{1,2}` et `{2,3}` ne sont pas égaux, l'un n'est pas inclus dans l'autre, l'un n'est pas un sur-ensemble de l'autre). Ainsi, les ensembles ne sont pas des arguments appropriés pour les fonctions qui dépendent d'un ordre total (par exemple, les fonctions `min()`, `max()` et `sorted()` produisent des résultats indéfinis si on leur donne des listes d'ensembles en entrée).
La comparaison des ensembles met en œuvre la réflexivité des éléments.

— La plupart des autres types natifs n'implémentent pas de méthodes de comparaisons, ils héritent donc du comportement par défaut.

Les classes définies par l'utilisateur qui particularisent les opérations de comparaison doivent, si possible, respecter quelques règles pour la cohérence :

— Le test d'égalité doit être réflexif. En d'autres termes, des objets identiques doivent être égaux :

`x is y` implique `x == y`

— La comparaison doit être symétrique. En d'autres termes, les expressions suivantes doivent donner le même résultat :

`x == y` et `y == x`
`x != y` et `y != x`
`x < y` et `y > x`
`x <= y` et `y >= x`

— La comparaison doit être transitive. Les exemples suivants (liste non exhaustive) illustrent ce concept :

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

— Si vous inversez la comparaison, cela doit en produire la négation booléenne. En d'autres termes, les expressions suivantes doivent produire le même résultat :

`x == y` et `not x != y`
`x < y` et `not x >= y` (pour une relation d'ordre total)
`x > y` et `not x <= y` (pour une relation d'ordre total)

Ces deux dernières expressions s'appliquent pour les collections totalement ordonnées (par exemple, les séquences mais pas les ensembles ou les tableaux de correspondances). Regardez aussi le décorateur `total_ordering()`.

— Le résultat de `hash()` doit être cohérent avec l'égalité. Les objets qui sont égaux doivent avoir la même empreinte ou être marqués comme non-hachables.

Python ne vérifie pas ces règles de cohérence. En fait, l'utilisation de valeurs non numériques est un exemple de non-respect de ces règles.

6.10.2 Opérations de tests d'appartenance à un ensemble

Les opérateurs *`in`* et *`not in`* testent l'appartenance. `x in s` s'évalue à `True` si *x* appartient à *s* et à `False` sinon. `x not in s` renvoie la négation de `x in s`. Tous les types séquences et ensembles natifs gèrent ces opérateurs, ainsi que les dictionnaires pour lesquels `in` teste si dictionnaire possède une clé donnée. Pour les types conteneurs tels que les listes, *n*-uplets (*tuple*), ensembles (*set*), ensembles figés (*frozen set*), dictionnaires (*dict*) ou *collections.deque*, l'expression `x in y` est équivalente à `any(x is e or x == e for e in y)`.

Pour les chaînes de caractères et chaînes d'octets, `x in y` vaut `True` si et seulement si *x* est une sous-chaîne de *y*. Un test équivalent est `y.find(x) != -1`. Une chaîne vide est considérée comme une sous-chaîne de toute autre chaîne, ainsi `"" in "abc"` renvoie `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`, for which the expression `x is z or x == z` is true, is produced while iterating over `y`. If an exception is raised during the iteration, it is as if *`in`* raised that exception.

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 is y[i] or x == y[i]`, and no lower integer index raises the `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*.

6.10.3 Comparaisons d'identifiants

Les opérateurs *is* et *is not* testent l'égalité des identifiants des objets : `x is y` est vrai si et seulement si *x* et *y* sont le même objet. L'identifiant d'un objet est déterminé en utilisant la fonction `id()`. `x is not y` renvoie le résultat contraire de l'égalité des identifiants [4].

6.11 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. User-defined objects can customize their truth value by providing a *__bool__()* method.

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.

Notez que ni *and* ni *or* ne restreignent la valeur et le type qu'ils renvoient à `False` et `True` : ils renvoient le dernier argument évalué. Ceci peut être utile, par exemple : si une chaîne `s` doit être remplacée par une valeur par défaut si elle est vide, l'expression `s or 'truc'` produit la valeur voulue. Comme *not* doit créer une nouvelle valeur, il renvoie une valeur booléenne quel que soit le type de son argument (par exemple, `not 'truc'` produit `False` plutôt que `''`.

6.12 Expressions d'affectation

```
assignment_expression  ::=  [identifier ":="] expression
```

Une expression d'affectation (parfois aussi appelée « expression nommée » ou « expression morse ») affecte l'*expression* à un *identifiant* et renvoie la valeur de l'*expression*.

Une utilisation classique concerne les correspondances d'expressions rationnelles :

```
if matching := pattern.search(data):
    do_something(matching)
```

Ou lorsqu'on traite le contenu d'un fichier par morceaux :

4. 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.

```
while chunk := file.read(9000):
    process(chunk)
```

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and `assignment` statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

Added in version 3.8 : Voir la **PEP 572** pour plus de détails sur les expressions d'affectation.

6.13 Expressions conditionnelles

```
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.

L'expression `x if C else y` commence par évaluer la condition *C*. Si *C* est vrai, alors *x* est évalué et sa valeur est renvoyée ; sinon, *y* est évalué et sa valeur est renvoyée.

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

6.14 Expressions lambda

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

Les expressions lambda sont utilisées pour créer des fonctions anonymes. L'expression `lambda parameters: expression` produit un objet fonction. Cet objet anonyme se comporte comme un objet fonction défini par :

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

Voir la section *Définition de fonctions* pour la syntaxe des listes de paramètres. Notez que les fonctions créées par des expressions lambda ne peuvent pas contenir d'instructions ou d'annotations.

6.15 Listes d'expressions

```
expression_list      ::=   expression ("," expression)* [","]
starred_list         ::=   starred_item ("," starred_item)* [","]
starred_expression   ::=   expression | (starred_item ",")* [starred_item]
starred_item         ::=   assignment_expression | "*" or_expr
```

Sauf lorsqu'elle fait partie d'un agencement de liste ou d'ensemble, une liste d'expressions qui contient au moins une virgule produit un *n*-uplet. La longueur du *n*-uplet est le nombre d'expressions dans la liste. Les expressions sont évaluées de la gauche vers la droite.

Un astérisque `*` indique *dépaquetage d'itérable* (*iterable unpacking* en anglais). Son opérande doit être un *iterable*. L'itérable est développé en une séquence d'éléments qui sont inclus dans un nouvel objet *n*-uplet, liste ou ensemble à l'emplacement du dépaquetage.

Added in version 3.5 : dépaquetage d'itérables dans les listes d'expressions, proposé à l'origine par la **PEP 448**.

A trailing comma is required only to create a one-item tuple, such as `1,` ; it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses : `()`.)

6.16 Ordre d'évaluation

Python évalue les expressions de la gauche vers la droite. Remarquez que lors de l'évaluation d'une affectation, la partie droite de l'affectation est évaluée avant la partie gauche.

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
```

6.17 Priorités des opérateurs

Le tableau suivant résume les priorités des opérateurs en Python, du plus prioritaire (portée la plus courte) au moins prioritaire (portée la plus grande). Les opérateurs qui sont dans la même case ont la même priorité. À moins que la syntaxe ne soit explicitement indiquée, les opérateurs sont binaires. Les opérateurs dans la même case regroupent de la gauche vers la droite (sauf pour la puissance et les expressions conditionnelles qui regroupent de la droite vers la gauche).

Notez que les comparaisons, les tests d'appartenance et les tests d'identifiants possèdent tous la même priorité et s'enchaînent de la gauche vers la droite comme décrit dans la section *Comparaisons*.

Opérateur	Description
`(expressions…)`, `[expressions…]`, `{key: value…}`, `{expressions…}`	Expression de liaison ou parenthèse, affichage de liste, affichage de dictionnaire, affichage de *set*
`x[indice]`, `x[indice:indice]`, `x(arguments…)`, `x.attribut`	indiçage, tranches, appel, référence à un attribut
`await x`	Expression `await`
`**`	Puissance [5]
`+x`, `-x`, `~x`	NOT (positif, négatif, bit à bit)
`*`, `@`, `/`, `//`, `%`	Multiplication, multiplication de matrices, division, division entière, reste [6]
`+`, `-`	Addition et soustraction
`<<`, `>>`	décalages
`&`	AND (bit à bit)
`^`	XOR (bit à bit)
`|`	OR (bit à bit)
`in`, *`not in`*, *`is`*, *`is not`*, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparaisons, y compris les tests d'appartenance et les tests d'identifiants
`not x`	NOT (booléen)
`and`	AND (booléen)
`or`	OR (booléen)
`if` -- `else`	Expressions conditionnelles
`lambda`	Expression lambda
`:=`	Expression d'affectation

Notes

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

CHAPITRE 7

Les instructions simples

Une instruction simple est contenue dans une seule ligne logique. Plusieurs instructions simples peuvent être écrites sur une seule ligne, séparées par des points-virgules. La syntaxe d'une instruction simple est :

```
simple_stmt   ::=   expression_stmt
                  | assert_stmt
                  | assignment_stmt
                  | augmented_assignment_stmt
                  | annotated_assignment_stmt
                  | pass_stmt
                  | del_stmt
                  | return_stmt
                  | yield_stmt
                  | raise_stmt
                  | break_stmt
                  | continue_stmt
                  | import_stmt
                  | future_stmt
                  | global_stmt
                  | nonlocal_stmt
                  | type_stmt
```

7.1 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   ::=   starred_expression
```

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

En mode interactif, si la valeur n'est pas `None`, elle est convertie en chaîne en utilisant la fonction native `repr()` et

la chaîne résultante est écrite sur la sortie standard sur sa propre ligne. Si le résultat est `None`, rien n'est écrit ce qui est usuel pour les appels de procédures.

7.2 Les assignations

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

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

Voir la section *Primaires* pour la définition des syntaxes de *attributeref*, *subscription* et *slicing*.

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 mutable (une référence à un attribut, une sélection ou une tranche), l'objet mutable 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*).

L'assignation d'un objet à une liste cible, optionnellement entourée par des parenthèses ou des crochets, est définie récursivement comme suit.

- Si la liste cible est une cible unique sans virgule de fin, optionnellement entre parenthèses, l'objet est assigné à cette cible.
- Sinon :
  - Si la liste cible contient une cible préfixée par un astérisque, appelée cible *étoilée* (*starred target* en anglais) : l'objet doit être un itérable avec au moins autant d'éléments qu'il y a de cibles dans la liste cible, moins un. Les premiers éléments de l'itérable sont assignés, de la gauche vers la droite, aux cibles avant la cible étoilée. Les éléments de queue de l'itérable sont assignés aux cibles après la cible étoilée. Une liste des éléments restants dans l'itérable est alors assignée à la cible étoilée (cette liste peut être vide).
  - Sinon : l'objet doit être un itérable avec le même nombre d'éléments qu'il y a de cibles dans la liste cible ; les éléments sont assignés, de la gauche vers la droite, vers les cibles correspondantes.

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

- Si la cible est une variable (un nom) :
  - si le nom n'apparaît pas dans une instruction `global` ou `nonlocal` (respectivement) du bloc de code courant, le nom est lié à l'objet dans l'espace courant des noms locaux ;
  - sinon le nom est lié à l'objet dans l'espace des noms globaux ou dans un espace de nommage plus large déterminé par `nonlocal`, respectivement.

  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).
- 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()`.

— Si la cible est une sélection : l'expression primaire de la référence est évaluée. Elle doit produire soit un objet séquence mutable (telle qu'une liste) ou un objet tableau de correspondances (tel qu'un dictionnaire). Ensuite, l'expression de la sélection est évaluée.
Si la primaire est un objet séquence mutable (telle qu'une liste), la sélection doit produire un entier. S'il est négatif, la longueur de la séquence lui est ajoutée. La valeur résultante doit être un entier positif ou nul, plus petit que la longueur de la séquence, et Python demande à la séquence d'assigner l'objet à l'élément se trouvant à cet indice. Si l'indice est hors limites, une `IndexError` est levée (une assignation à une sélection dans une séquence ne peut pas ajouter de nouveaux éléments à une liste).
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.
For user-defined objects, the *`__setitem__()`* method is called with appropriate arguments.

— Si la cible est une tranche : l'expression primaire de la référence est évaluée. Elle doit produire un objet séquence mutable (telle qu'une liste). L'objet assigné doit être un objet séquence du même type. Ensuite, les expressions de la borne inférieure et de la borne supérieure sont évaluées, dans la mesure où elles sont spécifiées (les valeurs par défaut sont zéro et la longueur de la séquence). Les bornes doivent être des entiers. Si une borne est négative, la longueur de la séquence lui est ajoutée. Les bornes résultantes sont coupées pour être dans l'intervalle zéro -- longueur de la séquence, inclus. Finalement, Python demande à l'objet séquence de remplacer la tranche avec les éléments de la séquence à assigner. La longueur de la tranche peut être différente de la longueur de la séquence à assigner, ce qui modifie alors la longueur de la séquence cible, si celle-ci le permet.

Particularité de l'implémentation CPython : 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.

Bien que la définition de l'assignation implique que le passage entre le côté gauche et le côté droit soient « simultanés » (par exemple, `a, b = b, a` permute les deux variables), le passage *à l'intérieur* des collections de variables que l'on assigne intervient de la gauche vers la droite, ce qui peut entraîner quelques confusions. Par exemple, le programme suivant affiche `[0, 2]` :

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

Voir aussi :

PEP 3132 -- dépaquetage étendu d'itérable
Spécification de la fonctionnalité `*cible`.

7.2.1 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                       ::=   "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                                  | ">>=" | "<<=" | "&=" | "^=" | "|="
```

Voir la section *Primaires* pour la définition des syntaxes des trois derniers symboles.

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.

Au contraire des assignations normales, les assignations augmentées évaluent la partie gauche *avant* d'évaluer la partie droite. Par exemple, `a[i] += f(x)` commence par s'intéresser à `a[i]`, puis Python évalue `f(x)`, effectue l'addition et, enfin, écrit le résultat dans `a[i]`.

À l'exception de l'assignation de *n*-uplets 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.

7.2.2 Les assignations annotées

Une assignation *annotée* est la combinaison, dans une seule instruction, d'une annotation de variable ou d'attribut et d'une assignation optionnelle :

```
annotated_assignment_stmt   ::=   augtarget ":" expression
                                  ["=" (starred_expression | yield_expression)]
```

La différence avec une assignation normale (voir *ci-dessus*) est qu'une seule cible est autorisée.

Pour des noms simples en tant que cibles d'assignation, dans une portée de classe ou de module, les annotations sont évaluées et stockées dans un attribut de classe ou de module spécial, `__annotations__`, qui est un dictionnaire dont les clés sont les noms de variables (réécrits si le nom est privé) et les valeurs sont les annotations. Cet attribut est accessible en écriture et est automatiquement créé au démarrage de l'exécution du corps de la classe ou du module, si les annotations sont trouvées statiquement.

Pour les expressions en tant que cibles d'assignations, les annotations sont évaluées dans la portée de la classe ou du module, mais ne sont pas stockées.

Si le nom est annoté dans la portée d'une fonction, alors ce nom est local à cette portée. Les annotations ne sont jamais évaluées et stockées dans les portées des fonctions.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last *__setitem__()* or *__setattr__()* call.

Voir aussi :

PEP 526 -- Syntaxe pour les annotations de variables
La proposition qui a ajouté la syntaxe pour annoter les types de variables (y compris les variables de classe et les variables d'instance), au lieu de les exprimer par le biais de commentaires.

PEP 484 -- Indices de type
La proposition qui a ajouté le module `typing` pour fournir une syntaxe standard pour les annotations de type qui peuvent être utilisées dans les outils d'analyse statique et les environnements de développement intégrés (*EDI*).

Modifié dans la version 3.8 : Dorénavant, côté droit des assignations annotées, peuvent figurer les mêmes expressions que pour les assignations normales. Auparavant, certaines expressions (comme des *n*-uplets sans parenthèse englobante) généraient des erreurs de syntaxe.

7.3 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.

