The Python Language Reference

Version 3.7.4

Guido van Rossum
and the Python development team

septembre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

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

CHAPITRE 1

Introduction

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

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

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

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

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

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

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

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

1.2 Notations

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

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

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

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

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

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

CHAPITRE 2

Analyse lexicale

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

Python 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.

Si aucune déclaration d'encodage n'est trouvée, l'encodage par défaut est utilisé : UTF-8. En plus, si les premiers octets du fichier sont l'indicateur d'ordre des octets UTF-8 (`b'\xef\xbb\xbf'`, *BOM* en anglais pour *byte order mark*), le fichier est considéré comme étant en UTF-8 (cette convention est reconnue, entre autres, par **`notepad`** de Microsoft).

Si un encodage est déclaré, le nom de l'encodage doit être reconnu par Python. 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 unde
id_continue    ::=   <all characters in id_start, plus characters in the categories Mn, Mc,
xid_start      ::=   <all characters in id_start whose NFKC normalization is in "id_start x
xid_continue   ::=   <all characters in id_continue whose NFKC normalization is in "id_cont
```

Les codes de catégories Unicode cités ci-dessus signifient :

- *Lu* – lettre 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* – liste explicite des caractères de PropList.txt pour la compatibilité descendante
- *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.

Un fichier HTML, ne faisant pas référence, listant tous les caractères valides pour Unicode 4.1 se trouve à https://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html.

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 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 :

_* L'identifiant spécial _ n'est pas importé par `from module import *`. Il est utilisé dans l'interpréteur interactif pour stocker le résultat de la dernière évaluation ; il est stocké dans le module `builtins`. Lorsque vous

n'êtes pas en mode interactif, _ n'a pas de signification particulière et n'est pas défini. Voir la section *L'instruction import*.

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

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

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

2.4 Littéraux

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

2.4.1 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*.

Description en français : les deux types de littéraux peuvent être encadrés par une paire de guillemets simples (') ou doubles ("). Ils peuvent aussi être encadrés par une paire de trois guillemets simples ou une paire de trois guillemets doubles (on appelle alors généralement ces chaînes *entre triples guillemets*). La barre oblique inversée peut être utilisée pour échapper des caractères qui auraient sinon une signification spéciale, tels que le retour à la ligne, la barre oblique inversée elle-même ou le guillemet utilisé pour délimiter la chaîne.

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.

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

Nouveau dans la version 3.3 : le support du préfixe historique pour les chaînes Unicode a été réintroduit 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 *Chaînes de caractères formatées littérales*. 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 ").

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	
`\\`	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	(1,3)
`\xhh`	caractère dont le code est *ooo* en hexadécimal	(2,3)

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	(4)
`\uxxxx`	caractère dont le code est *xxxx* en hexadécimal	(5)
`\Uxxxxxxxx`	caractère dont le code est *xxxxxxxx* en hexadécimal sur 32 bits	(6)

Notes :

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

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

(3) 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.

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

(5) Exactement quatre chiffres hexadécimaux sont requis.

(6) 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 : Les séquences d'échappement non reconnues produisent un avertissement DeprecationWarning. Dans les futures versions de Python, elles génèreront une erreur de syntaxe.

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.

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 Chaînes de caractères formatées littérales

Nouveau dans la version 3.6.

Une *chaine de caractères formatée littérale* 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.

1. http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt

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 | NULL | 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. Après l'expression, il peut y avoir un champ de conversion, introduit par un point d'exclamation `'!'`. La spécification de format peut aussi être ajouté, introduite par le caractère deux-points `':'`. Le champ à remplacer se termine par une accolade fermante `'}'`.

Les expressions dans les chaînes de caractères formatées littérales sont traitées comme des expressions Python normales entourées de parenthèses, à quelques exceptions près. Une expression vide n'est pas autorisée et une expression *lambda* doit être explicitement entourée de parenthèses. Les expressions de remplacement peuvent contenir des sauts de ligne (par exemple dans les chaînes de caractères entre triples guillemets) mais elles ne peuvent pas contenir de commentaire. Chaque expression est évaluée dans le contexte où la chaîne de caractères formatée apparaît, de gauche à droite.

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()`.

Le résultat est ensuite formaté en utilisant le protocole de `format()`. La spécification du format est passée à la méthode *__format__()* de l'expression ou du résultat de la conversion. Une chaîne vide est passée lorsque la spécification de format est omise. Le résultat formaté est alors inclus dans la valeur finale de la chaîne complète.

Les spécifications de format peuvent inclure des champs de remplacement imbriqués. Ces champs imbriqués peuvent inclure leurs propres champs de conversion et spécifications de format mais l'imbrication ne doit pas aller plus profond. Le mini-langage de spécification de format est le même que celui utilisé par la méthode `.format()` du type `str`.

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'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
```

Une conséquence de partager la même syntaxe avec les chaînes littérales normales est que les caractères dans les champs à remplacer ne doivent pas entrer en conflit avec le guillemet utilisé pour encadrer la chaine formatée littérale :

```
f"abc {a["x"]} def"    # error: outer string literal ended prematurely
f"abc {a['x']} def"    # workaround: use different quoting
```

La barre oblique inversée n'est pas autorisée dans les expressions des champs à remplacer et son utilisation génère une erreur :

```
f"newline: {ord('\n')}"  # raises SyntaxError
```

Pour inclure une valeur où l'échappement par barre oblique inversée est nécessaire, vous devez créer une variable temporaire.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```

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 soulignés 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 soulignés 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 soulignés 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'assignation 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).

Chaque objet possède un identifiant, un type et une valeur. *L'identifiant* d'un objet ne change jamais après sa création ; vous pouvez vous le représenter comme l'adresse de l'objet en mémoire. L'opérateur `is` compare les identifiants de deux objets ; la fonction `id()` renvoie un entier représentant cet identifiant.

CPython implementation detail : 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 *muables* (*mutable* en anglais) ; 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 muable peut varier lorsque la valeur de l'objet muable 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 tuples sont immuables alors que les dictionnaires et les listes sont muables.

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

CPython implementation detail : CPython 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

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.

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).

Notez que si vous utilisez les fonctionnalités de débogage ou de trace de l'implémentation, il est possible que des références qui seraient normalement supprimées soient toujours présentes. Notez aussi que capturer une exception avec l'instruction *`try`*...*`except`* peut conserver des objets en vie.

Certains objets font référence à des ressources "externes" telles que des fichiers ouverts ou des fenêtres. Ces objets libèrent ces ressources au moment où ils sont supprimés, mais comme le ramasse-miettes ne garantit pas qu'il supprime tous les objets, ces objets fournissent également un moyen explicite de libérer la ressource externe, généralement sous la forme d'une méthode `close()`. Nous incitons fortement les programmeurs à fermer explicitement de tels objets. Les instructions *`try`*...*`finally`* et *`with`* sont très pratiques pour cela.

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

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

3.2 Hiérarchie des types standards

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.

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

NotImplemented 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 `NotImplemented`. Les méthodes numériques et les comparaisons riches doivent renvoyer cette valeur si elles n'implémentent pas l'opération pour les opérandes fournis (l'interpréteur essaie alors l'opération en permutant les opérandes ou tout autre stratégie de contournement, en fonction de l'opérateur). Sa valeur booléenne est vraie.

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

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.

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

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

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

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.

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.

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`.

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

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

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

Les séquences se différencient en fonction de leur muabilité :

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

Les types suivants sont des séquences immuables :

Chaînes de caractères Une chaîne de caractères (*string* en anglais) est une séquence de valeurs qui représentent des caractères Unicode. Tout caractère dont le code est dans l'intervalle `U+0000 - U+10FFFF` peut être représenté dans une chaîne. Python ne possède pas de type `char` ; à la place, chaque caractère Unicode dans la chaîne est représenté par un objet chaîne de longueur `1`. La fonction native `ord()` convertit un caractère Unicode de la représentation en chaîne vers un entier dans l'intervalle `0 - 10FFFF` ; la fonction `chr()` convertit un entier de l'intervalle `0 - 10FFFF` vers l'objet chaîne de longueur `1` correspondant. `str.encode()` peut être utilisée pour convertir un objet `str` vers `bytes` selon l'encodage spécifié et `bytes.decode()` effectue l'opération inverse.

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

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

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

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

Tableaux d'octets Un objet *bytearray* est un tableau muable. Il est créé par la fonction native constructeur `bytearray()`. À part la propriété d'être muable (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`.

Le module d'extension `array` fournit un autre exemple de type de séquence muable, de même que le module `collections`.

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

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

Actuellement, il existe deux types d'ensembles natifs :

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

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

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

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

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

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

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

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

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

Attributs spéciaux :

Attribut	Signification	
`__doc__`	Chaîne de documentation de la fonction ou `None` s'il n'en existe pas ; n'est pas héritée par les sous-classes	Accessible en écriture
`__name__`	Nom de la fonction	Accessible en écriture
`__qualname__`	*qualified name* de la fonction Nouveau dans la version 3.3.	Accessible en écriture
`__module__`	Nom du module où la fonction est définie ou `None` si ce nom n'est pas disponible.	Accessible en écriture
`__defaults__`	Tuple contenant les valeurs des arguments par défaut pour ceux qui en sont dotés ou `None` si aucun argument n'a de valeur par défaut.	Accessible en écriture
`__code__`	Objet code représentant le corps de la fonction compilée.	Accessible en écriture
`__globals__`	Référence pointant vers le dictionnaire contenant les variables globales de la fonction — l'espace de noms global du module dans lequel la fonction est définie.	Accessible en lecture seule
`__dict__`	Espace de nommage accueillant les attributs de la fonction.	Accessible en écriture
`__closure__`	`None` ou tuple de cellules qui contient un lien pour chaque variable libre de la fonction. Voir ci-dessous pour les informations relatives à l'attribut `cell_contents`.	Accessible en lecture seule
`__annotations__`	Dictionnaire contenant les annotations des paramètres. Les clés du dictionnaire sont les noms des paramètres et la clé `"return"` est utilisée pour les annotations de la valeur renvoyée. Les entrées du dictionnaire ne sont présentes que si les paramètres sont effectivement annotés.	Accessible en écriture
`__kwdefaults__`	Dictionnaire contenant les valeurs par défaut pour les paramètres passés par mot-clé.	Accessible en écriture

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

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

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.

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

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).

Attributs spéciaux en lecture seule : `__self__` est l'objet instance de classe, `__func__` est l'objet fonction ;

__doc__ est la documentation de la méthode (comme __func__.__doc__) ; __name__ est le nom de la méthode (comme __func__.__name__) ; __module__ est le nom du module où la méthode est définie ou None s'il n'est pas disponible.

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

Les objets méthodes définies par l'utilisateur peuvent être créés quand vous récupérez un attribut de classe (par exemple *via* une instance de cette classe) si cet attribut est un objet fonction définie par l'utilisateur ou un objet méthode de classe.

Quand un objet méthode d'instance est créé à partir d'un objet fonction définie par l'utilisateur *via* une des instances, son attribut __self__ est l'instance et l'objet méthode est réputé lié. Le nouvel attribut de la méthode __func__ est l'objet fonction original.

Quand un objet méthode définie par l'utilisateur est créé à partir d'un autre objet méthode de la classe ou de l'instance, son comportement est identique à l'objet fonction sauf pour l'attribut __func__ de la nouvelle instance qui n'est pas l'objet méthode original mais son attribut __func__.

Quand un objet méthode d'instance est créé à partir d'un autre objet méthode de la classe ou de l'instance, son attribut __self__ est la classe elle-même et son attribut __func__ est l'objet fonction sous-jacent la méthode de classe.

Quand un objet méthode d'instance est appelé, la fonction sous-jacente (__func__) est appelée et l'objet instance de la classe (__self__) est inséré en tête de liste des arguments. Par exemple, si C est une classe qui contient la définition d'une fonction f() et que x est une instance de C, alors appeler x.f(1) est équivalent à appeler C.f(x, 1).