7.4 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)
```

7.5 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.

La suppression d'un nom détruit le lien entre ce nom dans l'espace des noms locaux, ou l'espace des noms globaux si ce nom apparaît dans une instruction `global` dans le même bloc de code. Si le nom n'est pas lié, une exception `NameError` est levée.

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).

Modifié dans la version 3.2 : Auparavant, il était illégal de supprimer un nom dans l'espace des noms locaux si celui-ci apparaissait comme variable libre dans un bloc imbriqué.

7.6 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.

Dans une fonction génératrice, l'instruction `return` indique que le générateur est terminé et provoque la levée d'une `StopIteration`. La valeur de retour (s'il y en a une) est utilisée comme argument pour construire l'exception `StopIteration` et devient l'attribut `StopIteration.value`.

Dans une fonction génératrice asynchrone, une instruction `return` vide indique que le générateur asynchrone est terminé et provoque la levée d'une `StopAsyncIteration`. Une instruction `return` non vide est une erreur de syntaxe dans une fonction génératrice asynchrone.

7.7 L'instruction `yield`

```
yield_stmt   ::=   yield_expression
```

L'instruction `yield` est sémantiquement équivalente à une *expression yield*. L'instruction *yield* peut être utilisée pour omettre les parenthèses qui seraient autrement requises dans l'instruction équivalente d'expression *yield*. Par exemple, les instructions *yield* :

```
yield <expr>
yield from <expr>
```

sont équivalentes aux instructions expressions *yield* :

```
(yield <expr>)
(yield from <expr>)
```

Les expressions et les instructions *yield* sont utilisées seulement dans la définition des fonctions *générateurs* et apparaissent uniquement dans le corps de la fonction génératrice. L'utilisation de *yield* dans la définition d'une fonction est suffisant pour que cette définition crée une fonction génératrice au lieu d'une fonction normale.

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

7.8 L'instruction `raise`

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

Si aucune expression n'est présente, `raise` propage l'exception en cours de traitement, aussi dénommée *exception active*. Si aucune exception n'est active, une exception `RuntimeError` est levée, indiquant que c'est une erreur.

Sinon, `raise` évalue la première expression en tant qu'objet exception. Ce doit être une sous-classe ou une instance de `BaseException`. Si c'est une classe, l'instance de l'exception est obtenue en instanciant la classe sans argument (au moment voulu).

Le *type* de l'exception est la classe de l'instance de l'exception, la *value* est l'instance elle-même.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so :

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining : if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed :

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used. The previous exception is then attached as the new exception's `__context__` attribute :

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause :

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

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

Modifié dans la version 3.3 : `None` est dorénavant autorisée en tant que `Y` dans `raise X from Y`.

Added the `__suppress_context__` attribute to suppress automatic display of the exception context.

Modifié dans la version 3.11 : si la trace d'appels de l'exception active est modifiée dans une clause *except*, une instruction `raise` postérieure lève à nouveau l'exception avec la trace modifiée. Auparavant, l'exception était levée à nouveau avec la trace qu'elle avait au moment de son interception.

7.9 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.

7.10 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 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.

7.11 L'instruction `import`

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

L'instruction de base *import* (sans clause *from*) est exécutée en deux étapes :

1. trouve un module, le charge et l'initialise si nécessaire
2. définit un ou des noms (*name* dans la grammaire ci-dessus) dans l'espace des noms locaux de la portée où l'instruction *import* apparaît.

Quand l'instruction contient plusieurs clauses (séparées par des virgules), les deux étapes sont menées séparément pour chaque clause, comme si les clauses étaient séparées dans des instructions d'importations individuelles.

Les détails de la première étape, de recherche et de chargement des modules, sont décrits largement dans la section relative au *système d'importation*, qui décrit également les différents types de paquets et modules qui peuvent être importés, de même que les points d'entrée pouvant être utilisés pour personnaliser le système d'importation. Notez que des erreurs dans cette étape peuvent indiquer soit que le module n'a pas été trouvé, soit qu'une erreur s'est produite lors de l'initialisation du module, ce qui comprend l'exécution du code du module.

Si le module requis est bien récupéré, il est mis à disposition de l'espace de nommage local suivant l'une des trois façons suivantes :

— Si le nom du module est suivi par `as`, alors le nom suivant `as` est directement lié au module importé.
— si aucun autre nom n'est spécifié et que le module en cours d'importation est un module de niveau le plus haut, le nom du module est lié dans l'espace des noms locaux au module importé ;
— si le module en cours d'importation n'est *pas* un module de plus haut niveau, alors le nom du paquet de plus haut niveau qui contient ce module est lié dans l'espace des noms locaux au paquet de plus haut niveau. Vous pouvez accéder au module importé en utilisant son nom pleinement qualifié et non directement.

La forme *from* utilise un processus un peu plus complexe :

1. trouve le module spécifié dans la clause *from*, le charge et l'initialise si nécessaire ;
2. pour chaque nom spécifié dans les clauses *import* :
   1. vérifie si le module importé possède un attribut avec ce nom ;
   2. si non, essaie d'importer un sous-module avec ce nom puis vérifie si le module importé possède lui-même cet attribut ;
   3. si l'attribut n'est pas trouvé, une `ImportError` est levée.

4. sinon, une référence à cette valeur est stockée dans l'espace des noms locaux, en utilisant le nom de la clause `as` si elle est présente, sinon en utilisant le nom de l'attribut.

Exemples :

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz imported, foo bound␣
↪locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz␣
↪bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz␣
↪bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

Si la liste des noms est remplacée par une étoile (`'*'`), tous les noms publics définis dans le module sont liés dans l'espace des noms locaux de la portée où apparaît l'instruction *import*.

Les *noms publics* définis par un module sont déterminés en cherchant dans l'espace de nommage du module une variable nommée `__all__`; Si elle est définie, elle doit être une séquence de chaînes désignant les noms définis ou importés par ce module. Les noms donnés dans `__all__` sont tous considérés publics et doivent exister. Si `__all__` n'est pas définie, l'ensemble des noms publics contient tous les noms trouvés dans l'espace des noms du module qui ne commencent pas par un caractère souligné (_). `__all__` doit contenir toute l'API publique. Elle est destinée à éviter l'exportation accidentelle d'éléments qui ne font pas partie de l'API (tels que des modules de bibliothèques qui ont été importés et utilisés à l'intérieur du module).

La forme d'*import* avec astérisque --- `from module import *` --- est autorisée seulement au niveau du module. Si vous essayez de l'utiliser dans une définition de classe ou de fonction, cela lève une `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 section *Importations relatives au paquet*.

`importlib.import_module()` est fournie pour gérer les applications qui déterminent dynamiquement les modules à charger.

Lève un évènement d'audit avec les arguments `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`.

7.11.1 L'instruction future

Une *instruction future* est une directive à l'attention du compilateur afin qu'un module particulier soit compilé en utilisant une syntaxe ou une sémantique qui sera disponible dans une future version de Python où cette fonctionnalité est devenue un standard.

L'instruction *future* a vocation à faciliter les migrations vers les futures versions de Python qui introduisent des changements incompatibles au langage. Cela permet l'utilisation de nouvelles fonctionnalités module par module avant qu'une version n'officialise cette fonctionnalité comme un standard.

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

La seule fonctionnalité qui nécessite l'utilisation de l'instruction `future` est `annotations` (voir la **PEP 563**).

Toutes les fonctionnalités (*feature* dans la grammaire ci-dessus) autorisées par l'instruction `future` sont toujours reconnues par Python 3. Cette liste comprend `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` et `with_statement`. Elles sont toutes redondantes car elles sont de toute manière activées ; elles ne sont conservées que par souci de compatibilité descendante.

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 où 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 calls to the built-in functions `exec()` and `compile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can 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__`.

7.12 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`.

Les noms listés dans l'instruction *`global`* ne doivent pas être définis en tant que paramètre formel, cible d'une

instruction *with* ou d'une clause *except*, ni dans la liste cible d'une boucle *for*, dans une définition de *class*, de fonction, d'instruction *import* ou une annotation de variable.

Particularité de l'implémentation CPython : L'implémentation actuelle ne vérifie pas toutes ces interdictions mais n'abusez pas de cette liberté car les implémentations futures pourraient faire la vérification ou modifier le comportement du programme sans vous avertir.

Note pour les programmeurs : *global* est une directive à l'attention de l'analyseur syntaxique. Elle s'applique uniquement au code analysé en même temps que l'instruction `global`. En particulier, une instruction `global` contenue dans une chaîne ou un objet code fourni à la fonction native `exec()` n'affecte pas le code *contenant* cet appel et le code contenu dans un telle chaîne n'est pas affecté par une instruction `global` placée dans le code contenant l'appel. Il en est de même pour les fonctions `eval()` et `compile()`.

7.13 L'instruction `nonlocal`

```
nonlocal_stmt   ::=   "nonlocal" identifier ("," identifier)*
```

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The *nonlocal* statement causes the listed identifiers to refer to names previously bound in nonlocal scopes. It allows encapsulated code to rebind such nonlocal identifiers. If a name is bound in more than one nonlocal scope, the nearest binding is used. If a name is not bound in any nonlocal scope, or if there is no nonlocal scope, a `SyntaxError` is raised.

The nonlocal statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

Voir aussi :

PEP 3104 -- Accès à des noms en dehors de la portée locale
Les spécifications pour l'instruction *nonlocal*.

Programmer's note : *nonlocal* is a directive to the parser and applies only to code parsed along with it. See the note for the *global* statement.

7.14 The `type` statement

```
type_stmt   ::=   'type' identifier [type_params] "=" expression
```

The `type` statement declares a type alias, which is an instance of `typing.TypeAliasType`.

For example, the following statement creates a type alias :

```
type Point = tuple[float, float]
```

This code is roughly equivalent to :

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` indicates an *annotation scope*, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's `__value__` attribute (see *Lazy evaluation*). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a *type parameter list* after the name. See *Generic type aliases* for more.

`type` is a *soft keyword*.

Added in version 3.12.

Voir aussi :

PEP 695 - Type Parameter Syntax
Introduced the `type` statement and syntax for generic classes and functions.

CHAPITRE 8

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.

Les instructions *if*, *while* et *for* implémentent les constructions classiques de contrôle de flux. *try* définit des gestionnaires d'exception et du code de nettoyage pour un groupe d'instructions, tandis que l'instruction *with* permet l'exécution de code d'initialisation et de finalisation autour d'un bloc de code. Les définitions de fonctions et de classes sont également, au sens syntaxique, des instructions composées.

Une instruction composée comporte une ou plusieurs « clauses ». Une clause se compose d'un en-tête et d'une « suite ». Les en-têtes des clauses d'une instruction composée particulière sont toutes placées au même niveau d'indentation. Chaque en-tête de clause commence par un mot-clé spécifique et se termine par le caractère deux-points (`:`) ; une suite est un groupe d'instructions contrôlées par une clause ; une suite se compose, après les deux points de l'en-tête, soit d'une ou plusieurs instructions simples séparées par des points-virgules si elles sont sur la même ligne que l'en-tête, soit d'une ou plusieurs instructions en retrait sur les lignes suivantes. Seule cette dernière forme d'une suite peut contenir des instructions composées ; ce qui suit n'est pas licite, principalement parce qu'il ne serait pas clair de savoir à quelle clause *if* se rapporterait une clause *else* placée en fin de ligne :

```
if test1: if test2: print(x)
```

Notez également que le point-virgule se lie plus étroitement que le deux-points dans ce contexte, de sorte que dans l'exemple suivant, soit tous les appels `print()` sont exécutés, soit aucun ne l'est :

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

En résumé :

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | match_stmt
                    | funcdef
                    | classdef
                    | async_with_stmt
                    | async_for_stmt
```

```
                    | async_funcdef
suite           ::=   stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=   stmt_list NEWLINE | compound_stmt
stmt_list       ::=   simple_stmt (";" simple_stmt)* [";"]
```

Notez que ces instructions se terminent toujours par un lexème `NEWLINE` suivi éventuellement d'un `DEDENT`. Notez également que les clauses facultatives qui suivent commencent toujours par un mot-clé qui ne peut pas commencer une instruction. Ainsi, il n'y a pas d'ambiguïté (le problème du *`else`* dont on ne sait pas à quel `if` il est relié est résolu en Python en exigeant que des instructions *`if`* imbriquées soient indentées les unes par rapport aux autres).

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

8.1 L'instruction `if`

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

```
if_stmt   ::=   "if" assignment_expression ":" suite
                ("elif" assignment_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.

8.2 L'instruction `while`

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

```
while_stmt   ::=   "while" assignment_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.

8.3 L'instruction `for`

L'instruction *`for`* est utilisée pour itérer sur les éléments d'une séquence (par exemple une chaîne, un *n*-uplet ou une liste) ou un autre objet itérable :

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

L'expression `starred_list` n'est évaluée qu'une seule fois ; elle doit produire un objet *iterable*. Un *iterator* est créé pour cet itérable. Le premier élément produit par l'itérateur est assigné à la liste cible (*target_list* dans la grammaire ci-dessus) en utilisant les règles des affectations (voir *Les assignations*), puis la « suite » est exécutée. Lorsque les

éléments de l'itérateur sont épuisés, 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 continue avec l'élément suivant, ou avec la clause `else` s'il n'y a pas d'élément suivant.

La boucle `for` effectue des affectations aux variables de la liste cible, ce qui écrase toutes les affectations antérieures de ces variables, y compris celles effectuées dans la suite de la boucle `for` :

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

Les noms dans la liste cible ne sont pas supprimés lorsque la boucle est terminée mais, si la séquence est vide, ils n'auront pas du tout été assignés par la boucle. Petite astuce : le type natif `range()` représente des suites arithmétiques immuables de nombres entiers ; par exemple, itérer sur `range(3)` renvoie successivement les entiers 0, 1 et 2.

Modifié dans la version 3.11 : les éléments étoilés sont maintenant autorisés dans l'expression liste.

8.4 L'instruction `try`

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

```
try_stmt    ::=  try1_stmt | try2_stmt | try3_stmt
try1_stmt   ::=  "try" ":" suite
                 ("except" [expression ["as" identifier]] ":" suite)+
                 ["else" ":" suite]
                 ["finally" ":" suite]
try2_stmt   ::=  "try" ":" suite
                 ("except" "*" expression ["as" identifier] ":" suite)+
                 ["else" ":" suite]
                 ["finally" ":" suite]
try3_stmt   ::=  "try" ":" suite
                 "finally" ":" suite
```

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.

8.4.1 clause `except`

La ou les clauses `except` précisent un ou plusieurs gestionnaires d'exceptions. Si aucune exception ne se produit dans la clause *try*, aucun gestionnaire d'exception n'est exécuté. Lorsqu'une exception se produit dans la suite de `try`, Python recherche un gestionnaire d'exception. Cette recherche inspecte les clauses `except`, l'une après l'autre, jusqu'à trouver une correspondance. Une clause `except` vide (c'est-à-dire sans expression), si elle est présente, doit être la dernière ; elle correspond à toute exception. Pour une clause `except` avec une expression, cette expression est évaluée et la clause correspond si l'objet résultant est « compatible » avec l'exception. Un objet est réputé compatible avec une exception s'il est la classe ou une classe mère (mais pas une *classe mère abstraite*) de l'objet exception ou si c'est un *n*-uplet dont un élément est la classe ou une classe mère (non-abstraite) de l'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).

Lorsqu'une clause `except` correspond, l'exception est affectée à la cible précisée après le mot-clé `as` dans cette clause `except`, si cette cible existe, et la suite de clause `except` est exécutée. Toutes les clauses `except` doivent avoir un bloc exécutable. Lorsque la fin de ce bloc est atteinte, l'exécution continue normalement après l'ensemble de l'instruction *`try`* (cela signifie que si deux gestionnaires imbriqués existent pour la même exception, et que l'exception se produit dans la clause `try` du gestionnaire interne, le gestionnaire externe ne gère pas l'exception).

Lorsqu'une exception a été affectée en utilisant `as cible`, elle est effacée à la fin de la clause `except`. C'est comme si :

```
except E as N:
    foo
```

avait été traduit en

```
except E as N:
    try:
        foo
    finally:
        del N
```

Cela veut dire que l'exception doit être assignée à un nom différent pour pouvoir s'y référer après la clause `except`. Les exceptions sont effacées parce qu'avec la trace de la pile d'appels qui leur est attachée, elles créent un cycle dans les pointeurs de références (avec le cadre de la pile), ce qui conduit à conserver tous les noms locaux de ce cadre en mémoire jusqu'au passage du ramasse-miettes.

Avant qu'une suite de clauses `except` ne soit exécutée, l'exception est stockée dans le module `sys`, où elle est accessible depuis le corps de la clause `except` en appelant `sys.exception()`. Lorsque vous quittez un gestionnaire d'exceptions, l'exception stockée dans le module `sys` est réinitialisée à sa valeur précédente :

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 clause except*

Les clauses except* sont utilisées pour gérer des ExceptionGroup. Le type de l'exception pour la correspondance est interprété de la même manière que dans le cas d'un *except* mais, dans le cas d'un groupe d'exceptions, il est possible d'avoir une correspondance partielle quand le type correspond à une ou plusieurs exceptions dans le groupe. Cela veut dire que plusieurs clauses except* peuvent être exécutées, chacune gérant une partie du groupe d'exceptions. Chaque clause ne s'exécute (au maximum) qu'une fois et gère un groupe d'exception constitué des exceptions qui correspondent. Chaque exception du groupe est gérée par une clause except* au plus, la première à laquelle elle correspond.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Any remaining exceptions that were not handled by any except* clause are re-raised at the end, along with all exceptions that were raised from within the except* clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

Si l'exception levée n'est pas un groupe d'exceptions et que son type correspond à l'une des clauses except*, elle est interceptée et encapsulée par un groupe d'exceptions avec une chaîne de message vide. :

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

Une clause except* doit avoir un type correspondant, et ce type ne peut pas être une sous-classe de BaseExceptionGroup. Il n'est pas possible de combiner *except* et except* dans un même *try*. Aucune clause *break*, *continue* ou *return* ne peut apparaître dans une clause except*.

8.4.3 clause `else`

La clause optionnelle `else` n'est exécutée que si l'exécution atteint la fin de la clause *try*, aucune exception n'a été levée, et aucun *return*, *continue*, ou *break* ont étés exécutés. Les exceptions dans la clause `else` ne sont pas gérées par les clauses *except* précédentes.

8.4.4 clause `finally`

Si `finally` est présente, elle définit un gestionnaire de « nettoyage ». La clause *try* est exécutée, y compris les clauses *except* et *else*. Si une exception se produit dans l'une des clauses et n'est pas traitée, l'exception est temporairement sauvegardée. La clause `finally` est exécutée. S'il y a une exception sauvegardée, elle est levée à nouveau à la fin de la clause `finally`. Si la clause `finally` lève une autre exception, l'exception sauvegardée est définie comme le contexte de la nouvelle exception. Si la clause `finally` exécute une instruction *return*, *break* ou *continue*, l'exception sauvegardée est jetée :

```
>>> 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 ».

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'
```

Modifié dans la version 3.8 : Avant Python 3.8, une instruction *continue* n'était pas licite dans une clause `finally` en raison d'un problème dans l'implémentation.

8.5 L'instruction `with`

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_stmt_contents ","? ")" | with_stmt_contents
with_stmt_contents  ::=  with_item (","  with_item)*
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. The context manager's *__enter__()* is loaded for later use.
3. The context manager's *__exit__()* is loaded for later use.
4. The context manager's *__enter__()* method is invoked.
5. If a target was included in the *with* statement, the return value from *__enter__()* is assigned to it.

   Note : The *with* statement guarantees that if the *__enter__()* method returns without an error, then *__exit__()* will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 7 below.

   ---

6. La suite est exécutée.
7. 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.

   If the suite was exited for any reason other than an exception, the return value from *__exit__()* is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

Le code suivant :

```
with EXPRESSION as TARGET:
    SUITE
```

est sémantiquement équivalent à :

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

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 sémantiquement équivalent à :

```
with A() as a:
    with B() as b:
        SUITE
```

Vous pouvez aussi écrire des gestionnaires de contexte sur plusieurs lignes pour plus d'un élément si ceux-ci sont placés entre parenthèses. Par exemple :

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

Modifié dans la version 3.1 : Prise en charge de multiples expressions de contexte.

Modifié dans la version 3.10 : prise en charge des parenthèses pour pouvoir écrire l'instruction sur plusieurs lignes.

Voir aussi :

PEP 343 — L'instruction « *with* »
La spécification, les motivations et des exemples de l'instruction `with` en Python.

8.6 L'instruction `match`

Added in version 3.10.

L'instruction *match* est utilisée pour le filtrage par motif. Sa syntaxe est :

```
match_stmt    ::=  'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr  ::=  star_named_expression "," star_named_expressions?
                   | named_expression
case_block    ::=  'case' patterns [guard] ":" block
```

Note : cette section utilise les guillemets simples pour désigner les *mots-clés ad-hoc*.

Le filtrage par motif prend un motif en entrée (`pattern` après `case`) et un champ de recherche (`subject_expr` après `match`). Le motif du filtre (qui peut contenir des sous-motifs de filtrage) est confronté au contenu du champ de recherche. La sortie est composée de :

— un indicateur de réussite ou d'échec pour le filtrage (on peut aussi dire que le motif a réussi ou échoué) ;
— la possibilité de lier les valeurs filtrées à un nom. Les pré-requis sont indiqués plus bas.

Les mots-clés `match` et `case` sont des *mots-clés ad-hoc*.

Voir aussi :

— **PEP 634** — Spécifications pour le filtrage par motif
— **PEP 636** — Tutoriel pour le filtrage par motif

8.6.1 Aperçu

Voici un aperçu du déroulement logique d'un filtrage par motif :

1. L'expression confrontée aux filtres, `subject_expr`, est évaluée pour obtenir la valeur résultante. Si l'expression contient une virgule, un *n*-uplet est construit en utilisant les règles classiques.
2. Chaque filtre des blocs `case_block` est confronté à la valeur résultante du champ de recherche. Les règles particulières pour la réussite ou l'échec sont décrites plus bas. La confrontation du filtre peut aussi conduire à lier un ou plusieurs noms présents dans le motif. Les règles pour lier les noms des motifs dépendent du type de filtre et sont décrites plus bas. **Le nommage effectué lors d'un filtrage par motif qui a réussi persiste à l'extérieur du bloc et le nom peut être utilisé après l'instruction** `match`.

   Note : en cas d'échec de la recherche, certains sous-filtres peuvent avoir réussi. Ne vous fiez pas aux nommages faits lors d'un filtrage qui a échoué. Inversement, ne vous fiez pas aux variables qui restent inchangées après

un filtrage infructueux. Le comportement exact dépend de l'implémentation et peut varier. Il s'agit d'un choix intentionnel afin de permettre aux implémentations d'ajouter des optimisations.

3. Si la recherche réussit, la garde correspondante (si elle existe) est évaluée. Dans ce cas, on est sûr que les nommages ont bien eu lieu.
   — Si la garde s'évalue à vrai ou s'il n'y a pas de garde, le `block` à l'intérieur du `case_block` est exécuté.
   — Sinon, le `case_block` est testé comme décrit ci-dessus.
   — S'il n'y a plus de bloc `case_block`, l'instruction est terminée.

Note : l'utilisateur ne doit jamais faire confiance à un filtre en cours d'évaluation. En fonction de l'implémentation, l'interpréteur peut mettre des valeurs en cache ou utiliser des optimisations qui évitent des réévaluations.

Voici un exemple d'instruction de filtrage par motif :

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

Dans cet exemple, `if flag` est une garde. Plus de détails sont fournis dans la prochaine section.

8.6.2 Gardes

```
guard   ::=   "if" named_expression
```

Une garde (`guard` qui fait partie du `case`) doit s'évaluer à vrai pour que le code à l'intérieur du bloc `case` soit exécuté. Elle s'écrit sous la forme du mot-clé *if* suivi d'une expression.

Le déroulement logique d'un bloc `case` qui comprend une garde est le suivant :

1. Vérification que le filtrage dans le bloc `case` est fructueux. Si le filtrage échoue, la garde n'est pas évaluée et on passe au bloc `case` suivant.
2. Si le filtrage est fructueux, évaluation de la garde.
   — Si la garde s'évalue à *vrai*, le bloc est sélectionné.
   — Si la garde s'évalue à *faux*, le bloc n'est pas sélectionné.
   — Si une exception est levée lors de l'évaluation de la garde, cette exception est propagée.

Les gardes étant des expressions, il est possible qu'elles aient des effets secondaires. L'ordre d'évaluation des gardes est du premier au dernier bloc `case`, un à la fois, en sautant les blocs `case` dont la recherche de motif échouent. L'évaluation des gardes s'arrête dès qu'un bloc `case` est sélectionné.

8.6.3 Bloc `case` attrape-tout

Un bloc `case` attrape-tout est un bloc qui réussit toujours. Une instruction `match` ne peut avoir qu'un seul bloc attrape-tout, et ce doit être le dernier.

Un bloc `case` est considéré attrape-tout s'il n'y a pas de garde et que le motif est attrape-tout. Un motif est attrape-tout si l'on peut déterminer, simplement à partir de sa syntaxe, qu'il correspond toujours. Seuls les motifs suivants sont attrape-tout :

- Les *Filtres AS* pour lesquels la partie gauche est attrape-tout
- Les *Filtres OU* contenant au moins un filtre attrape-tout
- Les *Filtres de capture*
- Les *Filtres attrape-tout*
- les filtres attrape-tout entre parenthèses

8.6.4 Filtres

Note : Cette section utilise des notations grammaticales qui ne font pas partie du standard EBNF :

- la notation `SEP.REGLE+` désigne `REGLE (SEP REGLE)*`
- la notation `!REGLE` désigne la négation logique de l'assertion `REGLE`

La syntaxe générale pour les filtres `patterns` est :

```
patterns         ::=  open_sequence_pattern | pattern
pattern          ::=  as_pattern | or_pattern
closed_pattern   ::=  | literal_pattern
                      | capture_pattern
                      | wildcard_pattern
                      | value_pattern
                      | group_pattern
                      | sequence_pattern
                      | mapping_pattern
                      | class_pattern
```

Les explications ci-dessous décrivent « en termes simples » ce qu'un modèle fait (merci à Raymond Hettinger pour son document qui a inspiré la plupart des descriptions). Notez que ces descriptions sont purement à fin d'illustration et peuvent ne **pas** être strictement conformes à l'implémentation sous-jacente. De plus, nous ne couvrons pas toutes les formes valides.

Filtres OU

Un filtre OU est composé de deux filtres ou plus séparés par des barres verticales `|`. La syntaxe est :

```
or_pattern   ::=   "|".closed_pattern+
```

Seul le dernier sous-filtre peut être *attrape-tout* et chaque sous-filtre doit être lié au même ensemble de noms pour éviter toute ambigüité.

Un filtre OU confronte chacun des sous-filtres à tour de rôle à la valeur du champ de recherche, jusqu'à ce que l'un d'eux réussisse. Le filtre OU réussit si l'un des sous-filtres a réussi, sinon il échoue.

En termes plus simples, `M1 | M2 | ...` teste le filtre par motif `M1`, s'il échoue il teste le filtre par motif `M2`, réussit immédiatement si l'un d'eux réussit, échoue dans le cas contraire.

Filtres AS

Un filtre AS confronte un filtre OU sur la gauche du mot-clé *as* au champ de recherche. La syntaxe est la suivante :

```
as_pattern   ::=   or_pattern "as" capture_pattern
```

Si le filtre OU échoue, le filtre AS échoue. Sinon, le filtre AS lie le champ de recherche au nom sur la droite du mot-clé `as` et réussit. `capture_pattern` ne peut pas être un _.

En termes simples, `M as NOM` filtre avec le motif `M` et, s'il réussit, définit `NOM = <subject>`.

Filtres littéraux

Un filtre littéral effectue une correspondance avec la plupart des *littéraux* en Python. La syntaxe est la suivante :

```
literal_pattern   ::=   signed_number
                      | signed_number "+" NUMBER
                      | signed_number "-" NUMBER
                      | strings
                      | "None"
                      | "True"
                      | "False"
                      | signed_number: NUMBER | "-" NUMBER
```

La règle `strings` et le lexème `NUMBER` sont définis dans *la grammaire de Python standard*. Les chaînes avec triples guillemets sont gérées. Les chaînes brutes et les chaînes d'octets sont gérées. Les *f-strings* ne sont pas gérées.

Les formes `signed_number '+' NUMBER` et `signed_number '-' NUMBER` permettent d'exprimer des *nombres complexes* ; vous devez indiquer un nombre réel sur la gauche et un nombre imaginaire sur la droite. Par exemple, `3 + 4j`.

En termes simples, `LITERAL` réussit seulement si `<subject> == LITERAL`. Pour les singletons `None`, `True` et `False`, l'opérateur *is* est utilisé.

Filtres de capture

Un filtre de capture lie la valeur du champ de recherche à un nom. La syntaxe est la suivante :

```
capture_pattern   ::=   !'_' NAME
```

Un simple caractère souligné _ n'est pas un filtre de capture (c'est ce que `!'_'` veut dire). C'est le motif pour désigner un filtre attrape-tout (lexème `wilcard_pattern`, voir plus bas).

Dans un filtre donné, un nom ne peut être lié qu'une seule fois. Par exemple, `case x, x: ...` est invalide mais `case [x] | x: ...` est autorisé.

Les filtres de capture réussissent toujours. La portée du lien est conforme aux règles définies pour l'opérateur d'affectation indiquées dans la **PEP 572** ; le nom devient une variable locale dans la fonction la plus intérieure à moins qu'il n'y ait une instruction *global* ou *nonlocal* qui s'applique.

En termes simples, `NAME` réussit toujours et définit `NAME = <subject>`.

Filtres attrape-tout

Un filtre attrape-tout réussit toujours (quel que soit le champ de recherche) et ne lie aucun nom. La syntaxe est la suivante :

```
wildcard_pattern   ::=   '_'
```

_ est un *mot-clé ad-hoc* dans un filtre par motif, mais seulement dans un filtre. Ailleurs, c'est un identifiant, comme d'habitude, même à l'intérieur d'une expression champ de recherche de `match`, d'une garde ou d'un bloc `case`.

En termes simples, _ réussit toujours.

Filtres par valeurs

Un filtre par valeur représente une valeur nommée de Python. Sa syntaxe est la suivante :

```
value_pattern   ::=   attr
attr            ::=   name_or_attr "." NAME
name_or_attr    ::=   attr | NAME
```

Le nom qualifié dans le filtre est recherché en utilisant la *méthode de résolution des noms* standard de Python. Le filtrage réussit si la valeur trouvée vérifie l'égalité avec la valeur du champ de recherche (en utilisant l'opérateur d'égalité ==).

En termes plus simples, `NOM1.NOM2` réussit seulement si `<subject> == NOM1.NOM2`

Note : si la même valeur apparaît plusieurs fois dans la même instruction `match`, l'interpréteur peut mettre en cache la première valeur trouvée et la réutiliser plutôt que de refaire une recherche. Ce cache est strictement limité à l'exécution de l'instruction `match` donnée.

Filtres de groupes

Un filtre de groupe permet au programmeur de souligner l'intention de regrouper des motifs en plaçant ceux-ci entre parenthèses. À part ça, il n'introduit aucune syntaxe supplémentaire. Sa syntaxe est la suivante :

```
group_pattern   ::=   "(" pattern ")"
```

En termes plus simples, `(P)` équivaut à `P`.

Filtres de séquences

Un filtre de séquence contient des sous-filtres par motif dont chacun doit correspondre à un élément d'une séquence. La syntaxe est similaire au déballage d'une liste ou d'un *n*-uplet.

```
sequence_pattern         ::=   "[" [maybe_sequence_pattern] "]"
                               | "(" [open_sequence_pattern] ")"
open_sequence_pattern    ::=   maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern   ::=   ",".maybe_star_pattern+ ","?
maybe_star_pattern       ::=   star_pattern | pattern
star_pattern             ::=   "*" (capture_pattern | wildcard_pattern)
```

Vous pouvez utiliser indifféremment des parenthèses `(...)` ou des crochets `[...]` pour encadrer les filtres à regrouper.

Note : un filtre seul entre parenthèses qui ne se termine pas par une virgule (par exemple `(3 | 4)`) est un *filtre de groupe*. En revanche, un filtre seul entre crochets (par exemple `[3 | 4]`) reste un filtre de séquence.

Il peut y avoir au plus un sous-filtre étoilé (lexème `star_pattern`) dans un filtre de séquence. Le filtre étoilé peut se trouver à n'importe quelle position. S'il n'y en a pas, le filtre de séquence est un filtre de séquence à longueur fixe, sinon c'est un filtre de séquence à longueur variable.

Voici le déroulement logique d'un filtrage par motif de séquence sur une valeur du champ de recherche :

1. Si la valeur du champ de recherche n'est pas une séquence [2], le filtre de séquence échoue.
2. Si la valeur du champ de recherche est une instance de `str`, `bytes` ou `bytearray`, le filtre de séquence échoue.
3. Les étapes suivantes dépendent de la longueur fixe ou non du filtre de séquence.

   Si le filtre de séquence est de longueur fixe :

   1. Si la longueur de la séquence champ de recherche n'est pas égale au nombre de sous-filtres, le filtre de séquence échoue.
   2. Les sous-filtres de la séquence sont confrontés aux éléments correspondants dans la séquence champ de recherche, de la gauche vers la droite. La recherche de correspondance s'arrête dès qu'un sous-filtre échoue. Si tous les sous-filtres réussissent la confrontation à l'élément du champ de recherche correspondant, le filtre de séquence réussit.

   Sinon, si le filtre de séquence est de longueur variable :

   1. Si la longueur de la séquence champ de recherche est plus petite que le nombre de sous-filtres sans étoile, le filtre de séquence échoue.
   2. Les sous-filtres sans étoile du début sont confrontés aux éléments correspondants comme pour un filtre de séquences de longueur fixe.
   3. Si les étapes précédentes ont réussi, le sous-filtre étoilé correspond à une liste formée des éléments restants du champ de recherche, en excluant les éléments restants qui correspondent à des sous-filtres sans étoile qui suivent le sous-filtre étoilé.
   4. Les sous-filtres sans étoile qui restent sont confrontés aux éléments restants du champ de recherche, comme pour un filtre de séquences de longueur fixe.

   Note : la longueur de la séquence champ de recherche est obtenue par `len()` (c.-à-d. avec le protocole `__len__()`). Cette longueur peut être mise en cache par l'interpréteur de la même manière que pour les *filtres par valeur*.

En termes plus simples, `[M1, M2, M3, ..., M<N>]` réussit seulement si tout ce qui suit a lieu :

— vérification que `<subject>` est une séquence,
— `len(subject) == <N>`,
— `M1` correspond à `<subject>[0]` (notez que cette correspondance peut lier des noms),
— `M2` correspond à `<subject>[1]` (notez que cette correspondance peut lier des noms),
— et ainsi de suite pour chaque filtre par motif / élément.

2. Dans le filtrage par motif, une séquence est définie comme suit :
— une classe qui hérite de `collections.abc.Sequence`
— une classe Python qui a été enregistrée en tant que `collections.abc.Sequence`
— une classe native dont le bit (CPython) `Py_TPFLAGS_SEQUENCE` est à 1
— une classe qui hérite d'une classe citée ci-dessus
Les classes suivantes de la bibliothèque standard sont des séquences :
— `array.array`
— `collections.deque`
— `list`
— `memoryview`
— `range`
— `tuple`

Note : Les champs de recherche du type `str`, `bytes` et `bytearray` ne correspondent pas avec des filtres de séquence.

Filtres associatifs

Un filtre associatif contient un ou plusieurs motifs clé-valeur. La syntaxe est similaire à la construction d'un dictionnaire :

```
mapping_pattern      ::=  "{" [items_pattern] "}"
items_pattern        ::=  ",".key_value_pattern+ ","?
key_value_pattern    ::=  (literal_pattern | value_pattern) ":" pattern
                        | double_star_pattern
double_star_pattern  ::=  "**" capture_pattern
```

Un seul sous-filtre doublement étoilé peut être présent dans le filtre associatif. Le filtre doublement étoilé doit être le dernier sous-filtre du filtre associatif.

Il est interdit d'avoir des clés en double dans les filtres associatifs. Une clé en double sous forme littérale lève une `Syntax Error`. Deux clés qui ont la même valeur lèvent une `ValueError` à l'exécution.

Voici le déroulement d'un filtrage associatif sur la valeur du champ de recherche :

1. Si la valeur du champ de recherche n'est pas un tableau associatif [3], le filtre associatif échoue.
2. Si chaque clé donnée dans le filtre associatif est présente dans le tableau associatif du champ de recherche, et que le filtre pour chaque clé correspond aux éléments du tableau associatif champ de recherche, le filtre associatif réussit.
3. Si des clés identiques sont détectées dans le filtre par motif, le filtre est déclaré invalide. Une `SyntaxError` est levée pour les valeurs littérales dupliquées ou une `ValueError` pour des clés s'évaluant à la même valeur.

Note : Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or *`__getitem__()`*.

En termes simples, `{CLÉ1: M1, CLÉ2: M2, ... }` réussit seulement si tout ce qui suit a lieu :
— vérification que `<subject>` est un tableau associatif,
— `CLÉ1 in <subject>`,
— `M1` correspond à `<subject>[CLÉ1]`,
— et ainsi de suite pour chaque paire CLÉ/Motif.

Filtres de classes

Un filtre de classe représente une classe et ses arguments positionnels et par mots-clés (s'il y en a). La syntaxe est la suivante :

```
class_pattern         ::=  name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments     ::=  positional_patterns ["," keyword_patterns]
                         | keyword_patterns
positional_patterns   ::=  ",".pattern+
keyword_patterns      ::=  ",".keyword_pattern+
keyword_pattern       ::=  NAME "=" pattern
```

Le même mot-clé ne doit pas être répété dans les filtres de classes.

Voici le déroulement d'un filtrage de classe sur la valeur du champ de recherche :

3. Dans le filtrage par motif, un tableau associatif est défini comme suit :
— une classe qui hérite de `collections.abc.Mapping`
— une classe Python qui a été enregistrée en tant que `collections.abc.Mapping`
— une classe native dont le bit (CPython) `Py_TPFLAGS_MAPPING` est à 1
— une classe qui hérite d'une classe citée ci-dessus
Les classes `dict` et `types.MappingProxyType` de la bibliothèque standard sont des tableaux associatifs.