Quand un objet méthode d'instance est dérivé à partir d'un objet méthode de classe, l'instance de classe stockée dans __self__ est en fait la classe elle-même. Ainsi, appeler x.f(1) ou C.f(1) est équivalent à appeler f(C, 1) où f est la fonction sous-jacente.

Notez que la transformation d'objet fonction en objet méthode d'instance se produit à chaque fois que l'attribut est récupéré à partir de l'instance. Dans certains cas, assigner l'attribut à une variable locale et appeler cette variable locale constitue une bonne optimisation. Notez aussi que cette transformation n'a lieu que pour les fonctions définies par l'utilisateur : les autres objets appelables (et les objets non appelables) sont récupérés sans transformation. Il est aussi important de noter que les fonctions définies par l'utilisateur qui sont attributs d'une instance de classe ne sont pas converties en méthodes liées ; ceci n'a lieu que pour les fonctions qui sont attributs de la classe.

Fonctions générateurs Une fonction ou une méthode qui utilise l'instruction *yield* (voir la section *L'instruction yield*) est appelée *fonction générateur*. Une telle fonction, lorsqu'elle est appelée, retourne 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 générateurs asynchrones Une fonction ou une méthode définie avec *async def* et qui utilise l'instruction *yield* est appelée *fonction générateur 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 Un objet fonction native est une enveloppe autour d'une fonction C. Nous pouvons citer len() et math.sin() (math est un module standard natif) comme fonctions natives. Le nombre et le type des arguments sont déterminés par la fonction C. Des attributs spéciaux en lecture seule existent : __doc__ contient la chaîne de documentation de la fonction (ou None s'il n'y en a pas) ; __name__ est le nom de la

fonction ; __self__ est défini à None ; __module__ est le nom du module où la fonction est définie ou None s'il n'est pas disponible.

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

Classes Les classes sont des appelables. Ces objets sont normalement utilisés pour créer des instances d'elles-mêmes mais des variations sont possibles pour les types de classes qui surchargent *__new__()*. Les arguments de l'appel sont passés à *__new__()* et, dans le cas classique, *__new__()* initialise une nouvelle instance.

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

Modules Les modules constituent l'organisation de base du code Python et sont créés par le *mécanisme d'import* soit avec l'instruction *import*, soit en appelant des fonctions telles que importlib.import_module() ou la fonction native __import__(). Un objet module possède un espace de nommage implémenté par un objet dictionnaire (c'est le dictionnaire référencé par l'attribut __globals__ des fonctions définies dans le module). Les références à un attribut sont traduites en recherches dans ce dictionnaire, par exemple m.x est équivalent à m.__dict__["x"]. Un objet module ne contient pas l'objet code utilisé pour initialiser le module (puisque celui-ci n'est plus nécessaire une fois l'initialisation terminée).

L'assignation 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 (en lecture-écriture) : *__name__* est le nom du module ; __doc__ est la chaîne de documentation du module (ou None s'il n'y en a pas) ; __annotations__ (optionnel) est un dictionnaire contenant les *annotations de variables* collectées durant l'exécution du corps du module ; *__file__* est le 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.

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

CPython implementation detail : 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.

Classes déclarées par le développeur Le type d'une classe déclarée par le développeur est créé au moment de la définition de la classe (voir la section *Définition de classes*). Une classe possède un espace de nommage implémenté sous la forme d'un objet dictionnaire. Les références vers les attributs de la classe sont traduits en recherches dans ce dictionnaire, par exemple C.x est traduit en C.__dict__["x"] (bien qu'il existe un certain nombre de fonctions automatiques qui permettent de trouver des attributs par d'autres moyens). Si le nom d'attribut n'est pas trouvé dans ce dictionnaire, la recherche continue dans les classes de base. Les classes de base sont trouvées en utilisant l'ordre de résolution des méthodes (*method resolution order* en anglais, ou MRO) *C3* qui a un comportement cohérent même en présence d'héritages en "diamant", où différentes branches d'héritages conduisent vers un ancêtre commun. Vous trouverez plus de détails sur l'ordre de résolution des méthodes MRO *C3* utilisé par Python dans la documentation de la version 2.3 disponible sur https://www.python.org/download/releases/2.3/mro/.

Quand une référence à un attribut de classe (disons la classe C) pointe vers un objet méthode de classe, elle est transformée en objet méthode d'instance dont l'attribut __self__ est C. Quand elle pointe vers un objet méthode statique, elle est transformée en objet encapsulé par l'objet méthode statique. Reportez-vous à la section *Implémentation de descripteurs* pour une autre manière dont les attributs d'une classe diffèrent de ceux réellement contenus dans son __dict__.

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

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

Attributs spéciaux : `__name__` est le nom de la classe ; `__module__` est le nom du module dans lequel la classe est définie ; `__dict__` est le dictionnaire contenant l'espace de nommage de la classe ; `__bases__` est un tuple contenant les classes de base, dans l'ordre d'apparition dans la liste des classes de base ; `__doc__` est la chaîne de documentation de la classe (ou `None` si elle n'existe pas) ; `__annotations__` (optionnel) est un dictionnaire contenant les *annotations de variables* collectées durant l'exécution du corps de la classe.

Instances de classes Une instance de classe est créée en appelant un objet classe (voir ci-dessus). Une instance de classe possède un espace de nommage implémenté sous la forme d'un dictionnaire qui est le premier endroit où sont recherchées les références aux attributs. Quand un attribut n'est pas trouvé dans ce dictionnaire et que la classe de l'instance contient un attribut avec ce nom, la recherche continue avec les attributs de la classe. Si un attribut de classe est trouvé et que c'est un objet fonction définie par l'utilisateur, il est transformé en objet méthode d'instance dont l'attribut `__self__` est l'instance. Les objets méthodes statiques et méthodes de classe sont aussi transformés ; reportez-vous ci-dessous à "Classes". Lisez la section *Implémentation de descripteurs* pour une autre façon de récupérer les attributs d'une classe, où la récupération *via* ses instances peut différer des objets réellement stockés dans le `__dict__` de la classe. Si aucun attribut de classe n'est trouvé et que la classe de l'objet possède une méthode *`__getattr__()`*, cette méthode est appelée pour rechercher une correspondance.

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

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

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

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`.

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

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

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

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

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

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

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

Objets cadres Un objet cadre représente le cadre d'exécution. Il apparait dans des objets traces (voir plus loin) et est passé comme argument aux fonctions de traçage actives.

Attributs spéciaux en lecture seule : `f_back` pointe vers le cadre précédent (l'appelant) ou `None` si c'est le pied de la pile d'appel ; `f_code` est l'objet code en cours d'exécution dans ce cadre ; `f_locals` est le dictionnaire dans lequel sont cherchées les variables locales ; `f_globals` est utilisé pour les variables globales ; `f_builtins` est utilisé pour les noms natifs ; `f_lasti` donne l'instruction précise (c'est un indice dans la chaîne de *bytecode* de l'objet code).

Attributs spéciaux en lecture-écriture : `f_trace`, s'il n'est pas `None`, c'est une fonction appelée à différentes occasions durant l'exécution du code (elle est utilisée par le débogueur). Normalement, un événement est déclenché pour chaque ligne de code source — ce comportement peut être désactivé en définissant `f_trace_lines` à `False`.

Une implémentation *peut* autoriser le déclenchement des événements *opcode* par *opcode* en définissant `f_trace_opcodes` à `True`. Notez que cela peut conduire à un comportement erratique de l'interpréteur si des exceptions levées la fonction de traçage s'échappent vers la fonction en train d'être tracée.

`f_lineno` est le numéro de la ligne courante du cadre — écrire dedans depuis une fonction trace fait sauter à la ligne demandée (seulement pour le cadre le plus bas). Un débogueur peut implémenter une commande "sauter vers" (aussi appelée "Définir la prochaine instruction" ou *Set Next Statement* en anglais) en écrivant dans `f_lineno`.

Les objets cadres comprennent une méthode :

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

Cette méthode efface toutes les références aux variables locales conservées dans le cadre. Par ailleurs, si le cadre est celui d'un générateur, le générateur se termine. Ceci permet de casser des références cycliques qui incluent des objets cadres (par exemple, lors de la capture d'une exception et du stockage de la pile d'appels pour une utilisation future).

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

Nouveau dans la version 3.4.

Objets traces Les objets traces représentent la pile de traces d'une exception. Un objet trace est implicitement créé quand une exception apparaît et peut être explicitement créé en appelant `types.TracebackType`.

Pour les traces créées implicitement, quand l'interpréteur recherche un gestionnaire d'exception en remontant la pile d'exécution, un objet trace est inséré devant l'objet trace courant à chaque nouveau niveau. Quand il entre dans le gestionnaire d'exception, la pile d'appels est rendue accessible au programme (voir la section *L'instruction try*). Elle est accessible par le troisième élément du tuple renvoyé par `sys.exc_info()` et comme attribut `__traceback__` de l'exception qui est traitée.

Quand le programme ne contient aucun gestionnaire adéquat, la pile de traces est écrite (joliment formatée) sur la sortie d'erreur standard ; si l'interpréteur est interactif, elle est rendue disponible pour l'utilisateur en tant que `sys.last_traceback`.

Pour les traces créées explicitement, il revient au créateur de la trace de déterminer comment les attributs `tb_next` doivent être liés pour former la pile complète des traces.

Attributs spéciaux en lecture seule : `tb_frame` pointe vers le cadre d'exécution du niveau courant ; `tb_lineno` donne le numéro de ligne où l'exception a été levée ; `tb_lasti` indique l'instruction précise. Le numéro de ligne et la dernière instruction dans la trace peuvent différer du numéro de ligne de l'objet cadre si l'exception a eu lieu dans une instruction *`try`* sans qu'il n'y ait de clause `except` adéquate ou sans clause *finally*.

Attributs spéciaux en lecture-écriture `tb_next` est le niveau suivant dans la pile d'exécution (en direction du cadre où l'exception a eu lieu) ou `None` s'il n'y a pas de niveau suivant.

Modifié dans la version 3.7 : Les objets traces peuvent maintenant être explicitement instanciés depuis le code Python et l'attribut `tb_next` des instances existantes peut être mis à jour.

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 découpe décrit s'il est appliqué à une séquence de *length* éléments. Elle renvoie un tuple de trois 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 ne sont pas appelables en tant que tels, bien que les objets qu'ils encapsulent le soient souvent. Les objets méthodes statiques sont créés par le constructeur natif `staticmethod()`.

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

3.3 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).

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

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.

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*).

Une implémentation typique crée une nouvelle instance de la classe en invoquant la méthode *__new__()* de la superclasse à l'aide de `super().__new__(cls[, ...])` avec les arguments adéquats, puis modifie l'instance nouvellement créée en tant que de besoin avant de la renvoyer.

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

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

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

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

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.

CPython implementation detail : 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.

  C'est l'implémentation par défaut des appels à *object.__repr__()* du type natif `object`.

`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 *formatted string literals* 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 d'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(self), '')`.

`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)`.

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

Par défaut, *__ne__()* délègue à *__eq__()* et renvoie le résultat inverse, sauf si c'est `NotImplemented`. Il n'y a pas d'autres relations implicites pour les opérateurs de comparaison. Par exemple, `(x<y or x==y)` n'implique pas `x<=y`. Pour obtenir une relation d'ordre total automatique à partir d'une seule opération, reportez-vous à `functools.total_ordering()`.

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

Il n'y a pas de versions avec les arguments interchangés de ces méthodes (qui seraient utilisées quand l'argument de gauche ne connaît pas l'opération alors que l'argument de droite la connaît) ; en lieu et place, *__lt__()* et *__gt__()* sont la réflexion l'une de l'autre, *__le__()* et *__ge__()* sont la réflexion l'une de l'autre et *__eq__()* ainsi que *__ne__()* sont réflexives. Si les opérandes sont de types différents et que l'opérande de droite est d'un type qui une sous-classe directe ou indirecte du type de l'opérande de gauche, alors la méthode symétrique de l'opérande de droite est prioritaire, sinon c'est la méthode de l'opérande de gauche qui est prioritaire. Les sous-classes virtuelles ne sont pas prises en compte.