1. Si `name_or_attr` n'est pas une instance de la classe native `type`, lève une `TypeError`.
2. Si la valeur du champ de recherche n'est pas une instance de `name_or_attr` (testé *via* `isinstance()`), le filtre de classe échoue.
3. S'il n'y a pas d'argument au filtre, le filtre réussit. Sinon, les étapes suivantes dépendent de la présence ou non de motifs pour les arguments positionnels ou par mot-clé.

   Pour un certain nombre de types natifs (indiqués ci-dessous), un motif positionnel seul est accepté, qui est confronté au champ de recherche en entier ; pour ces types, les motifs par mots-clés fonctionnent comme les autres types.

   S'il n'y a que des motifs par mot-clé (NdT : dans le sens « argument par mot-clé »), ils sont évalués comme ceci, un par un :

   I. Le mot-clé est recherché en tant qu'attribut du champ de recherche.
   — Si cela lève une exception autre que `AttributeError`, l'exception est propagée vers le haut.
   — Si cela lève l'exception `AttributeError`, le filtre échoue.
   — Sinon, le motif associé au mot-clé est confronté à la valeur de l'attribut du champ de recherche. Si cela échoue, le filtre de classe échoue ; si cela réussit, le filtre passe au mot-clé suivant.

   II. Si tous les motifs par mot-clé ont réussi, le filtre de classe réussit.

   Si des motifs positionnels sont présents, ils sont convertis en motifs par mot-clé en utilisant l'attribut `__match_args__` de la classe `name_or_attr` avant le filtrage :

   I. L'équivalent de `getattr(cls, "__match_args__", ())` est appelé.
   — Si cela lève une exception, elle est propagée vers le haut.
   — Si la valeur de retour n'est pas un *n*-uplet, la conversion échoue et une `TypeError` est levée.
   — S'il y a plus de motifs positionnels que `len(cls.__match_args__)`, une `TypeError` est levée.
   — Sinon, le motif positionnel `i` est converti en motif par mot-clé (le mot-clé sera `__match_args__[i]`). `__match_args__[i]` doit être une chaîne, sinon une `TypeError` est levée.
   — Si un mot-clé est dupliqué, une `TypeError` est levée.

   Voir aussi :

   Arguments positionnels dans le filtrage par motif sur les classes

   II. Une fois que tous les motifs positionnels ont été convertis en motifs par mot-clé,
   le filtre se déroule comme si tous les motifs étaient des motifs par mots-clés.

   Pour les types natifs suivants, le traitement des motifs positionnels est différent :
   — `bool`
   — `bytearray`
   — `bytes`
   — `dict`
   — `float`
   — `frozenset`
   — `int`
   — `list`
   — `set`
   — `str`
   — `tuple`

   Ces classes acceptent un argument positionnel seul et le filtre s'applique alors sur l'ensemble de l'objet plutôt que sur un simple attribut. Par exemple, `int(0|1)` réussit lorsqu'il est confronté à la valeur `0`, mais pas lorsque c'est la valeur `0.0`.

En termes simples, `CLS(P1, attr=P2)` réussit seulement si la séquence suivante est déroulée :
— `isinstance(<subject>, CLS)`
— convertit `P1` vers un motif par mot-clé en utilisant `CLS.__match_args__`
— Pour chaque argument par mot-clé `attr=P2` :
  — `hasattr(<subject>, "attr")`
  — `P2` correspond à `<subject>.attr`
— … et ainsi de suite pour les paires motif/argument par mot-clé.

Voir aussi :

— **PEP 634** — Spécifications pour le filtrage par motif
— **PEP 636** — Tutoriel pour le filtrage par motif

8.7 Définition de fonctions

Une définition de fonction définit un objet fonction défini par l'utilisateur (voir la section *Hiérarchie des types standards*) :

```
funcdef                 ::=  [decorators] "def" funcname [type_params] "(" [paramete
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" assignment_expression NEWLINE
parameter_list          ::=  defparameter ("," defparameter)* "," "/" ["," [paramet
                             | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_list_
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" paramet
                             | "**" parameter [","]
parameter               ::=  identifier [":" expression]
defparameter            ::=  parameter ["=" expression]
funcname                ::=  identifier
```

Une définition de fonction est une instruction qui est exécutée. Son exécution lie le nom de la fonction, dans l'espace de nommage 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. [4]

Une définition de fonction peut être encapsulée dans une ou plusieurs expressions *decorator* ; les décorateurs sont évalués lorsque la fonction est définie, dans la portée qui contient la définition de fonction ; le résultat doit être un appelable, qui est invoqué avec l'objet fonction comme seul argument ; la valeur renvoyée est liée au nom de la fonction en lieu et place de l'objet fonction. Lorsqu'il y a plusieurs décorateurs, ils sont appliqués par imbrication ; par exemple, le code suivant :

```
@f1(arg)
@f2
def func(): pass
```

est à peu près équivalent à :

```
def func(): pass
func = f1(arg)(f2(func))
```

sauf que la fonction originale n'est pas temporairement liée au nom `func`.

Modifié dans la version 3.9 : les fonctions peuvent être décorées par toute `expression d'affectation` valide. Auparavant, la grammaire était beaucoup plus restrictive ; voir la **PEP 614** pour obtenir les détails.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's `__type_params__` attribute. See *Generic functions* for more.

Modifié dans la version 3.12 : Type parameter lists are new in Python 3.12.

Lorsqu'un ou plusieurs *paramètres* sont de la forme *parameter* = *expression*, on dit que la fonction a des « valeurs de paramètres par défaut ». Pour un paramètre avec une valeur par défaut, l'*argument* correspondant peut être omis lors de l'appel, la valeur par défaut du paramètre est alors utilisée. Si un paramètre a une valeur par défaut, tous les paramètres suivants jusqu'à "`*`" doivent aussi avoir une valeur par défaut — ceci est une restriction syntaxique qui n'est pas exprimée dans la grammaire.

4. A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's *docstring*.

Les valeurs par défaut des paramètres sont évaluées de la gauche vers la droite quand la définition de la fonction est exécutée. Cela signifie que l'expression est évaluée une fois, lorsque la fonction est définie, et que c'est la même valeur « pré-calculée » qui est utilisée à chaque appel. C'est particulièrement important à comprendre lorsque la valeur d'un paramètre par défaut est un objet mutable (cas d'une liste ou un dictionnaire par exemple) : si la fonction modifie l'objet (par exemple en ajoutant un élément à une liste), la valeur par défaut est modifiée. En général, ce n'est pas l'effet voulu. Une façon d'éviter cet écueil est d'utiliser `None` par défaut et de tester explicitement la valeur dans le corps de la fonction. Par exemple :

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

La sémantique de l'appel de fonction est décrite plus en détail dans la section *Appels*. Un appel de fonction assigne toujours des valeurs à tous les paramètres mentionnés dans la liste des paramètres, soit à partir d'arguments positionnels, d'arguments par mots-clés ou de valeurs par défaut. S'il y a un paramètre de la forme `*identifier`, il est initialisé à un *n*-uplet recevant les paramètres positionnels en surplus, la valeur par défaut étant le *n*-uplet vide. S'il y a un paramètre de la forme `**identifier`, il est initialisé à un nouveau tableau associatif ordonné qui récupère tous les arguments par mot-clé en surplus, la valeur par défaut étant un tableau associatif vide du même type. Les paramètres après `*` ou `*identifier` sont forcément des paramètres par mot-clé et ne peuvent être passés qu'en utilisant des arguments par mot-clé. Au contraire, ceux avant `/` ne peuvent être passés qu'avec des arguments positionnels.

Modifié dans la version 3.8 : ajout de la syntaxe avec `/` pour indiquer les paramètre exclusivement positionnels (voir la **PEP 570**).

Les paramètres peuvent avoir une *annotation* sous la forme "`: expression`" après le nom du paramètre. Tout paramètre peut avoir une annotation, même ceux de la forme `*identifier` ou `**identifier`. Les fonctions peuvent avoir une annotation pour la valeur de retour, sous la forme "`-> expression`" après la liste des paramètres. Ces annotations peuvent prendre la forme de toute expression Python valide. Leur présence ne change pas la sémantique de la fonction. Les valeurs des annotations sont accessibles comme valeurs d'un dictionnaire dont les clés sont les noms des paramètres et défini comme attribut `__annotations__` de l'objet fonction. Si `annotations` est importé de `__future__`, les annotations sont conservées sous la forme de chaînes de caractères, permettant leur évaluation différée. Autrement, elles sont interprétées en même temps que la déclaration des fonctions. Dans le premier cas, les annotations peuvent être interprétées dans un ordre différent de l'ordre dans lequel elles apparaissent dans le fichier.

Il est aussi possible de créer des fonctions anonymes (fonctions non liées à un nom), pour une utilisation immédiate dans des expressions. Utilisez alors des expressions lambda, décrites dans la section *Expressions lambda*. Notez qu'une expression lambda est simplement un raccourci pour définir une fonction simple ; une fonction définie par une instruction "*`def`*" peut être passée (en argument) ou assignée à un autre nom, tout comme une fonction définie par une expression lambda. La forme "`def`" est en fait plus puissante puisqu'elle permet l'exécution de plusieurs instructions et les annotations.

Note pour les programmeurs : les fonctions sont des objets de première classe. Une instruction "`def`" exécutée à l'intérieur d'une définition de fonction définit une fonction locale qui peut être renvoyée ou passée en tant qu'argument. Les variables libres utilisées dans la fonction imbriquée ont accès aux variables locales de la fonction contenant le "`def`". Voir la section *Noms et liaisons* pour plus de détails.

Voir aussi :

PEP 3107 — Annotations de fonctions
La spécification originale pour les annotations de fonctions.

PEP 484 — Indications de types
Définition de la signification standard pour les annotations : indications de types.

PEP 526 — Syntaxe pour les annotations de variables
Ability to type hint variable declarations, including class variables and instance variables.

PEP 563 — Évaluation différée des annotations
Gestion des références postérieures à l'intérieur des annotations en préservant les annotations sous forme de chaînes à l'exécution au lieu d'une évaluation directe.

PEP 318 - Decorators for Functions and Methods
Function and method decorators were introduced. Class decorators were introduced in **PEP 3129**.

8.8 Définition de classes

Une définition de classe définit un objet classe (voir la section *Hiérarchie des types standards*) :

```
classdef      ::=  [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance   ::=  "(" [argument_list] ")"
classname     ::=  identifier
```

Une définition de classe est une instruction qui est exécutée. La liste d'héritage (*inheritance* entre crochets dans la grammaire ci-dessus) donne habituellement une liste de classes mères (voir *Métaclasses* pour des utilisations plus avancées). Donc chaque élément de la liste doit pouvoir être évalué comme un objet classe qui autorise les sous-classes. Les classes sans liste d'héritage héritent, par défaut, de la classe mère `object` ; d'où :

```
class Foo:
    pass
```

est équivalente à :

```
class Foo(object):
    pass
```

La suite de la classe est ensuite exécutée dans un nouveau cadre d'exécution (voir *Noms et liaisons*), en utilisant un espace de nommage local nouvellement créé et l'espace de nommage global d'origine (habituellement, la suite contient principalement des définitions de fonctions). Lorsque la suite de la classe termine son exécution, son cadre d'exécution est abandonné mais son espace des noms locaux est sauvegardé [5]. Un objet classe est alors créé en utilisant la liste d'héritage pour les classes mères et l'espace de nommage sauvegardé comme dictionnaire des attributs. Le nom de classe est lié à l'objet classe dans l'espace de nommage local original.

L'ordre dans lequel les attributs sont définis dans le corps de la classe est préservé dans le `__dict__` de la nouvelle classe. Notez que ceci n'est fiable que juste après la création de la classe et seulement pour les classes qui ont été définies en utilisant la syntaxe de définition.

La création de classes peut être fortement personnalisée en utilisant les *métaclasses*.

Les classes peuvent aussi être décorées. Comme pour les décorateurs de fonctions :

```
@f1(arg)
@f2
class Foo: pass
```

est à peu près équivalent à :

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

Les règles d'évaluation pour les expressions de décorateurs sont les mêmes que pour les décorateurs de fonctions. Le résultat est alors lié au nom de la classe.

Modifié dans la version 3.9 : les classes peuvent être décorées par toute `expression d'affectation` valide. Auparavant, la grammaire était beaucoup plus restrictive ; voir la **PEP 614** pour obtenir les détails.

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's `__type_params__` attribute. See *Generic classes* for more.

5. Une chaîne littérale apparaissant comme première instruction dans le corps de la classe est transformée en élément `__doc__` de l'espace de nommage et donc en *docstring* de la classe.

Modifié dans la version 3.12 : Type parameter lists are new in Python 3.12.

Note pour les programmeurs : les variables définies dans la définition de classe sont des attributs de classe ; elles sont partagées par les instances. Les attributs d'instance peuvent être définis dans une méthode en utilisant `self.name = value`. Les attributs de classe et d'instance sont accessibles par la notation "`self.name`", et un attribut d'instance masque un attribut de classe de même nom lorsqu'on y accède de cette façon. Les attributs de classe peuvent être utilisés comme valeurs par défaut pour les attributs d'instances, mais l'utilisation de valeurs mutables peut conduire à des résultats inattendus. Les *descripteurs* peuvent être utilisés pour créer des variables d'instances avec des détails d'implémentation différents.

Voir aussi :

PEP 3115 — Métaclasses dans Python 3000
La proposition qui a modifié la déclaration de métaclasses à la syntaxe actuelle, et la sémantique pour la façon dont les classes avec métaclasses sont construites.

PEP 3129 — Décorateurs de classes
La proposition qui a ajouté des décorateurs de classe. Les décorateurs de fonction et de méthode ont été introduits dans **PEP 318**.

8.9 Coroutines

Added in version 3.5.

8.9.1 Définition de fonctions coroutines

```
async_funcdef   ::=   [decorators] "async" "def" funcname "(" [parameter_list] ")"
                      ["->" expression] ":" suite
```

L'exécution de coroutines Python peut être suspendue et reprise à plusieurs endroits (voir *coroutine*). Les expressions `await`, `async for` et `async with` ne peuvent être utilisées que dans les corps de coroutines.

Les fonctions définies avec la syntaxe `async def` sont toujours des fonctions coroutines, même si elles ne contiennent aucun mot-clé `await` ou `async`.

C'est une `SyntaxError` d'utiliser une expression `yield from` dans une coroutine.

Un exemple de fonction coroutine :

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

Modifié dans la version 3.7 : `await` et `async` sont dorénavant des mots-clés ; auparavant, ils n'étaient traités comme tels que dans le corps d'une fonction coroutine.

8.9.2 L'instruction `async for`

```
async_for_stmt   ::=   "async" for_stmt
```

Un *itérable asynchrone* fournit une méthode `__aiter__` qui renvoie directement un *itérateur asynchrone*, celui-ci pouvant appeler du code asynchrone dans sa méthode `__anext__`.

L'instruction `async for` permet d'itérer facilement sur des itérables asynchrones.

Le code suivant :

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

est sémantiquement équivalent à :

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

Voir aussi *__aiter__()* et *__anext__()* pour plus de détails.

C'est une `SyntaxError` d'utiliser une instruction `async for` en dehors d'une fonction coroutine.

8.9.3 L'instruction `async with`

```
async_with_stmt   ::=   "async" with_stmt
```

Un *gestionnaire de contexte asynchrone* est un *gestionnaire de contexte* qui est capable de suspendre l'exécution dans ses méthodes *enter* et *exit*.

Le code suivant :

```
async with EXPRESSION as TARGET:
    SUITE
```

est sémantiquement équivalent à :

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

Voir aussi *__aenter__()* et *__aexit__()* pour plus de détails.

C'est une `SyntaxError` d'utiliser l'instruction `async with` en dehors d'une fonction coroutine.

Voir aussi :

PEP 492 — Coroutines avec les syntaxes *async* et *await*
La proposition qui a fait que les coroutines soient un concept propre en Python, et a ajouté la syntaxe de prise en charge de celles-ci.

8.10 Type parameter lists

Added in version 3.12.

```
type_params    ::=   "[" type_param ("," type_param)* "]"
type_param     ::=   typevar | typevartuple | paramspec
typevar        ::=   identifier (":" expression)?
typevartuple   ::=   "*" identifier
paramspec      ::=   "**" identifier
```

Functions (including *coroutines*), *classes* and *type aliases* may contain a type parameter list :

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an *annotation scope*) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

Type parameters come in three kinds :

- `typing.TypeVar`, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.
- `typing.TypeVarTuple`, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.
- `typing.ParamSpec`, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the `typing.TypeVar` can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For `typing.TypeVar`s declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__`

and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

`typing.TypeVarTuple`s and `typing.ParamSpec`s cannot have bounds or constraints.

The following example indicates the full set of allowed type parameter declarations :

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple,
   **SimpleParamSpec,
](
   a: SimpleTypeVar,
   b: TypeVarWithBound,
   c: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *d: SimpleTypeVarTuple,
): ...
```

8.10.1 Generic functions

Generic functions are declared as follows :

```
def func[T](arg: T): ...
```

This syntax is equivalent to :

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation : the syntax does not go through attribute access on the `typing` module, but creates an instance of `typing.TypeVar` directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters :

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the *lazy evaluation* of the `TypeVar` bound, this is equivalent to :

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

8.10.2 Generic classes

Generic classes are declared as follows :

```
class Bag[T]: ...
```

This syntax is equivalent to :

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an *annotation scope*, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from `typing.Generic`. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example :

```
@decorator
class Bag(Base[T], arg=T): ...
```

This is equivalent to :

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Generic type aliases

The *type* statement can also be used to create a generic type alias :

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to :

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,
↪))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an *annotation scope*. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

Notes

CHAPITRE 9

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.

9.1 Programmes Python complets

Bien que les spécifications d'un langage n'ont pas à préciser comment l'interpréteur du langage est invoqué, il est utile d'avoir des notions sur ce qu'est un programme Python complet. Un programme Python complet est exécuté dans un environnement dont l'initialisation est minimale : tous les modules intégrés et standard sont disponibles mais aucun n'a été initialisé, à l'exception de `sys` (divers services système), `builtins` (fonctions natives, exceptions et `None`) et `__main__`. Ce dernier est utilisé pour avoir des espaces de nommage locaux et globaux pour l'exécution du programme complet.

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 nommage de `__main__`.

Un programme complet peut être transmis à l'interpréteur sous trois formes : avec l'option `-c` *chaîne* en ligne de commande, avec un fichier passé comme premier argument de ligne de commande ou comme entrée standard. Si le fichier ou l'entrée standard est un périphérique tty, l'interpréteur entre en mode interactif ; sinon, il exécute le fichier comme un programme complet.

9.2 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 ;
- lors de l'analyse d'une chaîne de caractères passée à la fonction `exec()`.

9.3 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.

9.4 Entrée d'expression

`eval()` est utilisée pour évaluer les expressions entrées. Elle ignore les espaces en tête. L'argument de `eval()`, de type chaîne de caractères, doit être de la forme suivante :

```
eval_input   ::=   expression_list NEWLINE*
```

CHAPITRE 10

Spécification complète de la grammaire

Ceci est la grammaire complète de Python, issue directement de la grammaire utilisée pour générer l'analyseur syntaxique CPython (voir Grammar/python.gram). La version ci-dessous ne comprend pas les détails relatifs à la génération de code et la reprise sur erreur.

The notation is a mixture of EBNF and PEG. In particular, `&` followed by a symbol, token or parenthesized group indicates a positive lookahead (i.e., is required to match but not consumed), while `!` indicates a negative lookahead (i.e., is required *not* to match). We use the `|` separator to mean PEG's "ordered choice" (written as `/` in traditional PEG grammars). See **PEP 617** for more details on the grammar's syntax.

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   Python.
```

(suite sur la page suivante)

(suite de la page précédente)

```
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#            | first_alt
#            | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
# ==================

statements: statement+

statement: compound_stmt  | simple_stmts

statement_newline:
    | compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
    | return_stmt
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
    | 'continue'
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+

del_stmt:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from

# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block

# Function definitions
# --------------------

function_def:
    | decorators function_def_raw
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_
↪comment] block
    | ASYNC 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_
↪type_comment] block

# Function parameters
# -------------------

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
    | param default ',' TYPE_COMMENT?
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# -------------

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_
↪block]
    | ASYNC 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block␣
↪[else_block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | ASYNC 'with' '(' ','.with_item+ ','? ')' ':' block
    | ASYNC 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]


# Except statement
# ----------------
```

(suite sur la page suivante)

(suite de la page précédente)

```
except_block:
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ['as' NAME ] ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# ---------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' pattern_capture_target
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
    | name_or_attr '(' keyword_patterns ','? ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# ---------------

type_alias:
    | "type" NAME [type_params] '=' expression

# Type parameter declaration
# --------------------------

type_params: '[' type_param_seq  ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound]
    | '*' NAME
    | '**' NAME

type_param_bound: ':' expression

# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or
```

(suite sur la page suivante)

(suite de la page précédente)

```
eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
    | await_primary

# Primary elements
# ----------------
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Primary elements are things like "obj.something.something", "obj[something]",
↪"obj(something)", "obj" ...

await_primary:
    | AWAIT primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default*␣
↪[lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc

lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'
```

(suite sur la page suivante)

(suite de la page précédente)

```
lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
    | '{' (yield_expr | star_expressions) '='? [fstring_conversion] [fstring_full_
↪format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

string: STRING
strings: (fstring|string)+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']
```

(suite sur la page suivante)

(suite de la page précédente)

```
double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# ---------------------------

for_if_clauses:
    | for_if_clause+

for_if_clause:
    | ASYNC 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=') for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+␣
↪[',' kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
# ==================

# Generic targets
# ---------------
```

(suite sur la page suivante)

(suite de la page précédente)

```
# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'
```

(suite sur la page suivante)

(suite de la page précédente)

```
# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented block
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR ===========================



# ========================= START OF INVALID RULES ========================
```

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.

...

Peut faire référence à :

— L'invite de commande utilisée par défaut dans l'interpréteur interactif lorsqu'on entre un bloc de code indenté, dans des délimiteurs fonctionnant par paires (parenthèses, crochets, accolades, triple guillemets), ou après un avoir spécifié un décorateur.

— La constante `Ellipsis`.

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 mère abstraite

Les classes mères 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 subtilement fausses (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.abc`), les nombres (dans le module `numbers`), les flux (dans le module `io`) et les chercheurs-chargeurs du système d'importation (dans le module `importlib.abc`). Vous pouvez créer vos propres ABC avec le module `abc`.

annotation

Étiquette associée à une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour. Elle est utilisée par convention comme *type hint*.

Les annotations de variables locales ne sont pas accessibles au moment de l'exécution, mais les annotations de variables globales, d'attributs de classe et de fonctions sont stockées dans l'attribut spécial `__annotations__` des modules, classes et fonctions, respectivement.

Voir *annotation de variable*, *annotation de fonction*, les **PEP 484** et **PEP 526**, qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

argument

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 *paramètre* dans le glossaire, la question Différence entre argument et paramètre de la FAQ et la **PEP 362**.

gestionnaire de contexte asynchrone
: (*asynchronous context manager* en anglais) Objet contrôlant l'environnement à l'intérieur d'une instruction `async with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été Introduit par la **PEP 492**.

générateur asynchrone
: Fonction qui renvoie un *itérateur de générateur asynchrone*. Cela ressemble à une coroutine définie par `async def`, sauf qu'elle contient une ou des expressions `yield` produisant ainsi uns série de valeurs utilisables dans une boucle `async for`.

Générateur asynchrone fait généralement référence à une fonction, mais peut faire référence à un *itérateur de générateur asynchrone* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser l'ensemble des termes lève l'ambiguïté.

Un générateur asynchrone peut contenir des expressions `await` ainsi que des instructions `async for`, et `async with`.

itérateur de générateur asynchrone
: Objet créé par un *générateur asynchrone*.

C'est un *asynchronous iterator* qui, lorsqu'il est appelé via la méthode `__anext__()` renvoie un objet *awaitable* qui exécute le corps de la fonction du générateur asynchrone jusqu'au prochain `yield`.

Chaque `yield` suspend temporairement l'exécution, en gardant en mémoire l'emplacement et l'état de l'exécution (ce qui inclut les variables locales et les *try* en cours). Lorsque l'exécution de l'itérateur de générateur asynchrone reprend avec un nouvel *awaitable* renvoyé par `__anext__()`, elle repart de là où elle s'était arrêtée. Voir les **PEP 492** et **PEP 525**.

itérable asynchrone
: Objet qui peut être utilisé dans une instruction `async for`. Sa méthode `__aiter__()` doit renvoyer un *asynchronous iterator*. A été introduit par la **PEP 492**.

itérateur asynchrone
: Objet qui implémente les méthodes `__aiter__()` et `__anext__()`. `__anext__()` doit renvoyer un objet *awaitable*. Tant que la méthode `__anext__()` produit des objets *awaitable*, le `async for` appelant les consomme. L'itérateur asynchrone lève une exception `StopAsyncIteration` pour signifier la fin de l'itération. A été introduit par la **PEP 492**.

attribut
: Valeur associée à un objet et habituellement désignée par son nom *via* une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, cet attribut est référencé par *o.a*.

Il est possible de donner à un objet un attribut dont le nom n'est pas un identifiant tel que défini pour les *Identifiants et mots-clés*, par exemple en utilisant `setattr()`, si l'objet le permet. Un tel attribut ne sera pas accessible à l'aide d'une expression pointée et on devra y accéder avec `getattr()`.

attendable (*awaitable*)
: Objet pouvant être utilisé dans une expression `await`. Ce peut être une *coroutine* ou un objet avec une méthode `__await__()`. Voir aussi la **PEP 492**.

BDFL
: Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

fichier binaire
: A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

  Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets `str`.

référence empruntée
: In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

  Il est recommandé d'appeler `Py_INCREF()` sur la *référence empruntée*, ce qui la transforme *in situ* en une *référence forte*. Vous pouvez faire une exception si vous êtes certain que l'objet ne peut pas être supprimé avant la dernière utilisation de la référence empruntée. Voir aussi la fonction `Py_NewRef()`, qui crée une nouvelle *référence forte*.

objet octet-compatible
: Un objet gérant le protocole tampon et pouvant exporter un tampon (*buffer* en anglais) C-*contigu*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets `memoryview`. Les objets octets-compatibles peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

  Certaines opérations nécessitent de travailler sur des données binaires variables. La documentation parle de ceux-ci comme des *read-write bytes-like objects*. Par exemple, `bytearray` ou une `memoryview` d'un `bytearray` en font partie. D'autres opérations nécessitent de travailler sur des données binaires stockées dans des objets immuables (« *objets octets-compatibles en lecture seule* »), par exemple des `bytes` ou des `memoryview` d'un objet `bytes`.

code intermédiaire (*bytecode*)
: Le code source, en Python, est compilé en un code intermédiaire (*bytecode* en anglais), la représentation interne à CPython d'un programme Python. Le code intermédiaire est mis en cache dans un fichier `.pyc` de manière à ce qu'une seconde exécution soit plus rapide (la compilation en code intermédiaire a déjà été faite). On dit que ce *langage intermédiaire* est exécuté sur une *virtual machine* qui exécute des instructions machine pour chaque instruction du code intermédiaire. Notez que le code intermédiaire n'a pas vocation à fonctionner sur différentes machines virtuelles Python ou à être stable entre différentes versions de Python.

  La documentation du module dis fournit une liste des instructions du code intermédiaire.

appelable (*callable*)
: Un appelable est un objet qui peut être appelé, éventuellement avec un ensemble d'arguments (voir *argument*), avec la syntaxe suivante :

```
callable(argument1, argument2, argumentN)
```

  Une *fonction*, et par extension une *méthode*, est un appelable. Une instance d'une classe qui implémente la méthode *`__call__()`* est également un appelable.

fonction de rappel (*callback*)
: Une fonction (classique, par opposition à une coroutine) passée en argument pour être exécutée plus tard.

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.

variable de classe
: Une variable définie dans une classe et destinée à être modifiée uniquement au niveau de la classe (c'est-à-dire, pas dans une instance de la classe).

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
An object which controls the environment seen in a *with* statement by defining *__enter__()* and *__exit__()* methods. See **PEP 343**.

variable de contexte
Une variable qui peut avoir des valeurs différentes en fonction de son contexte. Cela est similaire au stockage par fil d'exécution (*Thread Local Storage* en anglais) dans lequel chaque fil d'exécution peut avoir une valeur différente pour une variable. Toutefois, avec les variables de contexte, il peut y avoir plusieurs contextes dans un fil d'exécution et l'utilisation principale pour les variables de contexte est de garder une trace des variables dans les tâches asynchrones concourantes. Voir `contextvars`.

contigu
Un tampon (*buffer* en anglais) est considéré comme contigu s'il est soit *C-contigu* soit *Fortran-contigu*. Les tampons de dimension zéro sont C-contigus et Fortran-contigus. Pour un tableau à une dimension, ses éléments doivent être placés en mémoire l'un à côté de l'autre, dans l'ordre croissant de leur indice, en commençant à zéro. Pour qu'un tableau multidimensionnel soit C-contigu, le dernier indice doit être celui qui varie le plus rapidement lors du parcours de ses éléments dans l'ordre de leur adresse mémoire. À l'inverse, dans les tableaux Fortran-contigu, c'est le premier indice qui doit varier le plus rapidement.

coroutine
Les coroutines sont une forme généralisée des fonctions. On entre dans une fonction en un point et on en sort en un autre point. On peut entrer, sortir et reprendre l'exécution d'une coroutine en plusieurs points. Elles peuvent être implémentées en utilisant l'instruction *async def*. Voir aussi la **PEP 492**.

fonction coroutine
Fonction qui renvoie un objet *coroutine*. Une fonction coroutine peut être définie par l'instruction *async def* et peut contenir les mots clés *await*, *async for* ainsi que *async with*. A été introduit par la **PEP 492**.

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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

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 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* ou le guide pour l'utilisation des 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.

dictionnaire en compréhension (ou dictionnaire en intension)
: Écriture concise pour traiter tout ou partie des éléments d'un itérable et renvoyer un dictionnaire contenant les résultats. `results = {n: n ** 2 for n in range(10)}` génère un dictionnaire contenant des clés `n` liées à leurs valeurs `n ** 2`. Voir *compréhensions*.

vue de dictionnaire
: Objets retournés par les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Ils fournissent des vues dynamiques des entrées du dictionnaire, ce qui signifie que lorsque le dictionnaire change, la vue change. Pour transformer une vue en vraie liste, utilisez `list(dictview)`. Voir dict-views.

chaîne de documentation (*docstring*)
: A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

typage canard (*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 mère 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
: Suite logique de termes et chiffres conformes à la syntaxe Python dont l'évaluation fournit une valeur. En d'autres termes, une expression est une suite d'éléments tels que des noms, opérateurs, littéraux, accès d'attributs, méthodes ou fonctions qui aboutissent à une valeur. Contrairement à beaucoup d'autres langages, les différentes constructions du langage ne sont pas toutes des expressions. On trouve également des *instructions* qui ne peuvent pas être utilisées comme expressions, tel que `while`. Les affectations sont également des instructions et non des 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.

f-string
: Chaîne littérale préfixée de `'f'` ou `'F'`. Les "f-strings" sont un raccourci pour *formatted string literals*. Voir la **PEP 498**.

objet fichier
: An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

  Il existe en réalité trois catégories de fichiers objets : les *fichiers binaires* bruts, les *fichiers binaires* avec tampon (*buffer*) et les *fichiers textes*. Leurs interfaces sont définies dans le module `io`. Le moyen le plus simple et direct de créer un objet fichier est d'utiliser la fonction `open()`.

objet fichier-compatible
: Synonyme de *objet fichier*.

encodage du système de fichiers et gestionnaire d'erreurs associé
: Encodage et gestionnaire d'erreurs utilisés par Python pour décoder les octets fournis par le système d'exploitation et encoder les chaînes de caractères Unicode afin de les passer au système.

L'encodage du système de fichiers doit impérativement pouvoir décoder tous les octets jusqu'à 128. Si ce n'est pas le cas, certaines fonctions de l'API lèvent `UnicodeError`.

Cet encodage et son gestionnaire d'erreur peuvent être obtenus à l'aide des fonctions `sys.getfilesystemencoding()` et `sys.getfilesystemencodeerrors()`.

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction `PyConfig_Read()` : regardez `filesystem_encoding` et `filesystem_errors` dans les membres de `PyConfig`.

Voir aussi *encodage régional*.

chercheur
: Objet qui essaie de trouver un *chargeur* pour le module en cours d'importation.

Depuis Python 3.3, il existe deux types de chercheurs : les *chercheurs dans les méta-chemins* à utiliser avec `sys.meta_path` ; les *chercheurs d'entrée dans path* à utiliser avec `sys.path_hooks`.

Voir les **PEP 302**, **PEP 420** et **PEP 451** pour plus de détails.

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*.

annotation de fonction
: *annotation* d'un paramètre de fonction ou valeur de retour.

Les annotations de fonctions sont généralement utilisées pour des *indications de types* : par exemple, cette fonction devrait prendre deux arguments `int` et devrait également avoir une valeur de retour de type `int` :

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

L'annotation syntaxique de la fonction est expliquée dans la section *Définition de fonctions*.

Voir *annotation de variable* et la **PEP 484**, qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

__future__
: Une *importation depuis le futur* s'écrit `from __future__ import <fonctionnalité>`. Lorsqu'une importation du futur est active dans un module, Python compile ce module avec une certaine modification de la syntaxe ou du comportement qui est vouée à devenir standard dans une version ultérieure. Le module `__future__` documente les possibilités pour *fonctionnalité*. L'importation a aussi l'effet normal d'importer une variable du module. Cette variable contient des informations utiles sur la fonctionnalité en question, notamment la version de Python dans laquelle elle a été ajoutée, et celle dans laquelle elle deviendra standard :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

ramasse-miettes
: (*garbage collection* en anglais) 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. Le ramasse-miettes peut être contrôlé en utilisant le module `gc`.

générateur
: Fonction qui renvoie un *itérateur de générateur*. Cela ressemble à une fonction normale, en dehors du fait qu'elle contient une ou des expressions `yield` produisant une série de valeurs utilisable dans une boucle *for* ou récupérées une à une via la fonction `next()`.

Fait généralement référence à une fonction génératrice mais peut faire référence à un *itérateur de générateur* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser les termes complets lève l'ambiguïté.

itérateur de générateur
: Objet créé par une fonction *générateur*.