`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`). *__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 de la comparaison des objets, en les plaçant un tuple 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)"`.

Si une classe ne définit pas de méthode *__eq__()*, elle ne doit pas définir l'opération *__hash__()* non plus ; si elle définit *__eq__()* mais pas *__hash__()*, les instances ne peuvent pas être utilisées en tant qu'élément dans une collection de hachables. Si une classe définit des objets mutables et implémente la méthode *__eq__()*, elle ne doit pas implémenter *__hash__()* puisque l'implémentation des collections hachables requiert que les clés soient des empreintes immuables (si l'empreinte d'un objet change, il ne sera plus trouvé correctement dans le stockage du dictionnaire).

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, *bytes* et objets *datetime* 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.

C'est un comportement voulu pour se protéger contre un déni de service qui utiliserait des entrées malicieusement choisies pour effectuer des insertions dans le dictionnaire dans le pire cas, avec une complexité en O(n^2). Lisez http://www.ocert.org/advisories/ocert-2011-003.html pour en obtenir les détails (article en anglais).

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*)
: Appelée pour implémenter les tests booléens et l'opération native bool() ; elle doit renvoyer False ou True. Quand cette méthode n'est pas définie, *__len__()* est appelée, si elle est définie, et l'objet est considéré vrai si le résultat est non nul. Si une classe ne définit ni *__len__()* ni *__bool__()*, toutes ses instances sont considérées comme vraies.

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 retourner 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 : Cette méthode peut être shuntée lorsque la recherche porte sur les méthodes spéciales en tant que résultat d'une invocation implicite *via* la syntaxe du langage ou les fonctions natives. Lisez *Recherche des méthodes spéciales*.

`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)`.

`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.

`object.`**`__dir__`**(*self*)
: Appelée quand `dir()` est appelée sur l'objet. Elle doit renvoyer une séquence. `dir()` convertit la séquence renvoyée en liste et effectue le classement.

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.

La fonction `__dir__` ne prend aucun argument et renvoie une liste de chaînes qui représente les noms accessibles du module. Si elle existe, cette fonction surcharge la fonction de recherche standard `dir()` du 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 affecte 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.

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

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

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

`object.`**`__set_name__`**(*self*, *owner*, *name*)
: Appelée au moment où la classe propriétaire *owner* est créée. La classe descripteur a été assignée à *name*.

  Nouveau dans la version 3.6.

L'attribut `__objclass__` est interprété par le module `inspect` comme spécifiant la classe où cet objet a été défini (le définir correctement peut vous aider dans l'introspection des classes dynamiques à l'exécution). Pour les appelables, cela peut indiquer qu'une instance d'un certain type (ou d'une certaine sous-classe) est attendue ou requise comme premier argument positionnel (par exemple, CPython définit cet attribut pour les méthodes non liées qui sont implémentées en 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 méta-classes.

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

Pour des liaisons avec des instances, la priorité à l'invocation du descripteur dépend des méthodes que le descripteur a définies. Un descripteur peut définir n'importe quelle combinaison de *`__get__()`*, *`__set__()`* et *`__delete__()`*. S'il ne définit pas *`__get__()`*, alors accéder à l'attribut retourne l'objet descripteur lui-même sauf s'il existe une valeur

dans le dictionnaire de l'objet instance. Si le descripteur définit *__set__()* ou *__delete__()*, c'est un descripteur de données ; s'il ne définit aucune méthode, c'est un descripteur hors-donnée. Normalement, les descripteurs de données définissent à la fois *__get__()* et *__set__()*, alors que les descripteurs hors-données définissent seulement la méthode *__get__()*. Les descripteurs de données qui définissent *__set__()* et *__get__()* sont toujours prioritaires face à une redéfinition du dictionnaire de l'instance. En revanche, les descripteurs hors-données peuvent être shuntés par les instances.

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

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

__slots__

Les `__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.

Note sur l'utilisation de *__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 (*Implémentation de descripteurs*) pour chaque nom de variable. Ainsi, les attributs de classe ne peuvent pas être utilisés pour des valeurs par défaut aux variables d'instances définies par *__slots__* ; sinon, l'attribut de classe surchargerait l'assignation par descripteur.
- 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* du *slot*).
- 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.
- Un *__slot__* non vide ne fonctionne pas pour les classes dérivées des types natifs à longueur variable tels que `int`, `bytes` et `tuple`.

— Tout itérable qui n'est pas une chaîne peut être assigné à un *__slots__*. Les tableaux de correspondance peuvent aussi être utilisés ; cependant, dans le futur, des significations spéciales pourraient être associées à chacune des clés.
— Les assignations de *__class__* ne fonctionnent que si les deux classes ont le même *__slots__*.
— 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`.

3.3.3 Personnalisation de la création de classes

Quand une classe hérite d'une classe parente, *__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.

Les arguments nommés qui sont donnés à la nouvelle classe sont passés à `__init_subclass__` de la classe parente. Par souci de compatibilité avec les autres classes qui utilisent `__init_subclass__`, vous devez enlever les arguments nommés dont vous avez besoin et passer les autres à la classe de base, comme ci-dessous :

```
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 `object.__init_subclass__` ne fait rien mais lève une erreur si elle est appelée avec un argument.

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

Nouveau dans la version 3.6.

Méta-classes

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
class MySubclass(MyClass):
    pass
```

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

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

- Les entrées MRO sont résolues
- la méta-classe appropriée est déterminée ;
- l'espace de nommage de la classe est préparé ;
- le corps de la classe est exécuté ;
- l'objet classe est crée.

Résolution des entrées MRO

Si une classe de base qui apparaît dans la définition d'une classe n'est pas une instance de `type`, alors Python y recherche une méthode `__mro_entries__`. S'il la trouve, il l'appelle avec le tuple original des classes de bases. Cette méthode doit renvoyer un tuple de classes qui est utilisé à la place de la classe de base. Le tuple peut être vide, dans ce cas la classe de base originale est ignorée.

Voir aussi :

PEP 560 — Gestion de base pour les types modules et les types génériques

Détermination de la méta-classe appropriée

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

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

La méta-classe la plus dérivée est choisie à partir des méta-classes explicitement spécifiées (s'il y en a) et les méta-classes (c'est-à-dire les `type(cls)`) de toutes les classes de base spécifiées. La méta-classe la plus dérivée est celle qui est un sous-type de *toutes* ces méta-classes candidates. Si aucune des méta-classes 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 méta-classe appropriée est identifiée, l'espace de nommage de la classe est préparé. Si la méta-classe possède un attribut `__prepare__`, il est appelé avec `namespace = metaclass.__prepare__(name, bases, **kwds)` (où les arguments nommés supplémentaires, s'il y en a, viennent de la définition de la classe).

Si la méta-classe 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 — Méta-classes 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.

CPython implementation detail : Dans CPython 3.6 et suivants, la cellule `__class__` est passée à la méta-classe 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 avertissement `DeprecationWarning` dans Python 3.6 et un `RuntimeError` dans Python 3.8.

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

- d'abord, `type.__new__` récupère, dans l'espace de nommage de la classe, tous les descripteurs qui définissent une méthode *`__set_name__()`* ;
- ensuite, toutes ces méthodes `__set_name__` sont appelées avec la classe en cours de définition et le nom assigné à chaque descripteur ;
- finalement, 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 méta-classes 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 méta-classe `abc.ABCMeta` implémente ces méthodes pour autoriser l'ajout de classes de base abstraites (ABC pour *Abstract Base Classes* en anglais) en tant que "classes de base virtuelles" pour toute classe ou type (y compris les types natifs).

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

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

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

Voir aussi :

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

3.3.5 Émulation de types génériques

Vous pouvez implémenter la syntaxe générique des classes comme spécifié par la **PEP 484** (par exemple `List[int]`) en définissant une méthode spéciale :

classmethod `object.__class_getitem__`(*cls*, *key*)
: Renvoie un objet représentant la spécialisation d'une classe générique en fonction des arguments types trouvés dans *key*.

Python recherche cette méthode dans l'objet de classe lui-même et, lorsqu'elle est définie dans le corps de la classe, cette méthode est implicitement une méthode de classe. Notez que ce mécanisme est principalement réservé à une utilisation avec des indications de type statiques, d'autres utilisations sont déconseillées.

Voir aussi :

PEP 560 — Gestion de base pour les types modules et les types génériques

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 `x.__call__(arg1, arg2, ...)`.

3.3.7 Émulation de types conteneurs

Les fonctions suivantes peuvent être définies pour implémenter des objets conteneurs. Les conteneurs sont habituellement des séquences (telles que les tuples ou les listes) ou des tableaux de correspondances (comme les dictionnaires),

mais ils peuvent aussi représenter d'autres conteneurs. Le premier ensemble de méthodes est utilisé soit pour émuler une séquence, soit pour émuler un tableau de correspondances ; la différence est que, pour une séquence, les clés doivent être soit des entiers *k* tels que `0 <= k < N` où *N* est la longueur de la séquence, soit des objets tranches qui définissent un intervalle d'éléments. Il est aussi recommandé que les tableaux de correspondances fournissent les méthodes `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()` et `update()` avec un comportement similaire aux objets dictionnaires standards de Python. Le module `collections.abc` fournit une classe de base abstraite `MutableMapping` pour aider à la création de ces méthodes à partir d'un ensemble de base composé de *`__getitem__()`*, *`__setitem__()`*, *`__delitem__()`* et `keys()`. Les séquences muables doivent fournir les méthodes `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` et `sort()`, comme les objets listes standards de Python. Enfin, les types séquences doivent implémenter l'addition (dans le sens de la concaténation) et la multiplication (dans le sens de la répétition) en définissant les méthodes *`__add__()`*, *`__radd__()`*, *`__iadd__()`*, *`__mul__()`*, *`__rmul__()`* et *`__imul__()`* décrites ci-dessous ; ils ne doivent pas définir d'autres opérateurs numériques. Il est recommandé que les tableaux de correspondances et les séquences implémentent la méthode *`__contains__()`* pour permettre l'utilisation efficace de l'opérateur `in` ; concernant les tableaux de correspondances, `in` doit rechercher dans les clés du tableau ; pour les séquences, il doit chercher dans les valeurs. Il est de plus recommandé que les tableaux de correspondances et les séquences implémentent la méthode *`__iter__()`* pour permettre une itération efficace dans le conteneur ; pour les tableaux de correspondances, *`__iter__()`* doit être la même que `keys()` ; pour les séquences, elle doit itérer sur les valeurs.

`object.`**`__len__`**`(`*self*`)`
: Appelée pour implémenter la fonction native `len()`. Elle doit renvoyer la longueur de l'objet, un entier >= 0. Par ailleurs, un objet qui ne définit pas de méthode *`__bool__()`* et dont la méthode *`__len__()`* renvoie zéro est considéré comme valant `False` dans un contexte booléen.

  CPython implementation detail : En CPython, la longueur doit valoir au maximum `sys.maxsize`. Si la longueur est plus grande que `sys.maxsize`, des propriétés (telles que `len()`) peuvent lever `OverflowError`. Afin d'éviter de lever `OverflowError` lors de tests booléens, un objet doit définir la méthode *`__bool__()`*.

`object.`**`__length_hint__`**`(`*self*`)`
: Appelée pour implémenter `operator.length_hint()`. Elle doit renvoyer une longueur estimée de l'objet (qui peut être plus grande ou plus petite que la longueur réelle). La longueur doit être un entier >= 0. Cette méthode est utilisée uniquement pour optimiser les traitements et n'est jamais tenue de renvoyer un résultat exact.

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

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

  ---

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

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

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

object.**__iter__**(*self*)
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.

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

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

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

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

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

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

3.3.8 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__()*, `x.__add__(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.

Si l'une de ces méthodes n'autorise pas l'opération avec les arguments donnés, elle doit renvoyer `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*)
object.**__rlshift__**(*self*, *other*)
object.**__rrshift__**(*self*, *other*)
object.**__rand__**(*self*, *other*)
object.**__rxor__**(*self*, *other*)
object.**__ror__**(*self*, *other*)

Ces méthodes sont appelées pour implémenter les opérations arithmétiques binaires (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) avec des opérandes renversés (intervertis). Ces fonctions ne sont appelées que si l'opérande de gauche n'autorise pas l'opération correspondante[3] et si les opérandes sont de types différents[4]. Par exemple, pour évaluer l'expression `x - y`, où *y* est une instance d'une classe qui possède une méthode *__rsub__()*, `y.__rsub__(x)` est appelée si `x.__sub__(y)` renvoie *NotImplemented*.

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

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

---

object.**__iadd__**(*self*, *other*)
object.**__isub__**(*self*, *other*)
object.**__imul__**(*self*, *other*)
object.**__imatmul__**(*self*, *other*)
object.**__itruediv__**(*self*, *other*)
object.**__ifloordiv__**(*self*, *other*)
object.**__imod__**(*self*, *other*)
object.**__ipow__**(*self*, *other*[, *modulo*])

3. *n'autorise pas* signifie ici que la classe ne possède pas de méthode adéquate ou que la méthode renvoie `NotImplemented`. N'assignez pas `None` à la méthode si vous voulez un repli vers la méthode symétrique de l'opérande de droite — cela aurait pour effet de *bloquer* un tel repli.

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

object.**__ilshift__**(*self*, *other*)
object.**__irshift__**(*self*, *other*)
object.**__iand__**(*self*, *other*)
object.**__ixor__**(*self*, *other*)
object.**__ior__**(*self*, *other*)

Ces méthodes sont appelées pour implémenter les assignations arithmétiques augmentées (+=, -=, *=, @=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=). Ces méthodes doivent essayer d'effectuer l'opération "sur place" (c'est-à-dire de modifier *self*) et de renvoyer le résultat (qui peut être, mais pas nécessairement, *self*). Si une méthode spécifique n'est pas définie, l'assignation augmentée se replie vers la méthode normale correspondante. Par exemple, si *x* est une instance d'une classe qui possède une méthode *__iadd__()*, `x += y` est équivalent à `x = x.__iadd__(y)` . Sinon, `x.__add__(y)` et `y.__radd__(x)` sont essayées, comme pour l'évaluation de `x + y`. Dans certaines situations, les assignations augmentées peuvent causer des erreurs inattendues (voir faq-augmented-assignment-tuple-error), mais ce comportement est en fait partie intégrante du modèle de données.

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

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

object.**__complex__**(*self*)
object.**__int__**(*self*)
object.**__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.

Note : Afin d'avoir un type de classe entier cohérent, lorsque *__index__()* est définie alors *__int__()* doit aussi être définie et les deux doivent renvoyer la même valeur.

---

object.**__round__**(*self*[, *ndigits*])
object.**__trunc__**(*self*)
object.**__floor__**(*self*)
object.**__ceil__**(*self*)

Appelée 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`).

Si *__int__()* n'est pas définie, alors la fonction native `int()` se replie sur *__trunc__()*.

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.

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

Voir aussi :

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

3.3.10 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 méta-classe" et se contourne en shuntant l'instance lors de la recherche des méthodes spéciales :

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

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

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

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

Note : Les objets *itérateur de générateur* renvoyés par les générateurs décorés par `types.coroutine()` ou `asyncio.coroutine()` sont aussi des *awaitables*, 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*.

Nouveau dans la version 3.5.

Voir aussi :

PEP 492 pour les informations relatives aux objets *awaitables*.

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*`)`
: Démarre ou reprend l'exécution d'une coroutine. Si *value* est `None`, c'est équivalent à avancer l'itérateur renvoyé par `__await__()`. Si *value* ne vaut pas `None`, cette méthode appelle la méthode `send()` de l'itérateur qui a causé la suspension de la coroutine. 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.

`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.

`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 *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

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

Nouveau dans la version 3.5.

Modifié dans la version 3.7 : Avant Python 3.7, `__aiter__` pouvait renvoyer un *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*)
: Cette méthode est sémantiquement équivalente à *__enter__()*, à la seule différence près qu'elle doit renvoyer un *awaitable*.

`object.`**`__aexit__`**(*self*, *exc_type*, *exc_value*, *traceback*)
: Cette méthode est sémantiquement équivalente à *__exit__()*, à la seule différence près qu'elle doit renvoyer un *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')
```

Nouveau dans la version 3.5.

Notes

CHAPITRE 4

Modèle d'exécution

4.1 Structure d'un programme

Un programme Python est construit à partir de blocs de code. Un *block* 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 sur la ligne de commande de l'interpréteur avec l'option `-c`) est 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 constructions suivantes conduisent à des opérations de liaison à des noms : les paramètres formels d'une fonction, les instructions *`import`*, les définitions de fonctions et de classes (le nom de la classe ou de la fonction est lié au bloc qui la définit) et les cibles qui sont des identifiants dans les assignations, les entêtes de boucles *`for`* ou après `as` dans une instruction *`with`* ou une clause *`except`*. L'instruction `import` sous la forme `from ... import *` lie tous les noms définis dans le module importé, sauf ceux qui commencent par un tiret bas ('_'). Cette forme ne doit 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 assignation ou instruction *import* a lieu dans un bloc défini par une définition de classe ou de fonction ou au niveau du module (le bloc de code de plus haut niveau).

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`.

Si une opération de liaison intervient dans un bloc de code, toutes les utilisations du nom dans le bloc sont considérées comme des références au bloc courant. Ceci peut conduire à des erreurs quand un nom est utilisé à l'intérieur d'un bloc avant d'être lié. La règle est subtile. Python n'attend pas de déclaration de variables et autorise les opérations de liaison n'importe où dans un bloc de code. Les variables locales d'un bloc de code peuvent être déterminées en parcourant tout le texte du bloc à la recherche des opérations de liaisons.

Si l'instruction *global* apparaît dans un bloc, toutes les utilisations du nom spécifié dans l'instruction font référence à la liaison de ce nom dans l'espace de nommage de plus haut niveau. Les noms sont résolus dans cet espace de nommage de plus haut niveau en recherchant l'espace des noms globaux, c'est-à-dire l'espace de nommage du module contenant le bloc de code ainsi que dans l'espace de nommage natif, celui du module `builtins`. La recherche commence dans l'espace de nommage globaux. Si le nom n'y est pas trouvé, la recherche se poursuit dans l'espace de nommage natif. L'instruction `global` doit précéder toute utilisation du nom considéré.

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.

L'instruction *nonlocal* fait que les noms correspondants font référence aux variables liées précédemment dans la portée de fonction englobante la plus petite possible. `SyntaxError` est levée à la compilation si le nom donné n'existe dans aucune portée de fonction englobante.

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__`.

Les blocs de définition de classe et les arguments de `exec()` ainsi que `eval()` sont traités de manière spéciale dans le cadre de la résolution des noms. Une définition de classe est une instruction exécutable qui peut utiliser et définir des noms. Toutes ces références suivent les règles normales de la résolution des noms à l'exception des variables locales non liées qui sont recherchées dans l'espace des noms globaux. L'espace de nommage de la définition de classe devient le dictionnaire des attributs de la classe. La portée des noms définis dans un bloc de classe est limitée au bloc de la classe ; elle ne s'étend pas aux blocs de code des méthodes – y compris les compréhensions et les expressions générateurs puisque celles-ci sont implémentées en utilisant une portée de fonction. Ainsi, les instructions suivantes échouent :

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

4.2.3 Noms natifs et restrictions d'exécution

CPython implementation detail : 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.

4.2.4 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.

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. L'instance peut être transmise au gestionnaire et peut apporter des informations complémentaires sur les conditions de l'exception.

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

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 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, créé 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'import 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 du paquet des noms natifs a été implémenté (voir la **PEP 420**).

1. Voir `types.ModuleType`.

5.1 importlib

Le module `importlib` fournit une API riche pour interagir avec le système d'import. 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'import. 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 des points (.), à l'instar de la syntaxe standard d'accès aux attributs en Python. Ainsi, vous pouvez avoir un module nommé `sys` et un paquet nommé `email`, qui a son tour possède un sous-paquet nommé `email.mime` avec 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 le 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 gelés. 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. Ceci permet aux méta-points d'entrée de surcharger le traitement effectué sur `sys.path`, les modules gelés 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'import 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 gelés et un qui sait importer les modules depuis un *chemin des importations* (c'est le *chercheur dans path*).

Modifié dans la version 3.4 : La méthode `find_spec()` des chercheurs dans les méta-chemins a remplacé `find_module()`, devenue obsolète. Bien qu'elle continue de fonctionner comme avant, le mécanisme d'importation essaie `find_module()` uniquement si le chercheur n'implémente pas `find_spec()`.

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

(suite sur la page suivante)

(suite de la page précédente)

```
        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.

Nouveau dans la 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.

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.

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.

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

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
    bar.py
```

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