Chaque `yield` suspend temporairement l'exécution, en se rappelant l'endroit et l'état de l'exécution (y compris les variables locales et les *try* en cours). Lorsque l'itérateur de générateur reprend, il repart là où il en était (contrairement à une fonction qui prendrait un nouveau départ à chaque invocation).

expression génératrice
Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une clause `for` définissant une variable de boucle, un intervalle et une clause `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
```

fonction générique
Fonction composée de plusieurs fonctions implémentant les mêmes opérations pour différents types. L'implémentation à utiliser est déterminée lors de l'appel par l'algorithme de répartition.

Voir aussi *single dispatch*, le décorateur `functools.singledispatch()` et la **PEP 443**.

type générique
Un *type* qui peut être paramétré ; généralement un *conteneur* comme `list` ou `dict`. Utilisé pour les *indications de type* et les *annotations*.

Pour plus de détails, voir types alias génériques et le module `typing`. On trouvera l'historique de cette fonctionnalité dans les **PEP 483**, **PEP 484** et **PEP 585**.

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.

***pyc* utilisant le hachage**
Un fichier de cache de code intermédiaire (*bytecode* en anglais) qui utilise le hachage plutôt que l'heure de dernière modification du fichier source correspondant pour déterminer sa validité. Voir *Invalidation de bytecode mis en cache*.

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__()` 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*.

La plupart des types immuables natifs de Python sont hachables, mais les conteneurs mutables (comme les listes ou les dictionnaires) ne le sont pas ; les conteneurs immuables (comme les n-uplets ou les ensembles figés) ne sont hachables que si leurs éléments sont hachables. Les instances de classes définies par les utilisateurs sont hachables par défaut. Elles sont toutes considérées différentes (sauf avec elles-mêmes) et leur valeur de hachage est calculée à partir de leur `id()`.

IDLE
Environnement d'apprentissage et 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.

chemin des importations
Liste de *entrées* dans lesquelles le *chercheur basé sur les chemins* cherche les modules à importer. Typiquement, lors d'une importation, cette liste vient de `sys.path` ; pour les sous-paquets, elle peut aussi venir de l'attribut `__path__` du paquet parent.

importation
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*.

arrêt de l'interpréteur
Lorsqu'on lui demande de s'arrêter, l'interpréteur Python entre dans une phase spéciale où il libère graduellement les ressources allouées, comme les modules ou quelques structures de données internes. Il fait aussi quelques appels au *ramasse-miettes*. Cela peut déclencher l'exécution de code dans des destructeurs ou des fonctions de rappels de *weakrefs*. Le code exécuté lors de l'arrêt peut rencontrer des exceptions puisque les ressources auxquelles il fait appel sont susceptibles de ne plus fonctionner, (typiquement les modules des bibliothèques ou le mécanisme de *warning*).

La principale raison d'arrêt de l'interpréteur est que le module `__main__` ou le script en cours d'exécution a terminé de s'exécuter.

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`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a *`__getitem__()`* method that implements *sequence* semantics.

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 (or passing it to the built-in function `next()`) 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.

Particularité de l'implémentation CPython : CPython does not consistently apply the requirement that an iterator define `__iter__()`.

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 clés pour déterminer comment les éléments sont classés ou groupés. On peut citer les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` et `itertools.groupby()`.

Il existe plusieurs moyens de créer une fonction clé. Par exemple, la méthode `str.lower()` peut servir de fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clés avec des expressions *`lambda`*, comme `lambda r: (r[0], r[2])`. Par ailleurs `attrgetter()`, `itemgetter()` et `methodcaller()` permettent de créer des fonctions clés. Voir le guide pour le tri pour des exemples de création et d'utilisation de fonctions clefs.

argument nommé
Voir *argument*.

lambda
Fonction anonyme sous la forme d'une *expression* et ne contenant qu'une seule expression, exécutée lorsque la fonction est appelée. La syntaxe pour créer des fonctions lambda est : `lambda [parameters]: expression`

LBYL
Regarde avant de sauter, (*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 "sauter". 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.

liste
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)
Écriture concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` génère la liste composée des nombres pairs de 0 à 255 écrits sous formes de chaînes de caractères et en hexadécimal (`0x…`). La clause *`if`* est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

chargeur
Objet qui charge un module. Il doit définir une méthode nommée `load_module()`. Un chargeur est typiquement donné par un *chercheur*. Voir la **PEP 302** pour plus de détails et `importlib.ABC.Loader` pour sa *classe mère abstraite*.

encodage régional
Sous Unix, il est défini par la variable régionale LC_CTYPE. Il peut être modifié par `locale.setlocale(locale.LC_CTYPE, new_locale)`.

Sous Windows, c'est un encodage ANSI (par ex. : `"cp1252"`).

Sous Android et VxWorks, Python utilise `"utf-8"` comme encodage régional.

`locale.getencoding()` can be used to get the locale encoding.

Voir aussi l'*encodage du systèmes de fichiers et gestionnaire d'erreurs associé*.

méthode magique
Un synonyme informel de *special method*.

tableau de correspondances (*mapping* en anglais)
Conteneur permettant de rechercher des éléments à partir de clés et implémentant les méthodes spécifiées dans les classes mères abstraites des `tableaux de correspondances` (immuables) ou `tableaux de correspondances mutables` (voir les classes mères abstraites). Les classes suivantes sont des exemples de tableaux de correspondances : `dict`, `collections.defaultdict`, `collections.OrderedDict` et `collections.Counter`.

chercheur dans les méta-chemins
: Un *chercheur* renvoyé par une recherche dans `sys.meta_path`. Les chercheurs dans les méta-chemins ressemblent, mais sont différents des *chercheurs d'entrée dans path*.

  Voir `importlib.abc.MetaPathFinder` pour les méthodes que les chercheurs dans les méta-chemins doivent implémenter.

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'auront 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 sûrs les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

  Plus d'informations sont disponibles dans : *Métaclasses*.

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
: Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

module
: Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

  Voir aussi *paquet*.

spécificateur de module
: Espace de nommage contenant les informations, relatives à l'importation, utilisées pour charger un module. C'est une instance de la classe `importlib.machinery.ModuleSpec`.

MRO
: Voir *ordre de résolution des méthodes*.

mutable
: Un objet mutable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

***n*-uplet nommé**
: Le terme "n-uplet nommé" s'applique à tous les types ou classes qui héritent de la classe `tuple` et dont les éléments indexables sont aussi accessibles en utilisant des attributs nommés. Les types et classes peuvent avoir aussi d'autres caractéristiques.

  Plusieurs types natifs sont appelés n-uplets, y compris les valeurs retournées par `time.localtime()` et `os.stat()`. Un autre exemple est `sys.float_info` :

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

  Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

espace de nommage
: L'endroit où une variable est stockée. Les espaces de nommage sont implémentés avec des dictionnaires. Il existe des espaces de nommage globaux, natifs ou imbriqués dans les objets (dans les méthodes). Les espaces de nommage favorisent la modularité car ils permettent d'éviter les conflits de noms. Par exemple, les fonctions `builtins.open` et `os.open()` sont différenciées par leurs espaces de nom. Les espaces de nommage

aident aussi à la lisibilité et la maintenabilité en rendant clair quel module implémente une fonction. Par exemple, écrire `random.seed()` ou `itertools.islice()` affiche clairement que ces fonctions sont implémentées respectivement dans les modules `random` et `itertools`.

paquet-espace de nommage
Un *paquet* tel que défini dans la **PEP 421** qui ne sert qu'à contenir des sous-paquets. Les paquets-espace de nommage peuvent n'avoir aucune représentation physique et, plus spécifiquement, ne sont pas comme un *paquet classique* puisqu'ils n'ont pas de fichier `__init__.py`.

Voir aussi *module*.

portée imbriquée
Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonction définie à l'intérieur d'une autre fonction a accès aux variables de cette dernière. Souvenez-vous cependant que cela ne fonctionne que pour accéder à des variables, pas pour les assigner. Les variables locales sont lues et assignées dans l'espace de nommage le plus proche. Tout comme les variables globales qui sont stockés dans l'espace de nommage global, le mot clef *`nonlocal`* permet d'écrire dans l'espace de nommage dans lequel est déclarée la variable.

nouvelle classe
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like *`__slots__`*, descriptors, properties, *`__getattribute__()`*, class methods, and static methods.

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__`.

Voir aussi *paquet classique* et *namespace package*.

paramètre
Entité nommée dans la définition d'une *fonction* (ou méthode), décrivant un *argument* (ou dans certains cas des arguments) que la fonction accepte. Il existe cinq sortes de paramètres :

— *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* : définit un argument qui ne peut être fourni que par position. Les paramètres *positional-only* peuvent être définis en insérant un caractère "/" dans la liste de paramètres de la définition de fonction après eux. Par exemple : *posonly1* et *posonly2* dans le code suivant :

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

— *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (`*`) seule dans la liste des paramètres avant eux. Par exemple, *kw_only1* et *kw_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

— *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.

Voir aussi *argument* dans le glossaire, la question sur la différence entre les arguments et les paramètres dans la FAQ, la classe `inspect.Parameter`, la section *Définition de fonctions* et la **PEP 362**.

entrée de chemin
Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

chercheur de chemins
chercheur renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

point d'entrée pour la recherche dans *path*
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

chercheur basé sur les chemins
L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

objet simili-chemin
Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet `str` ou un objet `bytes` représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin `str` ou `bytes` du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type `str` ou `bytes` à la place. A été Introduit par la **PEP 519**.

PEP
Python Enhancement Proposal (Proposition d'amélioration de Python). Une PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

Les PEP sont censées être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

Voir la **PEP 1**.

portion
Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la **PEP 420**.

argument positionnel
Voir *argument*.

API provisoire
Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changements cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

Même pour les API provisoires, les changements cassant la rétrocompatibilité sont considérés comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problèmes en conservant la rétrocompatibilité.

Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la **PEP 411** pour plus de détails.

paquet provisoire
Voir *provisional API*.

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)
```

nom qualifié
Nom, comprenant des points, montrant le "chemin" de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la **PEP 3155**. Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name* - *FQN* en anglais) signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

nombre de références
The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are "immortal" and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

paquet classique
paquet traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.
Voir aussi *paquet-espace de nommage*.

__slots__
Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

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 `bytes`. 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.
The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just *__getitem__()* and *__len__()*, adding `count()`, `index()`, *__contains__()*, and *__reversed__()*. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

ensemble en compréhension (ou ensemble en intension)
Une façon compacte de traiter tout ou partie des éléments d'un itérable et de renvoyer un *set* avec les résultats. `results = {c for c in 'abracadabra' if c not in 'abc'}` génère l'ensemble contenant les lettres « r » et « d » `{'r', 'd'}`. Voir *Agencements des listes, ensembles et dictionnaires*.

distribution simple
Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

tranche
(*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets `slice` en interne.

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`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

référence forte
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

Une référence forte est créée à l'aide de la fonction `Py_NewRef()`. Il faut normalement appeler `Py_DECREF()` dessus avant de sortir de sa portée lexicale, sans quoi il y a une fuite de référence.

Voir aussi *référence empruntée*.

encodages de texte
Une chaîne de caractères en Python est une suite de points de code Unicode (dans l'intervalle `U+0000`--`U+10FFFF`). Pour stocker ou transmettre une chaîne, il est nécessaire de la sérialiser en suite d'octets.

Sérialiser une chaîne de caractères en une suite d'octets s'appelle « encoder » et recréer la chaîne à partir de la suite d'octets s'appelle « décoder ».

Il existe de multiples codecs pour la sérialisation de texte, que l'on regroupe sous l'expression « encodages de texte ».

fichier texte
Objet fichier capable de lire et d'écrire des objets `str`. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*encodage de texte* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte (`'r'` ou `'w'`), `sys.stdin`, `sys.stdout` et les instances de `io.StringIO`.

Voir aussi *fichier binaire* pour un objet fichier capable de lire et d'écrire des *objets octets-compatibles*.

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)`.

alias de type
Synonyme d'un type, créé en affectant le type à un identifiant.

Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci :

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

indication de type
: L'*annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Les indications de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultées en utilisant `typing.get_type_hints()`.

Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

retours à la ligne universels
: Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la **PEP 278** et la **PEP 3116**, ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

annotation de variable
: *annotation* d'une variable ou d'un attribut de classe.

Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative :

```
class C:
    field: 'annotation'
```

Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int` :

```
count: int = 0
```

La syntaxe d'annotation de variable est expliquée dans la section *Les assignations annotées*.

Reportez-vous à *annotation de fonction*, à la **PEP 484** et à la **PEP 526** qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

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.

Voir aussi `venv`.

machine virtuelle
: Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *code intermédiaire* 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/) aux 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.

En mai 2000, Guido et l'équipe de développement centrale de Python sont partis vers BeOpen.com pour former l'équipe BeOpen PythonLabs. En octobre de la même année, l'équipe de PythonLabs est partie vers Digital Creations (désormais Zope Corporation ; voir https://www.zope.com/). En 2001, la Python Software Foundation (PSF, voir https://www.python.org/psf/) voit le jour. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de Python. Zope Corporation en est un sponsor.

Toutes les versions de Python sont Open Source (voir https://www.opensource.org/ pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et ultérieure	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 change-

ments. 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-compatibles 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

Le logiciel Python et sa documentation sont distribués sous *la licence d'utilisation PSF*.

Depuis Python 3.8.6, les exemples, recettes et autres codes présents dans la documentation sont sous la double licence d'utilisation PSF et *la licence Zero-Clause BSD*.

Certains logiciels faisant partie de Python sont soumis à d'autres licences. Ces licences sont incluses avec le code lié à celles-ci. Voir *Licences et remerciements pour les logiciels tiers* pour une liste non exhaustive de ces licences.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.12.3

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.12.3 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 3.12.3 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.12.3 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 3.12.3 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
   3.12.3.

4. PSF is making Python 3.12.3 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 3.12.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

```
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.3
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.3, 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 3.12.3, 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

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

C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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.2.5 LICENCE BSD ZERO-CLAUSE POUR LE CODE DANS LA DOCUMENTATION DE PYTHON 3.12.3

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 tiers

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

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code :

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 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*)

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Interfaces de connexion asynchrones

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice :

```
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.3.4 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
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.3.5 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.3.6 Les fonctions UUencode et UUdecode

Le module `uu` contient la note suivante :

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module `xmlrpc.client` contient la note suivante :

```
    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.3.8 test_epoll

The `test.test_epoll` module contains the following notice :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 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.3.10 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 *strtod* et *dtoa*

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice :

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies :

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
```

(suite sur la page suivante)

(suite de la page précédente)

```
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
```

(suite sur la page suivante)

(suite de la page précédente)

```
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--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.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--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.3.15 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) 1995-2011 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
```

C.3.16 cfuhash

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet *cfuhash* :

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * 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.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec` :

```
Copyright (c) 2008-2020 Stefan Krah. 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.3.18 Ensemble de tests C14N du W3C

Les tests de C14N version 2.0 du module `test` (`Lib/test/xmltestdata/c14n-20/`) proviennent du site du W3C à l'adresse https://www.w3.org/TR/xml-c14n2-testcases/ et sont distribués sous licence BSD modifiée :

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.
```

C.3.19 Audioop

The audioop module uses the code base in g771.c file of the SoX project. https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz

> This source code is a product of Sun Microsystems, Inc. and is provided for unrestricted use. Users may copy or modify this source code without charge.
>
> SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
>
> Sun source code is provided with no support and without any obligation on the part of Sun Microsystems, Inc. to assist in its use, correction, modification or enhancement.
>
> SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE OR ANY PART THEREOF.
>
> In no event will Sun Microsystems, Inc. be liable for any lost revenue or profits or other special, indirect and consequential damages, even if Sun has been advised of the possibility of such damages.
>
> Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, California 94043

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license :