```
from .foo import Foo
from .bar import Bar
```

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

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

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.

Nouveau dans la 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.

`__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.

`__package__`
: L'attribut `__package__` du module doit être défini. Sa valeur doit être une chaîne de caractères, qui peut être la même que son attribut `__name__`. Quand le module est un paquet, la valeur de `__package__` doit être définie à la même valeur que son `__name__`. Quand le module n'est pas un paquet, `__package__` doit être la chaîne vide pour les modules de niveau le plus haut, et le nom du paquet parent pour les sous-modules. Voir la **PEP 366** pour plus de détails.

  Cet attribut est utilisé à la place de `__name__` pour calculer les importations relatives explicites des modules principaux, comme défini dans la **PEP 366**. Il devrait avoir la même valeur que `__spec__.parent`.

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

`__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*.

  Quand `__package__` n'est pas défini, `__spec__.parent` est utilisé par défaut.

  Nouveau dans la 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. S'il est défini, la valeur de cet attribut doit être une chaîne. Le système d'importation peut décider de laisser `__file__` indéfini si cela ne fait pas sens de le définir (par exemple, lorsqu'on charge un module 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**).

Vous pouvez aussi définir __cached__ quand __file__ n'est pas défini. Cependant, ce scénario semble rare. Au final, c'est le chargeur qui utilise les valeurs de __file__ ou __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.

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 3.4 : L'utilisation de loader.module_repr() est devenue obsolète et le spécificateur de module est utilisé dorénavant par le mécanisme d'importation pour générer la représentation textuelle du module.

Par compatibilité descendante avec Python 3.3, la représentation textuelle du module est générée en appelant la méthode module_repr() du chargeur, si elle est définie, avant même d'essayer l'approche décrite ci-dessus. Cependant, cette méthode est obsolète.

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 *callable*) 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` contient une liste de chaînes de caractères indiquant des emplacements où chercher des modules ou des paquets. Elle est initialisée à partir de la variable d'environnement `PYTHONPATH` et de plusieurs autres valeurs par défaut qui dépendent de l'installation et de l'implémentation. Les entrées de `sys.path` désignent des répertoires du système de fichiers, des fichiers zip et possiblement d'autres "endroits" (lisez le module `site`) tels que des URL ou des requêtes dans des bases de données où Python doit rechercher des modules. `sys.path` ne doit contenir que des chaînes de caractères ou d'octets ; tous les autres types sont ignorés. L'encodage des entrées de chaînes d'octets est déterminé par chaque *chercheur d'entrée dans path*.

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.

Le chercheur dans *path* itère sur chaque entrée dans le *path* et, pour chacune, regarde s'il trouve un *chercheur d'entrée dans path* (`PathEntryFinder`) approprié à cette entrée. Comme cette opération est coûteuse (elle peut faire appel à plusieurs *stat()* pour cela), le chercheur dans *path* maintient un cache de correspondance entre les entrées et les "chercheurs d'entrée dans *path*". Ce cache est géré par `sys.path_importer_cache` (en dépit de son nom, ce cache stocke les objets chercheurs plutôt que les simples objets *importateurs*). Ainsi, la recherche coûteuse pour une *entrée de path* spécifique n'a besoin d'être effectuée qu'une seule fois par le *chercheur d'entrée dans path*. Le code de l'utilisateur peut très bien supprimer les entrées du cache `sys.path_importer_cache`, forçant ainsi le chercheur dans *path* à effectuer une nouvelle fois la recherche sur chaque entrée [3].

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.

3. Dans du code historique, il est possible de trouver des instances de `imp.NullImporter` dans `sys.path_importer_cache`. Il est recommandé de modifier ce code afin d'utiliser `None` à la place. Lisez portingpythoncode pour plus de détails.

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.

Pour indiquer au mécanisme d'importation que le spécificateur représente une *portion* d'un espace de nommage, le chercheur d'entrée dans *path* définit le chargeur du spécificateur à `None` et l'attribut *submodule_search_locations* à une liste contenant la portion.

Modifié dans la version 3.4 : La méthode `find_spec()` remplace `find_loader()` et `find_module()`, ces deux méthodes étant dorénavant obsolètes mais restant utilisées si `find_spec()` n'est pas définie.

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.

La méthode `find_loader()` prend un argument, le nom complètement qualifié du module en cours d'importation. `find_loader()` renvoie un couple dont le premier élément est le chargeur et le second est une *portion* d'espace de nommage. Quand le premier élément (c'est-à-dire le chargeur) est `None`, cela signifie que, bien que le chercheur d'entrée dans *path* n'a pas de chargeur pour le module considéré, il sait que cette entrée contribue à une portion d'espace de nommage pour le module considéré. C'est presque toujours le cas quand vous demandez à Python d'importer un paquet-espace de nommage qui n'est pas présent physiquement sur le système de fichiers. Quand un chercheur d'entrée dans *path* renvoie `None` pour le chargeur, la valeur du second élément du couple renvoyé doit être une séquence, éventuellement vide.

Si `find_loader()` renvoie une valeur de chargeur qui n'est pas `None`, la portion est ignorée et le chargeur est renvoyé par le chercheur dans *path*, mettant un terme à la recherche dans les chemins.

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()`.

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, `None` 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 Idées d'amélioration

XXX Ce serait vraiment bien de disposer d'un diagramme.

XXX * (`import_machinery.rst`) Pourquoi pas une section dédiée aux attributs des modules et paquets, développant ou remplaçant les entrées associées dans la page de référence du modèle de données ?

XXX *runpy*, *pkgutil* et autres dans le manuel de la bibliothèque devraient comporter un lien "Lisez aussi" en début de page pointant vers la section du nouveau mécanisme d'import.

XXX Ajouter des explications sur les différentes manières dont `__main__` est initialisé ?

XXX Ajouter des informations sur les pièges et bizarreries de `__main__` (c-à-d des extraits de la **PEP 395**).

5.10 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.

La **PEP 420** a introduit les *paquets-espaces de nommage* pour Python 3.3. **PEP 420** a aussi introduit le protocole `recherche du chargeur` comme une alternative à `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'import. 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* <*private name*> de cette classe. Les noms privés sont transformés en une forme plus longue avant que le code ne soit généré pour eux. La transformation insère le nom de la classe, avec les soulignés enlevés et un seul souligné inséré devant le nom. Par exemple, l'identificateur `__spam` apparaissant dans une classe nommée `Ham` est transformé en `_Ham__spam`. Cette transformation est indépendante du contexte syntaxique dans lequel l'identificateur est utilisé. Si le nom transformé est extrêmement long (plus de 255 caractères), l'implémentation peut le tronquer. Si le nom de la classe est constitué uniquement de traits de soulignement, aucune transformation n'est effectuée.

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

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

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   ::=   expression comp_for
comp_for        ::=   ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter       ::=   comp_for | comp_if
comp_if         ::=   "if" expression_nocond [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 (dans Python 3.7, de telles expressions lèvent un `DeprecationWarning` à la compilation, dans Python 3.8 et suivants, elles lèveront une `SyntaxError`).

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 une clause `for` or `async for` suivie par des clauses `for` ou `async for` additionnelles facultatives et, possiblement, des expressions *`await`*. Si la compréhension contient soit des clauses `async for`, soit des expressions `await`, 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**.

Nouveau dans la version 3.6 : Les compréhensions asynchrones ont été introduites.

Obsolète depuis la version 3.7 : `yield` et `yield from` sont obsolètes dans la portée implicite imbriquée.

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

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

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        ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list      ::=  key_datum ("," key_datum)* [","]
key_datum           ::=  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 donnée correspondante. Cela signifie que vous pouvez spécifier la même clé plusieurs fois dans la liste des couples clés-valeurs et, dans ce cas, la valeur finalement stockée dans le dictionnaire est la dernière donnée.

Une 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.

Nouveau dans la 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 muables). Les collisions entre les clés dupliquées ne sont pas détectées ; la dernière valeur (celle qui apparaît le plus à droite dans l'agencement) stockée prévaut pour une clé donnée.

6.2.8 Générateurs (expressions)

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

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

Une expression générateur produit un nouvel objet générateur. Sa syntaxe est la même que celle des compréhensions, sauf

qu'elle est entourée de parenthèses au lieu de crochets ou d'accolades.

Les variables utilisées dans une expression générateur 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 générateur elle-même, les expressions `yield` et `yield from` sont interdites dans les générateurs définis de manière implicite (dans Python 3.7, ces expressions signalent un `DeprecationWarning` à la compilation. En Python 3.8+ elles lèvent une `SyntaxError`).

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

Nouveau dans la version 3.6 : les expressions générateurs asynchrones ont été introduites.

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

Obsolète depuis la version 3.7 : `yield` et `yield from` sont obsolètes dans la portée implicite imbriquée.

6.2.9 Expressions `yield`

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

Une expression `yield` est utilisée pour définir une fonction *générateur* ou une fonction *générateur 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 un générateur et son utilisation dans le corps d'une fonction *`async def`* entraine que cette fonction coroutine devient un générateur 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 générateurs (dans Python 3.7, de telles expressions lèvent un `DeprecationWarning` à la compilation ; à partir de Python 3.8, elles lèveront une `SyntaxError`).

Obsolète depuis la version 3.7 : Les expressions *yield* sont obsolètes dans la portée implicite imbriquée utilisée dans l'implémentation des compréhensions et des expressions générateurs.

Les fonctions générateurs sont décrites plus loin alors que les fonctions générateurs asynchrones sont décrites séparément dans la section *Fonctions générateurs asynchrones*.

Lorsqu'une fonction générateur est appelée, elle renvoie un itérateur que l'on appelle générateur. Ce générateur contrôle l'exécution de la fonction générateur. 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. 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 (typiquement *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 générateurs 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 générateur 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'utilisation de `yield from <expr>` traite l'expression passée en paramètre comme un sous-itérateur. Toutes les valeurs produites par ce sous-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'assignation.

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 générateurs.

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

`generator.`**`__next__`**`()`

> Démarre l'exécution d'une fonction générateur ou la reprend à la dernière expression `yield` exécutée. Quand une fonction générateur 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 générateur. 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`**`(`*type*`[`, *value*`[`, *traceback*`]]`)

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

`generator.`**`close`**`()`

Lève une `GeneratorExit` à l'endroit où la fonction générateur a été mise en pause. Si la fonction générateur 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 générateurs :

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 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 générateurs 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 générateur asynchrone est appelée, elle renvoie un itérateur asynchrone, autrement appelé objet générateur asynchrone. Cet objet contrôle l'exécution de la fonction générateur. 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.

Dans une fonction générateur 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 relatif 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 générateur 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 générateurs.

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 générateur 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 générateur asynchrone et cet argu-

ment 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**(*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 générateur 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 générateur 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*.

coroutine `agen.`**aclose**()
Renvoie un *awaitable* qui, quand il s'exécute, lève une exception `GeneratorExit` dans la fonction générateur asynchrone à l'endroit où le générateur était en pause. Si la fonction générateur 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
```

La primaire doit s'évaluer à un objet d'un type qui gère les références aux attributs, ce qui est le cas de la plupart des objets. Cet objet doit alors produire l'attribut dont le nom est "identifier". Cette production peut être personnalisée en surchargeant la méthode *`__getattr__()`*. Si l'attribut n'est pas disponible, une exception `AttributeError` est levée. Sinon, le type et la valeur de l'objet produit sont déterminés par l'objet. Plusieurs évaluations successives d'une référence à un même attribut peuvent produire différents objets.

6.3.2 Sélections

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

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

La primaire doit s'appliquer à un objet qui gère les sélections (une liste ou un dictionnaire par exemple). Les objets définis par l'utilisateur peuvent gérer les sélections en définissant une méthode *__getitem__()*.

Pour les objets natifs, deux types d'objets gèrent la sélection :

Si la primaire est un tableau de correspondances, la liste d'expressions (*expression_list* dans la grammaire formelle ci-dessous) doit pouvoir être évaluée comme un objet dont la valeur est une des clés du tableau de correspondances et la sélection désigne la valeur qui correspond à cette clé (la liste d'expressions est un n-uplet sauf si elle comporte exactement un élément).

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).

La syntaxe formelle ne traite pas des cas d'indices négatifs dans les séquences ; cependant, toutes les séquences natives possèdent une méthode *__getitem__()* qui interprète les indices négatifs en ajoutant la longueur de la séquence à l'indice (de manière à ce que `x[-1]` sélectionne le dernier élément de `x`). La valeur résultante doit être un entier positif ou nul, inférieur au nombre d'éléments dans la séquence ; la sélection désigne alors l'élément dont l'indice est cette valeur (en comptant à partir de zéro). Comme la gestion des indices négatifs et des tranches est faite par la méthode *__getitem__()*, les sous-classes qui surchargent cette méthode doivent aussi savoir les gérer, de manière explicite.

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

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 assignations 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).

La sémantique pour une tranche est définie comme suit. La primaire est indicée (en utilisant la même méthode *__getitem__()* que pour les sélections normales) avec une clé qui est construite à partir de la liste de tranches (*slice_list* dans la grammaire), de cette manière : si la liste de tranches contient au moins une virgule (`,`), la clé est un n-uplet contenant la conversion des éléments de la tranche ; sinon, la conversion du seul élément de la tranche est la clé. La conversion d'un élément de tranche qui est une expression est cette expression. La conversion d'une tranche en tant que telle est un objet *slice* (voir la section *Hiérarchie des types standards*) dont les attributs `start`, `stop` et `step` sont les valeurs des expressions données pour la borne inférieure (*lower_bound* dans la grammaire), la borne supérieure (*up-*

per_bound dans la grammaire) et le pas (*stride* dans la grammaire), respectivement. En cas d'expression manquante, la valeur par défaut est `None`.

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 ::=  ["*"] 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 par mots-clés, mais elle n'affecte pas la sémantique.

La primaire doit s'évaluer à un objet appelable (une fonction définie par l'utilisateur, une fonction native, une méthode d'objet natif, un objet de classe, une méthode d'instance de classe ou tout objet possédant une méthode *__call__()* est un appelable). Toutes les expressions des arguments sont évaluées avant que l'appel ne soit exécuté. Référez-vous à la section *Définition de fonctions* pour la syntaxe des listes de *paramètres* formels.

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

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

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

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

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 mots-clés explicites, ils sont traités *avant* les arguments par mots-clés (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 hypothétique.

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 mot-clé est déjà présent (en tant qu'argument par mot-clé explicite, ou venant d'un autre dépaquetage), 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 : la classe doit définir une méthode `__call__()` ; l'effet est le même que si cette méthode était appelée.

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
```

Nouveau dans la 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 numériques 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`).

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
```

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

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

L'opérateur unaire ~ (inversion) produit l'inversion bit à bit de son argument entier. L'inversion bit à bit de `x` est définie comme `-(x+1)`. Elle s'applique uniquement aux nombres entiers.

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.

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

Nouveau dans la 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`.

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.

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.

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

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

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`.

du nombre de bits donné par le deuxième argument.

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)`.

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
```

L'opérateur `&` produit le ET logique de ses arguments, qui doivent être des entiers.

L'opérateur `^` produit le OU EXCLUSIF (XOR) logique de ses arguments, qui doivent être des entiers.

L'opérateur `|` produit le OU logique de ses arguments, qui doivent être des entiers.

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

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

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

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

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.

Comme tous les types sont des sous-types (directs ou indirects) de la classe `object`, ils héritent du comportement de comparaison par défaut de `object`. Les types peuvent personnaliser le comportement des comparaisons en implémentant des *méthodes de comparaisons riches*, comme `__lt__()`, décrites dans *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')`, `3 < x`, `x < 3`, `x == x`, `x != x` sont toutes fausses. Ce comportement est en accord avec IEEE 754.

— 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 suivent l'ordre lexicographique en utilisant la comparaison de leurs éléments, sachant que la réflexivité des éléments est appliquée.
Dans l'application de la réflexivité des éléments, la comparaison des collections suppose que pour un élément de collection `x`, `x == x` est toujours vrai. Sur la base de cette hypothèse, l'identité des éléments est d'abord testée, puis la comparaison des éléments n'est effectuée que pour des éléments distincts. Cette approche donne le même résultat qu'une comparaison stricte d'éléments, si les éléments comparés sont réflexifs. Pour les éléments non réflexifs, le résultat est différent de celui de la comparaison stricte des éléments, voire peut être surprenant : les valeurs non réflexives qui ne sont pas des nombres, par exemple, aboutissent au comportement suivant lorsqu'elles sont utilisées dans une liste :

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()`.

```
>>> nan = float('NaN')
>>> nan is nan
True
>>> nan == nan
False                  <-- the defined non-reflexive behavior of NaN
>>> [nan] == [nan]
True                   <-- list enforces reflexivity and tests identity first
```

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`.

— 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 allogènes 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 gelés (*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`.

Pour les classes allogènes qui définissent la méthode *__contains__()*, `x in y` renvoie `True` si `y.__contains__(x)` renvoie vrai, et `False` sinon.

Pour les classes allogènes qui ne définissent pas *__contains__()* mais qui définissent *__iter__()*, `x in y` vaut `True` s'il existe une valeur `z` telle que l'expression `x is z or x == z` renvoie vrai lors de l'itération sur `y`. Si une exception est levée pendant l'itération, c'est comme si *in* avait levé cette exception.

Enfin, le protocole d'itération « à l'ancienne » est essayé : si la classe définit *__getitem__()*, `x in y` est `True` si et seulement si il existe un entier positif ou nul *i*, représentant l'indice, tel que `x is y[i] or x == y[i]` et qu'aucun indice inférieur ne lève d'exception `IndexError` (si toute autre exception est levée, c'est comme si *in* avait levé cette 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
```

Dans le contexte des opérations booléennes et quand des expressions sont utilisées par des instructions de contrôle du flux d'exécution, les valeurs suivantes sont considérées comme fausses : `False`, `None`, zéro quel que soit le type, la chaîne vide et tout conteneur vide (y compris les chaînes, n-uplets, listes, dictionnaires, ensembles, ensembles gelés). Toutes les autres valeurs sont considérées comme vraies. Les objets allogènes peuvent personnaliser leur table de vérité en implémentant une méthode *__bool__()*.

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.

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.

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 conditionnelles

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

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.

6.13 Expressions lambda

```
lambda_expr          ::=   "lambda" [parameter_list] ":" expression
lambda_expr_nocond   ::=   "lambda" [parameter_list] ":" expression_nocond
```

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.

6.14 Listes d'expressions

```
expression_list      ::=   expression ("," expression)* [","]
starred_list         ::=   starred_item ("," starred_item)* [","]
starred_expression   ::=   expression | (starred_item ",")* [starred_item]
starred_item         ::=   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 (*tuple*). 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 *tuple*, *list* ou *set* à l'emplacement du

dépaquetage.

Nouveau dans la version 3.5 : dépaquetage d'itérables dans les listes d'expressions, proposé à l'origine par la **PEP 448**.

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

6.15 Ordre d'évaluation

Python évalue les expressions de la gauche vers la droite. Remarquez que lors de l'évaluation d'une assignation, la partie droite de l'assignation 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
```

6.16 Priorités des opérateurs

Le tableau suivant résume les priorités des opérateurs en Python, du moins prioritaire au plus prioritaire. 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 cases regroupent de la gauche vers la droite (sauf pour la puissance qui regroupe 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
lambda	Expression lambda
if – else	Expressions conditionnelle
or	OR (booléen)
and	AND (booléen)
not x	NOT (booléen)
in, *not in*, *is*, *is not*, <, <=, >, >=, !=, ==	Comparaisons, y compris les tests d'appartenance et les tests d'identifiants
|	OR (bit à bit)
^	XOR (bit à bit)
&	AND (bit à bit)
<<, >>	décalages
+, –	Addition et soustraction
*, @, /, //, %	Multiplication, multiplication de matrices, division, division entière, reste [5]
+x, –x, ~x	NOT (positif, négatif, bit à bit)
**	Puissance [6]
await x	Expression await
x[indice], x[indice:indice], x(arguments...), x.attribut	indiçage, tranches, appel, référence à un attribut
(expressions...), [expressions...], {clé: valeur...}, {expressions...}	liaison ou agencement de n-uplet, agencement de liste, agencement de dictionnaire, agencement d'ensemble

Notes

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

CHAPITRE 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
```

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 muables :

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

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 : 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.
  - 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.
Pour les objets allogènes, la méthode *`__setitem__()`* est appelée avec les arguments appropriés.

— 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 différent de la longueur de la séquence à assigner, ce qui modifie alors la longueur de la séquence cible, si celle-ci le permet.

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

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

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 ["=" expression]
```

La différence avec une assignation normale (voir *Les assignations*) est qu'une seule cible et qu'une seule valeur à droite ne sont autorisées.

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.

Si la partie droite est présente, une assignation annotée effectue l'assignation en tant que telle avant d'évaluer les annotations (là où c'est possible). Si la partie droite n'est pas présente pour une cible d'expression, alors l'interpréteur évalue la cible sauf pour le dernier appel à *__setitem__()* ou *__setattr__()*.

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 EDIs.

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 générateur, 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 générateur 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 générateur 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 générateur. L'utilisation de *yield* dans la définition d'une fonction est suffisant pour que cette définition crée une fonction générateur 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 la dernière exception active dans la portée courante. Si aucune exception n'est active dans la portée courante, 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.

Normalement, un objet *trace d'appels* est créé automatiquement quand une exception est levée et il lui est rattaché comme attribut `__traceback__`, en lecture-écriture. Vous pouvez créer une exception et définir votre propre trace d'appels d'un seul coup en utilisant la méthode des exceptions `with_traceback()` (qui renvoie la même instance d'exception avec sa trace d'appels passée en argument), comme ceci :

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

La clause `from` est utilisée pour chaîner les exceptions : si vous la fournissez, la seconde "expression" doit être une autre classe ou instance d'exception, qui est rattachée à l'exception levée en tant qu'attribut `__cause__` (en lecture-écriture). Si l'exception levée n'est pas gérée, les deux exceptions sont affichées :

```
>>> 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>
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>
RuntimeError: Something bad happened
```

Un mécanisme similaire est mis en œuvre implicitement si une exception est levée à l'intérieur d'un gestionnaire d'excep-

tion ou d'une clause *finally* : la première exception est rattachée à l'attribut __context__ de la nouvelle exception :

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Le chaînage d'exceptions peut être explicitement supprimé en spécifiant `None` dans la clause `from` :

```
>>> 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`.

Nouveau dans la version 3.3 : L'attribut __suppress_context__ pour supprimer l'affichage automatique du contexte de l'exception.

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 ni dans une clause *finally*, à l'intérieur de cette boucle. Elle fait continuer le flot d'exécution au prochain cycle de la boucle la plus imbriquée.

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

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" identifier]
                        ("," identifier ["as" identifier])* [","] ")"
                        | "from" 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.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo.bar.baz imported and bound as fbb
from foo.bar import baz    # foo.bar.baz imported and 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.

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 dans Python 3.7 qui nécessite l'utilisation de l'instruction *future* est `annotations`.

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 ou l'instruction *future* est exécutée.

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

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

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

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

Le code compilé par des appels aux fonctions natives `exec()` et `compile()` dans un module `M` comportant une instruction *future* utilise, par défaut, la nouvelle syntaxe ou sémantique associée à l'instruction *future*. Ceci peut être contrôle par des arguments optionnels à `compile()` — voir la documentation de cette fonction pour les détails.

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 boucle *for*, dans une définition de *class*, de fonction, d'instruction *import* ou une annotation de variable.

CPython implementation detail : 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)*
```

L'instruction *nonlocal* fait que les noms listés font référence aux variables liées précédemment, dans la portée la plus petite entourant l'instruction, à l'exception des variables globales. C'est important car le comportement par défaut pour les liaisons consiste à chercher d'abord dans l'espace des noms locaux. Cette instruction permet à du code encapsulé de se lier à des variables en dehors de la portée locale du code mais sans avoir de portée globale (c'est-à-dire de niveau module).

Les noms (*identifier* dans la grammaire ci-dessus) listés dans l'instruction *nonlocal*, au contraire de ceux listés dans une instruction *global*, doivent faire référence à des liaisons pré-existantes dans les portées englobantes (en effet, la portée dans laquelle devrait être créée la liaison ne peut pas être déterminée *a priori*).

Les noms listés dans l'instruction *nonlocal* ne doivent entrer en collision avec des liaisons déjà établies dans la portée locale.

Voir aussi :

PEP 3104 – Accès à des noms en dehors de la portée locale Les spécifications pour l'instruction *nonlocal*.

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* spécifie 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
                    | 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" expression ":" suite
               ("elif" expression ":" suite)*
               ["else" ":" suite]
```

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

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" 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 tuple ou une liste) ou un autre objet itérable :

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

La liste des expressions (*expression_list* dans la grammaire ci-dessus) est évaluée une seule fois ; elle doit produire un objet itérable. Un itérateur est créé pour le résultat de cette liste d'expression. La suite est ensuite exécutée une fois pour chaque élément fourni par l'itérateur, dans l'ordre renvoyé par l'itérateur. Chaque élément est assigné, à tour de rôle, à la liste cible (*target_list* dans la grammaire ci-dessus) en utilisant les règles des assignations (voir *Les assignations*), et ensuite la suite est exécutée. Lorsque les éléments sont épuisés (ce qui est immédiat lorsque la séquence est vide ou si un itérateur lève une exception `StopIteration`), 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 : la fonction native `range()` renvoie un itérateur sur des entiers approprié pour émuler la boucle classique en Pascal sur des entiers `for i := a to b do`; par exemple, `list(range(3))` renvoie la liste `[0, 1, 2]`.

Note : Il y a une subtilité lorsque la séquence est modifiée par la boucle (cela ne peut se produire que pour les séquences mutables, c'est-à-dire les listes). Un compteur interne est utilisé pour savoir quel est l'élément suivant, et ce compteur est incrémenté à chaque itération. Lorsqu'il a atteint la longueur de la séquence, la boucle se termine. Cela signifie que si la suite supprime l'élément courant (ou un élément précédent) de la séquence, l'élément suivant est sauté (puisqu'il reçoit l'indice de l'élément courant qui a déjà été traité). De même, si la suite insère un élément avant l'élément courant, l'élément courant est traité une deuxième fois à la prochaine itération. Ceci peut conduire à de méchants bugs, que vous pouvez éviter en effectuant une copie temporaire d'une tranche ou de la séquence complète, par exemple :

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

8.4 L'instruction `try`

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

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

La ou les clauses *except* spécifient 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 de base de l'objet exception ou si c'est un tuple contenant un élément qui est compatible avec 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]

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 d'exception correspond, l'exception est assignée à la cible spécifiée 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 assignée 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 l'exécution de la suite d'une clause `except`, les détails de l'exception sont stockés dans le module `sys` et sont accessibles *via* `sys.exc_info()`. `sys.exc_info()` renvoie un triplet composé de la classe de l'exception, de l'instance d'exception et d'un objet trace (voir la section *Hiérarchie des types standards*) identifiant le point du programme où l'exception est survenue. Les valeurs de `sys.exc_info()` sont remises à leurs anciennes valeurs (celles d'avant l'appel) au retour d'une fonction qui a géré une exception.

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.

Si *finally* est présente, elle spécifie 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

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.

de la nouvelle exception. Si la clause `finally` exécute une instruction *`return`* ou *`break`*, 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. Une instruction *`continue`* est illégale dans une clause `finally` (la raison est que l'implémentation actuelle pose problème — il est possible que cette restriction soit levée dans le futur).

La valeur de retour d'une fonction est déterminée par la dernière instruction *`return`* exécutée. Puisque la clause *`finally`* s'exécute toujours, une instruction `return` exécutée dans le `finally` sera toujours la dernière clause exécutée :

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Vous trouvez des informations supplémentaires relatives aux exceptions dans la section *Exceptions* et, dans la section *L'instruction raise*, des informations relatives à l'utilisation de l'instruction *`raise`* pour produire des exceptions.

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_item (","  with_item)* ":" suite
with_item   ::=   expression ["as" target]
```