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.
```

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

. . ., **155**
    ellipsis literal, 21
'''
    string literal, 11
. (*dot*)
    in numeric literal, 16
! (*exclamation*)
    in formatted string literal, 13
– (*moins*)
    opérateur binaire, 91
    opérateur unaire, 90
. (*point*)
    référence à un attribut, 86
' (*single quote*)
    string literal, 10
! motifs, 122
" (*double quote*)
    string literal, 10
"""
    string literal, 11
(*hash*)
    comment, 6
    source encoding declaration, 6
% (*pourcentage*)
    opérateur, 91
%=
    affectation augmentée, 102
& (*esperluette*)
    opérateur, 92
&=
    affectation augmentée, 102
() (*parenthèses*)
    agencement de *n*-uplet, 78
    appel, 87
    dans la liste cible d'affectation, 100
    définition de classe, 130
    définition de fonction, 128
    expression génératrice, 81
* (*astérisque*)
    dans la liste cible d'affectation, 100
    dans les appels de fonction, 88
    dans les expressions de liste, 96
    définition de fonction, 129
    instruction *import*, 108
    opérateur, 90
**
    dans les agencements de dictionnaire, 80
    dans les appels de fonction, 88
    définition de fonction, 129
    opérateur, 89
**=
    affectation augmentée, 102
*=
    affectation augmentée, 102
n-uplet
    objet, 86, 87, 96
    vide, 78
+ (*plus*)
    opérateur binaire, 91
    opérateur unaire, 90
+=
    affectation augmentée, 102
, (*virgule*), 78
    dans la liste cible, 100
    dans les agencements de dictionnaire, 80
    découpage, 87
    instruction *import*, 107
    instruction *with*, 118
    liste d'arguments, 87
    liste de paramètres, 128
    liste d'expressions, 79, 80, 96, 103, 130
    liste d'identifiants, 109, 110
/ (*barre oblique*)
    définition de fonction, 129
    opérateur, 90
//
    opérateur, 90
//=
    affectation augmentée, 102
/=
    affectation augmentée, 102
0b

integer literal, 15
0o
integer literal, 15
0x
integer literal, 15
2to3, **155**
: (*colon*)
in formatted string literal, 13
: (*deux-points*)
annotations de fonction, 129
dans les expressions de dictionnaire, 80
découpage, 87
expression lambda, 96
instruction composée, 114, 115, 118, 120, 128, 130
variable annotée, 102
:= (*deux points égal*), 95
; (*point-virgule*), 113
< (*plus petit*)
opérateur, 92
<<
opérateur, 91
<<=
affectation augmentée, 102
<=
opérateur, 92
!=
opérateur, 92
-=
affectation augmentée, 102
= (*égal*)
dans les appels de fonction, 87
définition de fonction, 128
instruction d'affectation, 100
= (*equals*)
class definition, 43
for help in debugging using string literals, 13
==
opérateur, 92
->
annotations de fonction, 129
> (*plus grand*)
opérateur, 92
>=
opérateur, 92
>>
opérateur, 91
>>=
affectation augmentée, 102
>>>, **155**
@ (*arobase*)
définition de classe, 130
définition de fonction, 128
opérateur, 90
[] (*crochets*)
dans la liste cible d'affectation, 100
expression de liste, 79
sélection (*ou indiçage*), 86
\ (*backslash*)
escape sequence, 11
\\
escape sequence, 11
\a
escape sequence, 11
\b
escape sequence, 11
\f
escape sequence, 11
\N
escape sequence, 11
\n
escape sequence, 11
\r
escape sequence, 11
\t
escape sequence, 11
\U
escape sequence, 11
\u
escape sequence, 11
\v
escape sequence, 11
\x
escape sequence, 11
^ (*caret*)
opérateur, 92
^=
affectation augmentée, 102
_ (*underscore*)
in numeric literal, 15, 16
_, identifiers, 9
__, identifiers, 9
__abs__() (*méthode object*), 51
__add__() (*méthode object*), 50
__aenter__() (*méthode object*), 56
__aexit__() (*méthode object*), 56
__aiter__() (*méthode object*), 55
__all__ (*attribut de module facultatif*), 108
__and__() (*méthode object*), 50
__anext__() (*méthode agen*), 85
__anext__() (*méthode object*), 55
__annotations__ (*attribut function*), 25
__annotations__ (*class attribute*), 28
__annotations__ (*function attribute*), 25
__annotations__ (*module attribute*), 28
__await__() (*méthode object*), 54
__bases__ (*class attribute*), 28
__bool__() (*méthode object*), 38
__bool__() (*object method*), 48
__buffer__() (*méthode object*), 53
__bytes__() (*méthode object*), 36
__cached__, 70

__call__() (*méthode d'objet*), 89
__call__() (*méthode object*), 48
__cause__ (*attribut d'exception*), 105
__ceil__() (*méthode object*), 51
__class__ (*instance attribute*), 29
__class__ (*method cell*), 44
__class__ (*module attribute*), 39
__class_getitem__() (*méthode de la classe object*), 46
__classcell__ (*class namespace entry*), 44
__closure__ (*attribut function*), 24
__closure__ (*function attribute*), 24
__code__ (*attribut function*), 25
__code__ (*function attribute*), 25
__complex__() (*méthode object*), 51
__contains__() (*méthode object*), 49
__context__ (*attribut d'exception*), 105
__debug__, 103
__defaults__ (*attribut function*), 25
__defaults__ (*function attribute*), 25
__del__() (*méthode object*), 35
__delattr__() (*méthode object*), 39
__delete__() (*méthode object*), 40
__delitem__() (*méthode object*), 49
__dict__ (*attribut function*), 25
__dict__ (*class attribute*), 28
__dict__ (*function attribute*), 25
__dict__ (*instance attribute*), 29
__dict__ (*module attribute*), 28
__dir__ (*module attribute*), 39
__dir__() (*méthode object*), 39
__divmod__() (*méthode object*), 50
__doc__ (*attribut function*), 25
__doc__ (*attribut method*), 26
__doc__ (*class attribute*), 28
__doc__ (*function attribute*), 25
__doc__ (*method attribute*), 25
__doc__ (*module attribute*), 28
__enter__() (*méthode object*), 52
__eq__() (*méthode object*), 36
__exit__() (*méthode object*), 52
__file__, 70
__file__ (*module attribute*), 28
__float__() (*méthode object*), 51
__floor__() (*méthode object*), 51
__floordiv__() (*méthode object*), 50
__format__() (*méthode object*), 36
__func__ (*attribut method*), 26
__func__ (*method attribute*), 25
__future__, **160**
    instruction *future*, 108
__ge__() (*méthode object*), 36
__get__() (*méthode object*), 40
__getattr__ (*module attribute*), 39
__getattr__() (*méthode object*), 38
__getattribute__() (*méthode object*), 38
__getitem__() (*mapping object method*), 34
__getitem__() (*méthode object*), 48
__globals__ (*attribut function*), 24
__globals__ (*function attribute*), 24
__gt__() (*méthode object*), 36
__hash__() (*méthode object*), 37
__iadd__() (*méthode object*), 51
__iand__() (*méthode object*), 51
__ifloordiv__() (*méthode object*), 51
__ilshift__() (*méthode object*), 51
__imatmul__() (*méthode object*), 51
__imod__() (*méthode object*), 51
__imul__() (*méthode object*), 51
__index__() (*méthode object*), 51
__init__() (*méthode object*), 35
__init_subclass__() (*méthode de la classe object*), 42
__instancecheck__() (*méthode class*), 45
__int__() (*méthode object*), 51
__invert__() (*méthode object*), 51
__ior__() (*méthode object*), 51
__ipow__() (*méthode object*), 51
__irshift__() (*méthode object*), 51
__isub__() (*méthode object*), 51
__iter__() (*méthode object*), 49
__itruediv__() (*méthode object*), 51
__ixor__() (*méthode object*), 51
__kwdefaults__ (*attribut function*), 25
__kwdefaults__ (*function attribute*), 25
__le__() (*méthode object*), 36
__len__() (*mapping object method*), 38
__len__() (*méthode object*), 48
__length_hint__() (*méthode object*), 48
__loader__, 70
__lshift__() (*méthode object*), 50
__lt__() (*méthode object*), 36
__main__
    module, 58, 137
__matmul__() (*méthode object*), 50
__missing__() (*méthode object*), 49
__mod__() (*méthode object*), 50
__module__ (*attribut function*), 25
__module__ (*attribut method*), 26
__module__ (*class attribute*), 28
__module__ (*function attribute*), 25
__module__ (*method attribute*), 25
__mro_entries__() (*méthode object*), 43
__mul__() (*méthode object*), 50
__name__, 70
__name__ (*attribut function*), 25
__name__ (*attribut method*), 26
__name__ (*class attribute*), 28
__name__ (*function attribute*), 25
__name__ (*method attribute*), 25
__name__ (*module attribute*), 28
__ne__() (*méthode object*), 36
__neg__() (*méthode object*), 51
__new__() (*méthode object*), 35
__next__() (*méthode generator*), 83
__objclass__ (*attribut object*), 40

__or__() (*méthode object*), 50
__package__, 70
__path__, 70
__pos__() (*méthode object*), 51
__pow__() (*méthode object*), 50
__prepare__ (*metaclass method*), 44
__qualname__ (*attribut function*), 25
__radd__() (*méthode object*), 50
__rand__() (*méthode object*), 50
__rdivmod__() (*méthode object*), 50
__release_buffer__() (*méthode object*), 53
__repr__() (*méthode object*), 36
__reversed__() (*méthode object*), 49
__rfloordiv__() (*méthode object*), 50
__rlshift__() (*méthode object*), 50
__rmatmul__() (*méthode object*), 50
__rmod__() (*méthode object*), 50
__rmul__() (*méthode object*), 50
__ror__() (*méthode object*), 50
__round__() (*méthode object*), 51
__rpow__() (*méthode object*), 50
__rrshift__() (*méthode object*), 50
__rshift__() (*méthode object*), 50
__rsub__() (*méthode object*), 50
__rtruediv__() (*méthode object*), 50
__rxor__() (*méthode object*), 50
__self__ (*attribut method*), 26
__self__ (*method attribute*), 25
__set__() (*méthode object*), 40
__set_name__() (*méthode object*), 42
__setattr__() (*méthode object*), 38
__setitem__() (*méthode object*), 49
__slots__, **167**
__spec__, 70
__str__() (*méthode object*), 36
__sub__() (*méthode object*), 50
__subclasscheck__() (*méthode class*), 45
__traceback__ (*attribut d'exception*), 105
__truediv__() (*méthode object*), 50
__trunc__() (*méthode object*), 51
__type_params__ (*attribut function*), 25
__type_params__ (*class attribute*), 28
__type_params__ (*function attribut*), 25
__xor__() (*méthode object*), 50
``None``
    object, 20
{} (*accolades*)
    expression de dictionnaire, 80
    expression d'ensemble, 80
{} (*curly brackets*)
    in formatted string literal, 13
| (*barre verticale*)
    opérateur, 92
|=
    affectation augmentée, 102
~ (*tilde*)
    opérateur, 90

A

abs
    built-in function, 51
aclose() (*méthode agen*), 85
addition, 91
affectation
    annotée, 102
    attribut, 100
    augmentée, 102
    cible liste, 100
    découpage, 101
    instruction, 100
    sélection (*ou indiçage*), 101
agencement
    dictionnaire, 80
    ensemble, 80
    liste, 79
alias de type, **168**
and
    opérateur, 95
    OU (*bit à bit*), 92
annotation, **155**
annotation de fonction, **160**
annotation de variable, **169**
annotations
    fonction, 129
annotée
    affectation, 102
anonyme
    fonction, 96
API provisoire, **166**
appartenance
    test, 95
appel, 87
    défini par l'utilisateur fonction, 89
    fonction, 89
    fonction native, 89
    instance, 89
    instance de classe, 89
    méthode, 89
    méthode native, 89
    objet classe, 89
    procédure, 99
appelable
    objet, 87
appelable (*callable*), **157**
argument, **155**
    définition de fonction, 128
    function, 24
    sémantique des appels, 87
argument nommé, **163**
argument positionnel, **166**
arithmétique
    conversion, 77
    opération, binaire, 90
    opération, unaire, 90
array

module, 23
arrêt de l'interpréteur, **162**
as
clause except, 116
instruction *import*, 107
instruction *match*, 120
instruction *with*, 118
mot-clé, 107, 115, 118, 120
ASCII, 4, 10
asend() (*méthode agen*), 85
assert
instruction, **103**
AssertionError
exception, 103
assertions
débogage, 103
assignment
class attribute, 28
class instance attribute, 29
statement, 23
async
mot-clé, 131
async def
instruction, 131
async for
dans les compréhensions, 79
instruction, 131
async with
instruction, 132
asynchronous generator
asynchronous iterator, 27
function, 27
athrow() (*méthode agen*), 85
atome, 77
attendable (*awaitable*), **156**
attribut, **156**
affectation, 100
effacement, 104
référence, 86
attribute, 20
assignment, class, 28
assignment, class instance, 29
class, 28
class instance, 29
generic special, 20
special, 20
AttributeError
exception, 86
augmentée
affectation, 102
await
dans les compréhensions, 79
mot-clé, 89, 131

B

b'
bytes literal, 11
b"
bytes literal, 11
backslash character, 6
BDFL, **157**
binaire
arithmétique opération, 90
OU (*bit à bit*) opération, 92
binary literal, 15
binding
name, 57
blank line, 7
bloc *case*, 122
bloc attrape-tout, 122
block, 57
code, 57
BNF, 4, 77
Boolean
object, 21
booléenne
opération, 95
boucle
instruction, 106, 107, 114
break
instruction, **106**, 114, 115, 117, 118
built-in
method, 27
built-in function
abs, 51
bytes, 36
chr, 22
complex, 51
divmod, 50
float, 51
hash, 37
id, 19
int, 51
len, 22, 23, 48
object, 27
open, 29
ord, 22
pow, 50
print, 36
round, 52
slice, 34
type, 19, 43
built-in method
object, 27
builtins
module, 137
byte, 22
bytearray, 23
bytecode, 29
bytes, 22
built-in function, 36
bytes literal, 10

C

C, 11
langage, 92

language, 20, 21, 27
cadre
exécution, 130
call
class object, 28
function, 24
instance, 48
callable
object, 24
caractère, 87
case
match, 120
mot-clé, **120**
C-contiguous, 158
chaînage
comparaisons, 92
chaîne
élément, 87
objet, 86, 87
chaîne de caractères
conversion, 99
chaîne de documentation, 130
chaîne de documentation (*docstring*), **159**
chaîne entre triple guillemets, **168**
chaîner
exception, 105
character, 22
chargeur, **163**
chemin des importations, **162**
chercheur, **160**
chercheur basé sur les chemins, **166**
chercheur dans les méta-chemins, **164**
chercheur de chemins, **166**
chr
built-in function, 22
cible, 100
contrôle de boucle, 106
effacement, 104
liste, 100, 114
liste affectation, 100
liste, effacement, 104
class
attribute, 28
attribute assignment, 28
body, 44
constructor, 35
instance, 29
object, 28
class instance
attribute, 29
attribute assignment, 29
object, 28, 29
class object
call, 28
classe, **157**
définition, 104, 130
instruction, 130
nom, 130
objet, 89, 130
classe mère abstraite, **155**
clause, 113
clé, 80
clear() (*méthode frame*), 33
close() (*méthode coroutine*), 55
close() (*méthode generator*), 83
co_argcount (*attribut codeobject*), 30
co_argcount (*code object attribute*), 29
co_cellvars (*attribut codeobject*), 30
co_cellvars (*code object attribute*), 29
co_code (*attribut codeobject*), 30
co_code (*code object attribute*), 29
co_consts (*attribut codeobject*), 30
co_consts (*code object attribute*), 29
co_filename (*attribut codeobject*), 30
co_filename (*code object attribute*), 29
co_firstlineno (*attribut codeobject*), 30
co_firstlineno (*code object attribute*), 29
co_flags (*attribut codeobject*), 30
co_flags (*code object attribute*), 29
co_freevars (*attribut codeobject*), 30
co_freevars (*code object attribute*), 29
co_kwonlyargcount (*attribut codeobject*), 30
co_kwonlyargcount (*code object attribute*), 29
co_lines() (*méthode codeobject*), 31
co_lnotab (*attribut codeobject*), 30
co_lnotab (*code object attribute*), 29
co_name (*attribut codeobject*), 30
co_name (*code object attribute*), 29
co_names (*attribut codeobject*), 30
co_names (*code object attribute*), 29
co_nlocals (*attribut codeobject*), 30
co_nlocals (*code object attribute*), 29
co_positions() (*méthode codeobject*), 31
co_posonlyargcount (*attribut codeobject*), 30
co_posonlyargcount (*code object attribute*), 29
co_qualname (*attribut codeobject*), 30
co_qualname (*code object attribute*), 29
co_stacksize (*attribut codeobject*), 30
co_stacksize (*code object attribute*), 29
co_varnames (*attribut codeobject*), 30
co_varnames (*code object attribute*), 29
code
block, 57
code intermédiaire (*bytecode*), **157**
code object, 29
collections
module, 23
comment, 6
comparaison, 92
comparaisons
chaînage, 92
comparisons, 36
compiler
fonction native, 110
complex
built-in function, 51

    number, 22
    object, 22
complex literal, 15
composé
    instruction, 113
compréhensions, 79
    dictionnaire, 80
    ensemble, 80
    liste, 79
conditionnelle
    expression, 95, 96
constant, 10
constructor
    class, 35
container, 20, 28
context manager, 52
contigu, **158**
continue
    instruction, **107**, 114, 115, 117, 118
contrôle de boucle
    cible, 106
conversion
    arithmétique, 77
    chaîne de caractères, 99
    string, 36
coroutine, 54, 82, **158**
    function, 27
couple clé-valeur, 80
CPython, **158**

D

data, 19
    type, 20
dbm.gnu
    module, 24
dbm.ndbm
    module, 24
débogage
    assertions, 103
décalage
    opération, 91
decimal literal, 15
décorateur, **158**
découpage, 87
    affectation, 101
DEDENT token, 7
def
    instruction, 128
défini par l'utilisateur
    fonction appel, 89
définition
    classe, 104, 130
    fonction, 104, 128
del
    instruction, **104**
    statement, 35
delimiters, 17
dépaquetage
    dans les appels de fonction, 88
    dictionnaire, 80
    itérable, 96
descripteur, **158**
destructeur, 100
destructor, 35
dictionary
    object, 24, 28, 37
dictionnaire, **159**
    agencement, 80
    compréhensions, 80
    objet, 80, 86, 101
dictionnaire en compréhension (*ou dictionnaire en intension*), **159**
distribution simple, **168**
division, 90
division entière, **160**
divmod
    built-in function, 50
documentation string, 31

E

e
    in numeric literal, 16
EAFP, **159**
écrire
    valeurs, 99
effacement
    attribut, 104
    cible, 104
    cible liste, 104
élément
    chaîne, 87
    séquence, 86
elif
    mot-clé, 114
Ellipse
    object, 21
else
    expression conditionnelle, 96
    fantôme, 114
    mot-clé, 106, 114, 115, 117
empty
    tuple, 22
encodage du système de fichiers et gestionnaire d'erreurs associé, **159**
encodage régional, **163**
encodages de texte, **168**
encoding declarations (*source file*), 6
ensemble
    agencement, 80
    compréhensions, 80
    objet, 80
ensemble en compréhension (*ou ensemble en intension*), **168**
entrée, 138
entrée de chemin, **166**

entrée standard, 137
environment, 58
environnement virtuel, **169**
error handling, 61
errors, 61
escape sequence, 11
espace de nommage, **164**
eval
    fonction native, 110, 138
évaluation