L'exécution de l'instruction *`with`* avec un seul "élément" (*item* dans la grammaire) se déroule comme suit :

1. L'expression de contexte (l'expression donnée dans le *`with_item`*) est évaluée pour obtenir un gestionnaire de contexte.
2. La méthode *`__exit__()`* du gestionnaire de contexte est chargée pour une utilisation ultérieure.
3. La méthode *`__enter__()`* du gestionnaire de contexte est invoquée.
4. Si une cible (*target* dans la grammaire ci-dessus) a été incluse dans l'instruction *`with`*, la valeur de retour de *`__enter__()`* lui est assignée.

Note : L'instruction *`with`* garantit que si la méthode *`__enter__()`* se termine sans erreur, alors la méthode

__exit__() est toujours appelée. Ainsi, si une erreur se produit pendant l'assignation à la liste cible, elle est traitée de la même façon qu'une erreur se produisant dans la suite. Voir l'étape 6 ci-dessous.

5. La suite est exécutée.
6. La méthode *__exit__()* du gestionnaire de contexte est invoquée. Si une exception a causé la sortie de la suite, son type, sa valeur et sa pile d'appels sont passés en arguments à *__exit__()*. Sinon, trois arguments `None` sont fournis.

   Si l'on est sorti de la suite en raison d'une exception et que la valeur de retour de la méthode *__exit__()* était fausse, l'exception est levée à nouveau. Si la valeur de retour était vraie, l'exception est supprimée et l'exécution continue avec l'instruction qui suit l'instruction *with*.

   Si l'on est sorti de la suite pour une raison autre qu'une exception, la valeur de retour de *__exit__()* est ignorée et l'exécution se poursuit à l'endroit normal pour le type de sortie prise.

Avec plus d'un élément, les gestionnaires de contexte sont traités comme si plusieurs instructions *with* étaient imbriquées :

```
with A() as a, B() as b:
    suite
```

est équivalente à :

```
with A() as a:
    with B() as b:
        suite
```

Modifié dans la version 3.1 : Prise en charge de multiples expressions de contexte.

Voir aussi :

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

8.6 Définition de fonctions

Une définition de fonction définit un objet fonction allogène (voir la section *Hiérarchie des types standards*) :

```
funcdef                 ::=  [decorators] "def" funcname "(" [parameter_list] ")"
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
dotted_name             ::=  identifier ("." identifier)*
parameter_list          ::=  defparameter ("," defparameter)* ["," [parameter_list_stara
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter ["
                             | "**" 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. [2]

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`.

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.

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 lorsqu'un paramètre par défaut est un objet mutable, tel qu'une liste ou un dictionnaire : 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 tuple recevant les paramètres positionnels en surplus, la valeur par défaut étant le tuple 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. 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.

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.

2. Une chaîne littérale apparaissant comme première instruction dans le corps de la fonction est transformée en attribut `__doc__` de la fonction et donc en *docstring* de la fonction.

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 Capacité d'indiquer des types pour les déclarations de variables, y compris les variables de classes et les variables d'instances

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.

8.7 Définition de classes

Une définition de classe définit un objet classe (voir la section *Hiérarchie des types standards*) :

```
classdef      ::=  [decorators] "class" classname [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 de base (voir *Méta-classes* 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 de base `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 [3]. Un objet classe est alors créé en utilisant la liste d'héritage pour les classes de base 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

3. 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.

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.

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**.

8.8 Coroutines

Nouveau dans la version 3.5.

8.8.1 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*). Dans le corps d'une coroutine, tout identificateur `await` ou `async` devient un mots-clé réservé ; 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()
```

8.8.2 L'instruction `async for`

```
async_for_stmt   ::=   "async" for_stmt
```

Un *itérable asynchrone* est capable d'appeler du code asynchrone dans l'implémentation de sa méthode *iter* ; un *itérateur asynchrone* peut appeler du code asynchrone dans sa méthode *next*.

L'instruction `async for` permet d'itérer facilement sur des itérateurs asynchrones.

Le code suivant :

```
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

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:
        BLOCK
else:
    BLOCK2
```

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.

8.8.3 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 EXPR as VAR:
    BLOCK
```

Est sémantiquement équivalent à :

```
mgr = (EXPR)
aexit = type(mgr).__aexit__
aenter = type(mgr).__aenter__(mgr)

VAR = await aenter
try:
    BLOCK
except:
    if not await aexit(mgr, *sys.exc_info()):
        raise
else:
    await aexit(mgr, 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.

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 de Python, exhaustive, telle qu'elle est lue par le générateur d'analyseur, et utilisée pour analyser des fichiers sources en Python :

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: 'async' funcdef
funcdef: 'def' NAME parameters ['->' test] ':' suite

parameters: '(' [typedargslist] ')'
typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])* [',' [
        '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']]]
      | '**' tfpdef [',']]]
  | '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']]]
  | '**' tfpdef [','])
tfpdef: NAME [':' test]
varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
```

(suite sur la page suivante)

(suite de la page précédente)

```
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']
)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal and annotated assignments, additional restrictions enforced by the␣
↪interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated | async_stmt
async_stmt: 'async' (funcdef | with_stmt | for_stmt)
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

test: or_test ['if' or_test 'else' test] | lambdef
```

(suite sur la page suivante)

(suite de la page précédente)

```
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401 (which really works :-)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: ['await'] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [testlist_comp] ']' |
       '{' [dictorsetmaker] '}' |
       NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (test|star_expr) ( comp_for | (',' (test|star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                   (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                  ((test | star_expr)
                   (comp_for | (',' (test | star_expr))* [',']) )

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)*  [',']

# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
# "test '=' test" is really "keyword '=' test", but we have no such token.
# These need to be in a single rule to avoid grammar that is ambiguous
# to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
# we explicitly match '*' here, too, to give it proper precedence.
# Illegal combinations and orderings are blocked in ast.c:
# multiple (test comp_for) arguments are blocked; keyword unpackings
# that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
            test '=' test |
            '**' test |
            '*' test )

comp_iter: comp_for | comp_if
sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_for: ['async'] sync_comp_for
```

(suite sur la page suivante)

(suite de la page précédente)

```
comp_if: 'if' test_nocond [comp_iter]

# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
```

ANNEXE A

Glossaire

>>> L'invite de commande utilisée par défaut dans l'interpréteur interactif. On la voit souvent dans des exemples de code qui peuvent être exécutés interactivement dans l'interpréteur.

. . . 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.

2to3 Outil qui essaie de convertir du code pour Python 2.x en code pour Python 3.x en gérant la plupart des incompatibilités qui peuvent être détectées en analysant la source et parcourant son arbre syntaxique.

2to3 est disponible dans la bibliothèque standard sous le nom de `lib2to3`; un point d'entrée indépendant est fourni via `Tools/scripts/2to3`. Cf. 2to3-reference.

classe de base abstraite Les classes de base abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes ou 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 utilisé 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 *variable annotation*, *function annotation*, **PEP 484** et **PEP 526**, qui décrivent cette fonctionnalité.

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 *parameter* 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 `with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été Introduit par la **PEP 492**.

générateur asynchrone Fonction qui renvoie un *asynchronous generator iterator*. 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 une fonction *asynchronous generator*.

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'endroit 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 la **PEP 492** et la **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 désignée par son nom via une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, il sera référencé par *o.a*.

awaitable Objet pouvant être utilisé dans une expression `await`. 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 Un *file object* capable de lire et d'écrire des *bytes-like objects*. Des fichiers binaires sont, par exemple, les fichiers ouverts en mode binaire (`'rb'`, `'wb'`, ou `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, les instances de `io.BytesIO` ou de `gzip.GzipFile`.

Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets `str`.

objet octet-compatible Un objet gérant les bufferobjects et pouvant exporter un tampon (*buffer* en anglais) C-*contiguous*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets `memoryview`. Les objets bytes-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 (*"read-only bytes-like objects"*), par exemples `bytes` ou `memoryview` d'un objet `byte`.

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.

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).

coercition Conversion implicite d'une instance d'un type vers un autre lors d'une opération dont les deux opérandes doivent être de même type. Par exemple `int(3.15)` convertit explicitement le nombre à virgule flottante en nombre entier `3`. Mais dans l'opération `3 + 4.5`, les deux opérandes sont d'un type différent (Un entier et un nombre à virgule flottante), alors qu'elles doivent avoir le même type pour être additionnées (sinon une exception `TypeError` serait levée). Sans coercition, toutes les opérandes, même de types compatibles, devraient être converties (on parle aussi de *cast*) explicitement par le développeur, par exemple : `float(3) + 4.5` au lieu du simple `3 + 4.5`.

nombre complexe Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de `-1`, souvent écrite `i` en mathématiques ou `j` par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe `j`, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

gestionnaire de contexte Objet contrôlant l'environnement à l'intérieur d'un bloc *`with`* en définissant les méthodes *`__enter__()`* et *`__exit__()`*. Consultez la **PEP 343**.

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 généralisées 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(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation *définitions de fonctions* et *définitions de classes* pour en savoir plus sur les décorateurs.

descripteur N'importe quel objet définissant les méthodes *__get__()*, *__set__()*, ou *__delete__()*. Lorsque l'attribut d'une classe est un descripteur, son comportement spécial est déclenché lors de la recherche des attributs. Normalement, lorsque vous écrivez *a.b* pour obtenir, affecter ou effacer un attribut, Python recherche l'objet nommé *b* dans le dictionnaire de la classe de *a*. Mais si *b* est un descripteur, c'est la méthode de ce descripteur qui est alors appelée. Comprendre les descripteurs est requis pour avoir une compréhension approfondie de Python, ils sont la base de nombre de ses caractéristiques notamment les fonctions, méthodes, propriétés, méthodes de classes, méthodes statiques et les références aux classes parentes.

Pour plus d'informations sur les méthodes des descripteurs, consultez *Implémentation de descripteurs*.

dictionnaire Structure de donnée associant des clés à des valeurs. Les clés peuvent être n'importe quel objet possédant les méthodes *__hash__()* et *__eq__()*. En Perl, les dictionnaires sont appelés "*hash*".

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.

***docstring* (chaîne de documentation)** Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elles est reconnue par le compilateur et placée dans l'attribut `__doc__` de la classe, de la fonction ou du module. Comme cette chaîne est disponible par introspection, c'est l'endroit idéal pour documenter l'objet.

duck-typing Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (*Si ça a un bec et que ça cancane, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes de base abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

EAFP Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés *try* et *except*. Cette technique de programmation contraste avec le style *LBYL* utilisé couramment dans les langages tels que C.

expression 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 Objet exposant une ressource via une API orientée fichier (avec les méthodes `read()` ou `write()`). En fonction de la manière dont il a été créé, un objet fichier peut interfacer l'accès à un fichier sur le disque ou à

un autre type de stockage ou de communication (typiquement l'entrée standard, la sortie standard, un tampon en mémoire, une socket réseau, ...). Les objets fichiers sont aussi appelés *file-like-objects* ou *streams*.

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*.

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 *variable annotation* et :pep :' 484', qui décrivent cette fonctionnalité.

__future__ Pseudo-module que les développeurs peuvent utiliser pour activer de nouvelles fonctionnalités du langage qui ne sont pas compatibles avec l'interpréteur utilisé.

En important le module `__future__` et en affichant ses variables, vous pouvez voir à quel moment une nouvelle fonctionnalité a été rajoutée dans le langage et quand elle devient le comportement par défaut :

```
>>> 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 générateur 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'ambigüité.

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**.

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 Un objet est *hachable* s'il a une empreinte (*hash*) qui ne change jamais (il doit donc implémenter une méthode *__hash__()*) et s'il peut être comparé à d'autres objets (avec la méthode *__eq__()*). Les objets hachables dont la comparaison par `__eq__` est vraie doivent avoir la même empreinte.

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 muables (comme les listes ou les dictionnaires) ne le sont pas ; les conteneurs immuables (comme les n-uplets ou les ensembles gelés) 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 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.

importer Processus rendant le code Python d'un module disponible dans un autre.

importateur Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

interactif Python a un interpréteur interactif, ce qui signifie que vous pouvez écrire des expressions et des instructions à l'invite de l'interpréteur. L'interpréteur Python va les exécuter immédiatement et vous en présenter le résultat. Démarrez juste `python` (probablement depuis le menu principal de votre ordinateur). C'est un moyen puissant pour tester de nouvelles idées ou étudier de nouveaux modules (souvenez-vous de `help(x)`).

interprété Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

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 quelques exception puisque les ressources auxquelles il fait appel pourraient 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 Objet capable de renvoyer ses éléments un à un. Par exemple, tous les types séquence (comme `list`, `str`, et `tuple`), quelques autres types comme `dict`, *objets fichiers* ou tout objet d'une classe ayant une méthode `__iter__()` ou `__getitem__()` qui implémente la sémantique d'une *Sequence*.

Les itérables peuvent être utilisés dans des boucles `for` et à beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()`, ...). Lorsqu'un itérable est passé comme argument à la fonction native `iter()`, celle-ci fournit en retour un itérateur sur cet itérable. Cet itérateur n'est valable que pour une seule passe sur le jeu de valeurs. Lors de l'utilisation d'itérables, il n'est habituellement pas nécessaire d'appeler `iter()` ou de s'occuper soi-même des objets itérateurs. L'instruction `for` le fait automatiquement pour vous, créant une variable temporaire anonyme pour garder l'itérateur durant la boucle. Voir aussi *itérateur*, *séquence* et *générateur*.

itérateur Objet représentant un flux de donnée. Des appels successifs à la méthode `__next__()` de l'itérateur (ou le passer à la fonction native `next()`) donne successivement les objets du flux. Lorsque plus aucune donnée n'est disponible, une exception `StopIteration` est levée. À ce point, l'itérateur est épuisé et tous les appels suivants à sa méthode `__next__()` lèveront encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `__iter__()` qui renvoie l'objet itérateur lui même, de façon à ce que chaque itérateur soit aussi itérable et puisse être utilisé dans la plupart des endroits où d'autres itérables sont attendus. Une exception notable est un code qui tente plusieurs itérations complètes. Un objet conteneur, (tel que `list`) produit un nouvel itérateur neuf à chaque fois qu'il est passé à la fonction `iter()` ou s'il est utilisé dans une boucle `for`. Faire ceci sur un itérateur donnerait simplement le même objet itérateur épuisé utilisé dans son itération précédente, le faisant ressembler à un conteneur vide.

Vous trouverez davantage d'informations dans typeiter.

fonction clé Une fonction clé est un objet appelable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction `locale.strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions 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])`. Vous noterez que le module `operator` propose des constructeurs de fonctions clefs : `attrgetter()`, `itemgetter()` et `methodcaller()`. Voir Comment Trier 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.

list Un type natif de *sequence* dans Python. En dépit de son nom, une `list` ressemble plus à un tableau (*array* dans la plupart des langages) qu'à une liste chaînée puisque les accès se font en 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 de base abstraite*.

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 de base abstraites `collections.abc.Mapping` ou `collections.abc.MutableMapping`. 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'aura jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûr les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : *Méta-classes*.

méthode Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

ordre de résolution des méthodes L'ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais) est, lors de la recherche d'un attribut dans les classes parentes, la façon dont l'interpréteur Python classe ces classes parentes. Voir The Python 2.3 Method Resolution Order pour plus de détails sur l'algorithme utilisé par l'interpréteur Python depuis la version 2.3.

module Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de 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*.

muable Un objet muable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

n-uplet nommé (*named-tuple* en anglais) Classe qui, comme un *n-uplet* (*tuple* en anglais), a ses éléments accessibles par leur indice. Et en plus, les éléments sont accessibles par leur nom. Par exemple, `time.localtime()` donne un objet ressemblant à un *n-uplet*, dont *year* est accessible par son indice : `t[0]` ou par son nom : `t.tm_year`).

Un *n-uplet nommé* peut être un type natif tel que `time.struct_time` ou il peut être construit comme une simple classe. Un *n-uplet nommé* complet peut aussi être créé via la fonction `collections.namedtuple()`. Cette dernière approche fournit automatiquement des fonctionnalités supplémentaires, tel qu'une représentation lisible comme `Employee(name='jones', title='programmer')`.

espace de 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 Ancien nom pour l'implémentation actuelle des classes, pour tous les objets. Dans les anciennes versions de Python, seules les nouvelles classes pouvaient utiliser les nouvelles fonctionnalités telles que *`__slots__`*, les descripteurs, les propriétés, *`__getattribute__()`*, les méthodes de classe et les méthodes statiques.

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* : l'argument ne peut être donné que par sa position. Python n'a pas de syntaxe pour déclarer de tels paramètres, cependant des fonctions natives, comme `abs()`, en utilisent.

— *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* Appelable dans la liste `sys.path_hook` qui donne un *chercheur d'entrée dans path* s'il sait où trouver des modules pour une *entrée dans path* donnée.

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 Python). Un 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 PEPs sont censés 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 **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 changement 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 changement cassant la rétrocompatibilité sont considérées comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problème 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 Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Le module `sys` définit une fonction `getrefcount()` que les développeurs peuvent utiliser pour obtenir le nombre de références à un objet donné.

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 *itérable* qui offre un accès efficace à ses éléments par un indice sous forme de nombre entier via la méthode spéciale *`__getitem__()`* et qui définit une méthode *`__len__()`* donnant sa taille. Voici quelques séquences natives : `list`, `str`, `tuple`, et `bytes`. Notez que `dict` possède aussi une méthode *`__getitem__()`* et une méthode *`__len__()`*, mais il est considéré comme un *mapping* plutôt qu'une séquence, car ses accès se font par une clé arbitraire *immuable* plutôt qu'un nombre entier.

La classe abstraite de base `collections.abc.Sequence` définit une interface plus riche qui va au-delà des simples *`__getitem__()`* et *`__len__()`*, en ajoutant `count()`, `index()`, *`__contains__()`* et *`__reversed__()`*. Les types qui implémentent cette interface étendue peuvent s'enregistrer explicitement en utilisant `register()`.

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`*.

struct sequence Un n-uplet (*tuple* en anglais) dont les éléments sont nommés. Les *struct sequences* exposent une interface similaire au *n-uplet nommé* car on peut accéder à leurs éléments par un nom d'attribut ou par un indice. Cependant, elles n'ont aucune des méthodes du *n-uplet nommé* : ni `collections.somenamedtuple._make()` ou `_asdict()`. Par exemple `sys.float_info` ou les valeurs données par `os.stat()` sont des *struct sequence*.

encodage de texte Codec (codeur-décodeur) qui convertit des chaînes de caractères Unicode en octets (classe *bytes*).

fichier texte *file object* 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'*text encoding* 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 *binary file* pour un objet fichier capable de lire et d'écrire *bytes-like objects*.

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 :

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Voir `typing` et **PEP 484**, qui décrivent cette fonctionnalité.

indication de type Le *annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Les indications de type sont facultatifs et ne sont pas indispensables à l'interpréteur Python, mais ils sont utiles aux outils d'analyse de type statique et aident les IDE à compléter et à réusiner (*code refactoring* en anglais) le code.

Les indicateurs de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultés en utilisant `typing.get_type_hints()`.

Voir `typing` et **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 la variable est expliquée dans la section *Les assignations annotées*.

Reportez-vous à *function annotation*, à la :pep :' 484' et à la **PEP 526** qui décrivent cette fonctionnalité.

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 *bytecode* produit par le compilateur de *bytecode*.

Le zen de Python Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant "`import this`" dans une invite Python interactive.

ANNEXE B

À propos de ces documents

Ces documents sont générés à partir de sources en reStructuredText par Sphinx, un analyseur de documents spécialement conçu pour la documentation Python.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page reporting-bugs qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

— Fred L. Drake, Jr., créateur des outils originaux de la documentation Python et rédacteur de la plupart de son contenu ;
— le projet Docutils pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
— Fredrik Lundh pour son projet Alternative Python Reference, dont Sphinx a pris beaucoup de bonnes idées.

B.1 Contributeurs de la documentation Python

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez Misc/ACKS dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse — Merci !

ANNEXE C

Histoire et licence

C.1 Histoire du logiciel

Python a été créé au début des années 1990 par Guido van Rossum, au Stichting Mathematisch Centrum (CWI, voir https://www.cwi.nl/) au Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclut de nombreuses contributions de la part d'autres personnes.

En 1995, Guido continua son travail sur Python au Corporation for National Research Initiatives (CNRI, voir https://www.cnri.reston.va.us/) de Reston, en Viriginie, d'où il diffusa plusieurs versions du logiciel.

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 http://www.zope.com/). En 2001, la Python Software Foundation (PSF, voir http://www.python.org/psf/) voit le jour. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de Python. Zope Corporation en est un sponsor.

Toutes les versions de Python sont Open Source (voir https://www.opensource.org/ pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et 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 changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non-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

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.4

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.7.4 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 3.7.4 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2019 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.7.4 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 3.7.4 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
   3.7.4.

4. PSF is making Python 3.7.4 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 3.7.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.4
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.4, 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 3.7.4, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

LICENCE D'UTILISATION LIBRE BEOPEN PYTHON VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2

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

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

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

C.3 Licences et remerciements pour les logiciels 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

Le module `_random` inclut du code construit à partir d'un téléchargement depuis http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. Voici mot pour mot les commentaires du code original :

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).
```

(suite sur la page suivante)

(suite de la page précédente)

```
Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Interfaces de connexion (*sockets*)

Le module `socket` utilise les fonctions `getaddrinfo()` et `getnameinfo()` codées dans des fichiers source séparés et provenant du projet WIDE : http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.
```

(suite sur la page suivante)

(suite de la page précédente)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Interfaces de connexion asynchrones

Les modules `asynchat` et `asyncore` contiennent la note suivante :

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

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

C.3.4 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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

Le module `test_epoll` contient la note suivante :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversion de *doubles* C vers et depuis les chaînes, est tiré d'un fichier du même nom par David M. Gay, actuellement disponible sur http://www.netlib.org/fp/. Le fichier original, tel que récupéré le 16 mars 2009, contient la licence suivante :

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et Mac OS X peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
```

(suite sur la page suivante)

(suite de la page précédente)

```
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
```

(suite sur la page suivante)

(suite de la page précédente)

```
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

Le module `pyexpat` est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

(suite sur la page suivante)

(suite de la page précédente)

```
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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
  * 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
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

Le module `_decimal` est construit en incluant une copie de la bibliothèque *libmpdec*, sauf si elle est compilée avec `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 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.
```

ANNEXE D

Copyright

Python et cette documentation sont :

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

Index

Non-alphabetical

A

B

C

D

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z