    ordre, 97
exc_info (*in module sys*), 33
except
    mot-clé, 115
except_star
    mot-clé, 116
exception, 61, 105
    AssertionError, 103
    AttributeError, 86
    chaîner, 105
    GeneratorExit, 83, 85
    handler, 33
    ImportError, 107
    lever, 105
    NameError, 78
    StopAsyncIteration, 85
    StopIteration, 83, 104
    TypeError, 90
    ValueError, 91
    ZeroDivisionError, 90
exception handler, 61
exclusif
    ou, 92
exec
    fonction native, 110
execution
    frame, 57
    restricted, 60
    stack, 33
exécution
    cadre, 130
execution model, 57
expression, 77, **159**
    conditionnelle, 95, 96
    générateur, 81
    instruction, 99
    lambda, 96, 129
    liste, 96, 99
    yield, 81
expression d'affectation, 95
expression génératrice, **161**
expression nommée, 95
extension
    module, 20

F

f'
    formatted string literal, 11
f"
    formatted string literal, 11
f-string, **159**
f_back (*attribut frame*), 32
f_back (*frame attribute*), 32
f_builtins (*attribut frame*), 32
f_builtins (*frame attribute*), 32
f_code (*attribut frame*), 32
f_code (*frame attribute*), 32
f_globals (*attribut frame*), 32
f_globals (*frame attribute*), 32
f_lasti (*attribut frame*), 32
f_lasti (*frame attribute*), 32
f_lineno (*attribut frame*), 32
f_lineno (*frame attribute*), 32
f_locals (*attribut frame*), 32
f_locals (*frame attribute*), 32
f_trace (*attribut frame*), 32
f_trace (*frame attribute*), 32
f_trace_lines (*attribut frame*), 32
f_trace_lines (*frame attribute*), 32
f_trace_opcodes (*attribut frame*), 32
f_trace_opcodes (*frame attribute*), 32
False, 21
fantôme
    else, 114
fichier binaire, **157**
fichier texte, **168**
filtrage par motif, **120**
finale
    virgule, 97
finalizer, 35
finally
    mot-clé, 104, 106, 107, 115, 118
find_spec
    finder, 66
finder, 66
    find_spec, 66
float
    built-in function, 51
floating point
    number, 21
    object, 21
floating point literal, 15
fonction, **160**
    annotations, 129
    anonyme, 96
    appel, 89
    appel, défini par l'utilisateur, 89
    définition, 104, 128
    générateur, 81, 104
    nom, 128
    objet, 89, 128
fonction clé, **162**
fonction coroutine, **158**
fonction de rappel (*callback*), **157**
fonction définie par l'utilisateur
    objet, 89, 128

fonction générique, **161**
fonction native
    appel, 89
    compiler, 110
    eval, 110, 138
    exec, 110
    objet, 89
    range, 115
    repr, 99
for
    dans les compréhensions, 79
    instruction, 106, 107, **114**
format() (*built-in function*)
    __str__() (*object method*), 36
formatted string literal, 13
forme
    lambda, 96
forme entre parenthèses, 78
Fortran contiguous, 158
frame
    execution, 57
    object, 32
free
    variable, 58
from
    expression *yield from*, 82
    import statement, 57
    instruction *import*, 107
    mot-clé, 81, 107
frozenset
    object, 23
fstring, 13
f-string, 13
function
    argument, 24
    call, 24
    object, 24, 27
    user-defined, 24
future
    instruction, 108

G

garbage collection, 19
garde, **121**
générateur, **160**
    expression, 81
    fonction, 81, 104
    itérateur, 104
    objet, 81, 82
générateur asynchrone, **156**
    objet, 85
generator
    function, 26
    iterator, 26
    object, 30
GeneratorExit
    exception, 83, 85
generic
    special attribute, 20
gestionnaire de contexte, **158**
gestionnaire de contexte asynchrone, **156**
GIL, **161**
global
    instruction, 104, **109**
    namespace, 24
    nom liaison, 109
grammar, 4
grouping, 7

H

hachable, 80, **161**
handle an exception, 61
handler
    exception, 33
hash
    built-in function, 37
hash character, 6
héritage, 130
hexadecimal literal, 15
hierarchy
    type, 20
hooks
    import, 66
    meta, 66
    path, 66

I

id
    built-in function, 19
identifiant, 78
identifier, 8
identité
    test, 95
identity of an object, 19
IDLE, **161**
if
    dans les compréhensions, 79
    expression conditionnelle, 96
    instruction, **114**
    mot-clé, 120
imaginary literal, 15
immuable, **161**
    objet, 78, 80
    valeur type, 78
immutable
    object, 22
immutable object, 19
immutable sequence
    object, 22
immutable types
    subclassing, 35
import
    hooks, 66
    statement, 28
import hooks, 66

import machinery, 63
importateur, **162**
importation, **162**
    instruction, **107**
ImportError
    exception, 107
in
    mot-clé, 114
    opérateur, 95
inclusif
    ou, 92
INDENT token, 7
indentation, 7
index operation, 22
indication de type, **169**
indices() (*méthode slice*), 34
instance
    appel, 89
    call, 48
    class, 29
    object, 28, 29
    objet, 89
instance de classe
    appel, 89
    objet, 89
instruction, **168**
    affectation, 100
    affectation annotée, 102
    affectation, augmentée, 102
    assert, **103**
    async def, 131
    async for, 131
    async with, 132
    boucle, 106, 107, 114
    break, **106**, 114, 115, 117, 118
    classe, 130
    composé, 113
    continue, **107**, 114, 115, 117, 118
    def, 128
    del, **104**
    expression, 99
    for, 106, 107, **114**
    future, 108
    global, 104, **109**
    if, **114**
    importation, **107**
    match, **120**
    nonlocal, 110
    pass, 103
    raise, **105**
    return, **104**, 117, 118
    simple, 99
    try, **115**
    type, 110
    while, 106, 107, **114**
    with, **118**
    yield, 104
int
    built-in function, 51
integer, 22
    object, 21
    representation, 21
integer literal, 15
interactif, **162**
internal type, 29
interpolated string literal, 13
interprété, **162**
interpréteur, 137
inversion, 90
invocation, 24
io
    module, 29
is
    opérateur, 95
is not
    opérateur, 95
item selection, 22
itérable, **162**
    dépaquetage, 96
itérable asynchrone, **156**
itérateur, **162**
itérateur asynchrone, **156**
itérateur de générateur, **160**
itérateur de générateur asynchrone, **156**

J

j
    in numeric literal, 16
Java
    language, 21

K

keyword, 9

L

lambda, **163**
    expression, 96, 129
    forme, 96
langage
    C, 92
language
    C, 20, 21, 27
    Java, 21
last_traceback (*in module sys*), 33
LBYL, **163**
Le zen de Python, **169**
leading whitespace, 7
len
    built-in function, 22, 23, 48
lever
    exception, 105
lexème DEDENT, 114
lexème NEWLINE, 114
lexical analysis, 5
lexical definitions, 4

liaison
    global nom, 109
    nom, 100, 107, 128, 130
ligne de commande, 137
line continuation, 6
line joining, 5, 6
line structure, 5
list
    object, 23
liste, **163**
    affectation, cible, 100
    agencement, 79
    cible, 100, 114
    compréhensions, 79
    effacement cible, 104
    expression, 96, 99
    objet, 79, 86, 87, 101
    vide, 79
liste en compréhension (*ou liste en intension*), **163**
literal, 10
littéral, 78
loader, 66
logical line, 5

M

machine virtuelle, **169**
magic
    méthode, 163
makefile() (*socket method*), 29
mapping
    object, 23, 29
match
    case, 120
    instruction, **120**
meta
    hooks, 66
meta hooks, 66
metaclass, 43
metaclass hint, 44
métaclasse, **164**
method
    built-in, 27
    object, 25, 27
    user-defined, 25
méthode, **164**
    appel, 89
    magic, 163
    objet, 89
    special, 168
méthode magique, **163**
méthode native
    appel, 89
    objet, 89
méthode spéciale, **168**
mode interactif, 137
module, **164**
    __main__, 58, 137
    array, 23
    builtins, 137
    collections, 23
    dbm.gnu, 24
    dbm.ndbm, 24
    extension, 20
    importer, 107
    io, 29
    namespace, 28
    object, 28
    objet, 86
    sys, 116, 137
module d'extension, **159**
module spec, 66
modulo, 91
moins, 90
mot-clé
    as, 107, 115, 118, 120
    async, 131
    await, 89, 131
    case, **120**
    elif, 114
    else, 106, 114, 115, 117
    except, 115
    except_star, 116
    finally, 104, 106, 107, 115, 118
    from, 81, 107
    if, 120
    in, 114
    yield, 81
motif AS, motif OR, motif de capture, motif attrape-tout, 122
MRO, **164**
multiplication, 90
multiplication matricielle, 90
mutable, **164**
    object, 23
    objet, 100, 101
mutable object, 19
mutable sequence
    object, 23

N

n-uplet nommé, **164**
name, 8, 57
    binding, 57
NameError
    exception, 78
NameError (*built-in exception*), 58
namespace, 57
    global, 24
    module, 28
    package, 65
négation, 90
NEWLINE token, 5
nom, 78
    classe, 130

fonction, 128
liaison, 100, 107, 128, 130
liaison, global, 109
redéfinir une liaison, 100
suppression de liaison, 104
transformation, 78
nom qualifié, **167**
nombre complexe, **157**
nombre de références, **167**
noms
privé, 78
None
objet, 99
nonlocal
instruction, 110
not
opérateur, 95
not in
opérateur, 95
notation, 4
NotImplemented
object, 20
nouvelle classe, **165**
null
opération, 103
number, 15
complex, 22
floating point, 21
numeric
object, 21, 29
numeric literal, 15

O

object, 19
``None``, 20
Boolean, 21
built-in function, 27
built-in method, 27
callable, 24
class, 28
class instance, 28, 29
code, 29
complex, 22
dictionary, 24, 28, 37
Ellipse, 21
floating point, 21
frame, 32
frozenset, 23
function, 24, 27
generator, 30
immutable, 22
immutable sequence, 22
instance, 28, 29
integer, 21
list, 23
mapping, 23, 29
method, 25, 27
module, 28
mutable, 23
mutable sequence, 23
NotImplemented, 20
numeric, 21, 29
sequence, 22, 29
set, 23
set type, 23
slice, 48
traceback, 33
tuple, 22
user-defined function, 24
user-defined method, 25
object.__match_args__ (*variable de base*), 52
object.__slots__ (*variable de base*), 41
objet, **165**
n-uplet, 86, 87, 96
appelable, 87
chaîne, 86, 87
classe, 89, 130
dictionnaire, 80, 86, 101
ensemble, 80
fonction, 89, 128
fonction définie par l'utilisateur, 89, 128
fonction native, 89
générateur, 81, 82
générateur asynchrone, 85
immuable, 78, 80
instance, 89
instance de classe, 89
liste, 79, 86, 87, 101
méthode, 89
méthode native, 89
module, 86
mutable, 100, 101
None, 99
séquence, 86, 87, 95, 101, 114
tableau de correspondances, 86, 101
trace d'appels, 105, 116
objet classe
appel, 89
objet fichier, **159**
objet fichier-compatible, **159**
objet octet-compatible, **157**
objet simili-chemin, **166**
octal literal, 15
open
built-in function, 29
opérateur
- (*moins*), 90, 91
% (*pourcentage*), 91
& (*esperluette*), 92
* (*astérisque*), 90
**, 89
+ (*plus*), 90, 91
/ (*barre oblique*), 90
//, 90
< (*plus petit*), 92

    <<, 91
    <=, 92
    !=, 92
    ==, 92
    > (*plus grand*), 92
    >=, 92
    >>, 91
    @ (*arobase*), 90
    ^ (*caret*), 92
    | (*barre verticale*), 92
    ~ (*tilde*), 90
    and, 95
    in, 95
    is, 95
    is not, 95
    not, 95
    not in, 95
    ou, 95
    précédence des opérateurs, 97
    ternaire, 96
opérateur morse, 95
opération
    binaire arithmétique, 90
    binaire OU (*bit à bit*), 92
    booléenne, 95
    décalage, 91
    null, 103
    puissance, 89
    unaire arithmétique, 90
    unaire OU (*bit à bit*), 90
operator
    overloading, 34
operators, 16
ord
    built-in function, 22
ordre
    évaluation, 97
ordre de résolution des méthodes, **164**
ou
    exclusif, 92
    inclusif, 92
    opérateur, 95
    OU (*bit à bit*), 92
OU (*bit à bit*)
    and, 92
    opération, binaire, 92
    opération, unaire, 90
    ou, 92
    xor, 92
overloading
    operator, 34

P

package, 64
    namespace, 65
    portion, 65
    regular, 64
paquet, **165**
paquet classique, **167**
paquet provisoire, **166**
paquet-espace de nommage, **165**
par défaut
    paramètre valeur, 128
paramètre, **165**
    définition de fonction, 127
    sémantique des appels, 88
    valeur, par défaut, 128
parser, 5
pass
    instruction, 103
path
    hooks, 66
path based finder, 72
path hooks, 66
PEP, **166**
physical line, 5, 6, 11
plus, 90
point d'entrée pour la recherche
        dans path, **166**
popen() (*in module os*), 29
portée imbriquée, **165**
portion, **166**
    package, 65
pow
    built-in function, 50
précédence des opérateurs
    opérateur, 97
primaire, 86
print
    built-in function, 36
print() (*built-in function*)
    __str__() (*object method*), 36
privé
    noms, 78
procédure
    appel, 99
programme, 137
puissance
    opération, 89
pyc utilisant le hachage, **161**
Python 3000, **166**
Python Enhancement Proposals
    PEP 1, 166
    PEP 8, 93
    PEP 236, 109
    PEP 252, 40
    PEP 255, 82
    PEP 278, 169
    PEP 302, 63, 76, 160, 163
    PEP 308, 96
    PEP 318, 130, 131
    PEP 328, 76, 160
    PEP 338, 76
    PEP 342, 82
    PEP 343, 46, 52, 120, 158
    PEP 362, 156, 165

PEP 366, 70, 76
PEP 380, 82
PEP 411, 166
PEP 414, 11
PEP 420, 63, 65, 71, 76, 160, 166
PEP 421, 165
PEP 443, 161
PEP 448, 80, 89, 97
PEP 451, 76, 160
PEP 483, 161
PEP 484, 103, 129, 155, 160, 161, 169
PEP 492, 54, 82, 133, 156, 158
PEP 498, 15, 159
PEP 519, 166
PEP 525, 82, 156
PEP 526, 103, 129, 155, 169
PEP 530, 79
PEP 560, 43, 47
PEP 562, 39
PEP 563, 109, 129
PEP 570, 129
PEP 572, 80, 96, 123
PEP 585, 161
PEP 614, 128, 130
PEP 617, 139
PEP 626, 31
PEP 634, 52, 120, 127
PEP 636, 120, 127
PEP 649, 59
PEP 688, 53
PEP 695, 59, 111
PEP 3104, 110
PEP 3107, 129
PEP 3115, 44, 131
PEP 3116, 169
PEP 3119, 45
PEP 3120, 5
PEP 3129, 130, 131
PEP 3131, 8
PEP 3132, 101
PEP 3135, 45
PEP 3147, 71
PEP 3155, 167
PYTHONHASHSEED, 38
Pythonique, **166**
PYTHONNODEBUGRANGES, 31
PYTHONPATH, 73

R

r'
    raw string literal, 11
r"
    raw string literal, 11
raise
    instruction, **105**
raise an exception, 61
ramasse-miettes, **160**
range
    fonction native, 115
raw string, 11
redéfinir une liaison
    nom, 100
référence
    attribut, 86
reference counting, 19
référence empruntée, **157**
référence forte, **168**
regular
    package, 64
relative
    importation, 108
replace() (*méthode codeobject*), 31
repr
    fonction native, 99
repr() (*built-in function*)
    __repr__() (*object method*), 36
representation
    integer, 21
reserved word, 9
restricted
    execution, 60
retours à la ligne universels, **169**
return
    instruction, **104**, 117, 118
round
    built-in function, 52

S

scope, 57, 58
sélection (*ou indiçage*), 86
    affectation, 101
send() (*méthode coroutine*), 55
send() (*méthode generator*), 83
sequence
    object, 22, 29
séquence, **167**
    élément, 86
    objet, 86, 87, 95, 101, 114
set
    object, 23
set type
    object, 23
simple
    instruction, 99
singleton
    tuple, 22
slice
    built-in function, 34
    object, 48
slicing, 22, 23
soft keyword, 9
sortie, 99
    standard, 99
source character set, 6
soustraction, 91
space, 7

special
    attribute, 20
    attribute, generic, 20
    méthode, 168
spécificateur de module, **164**
stack
    execution, 33
    trace, 33
standard
    sortie, 99
Standard C, 11
start (*attribut d'objet tranche*), 87
start (*slice object attribute*), 34
statement
    assignment, 23
    del, 35
    import, 28
    try, 33
    with, 52
statement grouping, 7
static type checker, **168**
stderr (*in module sys*), 29
stdin (*in module sys*), 29
stdio, 29
stdout (*in module sys*), 29
step (*attribut d'objet tranche*), 87
step (*slice object attribute*), 34
stop (*attribut d'objet tranche*), 87
stop (*slice object attribute*), 34
StopAsyncIteration
    exception, 85
StopIteration
    exception, 83, 104
string
    __format__() (*object method*), 36
    __str__() (*object method*), 36
    conversion, 36
    formatted literal, 13
    immutable sequences, 22
    interpolated literal, 13
string literal, 10
subclassing
    immutable types, 35
subscription, 22, 23
suite, 113
suppression de liaison
    nom, 104
syntax, 4
sys
    module, 116, 137
sys.exc_info, 33
sys.exception, 33
sys.last_traceback, 33
sys.meta_path, 66
sys.modules, 65
sys.path, 73
sys.path_hooks, 73
sys.path_importer_cache, 73
sys.stderr, 29
sys.stdin, 29
sys.stdout, 29
SystemExit (*built-in exception*), 61

T

tab, 7
tableau de correspondances
    objet, 86, 101
tableau de correspondances (*mapping en anglais*), **163**
tb_frame (*attribut traceback*), 33
tb_frame (*traceback attribute*), 33
tb_lasti (*attribut traceback*), 33
tb_lasti (*traceback attribute*), 33
tb_lineno (*attribut traceback*), 33
tb_lineno (*traceback attribute*), 33
tb_next (*attribut traceback*), 33
tb_next (*traceback attribute*), 33
termination model, 61
ternaire
    opérateur, 96
test
    appartenance, 95
    identité, 95
throw() (*méthode coroutine*), 55
throw() (*méthode generator*), 83
token, 5
trace
    stack, 33
trace d'appels
    objet, 105, 116
traceback
    object, 33
tranche, 87, **168**
transformation
    nom, 78
triple-quoted string, 11
True, 21
try
    instruction, **115**
    statement, 33
tuple
    empty, 22
    object, 22
    singleton, 22
typage canard (*duck-typing*), **159**
type, 20, **168**
    built-in function, 19, 43
    data, 20
    hierarchy, 20
    immuable valeur, 78
    instruction, 110
type générique, **161**
type of an object, 19
type parameters, 133
TypeError
    exception, 90

types, internal, 29

U

u'
    string literal, 10
u"
    string literal, 10
unaire
    arithmétique opération, 90
    OU (*bit à bit*) opération, 90
UnboundLocalError, 58
Unicode, 22
Unicode Consortium, 11
UNIX, 137
unreachable object, 19
unrecognized escape sequence, 12
user-defined
    function, 24
    method, 25
user-defined function
    object, 24
user-defined method
    object, 25

V

valeur, 80
    par défaut paramètre, 128
    type, immuable, 78
valeurs
    écrire, 99
value of an object, 19
ValueError
    exception, 91
variable
    free, 58
variable de classe, **157**
variable de contexte, **158**
variable d'environnement
    PYTHONHASHSEED, 38
    PYTHONNODEBUGRANGES, 31
    PYTHONPATH, 73
verrou global de l'interpréteur, **161**
vide
    n-uplet, 78
    liste, 79
virgule, 78
    finale, 97
vue de dictionnaire, **159**

W

while
    instruction, 106, 107, **114**
Windows, 137
with
    instruction, **118**
    statement, 52

X

xor
    OU (*bit à bit*), 92

Y

yield
    exemples, 83
    expression, 81
    instruction, 104
    mot-clé, 81

Z

ZeroDivisionError
    exception, 90